Optimizing Tradeoffs of Non-Functional Properties in Software

---

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Jonathan Dorn

August 2017

Abstract

Software systems have become integral to the daily life of millions of people. These systems provide much of our entertainment (e.g., video games [1], feature-length movies [2], and YouTube [3]) and our transportation (e.g., planes [4], trains [5] and automobiles [6]). They ensure that the electricity to power homes and businesses is delivered [7] and are significant consumers of that electricity themselves [8]. With so many people consuming software, the best balance between runtime, energy or battery use, and accuracy is different for some users than for others. With so many applications playing so many different roles and so many developers producing and modifying them, the tradeoff between maintainability and other properties must be managed as well.

Existing methodologies for managing these "non-functional" properties require significant additional effort. Some techniques impose restrictions on how software may be designed [9] or require time-consuming manual reviews [10]. These techniques are frequently specific to a single application domain, programming language, or architecture, and are primarily applicable during initial software design and development. Further, modifying one property, such as runtime, often changes another property as well, such as maintainability.

In this dissertation, we present a framework, exemplified by three case studies, for automatically manipulating interconnected program properties to find the optimal tradeoffs. We exploit evolutionary search to explore the complex interactions of diverse properties and present the results to users. We demonstrate the applicability and effectiveness of this approach in three application domains, involving different combinations of dynamic properties (how the program behaves as it runs) and static properties (what the source code itself is like). In doing so, we describe the ways in which those domains impact the choices of how to represent programs, how to measure their properties effectively, and how to search for the best among many candidate program implementations. We show that effective choices enable the framework to take unmodified human-written programs and automatically produce new implementations with better properties—and better tradeoffs between properties—than before.

Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

Jonathan Dorn

This dissertation has been read and approved by the Examining Committee:

Westley R. Weimer, Advisor

Baishakhi Ray, Committee Chair

Jason Lawrence

Stephanie Forrest

Chad Dodson

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, Dean, School of Engineering and Applied Science

August 2017

Acknowledgments

There are many, many people who have helped and supported me through this long process. This brief note can but hint at the innumerable conversations and interactions that led up to today.

First, of course, Wes Weimer, my advisor, who tolerates my enthusiastic use of commas in all their roles. Like the best of coaches, he always challenged me to give 110%, while understanding and encouraging me when it felt like 90% was all I had.

And to Wes and all of my collaborators in Virginia and abroad, who shared their time, ideas, and enthusiasm; I can only hope that they got half as much out of our discussions as I did. To all of my officemates over the years, who so often let me drag them along to a coffee shop so I could talk through my current set of challenges.

My profound thanks to my family for a lifetime of love and support, and, most recently, for willingly listening as I ramble incomprehensibly about one research project or another. And to the friends who have helped make grad school not just educational, but also fun.

Finally, thanks to everyone who went above and beyond the call of friendship to read some portion of this dissertation and give me feedback on flow and clarity. This document is all the better for their questions and comments.

Contents

List of Tables

List of Figures

List of Listings

List of Terms

Note: terms are linked to their associated glossary entry at their first introduction or definition, and again at their first use in each subsequent chapter. Within this glossary, terms are linked more comprehensively.

abstract syntax tree (AST) — a representation of a program as a tree in which interior nodes represent compound statements or expressions (such as for-loops or function calls) and their children represent their components or arguments (such as a then-branch or variable). 22

aliasing — an artifact caused by reconstructing a signal from too few samples. In computer graphics, aliasing often appears as jagged lines in place of smooth ones, small details that seem to appear or disappear, or regular features that seem to cluster instead of being evenly distributed. 6, 18, 48, 81

antialiasing — a collective term for various processes designed to reduce the incidence or visibility of aliasing. 7, 19

approximate computing — a broad set of techniques for trading output quality for improvements in some non-functional properties, usually run time or energy use. 46

attribute — see feature. 70

band-limit — modify a function so as to reduce its maximum frequency component. Often band-limiting reduces the coefficients of higher frequency components to small positive values rather than ensuring that they are zero. 21

bandwidth — the difference between the lowest frequency component of a signal (often 0 Hz) and the highest. 20

code coverage — a measure of the portion of a program executed by a test suite. Different coverage metrics include the fraction of lines, statements, branch conditions, and paths through the program that are covered. See also coverage. 9

convolution — a mathematical operator that combines two functions to produce a third function, defined as

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)\, g(t - \tau)\, d\tau.$$

In effect, the new function computes the average of f in the vicinity of t, weighted by g. 21, 48

coverage — in the context of testing, a measure of the fraction of requirements that a test suite exercises. The coverage of requirements is frequently approximated with code coverage. 71, 82

crossover — in a genetic algorithm (GA), the mechanism by which two existing individuals exchange portions of their genomes to form new individuals. 15, 57

current — in an electrical context, the flow of electric charge per unit of time, measured in amperes. 48

data center — a collection of computer systems and associated communication and storage systems, housed in one location. Data centers may serve many different organizations or a small number and may run any number of applications. 7, 45, 81

dynamic property — a property describing the behavior of a program as it executes, as opposed to a static property. 4, 82

energy — the capacity to do work, that is, exert a force over a distance [11]. Many different forms of energy are recognized, a small sampling of which includes chemical, electrical, gravitational, kinetic, mechanical, and thermal. 47, 81

evolutionary algorithm (EA) — a population-based search-based optimization (SBO) algorithm inspired by biological evolution. These algorithms mutate existing individuals to introduce new individuals into the population. They assume that the fitness of particular individuals can be measured and compared for order, but do not require the magnitude of the fitness to be otherwise meaningful. 3, 13, 35

fault — in the context of testing, the cause of an error. 69

feature — in the context of machine learning (ML), a measurable attribute of instances about which a prediction (e.g., classification or cluster assignment) is made. Decisions or computations are made in terms of the feature values for each instance. 70

fitness — an objective value in the context of an evolutionary algorithm (EA). 4, 14, 81

fitness function — an objective function in the context of an evolutionary algorithm (EA). 4, 14, 19, 46, 72

frame — a single image, usually a member of a sequence of such images that together constitute a movie or animation. 6, 19

functional correctness — a software system is said to be functionally correct if, for every valid input, it produces the correct output. 1, 81

generalization — in the context of machine learning (ML), the degree to which a model computes accurate values for examples outside of the training set. 70

generational genetic algorithm — a genetic algorithm (GA) in which parents are selected from the current population and their mutant offspring are added to a separate population. Once the new population is full of newly created mutants, it replaces the current population. Each cycle of replacement is called a generation. 15, 57

genetic algorithm (GA) — an evolutionary algorithm (EA) in which each individual is associated with a genome that describes it and new individuals are formed by crossing over of two existing members of the population. 11, 36, 46, 82

genome — in a genetic algorithm (GA), an encoding of the information required to recreate a particular individual. The particular encoding chosen impacts the effect of crossover, and can thus greatly affect the success of the algorithm. 14, 55

heuristic — an algorithm that sacrifices correctness in some cases to compute an answer more quickly (or at all). 3, 69

hill climbing — a metaheuristic that maintains a current solution and considers a sampling of solutions that are nearby in the search space before selecting the one with the greatest objective value. A metaheuristic analog to gradient ascent. 14

indicative workload — a set of inputs to a program that cause it to behave in a way that is typical of its use after being deployed. 4, 12, 48

individual — a candidate solution in the context of evolutionary algorithm (EA). 14

instance — in the context of machine learning (ML), a set of feature values associated with a single example, item, or situation. The term may also be used to refer to the example itself. 70

International System of Units (SI) — the successor to the metric system; a system of measurement based on seven base units: the meter (m), kilogram (kg), second (s), ampere (A), kelvin (K), mole (mol), and candela (cd). 47

just-in-time compiler (JIT) — a compiler that transforms program code at program run time, shortly before it must be executed. 11

kernel — in the context of signal and image processing, a function to be combined via convolution with the signal or image function to filter out noise or smooth it. 21

label — in the context of machine learning, the desired value for an instance to be used in training a model. 70

machine learning (ML) — the study of algorithms that learn from example or improve with experience. Machine learning algorithms are classified as *on-* or *off-line* depending on whether learning or training occurs continuously or as a discrete initial phase. Algorithms are orthogonally classified as *supervised*—in which the desired output for training examples are known in advanced—or *unsupervised*—in which the desired outputs are not known. 70, 82

metaheuristic — one of a class of search-based optimization (SBO) algorithms that attempt to guide the search to avoid local optima [12]. These algorithms make few assumptions about the representation of solutions or the range of the objective function [13]. 13

multi-objective optimization — an optimization problem involving more than one objective. Solutions often involve a Pareto frontier of solutions embodying different tradeoffs between objectives rather than a single best solution. 46, 75, 82

multiplicatively separable — a function $f(x_1, x_2)$ is multiplicatively separable if there exist functions g_1 and g_2 such that $f(x_1, x_2) = g_1(x_1)g_2(x_2)$. A function f of n variables is *completely* multiplicatively separable if there exist functions $g_1, \ldots, g_n$ such that $f(x_1, \ldots, x_n) = g_1(x_1) \cdots g_n(x_n)$. A function f of n variables is *partially* multiplicatively separable if $\{x_1, \ldots, x_n\}$ can be partitioned into two non-empty sets A and B with functions g_A and g_B such that $f(x_1, \ldots, x_n) = g_A(y_1, \ldots, y_{|A|})g_B(z_1, \ldots, z_{|B|})$ where $A = \{y_1, \ldots, y_{|A|}\}$ and $B = \{z_1, \ldots, z_{|B|}\}$. 28, 88

mutant — a slightly modified version of a program. In the context of evolutionary algorithms (EAs), an individual created from a parent by mutating the latter. In the context of mutation testing, program created by deliberately inserting a fault into the program being tested. 57, 72

mutate — in the context of evolutionary algorithms (EAs), to apply a transformation to an individual in the population to produce a new individual. 3, 14

mutation testing — a technique for evaluating test suites based on the number of mutants of the program being tested that are detected by at least one test in the suite. 72

non-functional property — a property of a software system that is conceptually separate from the function it computes. The separation between functional and non-functional properties is often blurred; image quality may

be considered a non-functional property while producing an image with at least some minimial quality may be considered a functional property. 1, 10, 81

Nyquist interval — the maximum sampling interval to allow unambiguous reconstruction of a continuous signal from the samples. The Nyquist interval is equal to half the period of the highest frequency present in the signal being sampled. 20

objective — the quantity being optimized by a search-based optimization (SBO) algorithm. A value or list of values summarizing the desirability of a particular candidate solution. See also objective function. 13, 57

objective function — a function that computes objective values for use by a search-based optimization (SBO) algorithm. Whether the most desirable candidate solution is assigned the largest or smallest objective value is implicit in the definition of this function. 13

oracle — also called an *oracle comparator*, generates correct output for a particular test input or workload and validates actual program output against the correct output. The test *passes* if the program output is sufficiently correct and *fails* otherwise. 16

parent — in the context of evolutionary algorithms (EAs), an individual from which a second individual was created by mutation or crossover. 15, 57

Pareto dominance — a relation between two sets of corresponding values. One set of values is said to Pareto dominate another if the first contains no values that are worse, and at least one value that is better, than the corresponding values in the second. 5, 75, 81

Pareto frontier — given a set of points, the Pareto frontier denotes the subset such that (a) no point in the frontier Pareto dominates any other point in the frontier and (b) every point not in the frontier is dominated by a point in the frontier. 5, 46, 81

pixel — picture element; a grid of pixels is used to display an image. 6, 19

population — in the context of search-based optimization (SBO) techniques, the set of solutions currently being considered. 3, 14

power — the rate at which energy is expended or consumed [11]. 47

prefiltering — an approach to computing an integral in which the formula is partially evaluated using known or preset values of variables, resulting in a simpler formula that may be evaluated fully later. 20

procedural shader — a user-defined shader program. 18, 81

profile — most commonly, a record of the amount of time spent executing each line or statement of a program. More generally, a profile may associate any metric of interest—such as execution count or energy use—with parts of a program. 12, 48, 72

profile-guided optimization (PGO) — a compiler-based optimization approach that targets the parts of a program that consume the most time, according to profiles collected while the program executes. See also superoptimization. 12, 56

readability — in this dissertation, the visual organization and presentation of a text that influences how easily it is understood. This is distinct from the complexity of the underlying algorithm or problem being solved. Some authors refer to this type of readability as extrinsic complexity or typographic or accidental readability. 71, 82

regression test — a test used to verify that a functional property of an earlier version of a piece of software is retained in a newer version. 4, 16

sample — measure the value of a continuous function at a discrete point. 19, 48

sampling interval — the interval (in time or distance, as appropriate) between samples. 20, 48

search space — in the context of search-based software engineering (SBSE), the collection of all possible candidate solutions. 3, 13, 35, 73

search-based optimization (SBO) — an optimization approach in which a candidate solution is selected from a search space of such solutions, according to a fitness function. These algorithms take as input an initial solution or solutions and consider additional solutions that are "near" some previously considered solution. Some examples of SBO algorithms include genetic algorithms, hill climbing, and simulated annealing. 3, 10, 81

search-based software engineering (SBSE) — the application of search-based optimization (SBO) techniques to software engineering problems. 2, 10, 54

shader — broadly, an algorithm for determining properties of primitives (such as vertices, surfaces, or pixels) in a graphics rendering pipeline. Typically, a shader is implemented using a lookup table with a preset interpolation algorithm or using a procedural shader. 18

static property — a property of the source code of a program, separate from its runtime behavior, as opposed to a dynamic property. 4, 82

superoptimization — an optimization approach that profiles several permutations of a short sequence of instructions, selecting the best such permutation. 11, 54

supersampling — an approach to approximating integrals, based on evaluating the integrand at several points and computing the piecewise combination of the samples. 20, 81

test — see test case. 16, 58, 69

test case — often called simply a test, the combination of an input (or workload) and an oracle to validate the program's output against the correct output. 16, 57, 69

test suite — a collection of test cases for a particular program. 71, 82

testing — the process of exercising a program to demonstrate errors, indicating the existence of faults. 16, 69, 71

texture mapping — the application of a stored or computed image to a rendered surface in 3D graphics systems. The technique is frequently used to vary the appearance of the surface, giving the appearance of texture. 19

tournament selection — a process in which the individual with the best fitness among a number of individual are chosen uniformly at random with replacement from a population is selected. 15

training — in machine learning (ML), the process by which the algorithm learns from example. 70

unit test — a (frequently automated) test of a single software component in isolation from the rest of the system. 8, 69, 82

variant — in this dissertation, a program generated by automatically transforming an original program. A variant may display different non-functional properties than the original. 2, 55

voltage — the electrical potential energy per unit of charge. 48

warehouse-scale computer (WSC) — a data center specifically designed to run very large distributed applications and internet services rather than a number of smaller services. The implication is that all of the individual systems in the warehouse act together as a single platform for the application. 45

workload — the task for which a program is executed. Workloads are often specified via particular command-line arguments when the program is started or via input files that the program reads when it starts running. 16, 45

List of Symbols

Dirac delta function (δ) the function defined by the constraints that $\delta(x) = 0$ if $x \neq 0$ and $\int_{-\infty}^{\infty} \delta(x)\, dx = 1$. 94

Fourier transform $\mathcal{F}$ an operator that generates a new function to compute the frequency content of an input function. For a function $f(x)$, the notation $\mathcal{F}_x\left[f(x)\right](\nu)$ indicates the Fourier transform of f as a function of the oscillation frequency ν and is defined as,

$$\mathcal{F}_x\left[f(x)\right](\nu) = \int_{-\infty}^{\infty} f(x)\, e^{-2\pi i \nu x}\, dx. \tag{1}$$

The inverse operator is defined as,

$$\mathcal{F}_\nu^{-1}\left[f(\nu)\right](x) = \int_{-\infty}^{\infty} f(\nu)\, e^{2\pi i \nu x}\, d\nu. \tag{2}$$

92

fract function ($fract$) a function returning the fractional part of its argument, defined as, $fract(x) = x - \lfloor x \rfloor$. 24, 92

Gauss error function (erf) a function defined as, $\operatorname{erf} x = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt$. 25, 84

Heaviside unit step (H) a function defined as,

$$H(x) = step(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise.} \end{cases} \tag{3}$$

23, 95

hypervolume indicator (I_H) the measure of the points dominated by a set of points (often a Pareto frontier). The hypervolume indicator is often used as a metric of the quality of a solution to a multi-objective optimization problem. 64

joule (J) the unit of work or energy in the International System of Units (SI). It is equal to the energy dissipated when one ampere (A) passes through a resistance of one volt (V) for one second. 47

truncate function ($trunc$) the function that returns the integer that is nearest to its argument in the direction of zero. That is, $trunc(x) = \lceil x \rceil$ when $x < 0$ and $trunc(x) = \lfloor x \rfloor$ when $x \geq 0$. 23

watt (W) the International System of Units (SI) unit of power, defined as energy consumed or work done per second. 47

Chapter 1

Introduction

SOFTWARE plays a vital and ever-expanding role in the modern world. Software systems trade on the major stock exchanges [7] and manage significant portions of our utility and shipping infrastructure [14]. Software running on aircraft is responsible for the safety of hundreds of millions of passengers per year in the U.S. alone [4, 15]. Virtual environments and 3D graphics contribute to billions of dollars brought in by 3D movies each year [2]. The development and deployment of all this software have significant impact, beyond the utility of the software itself. Software developers account for seven out of every 1000 workers in the U.S. [16, 17]. Data centers may account for over 1% of global energy consumption [8], an amount significantly affected by the efficiency of the applications deployed in those centers [18].

In addition to producing the correct outputs, which software engineers term functional correctness, these software systems posses many other non-functional properties [19] that are often also of critical importance. For example, while an aircraft autopilot must compute the correct control response to the current altitude, velocity, desired course, etc. (a functional property), it is equally important that the computation be completed quickly (a non-functional property). In part because of the variety of non-functional properties, researchers have proposed a number of methods to help ensure or optimize them. Many of these methods require significant additional effort beyond the challenge of writing functionally-correct software. For example, some researchers have designed frameworks for formal arguments demonstrating the safety of a system [20], while others advocate targeted code reviews for individual properties, such as efficiency or adherence to coding standards [10]. Others have proposed approaches that require evaluating (or approximating) challenging integrals by hand [21] or using a particular design to allow performance optimization [9]. These techniques are time-consuming to apply, are frequently specific to a single domain or software architecture, and are primarily applicable during the initial software design and development.

Compounding the cost of managing individual properties, there is frequently a tension between distinct non-functional properties; for example, all other things being equal, higher quality images require more time to produce. Developers must produce software that achieves a reasonable balance between these competing concerns. If different circumstances require different tradeoffs between properties, developers must either accept less desirable properties in some circumstances or develop multiple implementations that may be selected as necessary. For example, artists working in on 3D movies need to be able render scenes quickly to see the effects of the changes they make but can tolerate small errors in the images, while the high-quality images for the final movie may take hours [22]. Depending on the magnitude of the differences between these implementations, producing several implementations—such as separate applications to generate development images as opposed to the final movie—may significantly magnify the cost of development.

In this dissertation, we present a framework to provide assistance to programmers, allowing them to write a *single* initial implementation, *automatically* generate additional implementations with different non-functional properties, then present the best such implementations to the user. We view this as an optimization framework, since these additional implementations possess at least one non-functional property that has been improved relative to the original. This approach is applicable to a wide variety of domains and enables exploration of tradeoffs between different properties. We leverage the hypothesis that competent programmers produce software that is close to correct [23]. That is, in cases where the software does not yet have all the desired properties, developers tend to have written a program with many of them 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 that gap. Although many dynamic properties of programs cannot be determined statically [24], we observe that many interesting non-functional properties may be measured or approximated efficiently at run time. Thus, we apply search-based software engineering (SBSE) [25] techniques, which generate a large number of program variants and iteratively select the best, to optimize non-functional properties. We hypothesize that,

> **Thesis:** Search-based software engineering techniques applying local software transformations can automatically and effectively explore tradeoffs between a variety of measurable non-functional properties in existing software artifacts with indicative workloads across application domains.

We demonstrate the generality of our approach by applying it in the context of three application domains, covering a variety of non-functional properties. First, we use our approach to identify new tradeoffs between visual quality and run time in graphics programs (chapter 3). Second, we explore the tradeoff between energy use and output accuracy in data center applications (chapter 4). Finally, we apply SBSE to automatically generate test suites that maximize both readability and coverage (chapter 5).

In the remainder of this chapter, we introduce the structure of our optimization framework, the assumptions it makes, along with its inputs and outputs. Then we introduce the application areas in more detail and sketch out how each of them fits into the framework.

1.1 An Evolutionary Optimization Framework

In this dissertation, we use evolutionary algorithms (EAs) to improve non-functional properties of programs. EAs are a subset of search-based optimization (SBO) algorithms inspired by biological evolution [13]. We chose EAs because they make relatively few assumptions about the solutions (i.e., programs) being optimized. In particular, unlike Newton's method [26] or gradient descent [13], which assume it is possible to calculate the rate of change in the desirability of solutions, EAs require that new candidate solutions may be formed by transforming (or mutating) an existing solution and that the desirability of two solutions may be measured and compared for order. EAs organize all possible solutions as a network—the search space—such that any two solutions are connected by at least one path that may pass through some number of intermediate solutions along the way. Two solutions are adjacent in the search space, i.e., the path between them contains no intermediate solutions, if one may be reached by applying a single transformation to the other. This permits EAs to explore the search space by maintaining a (possibly singleton) set, called a population, of "current" solutions, inspecting the solutions adjacent to the population, and adding or removing solutions from the population [27]. Different algorithms use different heuristics governing which adjacent solutions to inspect and how to modify the population in an attempt to explore the search space efficiently. Our optimization framework is parameterized with respect to the specific search algorithm employed.

In the following chapters, every candidate solution will be a program[1] that can be generated by applying a sequence of transformations to the original program written by the software developer. The goal of the search algorithm is to find a sequence of transformations that will produce a program with the same or similar functional properties as the original, while simultaneously improving the target non-functional properties. To explore tradeoffs between properties, instead of a single optimized program, the algorithm identifies a set of programs that demonstrate different balances between the properties. The following subsections discuss the representations we consider (section 1.1.1), what we measure when a search algorithm inspects a new candidate (section 1.1.2), and how we define the set of programs exemplifying the best tradeoffs (section 1.1.3).

1.1.1 Representing Programs and Transformations

As described above, the choice of how to represent programs and transformations defines the search space [28]. This choice is subject to the competing concerns of expressiveness and efficiency. A more expressive choice of representation

[1] That is, a file or set of files that may be compiled or assembled to produce one or more executable programs.

implies that the search space is more likely to contain programs that have the desired functional and non-functional properties. However, a highly expressive representation may induce a search space in which the path to one of these desired programs contains many intermediate programs that must be inspected, resulting in a less efficient search. For example, one simple representation could treat every C program as a sequence of characters, with transformations that append or delete a single character relative to an existing program. This representation is completely expressive, since every program can be generated by a sequence of such appends and deletes. However, the path from one program to another, with high probability, includes a large number of character sequences that do not compile and thus do not maintain the desired functional and non-functional properties. The search algorithm must then take time to process each of the intermediate non-compiling sequences before finding any with the desired properties.

In our framework, transformations need not be semantics-preserving. That is, unlike the transformations applied by compilers [29] or many safety and security approaches [30, 31], we permit transformations that may alter the program functionality and subsequently check the functionality of the resulting program. Programs that deviate too far from the original's functional behavior may be treated as very undesirable or completely rejected from the search. We check the functional properties of each transformed program at the same time as we measure its non-functional properties, as described in the next subsection.

1.1.2 Measuring Program Properties

Before we can optimize program properties, we must be able to measure them. We formalize the mechanism for measuring the properties of a program as a function that takes a program as input and returns a list of values quantifying each property as output. EAs refer to this function as the fitness function and the value it returns as the fitness.

While some research (e.g., [32, 33, 34]) suggests that certain dynamic properties of programs may be approximated from static properties, in general it is necessary to run a program to determine how it behaves. Thus, a fitness function that measures a dynamic property must frequently incorporate an indicative workload, that is, a set of inputs that cause typical expected behavior. Since the fitness function will likely be evaluated for a large number of candidate programs, the time required to evaluate the property using these workloads is the primary factor determining the rate at which the search progresses. The indicative workload also serves a second important role in our framework. As mentioned above, if the transformations do not guarantee functional equivalence, the fitness function is also responsible for assuring the functional properties of the candidate program. In effect, the fitness function must include regression tests [35] to verify that the candidate's behavior is consistent with the original. This is typically accomplished by comparing the output produced by the program on the indicative workload to the output of the original. If the program fails any tests, the fitness function can assign it a pessimal fitness value to ensure that it cannot be returned as an optimized program.

Although some non-functional properties—such as run time—are straightforward to measure and quantify, others—such as image quality or readability—are not. A rich body of research has developed to describe how limitations in the fitness function's ability to capture the target properties affect the success of EAs; see Jin and Branke [36, § II and IV] for an overview. At a high level, there are three ways[2] in which a fitness function may capture an imprecise estimate of a property. First, the measurement may be subject to noise; this is frequently the case with measurements of physical quantities, such as time or power. We describe a noisy fitness function in chapter 4. Second, the function may compute an approximation of a complex, time-consuming, or even unmeasurable property, such as the aesthetic appeal of rendered images. Chapters 3 and 5 incorporate these sorts of approximating fitness functions. Third, the property may change over time, such as adherence to changing coding standards. Existing approaches to handling changing fitness values involve continuous optimization or, if changes can be recognized externally, restarting optimization after a change. We do not explicitly address fitness properties that change over time in this dissertation.

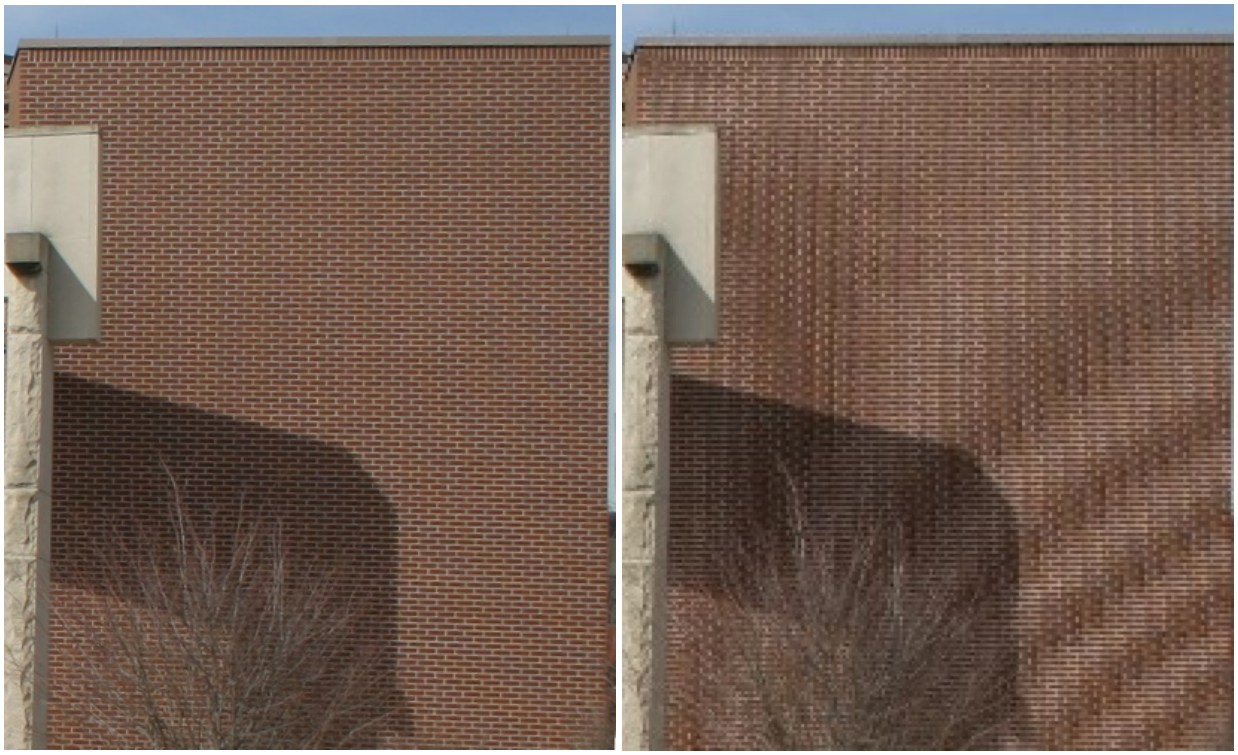
1.1.3 Optimizing Multiple Objectives

Where possible, our framework will optimize all measured properties simultaneously. For example, if it discovers a way to improve both run time and visual quality, the framework will exploit it. However, there are many cases in which improvements in one property come only at the expense of another. In these cases, which are at the heart of this dissertation, our framework is designed to generate an array of programs that exemplify different combinations of properties.

We formalize the desired result of our search through the concept of Pareto dominance [37]. One solution is said to Pareto dominate (or just "dominate") another if the first has no properties that are worse than the corresponding properties of the second *and* the first has at least one property that is better. To continue the earlier example, if we are considering the tradeoff between run time and visual quality and two programs have the same run time but one has better visual quality than the other, the first program dominates the second. If one has better run time while the other has better visual quality, neither one dominates the other. We define the set of programs displaying the optimum tradeoffs to be the programs that are not dominated by any other programs. This set of programs is called the Pareto frontier.

In the next three sections, we introduce the three application domains in which we will demonstrate our approach. Subsequent chapters will explore these domains and the particular representations, fitness functions, and search strategies they suggest.

[2] Jin and Branke [36] identify a fourth category, in which the candidate solution is itself subject to uncertainty. However, since the program source is fully specified, this category is not applicable to our framework.



(a) High-quality reference image.

(b) Visual aliasing caused by insufficient sampling rate.

Figure 1.1: Photograph of brick wall showing aliasing artifacts. The image on the left is a reference image. Aliasing appears in the image on the right as ripples in the wall surface, originating in the lower right of the image. Credit "Moire pattern of bricks" by Colin M.L. Burnett, via Wikimedia Commons, licensed under CC BY-SA 3.0.

1.2 Run Time and Visual Quality

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 [39], and aerospace engineering [40]. 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 [41]. 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 [42].

This performance must be achieved while maintaining a sufficient level of visual quality. For example, consider the two images in figure 1.1. In the image on the right, the wall appears to contain "ripples" that are caused by a mismatch between the pattern of the bricks and the pixels (picture elements) of the image. This mismatch, technically known as aliasing, affects computer generated images as well as photographs. Aliasing occurs when the pixels, which must each

display a single color, are spaced too widely relative to the color variation in the underlying image they are used to represent. The resulting undesired visual effects, or their absence, constitute an important component of visual quality. Antialiasing approaches attempt to reduce visible aliasing. For example, since the program source code controls the underlying image, changes to the program can alter the level of aliasing and therefore result in different image quality.

In this research thrust, we apply our optimization framework to the properties of visual quality (as caused by aliasing) and run time in graphics programs. As a special case of our thesis, we hypothesize that:

> **Hypothesis 1:** We can design local program transformations that, when applied to a particular class of graphics programs, produce programs that are non-dominated with respect to the run time and visual quality of existing antialiasing approaches.

In chapter 3, we evaluate this hypothesis by applying our technique to 12 graphics programs, including six used in previous work on addressing aliasing. We measure the visual quality and run time of the optimized programs identified by our technique and compare them to the same properties measured on the original programs and a baseline approach. Our results show that the programs identified by our technique are Pareto dominated by neither the original program nor the baseline approach, and thus represent a new and desirable tradeoff between these non-functional properties. Indeed, in some cases the programs we identify dominate both the original and the baseline approach.

1.3 Output Accuracy and Energy Use

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 [8]. The mechanical and electrical systems (such as lighting, cooling, air circulation, and uninterruptible power supplies) required to support the computation may quadruple [43] 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. This computational efficiency can hinge on relatively small local design and implementation decisions [44].

At the same time, many data center applications permit a "good enough" concept of correctness as they process massive amounts of data to produce acceptable user experiences [45]. The growing field of approximate computing [46] encompasses both hardware and software approaches that exploit human tolerance for imprecise results, whether due to a lack of a well-defined correct result or due to human perceptual limitations, to produce approximate answers while consuming less energy [47]. Existing approximate computing approaches require developers to annotate computations and data that may be handled approximately [48, 49, 50], target special-purpose systems [51], or operate on a small,

fixed set of program constructs [52, 53]. In contrast, we demonstrate that our framework permits optimization on un-annotated code targeting general-purpose systems and using arbitrary program constructs.

In this research thrust, we address the problem of optimizing energy usage in data center applications. In support of our main thesis, we hypothesize that:

> **Hypothesis 2:** We can apply generic, local program transformations to reduce the energy usage of data center applications.

> **Hypothesis 3:** By explicitly optimizing energy usage and output accuracy simultaneously, we can achieve greater energy reductions than optimizing energy usage alone while retaining a human-acceptable level of output quality.

In chapter 4, we evaluate these hypotheses on the PARSEC benchmark suite [54], which was designed to mimic data center application workloads. Our experiment consists of two separate optimization runs. The first run optimizes each program's energy requirements on indicative workloads. The second run optimizes both energy requirements and output accuracy simultaneously. As a baseline for comparison, we use loop perforation [52], which has previously been applied to several of the PARSEC benchmarks. Loop perforation also generates a Pareto frontier of programs exemplifying different tradeoffs between energy and accuracy, allowing us to compare the range of tradeoffs identified by both techniques. Our results show that our technique produces tradeoffs at least as good as, and better in many cases than, those identified by loop perforation.

1.4 Readability and Code Coverage

Programmers spend far more—as much as 10 times more [55]—time reading code than writing it [56, 57]. Reading code is a skill that must be acquired [58] because understanding programs is difficult [59, 60]. However, understanding programs "plays a role in nearly all software tasks" [61]. It is necessary to understand the program to write it, to review it, to test it, to debug it, and to extend it. 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 [62, 63] and industry [64] have produced coding standards documents designed, at least in part, to "improve the readability of code" [65].

We focus on the problem of readability as it applies to unit tests. Unit tests evaluate the functionality of the lowest-level components of a software system, with the intention that when a unit test the faulty component will be easier to identify and fix. Because of the sheer number of components in modern software systems and the resulting scope of functionality to test, automatic test generation is a rich field of research (e.g., [66, 67, 68, 69]). As a proxy for

the number of functional requirements tested, these approaches often attempt to generate a number of tests that together maximize code coverage—for example, the number of lines, branches, or true conditional expressions—evaluated while running the tests. However, when such a test fails, a developer must read the test to understand why and whether the fault lies in the application or in the test itself [70].

In this research thrust, we optimize the readability of automatically-generated high-coverage tests. We specialize our thesis in this application domain as follows:

> **Hypothesis 4:** We can apply local transformations and search-based algorithms to maximize the readability of automatically-generated high-coverage tests.

In chapter 5, we evaluate this hypothesis on 30 classes chosen from open-source projects. We use EvoSuite, an off-the-shelf test generation tool, to produce high-coverage tests for those classes. Using this suite as a baseline, we then use our search-based framework to identify variants of the baseline suite that optimize readability while maintaining the high coverage of the original suite. To evaluate the readability of our optimized suite, we conducted a human study in which we asked participants to identify which test they preferred in a head-to-head comparison and separately asked them to answer understanding questions about the generated tests. Our results show that the participants tended to prefer the readability-optimized tests and could answer questions about them more quickly at the same level of accuracy.

1.5 Summary

This dissertation presents a framework for optimization of multiple, potentially conflicting non-functional properties of programs. The framework accepts existing program code and automatically transforms it to produce similar programs with potentially different non-functional properties than the original, identifying the set of implementations representing the best tradeoffs between the measured properties. We apply our framework to three different application domains with different relevant properties.

Chapter 2

Background and Related Work

TRADITIONALLY, program optimization[1] has referred to changing a program so that it computes the same output in less time (e.g., [29, 71, 72]). Nonetheless, various efforts have considered optimizing specific other non-functional properties, such as energy use [73] or memory footprint [74]. This dissertation describes the optimization of a variety of non-functional properties in a search-based optimization (SBO) framework. In this chapter, we discuss background useful for placing that framework in context. We postpone discussion of the domain-specific background related to each set of non-functional properties until the relevant chapters.

We begin by describing the problem of program optimization from the perspective of traditional semantics-preserving compilers (section 2.1). We then show how this problem can be generalized as an example of an SBO problem and discuss the use of SBO in search-based software engineering (SBSE) (section 2.2). Finally, we discuss transformations that may alter the program semantics and mechanisms for managing the degree of change in section 2.3.

2.1 Compiler Optimizations

A compiler, such as GCC,[2] LLVM,[3] or MSVC,[4] reads the source code of a program and writes out the code for a functionally equivalent program [29]. We will consider the issue of transformations that produce functionally different programs in section 2.3. Usually, the programming languages of the input and output programs are different—humans often write in higher-level languages like C++, JavaScript or Python and rely on compilers to translate them into lower-level languages like assembly or machine code. Conceptually, compilers accomplish this transformation in phases: a front-end phase that translates the input program into an intermediate representation, an optimization phase

[1] In this context, we use "optimization" to refer to the process of improving (i.e., increasing or decreasing, as appropriate) the properties of a program. For reasons we will return to in section 2.2, we do not use "optimization" in the mathematical sense of identifying the best possible solution.

[2] https://gcc.gnu.org/

[3] http://llvm.org/

[4] https://www.visualstudio.com/

that modifies the intermediate representation, and a back-end that translates the intermediate representation into the target language [29]. In practice, the situation is somewhat messier, as some optimizations, such as machine-specific instruction choices, may be performed during the front- or back-end phases [75]. For example, the LLVM compilation framework supports optimization of separately compiled files at the time they are linked together [76]. Indeed, certain just-in-time compiler (JIT) environments (e.g., many Java environments) perform some optimizations during compilation to an intermediate representation (e.g., bytecode), but delay others until the time the program is run [77]. We will return to JITs in more detail in section 2.1.2. Regardless of when an optimization is applied, however, it generally modifies the contents of a particular representation—source code, intermediate representation, or output code—of a program, but does not change the language used to represent it.

As noted above, the purpose of program optimization is to produce a program with the same functionality but different non-functional properties—most often run time. Traditionally, compiler optimizations are applied statically. That is, the compiler will apply an optimization only when it is safe and likely to improve the program, but the compiler will usually not verify the magnitude of that improvement. (See section 2.1.2 for some techniques that do take dynamic behavior into account.) For example, peephole optimization [72] recognizes patterns in very short sequences of instructions—often just two or three instructions—in the target language and replaces them with an equivalent, more efficient sequence.

To ensure that optimizations do not change the program functionality, each compiler optimization includes a program analysis to determine the locations at which the optimization may be safely applied [78]. In some cases, such as peephole optimization, this analysis is as simple as pattern matching. Since applying an optimization necessarily changes the program, the set of safe locations for one optimization may be affected by the application of another. In particular, after applying one optimization, there may be more or fewer opportunities for a second optimization [79]. Indeed, the optimal order in which optimizations should be applied may vary from program to program or even within a single program. Compilers typically use offline heuristics to determine the order in which to apply optimizations, in some cases applying the same optimizations multiple times [75], although some researchers have suggested using genetic algorithms (GAs) to identify beneficial orderings [80, 81].

2.1.1 Superoptimization

Superoptimization is a technique that replaces short, typically loop-free sequences of instructions in a compiled program with a more efficient sequence [82] and may be considered an automated form of peephole optimization [83]. Unlike the optimizations mentioned in previous paragraphs, superoptimization does not rely on a human compiler writer to explicitly define the transformation between one sequence of instructions and another. Instead, large numbers of candidate sequences are generated automatically (either exhaustively or stochastically [84]) and checked for equivalence

with the original sequence. Equivalent sequences may then be compared according to their estimated performance. The potential need to check enormous numbers of sequences, as well as technical limitations, typically prevent measuring the performance directly. This, in turn, tends to limit superoptimization to generating sequences of assembly or machine language instructions, whose performance can be modeled more directly than instructions in higher-level languages.

Superoptimization is not typically implemented as part of an optimization phase in current commercial compilers. However, sequences of instructions identified by superoptimization may be incorporated as peephole optimizations.

2.1.2 Profile-Guided Optimization

Profile-guided optimization (PGO) [85, 86, 87] techniques incorporate some information about a program's dynamic behavior when determining optimizations to apply. They rely on indicative workloads, sets of inputs that cause the program to execute the functionality to be optimized. By running the workload and recording metrics, such as execution counts, the compiler can gain information about which transformations are more likely to produce an improvement. The dynamic metrics collected while running the indicative workload are collected into a profile, which maps parts of the program to the recorded metric values. For instance, a profile for a C program may map the functions, branches, and statements of the program to execution counts; a profile for an assembly program may map them to individual machine instructions.

PGO techniques use the information encoded in the profile to identify where to apply optimizations or to choose between competing optimizations. For example, certain transformations can improve the performance of some program behavior at the expense of other behavior [86]; knowing that some code executes frequently could allow a compiler to optimize access to the associated variables instead of others. Additionally, different transformations may be selected based on the profile, allowing performance optimizations to be applied to time-consuming portions of the program and memory-conserving optimizations to be applied elsewhere [88]. As with other compiler optimizations, traditional PGO approaches must determine the locations where optimizations may safely be applied without modifying the program's functionality. If the profile indicates that an optimization would be beneficial in one location but the analysis cannot determine that it is safe to do so, the optimization cannot be applied. As noted below (section 2.3) and in more detail in chapter 4, certain applications may tolerate optimizations that modify program functionality; however, in this section, we remain focused on traditional, semantics-preserving optimizations.

PGO has been integrated into many common compilers, including GCC,[5] LLVM,[6] and MSVC.[7] It is also heavily integrated into many JITs, including the Java HotSpot compiler [89]. Since JITs operate in conjunction with running the program, they implicitly operate on the most accurate indicative workload—the actual use case itself. They also

[5] https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html#index-fprofile-use, accessed 06/2017.
[6] https://clang.llvm.org/docs/UsersManual.html#profile-guided-optimization, accessed 06/2017.
[7] https://docs.microsoft.com/en-us/cpp/build/reference/profile-guided-optimizations, accessed 06/2017.

benefit from the opportunity to reevaluate the effectiveness of choices of transformations; those that fail to improve performance or become invalid may be removed, "deoptimizing" the program.

PGO highlights the fact that many transformations exist that may improve some programs under some conditions, but may negatively impact others. For this reason, we view program optimization as a special case of search-based optimization: the task of the optimization phase is to search for the transformations that improve the program in a particular situation. We now discuss the background and terminology of this important field.

2.2 Search-Based Optimization

Search-based optimization is an approach to optimization in which candidate solutions are evaluated to find the best one [90]. The "best" solution is defined in terms of an objective function—which computes a numerical value indicating how "good" the solution is—and a direction—i.e., whether the best solution has the smallest or largest objective value. This broad definition applies to a wide range of algorithms, including Newton's method [26], Dantzig's simplex algorithm for linear programming [91], and the Nelder-Mead simplex algorithm for non-linear optimization [92]. Different SBO algorithms are distinguished primarily by how they select new candidate solutions to evaluate. For example, Newton's method selects the next point as a function of the current point, its objective value, and the first derivative of the objective, while the Nelder-Mead algorithm maintains a set of points, computing new candidate points as a linear function of the points in the set. The possibly infinite set of all possible solutions is called the search space for the problem.

Search-based software engineering refers to the application of search-based optimization techniques to software engineering problems [93]. SBSE has been used to address an enormous variety of problems [28], and spawned its own dedicated conference [94]. In this dissertation, we are primarily interested in applying SBO algorithms to program optimization. Specifically, we find that metaheuristics, particularly evolutionary algorithms (EAs), are well suited to this problem.

2.2.1 Metaheuristics

Metaheuristics are a subset of SBO algorithms that make few assumptions about how solutions are represented or how the objective function measures their quality [13]. EAs, in turn, are a subset of metaheuristics inspired by biological evolution; they tend to maintain a set of candidate solutions and may require solutions to "compete," as described below. Unlike the Nelder-Mead algorithm, which requires solutions to be represented as points in $\mathbb{R}^N$ [92], metaheuristics interact with candidate solutions according to a finite set of abstract operations—essentially just the operations required to modify one or more solutions to get new solutions. The objective functions used for metaheuristics must permit comparing values for order (i.e., to determine whether one solution is better than another), but need not permit operations

such as differentiation, required by Newton's method. Finally, metaheuristics tend to be *stochastic* algorithms; that is, they incorporate randomness as part of their search to help manage vast search spaces.

Evolutionary algorithms use their own, biologically-inspired terminology for many of the concepts of search-based optimization and metaheuristics. Instead of a set of candidate solutions, EAs maintain a population of individuals. In many EAs, particularly genetic algorithms, solution representations are called genomes. A mutation operation modifies one individual to produce another. Each individual's objective value is called its fitness and the objective function is known as a fitness function. Throughout this dissertation, we will use the EA terminology interchangeably with the standard corresponding SBO terminology; the term "population" is more convenient than "set of candidate solutions." EAs frequently involve explicit competition between individuals in the population. For example, GAs frequently compare individuals' fitness values when generating new candidate solutions, selecting the individual with the best fitness to mutate. In some cases, the new individuals may compete to join the population or be discarded [13].

The metaheuristics we consider in this dissertation follow a common high-level algorithmic structure. Each algorithm initializes its population using the original program and sufficient mutants to reach the desired population size. In each iteration of the algorithm, some number of solutions are extracted from the population and mutated to produce new solutions. These new solutions either automatically constitute the population for the next iteration or may replace individuals with lower fitness in the population. This process repeats until a predefined number of fitness evaluations have been performed, at which point the best member of the population is returned. In the following sections, we concretize this algorithm for two metaheuristics, hill climbing and GAs.

2.2.2 Hill Climbing

Hill climbing is one of the simplest possible metaheuristic search algorithms [13], maintaining a single current solution in its population. In the basic algorithm, shown in listing 2.1, in each iteration the current solution is mutated and the mutant's fitness is evaluated. If the mutant's fitness is better than the current solution's, it replaces the current solution; otherwise the current solution is retained. The metaphor is with wandering up the side of a "hill" whose height is given by the fitness function; each mutation takes a "step" in a random direction. A common variant of this algorithm, steepest ascent hill climbing, generates a number (possibly an exhaustive enumeration) of mutants at each iteration, selecting the best for comparison to the current solution. As above, the algorithm performs a predefined number of iterations, then returns the current solution.

The performance of hill climbing algorithms is sensitive to the choice of mutation operation. Mutation operations that produce larger changes may inhibit converging to the true optimum, instead "stepping over" the optimum to the other side of the hill. By contrast, mutation operations that produce small changes may require more search iterations to reach the optimum, since each iteration makes less progress. In addition, small mutations may cause the search to

```
1: procedure HILLCLIMB(p, MaxCount)
2:     current ← p
3:     for 1 ≤ i ≤ MaxCount do
4:         q ← MUTATE(current)
5:         if FITNESS(q) is better than FITNESS(current) then
6:             current ← q
7:     return current
```

Listing 2.1: Hill climbing algorithm.

identify a local, rather than global, optimum. That is, if all mutants of the current solution have worse fitness, the current solution will not be updated, even though the search space contains solutions with better fitness. The standard way to address this latter problem is to repeat the search several times, starting with a different randomly generated-solution each time [28].

2.2.3 Genetic Algorithms

Genetic algorithms replace the metaphor of a single individual climbing a hill with a population of individuals competing for survival [95]. Listing 2.2 shows a generic generational genetic algorithm. After initializing the population (line 2), the algorithm enters its main loop. In each iteration, or *generation*, of the main loop (lines 4 to 12), we build a new population by selecting two parents taken from the current population (line 7), using crossover on the parents to get two children (line 8), and finally mutating the children (line 9). Once the new population has the desired size, it replaces the current population and a new generation begins.

```
1: procedure GENETICALGORITHM(p, MaxCount, PopSize)
2:     P ← INITIALIZEPOPULATION(p, PopSize)
3:     count ← PopSize
4:     while count < MaxCount do
5:         Q ← ∅
6:         while count < MaxCount and |Q| < PopSize do
7:             p1, p2 ← SELECTPARENTS(P, 2)
8:             q1, q2 ← CROSSOVER(p1, p2)
9:             r1, r2 ← MUTATE(q1), MUTATE(q2)
10:            Q ← Q ∪ {r1, r2}
11:            count ← count + 2
12:        P ← Q
13:    return GETBEST(P)
```

Listing 2.2: Genetic algorithm.

Relative to hill climbing, the genetic algorithm introduces a more complex process for generating new individuals, using selection (most often a tournament [13]) and crossover as well as mutation. In a tournament, two or more individuals are chosen at random, with replacement, from the population. The chosen individual with the best fitness "wins" the tournament and is selected to become one of the two parents. Two parents are required so that their

representations may be recombined through crossover. The implementation of crossover operators depends on the way in which solutions are represented. Although crossover operators have been defined for a number of other representations, such as trees or graphs [13, 27], we focus on crossover operators for the vector and list representations used in this dissertation. *One-point crossover* selects one point in each representation, swapping everything after the point in the first representation with everything after the point in the second representation. For fixed-length vectors, the points should be the same so that the new vectors are the same length as their parents. *Two-point crossover* selects two points in each representation, swapping the portions between the two points. As with one-point crossover, the points should be aligned for fixed-length vectors.

2.3 Semantics-Modifying Transformations

In this dissertation, we allow our searches to perform transformations (i.e., mutations) that may alter the semantics of the program. This allows for optimization opportunities unavailable to the semantics-preserving transformations discussed in section 2.1. However, our strategy of using stochastic search-based algorithms to generate programs carries with it the possibility that some of the programs created may have undesirable properties. To take an example that arises in chapter 5, stochastically modifying a program to optimize readability could produce a highly readable program that no longer does anything at all. We must therefore take precautions that the generated programs retain certain semantic properties even as we modify other properties. In particular, we desire a mechanism for detecting semantic similarity. Note that in many cases, such as the problem addressed in chapter 4, we desire (or at least tolerate) semantic change. Thus, instead of mechanism to determine program equivalence, which is undecidable in general [96], we adapt regression testing [35].

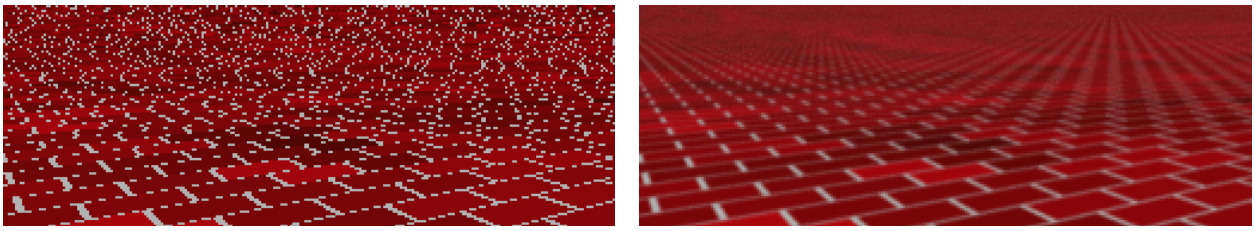
In general, software testing is the process of executing a program to check for differences between the program's required and demonstrated behavior [35]. Regression testing specifically looks for differences in behavior between new and old versions of the same software. Testing a piece of software may be conceptually broken down into running a set of test cases,[8] which are typically automated [97] and may frequently be reused from testing previously new functionality [35]. We define a test to consist of a set of inputs, or workload, along with an oracle comparator[9] to verify the behavior [98]. The result of the test, as determined by the oracle, may *pass* if the behavior is correct or *fail* if not. Alternatively, the oracle may be treat passing and failing on a continuum, returning a real number to indicate the confidence or degree of passing [99]. This latter interpretation of the oracle provides us with the basis for managing semantics-altering transformations.

[8] In this dissertation, we may often use "test" to mean "test case" for brevity.
[9] For brevity, we also use "oracle" to mean "oracle comparator."

We incorporate regression testing into all of the fitness functions in this dissertation. That is, each fitness function requires a workload and an oracle to verify the correct behavior of the modified program; naturally, the fitness function must also incorporate a mechanism to estimate the non-functional properties to be optimized. This implies that every fitness function in this dissertation attempts to run the modified programs generated during the search. The oracle comparators we use return real numbers to indicate the degree to which the program's functionality was affected. In all cases, we apply a threshold, assigning pessimal fitness to individuals whose functionality differs too greatly from the original program (e.g., by not producing any output at all).

2.4 Summary

This chapter presents background information related to program optimization interpreted as a search problem. This material forms the basis for the framework introduced in chapter 1. To make this framework concrete, in subsequent chapters, we instantiate this search framework with fitness functions (to estimate the desired program properties) and a set of mutation operators (along with the representation on which they act).



(a) Significant aliasing.

(b) Reference image, limited aliasing.

Figure 3.1: Example of aliasing in a rendered image. Image (a) shows significant aliasing artifacts, while image (b) has acceptable visual quality, but required over two orders of magnitude more time to render. Both images were produced at the same resolution; the pixels are enlarged to show detail.

Chapter 3

Run Time and Visual Quality

3.1 Introduction

MODERN graphics systems rely on procedural shaders for their flexibility and expressiveness in specifying the material appearance in a virtual scene [100]. These programs compute the visual properties of surfaces, lights, and even the "atmosphere" of a rendered scene [21]. Despite their prevalence, however, procedural shaders[1] remain sensitive to an especially problematic source of visual error known as aliasing. Aliasing artifacts may manifest in a number of ways, 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. For example, consider figure 3.1. Figure 3.1(a) shows jagged lines in the foreground (toward the bottom of the image) which become broken and seem to appear and disappear in the middle- and background (toward the top of the image). Compare these aliasing artifacts to figure 3.1(b), which displays the same scene, but rendered using aggressive and time-consuming

[1] In this dissertation we will often use "shader" to mean "procedural shader" when context clarifies that the shader is a program.

antialiasing techniques to the mitigate these artifacts. The challenge with respect to aliasing in procedural shaders is to develop techniques that reduce noticeable aliasing artifacts while maintaining interactive run times.

Although the prevalent automatic approach to reducing aliasing in procedural shaders requires significant additional computation and run time, it is possible for a developer to design a shader that is less susceptible to aliasing [21]. Shader development is frequently challenging on both an artistic and a programmatic front, as the shader must run efficiently and produce an aesthetically pleasing image. The requirement that the programmer must figure out how to produce the correct image and then address aliasing as well adds further complexity to an already demanding process. In this chapter, we use our optimization framework to develop an automatic approach to reducing aliasing while maintaining short run times in existing procedural shaders. This allows the developer to write a program that generates the correct image (notwithstanding aliasing) quickly, then apply automatic transformations to reduce aliasing. While we focus in this chapter specifically on shaders for texture mapping, which are responsible for the appearance of surfaces, our approach may apply to many other types of shaders.

To achieve this, we pay particular attention to the transformations our search framework applies. We take a bottom-up approach, designing local transformations to the source code that reduce aliasing where they are applied and that may be composed to reduce aliasing overall. The fitness function and search component of our framework identify those locations at which the transformations were most beneficial and prune away non-constructive compositions. The remainder of this chapter is organized as follows. First, we lay out the background on which we construct our transformations (section 3.2). Then we develop the local and composed transformations in section 3.3. We briefly discuss the fitness function and search components in section 3.4. In section 3.5 we present and discuss our experimental results. Finally, we place our approach in the context of related work in section 3.6 and conclude the chapter (section 3.7).

3.2 Computer-Generated Imagery Background

Computers display images as a grid of colored points called pixels. The problem for computer generated imagery is to determine which color to assign to each pixel so that a human observer will perceive it as a recognizable image. Since they are frequently assembled into animation sequences, these images (often called frames in this context) must be generated quickly enough to support the illusion of motion (*real-time graphics*, such as video games) or meet production deadlines (*offline rendering*, such as computer-animated movies). If we consider the scene definition—viewer position, geometry, light sources, surface and material properties—to imply a function that describes the image, then determining a color for the pixel reduces to sampling that function once for each pixel. Graphics researchers formalize this image

function as the *rendering equation* [101], which we present in simplified form below:

$$color(w) = \int_X I(x)\, m(x)\, p(x, w)\, dx, \tag{3.1}$$

where $x \in X$ is a vector concatenating the coordinates of a point on a surface, the light direction, and other parameters; w describes the pixel shape and location; and I, m, and p respectively represent the incoming light intensity, material properties, and the contribution of light reflected from point x to the pixel described by w.[2]

This integral often cannot be calculated analytically; in practice, rendering systems employ various heuristics to approximate it numerically [101]. The simplest approach is to ignore the integral completely and sample the integrand once. This approach may produce significant error if the point sampled is not sufficiently representative of the integral as a whole. For example, imagine an image of a picket fence. If the samples are taken every few inches so that every sample coincides with a fence slat, the result will appear to be a solid wall. Conversely, if the samples happen to consistently fall between slats, the result will not include the fence at all. The necessary sampling interval depends on the scene itself.

The Nyquist-Shannon sampling theorem [104] gives a precise mathematical relationship between the complexity of the image and the required distance between samples to represent it accurately. At a high level, images with more detail require greater numbers of samples to capture. More specifically, if the coefficients of the Fourier decomposition [105] of the image function are effectively zero for all frequencies greater than B (i.e., B is the bandwidth of the function), then the maximum sufficient sampling interval (called as the Nyquist interval) is $\frac{1}{2B}$. Aliasing arises when samples are more widely spaced than the Nyquist interval [106]. The preceding suggests two fundamental strategies to mitigate aliasing—reduce the sampling interval so that it is below the Nyquist interval or increase the Nyquist interval by removing high frequencies from the image. The two common approaches to antialiasing shaders, supersampling and prefiltering, take the former and latter strategies respectively.

3.2.1 Antialiasing Strategies

Supersampling evaluates more samples than the number of pixels, then computes the pixel color as a weighted average of the nearby samples. This reduces aliasing and thereby improves visual quality at the cost of significant increases in the computation required for the additional samples. Furthermore, since the Nyquist interval is dependent on the particular image function, the cost of supersampling below the Nyquist interval may vary significantly from scene to scene. Equivalently, supersampling with a fixed interval may still result in aliasing in some scenes. In practice, rather than sampling sufficiently frequently to eliminate aliasing, many renderers shift the sample coordinates by a small

[2] Some researchers have included terms and parameters for light emission [101], surface scattering [102], wavelength and time [103]. Taken together, x may have upwards of 10 dimensions. Further, the light intensity represented by I is itself an integral similar to equation 3.1 that captures the light reflected toward x along a particular direction.

random amount to disguise the aliasing as noise [21]. This allows for more consistent run times, but accepts a certain amount of aliasing as inevitable.

On the other hand, prefiltering computes an alternative image function with reduced detail (i.e. "filtering" out the coefficients for higher frequencies), so as to increase the Nyquist interval. Since increasing the Nyquist interval is equivalent to decreasing the bandwidth of the function, this is also known as band-limiting the function. Prefiltering techniques are designed to produce a function that can be evaluated more directly and efficiently at run time than equation 3.1. For example, prefiltering often takes the form of storing tables of precomputed integrals (e.g., mipmaps [107] or summed area tables [108]). The upside to this approach is that it offers the benefit of exact solutions in many cases with a constant number of operations at run time. The downside is that it increases storage requirements and can replace inexpensive computations with relatively expensive memory accesses. Furthermore, these precomputation strategies scale exponentially in the number of dimensions and so it is typically not practical to precompute integrals of functions with more than two or three dimensions. Recall that in equation 3.1, x may incorporate a three-dimensional point on a surface and a two-dimensional light direction, along with other parameters, resulting in a five—or more—dimensional function.

An alternative strategy to prefiltering is to construct an analytically band-limited version of the shader function. This can be mathematically expressed as the convolution of the function with a band-limiting function [106, 109]. Convolution is a way of combining two functions (e.g., f and k) into a new function that computes the weighted average of one (f) in the vicinity of a point, with the weights defined by the other (k) [21]. (The band-limiting function k is often referred to as a kernel, hence the mnemonic.) Convolution is frequently applied to measured data in signal and image processing [110], but is more difficult to apply directly to the functions generating that data. In most cases, the shader developer must manually calculate the convolution integral. Because this calculation is usually complicated and time-consuming for realistic shaders, it is rarely done in practice.

3.2.2 Band-Limiting for Procedural Shaders

This chapter explores the problem of automatically computing an exact analytic band-limited version of a procedural shader function. Our system takes as input a shader program written in a high-level language with any number of user-defined parameters and output a modified program that produces a band-limited version of the input shader. To permit the shader to be used in different situations with different sampling intervals—e.g., different monitor resolutions—we parameterize the band-limited shader with respect to the sampling intervals. The approach also allows limiting the function to different bandwidths without requiring significant rendering times.

We instantiate our optimization framework (see section 1.1) to transform an existing shader program into a band-limited version. We exploit the observation that, under certain conditions, band-limited subexpressions may be

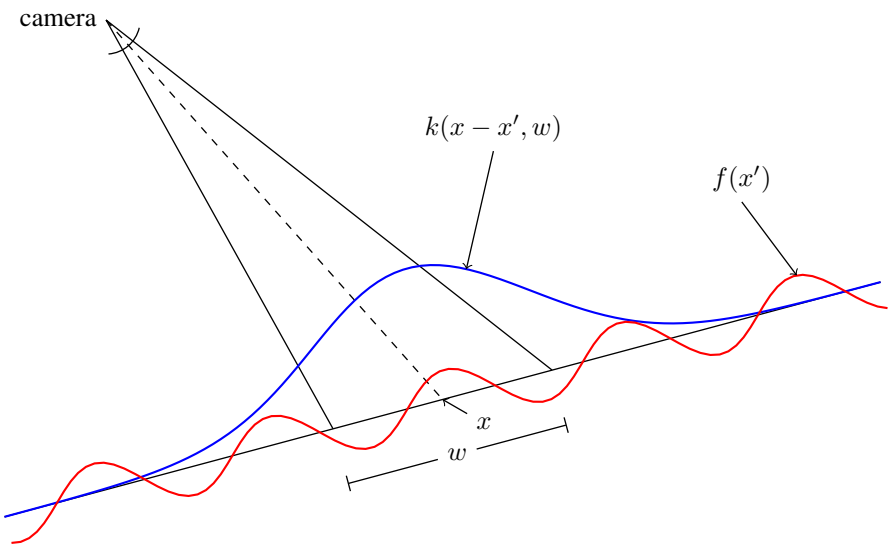


Figure 3.2: Determining the color of a pixel. The center of a single pixel projects to point x on the surface. The spacing between pixels at the camera corresponds to a local surface spacing of w. To determine the best single color for the pixel, we convolve the shader function $f(x')$ with the band-limiting kernel $k(x - x', w)$. Note that the sampling interval w is a parameter to the kernel function.

composed to produce a larger band-limited expression. Our approach improves upon formerly ad-hoc band-limiting strategies, such as replacing primitive functions that cause significant aliasing with others that cause less [21]. We represent shader programs as abstract syntax trees (ASTs) [29]; our transformations replace calls to functions that are susceptible to aliasing with the band-limited counterparts we derived (table 3.1). To support this, we develop two new analyses on shader program source code. The first recognizes subexpressions for which we have derived solutions to the band-limiting convolution (sections 3.3.1 and 3.3.2). This allows us to replace the original subexpression with a band-limited expression. The second analysis conservatively approximates the relevant sampling intervals for each subexpression (section 3.3.5). This allows the transformed subexpressions to be limited to the correct bandwidth; too small a bandwidth would remove desirable detail from the scene while too large a bandwidth would result in aliasing. As the fitness function for our search-based approach, we generate several images with the modified shader and compute the image difference with respect to the corresponding "ground truth" images. Since the true band-limited function is unknown (the purpose of the search is to find it) we use supersampling, which can approximate the desired image at the expense of long run times, to generate the ground truth images.

3.3 Band-Limiting Transformations

This chapter considers the problem of analytically computing the band-limited version of a procedural shader function. Figure 3.2 illustrates the problem in one dimension. Formally, given a shader function $f(x)$ of a single coordinate x, and a band-limiting kernel function $k(x, w)$ with sampling interval w, we desire the band-limited shader function

$\widehat{f}(x, w)$, obtained by convolving f with k:

$$\widehat{f}(x, w) = \int_{-\infty}^{\infty} f(x')\, k(x - x', w)\, dx'. \tag{3.2}$$

Extending this to multiple dimensions is straightforward: the single-dimensional variables in equation 3.2 are replaced by vectors with the appropriate dimensionality. We make no restrictions on the sampling intervals along different dimensions; in particular, we do not assume that the sampling interval in one dimension is a function of the interval in another. Thus, we replace w by a vector with the same number of dimensions as x. We return to the question of multiple dimensions in section 3.3.4. Note the similarity between equation 3.2 and equation 3.1, where $f(p) = I(p)\, m(p)$ and k represents the pixel.

Perhaps unsurprisingly, there is currently no known way of analytically computing these band-limited functions in general. This section describes conditions under which exact solutions are possible and derives closed-form expressions for several common such situations. We then discuss the composition of separately band-limited components and introduce strategies for approximation when exact solutions are not available. Recall that our bottom-up approach section 3.1 applies local band-limiting transformations to subcomponents of the shader and composes them to produce a shader that approximates the band-limited counterpart to the original.

3.3.1 Exact Band-Limited Shaders by Construction

First, note that, given a set of functions for which we have solutions to equation 3.2, the band-limited expression corresponding to any linear combination of those functions is straightforward to compute. Specifically, for any functions $f_1, \ldots, f_n$ and constants $c_0, c_1, \ldots, c_n$, such that

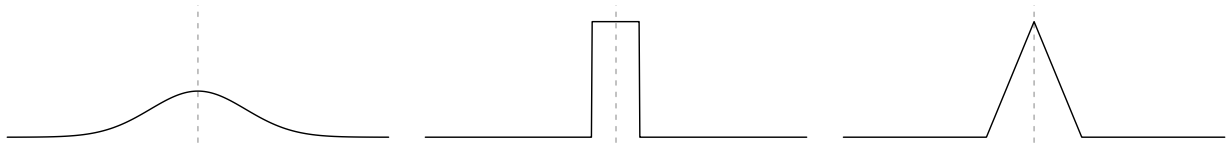
$$g(x) = c_0 + c_1 f_1(x) + \cdots + c_n f_n(x), \tag{3.3}$$

it is the case that

$$\widehat{g}(x, w) = c_0 + c_1 \widehat{f_1}(x, w) + \cdots + c_n \widehat{f_n}(x, w). \tag{3.4}$$

For example, since the truncate function ($trunc$) (i.e., rounding toward zero) can be expressed as $trunc(x) = 1 + \lfloor x \rfloor - step(x)$, where $step$ is the Heaviside unit step function (which returns zero for negative numbers and one for positive numbers), we may immediately conclude that $\widehat{trunc}(x, w) = 1 + \widehat{floor}(x, w) - \widehat{step}(x, w)$.

For many arbitrary functions, equation 3.2 will not have a closed-form solution. However, for many built-in functions that are commonly found in procedural shader languages, such as `floor()` or `saturate()`, this integral can be computed directly, given a suitable choice of k. The band-limiting kernel k incorporates the sampling interval



(a) Gaussian kernel (b) Rectangular kernel (c) Tent kernel

Figure 3.3: The band-limiting kernels considered in this chapter. The Gaussian kernel (a) is used for all derivations in table 3.1 except $fract_R$ and $fract_T$, which use the rectangular (b) and tent (c) kernels, respectively. All three kernels are shown to the same scale.

and in a sense represents the pixel; in this chapter we take k to be the Gaussian function (see figure 3.3(a)) with a standard deviation equal to the sampling interval w. The Gaussian function is commonly used in image processing applications [111] and we find that it permits more aesthetically pleasing results than using a kernel corresponding to rectangular pixels (e.g., figure 3.5(a)). We present the band-limited expressions for several built-in functions in table 3.1. Full derivations for these expressions are available in appendix A. The derivation and presentation of exact band-limited expressions in this table is a contribution of this work.

3.3.2 Band-Limiting the $fract(x)$ Function

We now discuss in some detail the derivation of a band-limited expression for the fract function (*fract*), defined as $fract(x) = x - \lfloor x \rfloor$. This function highlights some of the interesting challenges involved in calculating the band-limiting integrals even for ostensibly "simple" functions and provides a framework to demonstrate some useful derivation strategies. It also captures the "hard part" of other common functions, such as $\lfloor x \rfloor$, $\lceil x \rceil$, and $trunc(x)$, since it is straightforward to define the band-limited expressions for these functions in terms of $fract(x)$ while deriving them directly is much more challenging.

Recall that we derive the band-limited form of a function from equation 3.2, in this case substituting *fract* for f and, for k, the Gaussian function with standard deviation w. Specifically,

$$\widehat{fract}(x, w) = \int_{-\infty}^{\infty} fract(x') \frac{1}{w\sqrt{2\pi}} e^{-\frac{(x-x')^2}{2w^2}} \, dx'. \tag{3.5}$$

To solve equation 3.5, we use the convolution theorem [105], which states that the convolution of two functions f and k can be determined by taking the product of their Fourier transforms and the inverse Fourier transform of the result. The Fourier transform of a function $f(x)$ is a function describing the frequency content of $f(x)$.

For brevity, we sketch the derivation here; see appendix A.3.5 for the full derivation. First, we rewrite *fract* in terms of its Fourier series expansion:

$$fract(x) = \frac{1}{2} - \sum_{n=1}^{\infty} \frac{1}{\pi n} \sin(2\pi n x). \tag{3.6}$$

$f(x)$	$\hat{f}(x,w)$	Derivation
c	c	Prop. A.8
x	x	Prop. A.9
x^2	$x^2 + w^2$	Prop. A.27
$fract_G(x)$	$\frac{1}{2} - \sum_{n=1}^{\infty} \frac{e^{-2w^2\pi^2n^2}}{\pi n} \sin(2\pi nx)$	Prop. A.18
$fract_R(x)$	$\frac{1}{2w}\left(fract^2\left(x+\frac{w}{2}\right) + \left\lfloor x+\frac{w}{2}\right\rfloor - fract^2\left(x-\frac{w}{2}\right) - \left\lfloor x-\frac{w}{2}\right\rfloor\right)$	Prop. A.21
$fract_T(x)$	$\frac{1}{12w^2}\left(F(x+w) - 2F(x) + F(x+w)\right)$	Prop. A.24
	where $F(x) = 3x^2 + 2fract^3(x) - 3fract^2(x) + fract(x) - x$	
$\lvert x\rvert$	$x\,\mathrm{erf}\left(\frac{x}{w\sqrt{2}}\right) + w\sqrt{\frac{2}{\pi}}\,e^{-\frac{x^2}{2w^2}}$	Prop. A.14
$\lfloor x\rfloor$	$x - \widehat{fract}(x,w)$	Prop. A.17
$\lceil x\rceil$	$\widehat{floor}(x,w) + 1$	Prop. A.15
$\cos x$	$\cos x\, e^{-\frac{w^2}{2}}$	Prop. A.16
$saturate(x)$	$\frac{1}{2}\left(x\,\mathrm{erf}\left(\frac{x}{w\sqrt{2}}\right) - (x-1)\,\mathrm{erf}\left(\frac{x-1}{w\sqrt{2}}\right) + w\sqrt{\frac{2}{\pi}}\left(e^{-\frac{x^2}{2w^2}} - e^{-\frac{(x-1)^2}{2w^2}}\right) + 1\right)$	Prop. A.25
$\sin x$	$\sin x\, e^{-\frac{w^2}{2}}$	Prop. A.26
$step(x)$	$\frac{1}{2}\left(1 + \mathrm{erf}\,\frac{x}{w\sqrt{2}}\right)$	Prop. A.28
$trunc(x)$	$\widehat{floor}(x,w) - \widehat{step}(x,w) + 1$	Prop. A.29

Table 3.1: Band-limited versions of several common one-dimensional primitive shader functions. The band-limiting kernel used to derive the second column is the Gaussian function with a standard deviation equal to the sampling interval w, except $fract_R$ and $fract_T$, which use the rectangular and tent functions, respectively. The *fract* function, used as the basis of $\lfloor x\rfloor$, $\lceil x\rceil$, and $trunc(x)$, is defined: $fract(x) = x - \lfloor x\rfloor$. The *trunc* function truncates its argument to the nearest integer in the direction of zero. The Gauss error function (erf) is defined $\mathrm{erf}\,x = \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}\,dt$.

Following the convolution theorem, we take the product of the Fourier transforms of this function and of the Gaussian kernel, then apply the inverse Fourier transform to the product, obtaining the convolution of the original functions:

$$\widehat{fract}(x,w) = \frac{1}{2} - \sum_{n=1}^{\infty} \frac{e^{-2w^2\pi^2n^2}}{\pi n} \sin(2\pi nx). \tag{3.7}$$

Although this expression—labeled $fract_G$ in table 3.1—is exact, it contains an infinite sum that cannot be evaluated in practice. However, the terms in this sum are scaled by $e^{-2w^2\pi^2n^2}$. Thus, the absolute values of the later terms rapidly approach zero. Therefore, one approximation method is to simply truncate this sum after a fixed number of terms

or once an error bound is reached. Unfortunately, we found that this approach can still yield long running times and noticeable artifacts, especially when used in conjunction with other functions (we will return to this issue in section 3.3.7 and figure 3.5(c)).

As a second approximation to the convolution of *fract* with a Gaussian, we also considered its convolution with other band-limiting kernels. For illustrative purposes, we first derive the convolution of *fract* and the rectangular function (figure 3.3(b)),

$$\mathit{rect}(x, w) = \frac{1}{w}\left(\mathit{step}\left(x + \frac{w}{2}\right) - \mathit{step}\left(x - \frac{w}{2}\right)\right). \tag{3.8}$$

To solve this convolution, we use Heckbert's technique of repeated integration [112], which is based on the observation that the convolution of f and k is equivalent to the convolution of the n^{th} integral of f with the n^{th} derivative of k. We note that the derivative of *rect* is zero almost-everywhere, except at $x - \frac{w}{2}$ and $x + \frac{w}{2}$, where it is positive and negative, respectively. Functions with this form—zero almost-everywhere except a set of discreet points—are sometimes called pulse trains. Finite-length pulse trains are the key to using the technique of repeated integration, since convolving a function with one reduces to evaluating the function at a finite number of coordinates and multiplying by the appropriate coefficients. Turning to the indefinite integral of *fract*, we find that it is given by (see proposition A.20 for details),

$$\int \mathit{fract}(x)\,dx = \frac{\mathit{fract}^2(x) + \lfloor x \rfloor}{2}. \tag{3.9}$$

Evaluating this formula at $x - \frac{w}{2}$ and $x + \frac{w}{2}$ and multiplying, the result of the convolution is given in table 3.1 (indicated by fract_R to distinguish it from convolution with the Gaussian function).

For our final example, we convolve *fract* with the tent function (figure 3.3(c)),

$$\mathit{tent}(x, w) = \frac{1}{w}\max\left(0, 1 - \left|\frac{x}{w}\right|\right). \tag{3.10}$$

This function approximates the Gaussian better than the *rect* function while still permitting straightforward use of repeated integration. Although the first derivative of *tent* is not a pulse train, the second derivative is, with a negative coefficient at x and positive coefficients at $x \pm w$. Thus, we can compute the second indefinite integral of *fract* (see proposition A.23), evaluate it at these three locations and multiply by the appropriate coefficients to get the convolution of the original functions, indicated by fract_T in table 3.1.

3.3.3 Band-Limiting Control Flow

The goal of our bottom-up approach is to construct a band-limited shader program out of band-limited sub-components. Having discussed how we construct band-limited components corresponding to the primitive functions of the shader

programming language, we now turn our attention to the control flow of that language.

In general, we handle control flow by transforming the program to compute the same function without it. For example, we take advantage of the fact that many procedural shaders contain statically-unrollable loops and statically-bounded function nesting. This allows us to unroll loops and inline function calls throughout the shaders. (We return to the subject of loops and unbounded recursion in section 3.5.3.) After applying these transformations, we eliminate `if`-statements by executing both branches and joining the results using a ϕ-function [113]. These functions implement if-then-else (familiar to C programmers as the ternary operator `?:`) in an assignment statement and were introduced to simplify compiler analyses that require each variable to be assigned exactly once. Our use of ϕ-functions (i.e., executing both branches, then joining) is similar to the elimination of branches through conditional instructions [114, 115].

We build our ϕ-function on top of the *step* function. Note that $step(x)$ computes the Boolean expression $x \geq 0$ with 0 representing *false* and 1 representing *true*. This can be extended to arbitrary conditions of the form $a \geq b$, since this is equivalent to $a - b \geq 0$. With this same encoding of *true* and *false*, we implement Boolean *and* as multiplication, Boolean *or* as addition, and Boolean *not* as subtraction from 1. This allows us to implement arbitrary Boolean combinations of inequalities[3] by translating the Boolean operators into multiplication, addition, and subtraction and the inequalities into calls to *step*. Our ϕ-function multiplies the expression from the *true* branch by the translated condition and adds the product of the expression from the *false* branch and the translation of the negated condition. If a variable x is assigned along only one branch, we proceed as if the other branch included the assignment $x = x$. In effect, this transformation allows the band-limiting expression to integrate over both branches of the `if`-statement. Heidrich et al. [116] used the *step* function in a similar fashion to modify `if`-statements for use in their analysis, but did not to our knowledge use it to implement ϕ-functions.

We present an example of this transformation in listing 3.1. Since `y` is assigned only along the *true* (line 3) branch and `x` is assigned only along the *false* (line 5) branch, we introduce dummy assignments `x=x` and `y=y` along the other branches (i.e., after lines 3 and 5, respectively). The transformation then moves all assignments outside of the if statement, introducing temporary variables `y1` and `x1` to hold the values from the *true* branch as well as `y2` and `x2` for the *false* branch. Next, it introduces another temporary variable for the translated condition, using the identity `a < b` $\equiv$ `!(a >= b)`, then replacing the Boolean operators as described in the previous paragraph. Finally, it assigns the result of each ϕ-function to the corresponding variable (lines 14 and 15)

Now consider what happens if `a` $= 1$, `b` $= 2$, and `c` $= 3$. Then `cond` is set to $(1 - step(1 - 2))\,(1 - step(2 - 3))$, which evaluates to $(1 - 0)\,(1 - 0) = 1$, indicating the condition is true. In this case, `y` is set to `y1` (which from line 9 is equal to `x`) and `x` is set to (which from line 10 is set to `x`) `x1`. On the other hand, if `a` $=$ `c` $= 3$ and `b` $= 2$, then `cond` will be $(1 - step(3 - 2))\,(1 - step(2 - 3)) = 0$. This results in `y` being set to `y2` (which from line 12 is equal to `y`) and `x` being set to `x2` (which from line 11 is equal to `y`), as desired.

[3] Equalities involving spatially varying quantities have measure zero and are therefore effectively always false.

```
1  // before transformation
2  if (a < b && b < c) {
3    y = x;
4  } else {
5    x = y;
6  }

8  // after transformation
9  y1 = x;
10 x1 = x;
11 x2 = y;
12 y2 = y;
13 cond = (1-step(a-b))*(1-step(b-c));
14 y = cond*y1 + (1-cond)*y2; // y = φ(cond,y1,y2)
15 x = cond*x1 + (1-cond)*x2; // x = φ(cond,x1,x2)
```

Listing 3.1: Example of replacing `if`-statements in shader programs. Both branches of the original `if`-statement are executed, using temporary variable names, then any variables assigned in the bodies are joined using a ϕ-function.

By constructing our ϕ-function in terms of *step*, which appears in table 3.1, we are able to rewrite `if`-statements as expressions of sub-components that may be band-limited. This allows us to apply our bottom-up transformations to programs with combinations of simple functions and control flow.

3.3.4 Extension to Multiple Dimensions

Building expressions from arbitrary combinations of language primitives, whether done by the programmer or our ϕ-function transformation, can produce functions with high dimensionality. Instead of attempting to band-limit these analytically in all cases, we present several heuristics in this section that allow us to combine components that were band-limited separately into an approximately band-limited whole. We discuss the circumstances in which these heuristics produce exact answers as well as mechanisms to mitigate error when the result is less than exact.

So far, we have addressed band-limiting only one-dimensional functions. However, procedural shaders are often functions of multiple variables, $\vec{x} = \langle x_1, x_2, \ldots, x_n \rangle$ (e.g., texture coordinates, components of the normalized vector towards the camera, etc.). Although the convolution integral is easily extended to functions of multiple dimensions, solving it analytically becomes increasingly difficult with each additional dimension. Instead, our insight is to understand when the simpler one-dimensional solutions may be directly combined and introduce a heuristic to make other situations resemble the desired condition more closely.

Specifically, we identify partial multiplicative separability as an important condition for combining simpler solutions. (Throughout this chapter, we will often use *separability* to mean *partial multiplicative separability*.) In the special case where the shader function and the band-limiting kernel are separable, the band-limited version of $f(\vec{x})$ may be written

in terms of band-limited subexpressions. Formally, let us partition the dimensions of $\vec{x}$ into two disjoint sets, A and B, and write $f_A(\vec{x})$ to denote a function that depends only on the dimensions of $\vec{x}$ in set A and similarly with $f_B(\vec{x})$. Recall from section 3.3 that, for generality, we allow $\vec{w}$ to have the same dimensions as $\vec{x}$. Now, if there exist sets A and B such that

$$f(\vec{x}) = f_A(\vec{x})f_B(\vec{x}) \quad \text{and} \quad k(\vec{x},\vec{w}) = k_A(\vec{x},\vec{w})k_B(\vec{x},\vec{w}) \tag{3.11}$$

then

$$\widehat{f}(\vec{x},\vec{w}) = \widehat{f_A}(\vec{x},\vec{w})\widehat{f_B}(\vec{x},\vec{w}). \tag{3.12}$$

Using equations 3.4 and 3.12 to compute $\widehat{f}$ in terms of simpler band-limited functions is the core of our bottom-up approach (cf. section 3.1). These equations indicate that we will generate exact band-limited shaders under the right conditions, namely equations 3.3 and 3.11, when the shader function and band-limiting kernel may be decomposed into simpler functions for which the band-limited version is available.

In general, of course, many shader functions cannot be factored into simple products or linear combinations of known band-limited functions. In fact, there are many situations where the Gaussian band-limiting kernel itself cannot be factored. For example, consider the important case of shader functions of two spatial dimensions (e.g., surface texture coordinates). Recall from section 3.3 that the band-limiting kernel represents the pixel for which we desire a color. For simplicity, we treat the pixel temporarily as a rectangle instead of a Gaussian, and illustrate the following discussion in figure 3.4. The region of the surface subtended by a single pixel is a quadrilateral—the intersection of the surface and the pyramid formed by the viewpoint and the corners of the pixel—if we assume the surface is effectively planar over such a small region. In general, a quadrilateral cannot be expressed as the product of two independent rectangular functions, so it is not multiplicatively separable.

To handle this frequent situation, we start with the common approximation of the quadrilateral as a parallelogram [109] (lighter shaded region in figure 3.4). Our insight is to further approximate this parallelogram by its axis-aligned bounding rectangle (darker shaded region in the figure). If x and y denote the image plane coordinate axes and s and t denote the surface texture coordinate axes, the lengths $|\partial s/\partial x| + |\partial s/\partial y|$ and $|\partial t/\partial x| + |\partial t/\partial y|$ define extent of the bounding rectangle in s and t, respectively. Since the pixels of the screen induce horizontal and vertical sampling intervals that are the width and height, respectively, of the pixels, this approximation also gives us an approximation of the sampling intervals in surface coordinates. The bounding rectangle is multiplicatively separable, enabling separable surface functions to be band-limited exactly. Note that this approach is exact whenever the image plane axes are aligned with the texture coordinate axes (i.e., the quadrilateral is the same as the rectangle in the figure) and favors blurring over aliasing otherwise.

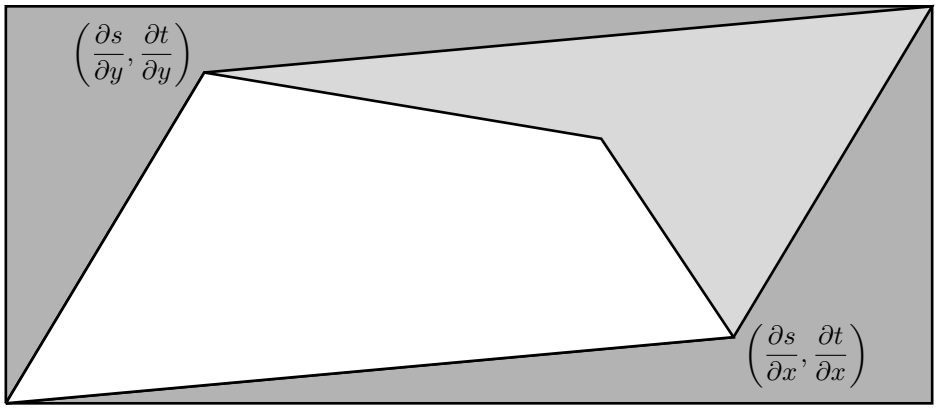


Figure 3.4: Axis-aligned approximation of the projection of a pixel onto a surface. To distinguish the pixel coordinate system from the coordinate system of the surface, we use x and y for the former and s and t for the latter. The figure is shown in the surface (s and t) coordinate system. Assuming locally planar geometry, the projection of a single pixel in the image plane subtends a quadrilateral (white region) in the surface coordinate system. We approximate this quadrilateral by the axis-aligned bounding rectangle (dark shading) around the parallelogram (light shading) formed by two of its edges. The shaded regions indicate the extent of over-approximation of the sampling intervals, causing too much band-limiting which manifests as over-blurring the image.

3.3.5 Determining the Sampling Rate

Having developed an approach to acquire separable sampling intervals on entering the function, we now discuss the use of sampling intervals within the function. To achieve the correct degree of band-limiting, a band-limited expression must incorporate the sampling intervals for each of the parameters of the original expression as additional parameters. For example, consider a modification of the checkerboard shader in listing 3.4 that replaces `fract(p.s)` with `fract(p.s*2)` to produce checks that are taller than they are wide. This doubles the effective interval between samples of the band-limited expression; the sampling interval parameter must be doubled to reflect this.

In general, the interval between samples of the result of an expression may be different from the interval between samples of its inputs. Note that we cannot simply compute the sampling interval based on the partial derivative with respect to the input parameters. For example, the derivative of $\lfloor x \rfloor$ with respect to x is 0 (indicating that the value is unchanging) almost everywhere, yet this does not imply that every function of $\lfloor x \rfloor$ is smooth and free from aliasing (we show an example of aliasing in such a function in section 3.3.7). Nor can we use finite differencing methods, such as `dFdx`/`dFdy` in the OpenGL Shading Language [117] or `Du`/`Dv` in the RenderMan Shading Language [21], which estimate the derivative by sampling their parameter at adjacent pixels. If the sampling interval is sufficiently small, these methods approximate the derivative closely and are subject to the same limitations as analytically-computed derivatives. For sufficiently large sampling intervals, these methods are themselves subject to aliasing.

Instead, we develop an analysis in the style of traditional dataflow analysis [78] to approximate the sampling interval as a function of the axis-aligned projection of the pixel onto the surface (see section 3.3.4 and figure 3.4). The values propagated through our analysis do not form a finite height lattice; thus, Kildall's proof [78] that dataflow analyses

terminate does not apply. However, the transformations described in section 3.3.3 produce shaders without loops, ensuring that our analysis terminates.

We make a simplifying approximation by modeling the subexpression sampling interval as a polynomial function of the axis-aligned sampling intervals in each dimension, where each term includes only dimensions of degree one. In the example of the modified checkerboard above, our analysis would determine the sampling interval for `p.s*2` is $2w_s$ (assuming the sampling interval for `p.s` is w_s). Our analysis assigns each constant expression c the dimensionless sampling interval with coefficient c. For addition and subtraction expressions, we assign the vector representing the sum of the polynomials for the daughter terms. Multiplication expressions are assigned the vector representing the product of the polynomials, unless that product would include dimensions with degree greater than one. For all other expressions, including multiplication that would result in dimensions with degree two or greater, we assign the average of the vectors for the inputs.

3.3.6 Band-Limiting Transformation in a Nutshell

We may now summarize the previous subsections to describe our band-limiting transformation. Preparatory to band-limiting, we extend the shader function to compute the sampling interval for each of the original shader's parameters, applying the axis-aligned approximation discussed in section 3.3.4. In the pseudo-code examples in this chapter, we use OpenGL-inspired fuctions—`dFdx(p)` and `dFdy(p)`—to retrieve the two quadrilateral sides that are the basis of the approximation.[4] This step is performed automatically once for the shader, independent of the number of times the band-limiting transformation is applied.

Our transformation applies to any node in the AST of a prepared shader that is not already band-limited. We define the set of nodes that are already band-limited as follows. First, we consider all constants and variable references to be band-limited since $f(x) = \widehat{f}(x)$ in the first two rows of table 3.1. Next, we consider all addition and subtraction nodes to be band-limited, following equation 3.4. Similarly, we consider multiplication nodes where one side of the multiplication is a constant to be band-limited for the same reason. Finally, we consider multiplication between expressions based on different input variables to be band-limited, following equation 3.12. The remaining AST nodes, mostly (if not exclusively) calls to primitive functions of the shading language, are considered non-band-limited. We refer to these remaining nodes as non-band-limited nodes.

To transform an AST, we select one of the non-band-limited nodes and replace it with a call to a function implementing the appropriate band-limited expression (e.g., from table 3.1). In general, an expression of n variables, $x_1, \ldots, x_n$, will have a corresponding band-limited expression of $2n$ variables, $x_1, \ldots, x_n, w_1, \ldots, w_n$. Each of the x_i in the non-band-limited expression is retained in the band-limited expression; for each w_i, we use the polynomial

[4] In this special case, it is safe to use `dFdx(p)` when calculating the sampling interval (despite the concerns raised in section 3.3.5) because the band-limited expression on the second line of table 3.1 is independent of the sampling rate and the derivative is truly constant. Thus, finite differencing will produce the correct derivative regardless of the sampling interval.

computed by the analysis described in section 3.3.5 for x_i. For example, to transform the expression $\cos x$, we construct the replacement expression $\cos x \, e^{-\frac{w^2}{2}}$ where w is the polynomial approximation of the sampling interval for x.

3.3.7 Example: Band-Limited Checkerboard

We conclude this section with an example of the preceding results, including a comparison of the different approximation strategies for $fract(x)$ (section 3.3.2). We start with the simple black-and-white checkerboard shader in listing 3.2, constructing the band-limited shader in listing 3.3.

```
1  float3 checker1(float2 p) {
2    float ss = floor(p.s + 0.5) - floor(p.s);
3    float tt = floor(p.t + 0.5) - floor(p.t);
4    return (float3)(ss*tt + (1-ss)*(1-tt));
5  }
```

Listing 3.2: A black-and-white checkerboard shader implemented using the $floor(x)$ function. The shaders in this chapter are written in a C-like pseudocode. Since the shader executes in the surface coordinate system, we use `p.s` and `p.t` to access the s and t surface coorinates, respectively.

```
1  float3 checker1(float2 p) {
2    float2 w = axis_aligned_approx(dFdx(p), dFdy(p));
3    float ss = floor_bl(p.s + 0.5, w.s) - floor_bl(p.s, w.s);
4    float tt = floor_bl(p.t + 0.5, w.t) - floor_bl(p.t, w.t);
5    return (float3)(ss*tt + (1-ss)*(1-tt));
6  }
```

Listing 3.3: Band-limited checkerboard shader corresponding to the shader in listing 3.2.

We first determine the projections of the screen-space vectors describing a single pixel, using our axis-aligned approximation (section 3.3.4) to compute the sampling interval in the surface s and t coordinate system (listing 3.3, line 2). We then construct the body of the band-limited shader in a bottom-up fashion. We first note the already-band-limited nodes in the function: the constants and variable references, along with the addition, subtraction, and multiplication operators. (The multiplication operators are band-limited because they are between expressions that depend on different sets of input parameters.) Next, we simply replace calls to `floor` with calls to the band-limited floor function from the table (here written `floor_bl`); our static analysis determines that the appropriate sampling intervals are the unscaled axis-aligned intervals (lines 3 and 4). Since we have addressed every node in the AST of this simple function, we have finished band-limiting the checkerboard shader.

Figure 3.5 shows the band-limited checkerboard shader applied to an infinite plane. The target image (figure 3.5(d)) was produced using supersampling. Figure 3.5(a) shows the effect of using the rectangular band-limiting kernel in $fract_R$ (which is part of the band-limited expression for `floor`, see table 3.1). Note the significant remaining aliasing near the top of the figure as compared to the tent band-limiting kernel in figure 3.5(b). Finally, figure 3.5(c) shows the effect of truncating the infinite series introduced with the Gaussian band-limiting kernel. For this figure, we truncated the series so that the shader run time matched that required for figure 3.5(b). We note that the figure using $fract_T$ demonstrates little error relative to the original shader (not shown) and the figures using $fract_G$ and $fract_R$, while taking time comparable to the original shader. On this shader, therefore, we have already met our goal of little aliasing with fast run times. For this reason, we use the $fract_T$ band-limited expression for `fract` throughout the remainder of this chapter.

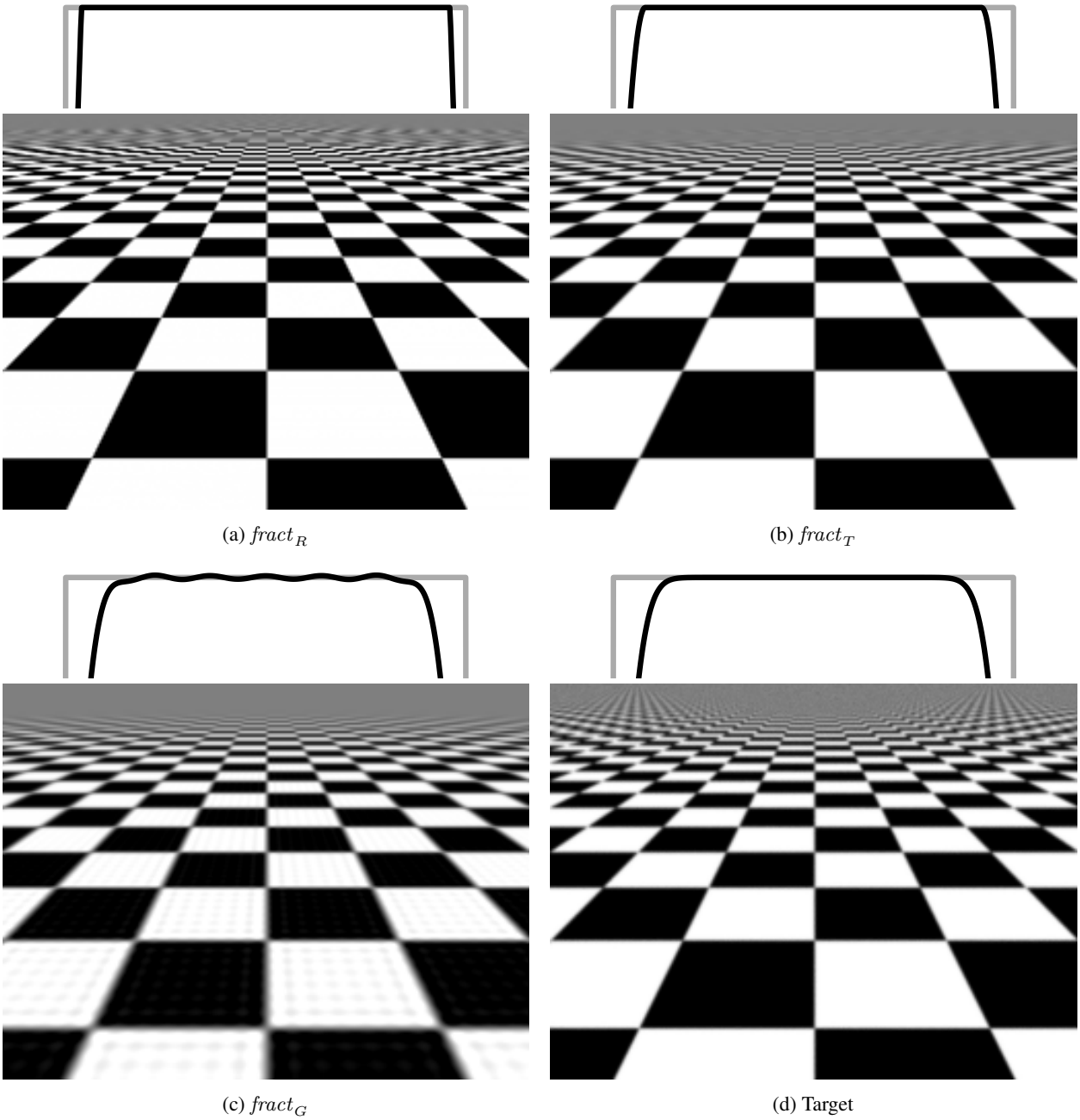
3.4 Search and Fitness

We have introduced a transformation that band-limits shaders under the assumption that they are partially multiplicatively separable and use limited function composition, employing a practical approximation that ensures the band-limiting kernel is separable. We now turn to the problem of shader functions that are not separable. In this section we consider an automated search strategy for approximating band-limited shaders in such situations. We motivate our approach with observations on two simple shaders: an alternate formulation of the checkerboard and a field of tiled circles.

Consider the checkerboard shader in listing 3.4. It produces an image (figure 3.6) that is essentially identical to that (figure 3.5) produced by the shader in listing 3.2. Note that this alternate implementation employs function composition (lines 2 and 3 in listing 3.4), to which the techniques of the previous section do not directly apply. However, if we replace the calls to `fract` and `step` with the band-limited expressions in table 3.1, despite the function composition, we produce the reasonable image in figure 3.6(c). This supports our insight that, even though we do not have an exact solution for situations involving function composition, it is sensible to consider the effect of composing band-limited subexpressions.

```
1 float3 checker2(float2 p) {
2   float ss = step(fract(p.s) - 0.5);
3   float tt = step(fract(p.t) - 0.5);
4   return (float3)(ss*tt + (1-ss)*(1-tt));
5 }
```

Listing 3.4: A multiplicatively inseparable black-and-white checkerboard shader. Compare this program with the code in listing 3.2.

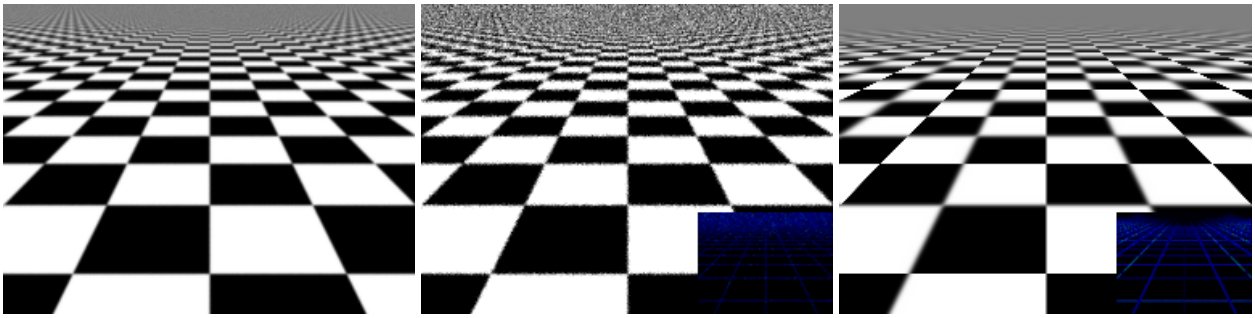


(a) $fract_R$ (b) $fract_T$

(c) $fract_G$ (d) Target

Figure 3.5: Renderings of a checkerboard using different band-limiting kernels. The target image (d) was rendered using the original shader with 2048 Gaussian-distributed samples per pixel. The remaining images were rendered using a single sample per pixel. The shapes of the corresponding approximations of a band-limited square pulse are given above each rendered image. The faint grid of gray dots and blurred edges of the foreground checkers in (c) is due to truncating the infinite series in the band-limited expression for $fract_G$ to achieve a run time similar to $fract_T$ (b). Truncating the series later reduces the visual effect at the cost of increased runtime.

Our second example to motivate the search, the circles shader in listing 3.5 (images in figure 3.7), is nearly as simple as the checkerboard shaders. It employs function composition of the same functions (`step` and `fract`) to the



(a) Target (2048 samples) (b) Original (16 samples) (c) Band-limited (1 sample)

Figure 3.6: Renderings with the multiplicatively inseparable checkerboard shader in listing 3.4. The band-limited image was generated using the $fract_T$ approximation to `fract`, given in table 3.1. The other two images were generated using a Gaussian distribution for consistency with the band-limiting kernel we use in this chapter. The false-color insets indicate the L^2 image error relative to the target image (a). Rendering (c) was an order of magnitude faster than rendering (b) with only 16% more visual error.

same depth. This time, however, as shown in figure 3.7(b), naively replacing each subexpression with band-limited expressions introduces unacceptable artifacts. The circles appear to be inscribed within a grid of lines and the distant portion of the image is black instead of a uniform gray. These artifacts are due to squaring the result of the band-limited `fract` and to applying `step` to the result of band-limited squares, respectively. Crucially, however, we observe that replacing only the call to `step` produces the more appealing image in figure 3.7(c).

```
1 float3 circles(float2 p) {
2   float ss = fract(p.s*10) - 0.5;
3   float tt = fract(p.t*10) - 0.5;
4   return (float3) step(0.2 - ss*ss - tt*tt);
5 }
```

Listing 3.5: A shader for an infinite grid of circles. Unlike the shader in listing 3.4, replacing each subexpression with its band-limited version produces an unacceptable result.

Taken together, the checkerboard and circles examples support our intuition that a band-limited shader can be approximated in the presence of function composition by substituting only a subset of the relevant functions. We exploit this observation in the following sections, using evolutionary algorithms (EAs) to identify subsets of expressions to band-limit such that their composition results in an approximately band-limited shader.

3.4.1 Exploring the Search Space

We define the search space with respect to the set of non-band-limited nodes in a shader program as follows. Each program in the search space is created by transforming a subset of such nodes as described in section 3.3.6. Thus, we

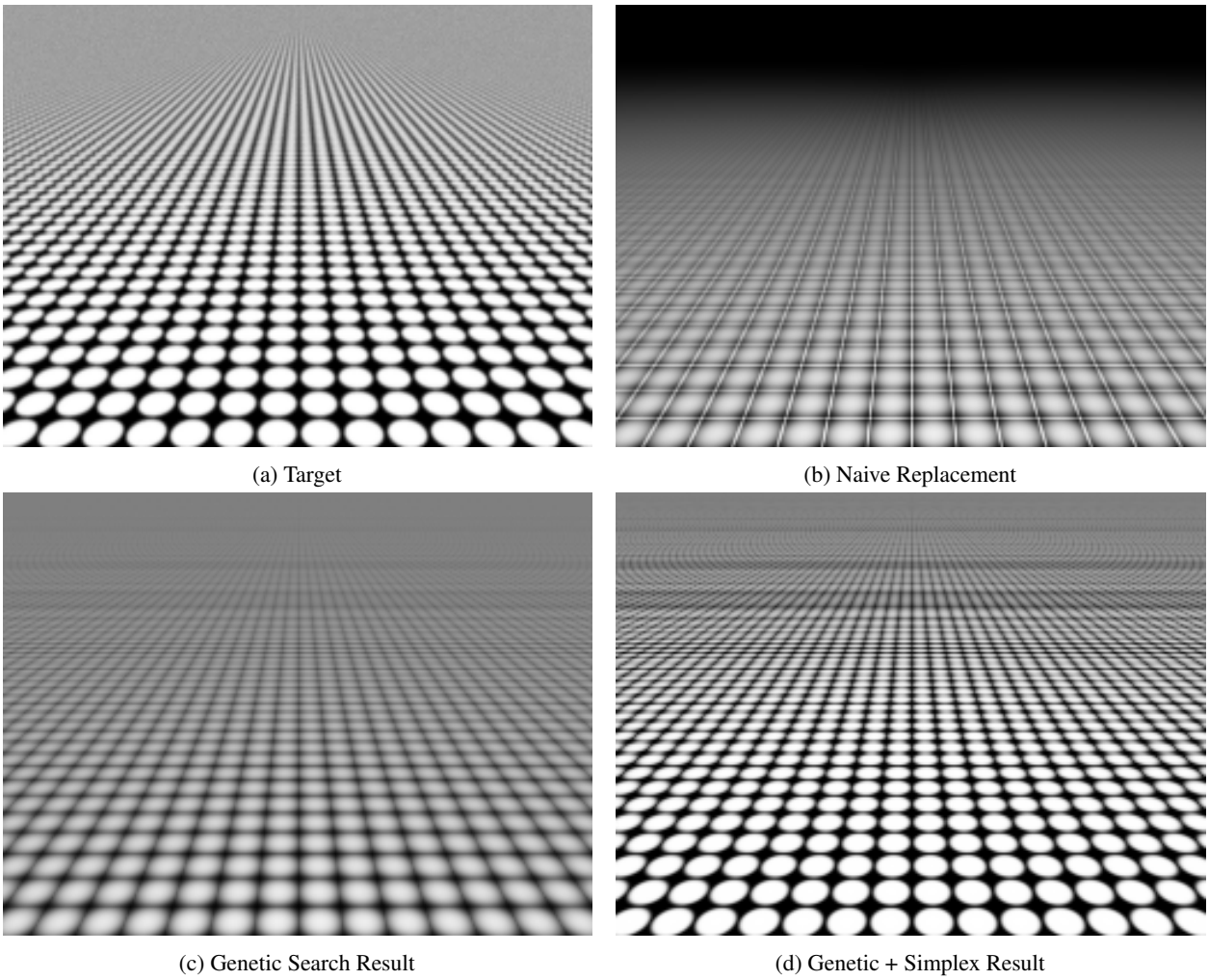
identify each program by the bit vector indicating which of its nodes are transformed [95]. Note that we only consider the non-band-limited nodes in the original program. By construction, any nodes inserted by a transformation are part of a band-limited expression that does not require further band-limiting. This implies that our search space is finite; once every node in the original shader has been transformed, no further transformations are possible. Thus, for very small shaders, it is possible to exhaustively evaluate every program in the search space. However, since the search space contains 2^N programs if the original shader has N non-band-limited nodes, it is not practical to explore the entire search space for most shaders.

As with the circles shader (listing 3.5), selecting different subsets of nodes to replace may result in different output quality. We therefore define the following measurement of the fitness of the new shader. For each original shader, we first construct target images using the same expensive supersampling techniques we used in figures 3.5(d), 3.6(a) and 3.7(a). During the search, we render the same scenes using the candidate band-limited shader and compute the differences between these and the target images. Algorithms to compute the difference between two images are well studied [118, 119, 120]; our fitness function is independent of the particular metric chosen. The fitness for the program is simply the sum of the differences for each pair of images.

We use a genetic algorithm (GA) (see section 2.2.3) to guide exploration of this search space. Although GAs have previously been shown to apply well to repairing faulty programs [121] and reducing shader run time [122, 123], to the best of our knowledge, such searches have not previously been applied to the problem of band-limiting shaders. The result of our search is the shader that produces images with the smallest measured error relative to the target images (for example, see figure 3.7(c)).

3.4.2 Adjusting the Sampling Interval

The transformations applied by the genetic search rely on the sampling intervals developed in section 3.3.5. However, these intervals are produced by an analysis that makes several simplifying approximations which may result in the estimated sampling interval being too large or too small. Once the genetic search is complete, we take the resulting shader and perform a second search to refine the sampling intervals by adjusting the coefficients of the polynomials. In this case, we use the Nelder-Mead simplex algorithm [92], a well-established non-linear multidimensional optimization algorithm, to learn new coefficients using the same fitness function we used during the genetic search. This algorithm produces a set of coefficients for the set of nodes that were band-limited in the genetic search result. Inserting these coefficients into the band-limited nodes of the shader produces our final result. For example, figure 3.7(d) shows the effects of the genetic and simplex searches in producing an approximately band-limited shader based on the circles in listing 3.5.



(a) Target

(b) Naive Replacement

(c) Genetic Search Result

(d) Genetic + Simplex Result

Figure 3.7: Renderings of circles (listing 3.5) showing the effect of using search to determine which expressions to replace and which coefficients to use to determine sampling interval. The target image (a) was rendered using the original shader with 2048 samples per pixel. Naïvely replacing every expression produces artifacts around each the circle (b). Using the genetic search (c) to determine which expressions to replace produces a result that, while still not perfect, more closely matches the desired image. Combining the results of the genetic and simplex search (d) produces an even closer approximation of the target.

3.5 Experiments

In this section, we describe our experimental setup, present our results, and discuss their implications.

3.5.1 Experimental Setup

We evaluate the effectiveness of our technique for band-limiting the shaders listed in table 3.2, drawn from the shaders used in previous work on antialiasing [124]. Several of these shaders (e.g., `brick` and `wood`) sample a random texture or employ a procedural noise function as a source of randomness. We treat these in the same way as any of

			L^2 error		Runtime (ms)		
Shader	Lines	Nodes	Ours	Supersampling	Ours	Supersampling	Description
`step`	1	1	**0.000**	0.001	**0.6**	17	Black and white plane
`ridges`	1	1	**0.011**	0.056	**1.3**	17	$fract(x)$
`pulse`	2	2	**0.021**	0.089	**2.2**	18	Black and white stripes
`noise1`	26	3	**0.042**	0.044	**39**	241	Super-imposed noise
`checker`	3	4	**0.072**	0.125	**4.1**	18	Checkerboard
`circles1`	3	5	**0.268**	0.308	**0.9**	18	Tiled circles
`wood`	51	18	0.141	**0.049**	**268**	1337	Wood grain
`brick`	40	26	**0.048**	0.118	**116**	683	Brick wall
`noise2`	66	28	0.222	**0.218**	**55**	319	Color mapped noise
`circles2`	71	74	0.157	**0.066**	**58**	1180	Overlapping circles
`perlin`	79	244	0.146[a]	**0.068**	**4.0**	71	Improved Perlin noise

[a] The best shader identified by our search produced the same amount of error as the original shader.

Table 3.2: Shaders used for evaluation. "Lines" indicates the number of non-comment, non-blank lines; "Nodes" lists the number of non-band-limited nodes that are candidates for replacement. "L^2 error" and "Runtime (ms)" indicate the performance of our antialiased shaders versus $16\times$ supersample antialiasing. The best error and runtime results for each shader are shown in bold. Our shaders are often an order of magnitude faster than $16\times$ supersampling while maintaining comparable or better image quality.

the functions in table 3.1, that is, as nodes that the search may or may not replace with a corresponding band-limited node. Specifically, our implementation uses summed area tables [108], which implement prefiltered texture images (see section 3.2.1), and Gabor noise [125], for which an anisotropic band-limited formulation is known.

Although our technique is independent of the exact scenes represented by the target images and the exact image difference metric used, we must choose some particular scene and metric to run our experiments. For these experiments, we chose a scene consisting of an infinite plane whose appearance is determined by the shader function. The target images render this scene, using 2048 Gaussian-distributed supersamples per pixel, from five different rotations (0, 30, 45, 60, and 90 degrees) around an axis perpendicular to the plane. To compute the difference between a candidate image and a target image, we calculate the total L^2 distance in RGB (i.e., $\sum_i \sqrt{\Delta r_i^2 + \Delta g_i^2 + \Delta b_i^2}$, where Δr_i indicates, for the i^{th} pixel in each image, the difference in the red component of the two colors). This is a common simple image difference metric used in previous work on shader transformations [126].

For each shader we ran 10 random restarts of the genetic search with a population of 200 candidate shaders for 20 generations. Thus, for roughly half of our shaders, our search exhaustively evaluated every variant shader that our technique generates, completing the search in under a minute. On the other hand, for the `brick` shader, the search evaluated less than one out of every 16 000 possible variants. For these shaders, the genetic search required roughly 8 hours for each random restart. We ran the simplex search using the same error metric and same set of representative scenes as the genetic search, with 100 random restarts.

3.5.2 Results

Table 3.2 reports the error and run time of the images produced by the best shader from the genetic search using the best coefficients from the simplex search. Note that the run times of our shaders are significantly better than 16× supersampling, which loosely corresponds to the state of the art, despite our search considering only image quality. Even though the band-limited subexpressions are typically more computationally-intensive than their non-band-limited counterparts, the benefit of taking a single sample per pixel outweighs the added complexity. The shaders produced by our technique are consistently faster than supersampling and in almost all cases have less error than the original shader. (We discuss the one shader that displayed the same error as the original below in section 3.5.3.) This implies that our shaders represent new points on their respective Pareto frontiers. In many cases (e.g., the `brick` shader), our shader Pareto dominates both the original shader and supersampling; that is, our shader produces images with less error and better run time.

We present example images generated using these shaders in figure 3.8. Each row of the figure contains the results for a different shader. In each row, the first column contains the target images for the shader, generated using supersampling. The second column contains the image produced by the original shader with no antialiasing, while the images in the third were approximately antialiased using 16× supersampling. The rightmost column the image produced by the approximately prefiltered shader generated by the genetic and simplex searches. The heatmap in the lower-right corner of each image displays the L^2 difference between that image and the target image, ranging from black indicating no difference through dark and light blue to green indicating significant difference.

Each image displays an infinite plane with uniform lighting. Since the plane is drawn in perspective, the sampling interval in the lower portion of each image is smaller than the interval in the upper portion. For example, in the second column (figure 3.8(i)) of the first row (figure 3.8(a)), the sampling interval at the bottom of the image is sufficiently small, allowing the checkerboard to appear with little noticeable aliasing (however, some aliasing appears as jagged edges to the checks and as blue lines in the heatmap inset). On the other hand, the sampling interval is much larger at the top of the image, resulting in significant aliasing, as indicated by the bright green region at the top of the heatmap inset. The third column (figure 3.8(j)) of the same row shows that 16× supersampling can reduce the jagged lines at edges in the foreground of the checkers image, but is insufficient to reduce the sampling interval in the background. In contrast, the prefiltered shader in the fourth column (figure 3.8(k)), produced by our technique, reduces the aliasing in both the foreground and background effectively. A similar pattern holds in the second row (figure 3.8(b)). As shown in table 3.2, our prefiltered `checker` and `brick` shaders produce images with roughly half as much error as 16× supersampling in roughly one-sixth the time.

In the next two rows (figures 3.8(c) and 3.8(d)) our prefiltered shaders generated images with about the same total error as 16× supersampling. In both cases, our prefiltering approach produced a shader that generates images with

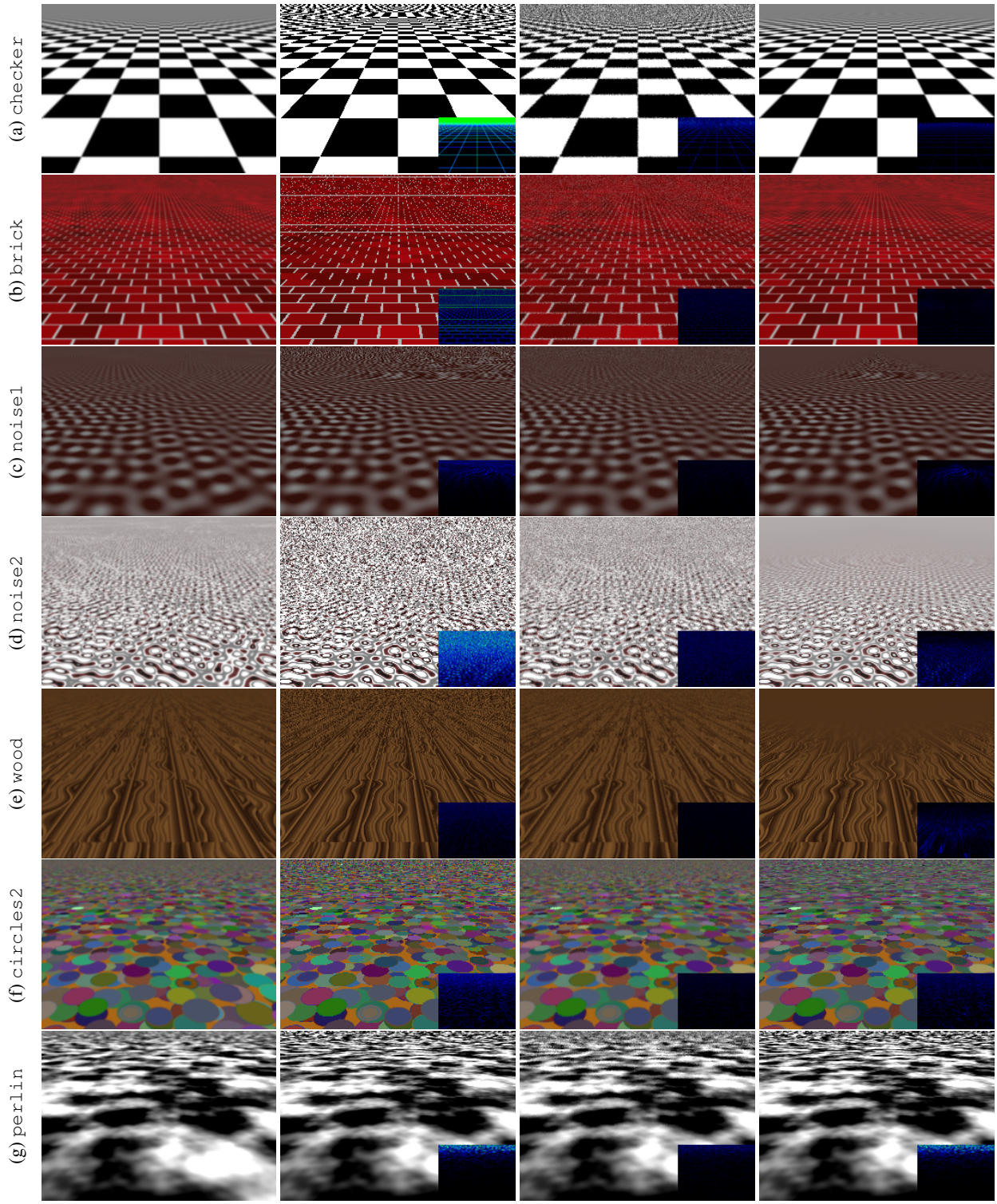


Figure 3.8: Comparison of images generated with different antialiasing techniques. From left to right, the techniques are (h) 2048× supersampling, (i) none, (j) 16× supersampling, and (k) our prefiltering approach.

reduced aliasing. However, the prefiltered `noise1` shader retains some aliasing in the middle distance, while the prefiltered `noise2` retains some slight aliasing in the foreground. The aliasing artifacts for both shaders are reflected in the blue regions of the heatmaps in the fourth column (figure 3.8(k)) of these rows. The heatmap for the `wood` shader (figure 3.8(e)) shows a similar pattern to that for the `noise2` shader. However, the largest source of error in this row is due to fact that band-limiting certain subexpressions resulted in slight shifts in the location of the wood grain in the foreground. Although the shifted grain still produces a woodlike appearance to human eyes, the simple L^2 error metric is unable to recognize the similarity. In the penultimate row (figure 3.8(f)), we see that our search failed to identify a shader that eliminated all of the aliasing in the far distance; nonetheless, our technique is able to mitigate much of the original aliasing. Finally, our search was unable to reduce the aliasing in the `perlin` shader used to generate last row (figure 3.8(g)). We discuss potential reasons for this limitation and possible improvements to address them in the next section.

3.5.3 Discussion

Our technique consistently produces shaders that are several times faster than $16\times$ supersampling. In many cases, these shaders also produce comparable or less error than supersampling. However, in some cases, such as the `wood` and `circles2` shaders in our evaluation, we found that although our technique was more accurate than using only a single sample per pixel, it did not outperform supersampling in terms of visual quality. Furthermore, for the `perlin` shader, our technique failed to achieve any significant improvement. This prompted us to analyze the `circles2` and `perlin` shaders to determine what aspects of those programs made the search more difficult. We summarize our findings and suggest directions for future research in this section.

Loops. Both shaders are structured around an outer loop. The `circles2` shader divides the texture space into cells and processes adjacent cells in a loop. The `perlin` shader accumulates several layers of noise, called octaves, in its outer loop, with an inner loop that sums the contributions from the vertices of the cell. The outer loop of the former shader and the inner loop of the latter have a number of iterations fixed by the algorithm and so may be easily unrolled completely. However, the number of octaves of Perlin noise to accumulate is often a user-controlled parameter. Similarly to Velázquez-Armendáriz et al. [127], we support loop unrolling for a predefined number of iterations and leave it to future work to address variable numbers of iterations. For our experiments, we fixed the number of octaves at four.

Significantly, the bounds of the loops we investigated do not depend on spatially varying parameters. However, we note that one common approach to manually band-limiting Perlin noise is to exclude higher-frequency octaves when the sampling interval grows sufficiently large. Previous work on automatic shader simplification has demonstrated the capability to unconditionally remove high-frequency octaves [122], but these techniques do not incorporate sampling

intervals in their transformations. This suggests a new band-limiting transformation to investigate, namely, introducing spatially-varying bounds to existing loops. Since arbitrary loops necessarily introduce high-frequency effects to represent the jump between an integer number of iterations, further research into band-limiting them is required.

Conditionals. Both the `circles2` and `perlin` shaders include if-statements, using them to determine the top-most circle and to identify the correct cell, respectively. As described in section 3.3.5, we always compute both branches and merge their values with a ϕ-function [113] based on $step$. Interestingly, the majority of nodes in `circles2` replaced by the genetic search were these $step$ nodes. This suggests future research could investigate the effects of different ϕ-function implementations on the success of the genetic search.

Lookup Tables. Perlin noise is often implemented using nested array accesses, where one array contains indices into a second array. Simply band-limiting the arrays in the way one might band-limit a texture [107, 108] would have the result of averaging the indices in the first array. This is unlikely to produce reasonable results. Recently, researchers have begun to investigate formal analysis and modeling of such nested array accesses [128]. Further research is required to discover automated transformations to handle these cases or to identify data structures and coding styles that handle them without requiring further transformation.

3.6 Related Work

In this section, we place the work described in this chapter in the context of the most relevant areas of related work.

Texture Prefiltering. Mipmapping [107] and summed area tables [108] are prefiltering strategies that produce lookup tables allowing a static texture to be band-limited in constant time. In contrast, our approach requires no additional memory tables and allows for dynamically adjusting shader parameters.

Norton and Rockwood [109] propose an approximation for shaders that can be decomposed as a sum of sines. They scale the amplitude of the sines based on the size of a pixel projected on the surface at that point using a power series approximation to a box kernel. Our approach is similar in spirit but more general, addressing a larger class of both shader and kernel functions.

Heitz et al. [129] describe a technique using precomputed lookup tables to band-limit 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.

Edge Antialiasing. Much recent effort has been directed at efficient algorithms for antialiasing edges in rendered images. Morphological Antialiasing [130] post-processes the rendered image to identify groups of pixels with a large

color gradient and particular spatial arrangement then blends their colors locally. The technique explicitly assumes that textures are separately band-limited using other techniques such as mipmapping.

Bala et al. [131] and Chajdas et al. [132] reduce the computational cost of supersample antialiasing by sampling shading at greater intervals than geometry and using interpolation. These techniques detect edges to avoid interpolating shading between unrelated regions. In the absence of edges, shading quality is predicated on shading having low frequency content relative to the geometry.

In contrast to these techniques, our approach addresses high-frequency and non-band-limited shaders.

Shader Simplification. Several researchers have investigated techniques for accelerating procedural shaders in contexts with reduced requirements for level-of-detail. Olano et al. [133] present a compiler technique for applying local transformations to a procedural shader, replacing memory accesses with constant colors. Pellacini's [134] 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. [122] employ a genetic algorithm that applies local syntactic simplifications to optimize the Pareto frontier between rendering time and image error. Wang et al. [123] also employ a genetic algorithm to search through code transformations to optimize a Pareto frontier over time, error, and memory consumption. Their transformations are informed by modeling the shader function as Bézier functions on the shaded surface.

Rather than addressing rendering time while tolerating a certain amount of infidelity in the resulting image, our approach explicitly addresses the visual property (greater sampling intervals) that enables previous techniques to tolerate error in shaders at lower levels of detail. We apply local transformations to the shader program, but produce a single shader with a dependent frequency spectrum.

Automatic Shader Bounds. Heidrich et al. [116] and Velázquez-Armendáriz et al. [127] 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 band-limit the output function instead of quantifying its bounds.

3.7 Conclusion and Future Work

This chapter explores the problem of automatically band-limiting procedural shaders as a special case of our optimization framework. To the best of our knowledge, this is the first project to address this problem. In general, this is a hard

problem involving finding a solution to the convolution of an input shader function and a band-limiting kernel parameterized by the sampling interval. We showed that in certain cases an analytic solution can be achieved and provide those results for a number of built-in functions that are common in modern procedural shader languages (table 3.1). We also demonstrated that exact solutions can be obtained for any linear combination or separable product of these functions. Building upon those insights, we developed a new approximation strategy for the many cases when an exact solution is not possible. Our approach integrates a meta-heuristic genetic search over possible subexpression replacements along with a non-linear simplex search over sampling interval parameters. We showed that in some cases this approximation strategy is able to find visually pleasing results (figure 3.8 and table 3.2) that require far less computational effort than supersampling. In a few cases, our search failed to find a satisfying result largely due to the difficulty of this search problem (section 3.5.3).

We see a number of interesting directions for future work. One idea is to study alternative methods of parameterizing the space of code transformations that are considered during the search. Our genetic algorithm considers the set of shader functions reachable by replacing a subset of the subexpressions with their band-limited counterparts. However, as noted in section 3.5.3, a common technique to band-limit certain shaders is to reduce the number of iterations of particular loops as the sampling interval increases. We also note that more terms of the infinite sum in the band-limited expression for $fract_G$ (see table 3.1) are needed when the sampling interval is small, but that the series may be truncated earlier when the sampling interval is larger. These examples suggest an explicit transformation that addresses loop bounds as a function of the sampling interval. Another possibility is to develop closed-form expressions for more complex expressions directly (i.e., expand table 3.1) instead of manually band-limiting exclusively one-dimensional functions. This would expand the set of subexpressions that can be exactly band-limited, potentially including some non-separable functions. Perhaps an approach that considers more function transformations would lead to better results.

It would be interesting to study how the design of the language may assist with this challenging task. As was demonstrated by the `checker1` and `checker2` shaders, there are typically many different mathematical functions that produce the same visual effect. Furthermore, it is frequently the case that one mathematical expression is preferable in terms of its suitability for this type of analysis (e.g., `checker1` permits a better solution than `checker2` with our method). It would be interesting to develop languages or language constructs that force a developer to produce shaders that permit analytic band-limited versions.

Chapter 4

Output Accuracy and Energy Use

4.1 Introduction

THE use of data centers and warehouse-scale computers (WSCs) has expanded in recent years to support a growing spectrum of applications [43]. At these scales, energy consumption can have significant economic and environmental impact. Between 2000 and 2010, data center energy usage more than doubled, accounting for over 1% of global energy consumption [8] and is projected to cost American businesses $13 billion per year by 2020 [135]. To help mitigate the environmental impact of these computing services, in 2016, Google announced its commitment to purchase $2.5 billion of renewable energy as part of a long-term goal to use energy exclusively from renewable sources [136]. The mechanical and electrical systems (such as lighting, cooling, air circulation, and uninterruptible power supplies) required to support warehouse-scale computation can quadruple [43] the power required by the computation itself. Because the load on many of these support systems grows with the computational load, computational efficiency is a significant determinant of the economic and environmental costs of data centers. In this setting, even modest reductions in energy consumption or heat generation can, through the scale of deployment, produce significant savings.

One of the difficulties in managing energy usage is lack of visibility into how implementation decisions relate to energy use [137]. Indeed, one artifact of the large number of variables that influence energy consumption is the wide variety of techniques that have been proposed to manage it. Researchers have approached energy reduction from several perspectives, including hardware (e.g., providing for voltage scaling and resource hibernation [138]), scheduling (e.g., predicting the interaction between workloads running on shared systems [139]), compilation (e.g., using instruction scheduling [73] to lower the switching activity between consecutive instructions [140]), and the API (e.g., selecting low-energy API implementations [137]). These perspectives are largely complementary. For example, a program that

uses a low-energy API may be compiled to minimize switching activity, then scheduled at a reduced priority by the OS to minimize interference with other tasks, allowing the processor to use a lower frequency/voltage combination. In this chapter we use our optimization framework to develop an automated approach to reduce energy usage (addressing the implementation decision challenge with search-based techniques) at the software level. Our approach fits in after API and compiler techniques but before OS and hardware techniques are applied. This allows us to leverage the sophisticated optimizations of modern compilers and to avoid the costs of runtime monitoring and just-in-time compilation.

Most existing approaches to energy reduction aspire to be semantics-preserving, avoiding optimization opportunities that may alter the program's behavior. However, the growing field of approximate computing [46] exploits the observation that, for many WSC applications, some variation in the output is easily tolerated, whether due to human perceptual limitations or to a lack of a single well-defined correct result [47]. For example, this same observation underlies many "lossy" image, video, and audio compression algorithms, which may introduce some error in reconstructing the media in exchange for reduced storage and bandwidth requirements. Data center applications, such as video streaming, may benefit from the same exchange. As a second example, many shopping and streaming services provide lists of recommended items; the lists should predominantly contain items the user will like, but need not contain all such items [141]. Although software techniques have received some attention [48, 49, 52], much of the recent work on approximate computing has focused on hardware approaches [46, 47, 51, 53]. Our insight is to treat approximate computing as a multi-objective optimization problem to be solved with a genetic algorithm (GA), exploiting the fact that some transformations will modify the program output to generate a Pareto frontier that explores the tradeoff between output accuracy and energy use. In chapter 3, we showed that our optimization framework is well-suited to optimizing output quality; in this chapter, instead of merely mitigating output differences introduced by certain transformations, we exploit them to achieve larger energy optimizations. Specifically, we use a multi-objective search to determine a Pareto-optimal frontier of programs that maximize output quality while minimizing energy use.

Thus, in this chapter, we instantiate our optimization framework to implement a post-compiler approximate computing approach for energy optimization. To achieve this, we select a program representation and transformations to fit between compilation and execution in the software deployment process. We develop tools and techniques to measure the energy of a program as part of an efficient fitness function. The rest of this chapter is organized as follows. Section 4.2 provides background on data centers and energy measurement. Next, section 4.3 describes our proposed energy measurement system to address the problem of using physical energy measurement in a fitness function. We discuss the program representation and transformations in section 4.4. In section 4.5 we place our representation and fitness function into our multi-objective search algorithm. Section 4.6 describes our experimental setup and reports results. Finally, section 4.7 places our approach in the context of related work and section 4.8 concludes the chapter.

4.2 Background on Energy and Power

This chapter addresses the problem of energy usage in data centers and particularly in WSCs. In this context, a data center consists of a collection of computer systems (i.e., the integration of CPU, memory, local storage, etc.), the communication and storage systems (e.g., network switches, routers, network attached storage, etc.), and support infrastructure (e.g., cooling, power, lighting, etc.) in a single location. A warehouse-scale computer is a data center designed to flexibly run very large distributed applications or services [43]. Although WSCs provide a unified platform on which to run these applications, they need not (and often do not) present the user with the abstraction of a single very large computer.

Computer systems in a data center consume energy to drive electrical signals on the CPU, memory, motherboard, network cards and cabling, etc., as well as to move physical media such as hard drives [43]. Electrical resistance and the first law of thermodynamics mean that regulating and distributing electricity to the various components consumes additional energy, as do cooling systems to remove the consequent waste heat from the system. Finally, the data center as a whole uses energy for its own environmental systems (e.g., lighting and air conditioning) as well as to distribute electricity to the individual computers. The rate at which energy is consumed is called power [142]. The units for energy and power in the International System of Units (SI) are joules (J) and watts (W) or $\mathrm{J\,s^{-1}}$, respectively. (The kilowatt hour (kWh), used for household energy measurement in the US, is a unit of energy equivalent to 3.6 MJ.) For example, if a server consumes 200 W while rendering a frame of a movie, taking an hour to do so, the energy required to produce the frame is 720 kJ. Although the companies managing data centers—and their electricity providers—measure the aggregate energy consumed, to optimize energy consumption on the scale of an individual program requires different techniques.

4.2.1 Estimating System and Program Energy

There are three main ways to measure or estimate the energy consumption of a computer system: simulation, modeling, and physical measurement. If a simulator represents the hardware with sufficient detail, it can associate each simulated activity with an energy cost [143]. As the simulation executes, these costs may be accumulated and reported. For example, the gem5 simulator was implemented with this level of detail, enabling research into energy-efficient hardware implementations [144]. This has the advantage of providing very precise control over the set of events that are included in the energy measurement. However, simulating hardware at a very high level of detail requires a correspondingly large amount of time—as much as three to six orders of magnitude more time than running directly on hardware [145, 146]. Although several techniques have been proposed to reduce this cost, they cannot eliminate it completely and result in increased inaccuracy [146, 147].

A common alternative to simulation is the use of a mathematical model based on performance counters on real hardware. They work by combining the performance counts—such as number of instructions issued or number of cache misses—in a mathematical formula designed to approximate the energy used [148, 149]. Apple's Activity Monitor[1] provides a similar estimate of energy, but uses an abstract unit of impact instead of approximating joules. Since the relevant performance counters can often be associated with individual processes, these approaches enable per-process energy accounting [148]. However, they depend on coefficients that estimate the energy associated with each counted event; these coefficients must be learned for each platform [150]. In addition, Haraldsson and Woodward observe that models may not accurately capture all of the components (e.g., memory, disk drives, network cards, fans, etc.) affected by the software [151]. These models become increasingly inaccurate as the energy consumption of the un-modeled components grows.

The most direct way to measure energy is with a power meter (along with a timer). External energy meters work by measuring the electrical current transmitted to a system or component, either in parallel (e.g., by clamping a sensor to the wire) or in series (e.g., by plugging the wire into the sensor). Since electrical power is the product of the current and the voltage, these meters must also measure the voltage on the same circuit to compute the power. Accumulated power measurements over a known time interval indicate the amount of energy consumed. For example, most houses connected to the electrical grid have a meter outside that accumulates the kWh of energy transmitted to the house. Commercial examples of such meters for individual appliances include the Watts up? PRO or Kill A Watt ®. Recently, open-source projects, such as the emonTx [152] have published designs for building meters for household use. Alternatively, Intel chips starting with the Haswell microarchitecture include registers that accumulate the energy consumed by certain large chip components [153]. Although direct measurement techniques avoid the accuracy and calibration issues of mathematical models and the performance issues caused by simulation, they typically measure the energy of an entire multi-core CPU or even a whole system, complicating the task of associating energy with a particular program.

4.2.2 Profiling

Recall from section 2.1.2 that a profile is a mapping from program elements to dynamic metrics about those elements. For example, these metrics may indicate execution counts or energy consumed by that part of the program while executing indicative workloads. Many profiling techniques work by periodically sampling the desired metric and associating it with the currently-executing instruction. This has the advantage of reducing the overhead introduced by the act of recording the metric; such overheads could affect the recorded energy use, for example. As with any sampling-based approach, these techniques are subject to aliasing if the metric varies too rapidly relative to the sampling interval, as discussed in chapter 3. In these cases, the profile may be smoothed via convolution to distribute the recorded

[1] https://support.apple.com/en-us/HT201464#energy, accessed 06/2017.

metric over nearby instructions [154]. However, certain metrics such as execution counts are not affected by the recording overhead, and may therefore be collected exactly.

Since profile information is collected dynamically, it is highly sensitive to the workload being executed. Thus, the more similar the workload is to the program's actual use in a data center, the more accurate the profile will be. However, in many cases, the actual workloads are not known in advance, are too numerous, or are rendered impractical to execute due to the profiling overhead. In these cases, indicative workloads cause the program to approximate the desired dynamic behavior; careful selection of the indicative workload can improve the approximation and, by extension, the results of using the profile.

4.3 Energy Measurement

As discussed section 4.2, energy measurement may be approached in several ways, including simulation, energy models, and physical measurement. We reject simulation, which is often orders of magnitude slower than running directly on hardware [146], as too time-consuming for our fitness function. We conducted a preliminary investigation into using an energy model, but found that the potential for model inaccuracies can have significant negative impact. Specifically, using the model suggested by Schulte et al. [155], we identified a variant of `freqmine` with over 70% predicted energy improvement, but only 2% actual improvement. To avoid misleading the search to such a significant degree, we decided to use physical measurements in our fitness function. We discuss our choice of measurement device in section 4.3.1 and address our approach to mitigating noise and the problem of multiple energy consumers in section 4.3.2.

4.3.1 Measurement Apparatus

Practical fitness functions must be both fast and accurate. In a single search, the fitness function may be evaluated on tens of thousands of variants (cf. sections 3.5.1 and 4.6.1); if each evaluation requires just one second to complete, the fitness function could be solely responsible for tens of hours of search time. Longer fitness evaluations increase the search time proportionally. As discussed below (section 4.3.2), fitness evaluations of programs running on different cores may not be usefully comparable, preventing us from leveraging parallelism to accelerate the search. At the same time, an inaccurate fitness function can cause the search to spend time evaluating undesirable variants while ignoring desirable ones. Although genetic algorithms, such as we use in section 4.5, often perform well in the presence of noise [27, 156, 157], noise mitigation techniques, such as increasing population size and acquiring multiple fitness estimates, are often necessary [36]. Notably, both of these mitigation strategies increase further the number of fitness evaluations conducted during the search.

To serve as part of the fitness evaluation for our experiments, we require an apparatus capable of measuring whole-system energy of individual server systems without hardware modifications. This apparatus must also have

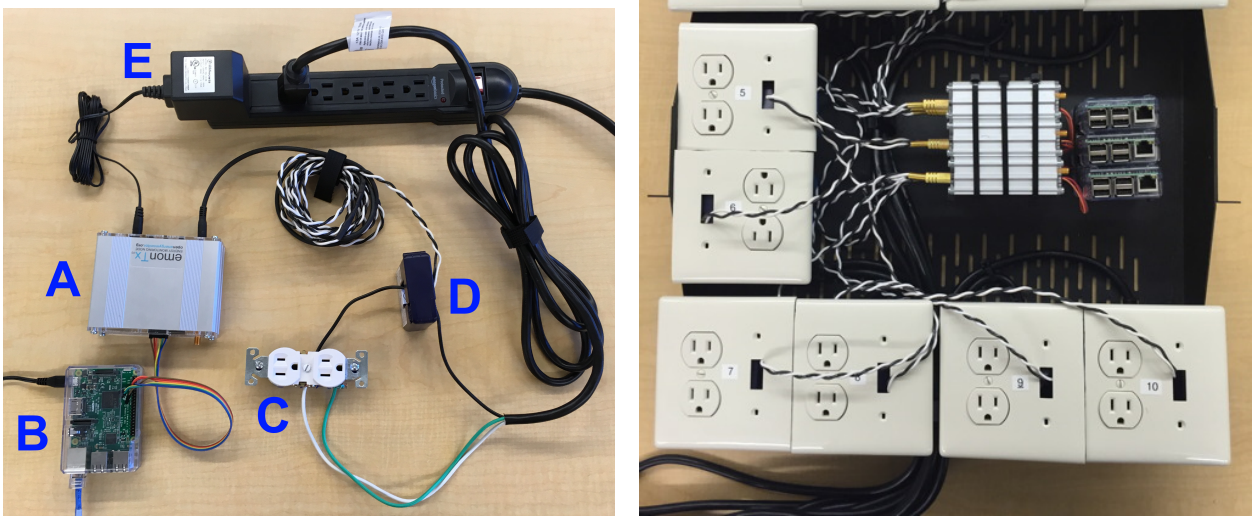
suitably fine-grained time and energy resolution. For example, with a device that reports energy consumption once a second, recording one extra reading after running our `freqmine` benchmark would increase the measured energy consumption by over 10%, potentially masking significant energy reductions. With a device that reports ten times a second, one extra reading would represent only a 1% increase. Additionally, to minimize noise due to overhead on the system under test, we require that the device be entirely self-contained without relying on monitoring software running on the system under test. As a practical matter, we also require it to be sufficiently cost effective to run several distinct experiments in parallel.

These constraints prevented us from using consumer-grade energy meters, such as the Watts up? PRO device.[2] These meters are typically designed for long-term monitoring and are not designed to capture rapid changes such as those caused by relatively short program executions. Although the Watts up? PRO meets several of our constraints, we found the response time too slow (1 Hz) for our experiments. In addition, while several solutions for measuring energy on mobile or embedded devices, such as LEAP [158], JetsonLeap [159], or the Monsoon Power Monitor[3] exist, these solutions are incompatible with the server class systems that run data center workloads. The lack of available energy measurement devices that met our requirements for server systems prompted us to construct our own energy meters.

Our final energy energy meter is shown in figure 4.1. We based our design on the emonTx V3 energy monitoring node [152]. This open source design consists of an ATmega328 microcontroller with sockets to connect an AC-AC voltage adapter and up to four current transformers. The microcontroller (**A** in figure 4.1(a)) is programmable using the Arduino API [160]. The current transformers (**D**) read the varying amperage on up to four separate lines (**C**) while the voltage adapter (**E**) reads the varying voltage from the same power source. We evaluated several different current transformers and chose Accu-CT ACT-0750 current transformers[4] rated for 5 A with the 1 V output option because our testing showed that these gave us the most precise measurements in range of powers used by our systems (i.e., 40 to 100 W).

Although this baseline hardware provides us with a cost-effective solution for high resolution time and energy measurements, we found that the default firmware needed to be completely rewritten to meet our time resolution requirements. Our software running on the microcontroller combines the signals from the current transformers with the voltage reading from the AC-AC voltage adapter to compute the real power on each line. This power is reported via a serial bus. Our present prototype implementation is capable of reading the inputs from the four current transformers and the voltage adapter at about 1200 Hz, which is significantly faster than can be transmitted via the serial controller. We therefore aggregate a configurable number of measurements together and report the average power usage less frequently. For all experiments in this chapter, the microcontroller reported measurements on the serial bus at 10 Hz. This is ten

[2] https://www.wattsupmeters.com/secure/products.php?pn=0, accessed 06/2017.
[3] https://www.msoon.com/LabEquipment/PowerMonitor/, accessed 06/2017.
[4] https://ctlsys.com/act-0750_series_split-core_current_transformers/, accessed 06/2017.



(a) Components of a single measurement circuit.

(b) Rack-mounted energy measurement circuits.

Figure 4.1: Custom energy meter setup. In (a), we show the components of a single measurement circuit outside of its enclosure. In (b), we show the final assembly of 10 circuits that was used to conduct the experiments in this chapter. The three emonTx V3 devices in the center are connected to three Raspberry Pis at the center-right. Each socket housing contains a components **C** and **D** from (a). Each emonTx measures the power usage of up to four connected sockets. The Raspberry Pis computes the energy usage for each channel of the connected emonTx and the readings available via TCP/IP over ethernet.

times faster than the sampling rate that was possible with the Watts up? PRO energy meters, and supports sampling energy consumption at a rate that makes large-scale searches feasible.

The code to convert the integral sensor readings into floating point current and voltage readings requires coefficients to scale the values properly. We calibrated these using a Watts up? PRO device as a baseline. Although the Watts up? PRO is not suitable for fitness evaluations, as a commercially calibrated meter, it is suitable for use as a baseline for calibration, which can tolerate slower responses. Note that properly calibrating real power measurements requires a resistive load such as a high-wattage light bulb or small heating element so that real power and apparent power are equal [161, § 8.4]. We chose to use a lamp with three 40-watt incandescent light bulbs as a load large enough that the limited power resolution of the Watts up? PRO provided four significant digits. After calibration we collected 2500 readings of the resistive load to confirm that the power readings from the microcontroller were reliable. We confirmed that they were approximately normally distributed (Shapiro-Wilk normality test, $p > 0.1$: large p-values fail to reject the null hypothesis that the distribution is normal [162]) and showed a small standard deviation relative to the average value (about 0.7%).

To support running experiments in parallel and measuring the energy usage of programs running on multiple

machines, we desired a solution that made the multiple separate energy measurements from a single microcontroller available via ethernet. This design allows the energy measurements to be dynamically distributed to different machines as necessary. We accomplished this by connecting the output of the microcontroller to the GPIO pins of a Raspberry Pi 3 [163] (**B** in figure 4.1(a)). We wrote a simple tool to read the power measurements from the GPIO pins, multiply by the time since the last reported measurement to obtain energy, and make the result available via TCP/IP. Note that the Raspberry Pi is simply a convenient system to distribute the energy readings. It is straightforward to connect the microcontroller to any system with a USB port using a UART to USB cable, allowing that system to distribute measurements or simply consume them directly.

As of 2017, the hardware required to monitor the energy consumption of a single machine costs $244. However, a single emonTx v3 node can simultaneously measure four different current transformers; the additional cost of measuring up to three more machines is only $47 per current transformer. Thus, the final hardware cost to monitor four machines is $385, just under $100 per machine. The completed apparatus, with capacity to measure the energy consumption of 10 machines (i.e., two emonTx devices measuring the power used by four machines each, with another measuring two machines), can be seen in figure 4.1(b).

This system provides fast, reliable measurement of a constant load, showing less than 1% deviation in the measurement of reference light bulbs. However, the energy usage of a computer system is much more complicated. In the next section, we discuss measures to mitigate measurement variation due to the server system itself.

4.3.2 Configuring the System to Minimize Noise

In addition to precise measurements of energy consumption from our meters, we also desire a low level of noise in the energy consumed by the systems running our search. That is, the system should add minimal variation to the measured energy; the less variation that must be ascribed to the system, the more certain we can be that any variation in energy level is due to the program being evaluated. We identify two broad categories of variation introduced by the system: energy that is unrelated to the program being evaluated (e.g., energy used by other processes running on the same system) and energy associated with the way in which the program is executed.

To reduce the variation in energy consumption due to different hardware platforms, we first ensured that all systems used for our experiments were purchased at the same time, to the same specifications, from a single distributor. We mitigated variation due to operating system activities by installing the Ubuntu Server 16.04.1 image (which is smaller than the desktop image) and only adding the minimum set of packages (mostly libraries) required to compile our benchmarks. Using a relatively small base distribution and adding few other packages reduced the chance that an unexpected `cron` job or daemon service would run during a fitness evaluation, which would add extra energy to the measurement. To limit race conditions with the upstream package maintainers distributing updates, we installed all

packages and software updates simultaneously to all machines. Next, we disabled dynamic frequency scaling governors by adding the argument `intel_pstate=disable` to the kernel boot arguments. Frequency scaling is used by the system to adjust power consumption; as with unexpected daemon processes, our search could misinterpret unexpected changes to the system frequency as an artifact of the program being evaluated. After the search, frequency scaling can be reenabled to take advantage of the additional power savings.

The second category of variation—variation associated with running the variant program—may be due to non-determinism in the behavior of the program itself or to non-determinism in which hardware resources are used to run the program. To mitigate the former, benchmarks were run in single-threaded mode wherever possible. With regard to the latter, manufacturing variations in the hardware itself can add surprisingly significant variations to the energy consumed while running a program. Over the course of preliminary runs, we detected unexpected variance that we were unable to attribute to our measurement equipment, the operating system, frequency scaling, or program behavior. Instead, we discovered that the differences were due to the CPU cores on which the program ran. In particular, the average energy consumed while running the benchmark on one core could be significantly higher than when running the same benchmark on another core. For example, on one of our systems we found about 0.4 J difference between running the benchmark on cores 5 and 6 (we restricted the OS to only schedule normal tasks to core 2 in both cases). Since the measurements on the same cores showed a standard deviation around 0.2 J, allowing programs to be scheduled to either core would represent a 100% increase in the measurement uncertainty.

To mitigate this source of uncertainty, we restricted the scheduler to always assign the fitness evaluation to one core on a given system and to assign all other processes to a second core. To determine which cores to use for each experimental system we measured the energy used while running a particular benchmark under each possible allocation of two cores. We then selected the combination with the smallest variance and used it for all experiments on that system. Although we carefully restrict the scheduler and select cores during the search, the subsequent evaluation demonstrates that the optimizations that we find do generalize to other cores and other machines with the same hardware.

4.3.3 Remaining Sources of Noise

The measures described above mitigate most of the noise we observe. However they do not eliminate noise completely. Some remaining potential sources of noise include environmental factors such as ambient temperature, the physical limitations of the measuring device, periodic system maintenance tasks scheduled by the Linux kernel, scheduling delays between opening the TCP/IP connection and starting the subprocess, and communication delays on the TCP/IP or serial communication channels. To address these and other, less predictable sources of noise, we follow the common procedure of averaging several energy measurements to compute the fitness [36].

4.3.4 Measuring Program Energy

This setup permits our fitness function to use the following procedure to measure the energy consumed by a process: on the machine that will run the process, (1) make a TCP/IP connection to the Raspberry Pi to receive continuous energy measurements, (2) launch the process and record the energy while waiting for it to complete, and (3) close the TCP/IP connection. Although it seems that the overhead from networking could impact our measurements, Langdon et al. have shown that this effect is negligible at the energy and time scales of our benchmarks [164].

The workload used when running the program during step (2) above is of critical importance. It must exercise the parts of the program to be optimized, and it should cause the program to run long enough that the performance may be measured accurately, but no longer. The first requirement stems from the fact that, unless the optimized part of the program is executed, there will be no way to measure the effect of the optimization. The second requirement is purely practical, since longer fitness evaluations result in longer search times.

4.4 Post-Compiler Representation

Having described our process for measuring the fitness of program variants, we now discuss how we generate those variants. The goal of providing post-compiler optimizations requires that we operate on programs produced by compilers. We choose to manipulate programs written in assembly language. Assembly language is commonly available as a compiler output format [29], permits optimizations in the spirit of instruction scheduling [73] and superoptimization [82], and has previously been successfully used as the basis for post-compiler optimizations [155].

Within this chapter, we consider an assembly language program to consist of a set of files, each of which contains an ordered list of lines. Each line is uniquely identified by a location, which we define to be a tuple containing the assembly file name and the number of the line within that file. Lines may be *instructions*, *directives*, *labels*, or *comments*. Instructions represent the operations to be performed by the CPU when the program is run; they translate more-or-less directly into the binary encoding of the operation in the executable program. Directives are commands to the program processing the assembly file. There are a number of directives for different purposes, including providing information about the program (e.g., debugging information), embedding literal data, and specifying instruction or data alignment. Labels identify the next instruction, for example, as the target of branch or jump instructions. The files comprising an assembly program are transmuted into an executable by *assembling* them—that is, converting them into object code—and *linking* the object code into the executable. We refer to this process as *building* the program.

Previous work applying search-based software engineering (SBSE) to assembly programs used a representation that very closely matches the assembly code itself [154]. However the sheer scale of many WSC applications—Kanev et al. reported in 2015 that binaries running in Google's data centers frequently exceeded 100 MB [165]—precludes

such a straightforward implementation. A population of hundreds of such programs would consume large amounts of memory unnecessarily, since many variants in the population would have very similar source code. Instead, we adopt a patch-based representation [166, 167] in which genomes consist of list of transformations that, when applied to the original program, produce the desired variant. Similar to previous work [154, 155], we implement three kinds of transformations to assembly programs: *delete* a line, *swap* two lines, and *copy* a line into a new location. The "target" of a transformation is the location of the line to be deleted, of either line to be swapped, or of the line after which a new line is to be copied. The "source" of the *copy* transformation is the line to be copied after the target line. Since the number of transformations applied to any variant during the search is small relative to the size of a program, this choice of representation results in significant memory savings during the search.

As the size of the program increases, so too does the time required to build and evaluate each variant, increasing the time to search a fixed number of variants or reducing the number of variants that may be searched in a fixed time. In the following sections, we develop a number of heuristics to avoid evaluating variants that have little chance of showing improved fitness. We divide these into heuristics that may be evaluated statically, without reference to the particular workload in the fitness function (section 4.4.1) and those that take the fitness function workload into account (section 4.4.2).

4.4.1 Static Heuristics

Given a sufficiently large supply of lines to copy, these transformations can produce any desired program by using the *copy* transformation to insert the lines of the desired program then using *delete* to remove any undesired lines of the original program. However, this requires a number of transformations on the order of the combined lengths of the original and desired programs. Even allowing for the smaller set of programs reachable via a much shorter sequence of transformations, the sizes of WSC programs results in an enormous number of programs that the search may potentially evaluate. Both the *copy* and *swap* transformations involve selecting two independent locations in the program (i.e., the locations of the two lines to swap or one location to copy from and another to copy to). However, many of these $\mathcal{O}(N^2)$ choices may be redundant, since certain instructions, such as those saving space for local variables at the beginning of a function, are duplicated exactly throughout the program. In fact, as many as 97% of the lines of larger assembly programs may be duplicates of the remaining 3% (see table 4.1). We therefore remove duplicate lines from consideration before selecting a line to copy, thus avoiding the cost of evaluating the fitness of the many identical programs that might otherwise be generated.[5]

[5]Related work in automated program repair [121] avoids duplicate fitness evaluations by caching fitness values, keyed by a hash of the program source code. Although we employ this optimization as well, the heuristics in this section are designed to avoid evaluating equivalent programs with *different* source code (cf. AE [168]).

In addition to redundant programs, naive application of our transformations may result in programs that cannot be successfully built, much less executed. To avoid the cost of generating and attempting to build such doomed variants, we introduce the following heuristics.

- When selecting target locations for the *copy* transformation, we exclude from consideration both assembly directives and labels that are not used as jump targets. For example, in listing 4.1 copying an instruction after line 1 has the same functional effect as copying it after line 2 or line 3. Our heuristic considers only the insertion after line 1.
- We exclude labels from consideration when selecting targets for the *delete* transformation and when selecting lines to insert using the *copy* transformation. Deleting a label that is referenced by some instruction or inserting a copy of label that already exists would prevent the program from building. Deleting or inserting a copy of a label that is never referenced would produce no change in the program's behavior, resulting in a redundant variant.
- When selecting lines for the *swap* transformation, we only allow labels to be swapped with lines in the same file. This is because swapping a label into a different file would produce the same effects discussed in the previous heuristic.

```
1  jle .L91
2  .p2align 4,,10
3  .p2align 3
4  movl  %r13d, (%r14)
```

Listing 4.1: Example of redundant insertion locations in sequence of assembly instructions. This code is taken from the `blackscholes` benchmark. Copying an instruction after line 1 has the same effect as copying it after line 2 or line 3.

4.4.2 Dynamic Heuristics

Just as inserting an unused label into a program will not alter the program's behavior, inserting an instruction that is not used (e.g., because the workload does not exercise the module containing the new instruction) is unlikely to affect energy usage. This leads to our insight that we can adapt the indicative workloads of profile-guided optimization (PGO), to target optimization efforts on a particular region. Previous work on program repair has used profile information for fault localization [121, 154], directing repair effort to regions associated with buggy behavior. We also use profiling, but instead use it to direct our search to transform regions associated with frequent execution, which are hopefully also associated with high energy use. Specifically, we profile the original program's behavior using the workload from our fitness function, collecting the instruction execution counts. We then adjust the probability that an instruction is the

target of a transformation in proportion to the execution count of that instruction in the profile. Since the profile describes the same behavior of the original program that we use to measure its fitness, we expect that regions of instructions with high counts in the profile are likely to magnify the effect of small improvements in energy consumption.

4.5 Search Algorithm

Our search operates on the compiled assembly program representation after compiler optimizations have been applied. We use this program to seed an initial population for a genetic algorithm as described in section 2.2.3. To evaluate the fitness of each variant in the population, we build the program and run it on the representative workload, measuring energy using the meter described in section 4.3. We then measure the difference in the program output, relative to the output of the original program run on the same workload. These two quantities—the program energy and the output error—constitute the objectives of our search. Note that while this error measurement is similar to a test case (it exercises the program on an input and compares the result to the desired output), in this case, the comparison to the desired output must result in a continuous estimate of quality rather than a binary *pass* or *fail*. We use a multi-objective search algorithm (NSGA-II [169]) to optimize both objectives simultaneously, allowing us to realize optimizations made possible by allowing slightly different output.

The final result of this search is a set of genomes representing the final Pareto frontier. Each genome in the set includes some number of transformations that, when applied to the original program, produce a variant with Pareto non-dominated fitness. However, due to the stochastic nature of the search, many of these transformations are quite often unnecessary to produce the desired fitness. We include a final post-processing phase to remove these superfluous transformations to reduce the chance that they alter the program's behavior on some untested corner case, and to simplify the task of understanding the identified optimizations. In the following sections, we describe the multi-objective genetic algorithm (section 4.5.1) and our post-processing algorithm for identifying and eliminating neutral mutations (section 4.5.2).

4.5.1 Multi-Objective Search

We initialize the GA population with the original program and $PopSize - 1$ mutants, where $PopSize$ indicates the desired number of individuals in the population. The NSGA-II algorithm is a generational genetic algorithm. In each generation, after evaluating the multi-dimensional fitness of each individual in the current population, the entire population is sorted using a non-dominated sorting algorithm [169]. This allows parents to be selected via tournament—two individuals are selected randomly with replacement, then the one with a better fitness according to the total ordering enforced by the sort is selected to win the tournament. Every two tournament winners undergo crossover, producing children which are mutated, and the mutants are set aside in a new population. Our algorithm is elitist: once the new population

contains *PopSize* variants, the two populations are concatenated, sorted again, and the *PopSize* variants with the best fitness according to the sort are retained in the next generation. This process continues until a predefined number of generations have occurred. We save the frontier of Pareto non-dominated variants from the final generation as the result of the search.

4.5.2 Edit Minimization

The genomes of the individuals on the final Pareto frontier generated by NSGA-II may include transformations that do not impact the energy use or measured error of the program on the workloads used by the fitness function. We therefore include a final minimization step to eliminate transformations that do not improve the fitness metrics.

The basis of our minimization algorithm is Delta Debugging [170], which takes as input a set of edits and identifies a 1-minimal subset of those edits that maintain the optimized performance as measured by the fitness function. The Delta Debugging algorithm runs in linear time and requires evaluating the fitness of a new collection of edits at each step. Due to the stochastic nature of energy measurements, we collect several samples of the fitness and apply a one-tailed Wilcoxon Rank-Sum Test [171] to determine whether the distribution of fitness values is worse than the distribution of values collected for the optimized variant. If the test for either objective indicates a difference between the distributions with $p < 0.05$, we treat that variant as "unoptimized." In all experiments described in this chapter, we collected at least 25 fitness samples for each Delta Debugging query, increasing this number as necessary so that the relative standard error was below 0.01. Increasing the number of samples increases the power of the statistical test, allowing it to distinguish the fitness of different genomes more effectively. We found that starting with 25 and using the relative standard error threshold provided a good tradeoff between runtime and smaller minimized genomes.

4.6 Evaluation

As discussed in section 4.5, the final result of our search and post-processing is a Pareto frontier of minimized genomes representing the best tradeoffs between output accuracy and energy use that the search discovered. This frontier allows the user to select which variant provides the most desirable tradeoff between these two properties. To evaluate the effectiveness of our technique at optimizing programs, we identify a maximum level of acceptable error for each benchmark and measure the largest energy reduction our search identified without producing more error. We also investigate the energy reduction achieved when no error is considered acceptable.

We introduce our benchmarks and experimental methodology in section 4.6.1. Then, in section 4.6.2, we discuss the results of our search at no error and human-acceptable error.

Benchmark		Assembly Lines	Unique Lines	Unique %	Executed Lines	Executed %	Tests (s)	Error Metric
`blackscholes`		12 437	3 504	28	637	5	2.7	RMSE
`bodytrack`		198 462	62 544	32	23 746	12	3.3	RMSE
`ferret`		80 811	26 883	33	15 181	19	6.4	Kendall's τ
`fluidanimate`		7 511	4 436	59	3 828	51	2.7	Hamming distance
`freqmine`		26 281	12 115	46	10 404	40	7.4	RMSE
`swaptions`		55 753	14 911	27	2 911	5	3.2	RMSE
`vips`		822 655	160 075	19	24 000	3	18.1	CIELAB distance
`x264`		205 801	58 754	29	41 063	20	5.7	CIELAB distance
`blender`	(car)	17 559 869	1 574 349	9	256 687	1	17.6	CIELAB distance
`blender`	(planet)	-	-	-	221 397	1	10.6	CIELAB distance
`libav`	(mpeg4)	22 831 124	698 445	3	42 747	0	1.3	CIELAB distance
`libav`	(prores)	-	-	-	34 634	0	2.7	CIELAB distance
Total lines		43 128 694	2 836 551		677 235			

Table 4.1: Data center benchmark applications. The benchmarks taken from the PARSEC suite are grouped at the top of the table. The `blender` and `libav` benchmarks each have two workloads; entries containing "-" indicate that the value is the same for the second workload. The columns indicate, from left to right, the name of the benchmark program, the number of lines in the compiled assembly, the number of those lines that are unique, the percentage of lines that are unique, the number of lines executed, the percentage of lines executed, the duration of the test workload, and the metric used to measure output error.

4.6.1 Benchmarks and Experimental Setup

Table 4.1 lists the benchmarks we use in our evaluation. We selected most of our benchmarks from the PARSEC benchmark suite [172] to allow direct comparisons with previous work [52, 155]. These benchmarks were designed to mimic the behavior of data center applications.

We also include two larger programs to investigate the capability of our implementation to scale to more realistic program sizes. First, we selected `blender`, a large 3D computer graphics application supporting a wide variety of tasks such as scene and character design as well as physics simulation and rendering. Second, we chose `libav`, a collection of audio and video processing libraries for manipulating and encoding multimedia. Both types of software are often used in large-scale server environments, and `blender` and `libav` are mature, production-scale programs that we believe represent realistic targets for optimization using our framework. The former project has been used for visual effects in the movie industry [173], while the latter is a fork of the `FFmpeg` encoder, which is used as a backend for video players (such as VLC [174]) and video streaming websites (including YouTube [3]). We note that `libav` and `blender` together are 28× the combined sizes of the PARSEC benchmarks. They allow for a more indicative assessment of our algorithm's ability to scale to larger datacenter-scale applications [165]. Unlike the PARSEC benchmarks, these programs incorporate a number of independent sets of features; we therefore include two different workloads for each to investigate optimizing separate features.

Each benchmark has an associated indicative workload used as a target for our optimizations. The PARSEC

benchmarks are distributed with several workloads; we used the same workloads as Schulte et al. [155] used for their experiments. For the `blender` workloads, we render scenes downloaded from the project's demo files web page.[6] We adapted the `libav` workloads from the project's test suite.

Profile Collection

As described in section 4.4.2, we use profiles of execution counts to guide our search toward transformations in regions that are frequently executed. Intuitively, these regions are more likely to have a significant impact on the energy use of the program as a whole. Recall (section 2.1.2) that a profile maps metrics, in this case execution counts, to program instructions. For the experiments in this chapter, we used Pin [175] to instrument basic blocks—sequences of assembly instructions with one entrance and one exit—and record the total number of times they were executed. Since executing any instruction in a basic block entails executing every instruction in that block, this gave us execution counts for every instruction. Table 4.1 shows the number of lines with non-zero execution counts in our profiles.

Error Metrics

Our multi-objective search algorithm considers both energy usage and output error. We describe our mechanism for measuring energy usage in section 4.3. However, the selection of error metrics that correlate with human perception of error inherently requires some amount of domain knowledge. The functions we use to compute this comparison depend on the benchmark, and are listed in table 4.1. We selected our error metrics based on each benchmark's documentation, targeting the primary user-visible output for benchmarks that produced more than one output.

To measure the output error for a variant, we compare its output to a reference output produced by the original program. Six of our benchmarks (two workloads apiece for `blender` and `libav`, plus `vips` and `x264`) use "CIELAB distance" as their error metric. All of these benchmarks produce images (or movies, which we treat as sequences of images) as their primary output. We compute the difference between the candidate and reference image as the total L^2 distance in the CIELAB color space [176] between pairs of pixels. (This is the same difference comparison presented in section 3.5.1, but here used with a perceptually-uniform color space.) The primary outputs of `blackscholes`, `bodytrack`, `freqmine`, and `swaptions` are vectors of numbers; for these benchmarks, we calculate the root-mean-square error (RMSE) relative to the reference vectors. The `ferret` benchmark computes a number of image similarity queries; for each query, the output consists of a list containing the names of images, ordered by similarity. For this benchmark's error metric, we use Kendall's τ, which quantifies the similarity of order between two sequences. Finally, `fluidanimate`'s output is a serialized C data structure; since this admits less human intuition about the meaning of "acceptable" levels of error, we simply compute the Hamming distance between the two files.

[6] https://www.blender.org/download/demo-files/, accessed 05/2015.

Human Acceptability

The error metrics we selected are intended to correlate with human perceptions of error. That is, the metrics should assign larger error to outputs that humans would rate as less accurate. However, the these metrics are only able to approximate human perception to a greater or lesser degree. Moreover, the degree of accuracy required may be different in different situations. Thus, it remains necessary for a human to make the final decision as to which variants on the Pareto frontier are acceptable.

In our experiments, we used the following protocol to determine the variants that demonstrated an acceptable level of error. We manually inspected the outputs of the minimized variants in the final Pareto frontier determined by the search (see section 4.5). For the benchmarks for which the primary outputs are images or movies (`blender`, `libav`, `vips`, and `x264`), we considered the output unacceptable if it showed any noticeable distortion on casual viewing. We performed a visual comparison for `bodytrack` as well, since its secondary output is an image in which a person's torso, limbs, and head are identified by boxes. We considered the `bodytrack` output unacceptable if the image contained the wrong number of boxes or if the boxes did not align with the person in the image. Both `blackscholes` and `swaptions` output lists of computed values for financial instruments; we considered these outputs acceptable if the calculated values were all within 5% of the reference values. We used the same threshold of acceptability for the list of frequency counts output by `freqmine`. We considered the lists of names of similar images produced by `ferret` to be acceptable if at least five image names in each list appeared in the corresponding list from the reference output. Finally, since the output `fluidanimate` is a serialized C data structure, admitting little human intuition, we only considered output that was identical to the reference to be acceptable.

These criteria for acceptability are necessarily somewhat subjective; other observers may find the 5% threshold too permissive for `swaptions`, for instance. Indeed, we consider the subjectivity of acceptability to be a strong argument in favor of producing a Pareto frontier instead of a single variant optimized under a predefined error threshold.

Comparison to Existing Techniques

We identified two previously published techniques to compare against. The first is GOA, another post-compiler optimization technique based on a genetic algorithm [155]. Their approach only addresses program energy usage, requiring that the optimized program produces identical output to the original, and does not incorporate our heuristics for guiding the search. The second is loop perforation, which transforms loops to skip iterations [52], for example, by only executing every other iteration. Unlike GOA, the motivation and evaluation of loop perforation includes consideration of the tradeoff between output accuracy and energy. Both of these techniques were originally evaluated on subsets of the PARSEC benchmark suite.

Experiments

We ran our technique on each benchmark using a *PopSize* of 512 individuals, the same size population Schulte et al. used for their single-objective GA. For all of the PARSEC benchmarks, we ran the search for 128 generations, resulting in 65 536 total fitness evaluations. In all cases except `vips`, this allowed the search to complete in under 72 hours, essentially "over the weekend." (The search for `vips` took longer than expected, either because we mistakenly used a different workload than Schulte et al. or because that workload takes significantly longer on our machines than the ones they used.) Note that, in addition to only considering the single-objective search, Schulte et al. reported results after 262 144 fitness evaluations, a significantly longer search than we conducted. With respect to identifying energy optimizations that produce no error, both of these differences should tend to favor their results relative to ours, since their search had a larger budget with which to evaluate variants to find optimizations and was not directed to expend part of that budget exploring optimizations that produced output containing error. Since the workloads for `blender` take much longer to process than most of our workloads, we only ran 64 generations, so that those searches completed in a week. Although the nominal workload durations for `libav` are in line with the PARSEC workloads, we discovered that the search took significantly longer than we expected; we were forced to terminate these experiments after a month of run time. We only consider the results after 32 generations for the mpeg4 workload and after 64 generations for the prores workload.

To facilitate evaluating the effectiveness of our search heuristics (sections 4.4.1 and 4.4.2), we conduct experiments using no heuristics, only the static heuristics, only the dynamic heuristics, and both heuristics together.

4.6.2 Results

Table 4.2 summarizes the results of our experiments. Based on these results, we can group the benchmarks into three overlapping categories. The first category contains `blackscholes`, `swaptions`, and `vips`, benchmarks on which we find significant energy reductions with no error, whether we employ our heuristics or not. We note that these are the same benchmarks on which Schulte et al. found their most significant improvements and that their results are consistent with the magnitude of improvements that we found, despite their longer search and focus on optimizations that produce identical output.

The second category consists of those benchmarks (`bodytrack`, `ferret`, `swaptions`, `vips`, `x264`, `blender`, and `libav`) for which we found significant energy improvements while maintaining an acceptable level of error. Figure 4.2 demonstrates this tradeoff for the `blender` benchmark's car workload. The figure shows energy reduction along the X axis, with error along the Y axis. The points in the figure represent the energy reduction and output error observed with the programs along the Pareto frontier, with minimum error at the bottom-left and maximum energy reduction at the top-right. The program represented by the lower-left point produces the same image as the original

	% Energy Reduction			
	No Heuristics		Best Heuristics	
Benchmark	No Error	Acceptable Error	No Error	Acceptable Error
`blackscholes`	91	91	**92**	**92**
`bodytrack`	0	0	0	**59**
`ferret`	0	**30**	0	**30**
`fluidanimate`	0	0	0	0
`freqmine`	0	0	**8**	**8**
`swaptions`	39	**68**	39	**68**
`vips`	21	**29**	21	**29**
`x264`	0	**65**	0	**65**
`blender` (car)	0	0	1	**10**
`blender` (planet)	0	0	0	0
`libav` (mpeg4)	0	0	0	**36**
`libav` (prores)	3	3	3	**92**
Average	13	24	14	**41**

Table 4.2: Energy reductions found by our technique. We present the largest energy reductions found while maintaining identical output to the original program ("No Error") and while maintaining an acceptable level of error, as described in section 4.6.1 ("Acceptable Error"). No amount of error was deemed acceptable for `fluidanimate`. We include the "No Heuristics" columns to highlight the effectiveness of our heuristics at directing the search toward profitable transformations.

program (indicated with "No error") using 1% less energy. The image meeting our definition of acceptability (indicated with "Human acceptable") was produced by the third program in the frontier, moving from left to right, using 10% less energy than the original. Although this represents the acceptable level of error, based on the guidelines we established above, different people or different situations may admit more or less error. For example, some users may find the images on either side of the one we selected to be the most acceptable in certain situations, leading to either 6% or 15% energy reduction instead. Recall from chapter 1 that we desire exactly this capability: from a single initial implementation, we automatically generated a range of implementations with different non-functional properties from which the user can select the best tradeoff for their situation.

The third category, containing `bodytrack`, `freqmine`, `blender`, and `libav`, shows significant improvement in the energy reductions when our heuristics are employed in the search. Figure 4.3 shows the Pareto frontiers our search discovered for the `libav` benchmark running the prores workload. Without applying any of our heuristics (figure 4.3(a)), our search was unable to find any useful tradeoffs. However, using the static heuristics (figure 4.3(b)), dynamic heuristics (figure 4.3(c)), or both (figure 4.3(d)), it discovered a number of programs generating different balances of energy usage and error. In particular, the dynamic heuristics enabled the search to find programs that use significantly less energy while producing very little error. Note, however, that the dynamic heuristics are not strictly better than the static; for example, the search found the largest energy reductions for `freqmine` using the static heuristics.

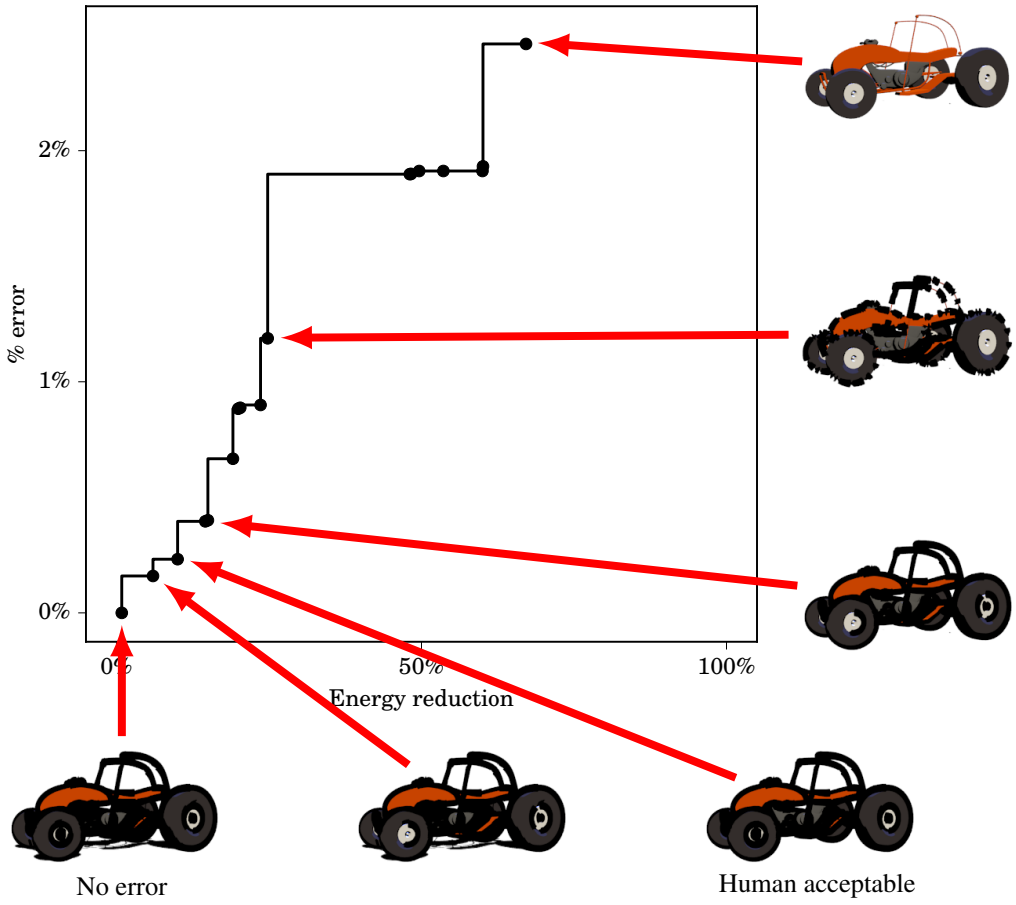
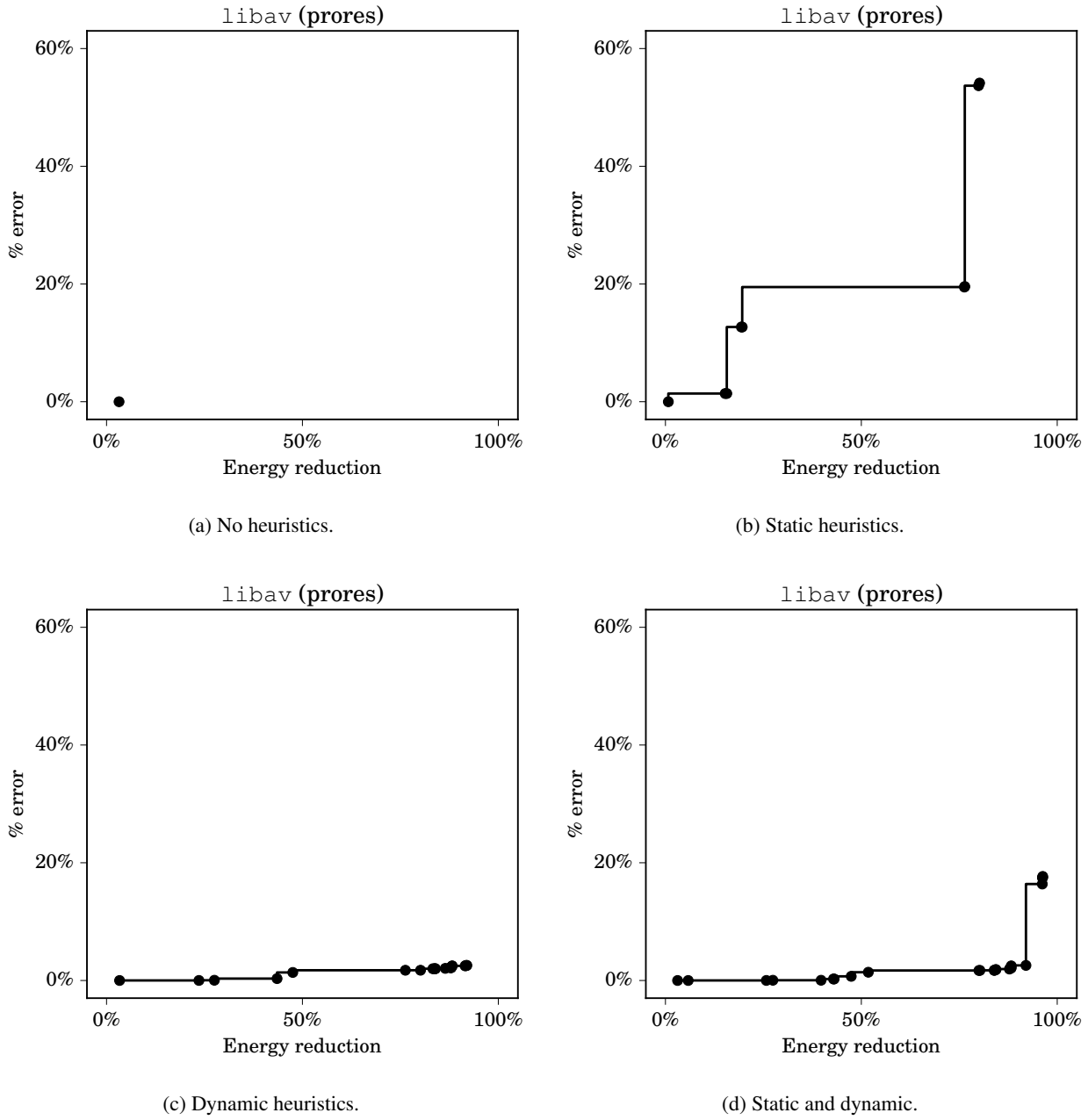


Figure 4.2: Annotated Pareto frontier for the `blender` benchmark running the car workload. The X and Y axes show energy reduction and output error, respectively. The points in the graph represent the programs along the frontier; the images produced by selected programs are shown around the edge of the figure.

As previously noted, our technique found energy improvments with identical output comparable to those seen by Schulte at al., and, when allowing some differences in the output, found signficantly larger improvements. We also compared our technique to loop perforation, which, like our approach, considers both energy and output accuracy objectives. We present the Pareto frontiers generated by our technique and loop perforation in figure 4.4; as with all frontiers presented in this chapter, the combination of large energy reduction and low error appears in the lower-right corner. As can be seen in the figure, the frontiers identified by our technique for `blackscholes`, `swaptions`, and `x264` dominate or coincide with the frontiers generated with loop perforation. However, the frontiers for `bodytrack` and `ferret` cross each other; some of the points on the loop perforation frontier dominate some of the points on our frontier. In such cases, more knowledge of the user's preferences is necessary to determine whether the preferred tradeoff was generated by one technique or the other. In the absense of such knowledge, the hypervolume indicator (I_H) is often used to evaluate the quality of a set of non-dominated points [37]. The hypervolume indicator of a Pareto



(a) No heuristics.

(b) Static heuristics.

(c) Dynamic heuristics.

(d) Static and dynamic.

Figure 4.3: Comparison of search results using different heuristics. All results are for the `libav` benchmark running the prores workload. The Pareto frontier for the control (no heuristics) in (a) contains just one point.

frontier is defined as the measure (that is, the area in our two-dimensional setting) of the points dominated by frontier. Computing this for `ferret` shows that for our technique, $I_H = 86\%$, while for loop perforation, $I_H = 70\%$. That is, our technique found optimizations that dominate more of the design space than were identified by loop perforation.

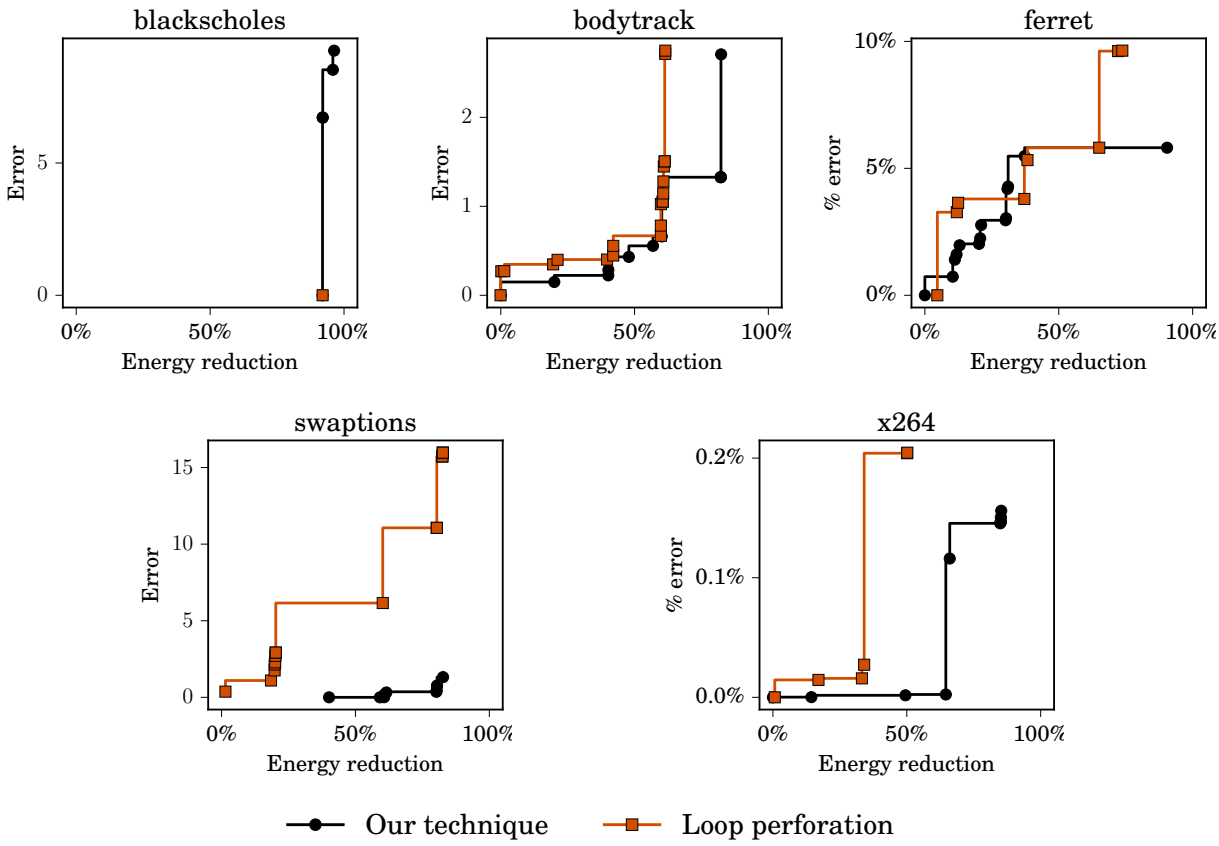


Figure 4.4: Comparison of our technique to loop perforation. The Y axes for `ferret` and `x264` are in percentages, since the maximum possible error is well defined. Since the other benchmarks use the RMSE metric, the maximum error is in principle infinite and scaling to percentages is not possible. The Pareto frontiers our technique produced for `blackscholes`, `swaptions`, and `x264` dominate the frontiers produced by loop perforation. The situation for `bodytack` and `ferret` is more nuanced.

4.7 Related Work

In this section, we discuss the broader context of energy optimization research as it relates to the work described in this chapter.

Semantics-Preserving Techniques. As discussed in section 2.1, semantics preserving optimizations have been well studied.

Profile-guided optimization uses profiles of program behavior to guide decisions about which optimizations to apply [86]. This allows the realization of optimization opportunities that cannot be determined statically, particularly optimizations that may improve performance under some conditions while reducing performance under others [85, 87]. Using profiles, these tradeoffs can be applied to improve the performance under the usual operating conditions of the software. At a high level, we also use profiles to identify the portions of a program where optimizations may have the

most benefit. However, unlike most profile-guided optimizations, we do not determine in advance what transformations to apply to sufficiently frequently executed code.

Superoptimization techniques [82, 84] attempt to search for the optimal sequence of machine instructions to implement desired functionality. Due to the exponential number of sequences containing a given number of instructions, these techniques are limited to very short sequences. Additionally, they are frequently limited to implementing loop-free code to simplify the task of determining whether a candidate sequence matches the desired functionality. In contrast, our approach operates on whole programs and relies on tests to validate functionality.

A number of approaches, such as voltage scaling [138] and clock gating [177], attempt to reduce the energy consumption of hardware by placing components or even an entire chip into a "low power" mode. These techniques do not modify the functionality of programs running on the hardware although they may introduce delays due to lower frequencies. These approaches are essentially orthogonal to our approach, allowing the benefits of both to be realized.

Approximate Computing. A number of researchers have introduced approximate computing techniques in hardware as well [45]. For example, simpler adder or multiplier circuits may be designed that consume significantly less energy while computing a similar function to a true adder or multiplier [46, 178, 179]. Other circuits may not meet timing constraints under all circumstances, leading to imprecision [180]. As with the semantics-preserving hardware techniques, these techniques are largely complementary to our approach. However, the approximations from these techniques may interact with those introduced by ours; maintaining the desired level of output quality would probably require evaluating the fitness function approximate hardware directly.

Precision scaling [181, 182] improves efficiency by altering arithmetic precision. For example, rounding values and using fewer bits to represent data can use less hardware and change memory layout, potentially improving cache performance.

Task skipping [183] and loop perforation [52] are software-level techniques that trade accuracy for energy by skipping computation. Although task skipping requires the developer to annotate tasks so that a runtime can determine which tasks to run, loop perforation is a fully automated technique. Our approach is more general than loop perforation in that our transformations may result in skipping loop iterations, but have the potential to produce other optimizations as well.

Genetic Algorithms for Power Improvement. Researchers have recently begun investigating the potential of GAs and related methods for reducing software energy consumption. For example, Linares-Vásquez et al. [184] use a GA to identify low-energy color palettes for mobile applications. Schulte et al. [155] and Bruce et al. [185] use GAs to optimize the energy use of desktop and data center applications. Although the first project modifies the program output slightly to reduce energy use, it relies on the behavior of a specific type of display. The latter two projects apply more

general transformations, but do not exploit the possibility of approximate computation. Our approach combines general transformations with approximate correctness to achieve larger improvements.

4.8 Conclusion

Data center scale computation accounts for a significant fraction of energy consumption and has a growing economic impact on business. Although advances in hardware and compilers partially address this problem, software perspectives on energy reduction are relatively unexplored. Advances in search-based software engineering have shown that automated program optimization techniques can successfully be applied to the domain of energy reduction, but current techniques do not scale and can only improve modeled (as opposed to measured) energy.

We combine off-the-shelf components and specialized firmware with insights from search-based software engineering and profile-guided optimization to develop a software-level energy-reduction approach that scales to much larger applications than were previously possible. We present inexpensive hardware and system configurations that allow the rapid sampling of real-world energy consumption capable of directing an evolutionary search when combined with a modified genetic algorithm. We also present large search space reductions and use precise instruction-level profiling to direct the search in order to find optimizations in programs with over 20 million lines of assembly. Our technique finds optimizations that reduce the energy consumption in half of our benchmarks without affecting the program output, and achieve 41% on average with human-acceptable levels of error.

Chapter 5

Readability and Test Coverage

5.1 Introduction

One of the most common ways of checking a program for faults is testing [66]. Although it is well known that thorough and early testing is more effective than testing scantily or late, the cost of developing and maintaining tests often discourages humans from writing them. This has led to significant research into automatic techniques for generating tests, particularly unit tests [66, 67, 68, 69, 186, 187, 188, 189]. These algorithms apply various techniques to automatically determine inputs to functions that cause them to execute different portions of their functionality. The resulting test cases, relying as they do on randomly generated or mathematically derived inputs, may be counter-intuitive to many developers [190]. Indeed, Beller et al. found that, when they interact with testing code, developers appear to spend a larger proportion of their time reading than they do when interacting with the product code (i.e., the source code of the program itself) [191]. When an automatically-generated test fails, the developer must read the test, which they did not write and may not have previously seen, to understand the source of the fault [70]. However, aside from a few heuristics [70, 190], little research has targeted the readability of the generated tests.

Readability, a judgment of how easy a program is to understand [192], has long been recognized as an important aspect of program quality [10, 58, 193, 194, 195]. Reading a program's source code is a primary way that programmers understand it, a necessary step in almost any maintenance activity [61]. Indeed, it has been estimated that when programmers interact with source code, they spend anywhere from two-thirds [191] to nine-tenths [55] of their time reading it—and, as mentioned above, this fraction is larger for test code than product code. Since US companies paid over $100B in software developer salaries in 2016 [16], the time and effort spent reading and understanding programs has significant economic impact.

In this chapter, we instantiate our optimization framework to optimize the readability of automatically generated

test cases. To accomplish this, we develop a function to automatically estimate the readability of unit tests to guide our search. The remainder of this chapter is organized as follows. First, we discuss some background relating to unit test generation and the framework we use to develop our readability estimator in section 5.2. Then, we describe our estimator in detail in section 5.3. In section 5.4, we present the representation and transformation we use to modify unit test readability, along with our choice of search algorithm. Section 5.5 presents our experimental evaluation of our approach. Finally, we place our approach in the context of related work in section 5.6 and section 5.7 concludes the chapter.

5.2 Readability Metrics and Testing Background

5.2.1 Machine Learning and Regression

The field of machine learning (ML) covers a wide variety of problems, approaches, and algorithms [196, 197]. In this chapter, we focus on regression models, which represent the desired quantity (readability) as a mathematical expression in terms of measured values (known as features or attributes) associated with instances of source code. In particular, we discuss simple linear and logistic regression models. Both of these types of models are trained off-line; that is, the expressions are determined during a distinct learning phase, after which they may be used without modification. In both cases, learning is supervised in the sense that it requires knowledge of the desired values (called labels) of the expression for each of the instances used during training. Simple linear regression models predict numeric values using the linear expression, $w_0 + w_1 a_1 + \cdots + w_n a_n$. Logistic regression models compute the probability that an instance falls in one of two classes using the expression, $(1 + e^{w_0 + w_1 a_1 + \cdots + w_n a_n})^{-1}$. In both linear and logistic expressions, the a_i represent feature values related to a particular instance, while the w_i are coefficients particular to the metric[1] and used for all instances. Training uses a set of instances (the "training set") to determine the coefficients that minimize the sum of the squared error between the predicted and desired outcome (linear regression) or maximize the likelihood of the desired classification (logistic regression).

As with any ML algorithm, there is a chance that these regression metrics may compute incorrect values for instances that are not part of the training set. In this case, the metric is said to fail to generalize. One common approach to check how well a metric may be expected to generalize is n-fold cross-validation [196], which splits the training set into n partitions (or *folds*), then trains a metric using $n - 1$ partitions and evaluates its performance on the remaining partition. So that each partition is used for evaluation exactly once, this is repeated n times and the resulting performances are averaged. Note that the cross-validation performance is *not* the performance of any single metric; in particular, it is not

[1] Throughout this chapter, we use *model* to refer to the choice of expression (linear or logistic) used to represent readability, including the choice of features but not coefficients. We use *metric* to refer to a model with a particular choice of coefficient values; that is, a metric is a function that computes readability given a set of features. One model may produce an infinite number of different metrics by selecting different coefficients during training.

the performance of a metric trained using the entire training set. However, the average cross-validation performance provides an estimate of how such a metric will perform on future instances (assuming, of course that the training set is representative of the future instances).

5.2.2 Readability

The fact that the notation in which a program is written affects the ease with which programmers write and maintain it has been recognized since the earliest days of programmable computers [198, 199]. This intuition led early language designers to develop conventions based on existing mathematical notation [71, 200]. However, some programs remain easier to write and maintain than others written in the same language. Conceptually, we may attribute these differences to variations in the underlying problems being solved, in the structures of the implemented solutions, and in the typographic presentation of the program [194]. The Halstead [201] and cyclomatic [202] complexity metrics are concerned primarily with the first and second categories. In this chapter, we hold the problem (testing) and solution structure (unit tests) constant and focus on the third category, which we call simply readability.

Researchers have proposed many different features of the presentation of programs that might affect their readability. Use of identifier names [61], comments [203], and indentation [204] are some of the most popular. Buse and Weimer incorporated these and other presentation features into a logistic regression model to classify code as readable or unreadable [192]. They trained a metric based on their model using the results of a human study of upper-level students at their university. However, although cross-validation did not indicate a lack of generality, Posnett et al. showed that the Buse and Weimer metric increasingly tends to classify code as unreadable as the code features become less similar to their training examples, proposing their own model and metric to mitigate this limitation [205]. Posnett's metric generalizes somewhat more than Buse's, although since both were trained using the same data, both display their best performance on short samples taken from open-source product code.

We desire a metric to estimate the readability of automatically generated test suites, which we discuss in the next section. We will return to the issue of readability metrics in the context of test suites in section 5.3.

5.2.3 Testing and Coverage

As described in section 2.3, software testing is the process of executing a program to check for differences between the program's required and demonstrated behavior [35] by running tests. Individual tests may be characterized by the requirements they cover [97]. For a collection of tests, called a test suite, we define coverage in terms of the fraction of requirements represented by the union of the requirements covered by the tests in the suite. However, given the difficulty of exhaustively verifying compliance with requirements as well as the reality of incomplete requirements specifications, coverage is often specified in terms of more easily measured quantities. For example, statement coverage measures the

number of program statements executed during the test, while branch coverage tracks which branches are taken, as measured by collecting a profile (see section 2.1.2) while running the tests. Alternatively, mutation testing [206] counts the number of mutants detected by the test suite. These metrics only approximate requirements coverage, since, for example, executing every statement or detecting every mutant may leave some requirements unevaluated; similarly some statements may be unreachable while some mutations leave functionality unaffected.

Test suites may be automatically generated by producing a number of inputs along with the necessary oracles. Techniques to produce inputs include random generation (fuzz testing) [189], solving symbolic path constraints [187, 207], and a combination of the two [66, 67, 208]. The associated oracles may implement simple heuristics (e.g., the program should not crash or leak memory) [189, 209], or be derived from current program behavior [210, 211, 212].

5.3 Measuring Test Case Readability

In this section, we develop an automated fitness function to estimate the readability of each test. Our approach is based on supervised machine learning techniques, in the spirit of the readability models developed by Buse and Weimer [192] and Posnett et al. [205]. As mentioned in section 5.2.2, it has been shown that these metrics do not generalize well to code that is too dissimilar from the training data they used [205]. Since our search is over automatically-generated test cases while the existing metrics were trained on product code from SourceForge[2] [192], we expect the existing metrics to provide poor indications of relative readability in our context. We therefore constructed our own metric to estimate relative readability.

We desire a readability metric that is capable of discriminating between two readable (or unreadable) instances to determine which is more (or less) readable. Thus, we reject the logistic regression models on which previous metrics were based, since logistic regression is not designed to distinguish between instances of the same class. Instead, we use a linear regression model which can treat readability as a continuous quantity. Linear regression models have the additional benefit (relative to other regression models, such as multilayer perceptrons [196]) of simplicity and of permitting direct interpretation of the coefficients.

In section 5.3.1 we collect unit tests and readability ratings to form the training set we used to learn coefficients for our metric. Then, in section 5.3.2, we describe our process for selecting the features of the model.

5.3.1 Training Data

To construct a set of unit tests with which to learn our metric coefficients, we desire examples of both developer-written and automatically-generated tests. Specifically, the developer-written examples should display combinations of features that humans consider readable while the automatically-generated tests represent the features produced by current

[2] https://sourceforge.net/

test-generation algorithms. By including both sets in our training data, we hope to bracket the search space of tests, improving the chance that our metric will generalize to new tests encountered during the search.

We collected developer-written training examples from eight open-source Java projects, including Apache Commons,[3] Apache POI,[4] GNU Trove,[5] JFreeChart,[6] Joda,[7] JDOM,[8] iText,[9] and Guava.[10] These projects incorporate extensive test suites that have been maintained over several years, arguing in favor of the acceptability of the tests they contain. We supplemented these tests with test suites generated by EvoSuite[11] [68] to maximize branch coverage, a common metric used in test generation research [186], for each project. From this pool of unit tests, we manually selected a diverse set of training examples.

To determine readability scores—the labels for our supervised learning algorithm—for these training examples, we conducted a set of IRB-approved (protocol #2012-0072-00) online human surveys. For each survey, we solicited participants to view and rate unit tests on Amazon Mechanical Turk,[12] a micro-task marketplace that has proven an effective resource for collecting large numbers of scientifically meaningful responses on tasks requiring human intelligence [213]. Before participating in our surveys, we required participants to complete a brief quiz to ensure that they were familiar enough with Java programming to provide a meaningful rating. The quiz requires potential participants to read and understand four Java methods; people answering correctly for three out of four methods were allowed to participate. Each participant who passed the quiz was then given a survey in one of two formats. In the first format, we asked participants to rate a random subset of training examples on a scale of 1 (low readability) to 5 (high readability). In the second, forced choice, format, we presented the participants with pairs of examples and asked them to select the member of each pair that was more readable. As in previous work [192], we did not define readability but instead asked participants to rate readability based on their own subjective impression.

We collected 15 669 readability scores from 231 participants on our training set of 450 unit tests using the first survey format and 4950 readability choices from 197 participants on 200 pairs of tests using the second survey format. Figure 5.1 shows the distribution of the average scores for each test in the set. The average readability score over all tests was 3.5, with two thirds falling in the range from 3.0–4.0. We note that this distribution of scores differs significantly from the bimodal distribution observed by Buse and Weimer [192]. This may be due to our use of different participants (their participants were primarily fourth-year students) or different types of code (complete unit tests as opposed to three simple statements take from product code [214]). We used the average readability scores (from the

[3] https://commons.apache.org/, accessed 06/2017.
[4] https://poi.apache.org/, accessed 06/2017.
[5] http://trove.starlight-systems.com/, accessed 06/2017.
[6] http://www.jfree.org/jfreechart/, accessed 06/2017.
[7] http://www.joda.org/, accessed 06/2017.
[8] http://www.jdom.org/, accessed 06/2017.
[9] https://github.com/itext/itextpdf, accessed 06/2017.
[10] https://github.com/google/guava, accessed 06/2017.
[11] This research is in collaboration with the authors of EvoSuite. Their expertise was instrumental in incorporating the techniques described in this chapter into their cutting-edge test generation framework.
[12] http://aws.amazon.com/mturk/, accessed 03/2015.

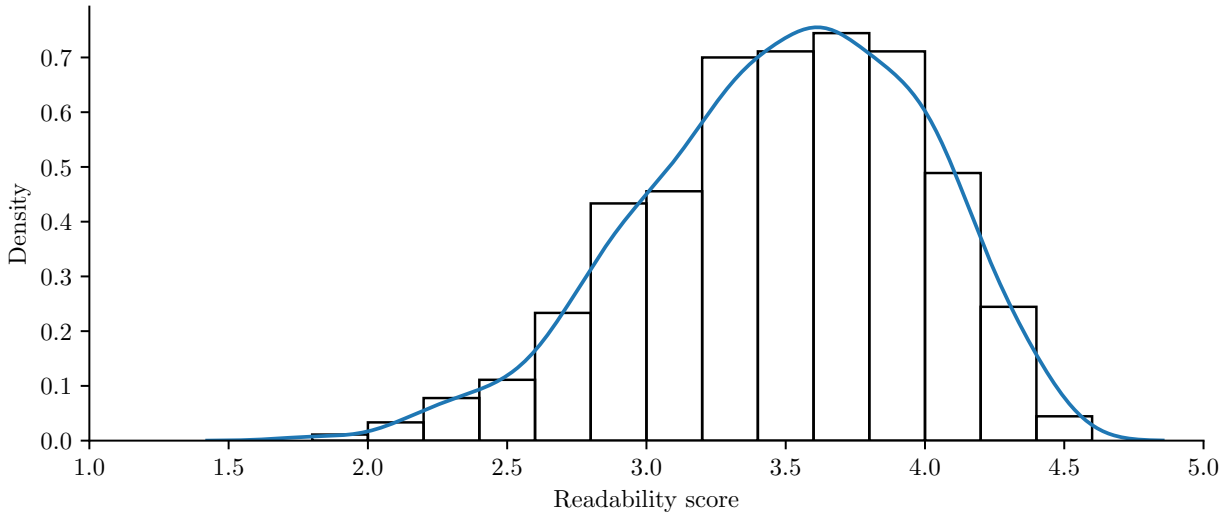


Figure 5.1: Distribution of average readability scores for unit tests in our training set.

first survey format) to train the regression metric and both the average readability scores and the majority readability choices (from the second survey format) to evaluate its performance, as described in the next section.

5.3.2 Feature Selection

Previous work suggests that regression techniques can effectively model readability as a function of syntactic, semantic, and textual complexity measures [192, 205]. We approach the problem of selecting the best set of features for our model as an attribute selection problem [196]. Thus, we first identified a number of relevant features, then selected the most relevant subset for use in our model. The larger set of features, numbering 116 in total, included all those used in previous models, as well as a number of new features measuring the number of assertions, exceptions, constructor or method calls, field accesses, branches, type casts, and different data types. (See Daka et al. [215] for a full discussion of the additional features we considered.)

To select the most relevant subset of these features, we used a wrapper selection algorithm [216]. Wrapper approaches operate by training a number of metrics, each with a different subset of features, then selecting the features that resulted in the metric with the best performance. We trained each metric using the test cases labeled with the average readability scores collected using the first (1–5 ranking) survey format. We defined the metric performance as the sum of the Pearson correlation with the labels, plus the average agreement with the majority of participants in the second (binary classification) survey format. Recall that the Pearson correlation ranges between -1 and 1, while the average agreement ranges between 0 (disagree on all training data) and 1 (agree on all training data). To help mitigate the danger of overfitting, we performed 10-fold cross-validation to compute the metric performance.

We used a steepest ascent hill climbing algorithm (see section 2.2.2) with random restarts as the core of our wrapper selection algorithm. Each restart of hill climbing selected a random feature, trained the metric, and recorded its performance as defined above. We then took the remaining $n - 1$ features and generated $n - 1$ two-feature models by pairing the previously selected feature with each of the remaining features in turn. After training metrics based on these models, we selected the model with the best performance and made its pair of features the new selected features. We then took the remaining $n - 2$ features, generated $n - 2$ three-feature models, identified the model with the best performance, and selected its features before starting the next iteration. This process continued, adding one feature at a time to the set of selected features, until no further performance improvements were found by adding another feature to the model. The set of features that produced the best performance after several random restarts gave us our final model. The final 16-feature metric is given in Daka et al. [215, table 1].

5.4 Test Case Representation and Transformations

Having introduced a fitness function that approximates human perception of readability, we next instantiate a search that generates test suites that optimize both code coverage and readability. We use EvoSuite [68], an extensible, search-based platform for generating high-coverage test suites, as the basis for our search algorithm. Unlike the optimization problem discussed in chapter 4, where a single set of mutation operators could modify either dimension of fitness, EvoSuite's transformations to optimize coverage tend to produce intermediate variants with low readability. Specifically, EvoSuite applies transformations that may introduce redundant statements that do not affect the coverage of the current individual, but instead provide opportunities for transformations that improve coverage in subsequent generations. Since our model penalizes (i.e., includes a negative coefficient for) longer tests, if we were to include readability as an additional objective in a multi-objective optimization algorithm, any tests including these redundant statements would be Pareto dominated by equivalent tests that do not include the redundancy. Since dominated variants are less likely to survive into the next generation, EvoSuite would have fewer opportunities for producing high-coverage test suites.

Instead, we search for a highly readable implementation of each test in the suite *after* having completed the search for a high-coverage test suite. Since we assume the test suite is already high-coverage, our search does not explicitly increase coverage. An advantage of this approach is that our search can optimize the readability of test suites created by other algorithms, or even human-written test suites, so long as they satisfy the assumptions of our representation. In particular, we represent tests as sequences of variable assignments and method calls. The tests generated by EvoSuite have exactly this structure; the only control flow consists of `try-catch` constructs surrounding particular statements. We assume that the test covers some requirements (or a proxy, such as branches) and that we are given a mechanism to verify that modified tests cover the same requirements.

Using this representation, we design a transformation operation that replaces assignments or method calls with alternatives that may be more readable. Specifically, our single transformation replaces the right-hand side of an assignment statement with a method or constructor call that returns a value of the type on the left-hand side. The search can verify that the test covers the same requirements and, if so, check whether the readability improved.

This transformation requires some attention to detail to ensure that the new test compiles. First, we supply the parameters of the new call with constants or with variables from previous lines of the test, chosen at random. If no in-scope variables with the correct type exist, a new assignment statement of the desired type is inserted immediately prior to the transformed statement and the right-hand side of the new assignment is generated using the same process. Second, the restriction on the return type ensures that method calls for which the variable is a parameter will continue to compile. Finally, we remove any variable assignments that are no longer used after replacing the old method call.

For example, consider line 6 in the test case in listing 5.1. Applying our transformation to this line replaces the constructor call on the right-hand side of the assignment with a new call that returns `ObjectHandlerAdapter`. In this example, we randomly chose a one-argument constructor of that type. To supply the parameter, we chose the constant `null`. The transformation now removes the assignment to the unused variable `rootHandler0`, after which `object0` is no longer used and its assignment is removed. This continues until all of the unused preceding lines in the test have been removed. The subjectively more readable test resulting from this transformation is shown in listing 5.2.

```
1 ElementName elementName0 = new ElementName("", "");
2 Class<Object> class0 = Object.class;
3 VirtualHandler virtualHandler0 =
      new VirtualHandler(elementName0, (Class) class0);
4 Object object0 = new Object();
5 RootHandler rootHandler0 =
      new RootHandler((ObjectHandler) virtualHandler0, object0);
6 ObjectHandlerAdapter objectHandlerAdapter0 =
      new ObjectHandlerAdapter((ObjectHandlerInterface) rootHandler0);
```

Listing 5.1: Example of test case before optimizing for readability. Long lines have been wrapped to fit on the page.

```
1 ObjectHandlerAdapter objectHandlerAdapter0 =
      new ObjectHandlerAdapter((ObjectHandlerInterface) null);
```

Listing 5.2: Test case in listing 5.1 after optimization. Long lines have been wrapped to fit on the page.

We implemented our search as a post-processing step in EvoSuite after the test suite is generated and minimized, but before assertions are inserted. Our search applies to each test individually, transforming the test as described above

under the constraint that the transformed test meets the same coverage objectives as the original.[13] This ensures that we maintain the same high level of coverage as the original generated test suite. In many cases, pruning tests that cover fewer requirements than the original renders the search space sufficiently small that we can explore it exhaustively, as we also saw in section 3.4.1. In particular, in our evaluation, we found just five alternatives for every test, on average.

5.5 Evaluation

We evaluate our readability optimization algorithm in terms of improvement in human perceptions of readability and improvement in human understanding.

We generated test suites using EvoSuite for classes selected from the open-source projects listed in section 5.3.1. Specifically, we selected 30 product classes from these projects for which EvoSuite was able to generate test suites with at least 80% branch coverage. We selected classes with fewer than 500 lines of code and relatively few dependencies to simplify the tasks assigned to participants in our human studies (see below for more details). Since EvoSuite internally uses stochastic search algorithms, we generated tests for these classes 10 times to get a variety of test implementations. This combination of classes, coverage, and multiple runs gave us a large number of tests to apply our readability optimization algorithm to. Our algorithm found alternative implementations for 56% of the tests we generated with EvoSuite. Only 5% of methods had no tests with alternative implementations. This suggests both that our algorithm should apply to a significant portion of generated tests suites and that further opportunities for readability optimization could exist.

To evaluate whether humans perceive the tests produced by our readability optimization algorithm as more readable, we conducted a human study. As in the earlier readability survey, all participants were required to pass a brief quiz to ensure familiarity with Java. We presented participants with pairs of tests, asking that they select the one they find more readable, as in the forced-choice survey described in section 5.3.1. In each pair, one test was optimized by our algorithm and the other was the output of EvoSuite without our algorithm. For this human study, we collected three pairs of tests for each class, with both tests in a pair covering the same branch. Our results, comprising 4150 choices from 131 Amazon Mechanical Turk participants, show that the study participants preferred the readability optimized test 69% of the time. In only one out of the 90 test pairs in the survey was the readability optimized test selected less than 50% of the time. We thus conclude that our algorithm successfully improves the readability of automatically generated tests as perceived by humans.

We also conducted a second human study to investigate whether humans were able to understand our more readable tests better. In this study, we presented participants with a test case along with the source code of the class being

[13] For example, the transformed test should cause the same `if`-statement branches to be evaluated to maintain a branch coverage objective. Alternatively, to maintain a mutation testing objective (see section 5.2.3), the transformed test should detect the same mutants as the original test.

	Readability		Time (min)		Correct (%)	
Class	Orig	Opt	Orig	Opt	Orig	Opt
`Attribute`	3.33	3.81	**4.7**	5.8	**60**	38
`ChainBase`	2.99	3.75	4.0	**3.8**	**76**	55
`CharRange`	3.82	4.04	4.4	**1.9**	50	**100**
`FilterListIterator`	3.10	3.73	4.8	**3.0**	71	**86**
`FixedOrderComparator`	2.63	3.30	4.9	**3.3**	**64**	23
`Option`	3.55	4.01	**2.2**	2.7	**100**	75
`PluginRules`	2.49	3.46	6.4	**6.0**	**73**	63
`RulesBase`	2.80	3.76	4.8	**3.2**	91	**100**
`StdXMLReader`	3.40	3.79	4.3	**3.8**	60	**61**
`YearMonthDay`	3.25	3.81	**6.1**	6.5	31	**62**
Average	3.14	3.75	4.6	**4.0**	68	68

Table 5.1: Human understanding task performance on 10 readability-optimized test cases. We present the average performance on the selected test cases for each class, with "Orig" indicating the test case was generated using the original EvoSuite algorithm (i.e., without optimizing readability) and "Opt" indicating it was generated using our readability optimization algorithm. The best response time and correctness scores for each class are indicated in bold.

tested and asked them whether the test would pass or fail. To gather failing tests for this study, we repeated the test generation process (with and without optimizing readability) described above, but modified EvoSuite to generate failing assertions. From this pool of passing and failing tests, we selected one pair of tests for each class—one optimized for readability and the other unoptimized—then chose the 10 pairs showing the largest difference in estimated readability according to our metric. After passing the Java familiarity quiz, participants were given one hour to determine whether ten tests (one randomly selected from each pair) would pass, with reference to the associated product code. Both the tests and the product code were presented with syntax highlighting in a web browser interface with IDE-like navigation and participants were instructed not to compile and run the code locally. In addition to recording whether their answers were correct, we also recorded the time they spent inspecting each test and associated product code.

Overall, the participants in this study, 30 students from the University of Sheffield, spent less time (238 s versus 274 s) inspecting the readability-optimized tests, on average. The Spearman's rank correlation between readability and response time was −0.22 ($p = 0.0001$), indicating a weak but significant connection; recall that a negative correlation indicates that response time decreases as readability increases. At least some of the residual difference may be due to differences in the methods and classes being tested. However, applying the Mann-Whitney U test to the optimized and unoptimized results for each class in turn, results in a p-value under 0.005 (using a lower threshold to account for repeated statistical tests) for `CharRange`. One possible explanation for the reduced response time on tests with higher readability scores would be if participants were in fact discouraged by the tests rather than finding them easier to understand. However, participants correctly identified whether the test would pass or fail in two thirds of cases, regardless of whether the tests were optimized for readability. This suggests that participants took the time to achieve similar levels of understanding for both optimized and unoptimized tests. We conclude that our readability optimization

algorithm allowed participants to reach this level of understanding more quickly.

5.6 Related Work

Readability Metrics. Readability metrics are well-established in the domain of non-software natural language. For example, the Flesch-Kincaid Grade Level [217] is integrated into popular editors, such as Microsoft Word. Metrics such as this, the Gunning-Fog Index [218], or the SMOG Index [219] are based on a few simple measurements, such as of of the number of syllables in words or the lengths of sentences.

To the best of our knowledge, Buse and Weimer were the first to apply a similar model of readability to source code [192]. Their model is based on character and token counts within the code being measured. Posnett et al. [205] developed a simpler model of code readability with fewer features, based on size, Halstead [201] metrics, and entropy. Both groups relied on the same human study data, 120 student participants evaluating 100 short product code samples, to determine coefficients for their metrics. Our work is based on the same readability model concept, but consists of a domain-specific model for unit tests, using dedicated test features, allowing us to train a metric with better performance and prediction power in our domain.

Understanding Tests. The problem of understanding tests, particularly failing tests, is well known. For human-written tests, Meszaros [220] and van Deursen et al. [221] recommend patterns to follow or avoid so that tests remain comprehensible and maintainable.

Several approaches have been suggested to simplify automatically-generated tests to aid in comprehension and debugging. Harman et al. [222] suggest reducing the number of tests generated in the first place. Fraser and Zeller [69] reduce the number of assertions in a test using mutation analysis. Zhang's SimpleTest algorithm [223] attempts to improve readability by transforming existing tests to use fewer statements. This transformation is the most similar to ours in both goal and behavior; however, our technique optimizes readability (as determined by our metric) directly and explores a more diverse set of tests, increasing the opportunities for optimization.

A few other researchers have proposed heuristics specifically to improve the readability of automatically-generated tests. For example, Fraser and Zeller [190] propose a test case generation algorithm based on models of common object usage learned from existing source code. Although their algorithm might improve the similarity between human-written code and sequences of instructions in the generated tests, it does not directly optimize measured readability. Afshan et al. [70] apply a natural language model to select natural string literals, which are often simply random characters, to more English-like text. Although we considered their language model when developing our model features, it did not provide significant predictive power on our training dataset.

On the basis that it is most important to understand a failing test, Leitner et al. [224] and Lei and Andrews [225] suggest algorithms specifically for simplifying failing tests and identifying the failure cause. Zhang et al. [226] synthesized natural language documentation to explain the failure. Xuan and Monperrus [227] refactor failing tests to have exactly one assertion to improve fault localization.

5.7 Conclusion

Unit tests, like most source code, must be read and understood by humans more often than they are written. This is especially true of automatically-generated tests, which are not written by humans. We introduce a technique to improve the readability of automatically generated tests to ease the burden on developers who must understand those tests. We use an approximate metric of human readability to estimate the readability of candidate tests during optimization. This metric is based on human ratings of readability; we show that humans continue to prefer tests that show high readability according to our metric. Our results also show that participants in our study were able to understand tests with higher readability scores more quickly.

Our technique to increase the readability of unit tests relies on a small set of changes to improve the appearance of tests. We generate alternative test implementations using these transformations and select the most readable. It would be interesting in the future to create alternative test implementations by integrating heuristic approaches. For example, our metric training data indicates that identifier names may play a particularly significant role in test case readability. Since many test case generation algorithms, including EvoSuite, use very simple algorithms for generating identifier names, improving these names (such as with language models [228, 229]), could be fruitful.

Our experiments also demonstrate potential distinctions between readability—i.e., the effect of textual presentation on understanding—and other influences on comprehensibility. Improving the readability of tests without taking other aspects, such as algorithm complexity, into account does not universally improve performance on tasks related to understanding. Including semantic features or explicit code complexity metrics as optimization targets may allow generating even more understandable tests.

Chapter 6

Conclusion

SOFTWARE plays a vital and ever-expanding role in the modern world. In addition to producing the correct outputs, which software engineers term functional correctness, software systems also possess non-functional properties [19] that can be of critical importance. Developers must produce software that achieves a reasonable balance between competing properties. In this dissertation, we presented a framework to provide assistance to programmers, allowing them to write a single initial implementation, automatically generate additional implementations with different non-functional properties, then present the best such implementations to the user. Our approach uses search-based optimization (SBO) algorithms to explore the program implementations made possible by a set of program transformations, selecting those with the best fitness. We demonstrated the generality of our approach by applying it in the context of three application domains, covering a variety of non-functional properties. In doing so, we highlighted the ways in which these different domains influenced the selection of domain-specific program transformations and search strategies.

In chapter 3, we investigated the problem of improving the visual quality of procedural shaders with respect to aliasing. This led us to develop a strategy for local program transformations that could reduce the aliasing of certain subexpressions in the program without the high runtime cost of previous techniques. Since our transformations introduced limited additional runtime by construction, the search algorithm we selected for this domain optimized only visual quality explicitly. We showed that our approach produced shaders that generated images with rendering time and output quality that Pareto dominated existing supersampling techniques in many cases. In most remaining cases, our technique introduced new points on a Pareto frontier that includes supersampling.

In chapter 4, we addressed the tradeoff between energy consumption and output quality in data centers. We chose a general set of program transformations capable of producing a wide variety of alternative implementations. To mitigate the limitations of existing simulation, performance counter, and consumer-level energy measurements, we

Venue	Title	Notes
TSE	Automatically Exploring Tradeoffs Between Software Output Fidelity and Energy Costs (*under submission*)	
PG	Towards Automatic Band-Limited Procedural Shaders [230]	
ESEC/FSE	Modeling Readability to Improve Unit Tests [215]	Distinguished paper
SSBSE	Generating Readable Unit Tests for Guava [231]	
ASPLOS	Post-compiler Software Optimization for Reducing Energy [155]	

Table 6.1: Publications supporting this dissertation.

developed an economical device to accurately measure energy consumption on servers in a data center setting. We used a multi-objective genetic algorithm (GA) to leverage the generality of our transformations to optimize both energy use and output quality, measuring the latter with a set of application-specific metrics. Our results showed that our approach achieved significant energy improvements while maintaining the same quality of output as the unoptimized program. By explicitly optimizing output quality simultaneously, we achieved even greater energy improvements while maintaining a human-acceptable quality of output. Finally, we showed that our approach is more effective than loop perforation, a state-of-the-art, but less general technique.

In chapter 5, we applied our framework to program readability as it applies to high-coverage test suites. We selected a transformation to produce alternative test implementations while retaining the structure common to automatically generated unit tests. To estimate the readability of large numbers of candidate test implementations, we used machine learning to develop a domain-specific readability metric for unit tests that correlated with human perception. Our search algorithm selected the most readable test implementation that maintained the coverage of the original test. We showed that humans find our tests to be more readable on average and that they were able to understand our tests more quickly than tests without our readability optimizations.

Table 6.1 lists peer-reviewed publications in support of the findings presented in this dissertation. The diversity of domains demonstrates the flexibility of the framework to apply to a plethora of properties, both static and dynamic. It also highlights the importance of considering multiple properties simultaneously when selecting transformations, fitness functions, and search algorithms. For example, although we were able to design transformations to modify aliasing with minimal effect on run time in chapter 3—particularly relative to the existing state of the art—naively implementing them in terms of existing antialiasing would have been insufficient. In chapters 4 and 5, the effect of our transformations were not limited to a single primary property, requiring us to embrace and exploit the simultaneous property changes (chapter 4) or to compensate in the search design (chapter 5). By bearing multiple properties in mind simultaneously when designing our representations, transformations, and searches, we successfully developed techniques that discovered useful balances of those properties in a wide variety of benchmark applications. These results demonstrate that it is possible to design and implement algorithms, not just for optimizing one property at a time, but for optimizing *tradeoffs* of non-functional properties in software.

Appendix A

Derivations of Band-Limited Expressions

In this appendix, we derive band-limited expressions for several common built-in shading language functions. These derivations fill in the rows of table 3.1.

As described in section 3.3, we define the band-limited expression for a function f, with respect to a band-limiting kernel k, as follows:

$$\widehat{f}(x, w) = \int_{-\infty}^{\infty} f(x')k(x - x', w)\, dx'. \tag{A.1}$$

This is equivalent, via a change of variables, to

$$\widehat{f}(x, w) = \int_{-\infty}^{\infty} f(x - x')k(x', w)\, dx'. \tag{A.2}$$

We desire that the integral of the band-limited function match that of the original function. To achieve this, we require that the band-limiting kernel be normalized such that,

$$\int_{-\infty}^{\infty} k(x, w)\, dx = 1. \tag{A.3}$$

Proposition A.1. *If* $\int_{-\infty}^{\infty} k(x', w)\, dx' = 1$, *then* $\int_{-\infty}^{\infty} \widehat{f}(x)\, dx = \int_{-\infty}^{\infty} f(x, w)\, dx$.

Proof. The result follows from manipulation of the integral for $\widehat{f}$.

$$\begin{aligned}
\int_{-\infty}^{\infty} \widehat{f}(x)\,dx &= \int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} f(x')k(x-x',w)\,dx' \right) dx && \text{substituting equation A.1} \\
&= \int_{-\infty}^{\infty} \left(\int_{-\infty}^{\infty} f(x')k(x-x',w)\,dx \right) dx' && \text{by Fubini's theorem [232]} \\
&= \int_{-\infty}^{\infty} f(x') \left(\int_{-\infty}^{\infty} k(x-x',w)\,dx \right) dx' \\
&= \int_{-\infty}^{\infty} f(x')\,dx' && \text{by assumption.} \qquad \square
\end{aligned} \tag{A.4}$$

Except as noted below, we will use a normalized Gaussian kernel with standard deviation w as our band-limiting kernel:

$$k(x,w) = \frac{1}{w\sqrt{2\pi}}\, e^{-\frac{x^2}{2w^2}}. \tag{A.5}$$

A.1 Useful Properties of the Gaussian Band-Limiting Kernel

In this section, we derive some useful properties of the Gaussian band-limiting kernel, starting with a demonstration that it is normalized as required.

Proposition A.2. *The Gaussian kernel defined above is normalized such that*

$$\int_{-\infty}^{\infty} \frac{1}{w\sqrt{2\pi}}\, e^{-\frac{x^2}{2w^2}}\,dx = 1. \tag{A.6}$$

Proof. We rearrange the integral to match the form given in Gradshteyn and Ryzhik [233, eq. 3.461-2], using $n = 0$ and $p = \frac{1}{2w^2}$,

$$\begin{aligned}
\int_{-\infty}^{\infty} \frac{1}{w\sqrt{2\pi}}\, e^{-\frac{x^2}{2w^2}}\,dx &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} e^{-\frac{x^2}{2w^2}}\,dx \\
&= \frac{2}{w\sqrt{2\pi}} \int_{0}^{\infty} e^{-\frac{x^2}{2w^2}}\,dx \quad \text{since } e^{-\frac{x^2}{2w^2}} \text{ is an even function of } x \\
&= \frac{2}{w\sqrt{2\pi}} \int_{0}^{\infty} e^{-px^2}\,dx \\
&= \frac{2}{w\sqrt{2\pi}} \left(\frac{1}{2}\sqrt{\frac{\pi}{p}} \right) \\
&= \frac{1}{w\sqrt{2}} \sqrt{2w^2} \\
&= 1. \qquad \square
\end{aligned} \tag{A.7}$$

The next several results relate the integral of the kernel to the Gauss error function (erf): $\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt$.

Proposition A.3. *Let* $f(x) = \frac{1}{w\sqrt{2\pi}} \int_0^x e^{-\frac{x'^2}{2w^2}}\,dx'$. *Then* $f(x) = \frac{1}{2} \operatorname{erf}\left(\frac{x}{w\sqrt{2}} \right)$.

Proof. We let $u = \frac{x'}{w\sqrt{2}}$, such that $du = \frac{dx'}{w\sqrt{2}}$. Using u-substitution, we have,

$$\begin{aligned} f(x) &= \frac{1}{w\sqrt{2\pi}} \int_0^x e^{-\frac{x'^2}{2w^2}}\, dx' \\ &= \frac{1}{w\sqrt{2\pi}} \int_0^{\frac{x}{w\sqrt{2}}} e^{-u^2} w\sqrt{2}\, du \\ &= \frac{1}{\sqrt{\pi}} \int_0^{\frac{x}{w\sqrt{2}}} e^{-u^2}\, du \\ &= \frac{1}{\sqrt{\pi}} \frac{\sqrt{\pi}}{2} \operatorname{erf}\left(\frac{x}{w\sqrt{2}}\right) \\ &= \frac{1}{2} \operatorname{erf}\left(\frac{x}{w\sqrt{2}}\right). \end{aligned} \tag{A.8}$$

□

Proposition A.4. *Let* $f(x) = \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{x'^2}{2w^2}}\, dx'$. *Then* $f(x) = \frac{1}{2} + \frac{1}{2} \operatorname{erf}\left(\frac{x}{w\sqrt{2}}\right)$.

Proof. We start by partitioning the integral in the definition of $f(x)$:

$$\begin{aligned} f(x) &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{x'^2}{2w^2}}\, dx' \\ &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^0 e^{-\frac{x'^2}{2w^2}}\, dx' + \frac{1}{w\sqrt{2\pi}} \int_0^x e^{-\frac{x'^2}{2w^2}}\, dx'. \end{aligned} \tag{A.9}$$

Since $e^{-\frac{x'^2}{2w^2}}$ is an even function of x', the value of the first integral is half the value of the integral from $-\infty$ to ∞, which we know to be 1. Therefore, the value of the first integral is $\frac{1}{2}$. Substituting this and the result from proposition A.3, we are left with,

$$f(x) = \frac{1}{2} + \frac{1}{2} \operatorname{erf}\left(\frac{x}{w\sqrt{2}}\right). \tag{A.10}$$

□

Proposition A.5. *Let* $f(x) = \frac{1}{w\sqrt{2\pi}} \int_a^b e^{-\frac{x'^2}{2w^2}}\, dx'$. *Then* $f(x) = \frac{1}{2}\left(\operatorname{erf}\left(\frac{b}{w\sqrt{2}}\right) - \operatorname{erf}\left(\frac{a}{w\sqrt{2}}\right)\right)$.

Proof. Note that we can add and subtract the integral from $-\infty$ to a without changing the value of $f(x)$:

$$\begin{aligned} f(x) &= \frac{1}{w\sqrt{2\pi}} \int_a^b e^{-\frac{x'^2}{2w^2}}\, dx' \\ &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^b e^{-\frac{x'^2}{2w^2}}\, dx' - \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^a e^{-\frac{x'^2}{2w^2}}\, dx' \\ &= \left(\frac{1}{2} + \frac{1}{2} \operatorname{erf}\left(\frac{b}{w\sqrt{2}}\right)\right) - \left(\frac{1}{2} + \frac{1}{2} \operatorname{erf}\left(\frac{a}{w\sqrt{2}}\right)\right) \quad \text{by proposition A.4} \\ &= \frac{1}{2}\left(\operatorname{erf}\left(\frac{b}{w\sqrt{2}}\right) - \operatorname{erf}\left(\frac{a}{w\sqrt{2}}\right)\right). \end{aligned} \tag{A.11}$$

□

Finally, we derive formulae for the definite integral of the product of the Gaussian kernel and the identity function.

Proposition A.6. *Let* $f(x) = \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^x x'\, e^{-\frac{x'^2}{2w^2}}\, dx'$. *Then* $f(x) = -\frac{w}{\sqrt{2\pi}}\, e^{-\frac{x^2}{2w^2}}$.

Proof. We let $u = -\frac{x'^2}{2w^2}$, such that $du = -\frac{x'}{w^2}\,dx'$. Using u-substitution,

$$\begin{aligned} f(x) &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{x} x'\, e^{-\frac{x'^2}{2w^2}}\, dx' \\ &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{-\frac{x^2}{2w^2}} e^u \left(-w^2\right)\, du \\ &= -\frac{w}{\sqrt{2\pi}} \int_{-\infty}^{-\frac{x^2}{2w^2}} e^u\, du \\ &= -\frac{w}{\sqrt{2\pi}}\, e^{-\frac{x^2}{2w^2}}. \end{aligned} \tag{A.12}$$

□

Proposition A.7. *Let* $f(x) = \frac{1}{w\sqrt{2\pi}} \int_x^\infty x'\, e^{-\frac{x'^2}{2w^2}}\, dx'$. *Then* $f(x) = \frac{w}{\sqrt{2\pi}}\, e^{-\frac{x^2}{2w^2}}$.

Proof. Note that, since x' is an odd function of x' and the Gaussian kernel is an even function of x', their product is an odd function. Thus, we see,

$$\begin{aligned} -f(x) &= -\frac{1}{w\sqrt{2\pi}} \int_x^\infty x'\, e^{-\frac{x'^2}{2w^2}}\, dx' \\ &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{-x} x'\, e^{-\frac{x'^2}{2w^2}}\, dx' && \text{since } x' \text{ is an odd function} \\ &= -\frac{w}{\sqrt{2\pi}}\, e^{-\frac{(-x)^2}{2w^2}} && \text{by proposition A.6} \\ f(x) &= \frac{w}{\sqrt{2\pi}}\, e^{-\frac{x^2}{2w^2}}. \end{aligned} \tag{A.13}$$

□

We will use these results to simplify the derivations throughout the rest of this appendix.

A.2 Combinations of Band-Limited Components

In this section, we derive results that will allow us to construct band-limited expressions in terms of sub-expressions. These results provide the underlying structure of our bottom-up approach to band-limiting shaders, as described in sections 3.3.1 and 3.3.4.

Proposition A.8. *Let* $f(x) = c$ *be a constant. Then* $\widehat{f}(x, w) = c$.

Proof. Starting with equation A.2,

$$\begin{aligned}\widehat{f}(x,w) &= \int_{-\infty}^{\infty} f(x-x')\,k(x',w)\,dx' \\ &= \int_{-\infty}^{\infty} c\,k(x',w)\,dx' \\ &= c\int_{-\infty}^{\infty} k(x',w)\,dx' \\ &= c \qquad\qquad \text{by equation A.3.} \quad \square\end{aligned} \tag{A.14}$$

Proposition A.9. *Let $f(x) = x$. If $k(x,w)$ is an even function (i.e., $k(x,w) = k(-x,w)$), then $\widehat{f}(x,w) = x$.*

Proof. Starting with equation A.2,

$$\begin{aligned}\widehat{f}(x,w) &= \int_{-\infty}^{\infty} f(x-x')k(x',w)\,dx' \\ &= \int_{-\infty}^{\infty} (x-x')k(x',w)\,dx' \\ &= \int_{-\infty}^{\infty} xk(x',w)\,dx' - \int_{-\infty}^{\infty} x'k(x',w)\,dx'.\end{aligned} \tag{A.15}$$

Since k is an even function of x' by assumption while x' is an odd function of x', their product is an odd function of x'. Therefore, the portion of the second integral from $-\infty$ to 0 has the same magnitude but opposite sign as the portion from 0 to ∞; the entire second integral evaluates to 0. Thus,

$$\begin{aligned}\widehat{f}(x,w) &= x\int_{-\infty}^{\infty} k(x',w)\,dx' \\ &= x \qquad\qquad \text{by equation A.3.} \quad \square\end{aligned} \tag{A.16}$$

Note that the band-limiting kernels considered in this dissertation—the Gaussian, rectangular, and tent kernels (see figure 3.3)—are all even functions. Thus when using any of these band-limiting kernels, if $f(x) = x$ then $\widehat{f}(x) = x$.

Proposition A.10. *Let $f(x) = g(x) + h(x)$. Then $\widehat{f}(x,w) = \widehat{g}(x,w) + \widehat{h}(x,w)$.*

Proof. Starting with equation A.1,

$$\begin{aligned}\widehat{f}(x,w) &= \int_{-\infty}^{\infty} f(x')k(x-x',w)\,dx' \\ &= \int_{-\infty}^{\infty} (g(x')+h(x'))k(x-x',w)\,dx' \\ &= \int_{-\infty}^{\infty} g(x')k(x-x',w)+h(x')k(x-x',w)\,dx' \\ &= \int_{-\infty}^{\infty} g(x')k(x-x',w)\,dx' + \int_{-\infty}^{\infty} h(x')k(x-x',w)\,dx' \\ &= \widehat{g}(x,w)+\widehat{h}(x,w). \qquad \square\end{aligned} \tag{A.17}$$

Proposition A.11. *Let $f(x) = cg(x)$, where c is a constant. Then $\widehat{f}(x,w) = c\widehat{g}(x,w)$.*

Proof. Starting with equation A.1,

$$\begin{aligned}\widehat{f}(x,w) &= \int_{-\infty}^{\infty} f(x')k(x-x',w)\,dx' \\ &= \int_{-\infty}^{\infty} cg(x')k(x-x',w)\,dx' \\ &= c\int_{-\infty}^{\infty} g(x')k(x-x',w)\,dx' \\ &= c\widehat{g}(x,w). \qquad \square\end{aligned} \tag{A.18}$$

Proposition A.12. *Let $f(x) = c_0 + c_1 f_1(x) + \cdots + c_n f_n(x)$, with functions $f_1, \ldots, f_n$ and constants $c_0, c_1, \ldots, c_n$. Then $\widehat{f}(x,w) = c_0 + c_1\widehat{f_1}(x,w) + \cdots + c_n\widehat{f_n}(x,w)$.*

Proof. Let $g_0(x) = c_0, g_1(x) = c_1 f_1(x), \ldots, g_n(x) = c_n f_n(x)$. Thus, $f(x) = g_0(x) + g_1(x) + \cdots + g_n(x)$. Then by proposition A.10,

$$\widehat{f}(x,w) = \widehat{g_0}(x,w) + \cdots + \widehat{g_n}(x,w). \tag{A.19}$$

By proposition A.8, $\widehat{g_0}(x,w) = c_0$. By proposition A.11, $\widehat{g_i}(x,w) = c_i\widehat{f_i}(x,w)$ for $1 \le i \le n$. Substituting into equation A.19, we have $\widehat{f}(x,w) = c_0 + c_1\widehat{f_1}(x,w) + \cdots + c_n\widehat{f_n}(x,w)$. $\square$

Proposition A.13. *Let f and k be partially multiplicatively separable functions of multiple dimensions such that $f = f_Y \cdot f_Z$ and $k = k_Y \cdot k_Z$, where k_Y and k_Z are functions of disjoint subsets of the dimensions of k and f_Y and f_Z are functions of the same disjoint subsets. Then $\widehat{f} = \widehat{f_Y} \cdot \widehat{f_Z}$.*

Proof. Let

$$f(\vec{x}) = f_Y(\vec{y})\, f_Z(\vec{z}) \qquad \text{and} \tag{A.20}$$

$$k(\vec{x}, \vec{w}) = k_Y(\vec{y}, \vec{u})\, k_Z(\vec{z}, \vec{v}), \tag{A.21}$$

where $\vec{x} = \{x_1, \ldots, x_{m+n}\}$, $\vec{y} = \{y_1, \ldots, y_m\}$, $\vec{z} = \{z_1, \ldots, z_n\}$, $\vec{w} = \{w_1, \ldots, w_{m+n}\}$, $\vec{u} = \{u_1, \ldots, u_m\}$, and $\vec{v} = \{v_1, \ldots, v_n\}$ are ordered sets of variables such that $\vec{y}$ and $\vec{z}$ partition $\vec{x}$ and $\vec{u}$ and $\vec{v}$ partition $\vec{w}$ in the same way. That is,

$$\begin{aligned} &\forall i, j \quad u_i = w_j \iff y_i = x_j \quad \text{and} \\ &\forall i, j \quad v_i = w_j \iff z_i = x_j. \end{aligned} \tag{A.22}$$

Let X, Y and Z be the domains of $\vec{x}$, $\vec{y}$, and $\vec{z}$, respectively.

Starting with the $(n+m)$-dimensional extension of equation A.1,

$$\begin{aligned} \widehat{f}(\vec{x}, \vec{w}) &= \int_X f(\vec{x})\, k(\vec{x} - \vec{x}', \vec{w})\, d\vec{x}' \\ &= \int_X f_Y(\vec{y})\, f_Z(\vec{z})\, k_Y(\vec{y} - \vec{y}', \vec{u})\, k_Z(\vec{z} - \vec{z}', \vec{v})\, d\vec{x}' && \text{substituting equations A.20 and A.21} \\ &= \int_Z \int_Y f_Z(\vec{z})\, k_Z(\vec{z} - \vec{z}', \vec{v})\, f_Y(\vec{y})\, k_Y(\vec{y} - \vec{y}', \vec{u})\, d\vec{y}'\, d\vec{z}' && \text{by Fubini's theorem [232]} \\ &= \int_Z \int_Y f_Z(\vec{z})\, k_Z(\vec{z} - \vec{z}', \vec{v}) \Big(f_Y(\vec{y})\, k_Y(\vec{y} - \vec{y}', \vec{u})\, d\vec{y}' \Big)\, d\vec{z}'. \end{aligned} \tag{A.23}$$

Since $f_Z(\vec{z})$ and $k_Z(\vec{z} - \vec{z}', \vec{v})$ are independent of $\vec{y}'$, we can move them outside the $\vec{y}'$ integrals, as follows,

$$\begin{aligned} \widehat{f}(\vec{x}, \vec{w}) &= \int_Z f_Z(\vec{z})\, k_Z(\vec{z} - \vec{z}', \vec{v}) \left(\int_Y f_Y(\vec{y})\, k_Y(\vec{y} - \vec{y}', \vec{u})\, d\vec{y}' \right) d\vec{z}' \\ &= \int_Z f_Z(\vec{z})\, k_Z(\vec{z} - \vec{z}', \vec{v})\, \widehat{f_Y}(\vec{y}, \vec{u})\, d\vec{z}' && \text{substituting equation A.1.} \end{aligned} \tag{A.24}$$

Since $\widehat{f_Y}(\vec{y}, \vec{u})$ is independent of $\vec{z}'$, we can move it outside the integral, giving us,

$$\begin{aligned} \widehat{f}(\vec{x}, \vec{w}) &= \widehat{f_Y}(\vec{y}, \vec{u}) \int_Z f_Z(\vec{z})\, k_Z(\vec{z} - \vec{z}', \vec{v})\, d\vec{z}' \\ &= \widehat{f_Y}(\vec{y}, \vec{u}) \widehat{f_Z}(\vec{z}, \vec{v}) && \text{substituting equation A.1.} \quad \square \end{aligned} \tag{A.25}$$

A.3 Shading Language Primitives

In this section, we derive band-limited expressions for primitive functions of a shading language. These expressions constitute the basis elements from which more complicated band-limited shaders may be constructed using the results in the previous section.

A.3.1 Absolute Value

Proposition A.14. *Let $f(x) = |x|$. Using the Gaussian band-limiting kernel,*

$$\widehat{f}(x,w) = x\,\mathrm{erf}\left(\frac{x}{w\sqrt{2}}\right) + w\sqrt{\frac{2}{\pi}}\,e^{-\frac{x^2}{2w^2}}. \tag{A.26}$$

Proof. We observe that,

$$|x-x'| = \begin{cases} x - x' & \text{if } x' < x, \\ x' - x & \text{otherwise.} \end{cases} \tag{A.27}$$

This allows us to substitute into equation A.2, partition, and rearrange as follows,

$$\begin{aligned} \widehat{f}(x,w) &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} |x-x'|\, e^{-\frac{x'^2}{2w^2}}\, dx' \\ &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{x} (x-x')\, e^{-\frac{x'^2}{2w^2}}\, dx' + \frac{1}{w\sqrt{2\pi}} \int_{x}^{\infty} (x'-x)\, e^{-\frac{x'^2}{2w^2}}\, dx' \\ &= \frac{x}{w\sqrt{2\pi}} \left(\int_{-\infty}^{x} e^{-\frac{x'^2}{2w^2}}\, dx' - \int_{x}^{\infty} e^{-\frac{x'^2}{2w^2}}\, dx' \right) \\ &\quad + \frac{1}{w\sqrt{2\pi}} \left(\int_{x}^{\infty} x'\, e^{-\frac{x'^2}{2w^2}}\, dx' - \int_{-\infty}^{x} x'\, e^{-\frac{x'^2}{2w^2}}\, dx' \right). \end{aligned} \tag{A.28}$$

Since the Gaussian kernel is an even function of x', the integral from x to ∞ is equal to the integral from $-\infty$ to $-x$. Thus,

$$\begin{aligned} \widehat{f}(x,w) &= \frac{x}{w\sqrt{2\pi}} \left(\int_{-\infty}^{x} e^{-\frac{x'^2}{2w^2}}\, dx' - \int_{-\infty}^{-x} e^{-\frac{x'^2}{2w^2}}\, dx' \right) + \frac{2w}{\sqrt{2\pi}}\, e^{-\frac{x^2}{2w^2}} && \text{by propositions A.6 and A.7} \\ &= \frac{x}{w\sqrt{2\pi}} \int_{-x}^{x} e^{-\frac{x'^2}{2w^2}}\, dx' + w\sqrt{\frac{2}{\pi}}\, e^{-\frac{x^2}{2w^2}} \\ &= \frac{x}{2} \left(\mathrm{erf}\left(\frac{x}{w\sqrt{2}}\right) - \mathrm{erf}\left(\frac{-x}{w\sqrt{2}}\right) \right) + w\sqrt{\frac{2}{\pi}}\, e^{-\frac{x^2}{2w^2}} && \text{by proposition A.5.} \end{aligned} \tag{A.29}$$

Since the error function is an odd function, we can simplify this to,

$$\begin{aligned} \widehat{f}(x,w) &= \frac{x}{2} \left(\mathrm{erf}\left(\frac{x}{w\sqrt{2}}\right) + \mathrm{erf}\left(\frac{x}{w\sqrt{2}}\right) \right) + w\sqrt{\frac{2}{\pi}}\, e^{-\frac{x^2}{2w^2}} \\ &= x\,\mathrm{erf}\left(\frac{x}{w\sqrt{2}}\right) + w\sqrt{\frac{2}{\pi}}\, e^{-\frac{x^2}{2w^2}}. \end{aligned} \tag{A.30}$$

□

A.3.2 Ceiling

Proposition A.15. *Let $f(x) = \lceil x \rceil$. Then, assuming the band-limiting function is even, $\widehat{f}(x,w) = \widehat{floor}(x,w) + 1$.*

Proof. Note that when x is not an integer, $\lceil x \rceil = \lfloor x \rfloor + 1$. Since the integers have zero measure and $\lceil x \rceil$ is finite for all finite x,

$$\widehat{f}(x,w) = \int_{-\infty}^{\infty} \lceil x \rceil \; e^{-\frac{(x-x')^2}{2w^2}} \, dx' = \int_{-\infty}^{\infty} (\lfloor x \rfloor + 1) \; e^{-\frac{(x-x')^2}{2w^2}} \, dx'. \tag{A.31}$$

Thus, if $g(x) = \lfloor x \rfloor + 1$, $\widehat{f}(x,w) = \widehat{g}(x,w)$. From proposition A.12, $\widehat{f}(x,w) = \widehat{g}(x,w) = \widehat{floor}(x,w) + 1$. □

A.3.3 Cosine

Proposition A.16. *Let $f(x) = \cos x$. Then, using the Gaussian band-limiting kernel, $\widehat{f}(x,w) = \cos x \, e^{-\frac{w^2}{2}}$.*

Proof. Using the identity $\cos(\alpha - \beta) = \cos\alpha\cos\beta + \sin\alpha\sin\beta$ [232], we can substitute into equation A.2 to get,

$$\begin{aligned}
\widehat{f}(x,w) &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} \cos(x - x') \, e^{-\frac{x'^2}{2w^2}} \, dx' \\
&= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} (\cos x \cos x' + \sin x \sin x') \, e^{-\frac{x'^2}{2w^2}} \, dx' \\
&= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} \cos x \cos x' \, e^{-\frac{x'^2}{2w^2}} \, dx' + \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} \sin x \sin x' \, e^{-\frac{x'^2}{2w^2}} \, dx' \\
&= \frac{\cos x}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} \cos x' \, e^{-\frac{x'^2}{2w^2}} \, dx' + \frac{\sin x}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} \sin x' \, e^{-\frac{x'^2}{2w^2}} \, dx'.
\end{aligned} \tag{A.32}$$

Note that $e^{-\frac{x'^2}{2w^2}}$ is an even function of x'. Since $\sin x'$ is an odd function of x', their product is an odd function, and the second integral evaluates to 0, since the portion from $-\infty$ to 0 has the same magnitude but opposite sign as the portion from 0 to ∞. Since $\cos x'$ is an even function of x', the first integral is equal to twice the integral from 0 to ∞. Thus, we are left with

$$\widehat{f}(x,w) = \frac{2\cos x}{w\sqrt{2\pi}} \int_{0}^{\infty} \cos x' \, e^{-\frac{x'^2}{2w^2}} \, dx'. \tag{A.33}$$

Gradshteyn and Ryzhik provide a solution for this integral [233, eq. 3.896-4], where $\beta = \frac{1}{2w^2}$ and $b = 1$,

$$\begin{aligned}
\widehat{f}(x,w) &= \frac{2\cos x}{w\sqrt{2\pi}} \int_{0}^{\infty} \cos bx' \, e^{-\beta x'^2} \, dx' \\
&= \frac{2\cos x}{w\sqrt{2\pi}} \left(\frac{1}{2} \sqrt{\frac{\pi}{\beta}} \, e^{-\frac{b^2}{4\beta}} \right) \\
&= \frac{\cos x}{w\sqrt{2}} \sqrt{2w^2} \, e^{-\frac{2w^2}{4}} \\
&= \cos x \, e^{-\frac{w^2}{2}}.
\end{aligned} \tag{A.34}$$

□

A.3.4 Floor

Proposition A.17. *Let $f(x) = \lfloor x \rfloor$ and let $fract(x) = x - \lfloor x \rfloor$. Then, assuming the band-limiting function is even, $\widehat{f}(x,w) = x - \widehat{fract}(x,w)$.*

Proof. Rearranging the definition of $fract$, we have $f(x) = \lfloor x \rfloor = x - fract(x)$. From propositions A.9 and A.12, then, $\widehat{f}(x, w) = x - \widehat{fract}(x, w)$. □

A.3.5 Fractional Part

As discussed in section 3.3.2, we derive the band-limited expression for the fract function ($fract$), defined as $fract(x) = x - \lfloor x \rfloor$, using several band-limiting kernels. As with the other derivations in this appendix, we start with the Gaussian kernel, but in this case we also include derivations for the rectangular and tent kernels.

Throughout this section, we adopt the notation $fract^n(x)$ to mean $(fract(x))^n$.

Gaussian Kernel ($fract_G$)

Directly computing the integral of the product of the $fract$ function with the Gaussian kernel is particularly challenging. Instead, we apply the convolution theorem [105], which links the convolution of f and k with the Fourier transforms of f and k. The Fourier transform is commonly used in signal processing and relates a function of position (e.g., $f(x)$) to a function of frequency (e.g., $f(\nu)$). Specifically, the theorem states that,

$$\widehat{f}(x, w) = \mathcal{F}_\nu^{-1}\left[\mathcal{F}_x\left[f(x)\right](\nu) \cdot \mathcal{F}_x\left[k(x, w)\right](\nu)\right], \tag{A.35}$$

where $\mathcal{F}_x[f(x)](\nu)$ is the Fourier transform of $f(x)$ as a function of the frequency ν and $\mathcal{F}_\nu^{-1}[f(\nu)](x)$ is its inverse.

The Fourier transform and inverse are defined as,

$$\begin{aligned} \mathcal{F}_x\left[f(x)\right](\nu) &= \int_{-\infty}^{\infty} f(x)\, e^{-2\pi i \nu x}\, dx \\ \mathcal{F}_\nu^{-1}\left[f(\nu)\right](x) &= \int_{-\infty}^{\infty} f(\nu)\, e^{2\pi i \nu x}\, d\nu. \end{aligned} \tag{A.36}$$

Proposition A.18. *Let $f(x) = fract(x) = x - \lfloor x \rfloor$. With the Gaussian band-limiting kernel (equation A.5),*

$$\widehat{f}(x, w) = \frac{1}{2} - \sum_{n=1}^{\infty} \frac{e^{-2w^2\pi^2n^2}}{\pi n} \sin(2\pi n x). \tag{A.37}$$

Proof. We apply the convolution theorem to compute the convolution of $fract(x)$ with a Gaussian kernel.

Rather than computing the Fourier transform of $fract(x)$ directly, we first derive the expression for its Fourier series expansion. The Fourier series expansion for a periodic function defines the function in terms of an infinite series of sines and cosines. For any periodic function $g(x)$ with T, the Fourier series is given by [105],

$$a_0 + \sum_{n=1}^{\infty} \left(a_n \cos\left(\frac{2\pi n x}{T}\right) + b_n \sin\left(\frac{2\pi n x}{T}\right)\right), \tag{A.38}$$

where

$$\begin{aligned} a_0 &= \frac{1}{T}\int_0^T g(x)\,dx \\ a_n &= \frac{2}{T}\int_0^T g(x)\cos\left(\frac{2\pi nx}{T}\right)\,dx \\ b_n &= \frac{2}{T}\int_0^T g(x)\sin\left(\frac{2\pi nx}{T}\right)\,dx. \end{aligned} \tag{A.39}$$

Substituting with $fract(x)$, for which $T = 1$,

$$\begin{aligned} a_0 &= \int_0^1 fract(x)\,dx \\ a_n &= 2\int_0^1 fract(x)\cos(2\pi nx)\,dx \\ b_n &= 2\int_0^1 fract(x)\sin(2\pi nx)\,dx. \end{aligned} \tag{A.40}$$

Since $fract(x) = x$ over the interval $[0, 1]$, we can replace $fract(x)$ with x in each of the integrals in equation A.40. Thus, $a_0 = \int_0^1 x\,dx = \frac{1}{2}$. To solve for a_n, we let $u = x$ and $dv = \cos(2\pi nx)\,dx$. Then $du = dx$ and $v = \frac{1}{2\pi n}\sin(2\pi nx)$. Using integration by parts, we have,

$$\begin{aligned} a_n &= 2\left[\frac{x}{2\pi n}\sin(2\pi nx)\right]_0^1 - \frac{2}{2\pi n}\int_0^1 \sin(2\pi nx)\,dx \\ &= 0 + \left[\frac{\cos(2\pi nx)}{2\pi^2 n^2}\right]_0^1 \\ &= 0. \end{aligned} \tag{A.41}$$

To solve for b_n, we let $u = x$ and $dv = \sin(2\pi nx)\,dx$, so that $du = dx$ and $v = -\frac{1}{2\pi n}\cos(2\pi nx)$. Using integration by parts again,

$$\begin{aligned} b_n &= 2\left[-\frac{x}{2\pi n}\cos(2\pi nx)\right]_0^1 + \frac{2}{2\pi n}\int_0^1 \cos(2\pi nx)\,dx \\ &= -\frac{1}{\pi n} + \left[\frac{\sin(2\pi nx)}{2\pi^2 n^2}\right]_0^1 \\ &= -\frac{1}{\pi n}. \end{aligned} \tag{A.42}$$

Substituting these coefficients back into equation A.38, we find,

$$fract(x) = \frac{1}{2} - \sum_{n=1}^{\infty}\frac{\sin(2\pi nx)}{\pi n}. \tag{A.43}$$

From this, we can compute the Fourier transform of $fract$,

$$\begin{aligned}\mathcal{F}_x\left[fract(x)\right](\nu) &= \int_{-\infty}^{\infty}\left(\frac{1}{2}-\sum_{n=1}^{\infty}\frac{\sin(2\pi nx)}{\pi n}\right)e^{-2\pi\nu x}\,dx \\ &= \frac{1}{2}\int_{-\infty}^{\infty}e^{-2\pi\nu x}\,dx-\sum_{n=1}^{\infty}\frac{1}{\pi n}\int_{-\infty}^{\infty}\sin(2\pi nx)\,e^{-2\pi\nu x}\,dx \\ &= \frac{1}{2}\mathcal{F}_x\left[1\right](\nu)-\sum_{n=1}^{\infty}\frac{1}{\pi n}\mathcal{F}_x\left[\sin(2\pi nx)\right](\nu) \\ &= \frac{1}{2}\delta(\nu)-\sum_{n=1}^{\infty}\frac{1}{2\pi n}i\left(\delta(\nu+n)-\delta(\nu-n)\right) \qquad \text{[105]},\end{aligned} \tag{A.44}$$

using the Dirac delta function (δ). The Fourier transform of our Gaussian kernel is,

$$\begin{aligned}\mathcal{F}_x\left[\frac{1}{w\sqrt{2\pi}}e^{-\frac{x^2}{2w^2}}\right](\nu) &= \frac{1}{w\sqrt{2\pi}}\,w\sqrt{2\pi}\,e^{-2w^2\pi^2\nu^2} \qquad \text{[105]} \\ &= e^{-2w^2\pi^2\nu^2}.\end{aligned} \tag{A.45}$$

We may now substitute equations A.44 and A.45 into equation A.35:

$$\begin{aligned}\widehat{f}(x,w) &= \mathcal{F}_\nu^{-1}\left[\mathcal{F}_x\left[f(x)\right](\nu)\cdot\mathcal{F}_x\left[k(x,w)\right](\nu)\right] \\ &= \mathcal{F}_\nu^{-1}\left[\mathcal{F}_x\left[fract(x)\right](\nu)\cdot\mathcal{F}_x\left[\frac{1}{w\sqrt{2\pi}}e^{-\frac{x^2}{2w^2}}\right](\nu)\right](x) \\ &= \mathcal{F}_\nu^{-1}\left[\frac{e^{-2w^2\pi^2\nu^2}}{2}\delta(\nu)-\sum_{n=1}^{\infty}\frac{e^{-2w^2\pi^2\nu^2}}{2\pi n}i(\delta(\nu+n)-\delta(\nu-n))\right](x).\end{aligned} \tag{A.46}$$

Since, by definition, $\delta(x)=0$ unless $x=0$, we may replace the ν in each exponential with the value that causes the argument of δ to be 0. In the first term, this means $\nu=0$; in the summation, $\nu=\pm n$, so $\nu^2=n^2$:

$$\begin{aligned}\widehat{f}(x,w) &= \mathcal{F}_\nu^{-1}\left[\frac{1}{2}\delta(\nu)-\sum_{n=1}^{\infty}\frac{e^{-2w^2\pi^2n^2}}{2\pi n}i\left(\delta(\nu+n)-\delta(\nu-n)\right)\right](x) \\ &= \frac{1}{2}-\sum_{n=1}^{\infty}\frac{e^{-2w^2\pi^2n^2}}{\pi n}\sin(2\pi nx) \qquad \text{[105]}.\end{aligned} \tag{A.47}$$

□

Rectangular Kernel ($fract_R$)

The band-limited expression in equation A.47 includes an infinite sum. As discussed in section 3.3.2, this summation may be truncated, but in some cases may still result in unacceptable run times or image quality. For this reason, in this section, we consider band-limiting with the rectangular function,

$$k(x,w)=\frac{1}{w}\left(step\left(x+\frac{w}{2}\right)-step\left(x-\frac{w}{2}\right)\right), \tag{A.48}$$

where $step$ is the Heaviside unit step (H), defined to equal 0 if $x < 0$ and 1 if $x \geq 0$.

We use Heckbert's technique of repeated integration [112] to derive the convolution of $fract(x)$ with a rectangular kernel. This technique is built upon the convolution theorem and states that the convolution of f and k is equal to the convolution of the integral of f (i.e., the function $F(x) = \int_0^x f(x')\,dx'$) and the derivative of k.

Proposition A.19. *The derivative of the rectangular kernel (equation A.48) is* $\frac{1}{w}\left(\delta\left(x + \frac{w}{2}\right) - \delta\left(x - \frac{w}{2}\right)\right)$.

Proof. Since $\frac{d}{dx} step(x) = \delta(x)$ [105], we can use the chain rule to get,

$$\frac{d}{dx}\,k(x,w) = \frac{1}{w}\left(\delta\left(x + \frac{w}{2}\right) - \delta\left(x - \frac{w}{2}\right)\right). \tag{A.49}$$

□

Proposition A.20. *Let* $F(x) = \int_0^x fract(x')\,dx'$. *Then,* $F(x) = \frac{1}{2}\left(fract^2(x) + \lfloor x \rfloor\right)$.

Proof. We start with the definition of $F(x)$ and compute the integrals between the discontinuities as follows,

$$\begin{aligned} F(x) &= \int_0^x fract(x')\,dx' \\ &= \int_0^{\lfloor x \rfloor} fract(x')\,dx' + \int_{\lfloor x \rfloor}^x fract(x')\,dx' \\ &= \sum_{n=0}^{\lfloor x \rfloor - 1} \int_n^{n+1} fract(x')\,dx' + \int_{\lfloor x \rfloor}^x fract(x')\,dx'. \end{aligned} \tag{A.50}$$

Note that, since the period of $fract(x)$ is 1, we can subtract any integer value from both bounds of integration without changing the value of the integral. In particular, since n and $\lfloor x \rfloor$ are both integers,

$$\begin{aligned} F(x) &= \sum_{n=0}^{\lfloor x \rfloor - 1} \int_0^1 fract(x')\,dx' + \int_0^{x - \lfloor x \rfloor} fract(x')\,dx' \\ &= \lfloor x \rfloor \int_0^1 fract(x')\,dx' + \int_0^{x - \lfloor x \rfloor} fract(x')\,dx'. \end{aligned} \tag{A.51}$$

Since $fract(x') = x'$ when $0 < x' < 1$, we can replace $fract(x')$ with x' in both integrals, leaving us with,

$$\begin{aligned} F(x) &= \lfloor x \rfloor \int_0^1 x'\,dx' + \int_0^{x - \lfloor x \rfloor} x'\,dx' \\ &= \frac{\lfloor x \rfloor}{2} + \frac{(x - \lfloor x \rfloor)^2}{2} \\ &= \frac{fract^2(x) + \lfloor x \rfloor}{2} \end{aligned} \tag{A.52}$$

□

Proposition A.21. *Let $f(x) = fract(x) = x - \lfloor x \rfloor$. With the rectangular band-limiting kernel (equation A.48),*

$$\widehat{f}(x,w) = \frac{fract^2\left(x+\frac{w}{2}\right) + \lfloor x+\frac{w}{2}\rfloor - fract^2\left(x-\frac{w}{2}\right) - \lfloor x-\frac{w}{2}\rfloor}{2w}. \tag{A.53}$$

Proof. Let $F(x) = \int_{-\infty}^{x} f(x')\,dx'$. Then, using Heckbert's technique of repeated integration,

$$\widehat{f}(x,w) = \int_{-\infty}^{\infty} f(x-x')k(x',w)\,dx' = \int_{-\infty}^{\infty} F(x-x')\frac{d}{dx'}k(x',w)\,dx'. \tag{A.54}$$

If we substitute equations A.49 and A.52, we get,

$$\begin{aligned}\widehat{f}(x,w) &= \int_{-\infty}^{\infty} \frac{fract^2(x-x') + \lfloor x-x'\rfloor}{2w}\left(\delta\left(x'+\frac{w}{2}\right) - \delta\left(x'-\frac{w}{2}\right)\right)\,dx' \\ &= \int_{-\infty}^{\infty} \frac{fract^2(x-x') + \lfloor x-x'\rfloor}{2w}\,\delta\left(x'+\frac{w}{2}\right)\,dx' \\ &\quad - \int_{-\infty}^{\infty} \frac{fract^2(x-x') + \lfloor x-x'\rfloor}{2w}\,\delta\left(x'-\frac{w}{2}\right)\,dx'.\end{aligned} \tag{A.55}$$

Since the delta function is zero everywhere except 0, the first and second integrals reduce to evaluating the fraction at $x' = -\frac{w}{2}$ and $x' = \frac{w}{2}$, respectively:

$$\widehat{f}(x,w) = \frac{fract^2\left(x+\frac{w}{2}\right) + \lfloor x+\frac{w}{2}\rfloor - fract^2\left(x-\frac{w}{2}\right) - \lfloor x-\frac{w}{2}\rfloor}{2w}. \tag{A.56}$$

□

Tent Kernel ($fract_T$)

The band-limited expression in equation A.56 permits noticeable aliasing. In this section, we consider band-limiting with the tent function,

$$k(x,w) = \frac{1}{w}\max\left(0, 1 - \left|\frac{x}{w}\right|\right). \tag{A.57}$$

We again use repeated integration to derive the convolution of $fract(x)$ with a tent kernel.

Proposition A.22. *The second derivative of the tent kernel (equation A.57) is $\frac{1}{w^2}\left(\delta(x+w) - 2\delta(x) + \delta(x-w)\right)$.*

Proof. Observe that the tent kernel consists of four piece-wise linear segments. Addressing each piece separately,

$$k(x,w) = \frac{1}{w}\max\left(0, 1 - \left|\frac{x}{w}\right|\right) = \begin{cases} 0 & \text{if } x < -w \\ 1 + \frac{x}{w^2} & \text{if } -w < x < 0, \\ 1 - \frac{x}{w^2} & \text{if } 0 < x < w, \\ 0 & \text{otherwise.} \end{cases} \tag{A.58}$$

The first derivative of k with respect to x is therefore,

$$\frac{d}{dx}\,k(x,w) = \begin{cases} 0 & \text{if } x < -w \\ \frac{1}{w^2} & \text{if } -w < x < 0, \\ -\frac{1}{w^2} & \text{if } 0 < x < w, \\ 0 & \text{otherwise.} \end{cases} \tag{A.59}$$

Note that this is equivalent to $\frac{1}{w^2}\left(step(x+w) - 2step(x) + step(x-w)\right)$. Since the derivative of $step(x)$ is $\delta(x)$, the derivative of this expression (respectively, the second derivative of $k(x,w)$) is,

$$\frac{d^2}{dx^2}\,k(x,w) = \frac{1}{w^2}\left(\delta(x+w) - 2\delta(x) + \delta(x-w)\right). \tag{A.60}$$

□

Proposition A.23. *Let $F(x) = \int_0^x \int_0^{x'} fract(x'')\,dx''\,dx'$ be the second integral of $fract(x)$. Then,*

$$F(x) = \frac{1}{12}\left(3x^2 + 2fract^3(x) - 3fract^2(x) - x + fract(x)\right). \tag{A.61}$$

Proof. We start with the definition of $F(x)$ and substitute the first integral of $fract(x)$, as follows,

$$\begin{aligned} F(x) &= \int_0^x \int_0^{x'} fract(x'')\,dx''\,dx' \\ &= \int_0^x \frac{fract^2(x') + \lfloor x' \rfloor}{2}\,dx' && \text{by proposition A.20} \\ &= \frac{1}{2}\int_0^x fract^2(x')\,dx' + \frac{1}{2}\int_0^x \lfloor x' \rfloor\,dx'. \end{aligned} \tag{A.62}$$

We compute the two integrals separately, starting with the first.

As above, we partition the first integral at the discontinuities:

$$\int_0^x fract^2(x')\,dx' = \sum_{n=0}^{\lfloor x \rfloor - 1} \int_n^{n+1} fract^2(x')\,dx' + \int_{\lfloor x \rfloor}^x fract^2(x')\,dx'. \tag{A.63}$$

Since the period of $fract(x)$ is 1, so is the period of $fract^2(x)$. We can therefore subtract any integer value from both bounds of integration without changing the value of the integral. In particular, since n and $\lfloor x \rfloor$ are both integers,

$$\begin{aligned}\int_0^x fract^2(x')\,dx' &= \sum_{n=0}^{\lfloor x \rfloor - 1} \int_0^1 fract^2(x')\,dx' + \int_0^{x-\lfloor x \rfloor} fract^2(x')\,dx' \\ &= \lfloor x \rfloor \int_0^1 fract^2(x')\,dx' + \int_0^{x-\lfloor x \rfloor} fract^2(x')\,dx'.\end{aligned} \tag{A.64}$$

Since $fract(x) = x$ when $0 < x < 1$, we can substitute into both integrals, leaving us with,

$$\begin{aligned}\int_0^x fract^2(x')\,dx' &= \lfloor x \rfloor \int_0^1 x'^2\,dx' + \int_0^{x-\lfloor x \rfloor} x'^2\,dx' \\ &= \frac{\lfloor x \rfloor}{3} + \frac{(x - \lfloor x \rfloor)^3}{3} \\ &= \frac{\lfloor x \rfloor + fract^3(x)}{3}.\end{aligned} \tag{A.65}$$

Returning to the second integral of equation A.62, we have,

$$\begin{aligned}\int_0^x \lfloor x' \rfloor\,dx' &= \int_0^x x' - fract(x')\,dx' && \text{by definition of } fract(x) \\ &= \int_0^x x'\,dx' - \int_0^x fract(x')\,dx' \\ &= \frac{x^2}{2} - \frac{fract^2(x) + \lfloor x \rfloor}{2} && \text{by proposition A.20} \\ &= \frac{x^2 - fract^2(x) - \lfloor x \rfloor}{2}.\end{aligned} \tag{A.66}$$

Substituting equations A.65 and A.66 into equation A.62 gives us,

$$\begin{aligned}F(x) &= \frac{1}{2}\int_0^x fract^2(x')\,dx' + \frac{1}{2}\int_0^x \lfloor x' \rfloor\,dx' \\ &= \frac{\lfloor x \rfloor + fract^3(x)}{6} + \frac{x^2 - fract^2(x) - \lfloor x \rfloor}{4} \\ &= \frac{2fract^3(x) + 2\lfloor x \rfloor}{12} + \frac{3x^2 - 3fract^2(x) - 3\lfloor x \rfloor}{12} \\ &= \frac{3x^2 - 2fract^3(x) - 3fract^2(x) - \lfloor x \rfloor}{12} \\ &= \frac{3x^2 - 2fract^3(x) - 3fract^2(x) - x + fract(x)}{12}.\end{aligned} \tag{A.67}$$

□

Proposition A.24. *Let $f(x) = fract(x)$. With the tent band-limiting kernel (equation A.57),*

$$\widehat{f}(x, w) = \frac{F(x+w) - 2F(x) + F(x-w)}{w^2}, \tag{A.68}$$

where $F(x)$ is defined as in proposition A.23.

Proof. Let $F(x) = \int_0^x \int_0^{x'} f(x'')\,dx''\,dx'$. Then, using repeated integration twice,

$$\widehat{f}(x,w) = \int_{-\infty}^{\infty} f(x-x')k(x',w)\,dx' = \int_{-\infty}^{\infty} F(x-x')\frac{d^2}{dx'^2}k(x',w)\,dx'. \tag{A.69}$$

If we substitute equation A.60, we get,

$$\begin{aligned}\widehat{f}(x,w) &= \frac{1}{w^2}\int_{-\infty}^{\infty} F(x-x')\left(\delta(x'+w) - 2\delta(x') + \delta(x'-w)\right)\,dx' \\ &= \frac{1}{w^2}\int_{-\infty}^{\infty} F(x-x')\delta(x'+w)\,dx' - \frac{2}{w^2}\int_{-\infty}^{\infty} F(x-x')\delta(x')\,dx' + \frac{1}{w^2}\int_{-\infty}^{\infty} F(x-x')\delta(x'-w)\,dx'.\end{aligned} \tag{A.70}$$

Since the delta function is zero everywhere except 0, the three integrals reduce to evaluating $F(x-x')$ at $x' = -w$, $x' = 0$, and $x' = w$, respectively:

$$\widehat{f}(x,w) = \frac{F(x+w) - 2F(x) + F(x-w)}{w^2}. \tag{A.71}$$

□

A.3.6 Saturate

Proposition A.25. *Let $f(x) = saturate(x) = \max(0, \min(1, x))$. Using the Gaussian band-limiting kernel,*

$$\widehat{f}(x,w) = \frac{1}{2}\left(x\,\mathrm{erf}\left(\frac{x}{w\sqrt{2}}\right) - (x-1)\,\mathrm{erf}\left(\frac{x-1}{w\sqrt{2}}\right) + w\sqrt{\frac{2}{\pi}}\left(e^{-\frac{x^2}{2w^2}} - e^{-\frac{(x-1)^2}{2w^2}}\right) + 1\right). \tag{A.72}$$

Proof. Substituting the definition of *saturate* into equation A.2,

$$\begin{aligned}\widehat{f}(x,w) &= \frac{1}{w\sqrt{2\pi}}\int_{-\infty}^{\infty} saturate(x-x')\,e^{-\frac{x'^2}{2w^2}}\,dx' \\ &= \frac{1}{w\sqrt{2\pi}}\int_{-\infty}^{\infty} \max(0, \min(1, x-x'))\,e^{-\frac{x'^2}{2w^2}}\,dx'.\end{aligned} \tag{A.73}$$

Note that $\min(1, x-x') \leq 0$ when $x \leq x'$; thus $\max(0, \min(1, x-x')) = 0$ when $x \leq x' < \infty$. We can therefore reduce the upper bound of the integral to x without affecting its value. Conversely, $\min(1, x-x') > 0$ when $x' < x$, meaning $\max(0, \min(1, x-x')) = \min(1, x-x')$ over the remaining bounds of integration. Thus, we can rewrite the integrand without the $\max$ function,

$$\widehat{f}(x,w) = \frac{1}{w\sqrt{2\pi}}\int_{-\infty}^{x} \min(1, x-x')e^{-\frac{x'^2}{2w^2}}\,dx'. \tag{A.74}$$

Now note that when $x' \leq x-1$, $\min(1, x-x') = 1$ and when $x-1 < x' \leq x$, $\min(1, x-x') = x-x'$. Thus, we can partition the integral into two terms without reference to the $\min$ function:

$$\begin{aligned}
\widehat{f}(x,w) &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{x-1} e^{-\frac{x'^2}{2w^2}}\, dx' + \frac{1}{w\sqrt{2\pi}} \int_{x-1}^{x} (x-x')\, e^{-\frac{x'^2}{2w^2}}\, dx' \\
&= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{x-1} e^{-\frac{x'^2}{2w^2}}\, dx' + \frac{1}{w\sqrt{2\pi}} \int_{x-1}^{x} x\, e^{-\frac{x'^2}{2w^2}}\, dx' - \frac{1}{w\sqrt{2\pi}} \int_{x-1}^{x} x'\, e^{-\frac{x'^2}{2w^2}}\, dx' \\
&= \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x-1}{w\sqrt{2}}\right)\right) + \frac{x}{2}\left(\operatorname{erf}\left(\frac{x}{w\sqrt{2}}\right) - \operatorname{erf}\left(\frac{x-1}{w\sqrt{2}}\right)\right) - \frac{1}{w\sqrt{2\pi}} \int_{x-1}^{x} x'\, e^{-\frac{x'^2}{2w^2}}\, dx'.
\end{aligned} \tag{A.75}$$

The last line follows from propositions A.4 and A.5.

Looking at the remaining integral of this equation, we can extend the upper bound of the integral and apply proposition A.7:

$$\begin{aligned}
\frac{1}{w\sqrt{2\pi}} \int_{x-1}^{x} x'\, e^{-\frac{x'^2}{2w^2}}\, dx' &= \frac{1}{w\sqrt{2\pi}} \int_{x-1}^{\infty} x'\, e^{-\frac{x'^2}{2w^2}}\, dx' - \frac{1}{w\sqrt{2\pi}} \int_{x}^{\infty} x'\, e^{-\frac{x'^2}{2w^2}}\, dx' \\
&= \frac{w}{\sqrt{2\pi}} \left(e^{-\frac{(x-1)^2}{2w^2}} - e^{-\frac{x^2}{2w^2}}\right).
\end{aligned} \tag{A.76}$$

Substituting back into equation A.75, we have,

$$\begin{aligned}
\widehat{f}(x,w) &= \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x-1}{w\sqrt{2}}\right)\right) + \frac{x}{2}\left(\operatorname{erf}\left(\frac{x}{w\sqrt{2}}\right) - \operatorname{erf}\left(\frac{x-1}{w\sqrt{2}}\right)\right) - \frac{w}{\sqrt{2\pi}}\left(e^{-\frac{x-1^2}{2w^2}} - e^{-\frac{x^2}{2w^2}}\right) \\
&= \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x-1}{w\sqrt{2}}\right) + x\operatorname{erf}\left(\frac{x}{w\sqrt{2}}\right) - x\operatorname{erf}\left(\frac{x-1}{w\sqrt{2}}\right) - w\sqrt{\frac{2}{\pi}}\left(e^{-\frac{(x-1)^2}{2w^2}} - e^{-\frac{x^2}{2w^2}}\right)\right) \\
&= \frac{1}{2}\left(x\operatorname{erf}\left(\frac{x}{w\sqrt{2}}\right) - (x-1)\operatorname{erf}\left(\frac{x-1}{w\sqrt{2}}\right) + w\sqrt{\frac{2}{\pi}}\left(e^{-\frac{x^2}{2w^2}} - e^{-\frac{(x-1)^2}{2w^2}}\right) + 1\right).
\end{aligned} \tag{A.77}$$

□

A.3.7 Sine

Proposition A.26. *Let $f(x) = \sin x$. Then, using the Gaussian band-limiting kernel, $\widehat{f}(x,w) = \sin x\, e^{-\frac{w^2}{2}}$.*

Proof. Using the identity $\sin(\alpha-\beta) = \sin\alpha\cos\beta - \cos\alpha\sin\beta$ [232], we can substitute into equation A.2 to get,

$$\begin{aligned}
\widehat{f}(x,w) &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} \sin(x-x')\, e^{-\frac{x'^2}{2w^2}}\, dx' \\
&= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} (\sin x\cos x' - \cos x\sin x')\, e^{-\frac{x'^2}{2w^2}}\, dx' \\
&= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} \sin x\cos x'\, e^{-\frac{x'^2}{2w^2}}\, dx' - \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} \cos x\sin x'\, e^{-\frac{x'^2}{2w^2}}\, dx'.
\end{aligned} \tag{A.78}$$

Note that $\sin x'$ is an odd function of x' while $e^{-\frac{x'^2}{2w^2}}$ is an even function of x'. Therefore, the second integral evaluates to 0, since the portion from $-\infty$ to 0 has the same magnitude but opposite sign as the portion from 0 to ∞. Thus, we are left with,

$$\begin{aligned}\widehat{f}(x,w) &= \frac{1}{w\sqrt{2\pi}}\int_{-\infty}^{\infty}\sin x\cos x'\,e^{-\frac{x'^2}{2w^2}}\,dx' \\ &= \frac{\sin x}{w\sqrt{2\pi}}\int_{-\infty}^{\infty}\cos x'\,e^{-\frac{x'^2}{2w^2}}\,dx' \\ &= \frac{2\sin x}{w\sqrt{2\pi}}\int_{0}^{\infty}\cos x'\,e^{-\frac{x'^2}{2w^2}}\,dx'.\end{aligned} \tag{A.79}$$

The final line follows, since $\cos x'$ and $e^{-\frac{x'^2}{2w^2}}$ are both even functions of x'. Gradshteyn and Ryzhik provide a solution for the integral [233, eq. 3.896-4], where $\beta = \frac{1}{2w^2}$ and $b = 1$,

$$\begin{aligned}\widehat{f}(x,w) &= \frac{2\sin x}{w\sqrt{2\pi}}\int_{0}^{\infty}\cos bx'\,e^{-\beta x'^2}\,dx' \\ &= \frac{2\sin x}{w\sqrt{2\pi}}\left(\frac{1}{2}\sqrt{\frac{\pi}{\beta}}\,e^{-\frac{b^2}{4\beta}}\right) \\ &= \frac{\sin x}{w\sqrt{2}}\sqrt{2w^2}\,e^{-\frac{2w^2}{4}} \\ &= \sin x\,e^{-\frac{w^2}{2}}.\end{aligned} \tag{A.80}$$

□

A.3.8 Squaring

Proposition A.27. *Let $f(x) = x^2$. Using the Gaussian band-limiting kernel, $\widehat{f}(x,w) = x^2 + w^2$.*

Proof. Starting with equation A.2,

$$\begin{aligned}\widehat{f}(x,w) &= \frac{1}{w\sqrt{2\pi}}\int_{-\infty}^{\infty}(x-x')^2\,e^{-\frac{x'^2}{2w^2}}\,dx' \\ &= \frac{1}{w\sqrt{2\pi}}\int_{-\infty}^{\infty}(x^2 - 2xx' + x'^2)\,e^{-\frac{x'^2}{2w^2}}\,dx' \\ &= \frac{x^2}{w\sqrt{2\pi}}\int_{-\infty}^{\infty}e^{-\frac{x'^2}{2w^2}}\,dx' - \frac{2x}{w\sqrt{2\pi}}\int_{-\infty}^{\infty}x'\,e^{-\frac{x'^2}{2w^2}}\,dx' + \frac{1}{w\sqrt{2\pi}}\int_{-\infty}^{\infty}x'^2\,e^{-\frac{x'^2}{2w^2}}\,dx' \\ &= x^2 - \frac{2x}{w\sqrt{2\pi}}\int_{-\infty}^{\infty}x'\,e^{-\frac{x'^2}{2w^2}}\,dx' + \frac{1}{w\sqrt{2\pi}}\int_{-\infty}^{\infty}x'^2\,e^{-\frac{x'^2}{2w^2}}\,dx'.\end{aligned} \tag{A.81}$$

Note that, since x' and $e^{-\frac{x'^2}{2w^2}}$ are odd and even functions of x', respectively, their product is an odd function of x'. Thus, the first integral evaluates to 0, since the portion from $-\infty$ to 0 has the same magnitude but opposite sign as the portion from 0 to ∞. Since x'^2 is an even function of x', the integrand in the second integral is also an even function of x' and evaluates to twice the integral from 0 to ∞. Thus, we are left with,

$$\widehat{f}(x,w) = x^2 + \frac{2}{w\sqrt{2\pi}}\int_{0}^{\infty}x'^2\,e^{-\frac{x'^2}{2w^2}}\,dx'. \tag{A.82}$$

Gradshteyn and Ryzhik provide a solution for this integral [233, eq. 3.461-2], with $p = \frac{1}{2w^2}$ and $n = 1$:

$$\begin{aligned}\widehat{f}(x,w) &= x^2 + \frac{2}{w\sqrt{2\pi}} \int_0^\infty x'^2\, e^{-px'^2}\, dx' \\ &= x^2 + \frac{2}{w\sqrt{2\pi}} \left(\frac{1}{4p} \sqrt{\frac{\pi}{p}} \right) \\ &= x^2 + \frac{1}{w\sqrt{2\pi}} \frac{2w^2}{2} \sqrt{2\pi w^2} \\ &= x^2 + w^2. \qquad \square\end{aligned} \tag{A.83}$$

A.3.9 Step Function

Proposition A.28. *Let*

$$f(x) = step(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise.} \end{cases} \tag{A.84}$$

Then $\widehat{f}(x,w) = \frac{1}{2} + \frac{1}{2} \operatorname{erf}\left(\frac{x}{w\sqrt{2}}\right)$.

Proof. Starting with equation A.2,

$$\begin{aligned}\widehat{f}(x,w) &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x-x')\, e^{-\frac{x'^2}{2w^2}}\, dx' \\ &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{\infty} step(x-x')\, e^{-\frac{x'^2}{2w^2}}\, dx'.\end{aligned} \tag{A.85}$$

From the definition of f, we can see that $step(x-x') = 0$ when $x < x'$ and $step(x-x') = 1$ when $x > x'$. Thus, we can simplify the integral without reference to the step function:

$$\begin{aligned}\widehat{f}(x,w) &= \frac{1}{w\sqrt{2\pi}} \int_{-\infty}^{x} e^{-\frac{x'^2}{2w^2}}\, dx' \\ &= \frac{1}{2} + \frac{1}{2} \operatorname{erf}\left(\frac{x}{w\sqrt{2}}\right) \qquad \text{by proposition A.4.} \qquad \square\end{aligned} \tag{A.86}$$

A.3.10 Truncation

Proposition A.29. *Let*

$$f(x) = trunc(x) = \begin{cases} \lceil x \rceil & \text{if } x \leq 0 \\ \lfloor x \rfloor & \text{otherwise.} \end{cases} \tag{A.87}$$

Then $\widehat{f}(x,w) = \widehat{floor}(x,w) - \widehat{step}(x,w) + 1$.

Proof. Starting with equation A.1, we split the integral to handle each case separately:

$$\begin{aligned}\widehat{f}(x,w) &= \int_{-\infty}^{\infty} f(x')\, e^{-\frac{(x-x')^2}{2w^2}}\, dx' \\ &= \int_{-\infty}^{0} \lceil x' \rceil\ e^{-\frac{(x-x')^2}{2w^2}}\, dx' + \int_{0}^{\infty} \lfloor x' \rfloor\ e^{-\frac{(x-x')^2}{2w^2}}\, dx' \\ &= \int_{-\infty}^{0} (\lfloor x' \rfloor + 1)\, e^{-\frac{(x-x')^2}{2w^2}}\, dx' + \int_{0}^{\infty} \lfloor x' \rfloor\ e^{-\frac{(x-x')^2}{2w^2}}\, dx' \quad \text{by proposition A.15.}\end{aligned} \tag{A.88}$$

Observe that, if $x < 0$, then $step(x) = 0$ and if $x > 0$, then $step(x) - 1 = 0$. We substitute these expressions into the first and second integrals, respectively:

$$\begin{aligned}\widehat{f}(x,w) &= \int_{-\infty}^{0} (\lfloor x' \rfloor - step(x') + 1)\, e^{-\frac{(x-x')^2}{2w^2}}\, dx' + \int_{0}^{\infty} (\lfloor x' \rfloor - step(x') + 1)\, e^{-\frac{(x-x')^2}{2w^2}}\, dx' \\ &= \int_{-\infty}^{\infty} (\lfloor x' \rfloor - step(x') + 1)\, e^{-\frac{(x-x')^2}{2w^2}}\, dx'.\end{aligned} \tag{A.89}$$

Thus, if $g(x) = \lfloor x \rfloor - step(x) + 1$, then $\widehat{f}(x,w) = \widehat{g}(x,w)$. From proposition A.12,

$$\widehat{f}(x,w) = \widehat{g}(x,w) = \widehat{floor}(x,w) - \widehat{step}(x,w) + 1. \tag{A.90}$$

□

Bibliography

[1] Brendan Sinclair. Gaming will hit $91.5 billion this year. http://www.gamesindustry.biz/articles/2015-04-22-gaming-will-hit-usd91-5-billion-this-year-newzoo. Accessed: 04/2016.

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

[3] Mike Melanson. Google's YouTube uses FFmpeg. https://multimedia.cx/eggs/googles-youtube-uses-ffmpeg/. Accessed: 2017-06-07.

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

[5] Mischa Wanek-Libman. FRA, FTA award $197 million in grants for PTC on passenger rail. http://www.rtands.com/index.php/cs/fra-fta-award-197-million-in-grants-for-ptc-on-passenger-rail.html. Accessed: 07/2017.

[6] Cadie Thompson. Tesla enhanced autopilot software update adds features. http://www.businessinsider.com/tesla-enhanced-autopilot-software-update-adds-features-2017-6. Accessed: 07/2017.

[7] Felix Salmon and Jon Stokes. Algoritms take control of wall street. *WIRED*, Dec 2010. https://www.wired.com/2010/12/ff_ai_flashtrading/, accessed 06/2017.

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

[9] 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.

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

[11] Erich Hausmann and Edgar P. Slack. *Physics*. D. Van Nostrand Company, New York, NY, 1941.

[12] Fred Glover and Gary A. Kochenberger, editors. *Handbook of Metaheuristics*. Kluwer Academic Publishers, 2003.

[13] Sean Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013. Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[14] Pavel Polityuk, Oleg Vukmanovic, and Stephen Jewkes. Ukraine's power outage was a cyber attack: Ukrenergo. http://www.reuters.com/article/us-ukraine-cyber-attack-energy-idUSKBN1521BA, accessed 06/2017, Jan 2017.

[15] Total passengers on U.S. airlines and foreign airlines U.S. flights increased 1.3% in 2012 from 2011. http://www.rita.dot.gov/bts/press_releases/bts016_13. Accessed: 2016-02-05.

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

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

[18] Lingjia Tang, Jason Mars, Xiao Zhang, Robert Hagmann, Robert Hundt, and Eric Tune. Optimizing Google's warehouse scale computers: The NUMA experience. In *International Symposium on High Performance Computer Architecture*, HPCA '13, pages 188–197, 2013.

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

[20] Richard Hawkins, Tim Kelly, John Knight, and Patrick Graydon. A new approach to creating clear safety arguments. In *Advances in Systems Safety*, pages 3–23. Springer, 2011.

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

[22] Fabio Pellacini, Kiril Vidimče, Aaron Lefohn, Alex Mohr, Mark Leone, and John Warren. Lpics: A hybrid hardware-accelerated relighting engine for computer cinematography. *ACM Transactions on Graphics*, 24(3), July 2005.

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

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

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

[26] Milton Abramowitz and Irene A. Stegun. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. ninth Dover printing, tenth GPO printing edition, 1964.

[27] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[28] Mark Harman. The current state and future of search based software engineering. In *International Conference on Software Engineering*, pages 342–357, 2007.

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

[30] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.*, 27(3):477–526, 2005.

[31] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78, 1998.

[32] Thomas Ball and James R. Larus. Branch prediction for free. In *Programming Language Design and Implementation*, pages 300–313, 1993.

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

[34] Dongsun Kim, Xinming Wang, Sunghun Kim, Andreas Zeller, Shing-Chi Cheung, and Sooyong Park. Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts. *IEEE Transactions on Software Engineering*, 37(3):430–447, 2011.

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

[36] Yaochu Jin and J. Branke. Evolutionary optimization in uncertain environments-a survey. *Transactions on Evolutionary Computation*, 9(3):303–317, June 2005.

[37] Eckart Zitzler, Lothar Thiele, Marco Laumanns, Carlos M Fonseca, and Viviane Grunert Da Fonseca. Performance assessment of multiobjective optimizers: An analysis and review. *IEEE Transactions on Evolutionary Computation*, 7(2):117–132, 2003.

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

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

[40] Pramod Kumar Jha, Chander Shekhar, and L Sobhan 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.

[41] Walter J Doherty and Arvind J Thadhani. The economic value of rapid response time. *IBM Report*, 1982.

[42] Ryusuke Villemin, Christophe Hery, Sonoko Konishi, Takahito Tejima, Ryusuke Villemin, and David 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.

[43] Urs Hoelzle and Luiz Andre Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[44] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of java collections classes. In *International Conference on Software Engineering*, ICSE '16. ACM, 2016.

[45] Swagath Venkataramani, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Approximate computing and the quest for computing efficiency. In *Design Automation Conference*, DAC '15, pages 120:1–120:6, 2015.

[46] Jie Han and Michael Orshansky. Approximate computing: An emerging paradigm for energy-efficient design. In *European Test Symposium*, ETS '13, pages 1–6, 2013.

[47] Rangharajan Venkatesan, Amit Agarwal, Kaushik Roy, and Anand Raghunathan. MACACO: Modeling and analysis of circuits for approximate computing. In *International Conference on Computer-Aided Design (ICCAD)*, pages 667–673, Nov 2011.

[48] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 309–328, 2014.

[49] Martin Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *International Conference on Supercomputing*, ICS '06, pages 324–334, 2006.

[50] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. *ACM Transactions on Computer Systems (TOCS)*, 32(3), September 2014.

[51] Z. Yang, J. Han, and F. Lombardi. Transmission gate-based approximate adders for inexact computing. In *International Symposium on Nanoscale Architectures*, NANOARCH '15, pages 145–150, 2015.

[52] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Joint Meeting of the European Software Engineering Conference and the Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, 2011.

[53] Jongsun Park, Jung Hwan Choi, and Kaushik Roy. Dynamic bit-width adaptation in DCT: An approach to trade off image quality and computation energy. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(5):787–793, 2010.

[54] 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.

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

[56] Peter 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.

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

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

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

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

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

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

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

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

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

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

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

[68] Gordon Fraser and Andrea Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

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

[70] Sheeva Afshan, Phil McMinn, and Mark Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *International Conference on Software Testing, Verification and Validation*, pages 352–361, March 2013.

[71] John W. Backus, Robert J. Beeber, Sheldon Best, Richard Goldberg, L. Mitchell Haibt, Harlan L. Herrick, Robert A. Nelson, David Sayre, Peter B. Sheridan, H. Stern, Irving Ziller, Robert A. Hughes, and Roy Nutt. The FORTRAN automatic coding system. In *Western Joint Computer Conference: Techniques for Reliability*, pages 188–198. ACM, 1957.

[72] William M McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965.

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

[74] Fan Wu, Westley Weimer, Mark Harman, Yue Jia, and Jens Krinke. Deep parameter optimisation. In *Genetic and Evolutionary Computation Conference*, pages 1375–1382, 2015.

[75] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[76] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[77] John Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):97–113, 2003.

[78] Gary A. Kildall. A unified approach to global program optimization. In *Principles of programming languages*, pages 194–206. ACM Press, 1973.

[79] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Transactions Programming Languages and Systems*, 19(6):1053–1084, November 1997.

[80] Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for reduced code space using genetic algorithms. *SIGPLAN Notices*, 34(7):1–9, May 1999.

[81] 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.

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

[83] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *ACM Sigplan Notices*, volume 41, pages 394–403. ACM, 2006.

[84] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 305–316, 2013.

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

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

[87] Rajiv Gupta, Eduard Mehofer, and Youtao Zhang. *The Compiler Design Handbook: Optimizations and Machine Code Generation*, chapter Profile Guided Compiler Optimizations, pages 143–174. CRC Press, 2002.

[88] Gilbert Joseph Hansen. *Adaptive Systems for the Dynamic Run-time Optimization of Programs.* PhD thesis, Pittsburgh, PA, USA, 1974.

[89] Michael Paleczny, Christopher Vick, and Cliff Click. The java HotSpot TM server compiler. In *Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium*, volume 1. USENIX Association, 2001.

[90] Edmund K Burke and Graham Kendall, editors. *Search methodologies*. Springer, 2005.

[91] David G. Luenberger. *Linear and Nonlinear Programming*. Kluwer Academic Publishers, second edition, 2004.

[92] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[93] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[94] Federica Sarro and Kalyanmoy Deb, editors. *Search Based Software Engineering - 8th International Symposium, SSBSE 2016*, volume 9962 of *Lecture Notes in Computer Science*, 2016.

[95] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[96] Michael Sipser. *Introduction to the Theory of Computation*. Second edition. 1997.

[97] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2008.

[98] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[99] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *Transactions on Software Engineering (TSE)*, 41(5):507–525, 2015.

[100] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. AK Peters / CRC Press, 3rd edition, 2008.

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

[102] FE Nicodemus, JC Richmond, JJ Hsia, IW Ginsberg, and T Limperis. Geometric considerations and nomenclature for reflectance, volume 161 of monograph. *National Bureau of Standards (US)*, 1977.

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

[104] Claude E. Shannon. Communication in the presence of noise. *Proceedings of the Institute of Radio Engineers*, 37(1):10–21, Jan 1949.

[105] R.N. Bracewell. *The Fourier Transform and Its Applications*. McGraw-Hill series in electrical engineering. McGraw-Hill, 2nd edition, 1986.

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

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

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

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

[110] Steven W. Smith. *The scientist and engineer's guide to digital signal processing*. California Technical Publishing, San Diego, California, 1997.

[111] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Pearson Prentice Hall, third edition, 2008.

[112] Paul S. Heckbert. Filtering by repeated integration. *SIGGRAPH Computer Graphics*, 20(4), Aug 1986.

[113] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[114] John R Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Principles of programming languages*, pages 177–189, 1983.

[115] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2003.

[116] Wolfgang Heidrich, Philipp Slusallek, and Hans-Peter Seidel. Sampling procedural shaders using affine arithmetic. *ACM Transactions on Graphics*, 17(3):158–176, July 1998.

[117] Randi J. Rost and Bill Licea-Kane. *OpenGL Shading Language*. Addison-Wesley Professional, 3rd edition, 2009.

[118] Michael P Eckert and Andrew P Bradley. Perceptual quality metrics applied to still image compression. *Signal processing*, 70(3):177–200, 1998.

[119] A. M. Eskicioglu and P. S. Fisher. Image quality measures and their performance. *IEEE Transactions on Communications*, 43(12), Dec 1995.

[120] Zhou Wang, Alan C. Bovik, Hamid R. Sheikh, and Eero P. Simoncelli. Image quality assessment: From error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, 2004.

[121] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72, 2012.

[122] Pitchaya Sitthi-Amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics*, 30(6), Dec 2011.

[123] Rui Wang, Xianjin Yang, Yazhen Yuan, Wei Chen, Kavita Bala, and Hujun Bao. Automatic shader simplification using surface signal approximation. *ACM Transactions on Graphics*, 33(6), Nov 2014.

[124] Lei Yang, Diego Nehab, Pedro V. Sander, Pitchaya Sitthi-amorn, Jason Lawrence, and Hugues Hoppe. Amortized supersampling. *ACM Transactions on Graphics*, 28(5), Dec 2009.

[125] Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. Procedural noise using sparse gabor convolution. *ACM Transactions on Graphics*, 28(3):54:1–54:10, Jul 2009.

[126] Pitchaya Sitthi-amorn, Nicholas Modly, Westley Weimer, and Jason Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics*, 30(6):152:1–152:12, 2011.

[127] Edgar Velázquez-Armendáriz, Shuang Zhao, Miloš Hašan, Bruce Walter, and Kavita Bala. Automatic bounding of programmable shaders for efficient global illumination. *ACM Transactions on Graphics*, 28(5):142:1–142:9, December 2009.

[128] Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. Dig: A dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology*, 23(4), September 2014.

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

[130] Alexander Reshetov. Morphological antialiasing. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 109–116, 2009.

[131] Kavita Bala, Bruce Walter, and Donald P. Greenberg. Combining edges and points for interactive high-quality rendering. *ACM Transactions on Graphics*, 22(3):631–640, Jul 2003.

[132] Matthäus G. Chajdas, Morgan McGuire, and David Luebke. Subpixel reconstruction antialiasing for deferred shading. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 15–22, 2011.

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

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

[135] J Whitney and P Delforge. Data center efficiency assessment. *Issue Paper, Aug*, 2014.

[136] Renewable energy google green. https://www.google.com/green/energy/. Accessed: 2016-12-10.

[137] Irene Manotas, Lori Pollock, and James Clause. SEEDS: A software engineer's energy-optimization decision support framework. In *International Conference on Software Engineering*, ICSE '14, pages 503–514, 2014.

[138] 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. *IEEE Journal of Solid-State Circuits*, 37(11):1441–1447, 2002.

[139] J. Mars, Lingjia Tang, K. Skadron, M.L. Soffa, and R. Hundt. Increasing utilization in modern warehouse-scale computers using bubble-up. *IEEE Micro*, 32(3):88–99, May 2012.

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

[141] Asela Gunawardana and Guy Shani. A survey of accuracy evaluation metrics of recommendation tasks. *Journal of Machine Learning Research*, 10(Dec):2935–2962, 2009.

[142] John D. Cutnell and Kenneth W. Johnson. *Physics*. John Wiley & Sons, Inc., 3rd edition, 1995.

[143] 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.

[144] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[145] AJ KleinOsowski and David J Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *IEEE Computer Architecture Letters*, 1(1):7–7, 2002.

[146] Rajat Todi. SPEClite: using representative samples to reduce SPEC CPU2000 workload. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 15–23. IEEE, 2001.

[147] Thomas F Wenisch, Roland E Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C Hoe. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, 2006.

[148] Andreas Merkel and Frank Bellosa. Balancing power consumption in multiprocessor systems. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 403–414. ACM, 2006.

[149] 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.

[150] Lev Mukhanov, Pavlos Petoumenos, Zheng Wang, Nikos Parasyris, Dimitrios S Nikolopoulos, Bronis R De Supinski, and Hugh Leather. Alea: a fine-grained energy profiling tool. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(1):1, 2017.

[151] Saemundur O. Haraldsson and John R. Woodward. Genetic improvement of energy usage is only as reliable as the measurements are accurate. In *Genetic and Evolutionary Computation Conference*, GECCO '15, pages 821–822, 2015.

[152] Hardware and offical latest firmware for emontx v3.4.x. https://openenergymonitor.org/emon/modules/emonTxV3. Accessed: 2017-06-07.

[153] Corey Gough, Ian Steiner, and Winston Saunders. *Energy efficient servers: blueprints for data center optimization*. Apress, 2015.

[154] Eric Schulte, Jonathan DiLorenzo, Stephanie Forrest, and Westley Weimer. Automated repair of binary and assembly programs for cooperating embedded devices. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 317–328, 2013.

[155] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 639–652, 2014.

[156] Brad L. Miller, Brad L. Miller, David E. Goldberg, and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9:193–212, 1995.

[157] Terry Jones and Stephanie Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *International Conference on Genetic Algorithms*, ICGA '95, pages 184–192, 1995.

[158] Dustin McIntire, Thanos Stathopoulos, Sasank Reddy, Thomas Schmidt, and William J. Kaiser. Energy-efficient sensing with the low power, energy aware processing (LEAP) architecture. *ACM Transactions on Embedded Computing Systems*, 11(2):27:1–27:36, July 2012.

[159] Tarsila Bessa, Pedro Quintão, Michael Frank, and Fernando Magno Quintão Pereira. JetsonLeap: A framework to measure energy-aware code optimizations in embedded and heterogeneous systems. In *Brazilian Symposium on Programming Languages*, SBLP '16, pages 16–30, 2016.

[160] Arduino - software. https://www.arduino.cc/en/Main/Software. Accessed: 2017-06-07.

[161] F.T. Ulaby and M.M. Maharbiz. *Circuits*. National Technology & Science Press, 2nd edition, 2013.

[162] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[163] Raspberry pi - teach, learn, and make with raspberry pi. https://www.raspberrypi.org/. Accessed: 2017-06-17.

[164] William B. Langdon, Justyna Petke, and Bobby R. Bruce. Optimising quantisation noise in energy measurement. In *Parallel Problem Solving from Nature*, PPSN XIV, pages 249–259, 2016.

[165] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *International Symposium on Computer Architecture*, ISCA '15, pages 158–169, 2015.

[166] Thomas Ackling, Bradley Alexander, and Ian Grunert. Evolving patches for software repair. In *Genetic and Evolutionary Computation Conference*, GECCO '11, pages 1427–1434, 2011.

[167] Claire Le Goues, Stephanie Forrest, and Westley Weimer. Representations and operators for improving evolutionary software repair. In *Genetic and Evoluationary Computation Conference*, GECCO '12, pages 959–966, 2012.

[168] Westley Weimer, Zachary P Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *International Conference on Automated Software Engineering*, ASE '13, pages 356–366, 2013.

[169] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, Apr 2002.

[170] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '99, pages 253–267, 1999.

[171] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[172] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[173] Yaroslav Kemnits. Hardcore henry – using blender for VFX. https://www.blender.org/news/hardcore-henry-using-blender-for-vfx/. Accessed: 2017-06-07.

[174] Videolan developers - vlc media player - videolan. http://www.videolan.org/developers/vlc.html. Accessed: 2017-06-07.

[175] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, 2005.

[176] James M. Kasson and Wil Plouffe. An analysis of selected computer interchange color spaces. *ACM Transactions on Graphics*, 11(4):373–405, October 1992.

[177] H. Jacobson, P. Bose, Zhigang Hu, A. Buyuktosunoglu, V. Zyuban, R. Eickemeyer, L. Eisen, J. Griswell, D. Logan, Balaram Sinharoy, and J. Tendler. Stretching the limits of clock-gating efficiency in server-class processors. In *Symposium on High-Performance Computer Architecture*, HPCA '05, pages 238–242, 2005.

[178] Shih-Lien Lu. Speeding up processing with approximation circuits. *IEEE Computer*, 37(3):67–73, 2004.

[179] Vaibhav Gupta, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy. Low-power digital signal processing using approximate adders. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32(1):124–137, January 2013.

[180] Nilanjan Banerjee, Georgios Karakonstantis, and Kaushik Roy. Process variation tolerant low power DCT architecture. In *Proceedings of the conference on Design, automation and test in Europe*, pages 630–635. EDA Consortium, 2007.

[181] Ye Tian, Qian Zhang, Ting Wang, Feng Yuan, and Qiang Xu. ApproxMA: Approximate memory access for dynamic precision scaling. In *Great Lakes Symposium on VLSI*, GLSVLSI '15, pages 337–342, 2015.

[182] O. Sarbishei and K. Radecka. Analysis of precision for scaling the intermediate variables in fixed-point arithmetic circuits. In *International Conference on Computer-Aided Design*, ICCAD '10, pages 739–745, 2010.

[183] Martin Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '07, pages 369–386, 2007.

[184] Mario Linares-Vásquez, Gabriele Bavota, Carlos Eduardo Bernal Cárdenas, Rocco Oliveto, Massimiliano Di Penta, and Denys Poshyvanyk. Optimizing energy consumption of GUIs in Android apps: A multi-objective approach. In *Joint Meeting of the European Software Engineering Conference and the Foundations of Software Engineering*, ESEC/FSE '15, pages 143–154, 2015.

[185] Bobby R. Bruce, Justyna Petke, and Mark Harman. Reducing energy consumption using genetic improvement. In *Genetic and Evolutionary Computation Conference*, GECCO '15, pages 1327–1334, 2015.

[186] Mohammad Alshraideh and Leonardo Bottaci. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification and Reliability*, 16(3):175–203, 2006.

[187] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation*, pages 209–224, 2008.

[188] Joe W. Duran and S.C. Ntafos. An evaluation of random testing. *Software Engineering, IEEE Transactions on*, SE-10(4):438–444, July 1984.

[189] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the Association for Computing Machinery*, 33(12):32–44, 1990.

[190] Gordon Fraser and Andreas Zeller. Exploiting common object usage in test case generation. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 80–89, 2011.

[191] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Foundations of Software Engineering*, pages 179–190. ACM, 2015.

[192] Raymond P.L. Buse and Westley Weimer. Learning a metric for code readability. *IEEE Trans. Software Eng.*, November 2009.

[193] James L. Elshoff and Michael Marcotty. Improving computer program readability to aid modification. *Commun. ACM*, 25(8):512–521, 1982.

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

[195] Mouloud Arab. Enhancing program comprehension: Formatting and documenting. *SIGPLAN Notices*, 27(2):37–46, February 1992.

[196] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, second edition, 2005.

[197] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill Higher Education, 1997.

[198] AE Glennie. The automatic coding of an electronic computer. *unpublished lecture notes dated December*, 14:15, 1952.

[199] Donald E Knuth and Luis Trabb Pardo. The early development of programming languages. *A history of computing in the twentieth century*, pages 197–273.

[200] Alan J Perlis and Klaus Samelson. Preliminary report: international algebraic language. *Communications of the ACM*, 1(12):8–22, 1958.

[201] M.H. Halstead. *Elements of Software Science*. Elsevier, New York, 1977.

[202] Thomas J. McCabe. A complexity measure. *IEEE Trans. Software Eng.*, 2(4):308–320, 1976.

[203] Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984.

[204] Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman. Program indentation and comprehensibility. *Communications of the ACM*, 26(11):861–867, November 1983.

[205] Daryl Posnett, Abram Hindle, and Premkumar T. Devanbu. A simpler model of software readability. In *Mining Software Repositories*, pages 73–82, 2011.

[206] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*, 37(5):649–678, 2011.

[207] Dirk Beyer, Adam J Chlipala, Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th International Conference on Software Engineering*, pages 326–335. IEEE Computer Society, 2004.

[208] Nikolai Tillmann and Jonathan de Halleux. Pex-white box test generation for .NET. In *International Conference on Tests and Proofs (TAP'08)*, pages 134–153, 2008.

[209] Richard A Eyre-Todd. The detection of dangling references in C++ programs. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2(1-4):127–134, 1993.

[210] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1–3):35–45, 2007.

[211] Claire Le Goues and Westley Weimer. Specification mining with few false positives. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 292–306, 2009.

[212] Gordon Fraser and Andreas Zeller. Generating parameterized unit tests. In *International Symposium on Software Testing and Analysis*, pages 364–374, 2011.

[213] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software*, 126:57–84, 2017.

[214] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, Reading, MA, USA, 1996.

[215] Ermira Daka, Jose Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE '15, pages 107–118, 2015.

[216] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 14(2):1137–1145, 1995.

[217] R. F. Flesch. A new readability yardstick. *Journal of Applied Psychology*, 32:221–233, 1948.

[218] R. Gunning. *The Technique of Clear Writing*. McGraw-Hill International Book Co., New York, 1952.

[219] G. Harry McLaughlin. SMOG grading — a new readability. *Journal of Reading*, May 1969.

[220] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[221] Arie van Deursen, Leon M. F. Moonen, and Gerard van den Bergh, A an Kok. Refactoring test code. Technical report, Centre for Mathematics and Computer Science (CWI), 2001.

[222] M. Harman, Sung Gon Kim, K. Lakhotia, P. McMinn, and Shin Yoo. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In *International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 182–191, 2010.

[223] Sai Zhang. Practical semantic test simplification. In *International Conference on Software Engineering (ICSE)*, pages 1173–1176, 2013.

[224] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. Efficient unit test case minimization. In *International Conference on Automated Software Engineering (ASE)*, pages 417–420, 2007.

[225] Yong Lei and James H. Andrews. Minimization of randomized unit test cases. In *International Symposium on Software Reliability Engineering (ISSRE)*, pages 267–276, 2005.

[226] Sai Zhang, Cheng Zhang, and Michael D. Ernst. Automated documentation inference to explain failed tests. In *International Conference on Automated Software Engineering (ASE)*, pages 63–72, 2011.

[227] Jifeng Xuan and Martin Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 52–63, 2014.

[228] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Foundations of Software Engineering*, ESEC/FSE 2015, pages 38–49, 2015.

[229] Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu, Michael Pradel, and Yue Luo. Nomen est omen: Exploring and exploiting similarities between argument and parameter names. In *International Conference on Software Engineering*, pages 1063–1073. IEEE, 2016.

[230] Jonathan Dorn, Connelly Barnes, Jason Lawrence, and Westley Weimer. Towards automatic band-limited procedural shaders. *Computer Graphics Forum*, 34(7):77–87, 2015.

[231] Ermira Daka, Jose Campos, Jonathan Dorn, Gordon Fraser, and Westley Weimer. Generating readable unit tests for Guava. In *Symposium on Search Based Software Engineering*, pages 235–241, 2015.

[232] Jr. Thomas, George B. and Ross L. Finney. *Calculus and analytic geometry*. Addison-Wesley, ninth edition, 1998.

[233] I. S. Gradshteyn and I. M. Ryzhik. *Table of integrals, series, and products*. Academic Press, Amsterdam, fifth edition, 1994.