THE UNIVERSITY OF CHICAGO

A COMPILER-BASED ONLINE ADAPTIVE OPTIMIZER

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY

KAVON FAR FARVARDIN

CHICAGO, ILLINOIS
DECEMBER 2020

To my cats Jiji and Pumpkin.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to thank my advisor John H. Reppy for his ample support, kindness, and advice over the past six years. Thanks also to the rest of my dissertation committee, Ravi Chugh, Hal Finkel, and Sanjay Krishnan, for their time and feedback. I am also grateful for my summer internships with Hal Finkel and Simon Peyton Jones that greatly expanded my horizons.

This dissertation would not exist without the social and intellectual support from my friends, family, and colleagues: Amna, Andrew M., Aritra, Brian, Charisee, Connor, Cyrus, Elnaz, The Happy Hour Gang, The Ionizers, Jean, Joe, Justina, Kartik, Lamont, The Ross Street Crew, Saeid, Sheba, Suhail, Will, and *numerous* others. In ways both subtle and overt, many of them have helped me grow as a person, persevere through difficult times, or both; I am incredibly lucky to have had them in my life.

Abstract

The primary reason for performing compiler optimizations before running the program is that they are "free" in the sense that the analysis and transformations involved in applying the optimizations do not contribute any overhead to program execution. For optimizations that are inherently or provably beneficial, there is no reason to perform them dynamically at runtime (or *online*). But many optimizations are situational or speculative, because their profitability depends on assumptions about dynamic information that cannot be fully determined statically.

Today, the main use of online adaptive optimization has been for language implementations that would otherwise have high baseline overheads. This overhead is normally due to the use of an interpreter or a large degree of dynamically-determined behavior (*e.g.*, dynamic types). The optimizations applied in online adaptive systems are those that, if the profiling data is a good predictor of future behavior, will certainly be profitable according to the optimization's cost model. But, there are many compiler optimizations whose cost models are unreliable!

In this dissertation, I describe an adaptive optimization system that uses a trial-and-error search over the available compiler optimization flags to automatically tune them for the program, with respect to its dynamic environment. This LLVM-based system is unique in that it is fully online: all profiling and adaptation happens while the program is running in production, thanks to newly developed techniques to manage the cost of tuning. For the first time, users can easily take advantage of automatic tuning that is specific

to the program's workload in production, by enabling just one option when initially compiling the program. The system transparently delivers up to a 2× speedup for a number C/C++ benchmarks relative to the best optimizations available in a production-grade compiler. Furthermore, the system adapts to future workloads, because the automatic tuning is continuous.

Part I

Introduction

Chapter 1

Motivation

The primary reason for performing compiler optimizations before running the program is that they are "free" in the sense that the analysis and transformations involved in applying the optimizations do not contribute any overhead to program execution. For optimizations that are inherently or provably beneficial, there is no reason to perform them dynamically at runtime (or *online*). But many optimizations are situational or speculative, because their profitability depends on assumptions about dynamic information that cannot be fully determined statically.

For example, unrolling a loop is situational because it is only profitable under the assumption that the loop has a large iteration bound. This assumption helps ensure that the overhead of additional code added by the optimization, which decides between entering either the unrolled or remainder loops, will be paid-off. Evidence to support such assumptions can sometimes be inferred statically, but *profile data*, which provides information about the dynamic behavior and environment of the program, is often what is relied upon when estimating whether a situational optimization will yield a profit.

As an additional example, let us consider expansive function inlining, *i.e.*, the inlining of a function that has more than one call-site. Inlining can be viewed as a speculative optimization because it specializes the body of the function to a particular call-site. This

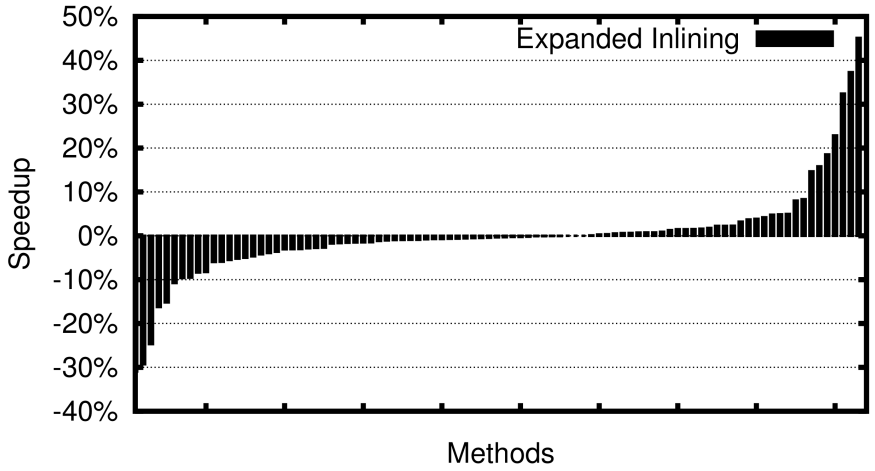


Figure 1.1: The affect of quadrupling the inlining threshold in the IBM J9 JVM compiler for the 101 hottest methods across 20 Java benchmark programs; from Lau et al. [2006].

specialization provides later intraprocedural optimization passes with information about how the arguments at that call-site will be used. The downside is that too much inlining can also degrade performance, because it increases the size of the program and causes ill effects during execution such as instruction cache misses. Compiler-writers often navigate this inlining trade-off by setting a simple threshold on the size of the callee when considering whether to inline a particular call-site. But, a simple metric such as function size does not guarantee an optimal decision. For example, Lau et al. [2006] experimented with quadrupling the default inlining threshold, which limits the size of the callee to be inlined, in the production-grade IBM J9 compiler. Figure 1.1 shows that the inlining trade-off is not optimally reasoned about with just a function size threshold. Raising the size threshold significantly benefits a handful of functions but significantly degrades another handful, with little change for the rest. Is there a better way to navigate the trade-offs when deciding whether to inline a call-site?

Cost Models All compiler optimizations rely on a cost model to account for performance trade-offs. A *cost model* predicts the net benefit of the transformation on the program's performance, signaling whether the transformation is likely to be profitable or not.

These cost models may be simple or not explicitly stated; *e.g.*, the implicit model behind eliminating useless code is that executing fewer instructions will reduce execution time.

For optimizations where an explicit cost model is needed to accurately estimate the trade-offs, such as for inlining, the models are often parameterized to rely on abstract threshold values, profile data, or both. The function-size threshold for inlining and the large loop-bound for unrolling are examples of simple but explicit cost models. Profile data helps by providing concrete evidence to support or refute assumptions, such as "this loop's bound is large," because it reduces uncertainty. The trouble is that profile data can only be collected *after* compilation by executing the program.

1.1 Adaptation

A compiler is a program translator and does not execute the program like an interpreter does. Compilers that are run prior to program execution, or ahead-of-time (AOT), cannot access profile data without additional help from the programmer. The programmer must set-up a build system that compiles the program once and performs a profiling run on a test workload. Then, the profiling data is fed back into the compiler again to recompile the program. This feedback loop of profile data is at the heart of *adaptive optimization* because it allows the compiler to adapt its optimizations based on information it is not cognizant of statically.

In the absence of profile data to guide cost models, compilers make either pessimistic or optimistic decisions, such as disabling an optimization or using heuristics to make assumptions about the typical program. Kennedy and Allen [2002, Chapter 5.11] discuss an example of this phenomenon for loop vectorization, where an optimistic assumption is used when faced with statically unknown loop bounds during cost modeling:

> When loop bounds are symbolic, trading off loop lengths versus strides and

other parameters is very difficult. With no additional input from the programmer, compilers must generally assume that all loops with unknown bounds are long enough for efficient vector execution.

Pessimism and optimism are two general approaches to dealing with risk, which in our case takes the form of the compiler possibly yielding a program with poor performance. As an alternative, the adaptive approach is to get a clearer picture of that risk through experimentation, in the form of actually running the program, and then basing the optimization's decisions on the information gained. The fundamental downside of using purely static optimization instead of adaptive optimization is that less information is known about the program's runtime behavior or environment to make smart decisions [Adve et al., 1997].

Online Adaptive Optimization Adaptive systems that operate *online* (*i.e.*, during program execution) can take advantage of up-to-the-second profile information and quickly react to changes in a program's workload or environment. The key advantage of performing *online adaptive optimization* (OAO) is that the optimizer can directly obtain fresh, runtime-only information instead of making ill-informed optimization decisions.

A number of implementations of programming languages such as SELF [Ungar and Smith, 1987], JAVA [Gosling et al., 1996], and JAVASCRIPT [ECMA, 1997] have been very successful in applying online adaptive optimization [Gal et al., 2009; Hölzle, 1994; Kotzmann et al., 2008]. These language implementations employ online adaptive optimization in the form of a profiler that gathers and analyzes data for a dynamic, or just-in-time (JIT), compiler in order to make better optimization decisions. Online adaptive optimization is not just an idea within the ivory tower of academia: it is a crucial technique for language implementations used by everyday people [Arnold et al., 2005; Aycock, 2003]. For example, all major web browsers, such as Firefox, Safari, and Chromium, rely on online adap-

tive optimization in their runtime system to accelerate their JAVASCRIPT interpreter [Gal et al., 2009; Google; Pizlo, 2014, 2016]. The runtime systems in these web browsers are so efficient at optimizing JAVASCRIPT that they have been repackaged and are widely used to run server and desktop applications, too [OpenJS Foundation, 2020a,b].

1.2 Trial and Error

Today, the main use of online adaptive optimization (OAO) has been for language implementations that would otherwise have high baseline overheads. This overhead is normally due to the use of an interpreter or a large degree of dynamically-determined behavior (*e.g.*, dynamic types). The optimizations applied in online adaptive systems are those that, if the profiling data is a good predictor of future behavior, will certainly be profitable according to the optimization's cost model.

By default, the HotSpot JAVA implementation executes programs by interpreting virtual machine instructions [Paleczny et al., 2001], which reduces program start-up time but is not efficient in the long run when compared to native execution. Let us consider two examples of how the profile data is used in OAO systems like HotSpot to improve performance. First, to reduce the overhead of running an interpreted program, profile data is used to balance the cost of running the JIT compiler with the benefits of patching-in a native version of the most commonly executed parts of code [Plezbert and Cytron, 1997]. As long as the program continues to execute those regions that were compiled to native code, the cost of running the compiler will quickly pay for itself. Second, compiler optimizations are forced to be pessimistic about how the value can be used when faced with values whose types can vary dynamically. So, profile data about particular program values is used to determine whether the type of a value appears to be stable. If so, the code is customized based on that assumption [Calder et al., 1997; Chambers et al., 1989]. If the type of the value changes in the future, then the optimization may be undone. Otherwise,

the type-based customizations are at least as efficient as the previous version of the code and usually significantly better. In both examples, the cost models for these optimizations are accurate as long as the profile data is an accurate summary of future behavior.

Search Methods *What if a compiler optimization's cost model is unreliable?* As discussed earlier, the trade-offs of function inlining are complex and difficult to model optimally, even with the availability of profile data [Kulkarni et al., 2013]. Instead, compiler-writers commonly use heuristics and abstract threshold values, which are hand-tuned to perform well on a fixed set of benchmark programs. Even still, the complex interactions involved in inlining make it difficult to develop good heuristics—some compiler-writers have even likened the interactions to black magic [Peyton Jones and Marlow, 2002]. These problems with cost models in compiler optimizations extend to more than just function inlining [Stephenson et al., 2003]. Thus, a cost model may not reliably produce optimal answers, which prevents programs from performing better due to the model's overly optimistic or pessimistic assumptions.

Utilizing search is one way to overcome an unreliable or incomplete cost model. Empirical *autotuning* (a portmanteau for "automatic tuning") is a form of adaptive optimization that uses a search-based method. For example, instead of cost-modeling, we can simply experiment with all combinations of optimization decisions, profiling the program's overall performance, and pick the combination with the best rewards based on the data. Thus, instead of feeding back profile data for the purposes of informing the optimization's cost model, the data is used by an autotuner as an indicator of the overall quality of one combination of decisions.

High-performance computing applications, such as scientific simulations on supercomputers, are the main users of autotuning today [Balaprakash et al., 2018]. An example of an empirical autotuning system for C programs is Orio [Hartono et al., 2009], which tunes loop optimizations such as unrolling, interchange, and tiling. When applied to

computational kernels like sparse matrix computations and a sequence of linear algebra operations, Orio was able to find configurations for loop optimizations that match or drastically improve performance relative to both production-grade compilers and hand-tuning by expert programmers. The magnitude of improvements for some kernels tuned by Orio ranged from 26% to 277%! Thus, search methods can find significant performance improvements that are missed by the typical cost models used by compiler optimizations.

Performing a search avoids the tension between optimism and pessimism because it eliminates the risk caused by uncertainty through experimentation. But, searching creates another avenue of risk: failing to find a worthwhile performance gain for the time and machine resources spent. Since the space of possible optimization decisions can grow exponentially, autotuning can quickly become intractable, regardless of the resources available or the search technique used. In some instances, the time necessary to perform the search can add days to the deployment of software to users [Hoste et al., 2010]. Additionally, if the autotuning process relies on a test workload that does not truly represent an end-user's workload, which can be expected for large-scale applications [McFarling, 2003], the program's performance may end up worse than if tuning were not performed at all!

1.3 Goals

Empirical autotuning offers an opportunity to harness the full power of compiler optimizations, but suffers from a number of practical drawbacks:

- Developers have to create a representative input for the software to accurately simulate its end-user's usage and perform tuning on the end-user's expected hardware.
- The build and release process for the software must be augmented with search and final-compilation phases.

- Once the software is deployed, the tuning remains fixed even as the hardware and end-user's usage evolves over time.

The goal of this dissertation is to investigate whether *online* autotuning for compiler optimizations can overcome these drawbacks, while still improving the overall performance of the tuned software. Online autotuning delays the process of tuning until after the software has been deployed in production to end-users. Specifically, experimentation happens *live* on the end-user's system; no test inputs are required. The use of search is what distinguishes online autotuning from more traditional language runtime systems that feature just-in-time compilation.

Autotuning has largely been limited to high-performance computing applications and requires expert knowledge to use [Basu et al., 2013]. One of the goals of this work is to make the case that online autotuning can be made accessible to average developers and still deliver good performance. Online autotuning is not a new idea (Section 3.1), but it suffers from a number of unique challenges that have prevented broader use. Central to these challenges is the balancing of search overheads with the performance gains. For example, the search may discover badly-optimized code in a failed experiment. How do we minimize the impact of this experiment on the program while it is running in production? On the other hand, if the search discovers better-optimized code, what should be done afterwards? Specifically, for how long should the search pause to exploit the better version of the code; *e.g.*, to repay the accrued debts of the search?

Roadmap This dissertation is separated into two major parts. Part I details the existing work and challenges surrounding online adaptive optimization, with Chapter 4 defining the scope and research goals of this dissertation. Part II introduces a new online adaptive optimization system called HALO, which serves as the vehicle of research for this work. Within Part II, I detail the design and implementation of HALO in Chapters 5, 6, and 7.

Then, I evaluate HALO in Chapter 8 with a number of benchmarks and summarize the results in Chapter 9.

Chapter 2

Background

Many experts throughout the years have highlighted the need for adaptive optimization to fully access the hardware's peak performance. Griswold et al. [1996] argued that "a small, stable programming language with abstraction features that support the development of self-tuning, optimizing, easily adaptable, integrable layered systems" is needed for the the next millennium. Smith [2000] advocated for the use of online adaptive optimization with concrete examples where an online approach could better optimize programs. Smith also believes that the largest barrier to adaptive optimization is people's aversion to mutating code, because of their fear of it silently introducing bugs. Since then, dynamic code generation has become a widely-accepted technique. When envisioning the next 50 years of compiler research, Hall et al. [2009] paraphrased a private communication with researcher David Kuck who discussed the importance of adaptive optimization:

> Compiler fundamentals are well understood now, but where to apply what optimization has become increasingly difficult over the past few decades. Compilers today are set to operate with a fixed strategy (such as on a single function in a particular data context) but have trouble shifting gears when different

code is encountered in a global context (such as in any whole application).

Kuck also said, "The best hope for the future is adaptive compilation that exploits equivalence classes based on 'codelet' syntax and optimization potential. This is a deep research topic, and details are only beginning to emerge. Success could lead to dramatic performance gains and compiler simplifications while embracing new language and architecture demands."

After discussing some important terminology, this chapter will explore a few general forms of adaptive optimization, at a high-level, to help provide necessary context. Then Chapter 3 takes a deeper look at the most closely related prior works and how they differ from this work.

2.1 Terminology

There is no consensus on the terminology surrounding adaptive optimization, but we will try to form a coherent universe of definitions. Smith [2000] defines *feedback-directed optimization* in the most general sense: "any technique that alters a program based on information gathered at run time." Then, Smith focuses specifically on feedback-directed optimization that improves the performance of programs. Others such as Hansen [1974] opted to use the term "adaptive" instead of "feedback-directed," but use the term in a way consistent with Smith. According to Cooper et al. [2002], the term *adaptive compilation* refers to the specialization of the compiler's optimization pipeline to particular programs, workloads, machines, or any combination of those, to minimize some objective function; but their definition does not require profile data. In contrast, *adaptive **re**compilation* refers to a technique that uses runtime information to dynamically select heavily-used code and recompiles (and optimizes) them for better performance [Hölzle, 1994]. Throughout this dissertation, *adaptive optimization* is equivalent to *feedback-directed optimization*, despite the

fact some works consider adaptation without runtime information.

The typical goal of adaptive optimization is to minimize metrics such as cache misses, energy consumption, or the execution time of frequently executed, or *hot*, code regions. These metrics may be directly or indirectly influenced by a set of configurable parameters for which values can be chosen arbitrarily without affecting the correctness of the program.[1] I refer to such parameters as *knobs*.

Much like its physical counterpart, a knob is defined in terms of a finite space of configurations, or options, and a state which indicates exactly one element from the configuration space that is currently-selected at any point in time. A collection of knobs can be thought of as a single knob whose configuration space is the cross-product of the spaces of each individual knob in the collection.

The kinds of knobs that one can imagine influencing a program are numerous. For example, an algorithm may offer a knob with two options that change the underlying data structure used by the implementation; *e.g.*, selecting either a linked-list or an array. Changing this knob only affects the performance of the program, not its result. While the inputs or data processed by the program affects the program's performance, they are not normally considered knobs unless the values can be instantiated arbitrarily among the allowed options.

The focus of this work is on the adaptive optimization of knob configurations that control the code optimization process within a compiler. Widely-used, production-grade compilers such as GCC have over a hundred such knobs available for experts to tweak, controlling various parts of the compilation pipeline [Fursin et al., 2014]. Ordinary programmers are typically familiar with just a single knob that controls the compiler's general optimization level, *i.e.*, the `-On` command-line flag, where `n` $\in [0, 3]$ and a larger `n` tells the compiler to more aggressively optimize. But, ignoring the other knobs available

[1] Prior work in approximate computing has considered knobs that trade program correctness for performance [Sidiroglou-Douskos et al., 2011]. Such perilous knobs will not be considered here.

within a compiler can leave significant performance gains untapped. For example, using just unbiased random selection over the knobs in GCC, Fursin et al. [2014] found 79 unique combinations of knob configurations that yielded better performance than `-O3`, in some cases up to 3.75× faster, after 5000 search iterations. For the remainder of this chapter, we tour some of the major works within the realm of compiler-based adaptive optimization.

2.2 Profile-guided Compilation

One of the earliest advocates for the use of a program profile data for compilers is Knuth [1971], who conducted an eye-opening study of real-world FORTRAN programs. Knuth found that there is a disconnect between what compiler writers thought was important to optimize and the kind of code that programmers wrote in practice. Program profiles of two forms were analyzed: static and dynamic.

A *static profile* is information about the program that is obtained through static analysis; *i.e.*, analyzing the program's source code without executing it. Knuth was interested in the style of code written in FORTRAN, such as how often GOTO statements appear in the code. These statistics were intended to inform compiler writers about where to focus their attention, based on how popular various language constructs are in practice. While static analysis is limited because it does not execute the program, there are a number of analytical techniques and effective heuristics that infer dynamic information [Ball and Larus, 1993; Cousot and Cousot, 1977; Patterson, 1995; Wagner et al., 1994].

In contrast to a static profile, a *dynamic profile* is information gathered through the execution of the program about its runtime characteristics. These characteristics are typically statistics about the program's source code, such as how often a function or statement is executed. There are two overall strategies for obtaining dynamic profile data. One is sample-based profiling that periodically interrupts a program thread to examine its state,

such as the next instruction to be executed or the callers of the current function. The other is instrumentation-based profiling that relies on the injection of code in places such as function entry-points and loop-backedges to record whenever those points in the program are reached. Throughout this dissertation, discussion of an unqualified "profile" refers to a dynamic profile.

Dynamic profile data can inform both humans when writing their programs and compilers when optimizing them. *Profile-guided compilation* (PGC) describes techniques where a compiler uses profile data, which consists of information that indicates where the program spends its time, to inform the optimizations applied to the program [Smith, 2000]. Profile-guided compilation has found its footing in many widely-used compilers such as GCC and CLANG [Chen et al., 2016; Hubička, 2005]. The kinds of profile data used in PGC typically includes statistics about how frequently a control-flow edge or path within a function is executed [Ball and Larus, 1994; Ball et al., 1998]. These profiles serve many uses, such as to optimize code layout for cache locality [Pettis and Hansen, 1990], focus compilation time on the important parts of the program [Whaley, 2001], inform cost models for speculative optimizations [Bodík et al., 1998; Gupta et al., 1997; Hazelwood and Conte, 2000], and even for instruction selection [Krishnaswamy and Gupta, 2002].

2.3 Dynamic Compilation

Dynamic compilation is any technique that uses a compiler *during* program execution, and is often called just-in-time (JIT) compilation. Early JIT systems used to implement virtual machines (VMs) were compile-only in the sense that the systems compiled VM code to native code whenever control-flow reached unseen code. Plezbert and Cytron [1997] observed that it is sometimes faster to interpret the code instead, pioneering a profile-driven technique for mixed-mode execution that interleaves interpretation and native execution of the program. This adaptive technique is now widely found in lan-

guage runtime systems for JAVA and JAVASCRIPT [Gal et al., 2009; Hookway and Herdeg, 1997; Kotzmann et al., 2008; Paleczny et al., 2001]. The CoreCLR language runtime system for C# (and other languages) uses a mixed ahead-of-time and just-in-time compilation strategy rather than interpretation [Strehovský, 2019]. For a more complete overview of early JIT compilation techniques, see Aycock [2003].

The LLVM-based ClangJIT project by Finkel et al. [2019] aims to bring JIT-compilation to C++ through annotations on template definitions. This essentially turns C++ into a sort of multi-staged language, because template instantiation can be selectively delayed until runtime and specialized on dynamic values or types [Veldhuizen, 2000]. Depending on the availability of in-language profiling facilities, a multi-staged language could offer the ability to implement adaptive optimization as a library.

Code Multi-versioning In essence, code multi-versioning is a solution to the classic code selection problem, where the performance of an algorithm or piece of code has a strong input or environment dependence [Rice, 1976]. Multi-versioning generates multiple versions of the same piece of code, with each version specialized to better handle a specific situation. Compilers featuring ahead-of-time multi-versioning suffer from a combinatorial explosion of code versions that need to be generated and stored in the binary, though recent work has used profiling to help mitigate the problem [Rodriguez et al., 2016; Zhou et al., 2014].

Dynamic code multi-versioning is most frequently realized in runtime systems using a tiered JIT optimization approach. A tiered JIT consists of several levels of optimizations that are reserved for different levels of code importance, as determined by profiling for code hotness [Arnold et al., 2000]. These tiers help balance the cost of compilation with the potential code improvements seen through more aggressive optimization, since compilation happens concurrently with program execution on the same machine. Gu and Verbrugge [2008] were able to use dynamic tracking of program phases to help predict

whether a method is worth recompiling at a higher optimization level, since the time spent at a lower level is lost performance. JIT-based runtime systems also use versioning and dynamically select code based on other information about the code, such as the most common kinds of arguments to a function [Hölzle et al., 1991].

2.4 Autotuning

At its core, *autotuning* (short for "automatic tuning") is an optimization problem where the goal is to find an optimal knob configuration within a huge space of possible configurations [Naono et al., 2010]. At a high-level, we can view this problem of autotuning more formally as an instance of black-box optimization (BBO) [Audet and Hare, 2017]. The "black-box" is f, the unknown cost function (or fitness function) that takes as input a configuration θ drawn from a large space of configurations Θ and outputs the configuration's cost c.[2] The goal is to find (or "select") $\theta' \in \Theta$ such that $f(\theta')$ is minimal. What makes this goal challenging is that the cost function is assumed to be expensive to evaluate, so exhaustive search methods that guarantee global optimality are not feasible in a huge space. In autotuning, the cost function may involve the process of recompiling the application according to the configuration, followed by a number of executions under a sample workload to obtain its average running time, which is output as the cost. There are three major strategies for tackling the selection problem for autotuning, which I describe in the following paragraphs. The space of existing work in automatic tuning is huge; for a good overview of autotuning for software and compilers, see Naono et al. [2010], Balaprakash et al. [2018], and Ashouri et al. [2018].

[2] In the case of multi-objective optimization, c may be a vector or there may be multiple cost functions [Durillo and Fahringer, 2014].

Model-free Selection A wide variety of heuristic search methods have been developed to navigate arbitrary configuration spaces efficiently [Blum and Roli, 2003]. A model-free strategy focuses on sampling the configuration space by choosing configurations and executing f to determine their quality. Two major types of meta-heuristics for model-free selection are *trajectory methods* that focus on the neighborhood of their best known state (*e.g.*, hill climbing and simulated annealing [Bertsimas and Tsitsiklis, 1993]) and *evolutionary strategies* that maintain a population of high-fitness states for recombination over a series of generations (*e.g.*, genetic algorithms, neuro-evolution [Stanley and Miikkulainen, 2002]). For example, Cooper et al. [1999] used genetic algorithms to search the space of compiler flags to minimize the executable's size for embedded systems. Knijnenburg et al. [2003] applied several heuristics, such as simulated annealing and random search, to tune loop unrolling and tiling factors for a number of CPU architectures. All of these search-methods obtain convergence by assuming the cost function is deterministic and remains fixed throughout the search process. This assumption does not necessarily hold for online automatic tuning.

Model-based Selection An autotuner that uses model-based selection relies on analytical models or machine-learning to learn how to select an optimal configuration. Thus, model-based strategies avoid employing search and directly produce a tuned configuration based on the characteristics of the program or machine. For example, Cavazos et al. [2007] collected data to train a model that learns how to predict good compiler optimization flags based on the state of performance counters. This pre-trained model is then used by their system to select compiler optimizations specific to the input program. The input program is run a few times to collect the performance counter data, which is input to the model to select one good configuration. Similarly, Milepost GCC [Fursin et al., 2011] uses a model trained to predict good compiler optimization flags based on features of the input program, *e.g.*, the number of basic blocks or number of assignment instructions. A

number of additional systems are summarized by Wang and O'Boyle [2018].

Hybrid Selection The hybrid selection strategy combines search-based strategies with an analytical or machine learning model. In the typical formulation, it is a strategy that relies on a model to help focus a search over the configuration space. For example, Balaprakash et al. [2013a] train a model to learn about the cost function *during* a search to progressively learn how to filter out future poor-performing or less-useful configurations. Some formulations train the model prior to using it to focus the search [Agakov et al., 2006]. In Active Harmony, the sensitivity of each knob is modeled based on configurations already tested to better focus the search [Chung and Hollingsworth, 2004]. Pouchet et al. [2008] leverage an analytical model to focus the search over configurations of high-level loop optimizations. As a final example, OpenTuner dynamically switches between multiple heuristics during the search, using an adaptive model that learns which heuristic has been most successful recently [Ansel et al., 2014]. Their adaptive model is based on a solution for the *multi-armed bandit* (MAB) problem, where one must select among a fixed number of actions that give a stochastic reward, and the goal is to maximize the accumulated rewards over time [Sutton and Barto, 1998].

2.5 Finding Balance

The major challenge faced by all online optimization systems is reaching a *break-even point*, which is the point at which the costs accrued to reoptimize is paid off by using a code version that is better than the original [Diniz et al., 1997; Kistler, 1999]. The optimizer may not reach a break-even point for a code region if: (1) the program's use of the optimized code region ends before the accumulated debt is paid off, or (2) the optimizer is unable to produce code that is better than the baseline.

The first problem is primarily mitigated through dynamic profiling and a prediction

model that determines whether the code region is likely to remain heavily-used in the future. For example, in early versions of the influential HotSpot adaptive optimizer, a code region is predicted to be hot if instrumentation-based performance counters exceed thresholds that were determined heuristically [Paleczny et al., 2001].

To ensure profitability under the more challenging second problem, it is important for an online adaptive optimizer to focus on high-impact optimizations for hot code regions, while keeping the system's operating overhead low [Kistler, 1999]. An online system will always introduce some sort of overhead, because replacing or modifying a piece of code at runtime has a cost. In fact, an online system may replace the code multiple times, either for profiling purposes or to experiment with differently-optimized versions of the code during a search. Furthermore, these new versions of code may have much worse performance than what it replaced, adding to the costs. Thus, knowing when to stop reoptimizing is important. Vuduc et al. [2004] developed a statistical technique that estimates the probability of finding a better configuration, based on the results of an empirical trial-and-error search.

Chapter 3

Related Work

Adaptive optimization is the general technique of utilizing information obtained during the runtime of the program to modify the program for performance improvement (Section 2.1). Because of the enormous volume of existing work that satisfies such a broad definition, it is not feasible to provide a survey of the area in this dissertation. Instead, the focus of this chapter is on works most relevant to the thesis (Chapter 4). No attempt is made to categorize the related works using an objective or rigid classification system; intuition is used instead.

3.1 By Similarity

This section contains a detailed overview of prior work subjectively believed to be most close to addressing the thesis. Distinctions between the objectives of this work and the prior work are emphasized as needed.

3.1.1 Active Harmony

The Active Harmony project has gone through a number of revisions and is the most closely related work. In Hollingsworth and Keleher [1999]; Keleher et al. [1999]'s work, Harmony was positioned as a global resource management system for the coming boom in distributed computing. The goal was to support online reconfiguration and adaptation of resources to optimize along trade-offs, such as throughput versus latency, for long-running applications like databases. The architecture of Harmony used a client-server model, where the server manages the simple greedy search and adaptation options that are described in a domain-specific language. Clients use the Harmony C API to synchronize their configuration with the server.

Later on, the Harmony project became more focused on general automatic tuning for application libraries and parameters [Ţăpuş et al., 2002]. The major idea was that many libraries or algorithms implement the same functionality, but some versions are more appropriate for certain situations than others. Notable earlier work by Whaley and Dongarra [1998] in the Automatically Tuned Linear Algebra Software (ATLAS) employed this idea. ATLAS is effectively an "active library" [Veldhuizen and Gannon, 1998] that automatically generates, tests and uses matrix-multiply kernels tailored for the particular CPU it is running on.

In Ţăpuş et al. [2002]'s work, Harmony would generate an API based on a specification that wraps one or more implementations of that interface. Clients would access the libraries through the wrapper API, which would dynamically monitor the performance of the underlying library implementation and tune both the choice of library and the chosen library's tuning parameters on-the-fly. The BBO search process for an optimal configuration uses a custom variant of the Nelder-Mead simplex method [Nelder and Mead, 1965] that does not assume the function being minimized is defined or continuous. Chung and Hollingsworth [2004] later greatly improved this search technique by analyz-

ing prior configurations and focused the search by determining which parameters are the most important through the use of parameter sensitivity testing.

Offline Harmony with CHiLL Tiwari et al. [2009a] combined Active Harmony with the CHiLL polyhedral loop transformation system [Chen et al., 2008] to autotune compiler optimizations in an offline setting. The tuning search process was made parallel using the Parallel Rank Ordering algorithm [Tabatabaee et al., 2005; Tiwari et al., 2009b], where multiple clients running the same application *and* performing the same work connect to the server and experiment with different configurations in parallel as directed by the central server. Essentially, one search for a single optimal configuration is parallelized across multiple machines. They were able to improve the performance of computational kernels by 1.4x–3.6x over the Intel compiler. Tiwari et al. [2011] further evaluated their offline CHiLL-based autotuner from 2009 on a full application called SMG2000. They were able to improve SMG2000's overall performance by 27% through the tuning of the main kernel, which saw a 2.37x speedup.

Online Harmony with CHiLL Tiwari and Hollingsworth [2011] describe their extensions to Active Harmony from 2009 to allow for online autotuning with dynamic code generation and loading. The key new feature is the ability to configure one or more code servers that handle compilation requests using a standalone compiler to produce shared libraries with the new code variant. A system similar to Online Harmony called AARTS was proposed by Teodoro and Sussman [2011].

In order to use Online Harmony for tuning with code generation, the user must first extract the code they would like to tune into a separate library that can be compiled with a standalone tool. The library source code along with the tuning configuration is placed on the server and given a unique name. Clients who connect refer to a library through the session name when initializing the tuning session (Figure 3.1). When the clients begin

```
// MPI Initialization and Harmony API initialization are omitted.
if (masterClient) {
  hdef_t* hdef = ah_def_alloc(); // Create Harmony tuning search.
  ah_def_name(hdef, "gemm"); // server knows the "gemm" kernel.
  ah_def_strategy(hdef, "pro.so"); // request PRO search algorithm
  ah_def_layers(hdef, "codegen.so"); // request codegen

  // initialize name, min, max, and step-size for each parameter.
  // here "Tx" means "tile loop x" and "Ux" means "unroll loop x"
  ah_def_int(hdef, "TI", 2, 500, 2, NULL);
  ah_def_int(hdef, "TJ", 2, 500, 2, NULL);
  ah_def_int(hdef, "TK", 2, 500, 2, NULL);
  ah_def_int(hdef, "UI", 1,   8, 1, NULL);
  ah_def_int(hdef, "UJ", 1,   8, 1, NULL);
  htask = ah_start(hdesc, hdef);
  ah_def_free(hdef);
} else { /* Other nodes join master client's session */ }
```

Figure 3.1: Client-side setup code for a parallelized online Active Harmony + CHiLL tuning session [Chen, 2019]. The tuned parameter names are known by the server to refer to loop tiling and unrolling for each loop nest.

to execute the tuning loop specified by the user, experimental code is sent by the server and dynamically loaded using `dlopen` and `dlsym` at the point where a new configuration is fetched through the Harmony C API (Figure 3.2). This code's performance is measured using a testing workload and reported back to the server.

It is important to recognize that Active Harmony is effectively a traditional autotuning system that is accessible through a C API. That is, the user of Harmony must manually synchronize the clients and setup their own tuning loop, which runs for a fixed period of time or until convergence, to test a new configuration on each iteration. The user must perform their own profiling, both to identify a function that is worth tuning and to evaluate each configuration sent by the server.

The Harmony server also does not explicitly tackle the problem of finding balance; *i.e.*, managing the exploration versus exploitation trade-off of online tuning to minimize overheads. The only case where this problem is addressed is when the server is not ready

```
for (i = 1; i < SEARCH_MAX; ++i) {
  // Retrieve a new point to test from the tuning session.
  // This call modifies tuned variables and may call dlopen, etc.
  fetch_configuration();

  // Execute and measure the client application kernel.
  memset(C, 0, sizeof(C)); // clear output matrix
  time_start = timer();
  code_so(500, A, B, C); // compute C=A*B for the 500x500 matrices
  time_end = timer();

  // Report our performance result to the Harmony server.
  perf = calculate_performance(time_end - time_start);
  ah_report(htask, &perf);

  if (!harmonized) {
    harmonized = check_convergence();
    if (harmonized) {
      // Harmony server has converged. One final fetch
      // to load the harmonized values and disconnect.
      fetch_configuration();
      ah_leave(htask);
      break;
    }
  }
}
```

Figure 3.2: The main loop of an online Active Harmony + CHiLL application that uses the code-server to search for code variants of a naive matrix multiplication implementation for optimally performing configurations [Chen, 2019].

with a new configuration. In this case, an Active Harmony client simply uses the existing configuration instead of blocking to wait for the server.

3.1.2 Kistler's Optimizer

Kistler [1999]'s dissertation[1] describes a general, extensible architecture for online program optimization built on the Oberon System 3 for the Macintosh [Gutknecht, 1994; Wirth and Gutknecht, 1989]. Kistler's system consists of five modular components: a

[1]See Kistler and Franz [2001, 2003] for a summary of Kistler [1999].

code-generating loader, continuous profiler, manager, optimizer, and replacer. The manager is a low-priority thread that periodically consults the profiling data being gathered, looking for changes in behavior. If the recent profiles of a function are considered different enough according to a similarity metric, the manager requests reoptimization of the function.

If the manager determines that the estimated speedup is not worth the cost of performing reoptimization, then no change is made to that function. Otherwise, a fixed-order sequence of *optimization components* are applied to the function. Each optimization component consists of a set of one or more optimization passes that perform some compiler optimization (*e.g.*, loop-invariant code motion), plus a profitability analysis that considers code features and the profiling data before deciding to apply the optimization.

Adaptive optimization is exhibited in Kistler's system from its use of dynamically toggled compiler optimizations based on continuously monitored program behavior. For example, an optimization component may look to see if a certain profiling counter exceeds some threshold in order to be deemed profitable. On the other hand, if an optimization component was applied in a previous version of the function, and new profiling data suggests that the function's performance became negative or did not change, then the component is marked as not-profitable in a database. Since the database's information is aged during execution and discarded after the application exits, an optimization component that was disabled may be reenabled, or a component that was skipped may be applied in the future. Thus, there is an air of search-based automatic tuning in Kistler's system.

It is important to recognize that the optimization components in Kistler's system toggle themselves independently based on their own cost models, without a central system monitoring the overall performance. If all optimization components were completely independent and isolated from each other, this would not be a problem. But in general,

the compiler optimizations applied to the program influence each other. Kistler's evaluation of the system does not discuss total system performance gain or loss for a given benchmark. But, even in an ideally optimized case, most benchmarks only saw a roughly 5% performance gain, except for one outlier which improved by 125%. An evaluation of adaptively toggling optimizations was not presented.

In contrast, the newly developed profile-guided optimizations described in the dissertation, *i.e.*, object layout and instruction scheduling, were effective. Adaptive object layout yielded an average 24% performance improvement and instruction trace scheduling a 4% improvement, over the baseline of an ideally optimized program.

Additionally, Kistler analyzed *break-even points*, which are points at which the adaptively optimized program saves enough execution time to overcome the cost of optimizing it (Section 2.5).

3.1.3 Jikes RVM

The Jikes RVM[2] is an adaptive Java virtual machine featuring JIT compilation [Arnold et al., 2000, 2002]. Jikes RVM periodically considers reoptimizing the hottest methods and uses a simple cost-benefit model based on execution time estimates to decide whether the compilation cost is worth the investment. The key innovations of Jikes are the low-overhead profiling techniques and continuous recompilation used to drive five adaptive optimizations:

1. Profiling is used to decide whether to promote the code's optimization level in order to balance compilation costs with code quality.

2. A dynamic call graph is continually updated based on profiling data to identify hot call edges and perform adaptive inlining.

[2]Originally called the Jalapeño Adaptive Optimization System.

3. Code within a method is laid out to prioritize locality based on profiling data.

4. Loop unrolling is adapted by either doubling the compiler's heuristic unrolling threshold, or halving it, based on whether profiling determines the loop is hot or cold respectively.

5. Path profiles are used to focus an optimistic transformation called *merge splitting*, where the goal is to eliminate control-flow merges within a method via code duplication to aid later optimization and analysis passes such as redundancy elimination [Arnold et al., 2002; Bodík et al., 1998; Chambers and Ungar, 1991].

A major distinguishing factor is Jikes's use of profile-guided heuristics to determine what optimizations to dynamically apply to the code, in contrast to the use of empirical search proposed by this work.

Jikes's dynamic profile-guided inlining yielded improvements of 11% on average and a peak of 73% on their benchmark suite for start-up performance. The instrumentation needed to generate profile data added at most 1–2% overhead. Overall, Jikes improved peak steady-state performance by 4.3% on average and up to 16.9% in one case. The top two most impactful optimizations were profile-guided code layout and merge splitting, which aided redundancy elimination in removing method accessor guards introduced by inlining.

3.1.4 ADAPT

Voss and Eigemann [2001] describe ADAPT, which is an online adaptive optimization system for FORTRAN 77. What sets ADAPT apart from prior systems, such as JVMs featuring JIT compilation like HotSpot and Jikes (Section 3.1.3), is ADAPT's use of an iterative search process to find a good optimization configuration for simple loop nests. ADAPT identifies a hot loop that executes long enough and begins experimenting with

differently optimized versions of that code. In essence, the work described in this dissertation is similar to ADAPT in that they both use search-based online tuning, although there are a number of important differences between the two.

ADAPT's search process is primarily driven by scripts written in a domain-specific language "AL." These scripts implement heuristics written by compiler developers that decide what to tune and how the tuning should be done for a given loop. For example, their AL script for loop unrolling is a program implementing a linear search of unrolling factors of at most ten that evenly divide the loop's bound. While these AL scripts are similar to offline tuning with a shell script, the AL language for these scripts features high-level constructs to simplify the compiler autotuning task. These constructs include the ability to specify a parameter space, constrain the tuning by the results of compiler analyses, and re-run the tuning if the user determines the best version has become stale.

One of the main shortcomings of relying on these scripts is that it offloads the most challenging aspects of online autotuning onto the users: effective search strategies and managing exploration and exploitation. ADAPT is evaluated with a trivial empirical autotuning task: the AL scripts only experiment with ten or fewer configurations using a simple linear search. Because the parameter space is incredibly small, the explore-exploit problem also did not need to be addressed.

The overheads of ADAPT are claimed to be at most 5%, but when using their do-nothing AL script to evaluate overheads, some programs actually ran a few percent *faster* than when not using ADAPT, with an average 1.6% improvement on Linux. The authors state the reason for this is that ADAPT always performs "inter-procedural constant propagation and applies some simplifications that may lead to improved performance" when initially compiling the program and runtime system components, so the true overheads are unknown.

The performance benefits of using ADAPT for autotuning were notable: average im-

provements of 35% on the backend flag selection problem and 18% on loop unrolling relative to their baseline on five SPEC2000 floating-point benchmarks. It is important to note that each of these benchmark programs ran for 8 to 70 minutes (average of 21.8 minutes), presumably to give the system time to experiment, but the authors did not specify how they determined the amount of work to be performed for each benchmark.

3.1.5 ADORE

Lu et al. [2004] describe a binary optimizer called ADORE (Adaptive Object code REoptimization) for speculative online adaptive optimization. ADORE is a uses performance monitoring, sampling-based profiling, and phase-change detection to deliver dynamically optimized cache behavior that improves performance by 3%–106% while incurring 1%–2% overhead on average for the SPEC 2000 benchmark programs considered. Intuitively, a *program phase* roughly corresponds to a contiguous period of time with consistent or "similar" program behavior within that period [Hind et al., 2003].

ADORE is a shared library that is linked into a single-threaded executable that is specially compiled to allow for code mutation, namely, the compiler reserves some free registers in the generated code for dynamic modification by ADORE. The dynamic optimization of ADORE is driven by the fixed rate at which sampling data is delivered to ADORE from the CPU's performance monitoring units (PMUs), which was empirically chosen to be 400,000 cycles/sample. The sampling data from the PMU contains information about data cache misses and recently-taken branches. Later work by Zhang et al. [2005] proposes additional hardware extensions that would enhance ADORE with event-driven optimization and better profiling to reduce overheads.

As the program is executing, ADORE periodically mutates hot code to redirect control-flow to a dynamically optimized code "trace," or piece of code that has a single entry-point but multiple exits. ADORE uses PMU sampling information to drive two opti-

mizations within a code trace. The first and primary optimization determines which load instruction is the most costly based on its data reference pattern in order to schedule an appropriate data-cache prefetching directive. The second utilizes the recently-taken branch data to build a path profile that is used to layout code within the trace for better instruction-cache locality. Once the code trace is in place, the ADORE system hibernates until a high-level phase change is detected, when it will then reoptimize the code based on the new sampling data.

There are two aspects of ADORE that put it outside of the class of search-based automatic tuners. First, its process of "searching" a configuration space of knobs is passive, being driven by the phase-changes of the program. Additionally, once a phase-change occurs, cost models are used to determine how and where a prefetching directive should be added, instead of using experimentation. No active effort is made to undo poorly-chosen prefetch directives, leaving it up to the phase detector to adjust the optimizations once the program's behavior changes.

3.1.6 Suda's Bayesian Online Autotuner

Suda [2007, 2010] tackles the high-level problems of online autotuning through the lens of a formal, abstract model of the online autotuning process. Empirical autotuning is seen as having two distinct kinds of executions of the program under a given configuration. There is the *trial* execution, which corresponds to an evaluation of an exploratory configuration (often with sample inputs), and the *practice* execution that exploits the best-discovered execution. Suda remarks that offline tuning is distinct from online tuning in that it performs all exploration before any exploitation. Suda employs a Bayesian approach to manage the case of online autotuning; *i.e.*, where one must strike a careful balance of exploration and exploitation.

Suda's proposed Bayesian approach is based on cost models, which can be either an-

alytical or trained via machine learning, that accurately describe the cost function to be minimized. The key limitations of the Bayesian approach are the reliance on a fixed, predetermined number of experiments during the tuning process, and strong dependence on the estimation of the accuracy of cost model. In addition, assumptions are made about the stability and deterministic behavior of the trial execution's reported cost, namely, that the costs are normally distributed around some mean when the configuration remains fixed.

One of the downsides Suda discovered is that if the cost model is deemed to be very accurate by the Bayesian method, then virtually no exploration will occur, but if the model is determined to be too inaccurate, then it effectively performs all exploration up-front before doing any exploitation. A proposed solution to this problem adds the assumption that the cost function is linear. Then, a fixed number of exploration steps n_{init} is chosen to be at least equal to the number of degrees-of-freedom in the linear model. For execution step $i > n_{init}$, the proposed system decides to exploit the best configuration if $i \geq c \log i$, for some arbitrary c, otherwise it will explore a new configuration. This type of decision-making process for whether to explore or exploit is effectively a non-random version of the approaches used to solve multi-armed bandit problems.

3.1.7 PEAK

Pan and Eigenmann [2008] describe PEAK, an automatic compiler-optimization tuning system that only needs partial-program executions rather than full executions to evaluate configurations. The advantage over whole-program executions is that the time to tune a particular function is greatly reduced, since a function will be called many times during a single program execution. PEAK is designed to perform autotuning using training data in a tuning stage in order to produce a final production-ready version of code, thus it is classified as an offline tuning system.

Nevertheless, from the viewpoint of the implementation techniques used in its pre-tuning and tuning stages, PEAK is similar to an online autotuner. Prior to tuning, PEAK analyzes profiling data to choose a worthwhile function to tune [Pan and Eigenmann, 2006]. During tuning, the tuned function (and its callees) is dynamically switched-out with a differently-optimized versions to evaluate multiple configurations within one execution of the program. PEAK uses a number of practical methods for comparing the performance of these different versions, since raw execution times from two arbitrary time slices of a program's execution may not have equivalent workloads for the function [Pan and Eigenmann, 2004]. Both of these techniques are solutions to problems faced by online autotuning frameworks, though PEAK only utilized them to make offline autotuning more efficient.

3.1.8 PetaBricks

Ansel et al. [2009] developed PetaBricks, an implicitly parallel language that incorporates runtime-tunable values. Tuning is performed for language-level program concepts such as parallelization techniques and algorithm choice, rather than compiler optimizations. SiblingRivalry, PetaBricks's technique for online performance auditing, partitions resources and then races two variants in real-time (through process forking) to determine which configuration is better [Ansel et al., 2012]. PetaBricks's search procedure uses evolutionary techniques and a multi-armed bandit model to choose the mutation operator to apply to the current best configuration to yield the next one [Ansel et al., 2011, 2014; Fialho et al., 2010; Maturana et al., 2009].

3.2 By Philosophy

This section contains an overview of a number of works that are *spiritually* similar to the goals of the proposed research, but are not sufficiently adjacent to the thesis. By spiritually, I mean that these works recognize the importance of leveraging dynamic profile data to improve program performance. The specific works chosen for inclusion in this section either intersect with the thesis in some ways, or are particularly interesting to consider.

3.2.1 Adaptive Fortran

Hansen [1974] created Adaptive Fortran (AF), one of the earliest known works in online adaptive optimization. Hansen was motivated by the findings of Knuth [1971] and others, who found that small parts of a program account for the majority of the execution time. Profile information is used by AF to determine where and when extra time should be spent dynamically optimizing the program. Thus, AF primarily reduces the cost of static compilation, optimization, and native code loading time by delaying these tasks so they occur on-demand at runtime. One of the challenges faced by AF was controlling the rate at which optimizations were applied to the program, because the system somtimes overoptimizes parts of the program before accurately determining whether the compilation overhead will pay off. For example, the heuristic for determining whether to further optimize code, predetermined execution frequency thresholds, worked well for some benchmarks but different thresholds were needed for others.

3.2.2 Dynamic Feedback

Diniz et al. [1997] developed a technique they call "dynamic feedback" for adapting programs to their runtime environment by dynamically selecting among a small number of predetermined versions of code (three in their evaluation). These versions implement

different optimization policies, such as whether two critical sections in the parallel code should be merged or more finely broken down in order to reduce synchronization overhead. What sets this work apart from pure code multi-versioning (Section 2.3) is the continuous adaptation loop that switches between two fixed-time phases. The performance sampling phase explores the effectiveness of each code version under the current process's environment by testing out all versions empirically for a short time. The production phase then exploits the best version determined in the sampling phase to amortize the cost of exploration. Under a number of assumptions, the authors provide an analysis of the worst-case performance bounds of dynamic feedback, which can be used to pick a time for the production phase that guarantees a minimum level of performance. The key limitation of Diniz et al.'s work is that the sampling phase does not scale to situations where a large number of code versions must be tested; the sampling phase would become too long because it tries all versions.

3.2.3 Dynamo

One of the major influential works in adaptive optimization is the Dynamo project [Bala et al., 2000], an online optimizer for native machine-code binaries. The project's main vision is that application end-users can adaptively optimize applications without needing access to their source code. Dynamo readapts to new program behaviors after reaching a steady state through periodic flushing of their code cache. Good results were achieved on SPEC95's benchmark suite relative to default optimization with a static compiler: up to 22% improvement in some cases and 9% on average through partial procedure inlining and improved code block layout. But, if the static compiler utilizes profile data to produce the optimized binary (*i.e.*, profile-guided compilation), Dynamo is unable to improve application performance. Recent work by Panchenko et al. [2019] that adaptively optimizes machine-code binaries, primarily through improved code layout, has been able

to significantly outperform the capabilities of existing profile-guided compilation in static compilers.

3.2.4 CoCo

Childers et al. [2003] proposed the Continuous Compiler (CoCo) framework and performed preliminary experiments with a simulator to gauge the effectiveness of CoCo in adapting loop optimizations for embedded ARM systems. While the philosophy of CoCo is the same as in this work, CoCo relies on expert-defined analytical models to predict the impact of an optimization, as detailed by Zhao et al. [2002]. Thus, hand-crafted models are used instead of any trial-and-error for adaptation. Follow on work by Zhao et al. [2005] extended the analytical models to predict the effectiveness of scalar optimizations, although the work was implemented using the Machine SUIF compiler [Smith and Holloway, 2002] instead of CoCo.

3.2.5 MATE

Morajko et al. [2007] created the Monitoring, Analysis and Tuning Environment (MATE) to perform dynamic automatic tuning for C/C++ applications. Their particular focus was on programs running on a cluster or supercomputer. For each application, MATE requires the developer to provide information about what can be tuned, where to measure for performance changes, and a performance model. Their performance model is a set of rules that signal if performance has deteriorated, improved, or has reached optimality. Their evaluation of MATE first focused on the scenario of tuning a communication library (similar to MPI [Gropp et al., 1999]) within a space of eight different options using exhaustive search, with a baseline running time of 12.2 minutes. The second scenario tuned a knob that controls the division of work in a parallel application under a variable workload.

MATE was sometimes able to improve the performance of an application. For example

in the first scenario, when tuning a certain binary option by itself, MATE slowed down the overall execution of the program by 5.1%. However, when tuning all three binary options, the execution time improved by 27.7%.

3.2.6 AOS

Hoste et al. [2010]'s Adaptive Optimization System (AOS) is built on top of the Jikes RVM (Section 3.1.3) and is similar to this work in terms of combining automatic tuning and dynamic compilation. The key difference is that AOS is an automatic tuner *for* a JIT compiler, not one that performs online adaptive optimization *using* a JIT compiler. That is, in a tuning phase prior to the use of the JIT compiler in production, AOS searches for an optimal set of optimizations to be applied to JIT-compiled methods at each optimization tier (*e.g.*, -O0, -O1, -O2). Thus, while AOS itself is an offline tuning system, the object being tuned is an online adaptive optimizer.

Notably, AOS's tuning phase is broken into two stages. The first stage is a search over a compiler optimization configuration space under a scenario where activity such as garbage-collection and concurrent runtime compilation are excluded (*e.g.*, the heap size is set to be very large). The evolutionary search narrows down a space of 2^{33} configurations down to eight high-quality configurations across the dimensions of execution-time and compile-time. The second phase then uses another evolutionary search to find an optimal mapping from configurations to optimization tiers, giving rise to 92 possible assignments to explore. This reduced search is performed in a more realistic environment where the previously-excluded overheads are now present. The total time needed to tune the Jikes RVM with AOS across 16 standard Java benchmark programs on a single machine take 550 hours for the first stage and 1,320 hours for the second stage, *i.e.*, a total of 1,870 machine hours or nearly 78 days. Since this offline tuning can be parallelized, they were able to complete the tuning in only 75 hours (three days) of actual time, which means on

the order of 25 machines were needed. Hoste et al. performed an extensive evaluation of AOS and were generally successful in showing that autotuning of a JIT compiler can match manual tuning.

3.2.7 Testarossa

Sanchez et al. [2011] describe an experimental augmentation of the IBM Testarossa JVM that utilizes a support-vector machine learner to produce function-specific compiler optimization plans, which toggle the optimizations to be applied to the JIT-compiled function. The model is trained ahead-of-time to recognize code features of functions and produce a specially-tuned optimization plan. Milepost GCC [Fursin et al., 2011] uses a similar approach, where machine learning is used to predict optimal program-specific plans, but Testarossa focuses on function-specific plans instead of entire programs.

Data to train the model is generated through iterated compilation while training programs execute. Both random search and a type of simulated annealing are used to explore the compilation-plan space during data collection. Their experiments showed that their machine-learning predicted optimization plans underperform in steady-state performance when compared to Testarossa's default optimization plans. But, compilation overhead (and thus JVM warm-up time) were reduced with the plans produced by the model.

Nuzman et al. [2013] extended the IBM Testarossa JVM with support for C/C++. Their main contribution was to show that adaptive optimization through JIT compilation can be implemented in such a way that the overheads are low enough for use by languages that are normally compiled to machine code. Nuzman et al. provide a detailed overview of their runtime system's infrastructure and how they avoid introducing overhead. Overall, they were able to demonstrate a 7% total average performance improvement through dynamic optimization of SPECint2006. This work is exploratory in the sense that their

dynamic optimizer under-utilizes the information available to it: they only optimize the code based on dynamically-profiled branch probabilities to improve code layout. Nevertheless, it shows that there is room for future work in this space.

Chapter 4

Thesis

The specific goal of this dissertation is to investigate the following high-level question: *can an automatic, online, search-based, adaptive recompilation system be effective in improving the performance of programs?* The key factors that distinguish this question from previous work is the use of search-based methods in an automatic, online system. The remainder of this section expands on what is meant by "effective" and "performance" under the context of the three broad barriers facing the wider use of search-based adaptive optimization, as described by Basu et al. [2013]: usability, generality, and managing overheads.

Usability There are two groups of software developers for whom usability of an adaptive optimizer matters: domain experts and average developers. A domain expert possesses the knowledge to profile and manually optimize their performance-critical system. Thus, the major usability concern for an expert user of adaptive optimization is its ability to deliver performance that matches or exceeds the expert's ability to perform manual optimization. On the other hand, an average application developer is interested in using adaptive optimization to meet their performance needs, but do not know precisely how to achieve them. An easy-to-use, portable system is a bigger usability concern for average users than its performance gains. This research aims to evaluate the usability of adaptive

optimizers for average developers.

Generality Basu et al. [2013] highlights two generality concerns: the *customization* available to expert users across different problem domains to guide the system and the *compatibility* across different programming languages, operating systems, and hardware. Manual customization of the optimization process by an expert, *i.e.*, programmer-directed adaptive optimization, may yield better results more quickly than a fully automatic system. This type of customization is not required for an online system to be considered effective, for two reasons.

First, in an online system, optimization is meant to occur during useful program execution, so the rate at which the optimizer can converge is constrained by the program's execution behavior and the overheads added by the online system's infrastructure. Thus, the value of customization is less clear in an online system, unless if the system is used like an offline optimizer, where the program is repeatedly performing the exact same work in a test environment. Second, customization primarily benefits domain experts, but the focus of this work is on usability for average developers. This work will view generality through the lens of compatibility.

Performance and Managing Overheads Any adaptive optimization system will have some overhead, whether it be the time it takes the optimizer to converge on an optimal solution, or the slowdown due to instrumentation and sampling for dynamic profiling. If the overheads outweigh the performance improvements, the system is either ineffective at managing overheads or finding performance gains.

Suppose a user has a program and one computing resource with which they would like to try search-based adaptive optimization, *e.g.*, autotuning. Applying a traditional autotuner, such as OpenTuner [Ansel et al., 2014], requires creating and running a tuning phase prior to obtaining and deploying the resulting optimized version for real-world

use. Because application-specific tuning is performed during a phase *prior* to real-world use, the focus of most offline systems is to minimize the tuning time by reducing the number of evaluations required to converge on a good result. While the total time to execute this process depends on a number of other factors, such as the running time of each program's test run and the user's patience, times on the order of hours to days are not unusual (*e.g.*, Section 3.2.6). Fortunately for offline systems, the cost of tuning is completely separate from the use of its resulting optimized configuration.

For online systems, the training phase is continuous and interleaves with the usage of its results. This property makes overhead management a central point of evaluation because of the need to pay-off the overhead of exploration. Because of this difference, the performance of an online optimization system is harder to evaluate relative to an offline system. The ability to adapt to unforeseen workloads or environments is the unique advantage that an online system offers over an offline one. Thus, the other point of evaluation is the performance improvements seen through latent, on-demand tuning that is specific to the workload or machine.

Part II

Halo: Wholly Adaptive LLVM Optimizer

Chapter 5

System Overview

This chapter provides an overview of a new search-based, online adaptive recompilation system called HALO[1] [Farvardin, 2020], which I developed to support my thesis (Chapter 4). The main goal of HALO is to leverage a novel combination of techniques evolved from the state-of-the-art to overcome the challenges of search-based online adaptive optimization. For generality, HALO optimizes programs represented in LLVM's intermediate representation (IR), instead of operating on any particular high-level language.

The LLVM compiler infrastructure is used by major compilers for C, C++, RUST, SWIFT, and other languages to target a variety of hardware architectures such as X86-64 and ARM. Lattner and Adve [2004] originally positioned LLVM as a compiler-based lifelong program-optimization framework. LLVM is designed around a common intermediate representation (IR) that is a typed, static single-assignment representation of the program with constructs that mirror a high-level assembly language with unstructured control-flow. In many respects, LLVM IR is similar to a verbose, normalized C language with support for features like exceptions. Thus, prior work in adaptive optimization for C-like languages is applicable at the level of LLVM IR with sufficient metadata from the compiler front-end.

[1] An acronym for Wholly Adaptive LLVM Optimizer.

Since HALO itself operates solely on a compiler IR, we extend the LLVM-based CLANG compiler for C/C++ with support for generating executables that utilize HALO for adaptive optimization. HALO is designed to be easy to use out-of-the-box for average developers. Users of the system do not need to make any changes to their C or C++ program's source code: simply add the `-fhalo` flag when compiling with CLANG to produce a HALO executable.[2]

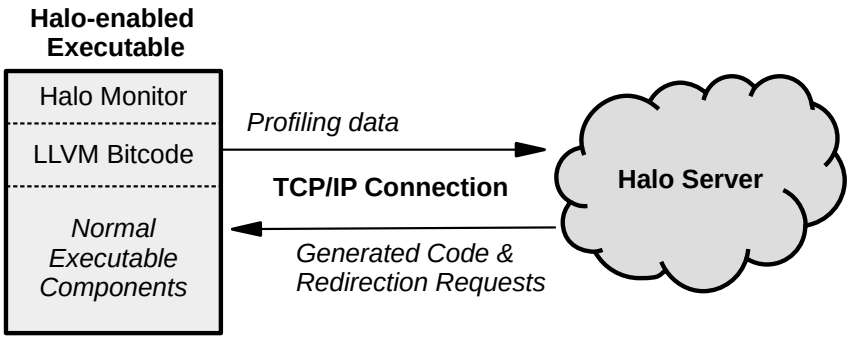


Figure 5.1: The client-server separation used by HALO.

The system is based on a client-server model (Figure 5.1) to isolate the activity of the adaptive optimizer from the running application. This separation prevents interference, *e.g.*, from compilation activity, that can degrade application performance. A *client* is a running executable that utilizes HALO's monitor subsystem, which is controlled by a server for performance enhancement. After establishing a connection, the most common type of message sent by the client consists of raw profiling data, with the server tuning each client by sending re-optimized versions of the client's code. The connection is terminated whenever the client process's `main` function returns, which invokes the monitor's destructor. For the remainder of this chapter, I describe in more detail the three major components that make up the HALO optimization system:

- A modified CLANG compiler (Section 5.1) that produces Halo-compatible executables for C and C++ programs.

[2]Currently, only Linux systems that use ELF object files are supported.

- The HALO monitor library (Section 5.2), also called `halomon`, that is linked into executables to perform profiling, patching, *etc.*
- HALO server (Section 5.3), a standalone application that performs adaptive optimization for connected clients.

5.1 Clang

The HALO system can optimize unmodified C and C++ programs through an augmented CLANG compiler. By simply providing the `-fhalo` flag, the CLANG compiler produces executables that can be modified during runtime by the HALO monitor (Section 5.2), even though the monitor does not export a public API. Instead, the monitor has requirements on the format of the executable, which CLANG ensures in the following ways.

First, a snapshot of the unoptimized LLVM IR corresponding to the application's source code is taken to be saved as data in a special section of the executable file. This "fat-binary" approach has been shown to be a low-overhead way of providing dynamic optimization for statically-compiled languages [Nuzman et al., 2013]. Next, early in CLANG's optimization pipeline, we analyze the program to determine which functions will be made patchable. Functions that are only ever called once, such as those in the `.text.startup` section or the `main` function are not patchable. All other functions are marked as patchable unless they have fewer than 100 LLVM IR instructions, have no loops, and are a leaf function. Finally, the `halomon` library is linked into the final executable. This library contains a single, static global class definition whose constructor will be run prior to `main`. The constructor spawns a thread for the monitor and the destructor is run when the executable exits to cleanly shutdown the monitor.

5.2 Halo Monitor

The HALO Monitor, also called `halomon`, is a static C++ library that is linked into every HALO-enabled executable by CLANG (Section 5.1). When the monitor is initially launched (in its own thread), it connects to the HALO server via a TCP/IP connection to receive requests (Figure 5.1). Users can specify the server's hostname and port through environment variables that are checked by the monitor, otherwise `localhost` and port 29000 are used by default. The Boost C++ asynchronous IO library is used for network communications [Koranne, 2011]. The types of requests sent by the server involve tasks such as profiling, code patching, and dynamic linking.

5.2.1 Instrumentation-based Profiling

The HALO monitor uses a simple instrumentation-based profiling method to provide an estimate of the frequency of calls to a patched function. The instrumentation consists of a call-counter that is incremented each time the client program calls a patched function, specifically, during the function redirection (Section 5.2.3). The monitor thread then records a timestamp every time it takes a snapshot of the latest counts before sending both pieces of information to the server. Based on the difference in time between timestamps and the number of calls that elapsed during that time, a metric of "number of calls per fixed time unit" is made available for measuring the quality of a tuning section.

One of the unique aspects of this call-rate metric is that it provides a signal for the rate of activity in the program with respect to one function, since the time between calls includes both the time spent inside and outside of the function. A similar and more traditional metric would be the amount of time elapsed to complete a call, which has been used in prior work to measure the performance of the function with respect to its usage by the program [Lau et al., 2006]. The downside of time-per-call is that it requires more

instrumentation: on both function entry and exit. To handle recursive calls efficiently, a timestamp could be pushed onto the program's call-stack instead of maintaining a separate stack of timestamps. Thus, the call-rate metric is less stable as a signal of performance compared to the time-per-call metric, but is simpler to implement efficiently. Nevertheless, both metrics are still fallible in the sense that they do not control for workload variations in the program.

5.2.2 Sampling-based Profiling

Modern CPUs offer performance-monitoring units (PMUs) that can be sampled to provide useful profile information with low overhead, at the cost of accuracy and consistency [Chen et al., 2013, 2016; Weaver, 2015]. The Linux kernel provides access to the PMU via the *perf-events* API [Corbet, 2009, 2011; Weaver, 2016], which is the primary source from which `halomon` obtains reports about the performance of the process to the server. To make use of the *perf-events* data, `halomon` first provides the server with information about the correspondence between code addresses and functions in the process, *i.e.*, the code map. The code map is built using two sources of information. The first piece is the name, size, and offset of every function symbol in the process by reading the executable's `.text` section. The second piece is the starting address of the `.text` section in the process's memory map, which can be obtained from the operating system.

Linux *perf-events* provides samples on a periodic basis from the PMUs after a number of CPU events (or a length of time) has been observed. A poorly chosen period length can cause the sampling to become "synchronized" with the code being executed, which yields biased samples [Chen et al., 2013]. We use the event "number of instructions retired" and set the period length to a large prime number (*e.g.* 15,485,867). This way, the number of instructions within a loop body is very unlikely to be a divisor of the period length. The size of the period directly influences the overhead of *perf-events* sampling, and we hand-

picked the length period to ensure that the overhead is less than one percent. Through *perf-events*, the `halomon` library is configured to save the following information from each sample, where a sample's information corresponds to the state of the PMUs when the event was triggered.

- `ip` — A pointer to the instruction that triggered the event.[3]
- `thread_id` — The thread ID (according to the operating system) that triggered the event. Each sample's information is specific to the particular thread.
- `time` — A nanosecond timestamp that indicates when the event occurred, relative to some fixed point in time.
- `call_context` — A stack of return addresses that are currently on the call stack. When the event is triggered, the kernel will walk the thread's call stack (by following frame-pointers) to a bounded depth to obtain this information.
- `branches` — An array of information about the most recently retired branch instructions. Modern PMUs can be configured to maintain a ring-buffer, called the *last-branch record*, that records metadata about branch instructions: its source address, destination address, and whether the branch was mispredicted [Chen et al., 2013; Kleen, 2016]. The size and types of branches recorded in the array are hardware dependent. For example, on Intel's Haswell, HALO specifies that only call and return instructions be recorded, but on Intel Ivybridge, all branches are included.

5.2.3 Code Patching

The `halomon` library uses live code patching to dynamically redirect control-flow in the running process to new code sent by the optimization server. The only points at which

[3]This pointer is not always exact due to sampling skid [Weaver, 2016].

code can be redirected are specially-compiled functions that were marked for patching. Dynamic code replacement is relatively straightforward for code that is interpreted by the runtime system [Bala et al., 2000; Gal et al., 2009]. Equivalent mechanisms for native code executing directly on the hardware have also been previously developed for C/C++ [Nuzman et al., 2013]. The specific mechanism used to perform patching in HALO is based on the XRay instrumentation infrastructure in LLVM.

Berris et al. [2016] developed XRay as a lightweight function tracing system for LLVM. XRay was designed to facilitate the instrumentation of large-scale systems, such as those used at Google, to track down failures or performance regressions in languages like C or C++. A key goal of XRay is that it does not add any noticeable overhead when instrumentation is disabled, since it is meant to be used for programs compiled to run in production. XRay is fundamentally quite simple: functions marked for XRay instrumentation are compiled such that, at each function's entry-point and all of its exits, a short-jump followed by empty space (filled with no-op instructions) is inserted (Figure 5.2).

```
__tunedFunction:
  jmp    funcBody
  nopw   0x200(%rax,%rax,1)   ; multi-byte no-op
funcBody:
  ; ... body of function ...
```

Figure 5.2: The entry-point of patchable function, in an unpatched state.

The space between the jump and the beginning of the function allows another thread to redirect control-flow to an XRay instrumentation routine by overwriting the first few instructions of the function. The instrumentation routine receives an ID that identifies the function that was redirected (Figure 5.3).

HALO extends XRay's patching mechanism with an additional "instrumentation routine" that actually performs a full redirection of control-flow from the original function, to a dynamically generated version instead. The redirection routine reads a pointer to

```
__tunedFunction:
  mov   $<function id>, %r10d
  call  __functionRedirection
funcBody:
  ; ... body of function ...
```

Figure 5.3: A patchable function that was dynamically redirected.

the dynamically loaded function pointer from an array (Figure 5.4). The array is primarily used because the XRay's existing implementation cannot directly patch in calls to dynamically generated code, but XRay could be modified to support such calls.

```
__functionRedirection:
  movq  _ZN6__xray17XRayRedirectTableE(%rip), %r11

  ; set r11 to the offset of this function's table entry
  shlq  $4, %r10      ; r10 contains the function id
  addq  %r10, %r11

  movq  (%r11), %r10  ; load the function pointer into r10
  incq  8(%r11)       ; increment the call counter

  ; if the function pointer is zero, go back to original
  testq %r10, %r10
  je noRedirect

  popq %r11    ; adjust stack to return to the function's caller
  jmpq *%r10   ; call the dynamically-loaded function

noRedirect:
  retq
```

Figure 5.4: The function redirection routine.

A mechanism such as on-stack replacement (OSR) would be needed in order to patch in code snippets *within* a function, such as a new loop body [D'Elia and Demetrescu, 2016]. OSR is a mechanism for dynamically redirecting control-flow at a certain point within a function to another equivalent point in different version of that function. The advantage of OSR is that we can fully optimize a function's very long-running loop, whereas

function-call replacement is limited to optimizing that loop's callees. Mosaner et al. [2019] recently developed simple techniques for performing OSR in LLVM via loop extraction and were able to improve warm-up time without significantly diminishing performance. But, Fink and Feng Qian [2003] found that OSR in an online adaptive optimization system primarily benefits debugging or optimizing pathological code. For performance improvements, they suggest investing effort on the actual code optimizations being adapted instead of the granularity at which they can be done. Thus, HALO replaces code at the function-level granularity, instead of using OSR.

5.2.4 Dynamic Linking

Each object file sent from the server to a client represents a portion of the program that was recompiled with a new tuning configuration. These object files contain one exposed function symbol that represents the entry-point into that version's code. The client uses standard `dlopen`-style dynamic linking to load the object files as a dynamic library, or "dylib" in the process. To facilitate this linking, the original executable (which itself is just an object file) is compiled such that any global symbols, such as variables and other patchable functions, are exposed for `dlopen` to access. This way, when linking a new object file into the process, references to mutable globals (*e.g.*, static local variables in C functions) refer to the version established when the process first launched, instead of creating a dylib-private version.

Currently, HALO does not try to garbage collect any dylibs that might be unused, since it is quite difficult to know when the code is truly dead without further coordination with the running process. Additionally, the dylibs are typically small (each at most a kilobyte or two) and the server sends new object files infrequently. Nevertheless, one possible garbage-collection solution is to have the monitor fork a child process that invokes Linux's `ptrace`, or a "process trace" on its parent, providing the child with debugger-like

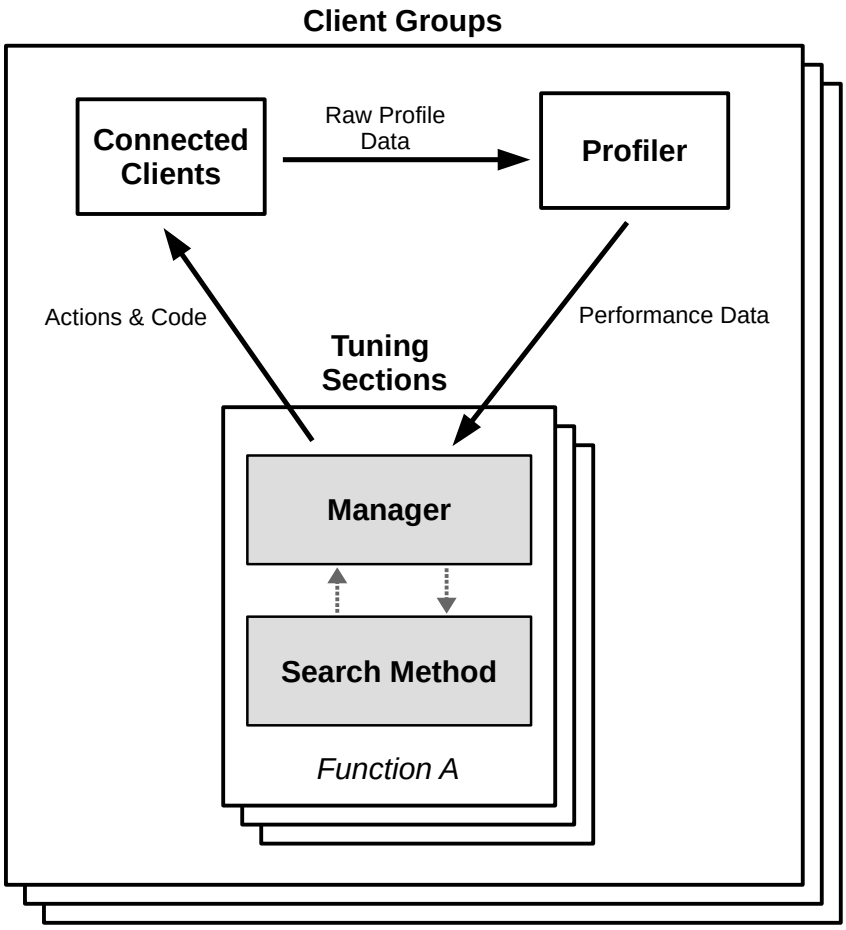


Figure 5.5: An overview of Halo Server's major structures and the flow of information. Solid arrows point to information consumers and dotted arrows indicate a dependence.

control to pause all threads and inspect their state. The child can then walk each thread's stack and look for any values that might be addresses pointing to dylibs, so that the parent can then free the unreferenced code later. But, this is not a good solution if any of the functions in the dylib have their address taken as a value, since the heap would need to be scanned too.

```
{ "kind": "flag",
  "name": "native-cpu",
  "default": false },

{ "kind": "int",
  "name": "inline-threshold-default",
  "scale": "1/100",
  "default": 2,
  "min": 0,
  "max": 30 },
...
```

Figure 5.6: An example of two JSON-formatted knob specifications used by HALO server.

5.3 Halo Server

The HALO server performs the reoptimization of the code being executed by connected client processes. The major structures and overall flow of information within the HALO server are illustrated in Figure 5.5. Client processes are grouped together into *client groups* based on the compatibility of their embedded LLVM IR and CPU architecture, both of which are sent by clients during registration with the server. The optimization server drives the adaptive optimization of client groups independently, based on profile data periodically sent by the clients of each group. Within each group are one or more *tuning sections*, which are subsets of the code running on the client that have been selected for reoptimization. Specifically, a tuning section is a pair consisting of a patchable "root" function and a set of functions reachable from that root, according to the program's call-graph. The root function is unique among all active tuning sections in the group, so the root function distinguishes a tuning section.

The server uses a JSON-formatted file that specifies all of the server's settings and defines the space of options to be searched during tuning. Figure 5.6 provides an example of specifications for two tunable parameters, or *knobs*, that control HALO server's compiler. I will refer to the entire space of compiler options available to be tuned as a *con-*

figuration space, which is equivalent to the cross-product of the sets containing all possible settings for each knob specified in the file. Additionally, a single element in the configuration space is called a *configuration*, which describes a particular setting of every knob (Section 2).

The values for the `name` and `kind` fields of JSON-formatted knob specifications (Figure 5.6) must be already known to the server, because they must be baked into specific parts of the server's compiler. The other fields are free for users to configure in order to constrain or broaden the search space. Because all integer-based knobs are required to form a contiguous range of values, these knobs offer a `scale` field specifying the prior scaling that was applied to the given range. In Figure 5.6, the `inline-threshold-default` knob has a $[0, 30]$ range with $1/100$ scaling, so the unscaled values form the set of integers $\{0, 100, 200, \dots\}$. In other words, the compiler will unscale the value by applying the scale factor's inverse to determine its true value. Scaling allows us to keep the tuner's search space small when probing a large space of possible values for a compiler option.

The search method of each tuning section consists of infrastructure to generate configurations and create parallel compilation jobs (Chapter 7). A compilation job produces an object file that is ready to be sent to clients. The object file and additional metadata are kept together and referred to as a *library* of the tuning section. This name is consistent with the idea that once an object file is sent to a client, it is loaded dynamically like an ordinary library (Section 5.2.4).

5.3.1 Calling-Context Tree

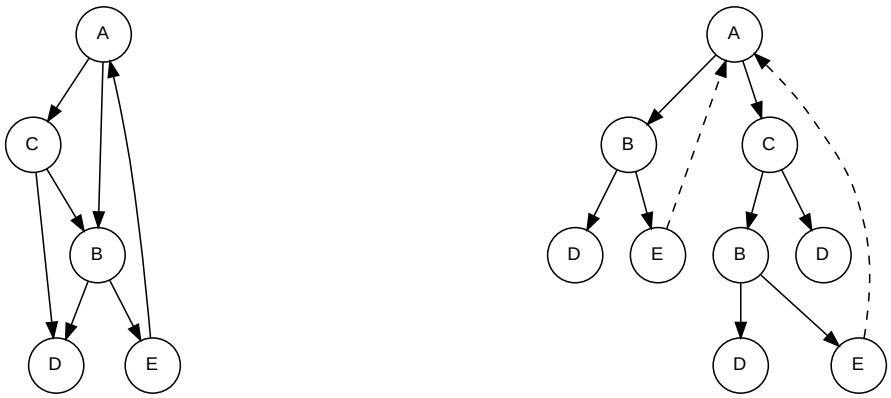
Since each client group can have multiple clients providing sampling-based profile data, HALO server uses a *calling-context tree* (CCT) to combine and manage these samples in each group's profiler. Ammons et al. [1997] originally proposed the CCT as an alternative to the more commonly used call-graphs to manage profile data. A call graph represents

the control-flow of the program at a function-level. Each vertex in a call-graph represents a function and an edge $A \rightarrow B$ represents the relation "A contains a call to B." A static call-graph (Figure 5.7a) is built using only static program information and does not contain complete control-flow information because of indirect function calls. Dynamic call graphs augment the static call-graph with additional control-flow information captured through profiling.

One of the downsides of using a call-graph to store performance metrics is that its structure does not keep track of the calling-context for a performance metric. A calling context is a dynamic sequence of functions that have been called at a given point during execution, *i.e.*, a stack of callers for the current function. Consider two performance samples s_1 and s_2 measured while in function B, where the context for s_1 is $A \rightarrow B$ and for s_2 is $A \rightarrow C \rightarrow B$. When using the call-graph in Figure 5.7a to store this information, we are forced to aggregate these two samples in the one vertex for B, which has two unique callers who may use B in vastly different ways.

The calling-context tree (Figure 5.7b) offers finer granularity for tracking samples, at the cost of more space. Specifically, the samples s_1 and s_2 end up at distinct vertices for B in a CCT. Insertion of a sample into a CCT happens by walking down the tree (rooted at A) while following the calling-context associated with the sample. The CCT is built on-demand as contexts are observed, so nodes are created as-needed during a walk of the calling context associated with a sample. A calling-context may contain repeated functions due to recursion, so the CCT is not *strictly* a tree. Nevertheless, the only types of non-tree edges a CCT will contain are back-edges [Ammons et al., 1997], which are marked with a dashed arrow in Figure 5.7b.

Processing a Sample's Calling-Context The profile data generated by `halomon` is well-suited, in theory, for a calling-context tree because the samples contain `call_context` metadata (Section 5.2.2). But in practice, the metadata is sometimes broken or incomplete



(a) Call Graph (b) Calling-Context Tree

Figure 5.7: A call-graph versus a calling-context tree for the same program.

for at least two reasons.

The first reason is that sampling happens asynchronously with the running program, so the call stack may be in an indeterminate state at the point where the sample was collected, resulting in a broken calling-context. The second reason is owed to tail-call optimization, which is a compiler optimization to reduce call-stack allocation in control-flow paths that end by returning the value of a function call. The optimization pops the stack-frame of the caller before making a call to a function; this way the callee will return directly to the caller's caller.

For example, suppose we have a path in the call-graph $F \rightarrow G \rightarrow H$, but G will simply return the value returned by H. Then, with tail-call optimization the value returned by H will, dynamically, go directly back to F. So, when a *perf-events* sample is taken in H, the calling-context will be $\ldots \rightarrow F \rightarrow H$. According to the program's call-graph, F does not call H directly, so the calling-context is incomplete and we become stuck at F during the walk to insert the sample.

To resolve a missing function during a calling-context walk, we first search for a shortest path in the CCT from the current, contextually-sensitive node F to a node corresponding to function H. In the case of multiple shortest paths, we pick the path with the highest

total hotness. If no such path in the CCT exists, then we check the call-graph to see if there is only one path from F to H and create fresh CCT nodes for those intermediate functions. Additionally, if the call-graph says that F contains a call to an unknown callee, then we assume H is one of those callees and create a CCT node for H. Otherwise, the sample is dropped.

Recording Performance Metrics After identifying the contextually appropriate CCT node by walking the calling-context, we update the node's *hotness* and *instructions-per-cycle* (IPC) according to the details within the sample. Hotness is a metric that provides an abstract measure of where time is being spent in the program. A CCT node's hotness is incremented whenever a calling-context walk ends at that node; *i.e.*, the `ip` field of the sample is within the function (Section 5.2.2). To compute an IPC, we compute the time that has elapsed since the last sample was observed at the CCT node (based on each sample's `time` field) and divide the sample period (*i.e.*, the fixed number of instructions between samples) by that time difference. For simplicity, the IPC measure is scaled for a 1 gigahertz clock-rate instead of trying to account for frequency scaling. We use a standard incremental update formula [Sutton and Barto, 1998] to maintain estimated averages for the performance metrics at each CCT node:

$$NewEstimate \leftarrow OldEstimate + StepSize \times (Observation - OldEstimate) \tag{5.1}$$

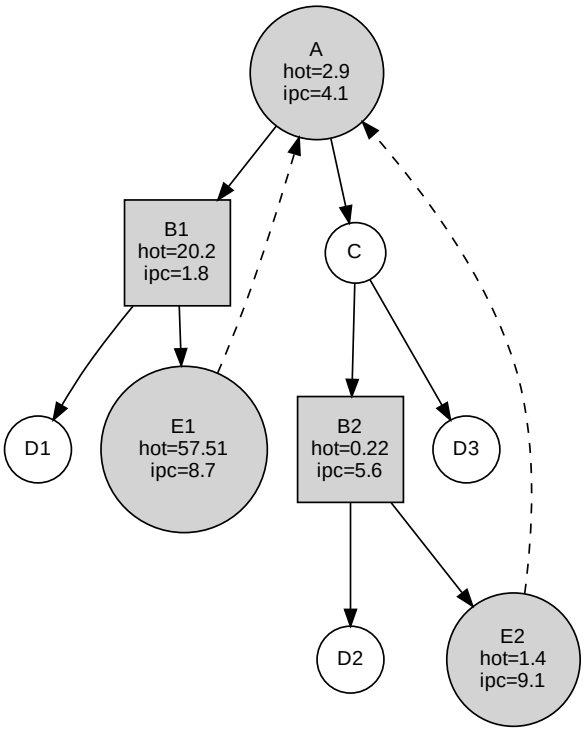
The $StepSize \in (0, 1]$ is a parameter that controls how much weight is placed on newer observations. For the hotness metric, we only use Equation 5.1 to *decay* the hotness value. To decay a hotness value, the hotness is updated according to Equation 5.1 using zero as the observation's value. This decay is applied once to every CCT node at the start of each time-step, so that the hotness tends towards zero unless enough samples are observed.

Additionally, performance metrics are recorded in each CCT node in two ways: generally (across all libraries) and separately for each library, depending on the library to which the sample belongs. Recording metrics per-library enables more accurate performance comparisons between libraries.

Leveraging the Last Branch Record Samples sent by `halomon` contain metadata about the most recently executed branch instructions (Section 5.2.2). After walking the sample's calling-context to reach the context-sensitive node for the sampled function, we walk the last-branch record backwards, from newest to oldest branch, going from each branch's target to its source. Each time we identify a call, *i.e.*, a branch whose target is to the beginning of a function, we move *up* the CCT from the current node to its ancestor (moving backwards in time) and increment the call-hotness of the edge we followed upwards. If the branch goes across two different functions, then a return occurred, so we move *down* the CCT from the current node to the function that was returned from, creating a new CCT node as-needed. The remaining kind of branches are those that stay within the same function, such as for loops.

In all three cases, the CCT node we end at after processing each branch has its hotness boosted and its IPC updated (if not already updated for the sample). As with the calling-context data, the `branches` metadata is sometimes incomplete or broken. In such cases where it is not obvious how to continue walking the records, we simply stop early.

Computing a Tuning Section's IPC One of the primary reasons for gathering profiling data is to facilitate performance comparisons. Because context-sensitive versions of a function in the CCT are assigned their own average IPC, and each tuning section can consist of more than one function, a total IPC for a group of functions is needed to make performance comparisons. Specifically, due to the nature of how code patching works in `halomon` (Section 5.2.3), the relevant nodes in the CCT are all those that are in the tuning



(a) CCT annotated with performance metrics.

$$
\begin{aligned}
H_i &= \text{sum of hotness in sub-tree } i \\
f_i(V) &= \frac{V_{hot}}{H_i} \times (V_{ipc})^{-1} \\
IPC_{B1} &= (f_1(B1) + f_1(E1) + f_1(A))^{-1} \\
&\approx 4.35 \\
IPC_{B2} &= (f_2(B2) + f_2(E2) + f_2(A))^{-1} \\
&\approx 5.02 \\
H_{tot} &= H_1 + H_2 \\
g_i &= \frac{H_i}{H_{tot}} \times (IPC_i)^{-1} \\
IPC_{tot} &= (g_1 + g_2)^{-1} \\
&\approx 4.38
\end{aligned}
$$

(b) Computing IPC_{tot} for the tuning section using repeated, weighted harmonic means.

Figure 5.8: Computing the total IPC of the tuning section $\{\mathbf{B}, A, E\}$.

section and are reachable from any CCT node representing the root function of the tuning section. To deal with cycles in the CCT, we exclude a node if it represents a function that has already been included during the depth-first search to identify reachable nodes.

Consider the example CCT in Figure 5.8a which is used to manage the profiling data for tuning section $\{\mathbf{B}, A, E\}$, where **B** is the root of the tuning section. In Figure 5.8a, CCT nodes representing a function in the tuning section are shaded with gray (root-functions also use a square) and display their hotness and IPC averages. There are two sub-trees corresponding to the tuning section: $\{\mathbf{B1}, A, E1\}$ and $\{\mathbf{B2}, A, E2\}$. To compute the total IPC for this tuning section (IPC_{tot}), we first compute IPC_{B1} and IPC_{B2} for each sub-tree, respectively, using a hotness-weighted harmonic mean of the IPCs for each function (Figure 5.8b). An IPC is a rate-of-time measure where the length of time spent in each function is not equal, so we use a harmonic mean to provide a more accurate average IPC than an

arithmetic mean [Ferger, 1931]. Using hotness-weighting further emphasizes the functions that should be the focus of the average measure during a particular time-step, since only the hotness (and not the IPC) decays as time passes. Once we have the IPCs for every sub-tree, to compute IPC_{tot} we again use a hotness-weighted harmonic mean, but of the sub-tree IPCs for the tuning section.

5.3.2 Tuning Section Selection

One of the challenges of automatic procedure-level optimization is the selection of procedures that are worth optimizing. For traditional JIT compilation systems, usually a single function or small code fragment is chosen for reoptimization, with priority given to code contributing to the most execution time in order to pay-off the cost of reoptimization.

For adaptive optimization systems like HALO (Chapter 3), the code where most of the execution time is spent is just as important as in ordinary JIT compilation systems. If the focus of HALO's tuning effort is not spent on the most-executed parts of code, overheads will take longer to pay-off. Because the code may frequently be replaced by HALO with differently-optimized versions, whether for performance comparisons or other adaptation, the frequency at which the function is invoked is especially important. Additionally, the tuning section may need to span multiple functions to be effectively tuned, unlike traditional JIT systems that choose the code fragment specifically because there are known opportunities for improving it.

Pan and Eigenmann [2006] recognized this problem of *tuning section selection*, which is the challenge of automatically selecting a subset of the program to be tuned based on profiling data. In their system, called PEAK, a call-graph is built using profiling data from a complete execution of the program on a test input. The edges of the call-graph are annotated with call-frequency and execution time. Using a two-step algorithm based on maximal edge-cuts, PEAK selects tuning sections to maximize execution-time coverage

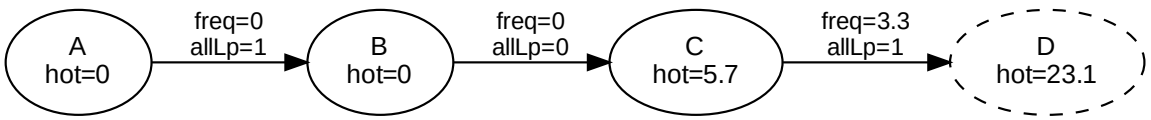


Figure 5.9: An ancestor chain of the CCT used to choose a tuning section root.

while ensuring that each tuning section's root is called often enough to be dynamically tuned.

In contrast, HALO uses a calling-context tree (Section 5.3.1) and performs online tuning section selection. Algorithm 5.1 summarizes the procedure for identifying a new tuning section root. At any given candidate node, we look at the node's contextually-sensitive parent node[4] to determine whether the candidate node should be chosen as a tuning root. The reason for inspecting the parent of the current node is that its immediate parent must be repeatedly making calls to the node in order for `halomon`'s redirection of the function to take hold. Just like PEAK, the goal is to avoid the situation where the tuning section spends a long time executing inside of the tuning section without the root being called often enough.

Figure 5.9 contains an example of an ancestor chain that is climbed (from right to left) to determine a tuning section's root. Suppose the node *D* (which represents function *D* in a specific calling context) is the hottest node in the CCT. Node *D* is the *start* node for FINDTUNINGROOT from Algorithm 5.1. When no tuning root has been chosen yet, the current node's parent must be patchable (Section 5.1) to qualify as a suitable tuning root (Lines 5–6). In our example, node *D*'s parent *C* is patchable as indicated by the solid outline for the node.

To expand the scope of the resulting tuning section, we greedily try to find a better tuning root higher up the ancestor chain, based on profiling data and static code features. Using the nodes in Figure 5.9 as an example, let us consider how scope expansion from

[4]Ignoring back-edges, which introduce a false parent.

```
   Input: start — A starting node from the CCT.
   Result: the name of a function to use as a tuning section root.
 1 candidate ← start
 2 chosen ← NONE
 3 while candidate.hasParent() do
 4 |   parent ← candidate.getParent()
 5 |   if chosen = NONE then
 6 |   |   if SUITABLETUNINGROOT(parent) then
 7 |   |   |   chosen ← parent // initial choice
 8 |   |   end
 9 |   |   candidate ← parent
10 |   |   continue
11 |   end
12 |   hotParent ← parent.hotness() ≥ 1
13 |   isCalled ← candidate.callFrequency() > 0
14 |   calledFromLoop ← SOMEPARENTCALLSFROMLOOPS(candidate, context)
15 |   if hotParent or isCalled or calledFromLoop then
16 |   |   chosen ← parent // revised choice
17 |   |   if SUITABLETUNINGROOT(parent) then
18 |   |   |   candidate ← parent
19 |   |   |   continue
20 |   |   end
21 |   end
22 |   break
23 end
24 return chosen
```

Algorithm 5.1: The implementation of FINDTUNINGROOT that identifies a new tuning section root.

Algorithm 5.1 works in general. The parent B of the currently-chosen tuning root C is a *candidate* tuning root if it is a patchable function. Then, we inspect the grandparent A of the currently-chosen tuning root C, *i.e.*, the parent of the candidate tuning root. At least one of these safety conditions (Lines 12–14) must be satisfied for the candidate B to become the new chosen tuning root:

1. The grandparent node A has a hotness that is above a small threshold.

2. The call-edge from the grandparent A to the parent B has a non-zero call-frequency.

3. One of the ancestors of the candidate B calls its child only from call-sites that are within natural loops.

If the candidate is chosen as the new tuning root, the greedy expansion continues. In our running example, B would be chosen as the final tuning section root because A satisfies the third condition above, as indicated by the `allLp` metadata.

All three of these safety conditions are designed to ensure that the parent of the final tuning root calls the root function often enough such that tuning can proceed. The third condition above leverages static analysis of the program to infer safety when profiling data is lacking. A lack of profiling data for the grandparent can happen, for example, when the grandparent contains a simple loop that calls the parent function that takes a long time to complete. Because the CCT is built using sampling-based profiling, the grandparent's repeated calls to the long-running parent will not be detected in the profile data. The call-counts provided by instrumentation-based profiling in `halomon` (Section 5.2.1) could fill this hole in the profiling data, but widespread instrumentation would be costly. Specifically, call-counts are only available for functions that are patched, so all functions would need to be patched (with no redirection) to gather call-counts. Thus, we opted to use static analysis instead of introducing overhead when selecting a tuning root.

Finally, once a tuning root has been selected, the prototype of HALO creates a tuning section that consists of all functions that are reachable from the root, according to the program's call-graph. Using all reachable functions has the benefit of maximizing coverage of the most-used parts of the program, but this strategy may include irrelevant, less-frequently executed code that bloats the tuning section. During tuning, each library starts with a copy of all the code in the tuning section before compilation. Thus, the all-reachable strategy is not feasible for larger real-world programs, because the server will require more compilation-time, and large object files can bloat client processes with too much cold code. We leave the specific strategy for reducing the size of the tuning section

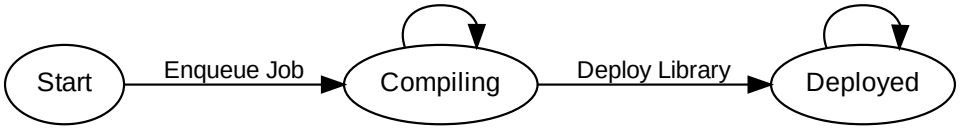


Figure 5.10: A state machine for a once-compiled tuning section (*i.e.*, the JIT-once manager).

while balancing program coverage to future work.

5.3.3 Tuning Section Managers

Each tuning section's manager is a state machine that is driven by a timer that triggers a step every 100 milliseconds. Because the manager is driven by data from the profiler, some waiting in the manager is unavoidable. Specifically, the profile data is being collected from live client processes, which send their data piecemeal on a periodic basis (on the order of tens of milliseconds between messages). To maintain uniformity with how other states operate, a poll-and-wait system (where the wait time is equal to the time between steps) was chosen instead of a blocking manager when there is insufficient new profile data. For example, in some states where profile data is not needed, the manager may want to give the client time to exploit the best-optimized version of code. In other states, the manager is waiting for one of many parallel compilation jobs to finish, so poll-and-wait is a simple solution that works for all of these scenarios.

To give a flavor for these state machines, let us consider a simple strategy for optimizing a tuning section that mirrors a standard JIT compiler: recompile the tuning section only once, at the highest default optimization level. This JIT-*once strategy* is represented in HALO with the tuning-section manager in Figure 5.10. Vertices in the graph represent possible machine states and edges represent actions that transition the machine from state to state. Some transition must be taken by the manager on every step, as driven by the timer. The transitions available depend on the manager's current state and the decision

```
if the compilation job is ready then
    // send the library to all clients
    transitionTo Deployed
else
    transitionTo Compiling
end
```

Algorithm 5.2: The decision procedure for the **Compiling** state of the JIT-once manager.

procedure for that state. For the JIT-once state machine, decision procedures for the **Start** and **Deployed** states are very simple, since only one transition is available. The decision procedure for the **Start** state enqueues a compilation job before transitioning to the **Compiling** state. The **Compiling** state's decision procedure is detailed in Algorithm 5.2.

Because clients can connect or disconnect from the group at any time, we check the currently-connected clients and synchronize them as-needed at the beginning of *all* decision procedures. For example, if a client has joined the group late, then that client's version of the tuning section's root has not been patched yet. So, that specific client's state is synchronized with the tuning section's manager by re-deploying that library (*i.e.*, sending the object file and a patching command) before invoking the decision procedure for the current state. Since synchronization is the only task required once in the **Deployed** state, the **Deployed** decision procedure simply transtions to itself.

5.3.4 Implementation Details

HALO server is implemented as a standalone, open source[5] program written in C++. The program uses Google's Protocol Buffers to serialize messages to send over the TCP/IP connection [Google, 2020]. External libraries are used to implement a number of other components in HALO server: XGBoost for machine learning [Chen and Guestrin, 2016], C++ Boost for networking and graph data structures [Koranne, 2011], LLVM 10 for opti-

[5]Source code is available at `https://github.com/halo-project`.

mization and code generation [Lattner and Adve, 2004], and Lohmann [2020]'s library for JSON parsing. Excluding libraries, comments, and whitespace, HALO server consists of 5829 lines of C++ code.

Chapter 6

Adaptive Recompilation

The adaptive tuning manager is the core piece of the HALO server that performs search-based adaptive recompilation. Each tuning section is assigned its own independent instance of the manager so that its decisions are specific to the code being optimized. As with all tuning managers (Section 5.3.3), the Adaptive manager is described by a state machine, illustrated in Figure 6.1, that is driven forward at discrete time-steps.

6.1 Finding Balance

Central to the manager is the **Decide** state (Figure 6.1) that can choose among three actions: (1) experiment with an unseen knob configuration, (2) retry an experiment using an existing knob configuration, or (3) pause to exploit the best library found so far. All of these actions play an important part of an online search within a non-stationary search space of knob configurations. Good decision-making in the manager is learned through the results, or rewards, of the actions taken. The Adaptive tuning manager's **Decide** state helps regulate the overhead of search and adapts the optimization of the tuning section over time.

The *multi-armed bandit* (MAB) is a learning problem where, at each discrete time-step,

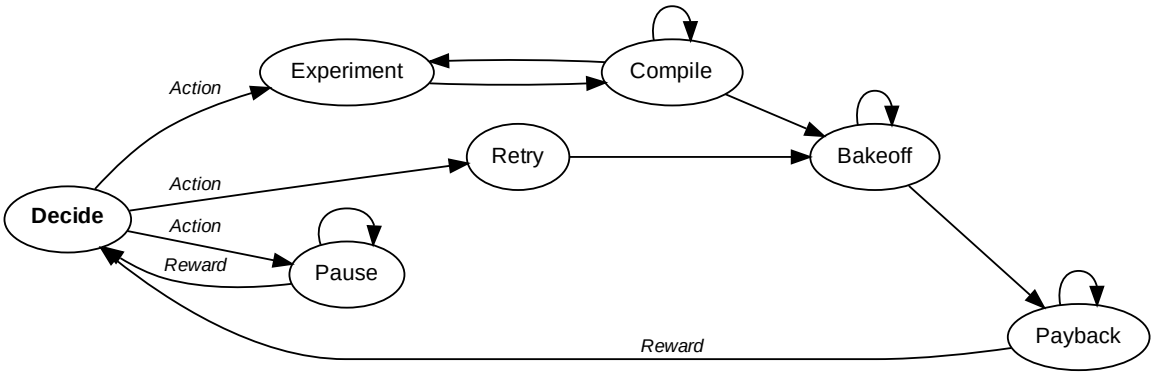


Figure 6.1: The state machine for the Adaptive tuning manager.

one must select among k actions that give a reward chosen from an unknown probability distribution [Sutton and Barto, 1998]. A "one-armed bandit" is another term for a slot machine that is used for gambling [Oxford English Dictionary]. The objective is to maximize the total rewards as one "gambles" by choosing among multiple actions (or slot machine levers) that may give low or negative rewards in the short-term.

In HALO, the Adaptive tuning manager's **Decide** state is viewed as an instance of the multi-armed bandit problem, where the actions correspond to transitions out of the **Decide** state (Figure 6.1). During any transition back to the **Decide** state, a reward is assigned to the outcome of the last action taken by the tuning manager[1]. Specifically, the rewards correspond to whether the outcome is good or not, with higher rewards given to actions that led to a positive outcome.

Solutions to the MAB problem focus on techniques that estimate the value of each available action through trial-and-error, using the estimates to make better decisions over time. In classic formulations of the MAB problem, the probability distributions for each lever are stationary, *i.e.*, the reward probabilities do not change over time. Thus, the value of each action may be estimated to be equal to the average of the total rewards received over time. In HALO, the rewards can change over time because the program's workload

[1]The state machine has persistent memory of the most recent transition out of the **Decide** state.

may fluctuate. For our non-stationary MAB problem, we use a weighted average that emphasizes the rewards of actions tried recently [Sutton and Barto, 1998].

Suppose we have a map $E : \mathit{Action} \rightarrow \mathit{Expectation}$ that maintains the expected value of taking an action. We define a function to update our expectation mapping E based on the reward r for an action a as follows:

$$\begin{aligned} \text{update}(E, a, r) &= E \pm \{a \mapsto \text{increment}(E, a, r)\} \\ \text{increment}(E, a, r) &= E[a] + \gamma\,(r - E[a]) \end{aligned} \tag{6.1}$$

where $\gamma \in (0, 1]$ is a constant step size, or learning rate, parameter (chosen empirically to be 0.1) and the $\pm$ operator overwrites a key-value pair in the map. The update rule in Equation 6.1, which is the same as Equation 5.1 from Section 5.3.1, incrementally calculates an *exponential recency-weighted average* reward for the action [Sutton and Barto, 1998].

Decision-making Knowing the expected rewards for each action serves as a guide for future decisions. Whenever it comes time to make a decision, we could simply pick the action with the highest reward, *i.e.*, make the greedy choice. The problem with greedy decision-making is that it can suffer from tunnel vision. More precisely, a full-greedy strategy results in poor decision-making, because of inherent inaccuracies in our estimate of the average reward of an action [Sutton and Barto, 1998]. This problem is especially an issue for non-stationary multi-armed bandit problems because knowledge about the expected rewards becomes out-of-date, so some exploration is required. One common alternative is the *ϵ-greedy strategy* that picks the action with highest expected reward with probability $1 - \epsilon$ (breaking ties arbitrarily). Typically ϵ is small (*e.g.*, 0.05 or 0.15) so that we are still mostly greedy, but will also explore other actions; we empirically chose a fixed $\epsilon = 0.1$ for HALO. When the greedy action is not selected, which happens with

probability ϵ, we *explore* by choosing among all actions uniformly at random.

The ϵ-greedy selection strategy is how HALO deals with the explore-exploit trade-off in online autotuning, *i.e.*, to keep exploring the configurations or to pause the search. The majority of the time, we exploit what is already known about each action's expected rewards with the greedy choice. But, we also occasionally explore by choosing an action at random.

Pausing Pausing allows each client to exploit the best-known library for the tuning section for a fixed period of time, without any additional overhead from sampling-based profiling or other activity. Thus, only the activity of the tuning manager is paused, not the clients.

Tying everything together, let us consider step-by-step how HALO server's tuning manager makes a decision that results in a pause action. During a step in the **Decide** state (Figure 6.1), suppose we are about to choose an action, *i.e.*, the next state to transition to. Further, let us say hypothetically that the current expected rewards are as follows:

$$E[\text{Experiment}] = -0.28$$
$$E[\text{Retry}] = -0.047$$
$$E[\text{Pause}] = 0$$

After flipping a biased coin, where *heads* ocurrs with probability $1 - \epsilon$, suppose the outcome is *heads*. Based on this outcome, we use the greedy strategy of choosing the action with the largest expected reward. According to our example's expected rewards, the **Pause** action would be selected. After transitioning to the **Pause** state, the tuning manager stays in that state for a fixed number of steps. During that time, the server only communicates with the clients when they first connect to ensure that they have the correct

library patched into the process. Then after pausing for long enough, we transition from **Pause** to **Decide** while carrying a reward of zero. Section 6.2 explains why the **Pause** action's reward is zero and discusses the rewards given for other actions. Before choosing the next action in the **Decide** step, the action-value map is updated:

$$\begin{aligned} E &= \text{update}(E, \text{Pause}, 0) \\ &= E \pm \{\text{Pause} \mapsto E[\text{Pause}] + \gamma(0 - E[\text{Pause}])\} \end{aligned}$$

using the update function from Equation 6.1. Finally, we have completed one cycle and are now ready to start with the next decision in the **Decide** state.

6.2 Bakeoffs

Accurately comparing the quality of two different libraries[2] is a major challenge in an online setting because we do not have control over *what* the program does, only *how* it is done. Specifically, an online system cannot re-run two versions of the code under the exact-same program state [Lau et al., 2006; Pan and Eigenmann, 2004]. It is difficult to re-run a new version of code for comparison with a prior version because the code may depend on the state of global memory, modify its arguments, perform other observable effects such as IO, or any combination of those. In particular, the program may be applying the function to different inputs on each call. Thus, if one were to record and compute the mean running-time of individual function invocations, the variance of the mean is significantly attributable to the function simply doing more work, *i.e.*, the variance is dependent on the function's inputs. This makes it difficult to make confident conclusions based on the mean running-time alone.

[2]Recall that a *library* is a version of the tuning section optimized and compiled to an object file according to a specific configuration of the knobs being tuned (Section 5.3).

Lau et al. [2006] recognized this problem and used an analysis of the variance of running-times to compute the probability that the difference between the means is significant. The running-times are collected during a live contest between two code versions called a *bakeoff*. A bakeoff is a contest that determines which version of code is better at the particular time the contest is held. Bakeoffs proceed by randomly alternating the version of a function that the program uses between the two contestants while recording how well each performs.

In HALO, a bakeoff begins after entering the **Bakeoff** state in Figure 6.1. The bakeoff proceeds by assuming all current and future clients during the contest are homogeneous. The decision procedure for the **Bakeoff** state is given in Algorithm 6.1. During the transition into the **Bakeoff** from either **Retry** or **Compile**, a candidate library is specified to compete with the current-best library that is being used by all clients. Thus, initially the *deployedLib* is set to the current-best library and the *otherLib* is the candidate, and clear the observations for both libraries (Lines 1–3). A time step is not counted towards the bakeoff if the *deployedLib* has insufficient fresh profile data, which is new data attributed to that library that has come in from a client since the last check (Lines 6–8). On Line 10 of Algorithm 6.1, the decision procedure compares the collected observations using a less-than function, COMPAREWITHSTUDENTST,[3] that may return *NoAnswer* (Section 6.2.1). The goal of the bakeoff is to find which library has a higher overall quality value, declaring that library the winner. Based on the result of the comparison, we swap the library used by the clients after a fixed number of time-steps (Lines 11–32).

6.2.1 Contest Rules

To determine whether one library is better than another, HALO uses an enhanced version of Lau et al.'s statistical analysis and measurement technique. Instead of measur-

[3] A comparison based on a statistical test derived from Student's t distribution [Student, 1908].

Input: *deployedLib* — The currently-deployed library
otherLib — The other competing library

```
1  if first invocation of the bakeoff decision procedure then
2      ClearObservations(deployedLib)
3      ClearObservations(otherLib)
4  end
5  deployedObs ← GetObservations(deployedLib)
6  if deployedObs does not have a new observation then
7      transitionTo Bakeoff
8  end
9  otherObs ← GetObservations(otherLib)
10 comparisonResult ← CompareWithStudentsT(deployedObs, otherObs)
11 switch comparisonResult do
12     case GreaterThan do transitionTo Payback
13     case LessThan do
14         swap deployedLib with otherLib
15         transitionTo Payback
16     case NoAnswer do
17         if deployedObs has no fresh observations then
18             transitionTo Bakeoff
19         end
20         StepsUntilSwap ← StepsUntilSwap − 1
21         if StepsUntilSwap = 0 then
22             if Swaps ≥ MAX then the bakeoff has timed out
23                 deploy bestLib
24                 transitionTo Payback
25             end
26             swap deployedLib with otherLib
27             Swaps ← Swaps + 1
28             reset StepsUntilSwap
29         end
30         transitionTo Bakeoff
31     end
32 end
```

Algorithm 6.1: The decision procedure for the **Bakeoff** state of the adaptive tuning manager.

ing running-times, HALO relies on indirect quality metrics to rate different versions of a tuning section. Indirect metrics are used to avoid overhead as much as possible. During development, the highest-overhead method, sampling-based profiling, incurred less than 5% overhead when enabled. The two quality metrics available are (1) the average number of instructions retired per CPU cycle (also referred to as an IPC),[4] and (2) the average number of calls to the tuning section's root function within a fixed time period (also referred to as the call frequency). Under both metrics, we consider a higher value to mean better quality, but sometimes better quality does not result in better performance due to the indirect nature of measurement.

For example, a higher call frequency may indicate that the program is performing better, since it is able to call the function more often. But, a sudden rise in call frequency may have occurred simply because the function's inputs or program workload have changed during the bakeoff. A higher IPC is also an indicator the code is performing better, since it is able to retire more instructions in the same amount of time while executing in the tuning section. The IPC metric is less dependent on the function's input than a running-time. For example, the IPC can remain steady if a loop's trip-count depends on an input to the function, while the call frequency or execution time will always be a function of the trip-count. But, the IPC still maintains some dependence on the data being processed by the function and the type of instructions used. Each control-flow path within a loop can have different IPCs, and even the same CPU instruction can require more cycles to complete, depending on its inputs and the state of the cache [Fog et al., 2011]. Additionally, vector instructions perform more work for each instruction retired, so comparing the IPCs from vectorized and non-vectorized code is not useful for determining performance.

Comparisons In Algorithm 6.1, the function COMPAREWITHSTUDENTST takes two sets of quality measurements and tries to determine whether one set has a larger mean than

[4]The IPC is an approximation that assumes a 1Ghz clock frequency (Section 5.3.1).

Table 6.1: Example quality observations for two libraries X and Y during a bakeoff.

	X	Y
	4.63284	4.80113
	5.50236	5.96274
	5.29374	4.95499
	4.5823	–
	4.08597	–
Mean	4.81944	5.23962
Variance	0.3301	0.3981
Std. Dev.	0.5745	0.631

the other. Because the variance and number of observations in each set can vary, the function uses a classic statistical inference to make its determination about the means of the two sets [Devore and Berk, 2012]. The statistical inference used by HALO provides a strong degree of confidence about the differences between the two means and relies on a test derived from Student's t distribution [Student, 1908].

If we did not take the variance and number of observations into consideration, it would be difficult to tell whether the difference between means is significant. Take for example the following quality observations for two libraries X and Y in Table 6.1. According to the sample means $\bar{X} = 4.81944$ and $\bar{Y} = 5.23962$, library Y appears to be 8.7% better than X. But, is this difference meaningful with respect to the observations? The sample variances, which are the sum of squared differences between each observation and the mean, are roughly the same, but the standard deviations (the square-root of the variance) might be too large. In fact, the standard deviation is roughly 0.6, which is larger than the difference between the means, 0.42, so the difference could just be profiling noise. With statistical inference, we determine the probability that library Y is greater than X based on these observations, giving us confidence about whether there truly is a difference.

Now, we discuss statistical inference by turning our attention to the implementation of the COMPAREWITHSTUDENTST function. Suppose we have two libraries for a tuning section, called A and B, with corresponding quality measurements $A_1 \ldots A_n$ and $B_1 \ldots B_m$.

To apply statistical inference, we first make the assumption that the A's and B's are independent and are drawn from normally-distributed populations with true means μ_A and μ_B. The ultimate goal is to determine whether the difference between the sample means, $\bar{A}$ and $\bar{B}$, is significant. We proceed with a two-sample t test with null-hypothesis H_0 and alternate hypotheses H_a and H_b:

$$\begin{aligned} H_0 &: \mu_A - \mu_B = 0 \\ H_a &: \mu_A - \mu_B > 0 \\ H_b &: \mu_A - \mu_B < 0 \end{aligned} \tag{6.2}$$

To perform the test, we take H_0 as our initial assumption, *i.e.*, that the two libraries have the same performance. Based on the quality measurements for `A` and `B`, we try to reject our initial assumption H_0 with strong confidence in favor of one of the two alternatives: H_a supposes that `A` strictly better than `B`, in terms of quality, whereas H_b supposes `B` is strictly better.

A t test gives us confidence estimates for these two alternate hypotheses. The higher the confidence level, the more certain we are that one of the alternative hypotheses is true, *i.e.*, that there is a significant difference in performance between the two versions. With s_A^2 representing sample variance of `A`, the test statistic t becomes

$$t = \frac{\bar{A} - \bar{B}}{\sqrt{\frac{s_A^2}{n} + \frac{s_B^2}{m}}} \tag{6.3}$$

and the degrees of freedom v estimated from the data is

$$v = \left\lfloor \frac{\left(\frac{s_A^2}{n} + \frac{s_B^2}{m}\right)^2}{\frac{(s_A^2/n)}{n-1} + \frac{(s_B^2/m)}{m-1}} \right\rfloor \tag{6.4}$$

We reject H_0 in favor of H_a with 95% confidence if $t \geq t_{.95,v}$, returning *GreaterThan* from CompareWithStudentsT. The threshold value $t_{.95,v}$ is the corresponding probability-value threshold[5] for Student's t distribution [Mandel, 1964, Table II]. Similarly, we reject H_0 in favor of H_b if $t \leq -t_{.95,v}$ and return *LessThan*. If neither or both of those inequalities are true, then we cannot determine with certainty whether `A` or `B` is better, so the comparison returns *NoAnswer*.

Using the data from Table 6.1 for libraries `X` and `Y`, let us determine whether library `Y` is better than `X` with CompareWithStudentsT. Our null-hypothesis H_0 is $\mu_X - \mu_Y = 0$ and alternative hypothesis H_Y is $\mu_X - \mu_Y < 0$, which postulates that `Y` is better than `X`. We ignore the other alternative hypothesis since $\bar{Y} > \bar{X}$. Computing based on Equation 6.3 and Equation 6.4, the test statistic $t = -0.9426$ and degrees of freedom $v = 0$. The probability-value threshold $t_{.95,0}$ is greater than 6.314 according to Mandel [1964, Table II]. Because t is not less than $-t_{.95,0}$, we do *not* reject our null-hypothesis in favor of hypothesis H_Y, so CompareWithStudentsT returns *NoAnswer*.

Lau et al. [2006] does not specifically mention two-sample hypothesis testing as we have described, but we believe that it is essentially what they use. What they describe is that is they construct a normal distribution with mean $|\mu_A - \mu_B|$ and variance $\frac{s_A^2}{n} + \frac{s_B^2}{m}$ to derive conclusions based on confidence intervals. This method is close to what is described earlier if a two-sample z test with null-hypothesis $H_0 : |\mu_A - \mu_B| = 0$ were used instead of a t test. Fundamentally, a z test either requires knowing the population variance ahead-of-time, or relying on the law of large numbers and using the sample variance in place of the population variance [Devore and Berk, 2012]. Lau et al. use the latter approach of requiring a large number of samples.

In practice, the population variance is unknown, so Lau et al.'s test required roughly 1,000 samples to discern a 10% speedup. Our use of a t test instead of a z test allows us to

[5]These threshold values pre-computed and stored in a static look-up table within HALO server.

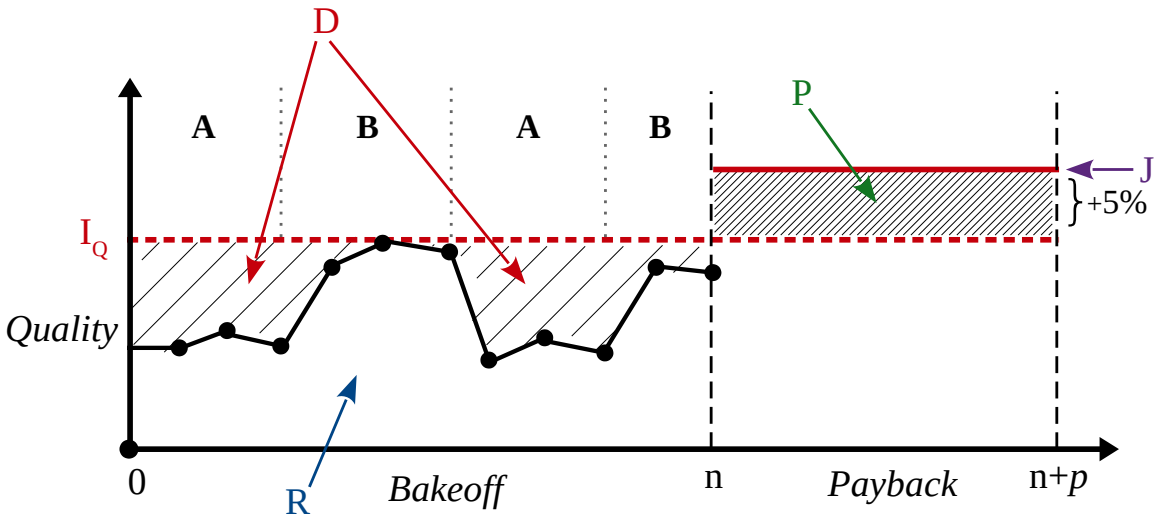


Figure 6.2: An overview of the calculation to determine how to amortize a bakeoff's overhead during the **Payback** state. Figure is not to scale.

make inferences with fewer numbers of samples. Specifically, we estimate the degrees-of-freedom v to use Student's t distribution instead of relying on the law of large numbers. It is important to recognize, however, that as the number of samples grows, the t distribution turns into a normal distribution. Thus, with a large number of samples, Lau et al.'s approach to comparing two means is essentially the same as HALO's.

6.2.2 Debt Repayment

Following the bakeoff, we transition from the **Bakeoff** state to the **Payback** state that helps offset the overhead of the bakeoff. Every bakeoff has some cost, or overhead, associated with it because profiling is enabled. With profiling enabled, clients experience a roughly 5% reduction in performance. The **Payback** state is designed around the idea of amortizing the cost of performing a bakeoff through a period of exploiting the best known library following the bakeoff with no profiling overhead. During the first step of the **Payback** state, HALO calculates the number of steps needed to amortize the bakeoff by examining the bakeoff's history.

Suppose we have a bakeoff history H for libraries `A` and `B` containing n quality observations, where at each time step i, the deployed library saw fresh profiling data, and thus H_i is a fresh quality rating for that library. Figure 6.2 contains an example plot of the bakeoff history $H_0 \ldots H_n$ with the horizontal axis t representing discretized time and the vertical representing quality. We can see that library `B` performed better than `A`, so that is the deployed library upon transitioning to the **Payback** state. The real performance R exhibited by the clients during the bakeoff is the area under the curve formed by H, *i.e.*, performance is the total quality during a period of time:

$$R = \int_0^n H_t \, \mathrm{d}t \approx \sum_{i=1}^{n} \left(\Delta t \times \frac{H_i + H_{i-1}}{2} \right) \tag{6.5}$$

The value R is directly computed as a Riemann sum using the midpoint rule. Had we not performed any switching during the bakeoff and only used `B`, ideally we would have observed a constant quality I_Q, the maximum observed quality of `B` in H. Knowing this ideal quality, we compute the debt D incurred during the bakeoff as:

$$I = I_Q \times n$$
$$D = |I - R|$$

Since we pause while using library `B` in the **Payback** state with sampling-based profiling disabled, we estimate the performance of `B` during debt repayment to be roughly 5% higher[6] than I_Q. It is through pausing while profiling is disabled that we pay off our debt D. Finally, we compute the total payment P and the number of payback steps p as:

[6]This 5% estimate is based on empirical overhead testing with HALO server during development.

$$J = I_Q \times 1.05$$
$$p = \left\lceil \frac{D}{J - I_Q} \right\rceil$$
$$P = (J - I_Q) \times p \geq D$$

After computing p, the decisions procedure for the **Payback** state transitions to itself p times before transitioning to the **Decide** state.

6.3 Exploration

So far, we have considered the decision-making process and how to evaluate a candidate library, but not how the candidate libraries are chosen. In order to adapt, HALO explores the space of knob configurations through two types of library selection decisions made by the Adaptive tuning manager: **Experiment** and **Retry**.

Experiment The first kind of exploration is the decision to transition to the **Experiment** state (Figure 6.1), which generates a completely new library based on a knob configuration that is chosen by the automatic tuning strategy (Chapter 7). After submitting a compile job to the thread pool, the Adaptive tuning manager transitions to **Compile** and waits for the job to complete. During this wait, the client continues to execute using the best-known library, without any profiling enabled. If the resulting library's object file is identical to an existing library (as determined by its SHA1 hash), then the new configuration is added to the existing library; effectively merging the two libraries. This merging of libraries is not an uncommon occurrence because not all knob changes will result in changes to how the program is compiled. Detecting duplicate libraries and merging them allows HALO server to use its compiler to rapidly search the configuration space without

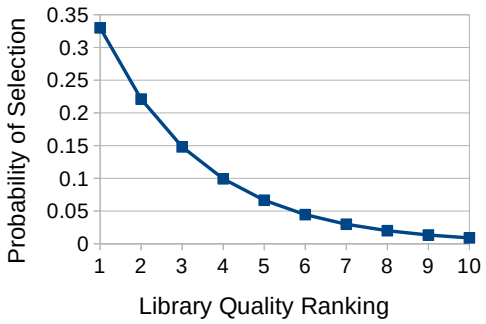


Figure 6.3: Probability of selecting library i in the **Retry** state, with the libraries ordered by descending quality.

involving the clients. Once a duplicate library is found in the **Compile** state, we transition back to **Experiment** to consult the search method for the next configuration to try. To prevent an infinite loop due to duplicate compiles, the number of backward transitions from **Compile** is limited. Once the limit is reached, an exiting library is chosen uniformly at random to be the candidate instead of compiling.

Retry The second kind of exploration is to try one of the existing libraries, excluding the current-best. Retrying a library can be beneficial if the workload or environment has changed to favor a library that was previously considered unfavorable. To select an existing library, we first rank the libraries in descending order according to the average quality following their most recent bakeoff. Then, we perform a random selection process that consists of repeated Bernoulli trials, so that we bias the selection towards better-performing libraries. We begin at the first-ranked library and flip a coin that yields *heads* with probability 1/3. If the coin yielded *heads*, the library is chosen to compete in a bakeoff, otherwise we move to the next library and flip the coin again. The probability distribution induced by this selection process follows a geometric distribution with probability of success 1/3, as illustrated in Figure 6.3.

6.4 Rewards

During the transition from **Payback** to **Decide**, the reward assigned to the decision is selected based on the result of the bakeoff. The reward values were manually chosen to fall within the range $[-1, 1]$, with positive values indicating desirable outcomes:

- **Candidate Won Bakeoff** $= 1$
- **Bakeoff Timed-out** $= -0.5$
- **Candidate Lost Bakeoff** $= -1$

The rewards are fixed values because the cost of a bakeoff is amortized (Section 6.2.2). But, the *outcome* of each bakeoff is still unknown, which is why the decision problem in the **Decide** state is modeled as a multi-armed bandit. The range of rewards are centered around zero because the **Pause** action always yields a zero reward. A bakeoff times-out because of a competition between two similar-performing libraries. Since no bakeoff is completely free, a small negative reward is given in the case of a time-out to discourage fruitless bakeoffs. To encourage early exploration, the initial estimated reward for the **Explore** action is set to $1/3$, with the other actions initially estimated to yield a reward of zero.

Chapter 7

Automatic Tuning

Autotuning in HALO is focused on tuning LLVM's optimization and compilation pipeline for the code contained in each tuning section (Section 5.3). The autotuner's *search method* is a heuristic that explores the configuration space by selecting interesting candidates from the space based on the quality of candidates selected already. Section 7.1 discusses the types of LLVM compiler optimizations that are tuned by HALO for each tuning section. The remaining sections of this chapter discuss the search methods used during tuning.

7.1 Compiler Optimization Tuning

Each tuning section is considered to be a fresh instance of a tuning task. The tuning is performed seamlessly while the client processes are executing (online) using dynamic code patching (Section 5.2.3). Non-root functions included in the tuning section are made private to the tuning section's compilation unit, which is an LLVM IR module. This way, cross-function specializations or inlining that normally would be impossible can occur during tuning. Global values, especially those which are mutable, are marked as external to the module so that dynamic linking resolves them to the correct symbols in the client process. Additionally, when the unoptimized module is initially created for a tuning

section, all natural loops within functions are given module-unique names to facilitate the tuning of optimizations for a particular loop.

Table 7.1 details all of the current knobs tuned by HALO. Because we create a set of knobs for each loop identified in the module, the size of the configuration space depends on the tuning section. All knobs may also be set to a value indicating that the setting is not specified, leaving it up to LLVM to decide or use its default value. Overall, there are nearly $6.0852l \times 10^{19}$ possible configurations in the space searched by HALO, where l is the number of loops in the tuning section. Table 7.1 is separated into three groups of knobs. The first group are knobs which change an aspect of the LLVM compiler for the entire module. For example, "Experimental Alias Analysis" enables non-default alias analysis infrastructure when optimizing the code. All of these non-speculative general knobs control an existing aspect of LLVM's optimizations or code generation. I refer to this group as *non-speculative* because I believe the knobs in the group are only beneficial. These non-speculative knobs may exist in LLVM for a few different reasons: (1) the option's net benefit for application runtime is not usually worth the extra compile-time required, (2) the option may generate code that is not fully supported for the commonly-targeted CPU or ABI version, or (3) the option controls a feature that is a work-in-progress. All of the other knobs are *speculative* in the sense that they control an aspect of the compiler's optimizations that requires some cost modeling or have trade-offs, and thus are more likely to yield a performance regression. Specifically, speculative knobs do not only exist to save on compile time for common cases. The remainder of this section details what the speculative knobs tuned by HALO control and why they were chosen for tuning.

7.1.1 Function Inlining

The LLVM function inliner's goal is to eliminate call overhead and specialize the called function for a particular call-site. The *Inlining Threshold* knob sets an abstract cost thresh-

Table 7.1: Settings tuned by HALO for each tuning section. All options are integers. The Default column indicates LLVM's default setting for the given option.

General Knobs (non-speculative)	**Options**	**Default**
Use Function Attributor Pass	$i \in [0,1]$	0
Use Partial Inliner	$i \in [0,1]$	0
Use Unroll-and-Jam	$i \in [0,1]$	0
Use NewGVN	$i \in [0,1]$	0
Use NewGVN Hoist	$i \in [0,1]$	0
Use GVN Sinking	$i \in [0,1]$	0
Extra Vectorizer Passes	$i \in [0,1]$	0
Experimental Alias Analysis	$i \in [0,1]$	0
Tune for Native CPU	$i \in [0,1]$	0
Use Interprocedural Register Allocation	$i \in [0,1]$	0
Use PBQP Register Allocator	$i \in [0,1]$	0
Optimization Pipeline Level	$i \in [2,3]$	–
Codegen Optimization Level	$i \in [2,3]$	–
General Knobs	**Options**	**Default**
Inlining Threshold	$\{100i \mid i \in [0,30]\}$	2.25
Jump-threading Threshold	$i \in [0,100]$	6
SLP Vectorization Cost Threshold	$i \in [-50,50]$	0
Use Loop Prefetching for Writes	$i \in [0,1]$	0
Loop Prefetching Distance	$i \in [0,100]$	0
LICM Versioning Threshold	$i \in [0,100]$	*disabled*
Loop Interchange Cost Threshold	$i \in [-100,100]$	*disabled*
Knobs Per Loop in Module	**Options**	**Default**
Unrolling Factor ($i = 0$ disabled)	$\{2i \mid i \in [0,16]\}$	*unset*
Disable Runtime Unrolling	$i \in [0,1]$	*unset*
Enable Loop Distribution	$i \in [0,1]$	*unset*
Vectorization Width ($i = -1$ disabled)	$\{2^i \mid i \in [-1,4]\}$	*unset*
Interleaving Count ($i = -1$ disabled)	$i \in [-1,4]$	*unset*

old that helps avoid excessive function inlining that hurts program performance. Setting this knob to a higher value indicates a higher tolerance for risk, so more aggressive inlining will happen. A number of existing works have found that function inlining is worth tuning [Cavazos and O'Boyle, 2005; Kulkarni et al., 2013; Lau et al., 2006; Waterman, 2006]. For LLVM, a higher threshold value indicates a stronger willingness to perform inlining at a call-site, because the threshold sets the tolerance for program size increase from inlining a call-site. But the threshold cannot be described in a simple way due to the ad-hoc nature of the cost model's decision procedure. Based on my examination of the source code, there are a number of ad-hoc bonuses and multipliers that scale the estimated costs based on properties of the targeted CPU and the callee. For example, a bonus is applied "if the callee has a single reachable basic block at the given callsite context."

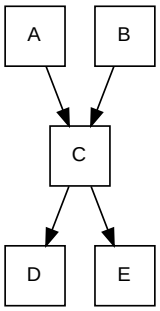
7.1.2 Jump Threading

In LLVM, *jump threading* duplicates basic blocks within a function to eliminate redundant conditionals and expose additional optimization opportunities by generating new control-flow paths within a function. Specifically, from the LLVM source code:[1]

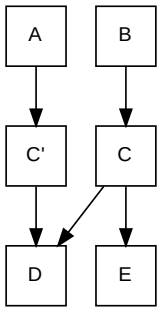
> [Jump threading] looks at blocks that have multiple predecessors and multiple successors. If one or more of the predecessors of the block can be proven to always jump to one of the successors, we forward the edge from the predecessor to the successor by duplicating the contents of this block.

Thus, jump threading is a speculative optimization that achieves an effect similar to merge splitting, which was performed in the context of Jikes RVM (Section 3.1.3). The threshold value tuned by HALO controls the maximum number of instructions allowed in the duplicated basic block, with a value of zero disabling jump threading.

[1] Found in `include/llvm/Transforms/Scalar/JumpThreading.h` from LLVM 11.



(a) Before jump threading.



(b) After jump threading.

Figure 7.1: An example of jump threading when it is proven that A always flows to D.

The example of jump threading in Figure 7.1 considers a simple control-flow graph, where C is the basic block with multiple successors and predecessors. Suppose that in basic block A, we have the assignment $x \leftarrow 5$, and C branches to D only if $x > 0$. Static analysis can then prove that any control-flow out of A will always take the edge $\texttt{C} \rightarrow \texttt{D}$. At the cost of increased code size, jump threading gives A its own copy of block C, called C', that unconditionally goes to D. Now, any code in C' can be specialized with respect to its single predecessor A, without having to consider B.

7.1.3 SLP Vectorization

First introduced by Larsen and Amarasinghe [2000], SLP Vectorization is an optimization that increases the amount of superword-level parallelism (hence the name "SLP") within a basic block. It is a form of automatic vectorization that focuses on identifying groups of independent instructions whose operands can be replaced with vectors; so the group's operations can happen in parallel with a single vector operation (Figure 7.2). The cost of performing SLP vectorization is the additional work required to transfer each individual value to and from a vector register, *i.e.*, packing and unpacking the values. HALO tunes an abstract profitability threshold used within LLVM's cost model that controls whether to perform SLP vectorization, with negative values making the optimizer more willing to

```
a = b + c[0]
d = e + c[1]
f = g + c[2]
h = i + c[3]
```

(a) Before SLP vectorization.

```
vecC = *c              // load
vecT = {b, e, g, i}    // pack
vecOut = vecT + vecC
a, d, f, h = vecOut    // unpack
```

(b) After SLP vectorization.

Figure 7.2: An example of SLP vectorization, using a syntax similar to C.

vectorize.

7.1.4 Loop Prefetching

Software prefetching is an optimization that recognizes memory access patterns within loop bodies and decides whether to insert cache prefetching hints for the CPU to reduce memory stalls. On the x86-64, these hints are special CPU instructions that provide a hint to the CPU that the cache-line referenced by the given address should be brought into the first-level cache because it will be used soon [Intel, 2020]. Too many prefetch instructions, prefetches that are too early, or prefetches that are only conditionally used, can pollute the first-level cache. On the other hand, a well-timed prefetch can significantly improve performance by reducing stalls on memory reads. Software prefetching has historically been a prime candidate for adaptive optimization [Chilimbi and Hirzel, 2002; Lee et al., 2012; Lu et al., 2004; Saavedra and Daeyeon Park, 1996]. The knob controlling distance is in terms of number of instructions between the prefetch instruction and the first use of data on that cache line. This knob is the primary way that HALO tunes LLVM's software prefetching pass. A distance of zero disables the use of software prefetching. HALO also tunes a Boolean flag that controls whether LLVM will insert prefetch hints for memory writes, which can help reduce write stalls.

7.1.5 LICM Versioning

Loop-invariant code motion (LICM) is an optimization that identifies instructions within a loop's body that either provide the same result across all loop iterations, or otherwise does not need to be performed on each iteration, and moves such instructions outside of the loop. The most important types of instructions that benefit from LICM involve memory access [Muchnick, 1997]. In LLVM, a pass that runs just prior to LICM, called *LICM versioning*, helps expose additional opportunities to move memory accesses out of loops. The LICM versioning pass identifies loops that would fail to benefit from LICM due to conservative assumptions about memory aliasing. Then, a duplicate version of the loop is created that makes optimistic assumptions about memory aliasing. The optimistic assumptions that enable code motion are checked dynamically, then either the conservative loop or the optimistic loop is chosen based on the result of the test. HALO tunes the LICM versioning threshold value that represents a minimum percentage of instructions in the loop's body that must appear to be loop invariant. A higher threshold value makes LICM versioning less likely to activate and duplicate a loop.

7.1.6 Loop Interchange

Loop interchange is a loop transformation that reorders a nesting of loops, which can be used to improve spatial cache locality or aid in performing loop vectorization [Kennedy and Allen, 2002]. In LLVM, the loop interchange pass focuses on identifying loop nests with memory access patterns that may be improved by reordering the nesting of loops. A classic example of loop interchange is when iterating through a large two-dimensional array. If the hardware cache is better-suited for a row-major access pattern, but the programmer wrote the array iteration in a column-major ordering, loop interchange can transform the loop nest to row-major. The knob tuned by HALO is a threshold used by LLVM's abstract cost model that is compared with the estimated cost of a reordering to determine

whether interchange will happen.

7.1.7 Loop Unrolling

Loop unrolling is an optimization that copies the body of a loop multiple times and adjusts the loop's condition (Figure 7.3). The number of times the loop's body will be copied is referred to as the *unrolling factor*. Unrolling reduces the number of loop-control instructions executed, because it reduces the loop's *trip count*, *i.e.*, the number of times the loop iterates. But more importantly, copying the loop body can create additional opportunities for better instruction scheduling and redundancy elimination [Muchnick, 1997]. One downside is that a high unrolling factor increases the size of the code. HALO tunes the unrolling factor for each individual loop, *i.e.*, one unrolling-factor knob is created for each loop in the tuning section. Many successful autotuners have focused on tuning loop unrolling factors [Hartono et al., 2009; Stephenson and Amarasinghe, 2005; Tiwari and Hollingsworth, 2011; Yi et al., 2007].

If a loop's trip count is not known statically to be a multiple of the unrolling factor, unrolling the loop will result in two loops: one rolled loop to handle the remainder and then the unrolled loop (the first and seconds loops of Figure 7.3b, respectively). This type of loop unrolling is referred to as *runtime unrolling* in LLVM and can be disabled on a per-loop basis. If the *Disable Runtime Unrolling* knob is enabled, then the loop associated with that knob is only unrolled if the loop's trip count is known to be a multiple of the unrolling factor, so that no additional loop is produced.

An additional form of loop unrolling for nested loops, called *unroll-and-jam*, unrolls the outer loop multiple times and then combines (or jams) the copies of the inner loop together [Kennedy and Allen, 2002]. HALO only tunes LLVM's unroll-and-jam pass by enabling or disabling the pass, instead of tuning unroll-and-jam on more finely by specifying an unroll-and-jam factor on a per-loop basis. Additional static analysis would be

```
for (int i=0; i < n; i++) {
  f(i);
}
```

(a) Original loop.

```
int i = 0;
int rem = n % 4;
for (; i < rem; i++) {
  f(i);
}
for (; i < n; i+=4) {
  f(i);
  f(i+1);
  f(i+2);
  f(i+3);
}
```

(b) After unrolling.

Figure 7.3: An example of loop unrolling, using an unrolling factor of four.

needed in HALO to identify loops for which unroll-and-jam may apply.

7.1.8 Loop Vectorization

Automatic loop vectorization is an old but important topic in compilers for high performance computing [Allen and Kennedy, 1987; Padua and Wolfe, 1986]. Given a loop with no dependencies between each iteration, it is possible to execute the loop iterations in parallel, because the iteration order does not matter. One way to execute loop iterations in parallel is to transform the loop such that vector instructions are used to perform the work in parallel [Kennedy and Allen, 2002]. Thus, the goal of automatic loop vectorization is to leverage modern hardware's availability of vector registers and operations. As a rough analogy, loop vectorization can be thought of as loop unrolling (Section 7.1.7) followed by SLP vectorization on the unrolled loop's body (Section 7.1.3). But, loop vectorization normally focuses specifically on parallelizing loop iterations that operate on arrays, since vectorization is more efficient if the values are already packed together in memory.

HALO tunes three aspects of the automatic loop vectorization process in LLVM. The first aspect is *vectorization width* that controls the number of loop iterations to perform in parallel. Vectorization width is analogous to an unrolling factor and is tuned on a per-

loop basis, but HALO only considers widths that are a multiple of two, because vector registers do not hold an odd number of elements. The other two aspects of vectorization, interleaving [Nuzman et al., 2006] and distribution [Kennedy and Allen, 2002, Section 6.2.2], control more complex situations that arise during vectorization. All of these three aspects are tuned on a per-loop basis, *i.e.*, three separate knobs controlling vectorization are generated for each loop in the tuning section.

7.2 Random Search

When attempting to search a large space of knob configurations, simple random search over the entire space is easy to implement and a good first strategy to try. Seymour et al. [2008] compared a number of more sophisticated search heuristics in the context of empirical autotuning for compilers, including Nelder-Mead simplex, genetic algorithms, and simulated annealing. They found that random search is the most effective overall strategy. The primary reason, according to Seymour et al., is that the search spaces are not overly complex: "there are many points with performance within 5% of the true maximum." Knijnenburg et al. [2003] also found that random search can quickly find good performing configurations, which is very desirable for an overhead-sensitive system such as HALO. But, better-customized search strategies, such as the Nelder-Mead simplex algorithm in Active Harmony [Chung and Hollingsworth, 2004; Tiwari et al., 2009a], are more likely to avoid particularly bad configurations than random search for search spaces with few good configurations.

In HALO, we leverage two flavors of random search: local and global. Global random search is quite simple: for each knob, select among all possible options uniformly at random. Because a knob can be "unset" in HALO, all knobs in Table 7.1 are considered to have an additional option that tells the compiler to use LLVM's default setting.

Local Random Search Since a knob is defined as a range of integers, they allow us to loosely think of the distance between two knob configurations as some norm that combines the distances between each knob value in each configuration. To perform local random search, we define a *perturbing* function that accepts a knob configuration C plus a real-valued energy level $e \in [0, 1]$ and returns a configuration that is "nearby" the given configuration.[2] For each knob in the input configuration, the perturbing function draws values from a normal distribution defined over the knob's integer range, where the mean value is $C[k]$ for knob k. To help determine the variance of the distribution with respect to a knob, we define a vibration factor

$$V(knob, energy) = \frac{|knob.max - knob.min| \times energy}{2}$$

so that a higher energy level corresponds to a greater variance. Thus, the perturbing function produces a value nearby $C[k]$ for the new configuration's setting for k by first drawing v from the normal distribution $\mathcal{N}(C[k], \sigma^2)$, where $\sigma = V(k, e)$. Then, v is clamped within the knob's range and rounded to the nearest integer.

7.3 Surrogate Search

HALO's primary implementation of automatic tuning uses a hybrid strategy that combines a search heuristic with a model to help filter out bad configurations (Section 2.4). Specifically, HALO's *surrogate search* strategy combines random search with supervised learning to progressively model the configuration space during exploration. A fresh learning model is dynamically trained from scratch for each tuning section. The automatic tuning problem is viewed as an instance of black-box optimization, where the goal

[2]This notion is borrowed from work in simulated annealing [Bertsimas and Tsitsiklis, 1993], where a point in space is like a particle that is vibrating within its neighborhood according to its energy-level.

is to find a maximal input to a quality function f. In HALO, that f takes a knob configuration C and outputs the configuration's quality as a real number. To execute that black-box quality function, we must conduct a bakeoff that is expensive and time consuming. The purpose of the machine learning model is to *predict* the quality of a configuration, based solely on characteristics of the knob configuration. The model serves as a *surrogate* for conducting a bakeoff, allowing HALO to rapidly explore the configuration space and only attempt a bakeoff for high-quality candidates. This idea is an extension of work by Nelson et al. [2015], who used a surrogate to accelerate an exhaustive search of a small configuration space.

7.3.1 Bootstrapping

The importance of any particular knob in predicting the quality of a library is dependent upon the code contained in the tuning section. While other works have used a generic model that is trained to consider code features [Fursin et al., 2011], HALO trains a model specific to each tuning section. Thus, a process for bootstrapping the model is needed to generate data that the supervised model can learn from.

Bootstrapping consists of conducting bakeoffs using configurations selected by some other means, in order to gather some initial data points. The selection of initial configurations can have a dramatic effect on the number of search iterations required to find an optimal configuration [Balaprakash et al., 2013b]. Thus, the initial configurations consist of two hand-picked configurations that are expected to perform well, as well as some that are exotic. The first of these hand-picked configurations unsets all knobs and then sets only the optimization and code generation levels to `-O3`, along with enabling *Tune for Native CPU* (Table 7.1). The second hand-picked configuration is a superset of the first that additionally enables:

- Interprocedural Register Allocation

- The PBQP (Partitioned Boolean Quadratic Problems) Register Allocator
- The Function Attributor Pass
- Experimental Alias Analysis

These knobs were chosen entirely based on my own gut-feeling that these knobs are sensible to have enabled, but are not enabled by default in LLVM. The remaining initial configurations are randomly chosen to provide some variety in the initial batch of training data for the surrogate.

Input: *history* — a map from configurations to quality from its last bakeoff.
expertConfigs — current set of hand-picked configurations.
readyConfigs — current set of configurations selected for a bakeoff.
Result: a configuration to try in the next bakeoff.

```
1  minPrior ← 5
2  if readyConfigs = ∅ then
3  |  if history.size() < minPrior then
4  |  |  if expertConfigs ≠ ∅ then
5  |  |  |  return REMOVEONECONFIG(expertConfigs)
6  |  |  else
7  |  |  |  return GENUNIFORMRANDOMCONFIG()
8  |  |  end
9  |  else
10 |  |  readyConfigs ← readyConfigs ∪ SURROGATESEARCH(history,...)
11 |  end
12 end
13 return REMOVEONECONFIG(readyConfigs)
```

Algorithm 7.1: The surrogate-based configuration selection method.

Algorithm 7.1 summarizes the procedure for selecting a configuration while in the **Explore** state of the Adaptive tuning manager (Figure 6.1). All of the inputs to the procedure are mutable values. The function GENUNIFORMRANDOMCONFIG uses the global random search strategy from Section 7.2 to generate well-varied configurations to explore the configuration space.

7.3.2 Generating Configurations

The SURROGATESEARCH function (Algorithm 7.2) generates a set of configurations that are worth experimenting with in a bakeoff. Similar to Nelson et al. [2015], the generation process relies on supervised learning to estimate a function that predicts the quality of a configuration, based on the existing history of configurations and their bakeoff performances. In Algorithm 7.2, the approximated function is modeled as the surrogate $\mathcal{S}$ that allows us to quickly filter the configurations in $\mathcal{X}$. Each row of matrix $\mathcal{X}$ represents a single configuration, denoted $\mathcal{X}_r$ for some row r. The vector $\mathcal{Y}$ output by PREDICT contains the predicted quality of each configuration in matrix $\mathcal{X}$. Specifically, configuration $\mathcal{X}_i$ has the corresponding predicted quality $\mathcal{Y}_i$. By default, the *batchSize* is set to 10 and controls how often we retrain a new surrogate to predict the quality of another *exploreSize* configurations, which is on the order of hundreds.

To prevent future overfitting of the model to its own opinions, a small percentage of the configurations returned by SURROGATESEARCH are chosen uniformly at random instead of relying on the surrogate's quality estimate. This percentage is controlled by *batchPct* and by default 80% of the configurations returned are the surrogate's top predictions, with the rest randomly chosen. The configurations in $\mathcal{X}$ that are explored by the surrogate are made up, by default, of equal-parts global random selection and local random selection using the configuration perturbation discussed in Section 7.2.

While supervised learning requires many examples to be effective, each quality rating in HALO is associated with a library that can contain multiple knob configurations. Thus, the merging of equivalent configurations into libraries after compilation (Section 6.3) helps generate additional data to train the surrogate model, since those configurations will be assigned the same quality rating.

The underlying surrogate model is based on fairly standard gradient-boosted decision

Input: *history* — a map from config to quality from its last bakeoff
bestConfig — best config currently
exploreSize — number of configs to try with surrogate
explorePct — proportion of global versus local search
batchSize — number of configs requested
batchPct — proportion of random versus surrogate-chosen configs
Result: a set of configurations, of size *batchSize*

$\mathcal{X} \leftarrow$ empty matrix
$\mathcal{S} \leftarrow$ TRAINMODEL(*history*) `// create the surrogate`
$numGlobal \leftarrow \lceil exploreSize \times explorePct \rceil$
for $i = 1 \ldots exploreSize$ **do**
  if $i \leq numGlobal$ **then**
    $\mathcal{X}_i \leftarrow$ GENUNIFORMRANDOMCONFIG()
  else
    $\mathcal{X}_i \leftarrow$ GENNEARBYCONFIG(*bestConfig*, ...)
  end
end
$\mathcal{Y} \leftarrow$ PREDICT($\mathcal{S}, \mathcal{X}$) `// get predictions from surrogate`
$\mathcal{A} \leftarrow \{1 \ldots exploreSize\}$ `// the available config numbers`
$chosen \leftarrow \emptyset$
$numRandom \leftarrow \lceil batchSize \times batchPct \rceil$
for $k = 1 \ldots batchSize$ **do**
  if $k \leq numRandom$ **then**
    $chosen \leftarrow chosen +$ GENUNIFORMRANDOMCONFIG()
  else
    $j \leftarrow \text{argmax}_{i \in \mathcal{A}} \mathcal{Y}_i$ `// determine best config number`
    $chosen \leftarrow chosen + \mathcal{X}_j$
    $\mathcal{A} \leftarrow \mathcal{A} - j$ `// remove j from available`
  end
end
return *chosen*

Algorithm 7.2: Implementation of SURROGATESEARCH that uses a model to find candidate configurations for future bakeoffs.

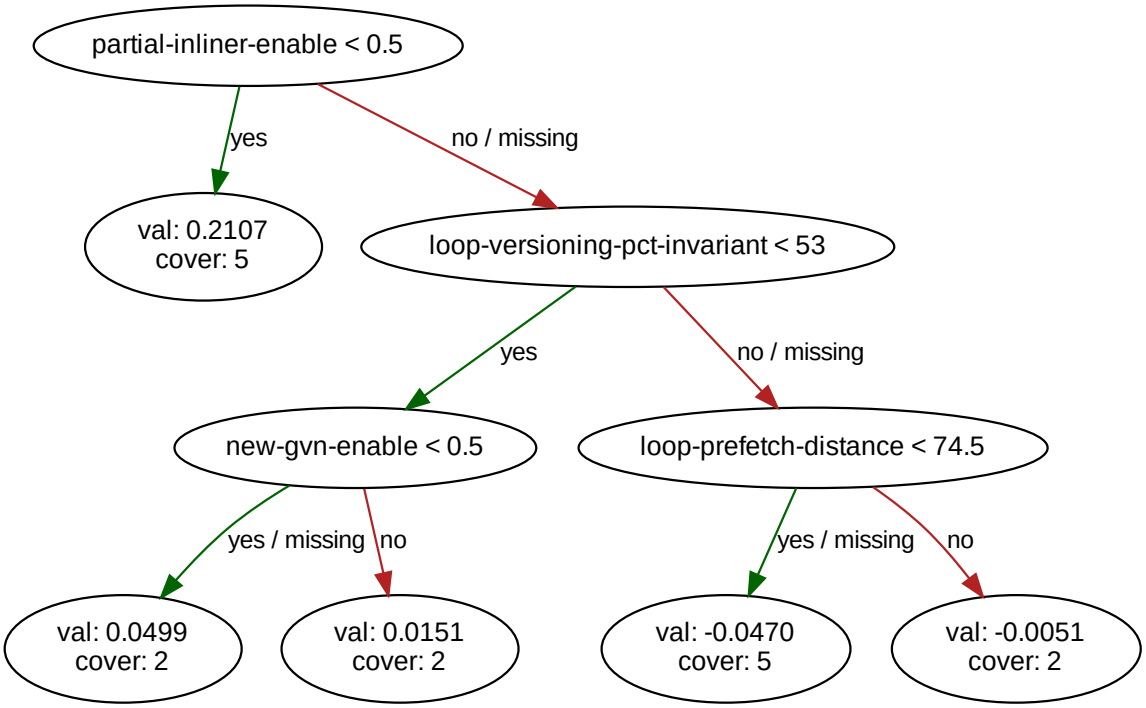


Figure 7.4: An example decision tree from the **spectralnorm** benchmark.

trees [Friedman, 2001, 2002].[3] The TRAINMODEL procedure used in Algorithm 7.2 employs simple cross-validation with a randomly-selected held-out set for testing, and the learning objective function uses regression with squared loss. Each knob within a configuration corresponds to a feature of the model. The initial prediction of all instances prior to training, *i.e.*, the global bias, is set to the mean of the entire dataset. Decision trees were chosen as the learner of the model for two practical reasons: (1) decision trees are somewhat easier to read in debugging dumps than other learned models, and (2) there are high-quality implementations accessible from C++, which is HALO's implementation language. Figure 7.4 is an example of one decision tree in the trained model that was obtained after 60 executions of the **spectralnorm** benchmark on the **workstation** machine from Chapter 8.

At each leaf of the decision tree, *val* is an indicator of the quality of matching configu-

[3]The model is implemented using XGBoost's C library [Chen and Guestrin, 2016].

rations (with larger values meaning better quality), and *cover* tells us how many samples in the training set matched that node. The tree in Figure 7.4 appears to have overfit, probably because of a small number of examples. But, the first configuration that is at least 1.75× better for **spectralnorm** came as a suggestion from SURROGATESEARCH.

Chapter 8

Experimental Results

This chapter provides an evaluation of the capabilities of a prototype of HALO, as described in earlier chapters, across a number of C and C++ benchmark programs and machines. We start with an empirical performance comparison that compares HALO with ahead-of-time optimization and traditional just-in-time optimization (Section 8.3). Then, we take a deeper look at the two quality metrics in HALO in Section 8.2. Finally, in Section 8.4 we consider the overhead of compiling and distributing binaries that are compatible with HALO when the server is not available.

8.1 Experiment Setup

A number of benchmark programs from the LLVM compiler's test suite were chosen to evaluate HALO. Programs were primarily chosen if they are not heavily reliant on external libraries, because such library code will not be visible to HALO for optimization because of the lack of source code for the libraries.[1] The prototype of HALO only selects and optimizes one tuning section. Thus, the programs were also chosen because they have one consistent tuning section, as identified by HALO, to ensure consistency during

[1] Also, the HALO prototype does not try to merge LLVM modules from separate compilation units.

the evaluation. The benchmark programs are:

- **lpbench** — a modernized version of the classic LINPACK benchmark [Dongarra et al., 2003]. Matrices of size 1000×1000 are used. Benchmark consists of 262 lines of C code.

- **matrix** — a synthetic benchmark that multiplies square matrices of size 256×256. Benchmark consists of 62 lines of C++ code.

- **n-body** — a benchmark that performs a double-precision N-body simulation. It models the orbits of Jovian planets, using a symplectic-integrator [Gouy, 2020]. Benchmark consists of 139 lines of C code.

- **perlin** — a program that applies Perlin [1985, 2002]'s noise filter to arbitrary data. Benchmark consists of 74 lines of C code.

- **spectralnorm** — a synthetic benchmark that computes the spectral norm of a 2000×2000 matrix using the power method [Gouy, 2020]. Benchmark consists of 62 lines of C code.

- **sphereflake** — a ray-tracer that utilizes a bounding-volume hierarchy for collision detection. An image with dimensions 1280×1280 pixels is rendered for a scene resembling a three-dimensional Koch [1904] snowflake made of reflective spheres. Benchmark consists of 224 lines of C++ code.

These programs perform one heavy-duty task so that we can evaluate the most important aspects of HALO within a reasonable amount of time. Specifically, the majority of the activity in these programs happens within a single tuning section, so that we can explore the the long-term behavior of the HALO prototype.

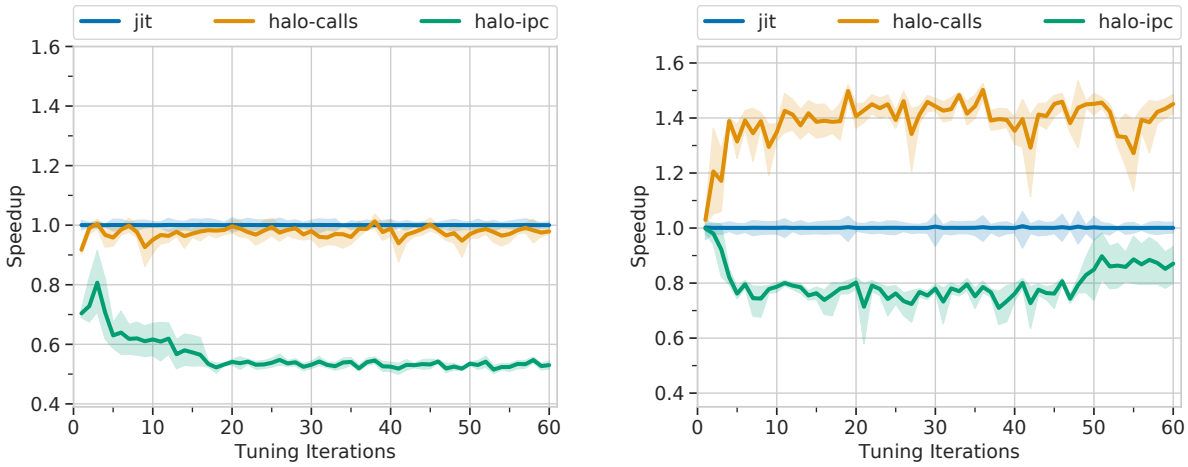
A *trial* tests a benchmark by repeatedly executing it on its fixed workload for a certain number of *tuning iterations*. Time is measured for each tuning iteration for complete

program executions using the GNU `time` command. We perform five trials of each benchmark and report the average running time of each tuning iteration. For example, the reported value for a program's ninth tuning iteration is based on the average across the five trials of the running time recorded for the ninth tuning iteration. For experiments where no tuning will happen (*i.e.*, the benchmark was compiled without HALO support), we assume all tuning iterations after the first will have the same results, since the program will not change dynamically.

Machine Specifications The benchmark programs come with one fixed workload and it is difficult to fabricate different workloads for these benchmark programs to test adaptive optimization. Thus, we evaluate the effectiveness of HALO for another piece of latent information: hardware differences. We evaluate HALO with three types of machines:

- **workstation** — Two sixteen-core Intel Xeon Gold 6142 CPUs, each with a 2.6 GHz base frequency and 22 MB L3 cache.
- **desktop** — Four-core Intel i7-3770 CPU with a 3.4GHz base frequency and 8MB L3 cache.
- **mobile** — Broadcom BCM2837B0 that consists of four ARM Cortex-A53 cores with a 1.4GHz base frequency and 512 KB L3 cache; running in 32-bit mode (`armhf`) with fan-cooled heatsinks.

During each tuning iteration, the computational kernel of the benchmark is repeated a number of times to emphasize its execution-time overhead. The workloads placed on these computational kernels are scaled up so that each tuning iteration requires roughly 5–30 seconds to complete on the **desktop**. When running on **mobile**, the amount of work per iteration is halved to save time, because the **mobile** machine is two to four times slower than the others.



(a) lpbench (b) matrix

Figure 8.1: Comparing the Call and IPC metrics on the **workstation**. Higher speedups are better.

8.2 Quality Metrics

There are two quality metrics in HALO: call frequency and IPC (instructions per cycle). Figure 8.1 shows two examples where the IPC metric makes performance significantly worse over time. Deeper investigation into why the IPC metric is non-competitive has revealed that the measure's accuracy is good, *i.e.*, it does detect that the IPC has increased, but higher IPC alone does not mean a faster program. For this investigation, HALO's adaptive tuning manager was modified to permanently transition into the **Pause** state once a knob configuration at least 20% better than the original library is found. Using the Linux `perf-stat` profiling tool for the **matrix** benchmark running under HALO gives the following statistics for one complete workload execution:

	JIT-once	Adaptive + IPC Metric
seconds elapsed	24.245	39.371
cycles elapsed	93,489,137,549	151,719,013,517
instructions retired	93,727,624,572	221,620,959,255
CPU Frequency	3.837 GHz	3.857 GHz
instructions per cycle	1.00	1.46

While the adaptive manager using the IPC metric does find a knob configuration that delivers a 1.46× higher IPC, the *number of instructions* required to complete the **matrix** workload has more than doubled! The core problem with the IPC metric is that it does not account for vector instructions, which complete more work per instruction than ordinary instructions. Within the performance monitoring unit of the CPU, which is the source of IPC measurement in HALO, each vector instruction counts as a single retired instruction. Some possible ways to correct the IPC metric include a scaling factor computed based on a static measure of the proportion of instructions in the library that are vector instructions.

8.3 Performance Comparison

In this section, we evaluate the effectiveness of HALO's ability to optimize programs in production across six benchmark programs. Each benchmark's trial begins by launching a fresh instance of the HALO server. That same server instance is then used for all tuning iterations within the trial, so that tuning progress is resumed between benchmark executions.

For the **desktop** and **workstation** machines, the server process is running on the same machine alongside the benchmark programs. To help avoid interference with the client programs on these machines, the server's parallel compilation thread-pool is limited to two workers. To lighten the load on the under-powered **mobile** machine, the benchmark

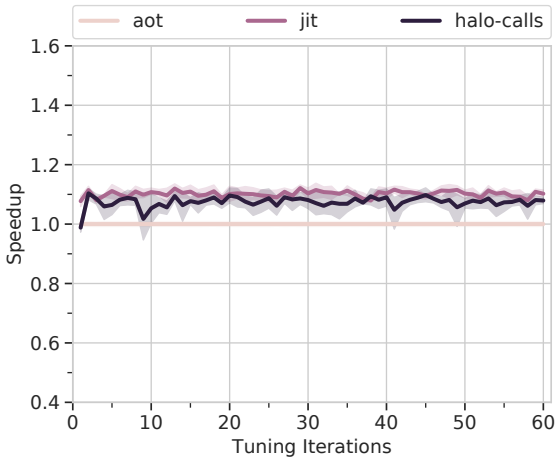
results for the **mobile** machine connect to a laptop computer running HALO server. Both the laptop and the **mobile** machine are on the same local-area network and use wired Ethernet for a stable and low-latency connection. We ran each benchmark for 60 tuning iterations and measured the execution time of each iteration to plot the performance over time. We evaluate HALO in conjunction with the adaptive tuning manager (Chapter 6) under only the call frequency metric, because empirically the IPC metric is no better than the call metric for the benchmark suite (Section 8.2). When compiling benchmarks for HALO, the `-O1` optimization level is applied to generate the initial executable, though the executable's LLVM IR is always captured prior to any optimizations.

For comparison with HALO, we measure the running time of each benchmark under the best ahead-of-time optimization methods available to ordinary programmers, called AOT, as a baseline. The AOT method does not use HALO and instead utilizes a two-stage compilation procedure, where the program is first compiled and run with instrumentation to gather profile data. That profiling data is used by the compiler when compiling the program for the second and final time, using the flags `-fprofile-instr-use=<file> -O3 -mcpu=native`.

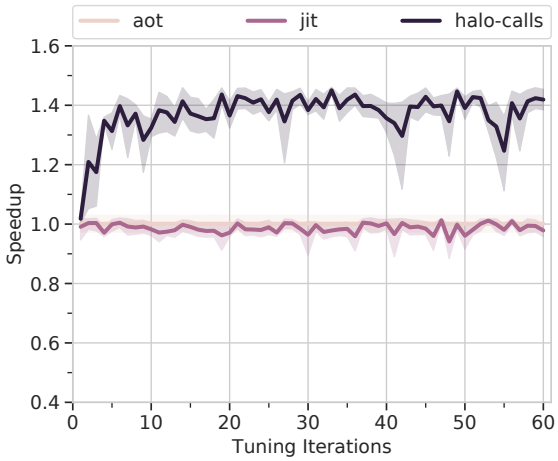
As a secondary point of comparison, we also measure the performance of a simple HALO tuning manager based on existing methods for online adaptive optimization. The JIT method utilizes HALO's JIT-once tuning manager (Section 5.3.3) that compiles the tuning section with options equivalent to `-O3 -mcpu=native`. This manager is designed to simulate a traditional single-stage JIT compiler that does not perform search.

Workstation Performance Figure 8.2 illustrates the performance on the **workstation** machine relative to AOT. The line represents the average speedup as the tuning progresses, with the bands surrounding the line indicating a 95% confidence interval for the trials.

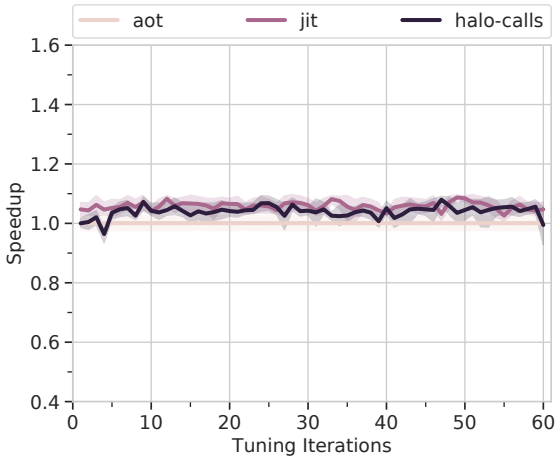
In all cases, HALO matches or exceeds the performance of AOT, which uses ahead-



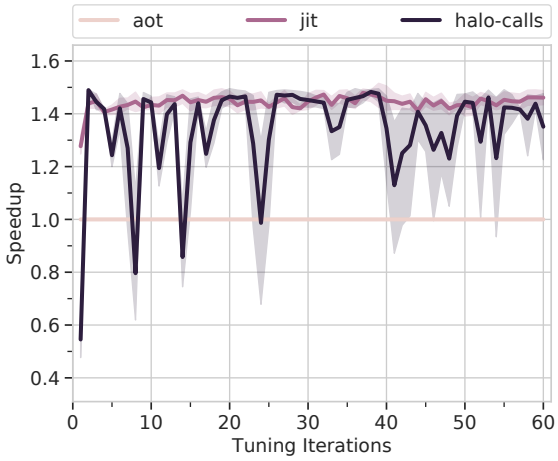
(a) lpbench



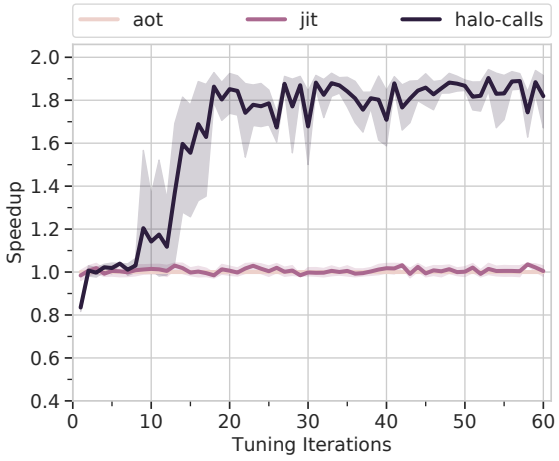
(b) matrix



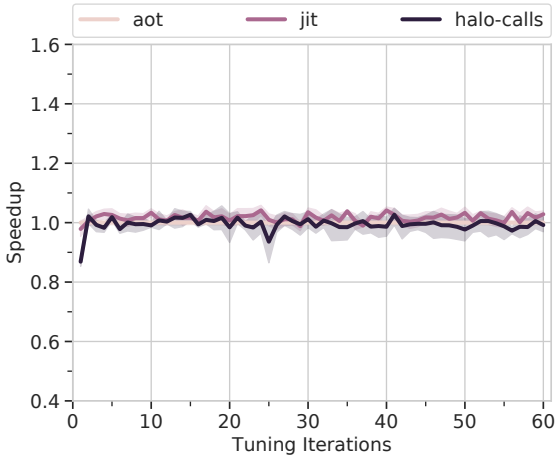
(c) n-body



(d) perlin



(e) spectralnorm



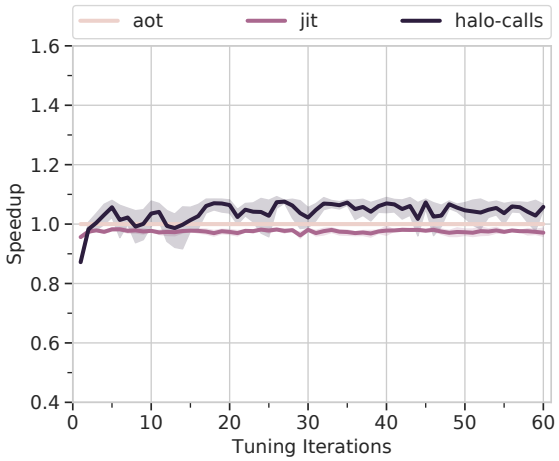
(f) sphereflake

Figure 8.2: Results for the **workstation** machine. Higher speedups are better.

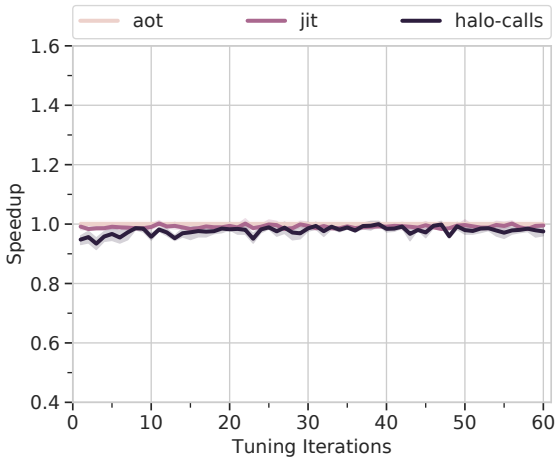
of-time profile-guided compilation, with the exception of **perlin**. For **perlin**, there are large spikes of performance regressions that last for one or two iterations at a time, with the spikes progressively dampening in severity over time. All spikes in performance for the adaptive tuning manager are caused by bakeoff activity. HALO is unable to find any knob configuration for **perlin** that is better than the default that the JIT-once manager uses, encountering many that are significantly worse and cause a noticeable regression.

More surprising, however, is that the adaptive tuning manager delivers significant performance improvements compared to the JIT-once manager. By the fifth tuning iteration with the adaptive tuning manager, the **matrix** benchmark sees a speedup of approximately 1.4× compared to the JIT-once strategy. Similarly, the adaptive manager achieves a large 1.8× speedup for **spectralnorm** by the 15th tuning iteration.

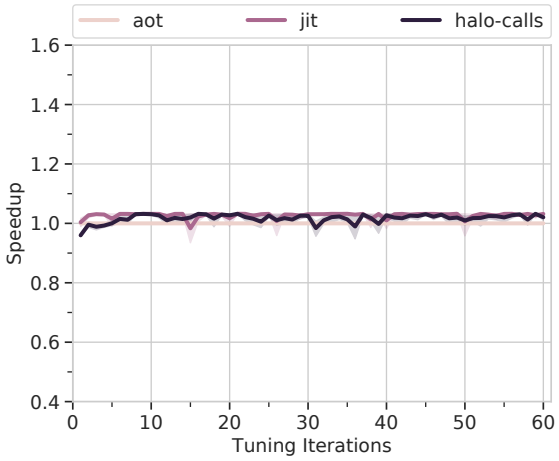
Desktop Performance As shown in Figure 8.3, performance on the **desktop** machine is similar to the **workstation** with some slight differences. HALO finds a small speedup on **lpbench** of approximately 1.05× relative to AOT, when there is no gain compared to JIT-once on the **workstation**. This is one example where the performance of the default optimizations used in JIT-once are sensitive to the machine, because HALO was able to find a better knob configuration for **lpbench** that is specific to the **desktop**. Next, while **matrix** saw a notable speedup on the **workstation**, the benchmark performs no better on the **desktop** than any other strategy. For **spectralnorm** the adaptive manager still delivers a huge speedup of approximately 2× over the JIT-once and AOT strategies for the **desktop**. Finally, for **sphereflake** the AOT strategy outpaces all other. The design of **sphereflake**'s ray tracer makes heavy use of recursive function calls, so it is possible that the overhead of the code redirection mechanism introduces overhead for the JIT-once manager.



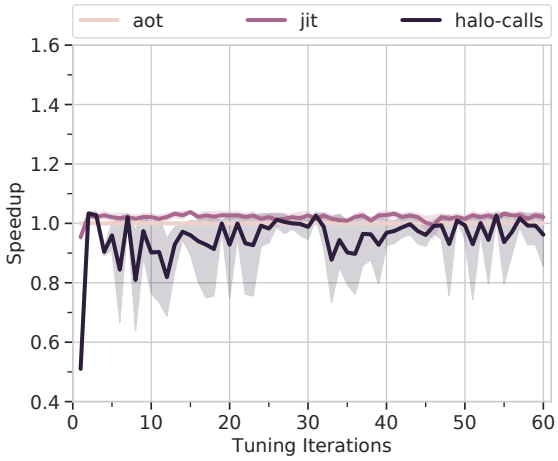
(a) lpbench



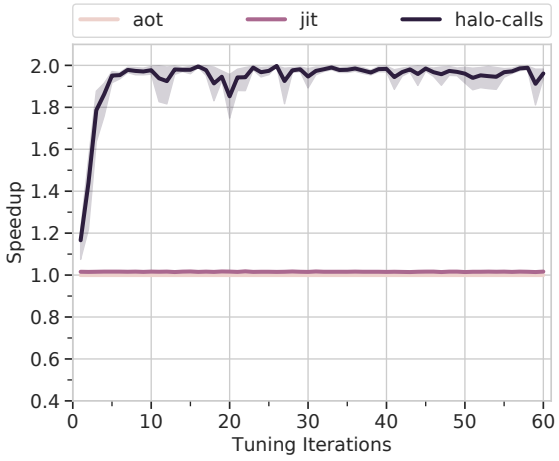
(b) matrix



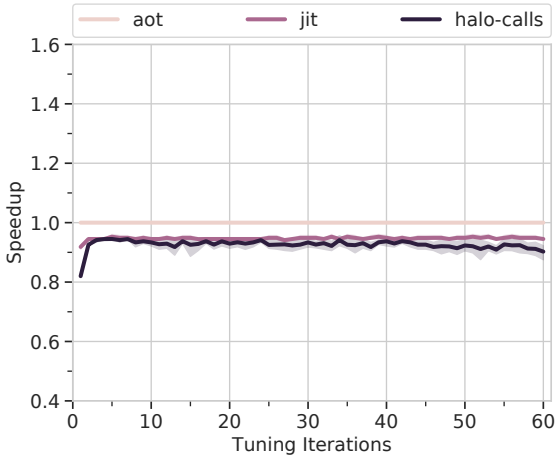
(c) n-body



(d) perlin

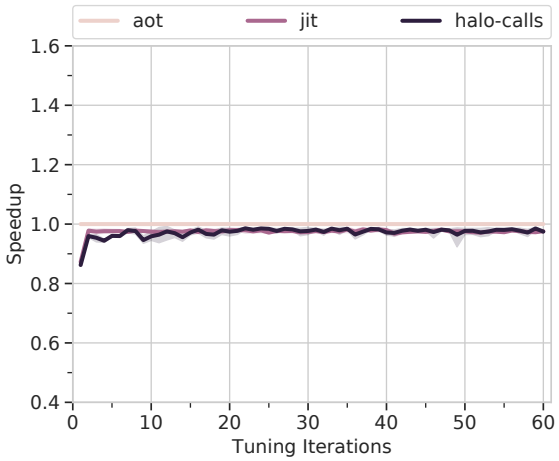


(e) spectralnorm

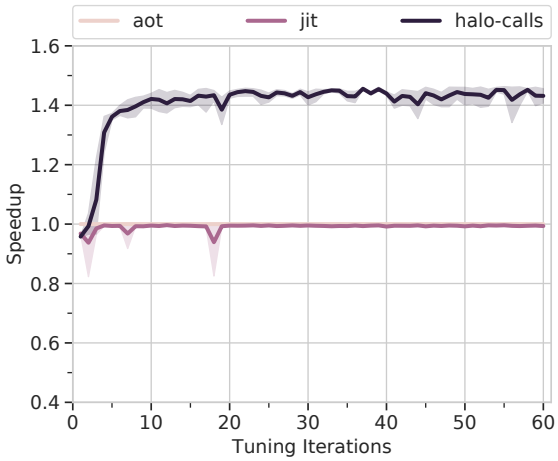


(f) sphereflake

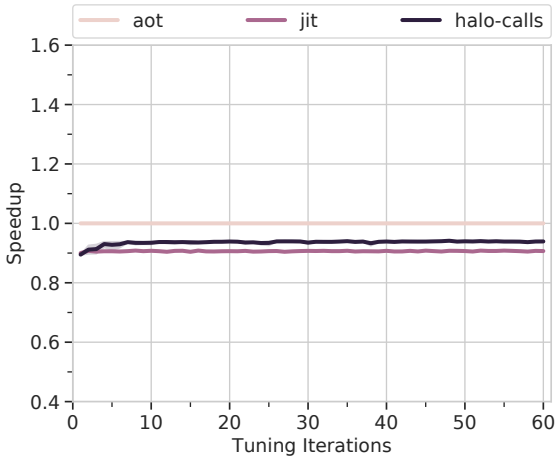
Figure 8.3: Results for the **desktop** machine. Higher speedups are better.



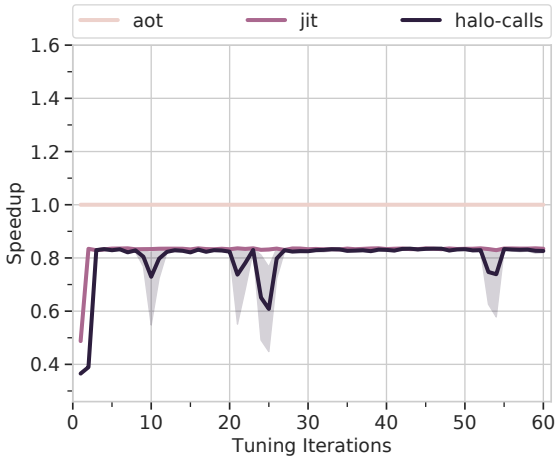
(a) lpbench



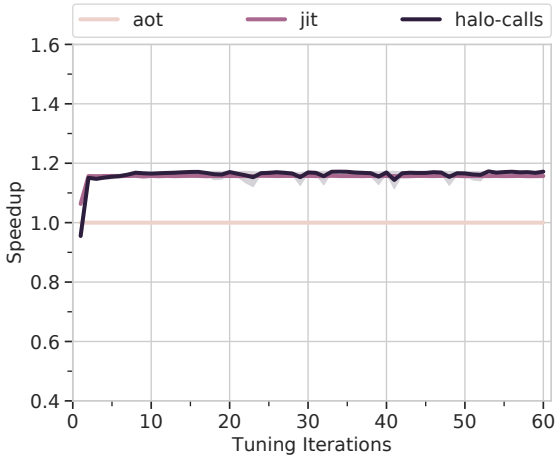
(b) matrix



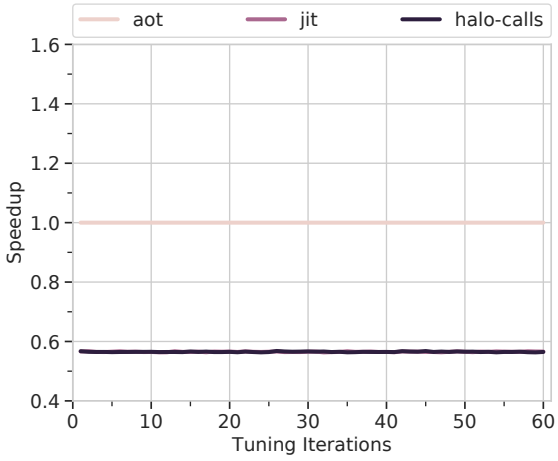
(c) n-body



(d) perlin



(e) spectralnorm



(f) sphereflake

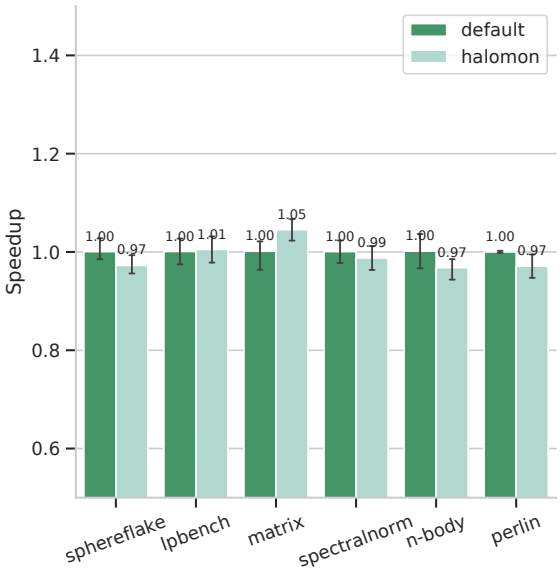
Figure 8.4: Results for the **mobile** machine. Higher speedups are better.

Mobile Performance The results for the **mobile** machine, shown in Figure 8.4, are unlike the previous machines. Specifically, **perlin** and **sphereflake** perform significantly worse than the best ahead-of-time compilation, regardless of the tuning manager. On a positive note, the adaptive manager again manages to stay on target and not regress much compared to the JIT-once manager. In terms of gains, the **mobile** machine sees a significant 1.4× speedup for the **matrix** benchmark compared to the other two strategies. Additionally, when compared to JIT-once, the adaptive manager is able to find a knob configuration that slightly outperforms the default configuration used by the JIT-once strategy. Finally, while **spectralnorm** saw significant improvement by the adaptive tuning manager on the other machines relative to the JIT-once strategy, for the **mobile** machine the performance sticks closely to JIT-once.

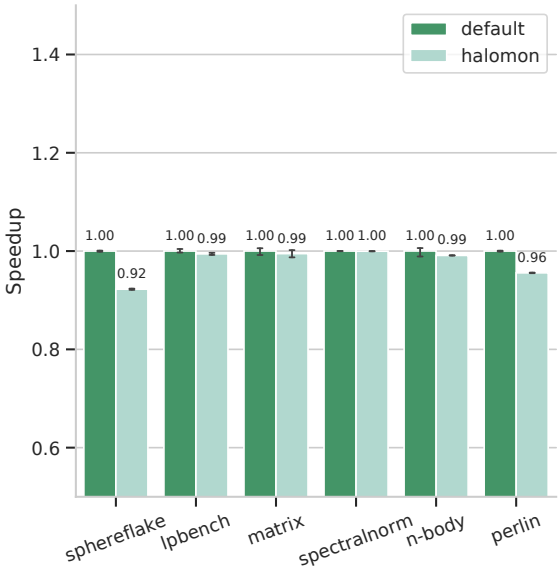
8.4 Offline Overhead

Because the HALO system is designed around a client-server model, in some situations a server may not be available to the client. For example, the **mobile** machine is too underpowered to run the HALO server process locally. Executables compiled with support for HALO execute normally if there is no server available; the monitor thread spawned by `halomon` stops after a fixed number of connection attempts. But, when compiling programs with support for HALO, the code in the executable has additional machinery added to support dynamic code patching (Section 5.2.3).

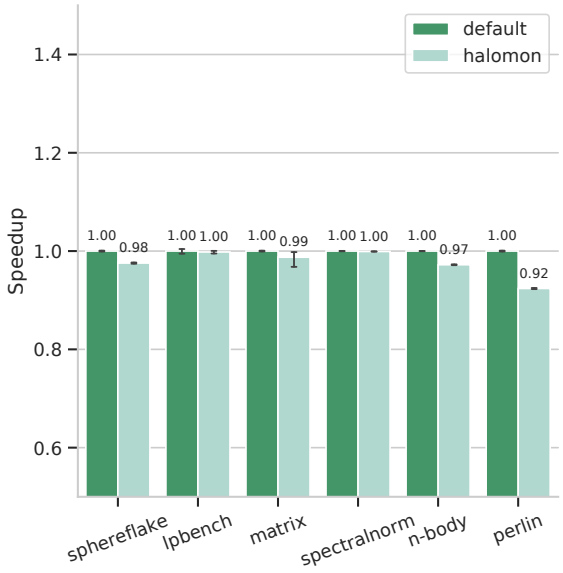
Figure 8.5 illustrates the overheads when no server is available for two unique CPU architectures when compiling all benchmarks at the `-O1` optimization level. The `-O1` level does not perform any function inlining, so Figure 8.5 represents the worst-case overhead of HALO's injection of no-op instructions to support dynamic code patching. The no-ops can increase the overhead of function calls, so inlining helps eliminate that overhead. There is little to no perceptible overhead relative to the default of not including HALO for



(a) **workstation**



(b) **desktop**



(c) **mobile**

Figure 8.5: Comparing the worst-case overhead of HALO-enabled executables when no server is available, relative to default compilation at `-O1`. Higher speedups are better.

the majority of benchmarks. For **sphereflake** and **perlin**, the overhead is notable because these benchmarks make heavy use of function calls in their workload. Thus, the heuristic used to determine whether a function should be patchable (Section 5.1) is not particularly effective for those two benchmarks in the worst case.

Chapter 9

Conclusions

The goal of this work is to investigate one core question from Chapter 4, "*can an automatic, online, search-based, adaptive recompilation system be effective in improving the performance of programs?*" The three most important aspects of this question involve usability, generality, and performance. In Chapters 5, 6, and 7, I describe HALO, an online, search-based adaptive recompilation system built to investigate this core question. Let us consider how this system fares with respect to these three important aspects.

Usability For the average user, HALO is an easy-to-use system for a few reasons. First, unlike PEAK (Section 3.1.7) and Active Harmony (Section 3.1.1), one does not have to perform program profiling or refactoring in advance. HALO will dynamically and automatically identify tuning sections (Section 5.3.2). Additionally, the HALO server does not need to know ahead-of-time about the program being tuned, because it automatically generates a set of knobs tailored to each tuning section. Second, while Active Harmony requires the user to manually profile and report the quality of each test library sent by the server, in HALO, all of the profiling and library experimentation is handled seamlessly for the user (Section 6.2). A user of HALO only needs to add `-fhalo` when compiling a program and ensure that an instance of HALO server is available when the program is

Table 9.1: Visual summary of HALO's performance relative to the JIT-once strategy. The symbol ✓ means *better*, ● means *about the same*, and ✗ means *worse*.

	workstation	desktop	mobile
lpbench	●	✓	●
matrix	✓	●	✓
n-body	●	●	✓
perlin	✗	✗	●
spectralnorm	✓	✓	●
sphereflake	●	●	●

launched.

Generality Under the lens of compatibility, HALO is very generic and can be ported to many systems and languages without much trouble. First, HALO is based upon the LLVM compiler infrastructure and the server tunes programs that are represented as LLVM IR; an industry-standard, program-agnostic representation (Chapter 5). Compilers that already use LLVM, such as the SWIFT compiler [Apple, 2020], can use HALO without much difficulty. Second, the HALO server's design can handle many heterogeneous clients, tuning different programs running on different architectures (Section 5.3). HALO has already been demonstrated to work across platforms: I hosted HALO server on an x86-64 machine to tune programs running on an ARM device (Chapter 8). While the prototype only works on Linux, the dependence is primarily due to HALO's use of *perf-events*. But, all major operating systems provide access to facilities equivalent to Linux's *perf-events* to enable sampling-based profiling.

Performance and Managing Overheads Based on the results of the performance evaluation (Chapter 8), a search-based online adaptive optimization system is feasible and can yield significant performance improvements. Table 9.1 helps summarize the results from Figures 8.2, 8.3, and 8.4, showing that HALO is about the same or significantly better

than the JIT-once strategy, for the most part. Keeping performance in line with the JIT-once strategy, while searching, is a difficult balancing act that required new techniques (Section 6). Performance improvements with HALO vary depending on the program and machine combination, lending further evidence that online, search-based adaptive recompilation can be profitable. But more work is needed to improve performance auditing, *i.e.*, quality metrics, without introducing additional overhead. One major takeaway is that search can be very hit-or-miss: most of the time a knob configuration better than the default was not found, but for configurations that were better, they were significantly so.

9.1 Future Work

In this section we consider future directions of this work and some practical concerns for a real-world version of a system such as HALO.

Quality Metrics For many programs, their performance is primarily constrained by memory accesses, so a quality metric based on cache miss-rates could be quite effective. While Linux's *perf-events* API can provide access to events in the CPU's cache hierarchy, I was not able to utilize this information in HALO. A refactoring of the infrastructure that receives sampling data from the client would have been required, because it was initially designed under the assumption that a performance metric can be updated after each sample arrives. But, to calculate a meaningful cache miss-rate, a larger window of samples would need to be collected. Going further, there is additional information in each *perf-events* sample, such as branch misses, that could be used to build a better quality metric using machine learning [Cavazos et al., 2007].

Population Experiments When there are multiple code-compatible HALO clients connected to a server, the tuning effort can become a collaborative process. In particular, randomized controlled trials among the population of similar clients can be used to determine the quality of a library. These trials on a subset of the population can help mitigate the negative impacts of a poor-performing library on the productivity of the entire group. Additionally, experiments on only a subset of clients allows for headroom to use a heavier-weight quality metric that is more accurate.

Separate Compilation The mechanism used by HALO for embedding the program as LLVM IR into object files is fairly generic, so adaptive optimization across libraries could be supported. The LLVM IR for each statically-linked library could be combined and included in the final executable during link-time. For dynamically-linked libraries, runtime look-ups by the HALO monitor could identify whether the library has LLVM IR to send to the server. It is unlikely that proprietary software would include the LLVM IR representation of the program in their object files, but it is possible to compile assembly code to LLVM IR [Korenčik, 2019].

Reinforcement Learning Adaptation in HALO happens through periodic but random experimentation among the available actions, which is typical for solutions to multi-armed bandit problems (Section 6). But, multi-armed bandits are a special case of the more general reinforcement learning problem [Sutton and Barto, 1998]. Reinforcement learning takes into consideration the state of some *environment* under which an action must be selected, where the environment may influence the value of each action. The multi-armed bandit problem effectively ignores its environment, so its ability to adapt to non-stationary problems is based on pure luck. Thus, evolving HALO's adaptive tuning manager to use reinforcement learning could make HALO more effective. For example, the program's *phase* is a profiling-based indicator that the program has changed the type

of work it is doing [Hind et al., 2003; Sherwood et al., 2002]. If the program's phase changes, *e.g.*, going from encoding to decoding, HALO's reinforcement learning problem with the current phase as part of its environment would notice that change and may be able to adapt faster. One of the downsides of reinforcement learning models is that they often require a large amount of training data, which is difficult to collect through online experimentation alone.

Security and Privacy An inherent problem with all computer system designs is the existence of hostile actors. The current prototype of HALO does not account for any threat models, but for a production version the security and privacy of HALO are paramount. One major concern is authentication, *i.e.*, ensuring that clients are only communicating with a HALO server provided by a trusted party. By design, HALO server has the ability to fully monitor and inject arbitrary code on all connected clients, so a high degree of trust is required if the server is hosted by a third party. Strong encryption between the server and clients is needed to maintain the privacy of clients, as their profiling data can reveal information about their pattern of program usage to an attacker. Additionally, the profile data from each user could be used to identify users across server reconnections, raising concerns for anonymity.

9.2 Vision

The HALO project is part of a broader vision for future online adaptive optimization technology where all programs are specially optimized for each user. Through the online search-based techniques described in this dissertation, compiler-writers can develop more speculative optimizations to be tested in production. Compared to ahead-of-time compilation, the risks of using speculative optimizations are lower because the optimization can be quickly undone at runtime based on profile data, as HALO does through bake-

offs. Additionally, when there are many active users of the same program, such as for video games or mobile apps, online autotuning can become a collaborative process. Ultimately, gaining mainstream adoption of new technologies is difficult, but I hope this work serves as a stepping-stone for mainstream use of autotuning.

Abbreviations

AOT Ahead-of-time

BBO Black-box Optimization

CCT Calling-context tree

HALO Wholly Adaptive LLVM Optimizer

JIT Just-in-time

LICM Loop-invariant Code Motion

MAB Multi-armed Bandit

OAO Online Adaptive Optimization

SLP Superword Level Parallelism

VM Virtual Machine

References

S. V. Adve, D. Burger, R. Eigenmann, A. Rawsthorne, M. D. Smith, C. H. Gebotys, M. T. Kandemir, D. J. Lilja, A. N. Choudbary, and J. Z. Fang and. Changing interaction of compiler and architecture. *Computer*, 30(12):51–58, December 1997. ISSN 0018-9162. doi: 10.1109/2.642815.

F. Agakov, E. Bonilla, J. Cavazos, B. Franke, G. Fursin, M. F. P. O'Boyle, J. Thomson, M. Toussaint, and C. K. I. Williams. Using Machine Learning to Focus Iterative Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 295–305, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 978-0-7695-2499-3. doi: 10.1109/CGO.2006.37. URL http://dx.doi.org/10.1109/CGO.2006.37.

Randy Allen and Ken Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987. ISSN 0164-0925. doi: 10.1145/29873.29875. URL https://doi.org/10.1145/29873.29875.

Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96, New York, NY, USA, 1997. ACM. ISBN 978-0-89791-907-4. doi: 10.1145/258915.258924. URL http://doi.acm.org/10.1145/258915.258924. event-place: Las Vegas, Nevada, USA.

Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A Language and Compiler for Algorithmic Choice. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 38–49, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542481. URL http://doi.acm.org/10.1145/1542476.1542481.

Jason Ansel, Maciej Pacula, Saman Amarasinghe, and Una-May O'Reilly. An Efficient Evolutionary Algorithm for Solving Bottom Up Problems. In *Annual Conference on Genetic and Evolutionary Computation*, Dublin, Ireland, July 2011. URL http://groups.csail.mit.edu/commit/papers/2011/ansel-gecco11-pbautotuner.pdf.

Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. SiblingRivalry: Online Autotuning Through Local Competitions. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems - CASES '12*, page 91, Tampere, Finland, 2012. ACM Press. ISBN 978-1-4503-1424-4. doi: 10.1145/2380403.2380425. URL http://dl.acm.org/citation.cfm?doid=2380403.2380425.

Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 303–316, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2809-8. doi: 10.1145/2628071.2628092. URL http://doi.acm.org/10.1145/2628071.2628092.

Apple. The Swift Programming Language, September 2020. URL https://github.com/apple/swift. original-date: 2015-10-23T21:15:07Z.

M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2):449–466, February 2005. ISSN 0018-9219. doi: 10.1109/JPROC.2004.840305.

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 47–65, New York, NY, USA, 2000. ACM. ISBN 978-1-58113-200-7. doi: 10.1145/353171.353175. URL http://doi.acm.org/10.1145/353171.353175. event-place: Minneapolis, Minnesota, USA.

Matthew Arnold, Michael Hind, and Barbara G. Ryder. Online Feedback-directed Optimization of Java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 111–129, New York, NY, USA, 2002. ACM. ISBN 978-1-58113-471-1. doi: 10.1145/582419.582432. URL http://doi.acm.org/10.1145/582419.582432. event-place: Seattle, Washington, USA.

Amir H. Ashouri, William Killian, John Cavazos, Gianluca Palermo, and Cristina Silvano. A Survey on Compiler Autotuning Using Machine Learning. *ACM Comput. Surv.*, 51 (5):96:1–96:42, September 2018. ISSN 0360-0300. doi: 10.1145/3197978. URL http://doi.acm.org/10.1145/3197978.

Charles Audet and Warren Hare. *Derivative-Free and Blackbox Optimization*. Springer Series in Operations Research and Financial Engineering. Springer International Publishing, 2017. ISBN 978-3-319-68912-8. doi: 10.1007/978-3-319-68913-5. URL https://www.springer.com/gp/book/9783319689128.

John Aycock. A Brief History of Just-in-time. *ACM Comput. Surv.*, 35(2):97–113, June 2003. ISSN 0360-0300. doi: 10.1145/857076.857077. URL http://doi.acm.org/10.1145/857076.857077.

Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A Transparent Dynamic Optimization System. page 12, 2000.

P. Balaprakash, R. B. Gramacy, and S. M. Wild. Active-learning-based surrogate models for empirical performance tuning. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–8, September 2013a. doi: 10.1109/CLUSTER.2013.6702683.

P. Balaprakash, J. Dongarra, T. Gamblin, M. Hall, J. K. Hollingsworth, B. Norris, and R. Vuduc. Autotuning in High-Performance Computing Applications. *Proceedings of the IEEE*, 106(11):2068–2083, November 2018. ISSN 0018-9219. doi: 10.1109/JPROC.2018.2841200.

Prasanna Balaprakash, Stefan M. Wild, and Paul D. Hovland. An Experimental Study of Global and Local Search Algorithms in Empirical Performance Tuning. In Michel Daydé, Osni Marques, and Kengo Nakajima, editors, *High Performance Computing for Computational Science - VECPAR 2012*, Lecture Notes in Computer Science, pages 261–269, Berlin, Heidelberg, 2013b. Springer. ISBN 978-3-642-38718-0. doi: 10.1007/978-3-642-38718-0_26.

Thomas Ball and James R. Larus. Branch Prediction for Free. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 300–313, New York, NY, USA, 1993. ACM. ISBN 978-0-89791-598-4. doi: 10.1145/155090.155119. URL http://doi.acm.org/10.1145/155090.155119. event-place: Albuquerque, New Mexico, USA.

Thomas Ball and James R. Larus. Optimally Profiling and Tracing Programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, July 1994. ISSN 0164-0925. doi: 10.1145/183432.183527. URL http://doi.acm.org/10.1145/183432.183527.

Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge Profiling Versus Path Profiling: The Showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 134–148, New York, NY, USA, 1998. ACM. ISBN 978-0-89791-979-1. doi: 10.1145/268946.268958. URL http://doi.acm.org/10.1145/268946.268958. event-place: San Diego, California, USA.

Protonu Basu, Mary Hall, Malik Khan, Suchit Maindola, Saurav Muralidharan, Shreyas Ramalingam, Axel Rivera, Manu Shantharam, and Anand Venkat. Towards making autotuning mainstream. *The International Journal of High Performance Computing Applications*, 27(4):379–393, November 2013. ISSN 1094-3420. doi: 10.1177/1094342013493644. URL https://doi.org/10.1177/1094342013493644.

Dean Michael Berris, Alistair Veitch, Nevin Heintze, Eric Anderson, and Ning Wang. XRay: A Function Call Tracing System. Technical report, 2016.

Dimitris Bertsimas and John Tsitsiklis. Simulated Annealing. *Statistical Science*, 8(1):10–15, 1993. doi: 10.1214/ss/1177011077. URL https://doi.org/10.1214/ss/1177011077.

Christian Blum and Andrea Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. *ACM Comput. Surv.*, 35(3):268–308, September 2003. ISSN 0360-0300. doi: 10.1145/937503.937505. URL http://doi.acm.org/10.1145/937503.937505.

Rastislav Bodík, Rajiv Gupta, and Mary Lou Soffa. Complete Removal of Redundant Expressions. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 1–14, New York, NY, USA, 1998. ACM. ISBN 978-0-89791-987-6. doi: 10.1145/277650.277653. URL http://doi.acm.org/10.1145/277650.277653. event-place: Montreal, Quebec, Canada.

Brad Calder, Peter Feller, and Alan Eustace. Value Profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 30, pages 259–269, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 978-0-8186-7977-3. URL http://dl.acm.org/citation.cfm?id=266800.266825. event-place: Research Triangle Park, North Carolina, USA.

J. Cavazos and M. F. P. O'Boyle. Automatic Tuning of Inlining Heuristics. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 14–14, November 2005. doi: 10.1109/SC.2005.14.

J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. P. O'Boyle, and O. Temam. Rapidly Selecting Good Compiler Optimizations using Performance Counters. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 185–197, March 2007. doi: 10.1109/CGO.2007.32.

C. Chambers, D. Ungar, and E. Lee. An Efficient Implementation of SELF a Dynamically-typed Object-oriented Language Based on Prototypes. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 49–70, New York, NY, USA, 1989. ACM. ISBN 978-0-89791-333-1. doi: 10.1145/74877.74884. URL http://doi.acm.org/10.1145/74877.74884. event-place: New Orleans, Louisiana, USA.

Craig Chambers and David Ungar. Making Pure Object-oriented Languages Practical. In *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '91, pages 1–15, New York, NY, USA, 1991. ACM. ISBN 978-0-201-55417-5. doi: 10.1145/117954.117955. URL http://doi.acm.org/10.1145/117954.117955. event-place: Phoenix, Arizona, USA.

Chun Chen, Jacqueline Chame, and Mary Hall. CHiLL: A framework for composing high-level loop transformations. Technical report, 2008.

D. Chen, N. Vachharajani, R. Hundt, X. Li, S. Eranian, W. Chen, and W. Zheng. Taming Hardware Event Samples for Precise and Versatile Feedback Directed Optimizations. *IEEE Transactions on Computers*, 62(2):376–389, February 2013. ISSN 0018-9340. doi: 10.1109/TC.2011.233.

Dehao Chen, David Xinliang Li, and Tipp Moseley. AutoFDO: Automatic Feedback-directed Optimization for Warehouse-scale Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 12–23, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3778-6. doi: 10.1145/2854038.2854044. URL http://doi.acm.org/10.1145/2854038.2854044. event-place: Barcelona, Spain.

Ray Chen. Active Harmony Example on GitHub, June 2019. URL https://github.com/ActiveHarmony/harmony/blob/master/example/code_generation/gemm.c. original-date: 2016-12-02T20:27:04Z.

Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, August 2016. Association for Computing Machinery. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2939785. URL https://doi.org/10.1145/2939672.2939785.

B. Childers, J. W. Davidson, and M. L. Soffa. Continuous compilation: a new approach to aggressive and adaptive code transformation. In *Proceedings International Parallel and Distributed Processing Symposium*, pages 10 pp.–, April 2003. doi: 10.1109/IPDPS.2003.1213375.

Trishul M. Chilimbi and Martin Hirzel. Dynamic Hot Data Stream Prefetching for General-purpose Programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 199–209, New York, NY, USA, 2002. ACM. ISBN 978-1-58113-463-6. doi: 10.1145/512529.512554. URL http://doi.acm.org/10.1145/512529.512554. event-place: Berlin, Germany.

I-Hsin Chung and Jeffrey K. Hollingsworth. Using Information from Prior Runs to Improve Automated Tuning Systems. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 30–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 978-0-7695-2153-4. doi: 10.1109/SC.2004.65. URL https://doi.org/10.1109/SC.2004.65.

Keith D. Cooper, Philip J. Schielke, and Devika Subramanian. Optimizing for Reduced Code Space Using Genetic Algorithms. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '99, pages 1–9, New York, NY, USA, 1999. ACM. ISBN 978-1-58113-136-9. doi: 10.1145/314403.314414. URL http://doi.acm.org/10.1145/314403.314414. event-place: Atlanta, Georgia, USA.

Keith D. Cooper, Devika Subramanian, and Linda Torczon. Adaptive Optimizing Compilers for the 21st Century. *The Journal of Supercomputing*, 23(1):7–22, August 2002. ISSN 1573-0484. doi: 10.1023/A:1015729001611. URL https://doi.org/10.1023/A:1015729001611.

Jonathan Corbet. KS2009: The future of perf events, October 2009. URL https://lwn.net/Articles/357481/.

Jonathan Corbet. Raw events and the perf ABI, May 2011. URL https://lwn.net/Articles/441209/.

Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, January 1977. Association for Computing Machinery. ISBN 978-1-4503-7350-0. doi: 10.1145/512950.512973. URL https://doi.org/10.1145/512950.512973.

Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active Harmony: Towards Automated Performance Tuning. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, pages 1–11, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press. ISBN 978-0-7695-1524-3. URL http://dl.acm.org/citation.cfm?id=762761.762771. event-place: Baltimore, Maryland.

Daniele Cono D'Elia and Camil Demetrescu. Flexible On-stack Replacement in LLVM. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 250–260, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3778-6. doi: 10.1145/2854038.2854061. URL http://doi.acm.org/10.1145/2854038.2854061. event-place: Barcelona, Spain.

Jay L. Devore and Kenneth N. Berk. *Modern Mathematical Statistics with Applications*. Springer Texts in Statistics. Springer-Verlag, New York, 2 edition, 2012. ISBN 978-1-4614-0390-6. doi: 10.1007/978-1-4614-0391-3. URL https://www.springer.com/gp/book/9781461403906.

Pedro C. Diniz, Martin C. Rinard, Pedro C. Diniz, and Martin C. Rinard. Dynamic feedback: an effective technique for adaptive computing. *ACM SIGPLAN Notices*, 32(5):71–84, May 1997. ISSN 0362-1340. doi: 10.1145/258915.258923. URL http://dl.acm.org/citation.cfm?id=258915.258923.

Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003. ISSN 1532-0634. doi: 10.1002/cpe.728. URL http://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.728. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.728.

Juan Durillo and Thomas Fahringer. From Single-to Multi-objective Auto-tuning of Programs: Advantages and Implications. *Sci. Program.*, 22(4):285–297, October 2014. ISSN 1058-9244. doi: 10.1155/2014/818579. URL https://doi.org/10.1155/2014/818579.

ECMA. ECMAScript: A general purpose, cross-platform programming language. Technical Report ECMA-262, June 1997. URL http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%201st%20edition,%20June%201997.pdf.

Kavon Farvardin. Halo Project, 2020. URL https://github.com/halo-project.

Wirth F. Ferger. The Nature and Use of the Harmonic Mean. *Journal of the American Statistical Association*, 26(173):36–40, March 1931. ISSN 0162-1459. doi: 10.1080/01621459.1931.10503148. URL https://amstat.tandfonline.com/doi/abs/10.1080/01621459.1931.10503148. Publisher: Taylor & Francis.

Álvaro Fialho, Raymond Ros, Marc Schoenauer, and Michèle Sebag. Comparison-Based Adaptive Strategy Selection with Bandits in Differential Evolution. In Robert Schaefer, Carlos Cotta, Joanna Kołodziej, and G unter Rudolph, editors, *Parallel Problem Solving from Nature, PPSN XI*, Lecture Notes in Computer Science, pages 194–203. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-15844-5.

S. J. Fink and Feng Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 241–252, March 2003. doi: 10.1109/CGO.2003.1191549.

Hal Finkel, David Poliakoff, and David F. Richards. ClangJIT: Enhancing C++ with Just-in-Time Compilation. *arXiv:1904.08555 [cs]*, April 2019. URL http://arxiv.org/abs/1904.08555. arXiv: 1904.08555.

Agner Fog et al. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. *Copenhagen University College of Engineering*, 93:110, 2011. URL https://www.agner.org/optimize/instruction_tables.pdf.

Jerome H. Friedman. Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics*, 29(5):1189–1232, 2001. ISSN 0090-5364. URL http://www.jstor.org/stable/2699986. Publisher: Institute of Mathematical Statistics.

Jerome H. Friedman. Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378, February 2002. ISSN 0167-9473. doi: 10.1016/S0167-9473(01)00065-2. URL http://www.sciencedirect.com/science/article/pii/S0167947301000652.

Grigori Fursin, Yuriy Kashnikov, Abdul Wahid Memon, Zbigniew Chamski, Olivier Temam, Mircea Namolaru, Elad Yom-Tov, Bilha Mendelson, Ayal Zaks, Eric Courtois, Francois Bodin, Phil Barnard, Elton Ashton, Edwin Bonilla, John Thomson, Christopher K. I. Williams, and Michael O'Boyle. Milepost GCC: Machine Learning Enabled Self-tuning Compiler. *International Journal of Parallel Programming*, 39(3):296–327, June 2011. doi: 10.1007/s10766-010-0161-2. URL https://doi.org/10.1007/s10766-010-0161-2.

Grigori Fursin, Renato Miceli, Anton Lokhmotov, Michael Gerndt, Marc Baboulin, D. Malony, Allen, Zbigniew Chamski, Diego Novillo, and Davide Del Vento. Collective Mind: Towards practical and collaborative auto-tuning. *Automatic Application Tuning for HPC Architectures*, 22(4):309–329, July 2014. doi: 10.3233/SPR-140396. URL http://hal.inria.fr/hal-01054763.

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. Trace-based just-in-time type specialization for dynamic languages. page 14, 2009.

Google. V8 JavaScript engine. URL `https://v8.dev/`.

Google. Protocol Buffers, 2020. URL `https://developers.google.com/protocol-buffers`.

James Gosling, Bill Joy, and Guy Steele. *The Java™ Language Specification*. Addison-Wesley, first edition, August 1996. ISBN 0-201-63451-1.

Isaac Gouy. The Computer Language Benchmarks Game, 2020. URL `https://benchmarksgame-team.pages.debian.net/benchmarksgame/`.

William G. Griswold, Richard Wolski, Scott B. Baden, Stephen J. Fink, and Scott R. Kohn. Programming language requirements for the next millennium. *ACM Computing Surveys (CSUR)*, 28(4es):194, December 1996. ISSN 0360-0300. doi: 10.1145/242224.242475. URL `http://dl.acm.org/citation.cfm?id=242224.242475`.

William Gropp, Rajeev Thakur, and Ewing Lusk. *Using MPI-2: Advanced Features of the Message Passing Interface*. MIT Press, Cambridge, MA, USA, 2nd edition, 1999. ISBN 978-0-262-57134-0.

Dayong Gu and Clark Verbrugge. Phase-based Adaptive Recompilation in a JVM. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 24–34, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-978-4. doi: 10.1145/1356058.1356062. URL `http://doi.acm.org/10.1145/1356058.1356062`. event-place: Boston, MA, USA.

R. Gupta, D. A. Benson, and J. Z. Fang. Path profile guided partial dead code elimination using predication. In *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 102–113, November 1997. doi: 10.1109/PACT.1997.644007.

Jürg Gutknecht. Oberon system 3: Vision of a future software technology. *Software Concepts and Tools*, 15(1):26–26, 1994.

Mary Hall, David Padua, and Keshav Pingali. Compiler Research: The Next 50 Years. *Commun. ACM*, 52(2):60–67, February 2009. ISSN 0001-0782. doi: 10.1145/1461928.1461946. URL `http://doi.acm.org/10.1145/1461928.1461946`.

Gilbert J. Hansen. Adaptive systems for the dynamic run-time optimization of programs. Technical report, Carnegie-Mellon University Department of Computer Science, 1974.

A. Hartono, B. Norris, and P. Sadayappan. Annotation-based empirical performance tuning using Orio. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–11, May 2009. doi: 10.1109/IPDPS.2009.5161004.

K. M. Hazelwood and T. M. Conte. A lightweight algorithm for dynamic if-conversion during dynamic optimization. In *Proceedings 2000 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.PR00622)*, pages 71–80, October 2000. doi: 10.1109/PACT.2000.888332.

Michael Hind, V. T. Rajan, and Peter F. Sweeney. Phase Shift Detection: A Problem Classification. Technical Report RC-22887, IBM Thomas J. Watson Research Lab, August 2003. URL https://domino.research.ibm.com/library/cyberdig.nsf/papers/E0A8A3AD9833F08485256D90006049F0/$File/RC22887.pdf.

Jeffrey K. Hollingsworth and Peter J. Keleher. Prediction and adaptation in Active Harmony. *Cluster Computing*, 2(3):195, November 1999. ISSN 1573-7543. doi: 10.1023/A:1019034926845. URL https://doi.org/10.1023/A:1019034926845.

Urs Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. PhD Thesis, Stanford University, Stanford, CA, USA, 1994.

Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In Pierre America, editor, *ECOOP'91 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 21–38. Springer Berlin Heidelberg, 1991. ISBN 978-3-540-47537-8.

Raymond J Hookway and Mark A Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9:3–12, 1997.

Kenneth Hoste, Andy Georges, and Lieven Eeckhout. Automated Just-in-time Compiler Tuning. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 62–72, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. doi: 10.1145/1772954.1772965. URL http://doi.acm.org/10.1145/1772954.1772965. event-place: Toronto, Ontario, Canada.

Jan Hubička. Profile driven optimisations in gcc. In *GCC Summit Proceedings*, pages 107–124. Citeseer, 2005.

Intel. Intel® 64 and IA-32 Architectures Optimization Reference Manual, May 2020. URL https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html. Library Catalog: software.intel.com.

P. J. Keleher, J. K. Hollingsworth, and D. Perkovic. Exposing application alternatives. In *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*, pages 384–392, June 1999. doi: 10.1109/ICDCS.1999.776540.

Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 978-1-55860-286-1.

T. Kistler and M. Franz. Continuous program optimization: Design and evaluation. *IEEE Transactions on Computers*, 50(6):549–566, June 2001. ISSN 0018-9340. doi: 10.1109/12.931893.

Thomas Kistler and Michael Franz. Continuous Program Optimization: A Case Study. *ACM Trans. Program. Lang. Syst.*, 25(4):500–548, July 2003. ISSN 0164-0925. doi: 10.1145/778559.778562. URL http://doi.acm.org/10.1145/778559.778562.

Thomas Peter Kistler. *Continuous Program Optimization*. PhD Thesis, University of California, Irvine, 1999.

Andi Kleen. An introduction to last branch records, March 2016. URL https://lwn.net/Articles/680985/.

P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. Combined Selection of Tile Sizes and Unroll Factors Using Iterative Compilation. *The Journal of Supercomputing*, 24(1):43–67, January 2003. ISSN 1573-0484. doi: 10.1023/A:1020989410030. URL https://doi.org/10.1023/A:1020989410030.

Donald E. Knuth. An Empirical Study of FORTRAN Programs. *Software: Practice and Experience*, 1(2):105–133, 1971. ISSN 1097-024X. doi: 10.1002/spe.4380010203. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380010203.

H. von Koch. Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire. *Arkiv för Matematik, Astronomi och Fysik*, 1:681–702, 1904. ISSN 0365-4133.

Sandeep Koranne. Boost C++ Libraries. In Sandeep Koranne, editor, *Handbook of Open Source Tools*, pages 127–143. Springer US, Boston, MA, 2011. ISBN 978-1-4419-7719-9. doi: 10.1007/978-1-4419-7719-9_6. URL https://doi.org/10.1007/978-1-4419-7719-9_6.

Lukáš Korenčik. Decompiling Binaries into LLVM IR Using McSema and Dyninst, 2019. URL https://is.muni.cz/th/pxe1j/.

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. Design of the Java HotSpot™ Client Compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):7:1–7:32, May 2008. ISSN 1544-3566. doi: 10.1145/1369396.1370017. URL http://doi.acm.org/10.1145/1369396.1370017.

Arvind Krishnaswamy and Rajiv Gupta. Profile guided selection of ARM and thumb instructions. *ACM SIGPLAN Notices*, 37(7):56–64, June 2002. ISSN 0362-1340. doi: 10.1145/566225.513840. URL https://doi.org/10.1145/566225.513840.

S. Kulkarni, J. Cavazos, C. Wimmer, and D. Simon. Automatic construction of inlining heuristics using machine learning. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–12, February 2013. doi: 10.1109/CGO.2013.6495004.

Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. *ACM SIGPLAN Notices*, 35(5):145–156, May 2000. ISSN 0362-1340. doi: 10.1145/358438.349320. URL https://doi.org/10.1145/358438.349320.

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. ISBN 978-0-7695-2102-2. URL http://dl.acm.org/citation.cfm?id=977395.977673. event-place: Palo Alto, California.

Jeremy Lau, Matthew Arnold, Michael Hind, and Brad Calder. Online Performance Auditing: Using Hot Optimizations Without Getting Burned. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 239–251, New York, NY, USA, 2006. ACM. ISBN 978-1-59593-320-1. doi: 10.1145/1133981.1134010. URL http://doi.acm.org/10.1145/1133981.1134010. event-place: Ottawa, Ontario, Canada.

Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When Prefetching Works, When It Doesn't, and Why. *ACM Trans. Archit. Code Optim.*, 9(1):2:1–2:29, March 2012. ISSN 1544-3566. doi: 10.1145/2133382.2133384. URL http://doi.acm.org/10.1145/2133382.2133384.

Niels Lohmann. nlohmann/json, September 2020. URL https://github.com/nlohmann/json. original-date: 2013-07-04T08:47:49Z.

Jiwei Lu, Howard Chen, Pen-Chung Yew, and Wei-Chung Hsu. Design and Implementation of a Lightweight Dynamic Optimization System. *Journal of Instruction-Level Parallelism*, page 24, April 2004.

John Mandel. *The Statistical Analysis of Experimental Data*. Dover Publications, 1964.

J. Maturana, A. Fialho, F. Saubion, M. Schoenauer, and M. Sebag. Extreme compass and Dynamic Multi-Armed Bandits for Adaptive Operator Selection. In *2009 IEEE Congress on Evolutionary Computation*, pages 365–372, May 2009. doi: 10.1109/CEC.2009.4982970.

S. McFarling. Reality-based optimization. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.*, pages 59–68, March 2003. doi: 10.1109/CGO.2003.1191533.

A. Morajko, P. Caymes-Scutari, T. Margalef, and E. Luque. MATE: Monitoring, Analysis and Tuning Environment for parallel/distributed applications. *Concurrency and Computation: Practice and Experience*, 19(11):1517–1531, 2007. ISSN 1532-0634. doi: 10.1002/cpe.1126. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1126.

Raphael Mosaner, David Leopoldseder, Manuel Rigger, Roland Schatz, and Hanspeter Mössenböck. Supporting On-Stack Replacement in Unstructured Languages by Loop Reconstruction and Extraction. *arXiv:1909.08815 [cs]*, September 2019. doi: 10.1145/3357390.3361030. URL http://arxiv.org/abs/1909.08815. arXiv: 1909.08815.

Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda, editors. *Software Automatic Tuning: From Concepts to State-of-the-Art Results*. Springer-Verlag, New York, 2010. ISBN 978-1-4419-6934-7. doi: 10.1007/978-1-4419-6935-4. URL http://www.springer.com/gp/book/9781441969347.

J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, January 1965. ISSN 0010-4620. doi: 10.1093/comjnl/7.4.308. URL https://academic.oup.com/comjnl/article/7/4/308/354237.

T. Nelson, A. Rivera, P. Balaprakash, M. Hall, P. D. Hovland, E. Jessup, and B. Norris. Generating Efficient Tensor Contractions for GPUs. In *2015 44th International Conference on Parallel Processing*, pages 969–978, September 2015. doi: 10.1109/ICPP.2015.106.

Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. *ACM SIGPLAN Notices*, 41(6):132–143, June 2006. ISSN 0362-1340. doi: 10.1145/1133255.1133997. URL http://doi.org/10.1145/1133255.1133997.

Dorit Nuzman, Revital Eres, Sergei Dyshel, Marcel Zalmanovici, and Jose Castanos. JIT Technology with C/C++: Feedback-directed Dynamic Recompilation for Statically Compiled Languages. *ACM Trans. Archit. Code Optim.*, 10(4):59:1–59:25, December 2013. ISSN 1544-3566. doi: 10.1145/2541228.2555315. URL http://doi.acm.org/10.1145/2541228.2555315.

OpenJS Foundation. Electron: Build cross-platform desktop apps with JavaScript, HTML, and CSS, 2020a. URL https://www.electronjs.org/.

OpenJS Foundation. Node.js: a JavaScript runtime built on Chrome's V8 JavaScript engine, 2020b. URL https://nodejs.org/.

Oxford English Dictionary. one-armed, adj. URL https://www.oed.com/view/Entry/131365.

David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986. ISSN 0001-0782. doi: 10.1145/7902.7904. URL https://doi.org/10.1145/7902.7904.

Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot™ Server Compiler. page 13, April 2001.

Zhelong Pan and Rudolf Eigenmann. Rating Compiler Optimizations for Automatic Performance Tuning. In *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, SC '04, pages 14–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 978-0-7695-2153-4. doi: 10.1109/SC.2004.47. URL https://doi.org/10.1109/SC.2004.47.

Zhelong Pan and Rudolf Eigenmann. Fast, Automatic, Procedure-level Performance Tuning. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 173–181, New York, NY, USA, 2006. ACM. ISBN 978-1-59593-264-8. doi: 10.1145/1152154.1152182. URL http://doi.acm.org/10.1145/1152154.1152182. event-place: Seattle, Washington, USA.

Zhelong Pan and Rudolf Eigenmann. PEAK—a fast and effective performance tuning system via compiler optimization orchestration. *ACM Transactions on Programming Languages and Systems*, 30(3):1–43, May 2008. ISSN 01640925. doi: 10.1145/1353445.1353451. URL http://portal.acm.org/citation.cfm?doid=1353445.1353451.

Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 2–14, February 2019. doi: 10.1109/CGO.2019.8661201.

Jason R. C. Patterson. Accurate Static Branch Prediction by Value Range Propagation. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 67–78, New York, NY, USA, 1995. ACM. ISBN 978-0-89791-697-4. doi: 10.1145/207110.207117. URL http://doi.acm.org/10.1145/207110.207117. event-place: La Jolla, California, USA.

Ken Perlin. An image synthesizer. *ACM SIGGRAPH Computer Graphics*, 19(3):287–296, July 1985. ISSN 0097-8930. doi: 10.1145/325165.325247. URL https://doi.org/10.1145/325165.325247.

Ken Perlin. Improving noise. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '02, pages 681–682, New York, NY, USA, July 2002. Association for Computing Machinery. ISBN 978-1-58113-521-3. doi: 10.1145/566570.566636. URL https://doi.org/10.1145/566570.566636.

Karl Pettis and Robert C. Hansen. Profile Guided Code Positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 16–27, New York, NY, USA, 1990. ACM. ISBN 978-0-89791-364-5. doi: 10.1145/93542.93550. URL http://doi.acm.org/10.1145/93542.93550. event-place: White Plains, New York, USA.

Simon Peyton Jones and Simon Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4-5):393–434, July

2002. ISSN 1469-7653, 0956-7968. doi: 10.1017/S0956796802004331. URL http://www.cambridge.org/core/journals/journal-of-functional-programming/article/secrets-of-the-glasgow-haskell-compiler-inliner/8DD9A82FF4189A0093B7672193246E22. Publisher: Cambridge University Press.

Filip Pizlo. Introducing the WebKit FTL JIT, May 2014. URL https://webkit.org/blog/3362/introducing-the-webkit-ftl-jit/.

Filip Pizlo. Introducing the B3 JIT Compiler, February 2016. URL https://webkit.org/blog/5852/introducing-the-b3-jit-compiler/.

Michael P. Plezbert and Ron K. Cytron. Does "Just in Time" = "Better Late Than Never"? In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 120–131, New York, NY, USA, 1997. ACM. ISBN 978-0-89791-853-4. doi: 10.1145/263699.263713. URL http://doi.acm.org/10.1145/263699.263713. event-place: Paris, France.

Louis-Noël Pouchet, Cédric Bastoul, Albert Cohen, and John Cavazos. Iterative Optimization in the Polyhedral Model: Part Ii, Multidimensional Time. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 90–100, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375594. URL http://doi.acm.org/10.1145/1375581.1375594. event-place: Tucson, AZ, USA.

John R. Rice. The Algorithm Selection Problem. In Morris Rubinoff and Marshall C. Yovits, editors, *Advances in Computers*, volume 15, pages 65–118. Elsevier, January 1976. doi: 10.1016/S0065-2458(08)60520-3. URL http://www.sciencedirect.com/science/article/pii/S0065245808605203.

Victor Rodriguez, Abraham Duenas, and Evgeny Stupachenko. Function multiversioning in GCC 6, June 2016. URL https://lwn.net/Articles/691932/.

R. H. Saavedra and Daeyeon Park. Improving the effectiveness of software prefetching with adaptive executions. In *Proceedings of the 1996 Conference on Parallel Architectures and Compilation Technique*, pages 68–78, October 1996. doi: 10.1109/PACT.1996.552556.

R. N. Sanchez, J. N. Amaral, D. Szafron, M. Pirvu, and M. Stoodley. Using machines to learn method-specific compilation strategies. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 257–266, April 2011. doi: 10.1109/CGO.2011.5764693.

Keith Seymour, Haihang You, and Jack Dongarra. A comparison of search heuristics for empirical code optimization. In *2008 IEEE International Conference on Cluster Computing*, pages 421–429, September 2008. doi: 10.1109/CLUSTR.2008.4663803. ISSN: 2168-9253.

Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International*

Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS X, pages 45–57, New York, NY, USA, 2002. ACM. ISBN 978-1-58113-574-9. doi: 10.1145/605397.605403. URL http://doi.acm.org/10.1145/605397.605403. event-place: San Jose, California.

Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 124–134, Szeged, Hungary, September 2011. Association for Computing Machinery. ISBN 978-1-4503-0443-6. doi: 10.1145/2025113.2025133. URL https://doi.org/10.1145/2025113.2025133.

Michael D. Smith. Overcoming the Challenges to Feedback-directed Optimization (Keynote Talk). In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, DYNAMO '00, pages 1–11, New York, NY, USA, 2000. ACM. ISBN 978-1-58113-241-0. doi: 10.1145/351397.351408. URL http://doi.acm.org/10.1145/351397.351408.

Michael D Smith and Glenn Holloway. An Introduction to Machine SUIF and Its Portable Libraries for Analysis and Optimization. *Division of Engineering and Applied Sciences, Harvard University*, 2002.

Kenneth O. Stanley and Risto Miikkulainen. Evolving Neural Networks through Augmenting Topologies. *Evolutionary Computation*, 10(2):99–127, June 2002. ISSN 1063-6560. doi: 10.1162/106365602320169811. URL https://www.mitpressjournals.org/doi/10.1162/106365602320169811.

M. Stephenson and S. Amarasinghe. Predicting unroll factors using supervised classification. In *International Symposium on Code Generation and Optimization*, pages 123–134, March 2005. doi: 10.1109/CGO.2005.29.

Mark Stephenson, Saman Amarasinghe, Martin Martin, and Una-May O'Reilly. Meta Optimization: Improving Compiler Heuristics with Machine Learning. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 77–90, New York, NY, USA, 2003. ACM. ISBN 978-1-58113-662-3. doi: 10.1145/781131.781141. URL http://doi.acm.org/10.1145/781131.781141. event-place: San Diego, California, USA.

Michal Strehovský. Code generation and execution strategies in CoreCLR, July 2019. URL https://github.com/dotnet/coreclr/blob/master/Documentation/design-docs/code-generation-strategies.md.

Student. The Probable Error of a Mean. *Biometrika*, 6(1):1–25, 1908. ISSN 0006-3444. doi: 10.2307/2331554. URL http://www.jstor.org/stable/2331554. Publisher: [Oxford University Press, Biometrika Trust].

Reiji Suda. A Bayesian Method for Online Code Selection : Toward Efficient and Robust Methods of Automatic Tuning. *Proc. iWAPT 2007*, pages 23–31, 2007. URL https://ci.nii.ac.jp/naid/10026764846/.

Reiji Suda. A Bayesian Method of Online Automatic Tuning. In Ken Naono, Keita Teranishi, John Cavazos, and Reiji Suda, editors, *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, pages 275–293. Springer, New York, NY, 2010. ISBN 978-1-4419-6935-4. doi: 10.1007/978-1-4419-6935-4_16. URL https://doi.org/10.1007/978-1-4419-6935-4_16.

Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998. ISBN 978-0-262-19398-6.

V. Tabatabaee, A. Tiwari, and J. K. Hollingsworth. Parallel Parameter Tuning for Applications with Performance Variability. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, pages 57–57, November 2005. doi: 10.1109/SC.2005.52.

George Teodoro and Alan Sussman. AARTS: Low Overhead Online Adaptive Autotuning. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 1–11, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0708-6. doi: 10.1145/2000417.2000418. URL http://doi.acm.org/10.1145/2000417.2000418. event-place: San Jose, California, USA.

A. Tiwari and J. K. Hollingsworth. Online Adaptive Code Generation and Tuning. In *2011 IEEE International Parallel Distributed Processing Symposium*, pages 879–892, May 2011. doi: 10.1109/IPDPS.2011.86.

A. Tiwari, C. Chen, J. Chame, M. Hall, and J. K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, May 2009a. doi: 10.1109/IPDPS.2009.5161054.

Ananta Tiwari, Vahid Tabatabaee, and Jeffrey K. Hollingsworth. Tuning parallel applications in parallel. *Parallel Computing*, 35(8):475–492, August 2009b. ISSN 0167-8191. doi: 10.1016/j.parco.2009.07.001. URL http://www.sciencedirect.com/science/article/pii/S0167819109000805.

Ananta Tiwari, Jeffrey K Hollingsworth, Chun Chen, Mary Hall, Chunhua Liao, Daniel J Quinlan, and Jacqueline Chame. Auto-tuning full applications: A case study. *The International Journal of High Performance Computing Applications*, 25(3):286–294, August 2011. ISSN 1094-3420. doi: 10.1177/1094342011414744. URL https://doi.org/10.1177/1094342011414744.

David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 227–242, New York, NY, USA, 1987. ACM. ISBN 978-0-89791-247-1. doi: 10.1145/38765.38828. URL http://doi.acm.org/10.1145/38765.38828. event-place: Orlando, Florida, USA.

Todd L. Veldhuizen. *Five compilation models for C++ templates*. 2000.

Todd L. Veldhuizen and Dennis Gannon. Active Libraries: Rethinking the roles of compilers and libraries. *arXiv:math/9810022*, SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing, October 1998. URL http://arxiv.org/abs/math/9810022. arXiv: math/9810022.

Michael J. Voss and Rudolf Eigemann. High-level Adaptive Program Optimization with ADAPT. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 93–102, New York, NY, USA, 2001. ACM. ISBN 1-58113-346-4. doi: 10.1145/379539.379583. URL http://doi.acm.org/10.1145/379539.379583.

Richard Vuduc, James W. Demmel, and Jeff A. Bilmes. Statistical Models for Empirical Search-Based Performance Tuning. *The International Journal of High Performance Computing Applications*, 18(1):65–94, February 2004. ISSN 1094-3420. doi: 10.1177/1094342004041293. URL https://doi.org/10.1177/1094342004041293.

Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate Static Estimators for Program Optimization. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 85–96, New York, NY, USA, 1994. ACM. ISBN 978-0-89791-662-2. doi: 10.1145/178243.178251. URL http://doi.acm.org/10.1145/178243.178251. event-place: Orlando, Florida, USA.

Z. Wang and M. O'Boyle. Machine Learning in Compiler Optimization. *Proceedings of the IEEE*, 106(11):1879–1901, November 2018. ISSN 0018-9219. doi: 10.1109/JPROC.2018.2817118.

Todd Waterman. *Adaptive compilation and inlining*. Thesis, 2006. URL https://scholarship.rice.edu/handle/1911/18991.

V. M. Weaver. Self-monitoring overhead of the Linux perf_ event performance counter interface. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 102–111, March 2015. doi: 10.1109/ISPASS.2015.7095789.

Vincent M Weaver. Advanced Hardware Profiling and Sampling (PEBS, IBS, etc.): Creating a New PAPI Sampling Interface. Technical Report UMAINE-VMW-TR-PEBS-IBS-SAMPLING-2016-08, University of Maine, August 2016.

John Whaley. Partial Method Compilation Using Dynamic Profile Information. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 166–179, New York, NY, USA, 2001. ACM. ISBN 978-1-58113-335-6. doi: 10.1145/504282.504295. URL http://doi.acm.org/10.1145/504282.504295. event-place: Tampa Bay, FL, USA.

R.C. Whaley and J.J. Dongarra. Automatically Tuned Linear Algebra Software. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 38–38, November 1998. doi: 10.1109/SC.1998.10004.

N. Wirth and J. Gutknecht. The Oberon System. *Software: Practice and Experience*, 19(9):857–893, 1989. ISSN 1097-024X. doi: 10.1002/spe.4380190905. URL `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.4380190905`.

Qing Yi, Keith Seymour, Haihang You, Richard Vuduc, and Dan Quinlan. POET: Parameterized Optimizations for Empirical Tuning. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, March 2007. doi: 10.1109/IPDPS.2007.370637. ISSN: 1530-2075.

Weifeng Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*, pages 87–98, September 2005. doi: 10.1109/PACT.2005.7.

M. Zhao, B. R. Childers, and M. L. Soffa. A model-based framework: an approach for profit-driven optimization. In *International Symposium on Code Generation and Optimization*, pages 317–327, March 2005. doi: 10.1109/CGO.2005.2.

Min Zhao, Bruce Childers, and Mary Lou Soffa. FPO: A framework for predicting the impact of optimizations. Technical Report TR–02–102, University of Pittsburgh, Department of Computer Science, 2002.

Mingzhou Zhou, Xipeng Shen, Yaoqing Gao, and Graham Yiu. Space-efficient Multi-versioning for Input-adaptive Feedback-driven Program Optimizations. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 763–776, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2585-1. doi: 10.1145/2660193.2660229. URL `http://doi.acm.org/10.1145/2660193.2660229`. event-place: Portland, Oregon, USA.