Halo: Wholly Adaptive LLVM Optimizer

A Research Proposal

Kavon Farvardin
University of Chicago
kavon@cs.uchicago.edu

October 14, 2019

Abstract

Low-level languages like C, C++, and RUST are the language-of-choice for performance-sensitive applications and have major implementations that are based on the LLVM compiler framework [Lattner and Adve, 2004]. The heuristics and trade-off decisions used to guide the static optimization of these programs during compilation can be automatically tuned during execution (online) in response to their actual execution environment. For performance-sensitive programs found in scientific computing, mobile devices, and internet services, an online adaptive optimization system that can find performance improvements missed by static optimization is highly desirable. The main objective of this research proposal is to explore what is possible in the space of online adaptive optimization for LLVM-based language implementations.

Contents

Acronyms

AOT	Ahead-of-time
BBO	Black-box Optimization
HALO	Wholly Adaptive LLVM Optimizer
JIT	Just-in-time
MAB	Multi-armed Bandit
OAO	Online Adaptive Optimization
RL	Reinforcement Learning
VM	Virtual Machine

1 Introduction

The primary reason for performing compiler optimizations before runtime (or *statically*) is that they are "free" in the sense that the analysis and transformations involved in applying the optimizations do not contribute any overhead to the running program. For optimizations that are inherently or provably beneficial, there is no reason to perform these at runtime (or *dynamically*). Unfortunately, not all optimizations will improve the performance of the generated code. Performing function inlining can open the door to additional optimization opportunities that enhance the code, but too much inlining can also degrade performance because it increases the size of the program, which leads to more instruction cache misses, *etc.* In other cases, program optimizations are too niche to try applying to the entire program, so they are artificially limited or disabled to speed up compilation time for the common case. The fundamental trade-off of using purely static optimizations instead of dynamic optimization is that less information is known about the program's runtime behavior to make smart decisions [Adve et al., 1997].

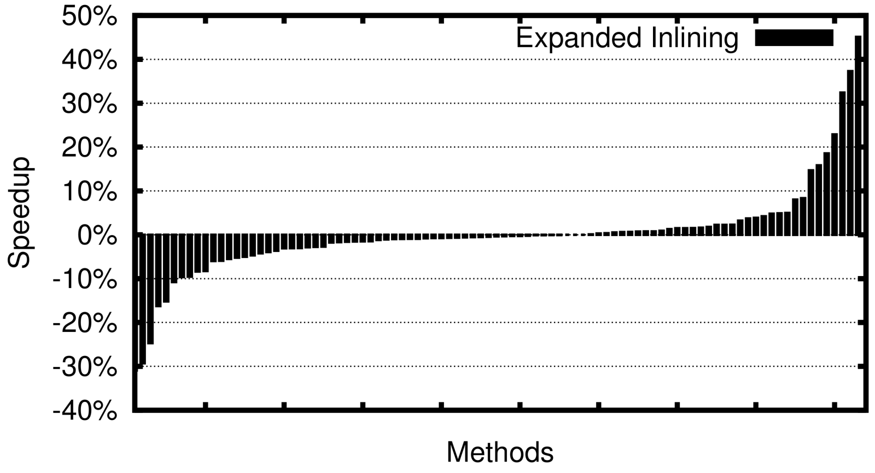

Figure 1: The affect of quadrupling the inlining threshold in the IBM J9 JVM compiler for the 101 hottest methods across 20 Java benchmark programs; from Lau et al. [2006].

Compiler optimizations rely on an *execution-cost model* to account for these kinds of performance trade-offs. An execution-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: the implicit model behind dead instruction elimination is that executing fewer instructions will reduce running time.

For optimizations where an actual cost model is needed to accurately estimate the trade-offs, the models are often parameterized to rely on abstract "threshold" values or information only known during execution of the program. The kinds of dynamic information include CPU cache pressure, branch target probabilities, or properties of a program variable's runtime value. In other cases, the cost model consists of an analytical system plus heuristics [George et al., 1994; Pouchet et al., 2011]. Compilers that run prior to program execution, or ahead-of-time (AOT), cannot access dynamic information directly.

In the absence of profile data to guide cost models, compilers must make conservative decisions such as disabling the optimization or using heuristics to guess default parameter values for the average case to avoid poor performance. 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 shows their results: raising the threshold helps significantly benefits some functions but significantly degrades others. In another example, Kennedy and Allen [2002, Chapter 5.11] discuss the problem of statically unknown loop bounds in cost modelling for loop vectorization: “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.” During program execution a *profiler* that gathers and analyzes dynamic information can be used in conjunction with a dynamic, or just-in-time (JIT), compiler to make better on-the-fly decisions about how to optimize the program [Aycock, 2003].

Automatic Tuning The broad concept of *adaptive optimization* is the modification of a program based on information gathered during the program’s execution in order to minimize (or maximize) some performance metric [Hansen, 1974].[1] Typically the objective 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 program settings or optimization decisions, known as a *configuration*. A type of adaptive optimization called *autotuning* is the process of identifying an optimal configuration within the space of possible choices. The term “autotuning” is a portmanteau for “automatic tuning,” which is meant in contrast with human-in-the-loop (or manual) performance tuning.

Autotuning is frequently applied to difficult problems where cost models are unknown or manual tuning is infeasible (*e.g.*, millions of viable configurations). One such difficult problem for compilers is determining the order of optimization passes to be applied to a program. The standard approach for AOT compilers is to manually develop a balanced, but fixed pass sequence that will be applied to all programs. An adaptive compiler such as Milepost GCC crafts a specialized optimization-pass sequence for each compiled program, yielding better performance over using a generic sequence [Fursin et al., 2011].

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 environment through the use of JIT compilation. The key advantage of performing optimizations online is that the optimizer can directly obtain runtime-only information instead of making conservative assumptions. Thus, for traditional AOT compilation approaches, the optimizations applied to each application are fundamentally limited because these compilers are forced to make assumptions about the program, leaving potential performance improvements untapped.

Programming languages such as JAVASCRIPT, JAVA, and SELF often have a high baseline-overhead relative to native execution due to interpretation and dynamically-determined behavior (*e.g.*, dynamic types). For these languages, online adaptive optimization (OAO) has been enormously successful in removing some or all of these overheads [Arnold et al., 2005; Gal et al., 2009; Hölzle, 1994; Kotzmann et al., 2008]. For example, the popular NODEJS platform and all major web browsers today rely on OAO to execute JAVASCRIPT efficiently. These systems use OAO in the form of virtual machines (VMs) that utilize selective JIT compilation [Plezbert and Cytron, 1997]. Based on profile information, they select the frequently-executed code regions for compilation and execution as native code instead of interpretation. Some systems further utilize value profiling [Calder et al., 1997] to infer properties of the program’s dynamic behavior, such as “the type of this function’s parameter is invariant,” to remove overheads due to the opaque treatment of the value as a more generic type that is required for correctness without profile information.

[1]Profile-guided optimization is a form of adaptive optimization, which itself is a specific form of feedback-directed optimization [Smith, 2000].

Objective Low-level languages like C, C++, and RUST are the language-of-choice for performance-sensitive applications and have major implementations that are based on the LLVM compiler framework [Lattner and Adve, 2004]. The heuristics and trade-off decisions used to guide the static optimization of these programs during compilation can be automatically tuned during execution (online) in response to their actual execution environment. For performance-sensitive programs found in scientific computing, mobile devices, and internet services, an online adaptive optimization system that can find performance improvements missed by static optimization is highly desirable. The main objective of this research proposal is to explore what is possible in the space of online adaptive optimization for LLVM-based language implementations.

2 Rationale

Many experts throughout the years have highlighted the need for adaptive optimization to fully access the hardware's peak performance. Griswold et al. [1996] believe 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] provide concrete examples where adaptive optimization 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. However, today dynamic code generation is 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."

2.1 Background

LLVM IR is a typed, static single-assignment representation of a program with constructs that mirror a high-level assembly language with unstructured control-flow [Lattner and Adve, 2004]. 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 are also applicable to LLVM IR.

Most C-like languages are optimized and compiled to native code ahead-of-time, thus their baseline overheads are much lower than interpreted or bytecode-compiled languages. However, the compiler optimizations applied to these C-like programs can be tuned to great benefit in an adaptive optimization system [Balaprakash et al., 2018]. For example, Orio is an autotuning system for C programs that tunes loop transformations such as unrolling, interchange, and tiling [Hartono et al., 2009]. When applied to computational kernels like sparse matrix computations and a sequence of linear algebra operations found in real scientific applications, Orio was able to match or significantly improve the performance relative to hand-tuned (*i.e.*, experts and the popular BLAS library) and compiler-generated code.

Empirical autotuning utilizes a trial-and-error search through the option space to find an optimal configuration. Domain specific methods have been used to efficiently guide this search process [Lee et al., 2019; Triantafyllis et al., 2003]. In many cases, empirical autotuners utilize black-box optimization techniques, where the program is treated as an opaque entity that accepts a configuration and outputs the configuration's quality [Almagor et al., 2004; Chabbi et al., 2011; Cooper et al., 1999, 2002; Fursin and Temam, 2010; Hoste et al., 2010; Kulkarni et al., 2009]. More formally, the goal of black-box optimization (BBO) is to find a configuration vector $\theta \in \Theta$ that has minimal cost $c = f(\theta)$, where f is the unknown cost function (or fitness function), c is a scalar value, and $|\Theta|$ is large.[2] The cost function is assumed to be expensive to probe, so exhaustive search methods that guarantee global optimality are not feasible.

A wide variety of heuristic search methods that efficiently explore the configuration space Θ to find locally-optimal configurations have been developed for BBO and applied to autotuning [Blum and Roli, 2003]. Two major types of meta-heuristics used for BBO 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]). All of these methods obtain convergence by assuming the cost function is deterministic and remains fixed throughout the search process, which is only the case for offline autotuning.

2.2 Challenges

There are a number of unique problems faced by *online* autotuning systems, relative to their offline counterparts, primarily because the tuned program is live and performing real work.

2.2.1 Profiling

Accurately comparing the quality of different versions of code is a major challenge, because 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, and/or perform other observable effects such as IO. In particular, the program may be applying the function to different inputs, which influence the amount of work the code must perform before it completes. Thus, the cost function used to determine the quality of a code version is no longer simple or deterministic.

2.2.2 Adaptation

The performance characteristics of code are often dependent on the users and/or data they are processing, and it is difficult to predict all such uses over the lifetime of a program once deployed [McFarling, 2003]. All offline systems suffer a limited ability to adapt due to the nature of their two-phase structure: test then deploy. The quality of the deployed configuration depends on the quality of the test phase that produced it. Constructing a small but representative suite of inputs is a major usability challenge, because the profile information gathered during the test phase should match the program's behavior after deployment. In

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

Wall [1991]'s evaluation of real and estimated profiles for predicting program behavior, they concluded that "even a real profile was never good as a perfect profile from the same run being measured. It was often quite close, however, and was usually at least half as good." An online adaptive optimization system can close this information gap by continually monitoring and reacting to changes for whichever environment the program encounters.

Most existing OAO systems for high-level languages adapt to change by "deoptimizing", or reverting, to either interpretation or a safer default version of the code. One case is when the program's environment has changed such that the assumptions made by the code no longer hold in the large. For example, the code may have been optimized based on assumptions inferred through profiling, such as the invariance of some property of a value, and once that assumption is violated, the code is deoptimized [Chambers and Ungar, 1989; Gal et al., 2009]. The other case is to limit the amount of dynamically generated code in the JIT compiler's cache. In Dynamo [Bala et al., 2000], when the program begins to encounter a lot of unoptimized code, this is taken as a signal that there may be dead code in the cache, so a full flush is issued. Thus, these systems re-adapt once reaching a steady state as a side-effect of their implementation, not as the main focus.

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 versions of code optimized for different thread locking heuristics (Section 5.2.2). That is, their system employs a *runtime policy* that takes into account the current environment when choosing a configuration. This is the same goal an online empirical autotuning system must have: the search for a high-quality policy, not only a configuration.

Chuang et al. [2006] estimated the effectiveness of a hypothetical system that is able to implement a dynamic policy for dispatching to differently optimized versions of code. They examined the performance of applications running on the Itanium 2 using the Intel compiler by varying the prefetching aggressiveness, loop scheduling, and load speculation settings. Their experiments found that for the SPEC CPU 2006 benchmarks, a 2–15.7% performance improvement might be achieved through the use of a runtime policy that dynamically selects code versions instead of a fixed code version.

2.2.3 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 explore the space of possible re-optimizations is paid-off by exploiting (or 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 scenario is primarily mitigated through dynamic profiling and a prediction model that determines whether the a code region is likely to remain hot in the future. A sampling-based hot-code profiler periodically interrupts a program thread to examine its state, such as the current value of the instruction pointer or the return addresses on the call stack. Instrumentation-based profilers inject bits of code at places such as function entry-points and loop-backedges to update counters whenever they are executed. Paleczny et al. [2001] describe their use of the latter approach in the influential HotSpot adaptive optimizer, which predicts that a code region is hot if the counters exceed thresholds that were determined heuristically.

A risk-benefit model is needed to mitigate the more challenging second scenario. To ensure profitability, it is important for an OAO system to focus on high-impact optimizations for hot code regions, while keeping the system's operating overhead low. 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. Furthermore, these new versions of code may have much worse performance than what it replaced, adding onto the costs.

3 Thesis

The specific goal of this research project is to investigate the following high-level question: *can an online adaptive compiler optimization system be effective in improving the performance of LLVM IR programs?* 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 adaptive optimization, as described by Basu et al. [2013]: usability, generality, and managing overheads. Then, Section 4 discusses the specific action plan of this research proposal and Section 5 provides an overview of related work.

Usability There are two groups of software developers for whom usability of an OAO matters: average developers and domain experts. 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 OAO is its ability to deliver performance improvement that matches or exceeds the expert's ability to perform manual optimization. On the other hand, an average application developer is interested in using OAO to meet their performance needs, but do not know precisely how to achieve them. An easy-to-use, portable OAO system is a bigger usability concern for average users than its performance gains. With respect to usability, our research aims to evaluate the effectiveness of an OAO system for average developers.

Generality Basu et al. 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 sooner 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 it is used like an offline optimizer, where the program is repeatedly performing the exact same work. Second, customization primarily benefits domain experts, but our focus is on usability by average developers. Thus, our project will evaluate effectiveness in the context of generality through the lens of compatibility, leaving customization to future work.

Performance and Managing Overheads Any adaptive optimization system will have some sort of overhead or additional cost, whether it be the time it takes the optimizer to converge on a locally-optimal solution, or the slowdown due to intermittent instrumentation and sampling for runtime 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 on which they would like to try adaptive optimization. Applying a traditional offline adaptive optimizer consists of creating and a running tuning phase prior to obtaining and deploying the resulting optimized version for real-world use. Because application-specific tuning is required during a phase prior to real-world use, the focus of most offline systems is to minimize the tuning time by reducing of 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 test run and the user's patience, times on the order of hours to days are not unusual (*e.g.*, Section 5.2.5). 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. Thus, overhead management becomes a more difficult task because of the need to balance the rate of experimentation with its costs, paying off negative balances by pausing to exploit the best seen. Because of this, the performance of an online optimization system is harder to evaluate, especially relative to an offline system. The ability to maintain optimality during unexpected workloads is the unique advantage that an online system offers over an offline one. Thus, the system's ability to adapt is a major factor when evaluating a system's performance.

4 Research Plan

To investigate this thesis, we propose building an online adaptive optimization system called HALO.[3] The goal is to try developing a simple and effective system that uses a novel combination of techniques. HALO optimizes programs represented in LLVM's intermediate representation (IR), which is a C-like static single-assignment IR that can be serialized as "bitcode" [Lattner and Adve, 2004]. LLVM is used by major compilers for C++, RUST, SWIFT, and other languages to target a variety of hardware architectures such as X86-64 and ARM. To evaluate HALO, we also extend the LLVM-based CLANG compiler for C/C++ with support for generating executables that utilize HALO for adaptive optimization. For the remainder of this section, we discuss current ideas about how to best execute the proposed investigation (Section 3) and overcome the associated challenges.

4.1 Overview

HALO utilizes a client-server model (Figure 2) to isolate the activity of the optimizer from the running application to prevent interference that degrades application performance. A *client* is a running executable that utilizes HALO for performance enhancement. When compiling a program for HALO usage, LLVM bitcode corresponding to the application's source code, plus some metadata, is embedded in the output executable file alongside the (potentially pre-optimized) machine code and linked with the monitoring library. 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]. Users of HALO 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.[4]

When a HALO executable is launched, the monitoring system is initialized and registers itself with an optimization server. The monitor is responsible for tasks such as profiling the process, dynamically linking

[3] An acronym for W<u>h</u>olly <u>A</u>daptive <u>L</u>LVM <u>O</u>ptimizer.

[4] We only support ELF object files for the sake of our project timeline; there is no technical reason for excluding Mach-O, *etc.*

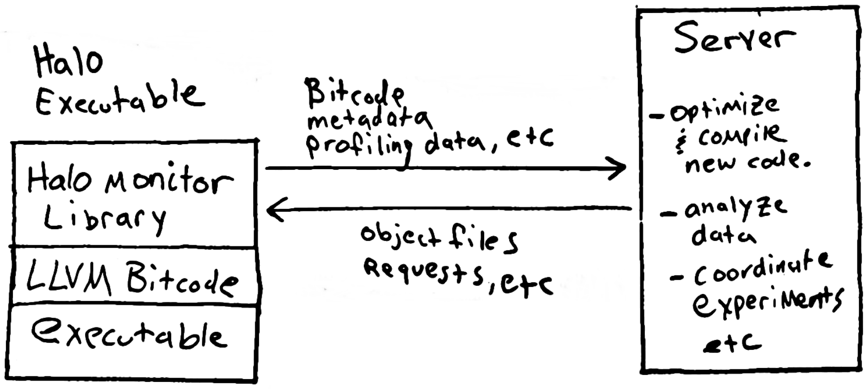

Figure 2: The client-server separation used by HALO. Communication occurs over a TCP/IP connection with an RPC-style protocol.

and managing object files sent by the server, patching in new code versions, *etc.* The HALO server will group clients together by compatibility (*e.g.*, bitcode and CPU architecture) for the purposes of collective optimization (Section 4.2.1). An optimization server drives the adaptive optimization system and can serve multiple groups of clients, with each group being optimized independently.

4.2 Addressing Challenges

In this section, we detail our tentative plans and ideas with regards to the challenges faced by online adaptive optimization as discussed in Section 2.2.

4.2.1 Profiling

Modern CPUs offer performance-monitoring units (PMUs) that can provide rich profile information (*e.g.*, calling context and branches taken) with low overhead, at the cost of accuracy and consistency. Nevertheless, Chen et al. [2013] has shown that it is feasible to use the noisy PMU data to create accurate block edge profiles for a program with a 1% average overhead. With block edge profiles in-hand, it is possible to estimate hot paths within a function [Ball et al., 1998] to better inform existing optimizations in LLVM. These hot paths can also be used to help identify whether two clients are executing similar workloads, or to narrow-down the search process by focusing only on options that would affect the hot functions.

There is some existing work by Pan and Eigenmann [2006] on utilizing function entry-counts to choose code that is worth tuning, *i.e.*, it is called frequently and executes long enough. While that work was only used for offline tuning with a single snapshot of data, we could adapt that work for HALO. Specifically, the use of profiling information to detect program phases [Hind et al., 2003; Nagpurkar et al., 2006] might be helpful in determining whether a function is worth tuning, since online tuning should focus on hot functions within a repeating phase to ensure that costs can be paid off in the long run.

Population Experiments When multiple HALO clients are code-compatible, the tuning effort can become a collaborative process where clients with similar behavior in-the-large, *i.e.*, similar profiles, are tuned together by the server. In particular, randomized controlled trials among the population of similar clients can be used to determine the quality of a code variant during the empirical search process. This helps mitigate the negative impacts of a poor-performing code variant on the productivity of the group of clients as a whole, because we only experiment on a subset.

Performance Comparison We are looking into using non-parametric statistics to help determine whether the running-time measurements of two code versions are comparable. Tabatabaee et al. [2005]; Tiwari et al. [2009b] examined the problem of controlling for transient performance variability, such as operating system or other application activity, when comparing the performance of differently optimized code when the workload given to them is the same. Based on their observations about the distribution of such transient variability, they prove that it is better to gather a fixed number of samples and take the minimum of those samples for comparison, instead of averaging. This is not a complete solution for HALO, because the workloads may change across multiple samples. Taking the environment context into account when analyzing the running-time measurements, such as the input arguments to the function or the calling context, is another approach used in prior work [Pan and Eigenmann, 2004]. Other signals of performance changes beyond the experimental function's execution time, such as the average instructions retired per cycle for some time interval, are also under consideration.

4.2.2 Adaptation and Finding Balance

This is the most novel part of the proposed research because there is little prior work in OAO that deeply examines the challenges faced by an empirical trial-and-error adaptive process.

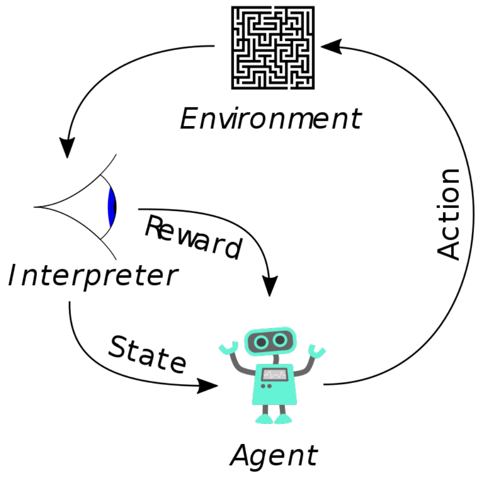

Figure 3: The interactive reinforcement learning cycle.

Reinforcement learning (RL) is a major area of machine learning that examines the situation of some agent trying to maximize rewards through interaction with its dynamic environment [Sutton and Barto, 1998]. The online learning task is modelled as a Markov decision process where an agent repeatedly chooses an *action* based on the current *state* of the environment, receiving a subsequent reward for each action that the agent tries to maximize in the long run (Figure 3). What is learned is the agent's behavioral *policy*, which maps state-action pairs to probabilities of taking the action in that state. One way to think about RL is that it can be used to solve dynamic programming problems where the reward of each action is unknown, because RL can learn optimal behavior through experience using the reward signal as its guide.

Currently the focus of this research is to apply reinforcement learning and/or their related multi-armed bandit (MAB) models to problems in OAO. There are a number of challenges in OAO that appear to be amenable through RL techniques, which varies based on the granularity of decision-making. Table 1 summarizes current ideas for applying RL or MABs in HALO, with each row indicating a potential model.

Purpose	**Environment**	**Actions**
Risk-benefit Management	Profiling statistics	Reoptimize, do nothing, or revert
Search Heuristics	Optimization Config.	Change unroll factor, speculation, *etc.*
Compiler Pass Ordering	Code features	Choose a set of passes
Loop Transform Ordering	Loop features	Choose a transformation
Function Versioning	Caller, arguments, phase, *etc.*	Select a code version

Table 1: Rough ideas for applying RL and/or MAB models to OAO problems in HALO. In all cases, the reward is some indicator of better performance or less overhead obtained from the action.

4.2.3 Autotuning

Autotuning in HALO will be specialized for tuning LLVM's optimization and compilation pipeline. Prior experience in Farvardin [2019] suggests that using supervised machine learning and/or Bayesian optimization may help prune the search space. Currently, one function (and its transitive callees) make up a *tuning section* and each section is tuned independently. Intuition suggests that this definition of a tuning section may include too much irrelevant code, so we plan to develop a selection policy based on Pan and Eigenmann [2008] and our unique implementation constraints. Table 2 summarizes the LLVM optimization parameters that are believed to yield the greatest benefit from autotuning.

Name	**Approx. Kinds of Options**
Pass Pipeline	Enable passes not on by default, disable risky passes.
Function Inlining	Threshold, individual `always-inline` marking.
Per-Loop Unrolling (+ Jam)	Enable/disable, count.
Per-Loop Vectorization and Interleaving	Enable/disable, width, count.
Per-Loop LICM Versioning	Enable/disable.
Per-Loop Distribution	Enable/disable.
Instruction Scheduling	Pre and Post RA scheduling enable/disable, *etc.*
Register Allocation	Interprocedural flag, algorithm choice.
Speculative Execution	`spec-exec-max-*` options in codegen.
Merge Splitting	Enable/disable per merge point.

Table 2: The types of risky but high-value optimizations under consideration for automatic tuning with HALO, for each tuning section.

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 over function-call replacement is that we can more quickly optimize a function's very long-running loops, instead of just the loop's callees, since this type of function is invoked infrequently but dominates the running time. 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 in the actual code optimizations being adapted instead of the granularity at which they can be done. Thus, HALO will not initially use OSR, meaning the smallest granularity of code that is optimized by the server is an LLVM IR function.

One under-explored idea in online autotuning is focusing the tuning process based on dynamic profile information, *i.e.*, if we know what paths are hot in the original code, tunable program structures such as a loop in that path will become the initial focus of the autotuner, since we know those parameters are currently active. This may result in fewer experiments because tuning a parameter that affects a cold code region should not affect the performance much in either direction. Ignoring this profile information could be problematic in an online autotuning engine because the search process or models may learn false notions that the parameter is not important or effectful, when in fact it might simply be unimportant *right now*, but can become useful in the future. The challenge in implementing this idea is that it is difficult to merge path profiles from differently optimized versions of a function, *i.e.*, the control-flow graph may be significantly different after some transformations.

4.3 Evaluation

Clang and LLVM's own support for offline profile-guided optimization and/or link-time optimization should serve as very good baselines of comparison for all benchmarks tested with HALO. To evaluate the effectiveness of adaptation, comparing the performance of programs over multiple successive runs similar to Arnold et al. [2002, Section 4] seems to be a fair measure. With the education discount SPEC CPU 2017 will cost $250, so it would be worthwhile to purchase a license and compare against SPEC and possibly other benchmarks.

To gauge the effectiveness of online adaptability with prior work, we can test a few small kernels with Active Harmony (Section 5.1.1) for comparison. It will be difficult to make a direct comparison, since Harmony alone is essentially a black-box optimization system made available through a C API. It would also be reasonable to compare against OpenTuner [Ansel et al., 2014], a well-regarded general-purpose offline autotuning framework. Like Harmony, it is a general-purpose black-box optimization system. Both OpenTuner and Active Harmony appear to be well-documented and are open source tools. Based on an examination of the Harmony source code, it appears to be hard to extend the system to new kernel due to the effort required and software dependencies.

One possible way to compare HALO with these existing systems would be to create an alternate search strategy in HALO that calls out to (or implements) the search strategies in Harmony and/or OpenTuner. Since BBO is not designed for search in a volatile or dynamic environment, this may seem like attacking a straw man. However if properly setup, such experiment may shed light on the usefulness of the approaches developed in this research over a naive approach. Otherwise, a comparison with offline tuning of LLVM compilation flags via OpenTuner would provide a reasonable baseline for comparison with state-of-the-art autotuning techniques.

4.3.1 An Early Experiment

Figure 4 illustrates the results of the most recent experiment of HALO's current overheads for one benchmark program. Based on sampling-based profiling, this experiment uses a simple heuristic, the most sampled function within a fixed time interval, to determine which function to optimize. If the function has

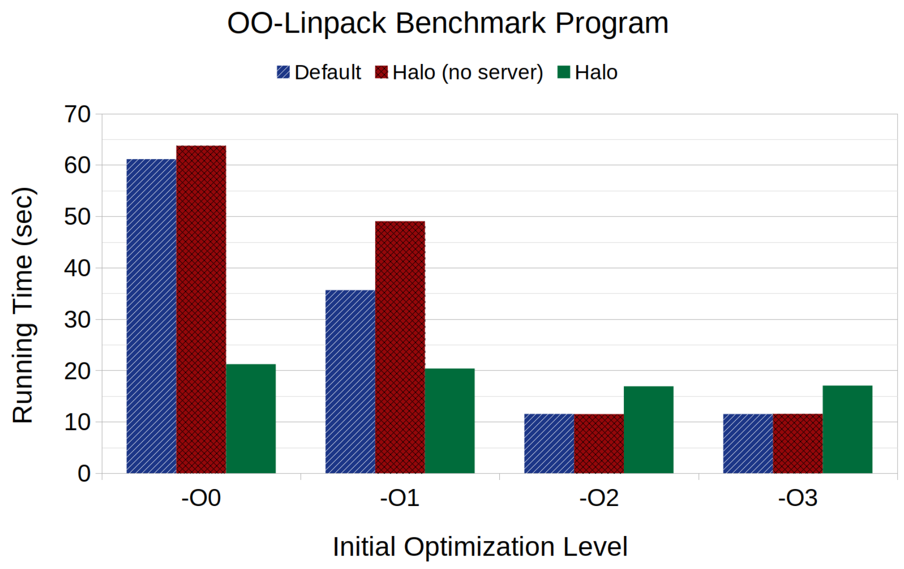

Figure 4: An early experiment to test overheads in HALO for a C++ program. HALO's current profile-guided remote JIT-compilation functionality can improve the performance of less optimized programs by recompiling their hottest functions at the default `-O3` level.

not already been optimized once by the server, it is recompiled at LLVM's default `O3` level and sent to the client to be linked and patched in dynamically. In this experiment the server and client are running on the same machine, and the HALO-enabled binary is run both with and without connecting to a server for optimization to see if there is inherent overhead.

If there is little to no overhead to compiling a program with HALO enabled, the *Default* and *Halo (no server)* bars should be level in all cases. This experiment and some debugging revealed that we must selectively choose which functions should be patchable by HALO when compiling the program statically, because the ability to dynamically patch a function adds some call overhead. Compiling with HALO currently makes *all* functions in the program patchable, no matter their size. Thus, the reason why HALO-compiled `O2` and `O3`, when run with no server, are no slower than default compilation is thanks to the inliner that is enabled at `O2` and higher in LLVM.

The performance improvements in this experiment also demonstrate that current profile-guided JIT-compilation functionality in HALO improves the performance of the C++ program if initially compiled at a lower optimization level. At higher levels, there appears to be an overhead or performance wall that should be solvable.

4.4 Facilities and Timeline

Currently I have access to a few HPC clusters at Argonne National Laboratory. If feasible I will try to utilize some of those resources for the evaluation of HALO in the context of many concurrent clients. As a backup, I also have exclusive access to machines other than my personal laptop: a smartphone-class ARM computer, a laptop with an Intel Haswell, a desktop with an Intel Ivybridge. There may also be some machines managed by John Reppy that I have used in the past that are available and compatible with HALO. In Table 3 we attempt to forecast the time needed to complete the proposed project.

Elapsed Time (weeks)	Tasks Completed
1	*perf_events* profile data collection
3	Serialization, TCP/IP, RPC
2	Integration with CLANG
1	Initial benchmarks, test suite, build system fixes
1	Parallel server-side compilation
2	Client-side dynamic linking, code patching, running-time data
3	Debugging, refactoring, research, and miscellaneous
13 weeks	*Total*
Estimated Time (weeks)	**Tasks Remaining**
4	Profile data analysis
4	Compiler optimization tuning
12	Implementation of adaptive optimization
8	Writing & evaluation
28 weeks	*Total up to May 5, 2020*
+20 weeks	*Spare time until Oct 2, 2020*

Table 3: Summary of progress thus far and forecasted tasks for this research as of Oct 29, 2020. May dissertation deadline is for Spring '20 quarter and the Oct deadline is the end of the first week of Autumn '20.

5 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. Due to the enormous volume of existing work that satisfies such a broad definition, it is not feasible for us to provide a survey of the area. Instead, the focus of this section is on works most similar or influential to the proposed research. No attempt is made to categorize the relevant works into subsections by an objective or rigid classification system; intuition is used instead.

5.1 By Similarity

This section contains a detailed overview of prior work that is subjectively believed to be the most similar to the proposed research.

5.1.1 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] Harmony was positioned as a global resource management system for the coming distributed-computing boom. 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 in Ţăpuş et al. [2002]. The major idea was that many libraries or algorithms implement the same functionality, but some are more appropriate for certain situations than others. 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 analyzing 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 were able to improve SMG2000's overall performance by 27% through the tuning of the main kernel, which saw a 2.37x speed-up.

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, with CHiLL used in their evaluation, to produce shared libraries with the new code variant. A system similar to Online Harmony called AARTS was proposed by Teodoro and Sussman [2011].

```
// MPI Initialization and Harmony API initialization are omitted.
if (masterClient) {
    hdef_t* hdef = ah_def_alloc(); // Create a new Harmony tuning search.
    ah_def_name(hdef, "gemm"); // code 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.
    // for gemm, "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 5: Client-side setup code for a parallelized online Active Harmony + CHiLL tuning session [Chen, 2019]. The tuned parameter names are known to the server to refer to loop tiling and unrolling for each loop nest.

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 5). When the clients begin 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 6). 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. Harmony server also does not manage the exploration versus exploitation trade-off of online tuning, except for when the server is not ready with a new configuration and the existing one is used. The work by Farvardin [2019] is both more adaptive and explicit about budgeting exploration costs.

```
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 6: 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].

5.1.2 Kistler's Optimizer

Kistler [1999]'s dissertation[5] describes a general, extensible architecture for online program optimization built on Oberon System 3 for the Macintosh [Gutknecht, 1994; Wirth and Gutknecht, 1989]. Kistler's system consists of five modular components: a code-generating loader, continuous profiler, manager, optimizer,

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

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, their system is essentially an ad-hoc solution to an adaptive optimization task in the spirit or reinforcement learning.

Kistler's evaluation of the system does not discuss total system performance gain or loss for a given benchmark. This is because 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 presumably because it was not effective. It would be surprising to find that they were effective, because the optimization components toggle themselves independently, without a central system to balance costs, *etc.*

5.1.3 Jikes RVM

The Jikes RVM[6] 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 a five online adaptive optimizations. First, profiling is used to decide whether to promote the code's optimization level in order to balance compilation costs with code quality. Second, a dynamic call graph is continually updated based on profiling data to identify hot call edges and perform adaptive inlining. Third, code within a method is laid out to prioritize locality based on profiling data. Fourth, loop unrolling is also 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. Finally, 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; Bodk et al., 1998; Chambers and Ungar, 1991]. A major a distinguishing factor is Jikes's use of profile-guided heuristics to determine what optimizations to dynamically apply to the code, in contrast to HALO's use of empirical search.

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. Instrumentation to generate profile data were added

[6]Originally called the Jalapeño Adaptive Optimization System.

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.

5.1.4 ADAPT

Voss and Eigemann [2001] describe ADAPT, which is an OAO system for FORTRAN 77. What sets ADAPT apart from prior systems, such as JVMs featuring JIT compilation like HotSpot and Jikes (Section 5.1.3), 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. HALO's structure is modelled after ADAPT, though 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 10 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. All of the AL scripts used to evaluate ADAPT experiment with at most 10 configurations using linear search, which is a trivial autotuning task. 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 improvements 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 – 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.

5.1.5 ADORE

Lu et al. [2004] describe a binary optimizer called ADORE (Adaptive Object code RE-optimization) for speculative online adaptive optimization. ADORE is a uses performance monitoring, sampling-based profiling, and phase 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.

ADORE is 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 Weifeng 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 exists. ADORE uses PMU sampling information for two optimizations within a code trace. The first and primary optimization determines which load instruction is the most costly and 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, where it will then reoptimize the code based on the new sampling data.

HALO is similar to ADORE in that: (1) they do not require modification of the original source program (2) use the same method of launching an additional thread for dynamic optimization before the program's `main` function is invoked, (3) use of the CPU's PMU for profiling and accesses them through Linux *perf_events*. What is different is that ADORE still applies heuristics to determine how and where a prefetching directive should be added, instead of using experimentation. No explicit 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. Thus, the exploration-exploitation trade-off is not explicitly considered in this work.

5.1.6 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.

5.1.7 PetaBricks

Ansel et al. [2009] developed PetaBricks, an implicitly parallel language for online autotuning of higher-level concepts such as parallelization techniques and algorithm choice. SiblingRivalry, PetaBricks's technique for online performance auditing, partitions resources and 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].

5.2 By Philosophy

This section contains a detailed overview of a number of works that are spiritually similar to the goals of the proposed research. Namely, they are either explicitly designed to be some sort of online adaptive optimization system, or similar enough to one.

5.2.1 Adaptive Fortran

The earliest known work in online adaptive optimization is Adaptive Fortran (AF) [Hansen, 1974], which aimed to address the same problems that motivate this proposal. Motivated by the findings of Knuth [1971] and others, profile information is used to determine when and where 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 in controlling the rate at which optimizations were applied to the program and sometimes kicks in too early in order to pay off the cost. For example, their heuristic for determining whether to further optimize code, predetermined execution frequency thresholds, worked well for some benchmarks but different thresholds were needed for others.

5.2.2 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. These versions implement different optimization policies, such as whether two critical sections in the 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 5.4) 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 each version 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.

5.2.3 CoCo

Childers et al. [2003] proposed the Continuous Compiler (CoCo) framework and performed prelimary experiments with a simulator to gauge the effectiveness of CoCo in adapting loop optimizations. The philosophy of CoCo is the same as HALO's, though CoCo fully relies on expert-defined analytical models to predict the impact of an optimization, which are detailed in Zhao et al. [2002]. Follow on work by Zhao et al. [2005] extended their analytical models to predict the effectiveness of scalar optimizations, though the work was implemented using the Machine SUIF compiler [Smith and Holloway, 2002].

5.2.4 MATE

Morajko et al. [2007] developed the Monitoring, Analysis and Tuning Environment (MATE) to perform dynamic automatic tuning for C/C++ applications, particularly for those running on a cluster. 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, *i.e.*, a set of rules that signal whether there is a performance bottleneck and/or optimal conditions have been found. Their evaluation of MATE first focused on the scenario of tuning a communication library (similar to MPI) within a space of 8 total options using exhaustive search, with a baseline running time of 12.2 minutes. The second scenario tuned one floating-point value within $(0, 1]$ that controls the division of work in a parallel application under a variable workload to demonstrate its ability to adapt over the default.

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%, though it is unclear whether the cause was a lucky ordering of the configuration space searched.

5.2.5 AOS

Hoste et al. [2010]'s Adaptive Optimization System (AOS) is built on top of the Jikes RVM (Section 5.1.3) and is similar to HALO in terms of combining automatic tuning and JIT compilation. The key difference is that AOS is an automatic tuner *for* a JIT compiler, not one 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 8 Pareto-optimal 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 1320 hours for the second stage, *i.e.*, a total of 1870 machine hours or nearly 78 days. Since this offline tuning can be parallelized, they were able to complete the tuning in only 75 hours (3 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.

5.2.6 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. A similar approach was used in Milepost GCC [Fursin et al., 2011] where machine learning was used to predict optimal application-specific plans, but this work focuses on function-specific plans.

Data to train the model is generated through iterated JIT recompilation 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 under-perform in steady-state performance when compared to Testarossa's default optimization plans. However, compilation-time and thus JVM warm-up time, were reduced with the plans produced by the model.

5.2.7 Testarossa for C/C++

Nuzman et al. [2013] extended the IBM Testarossa JVM to support C/C++, where profile-guided optimization is used to dynamically optimize programs. 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.

5.3 By Structure

This section provides an overview of works that are structurally similar to the infrastructure needed to perform online adaptive optimization as envisioned by this proposal.

Early JIT systems for virtual machines (VMs) were compile-only in the sense that VM code compiled to native code and then executed when control-flow first reaches it. 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 a widely used in language runtime systems like HotSpot [Gal et al., 2009; Hookway and Herdeg, 1997; Kotzmann et al., 2008; Paleczny et al., 2001]. CoreCLR currently uses a mixed AOT and JIT compilation execution strategy rather than interpretation [Strehovský, 2019]. For a more complete overview of early JIT compilation techniques, see Aycock [2003].

Lattner and Adve [2004] originally positioned LLVM as a compiler-based lifelong program optimization framework based on its generality, modularity, and common data format for programs (LLVM IR). Today, LLVM is widely used for runtime systems that employ JIT compilation such as Java, Julia, and Javascript [Bezanson et al., 2012; Geoffray et al., 2010; Pizlo, 2014; Reames, 2017]. Profile-guided compilation and link-time optimization are also supported by LLVM [Chen et al., 2016; Johnson et al., 2017]. Thus, LLVM makes for a good substrate for implementing HALO.

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.

5.4 Others

This section discusses works not sufficiently covered by prior sections but are important to discuss in the context of the research proposal.

5.4.1 Offline Tuning

The space of existing work in offline autotuning is huge and for a good overview of it see Ashouri et al. [2018]; Balaprakash et al. [2018] . Among the works that are simply *must-reads* are Milepost GCC [Fursin et al., 2011], Orio [Hartono et al., 2009], Active Harmony [Tiwari et al., 2009a], and OpenTuner [Ansel et al., 2014].

5.4.2 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 and/or environment dependence [Rice, 1976]. 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 some have worked on reducing this problem through profiling [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 progressively hotter code [Arnold et al., 2000]. These tiers help reduce overall compilation effort for once hot code and can improve performance of long-term hot code. 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 version 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].

5.4.3 Control Systems

Ribler et al. [2001] describes an online adaptive control system called Autopilot that is designed to sense and react to changes during program execution. Autopilot relies on knowing in advance when and how to interpret a signal as triggering an action. Thus, Autopilot and other control-based adaptive systems generally require some sort of cost model provided by the user. One exception to this is CALOREE, which can tune an abstract parameter that controls a known trade-off according sensor feedback while meeting user-specified deadlines [Mishra et al., 2018]. The difficulty in applying this type of work to more general autotuning is that the trade-offs are often not well understood and there are many parameters making up a huge space of options.

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.

L. Almagor, Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven W. Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. Finding Effective Compilation Sequences. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, pages 231–239, New York, NY, USA, 2004. ACM. ISBN 978-1-58113-806-1. doi: 10.1145/997163.997196. URL `http://doi.acm.org/10.1145/997163.997196`. event-place: Washington, DC, 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`.

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 Jalapeo 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`.

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, 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.

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`.

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`.

Jeff Bezanson, Stefan Karpinski, Viral B. Shah, and Alan Edelman. Julia: A Fast Dynamic Language for Technical Computing. *arXiv:1209.5145 [cs]*, September 2012. URL `http://arxiv.org/abs/1209.5145`. arXiv: 1209.5145.

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 Bodk, 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.

Milind M. Chabbi, John M. Mellor-Crummey, and Keith D. Cooper. Efficiently Exploring Compiler Optimization Sequences with Pairwise Pruning. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '11, pages 34–45, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0708-6. doi: 10.1145/2000417.2000421. URL `http://doi.acm.org/10.1145/2000417.2000421`. event-place: San Jose, California, USA.

C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-typed Object-oriented Programming Language. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 146–160, New York, NY, USA, 1989. ACM. ISBN 978-0-89791-306-5. doi: 10.1145/73141.74831. URL `http://doi.acm.org/10.1145/73141.74831`. event-place: Portland, Oregon, 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.

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.

Peng-fei Chuang, Howard Chen, Gerolf F Hoehner, Daniel M Lavery, and Wei-Chung Hsu. Dynamic Prole Driven Code Version Selection. *Performance Improvement*, page 9, 2006.

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.

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.

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.

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.

Kavon Farvardin. atJIT: autotuning C++, just-in-time!, May 2019. URL https://github.com/kavon/atJIT. original-date: 2018-06-15T15:38:31Z.

lvaro Fialho, Raymond Ros, Marc Schoenauer, and Michle 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.

Grigori Fursin and Olivier Temam. Collective Optimization: A Practical Collaborative Approach. *ACM Trans. Archit. Code Optim.*, 7(4):20:1–20:29, December 2010. ISSN 1544-3566. doi: 10.1145/1880043.1880047. URL http://doi.acm.org/10.1145/1880043.1880047.

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.

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.

Nicolas Geoffray, Gal Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. VMKit: A Substrate for Managed Runtime Environments. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 51–62, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-910-7. doi: 10.1145/1735997.1736006. URL `http://doi.acm.org/10.1145/1735997.1736006`. event-place: Pittsburgh, Pennsylvania, USA.

Lal George, Florent Guillame, and John H. Reppy. A portable and optimizing back end for the SML/NJ compiler. In Peter A. Fritzson, editor, *Compiler Construction*, Lecture Notes in Computer Science, pages 83–97. Springer Berlin Heidelberg, 1994. ISBN 978-3-540-48371-7.

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`.

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.

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.

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.

Teresa Johnson, Mehdi Amini, and Xinliang David Li. ThinLTO: Scalable and Incremental LTO. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 111–121, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5090-4931-8. URL `http://dl.acm.org/citation.cfm?id=3049832.3049845`. event-place: Austin, USA.

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.

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`.

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`.

Prasad A. Kulkarni, David B. Whalley, Gary S. Tyson, and Jack W. Davidson. Practical Exhaustive Optimization Phase Order Exploration and Evaluation. *ACM Trans. Archit. Code Optim.*, 6(1):1:1–1:36, April 2009. ISSN 1544-3566. doi: 10.1145/1509864.1509865. URL `http://doi.acm.org/10.1145/1509864.1509865`.

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.

Wen-Chuan Lee, Yingqi Liu, Peng Liu, Shiqing Ma, Hongjun Choi, Xiangyu Zhang, and Rajiv Gupta. White-box Program Tuning. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, CGO 2019, pages 122–135, Piscataway, NJ, USA, 2019. IEEE Press. ISBN 978-1-72811-436-1. URL `http://dl.acm.org/citation.cfm?id=3314872.3314889`. event-place: Washington, DC, USA.

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.

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.

Nikita Mishra, Connor Imes, John D. Lafferty, and Henry Hoffmann. CALOREE: Learning Control for Predictable Latency and Low Energy. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 184–198, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3173184. URL `http://doi.acm.org/10.1145/3173162.3173184`. event-place: Williamsburg, VA, USA.

A. Morajko, P. CaymesScutari, 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.

Priya Nagpurkar, Chandra Krintz, Michael Hind, Peter F. Sweeney, and V. T. Rajan. Online Phase Detection Algorithms. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 111–123, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 978-0-7695-2499-3. doi: 10.1109/CGO.2006.26. URL `http://dx.doi.org/10.1109/CGO.2006.26`.

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`.

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`.

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. PEAKa 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`.

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

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-Nol Pouchet, Uday Bondhugula, Cdric Bastoul, Albert Cohen, J. Ramanujam, P. Sadayappan, and Nicolas Vasilache. Loop Transformations: Convexity, Pruning and Optimization. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 549–562, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926449. URL `http://doi.acm.org/10.1145/1926385.1926449`. event-place: Austin, Texas, USA.

Philip Reames. Falcon: An Optimizing Java JIT, October 2017.

Randy L. Ribler, Huseyin Simitci, and Daniel A. Reed. The Autopilot performance-directed adaptive control system. *Future Generation Computer Systems*, 18(1):175–187, September 2001. ISSN 0167-739X. doi: 10.1016/S0167-739X(01)00051-6. URL `http://www.sciencedirect.com/science/article/pii/S0167739X01000516`.

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 multi-versioning in GCC6, June 2016. URL `https://lwn.net/Articles/691932/`.

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.

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`.

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`.

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 Auto-tuning. 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`.

Spyridon Triantafyllis, Manish Vachharajani, Neil Vachharajani, and David I. August. Compiler Optimization-space Exploration. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 204–215, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 978-0-7695-1913-5. URL `http://dl.acm.org/citation.cfm?id=776261.776284`. event-place: San Francisco, California, USA.

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

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`.

David W. Wall. Predicting Program Behavior Using Real or Estimated Profiles. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 59–70, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: 10.1145/113445.113451. URL `http://doi.acm.org/10.1145/113445.113451`.

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.

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`.

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 TR02102, 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.