WANG, TAO. Compiler-based Auto-tuning and Synchronization Validation for HPC Applications. (Under the direction of Dr. Frank Mueller and Dr. Guoliang Jin.)

Modern high performance computing (HPC) architectures feature multi-core processors with deep memory hierarchies, complex out-of-order instruction pipelines, powerful single instruction multiple data (SIMD) components, and heterogeneous accelerators. In practice, due to these architectural complexities, performance portability is a serious problem since programs tuned for one architecture usually achieve sub-optimal performance on another, which translates into excessive waste of energy and entails significant performance tuning efforts. Therefore, automatic performance tuning techniques are in high demand at DOE laboratories.

Existing tools are limited in different ways. On one side, traditional compiler-based auto-tuning approaches generate many functionally equivalent code variants and evaluate them on a tuning input to identify the best one. To generate a code variant, these approaches usually compile all program source files with the same compilation flags. Furthermore, after identifying the best code variants, the final performance evaluation is usually done on a small number of testing inputs. However, these experimental settings have limitations in two ways. First, different source modules may need specialized flags to achieve the best performance. Second, a program may have severe input sensitivity so that the tuned executable yields sub-optimal performance on many other inputs in practice.

Another problem is posed by multi-threaded HPC program correctness issues, such as deadlocks and data races, due to ad hoc synchronizations introduced by developers for performance purposes. There is also a need for novel tools to support bug detection and semantics validation for ad hoc synchronization constructs, e.g., ad hoc barriers. The state-of-art ad hoc synchronization analysis tools can only detect simple happen-before relationships between different program points and cannot detect complex synchronization constructs, such as ad hoc barriers, neither can they enumerate thread interleaving space to validate their dynamic semantics correctness.

This dissertation addresses these limitations of auto-tuning and ad hoc synchronization analysis technologies. We first propose a fine-grained compilation framework, FuncyTuner, to specialize the compilation for HPC program hot loops by utilizing per-loop profiling information to search the extremely large compilation flag space. Compared to the state-of-art, FuncyTuner improves performance of modern parallelized scientific programs by 4.5% to 10.7% (geometric mean) relative to the baseline. We then propose CodeSeer to evaluate different types of program sensitivity and build machine learning models to tackle the challenges presented by highly sensitive programs. Our experimental results show that all HPC programs expose certain type of basic input sensitivity and tuning inputs should be selected carefully. For those with high sensitivity, CodeSeer predictive models achieve 92% prediction accuracy while introducing less than 0.01 second online prediction overhead. Second, we contribute a framework, BARRIERFINDER, to automatically identify complex ad
hoc synchronizations and infer their enforced order relationships. BARRIERFINDER features various
techniques, including program slicing and bounded symbolic execution, to efficiently explore the
interleaving space of ad hoc synchronizations within multi-threaded programs for their traces.
BARRIERFINDER then uses these traces to characterize ad hoc synchronizations into different types,
such as barriers. Our evaluation shows that BARRIERFINDER is both effective and efficient in its
analysis. BARRIERFINDER also reliably detects deadlocks and atomicity violations for counter-based
barrier implementations.
Compiler-based Auto-tuning and Synchronization Validation for HPC Applications

by
Tao Wang

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina
2020

APPROVED BY:

Dr. James Tuck
Dr. Min Chi

Dr. Frank Mueller
Co-chair of Advisory Committee

Dr. Guoliang Jin
Co-chair of Advisory Committee
DEDICATION

To my parents.
ACKNOWLEDGEMENTS

I owe great thanks to my academic advisors, Drs. Frank Mueller, Guoliang Jin, James Tuck, Min Chi, my PhD program advisor, Dr. George Rouskas, my Office of International Services (OIS) advisor Ms. Mollie LoJacono, and many of my close friends. Without their generous supports, I would not finish this dissertation.

Dr. Frank Mueller is extremely patient and allows me to argue with him freely most of the time. Such a high degree of academic freedom is the essential soil that supports my growth to do independent research. His integrity, diligence, optimism, and persistence are the genuine wealth that I inherit from him. He demonstrates to me what a good scholar looks like, how to approach a new problem both incrementally and out-of-box, and always be eager to explore new fields. He encourages me when I seriously doubted whether I could finish this degree. Without him, I may have stopped my PhD study.

I am blessed to have a great co-advisor, Dr. Guoliang Jin. He was my hand-holder at the beginning of PhD study. I still remember the early time that we spent together polishing our papers and presentations over and again. These experiences are indispensable to guide me into the right track. Over the years, he built up his own research team, strived to maintain the high quality of teaching and research, which are good examples for me to learn. It was my great honor to have Dr. Tuck and Dr. Chi in my PhD committee. Dr. Tuck is an expert in compiler and CPU architecture research while Dr. Chi specializes in machine learning technological innovations. They spent a significant amount of time in my oral exam and this defense. The tightness of my defense schedule put a lot of stress on them and they had to get up so early in the morning to make my defense possible.

My PhD program advisor, Dr. George Rouskas, offered many great suggestions and represented the fairness of our PhD program. He always considered possible solutions for me when I had troubles to make progress. Like the guardian of the Constitution, there is no way of over-estimating his contributions. My OIS advisor, Ms. Mollie LoJacono provided significant supports when I needed them most. Her considerate thoughts helped me avoid potential huge immigration and employment troubles related to my PhD program. Ms. Carol Allen, Ms. Kathleen Luca, and Ms. Linda Honeycutt in NCSU CSC department also provided a lot of services with the highest standard of professional spirits. Their smiles were the sunshine that warmed my heart then and forever.

I would like to thank the following collaborators, close friends, mentors and professors. All the time that we spent together is a valuable part of my merry memory. I picked up certain valuable wisdom from each of you. Your supports helped me overcome challenges in life and work, inspired me to be a better person who trusts others and is always willing to help others. You will never be forgotten!

Prof. Alex Aiken (Stanford University),
Prof. Alan Edelman (Massachusetts Institute of Technology),
Dr. Amir Bahmani (Stanford University),
Dr. Amir H. Ashouri (University of Toronto),
Ms. Shiran Wang (Duke University),
Mr. Shudi Shao (NCSU),
Dr. Silvio Rizzi (Argonne National Laboratory),
Mr. Stan Kvasov (MediaTek),
Mr. Stephen Sechrist (College of William and Mary),
Dr. Todd Gamblin (Lawrence Livermore National Laboratory),
Mr. Tongjie Chen (NCSU),
Prof. Xiaobing Feng (Chinese Academy of Sciences),
Dr. Xiao Yu (NCSU),
Mr. Ya Cui (High School Affiliated to Nanjing Normal University),
Dr. Yung-Chia Lin (MediaTek),
Mr. Yunfeng Wu (Tsinghua University),
Prof. Yungang Bao (Chinese Academy of Sciences),
Mr. Vince Del Vecchio (MediaTek),
Dr. Vitali Morozov (Argonne National Laboratory),
JD. Weihang Zhao (College of William and Mary),
Prof. Weiwu Hu (Chinese Academy of Sciences),
Prof. Wenlin Li (Chinese Academy of Sciences),
Prof. Zexian Cao (Chinese Academy of Sciences),
Mr. Zhengyi Qiu (NCSU)
# TABLE OF CONTENTS

**LIST OF TABLES** ................................................................. viii

**LIST OF FIGURES** ............................................................... ix

**Chapter 1**  **Introduction** .................................................. 1
  1.1 Motivation ................................................................. 1
  1.2 Background ............................................................... 2
  1.3 Hypothesis ................................................................. 3
  1.4 Solutions and Contributions ............................................ 4

**Chapter 2**  **FuncyTuner: Auto-tuning Modern Scientific Applications With Per-loop Compilation** .................................................. 5
  2.1 Introduction ............................................................... 6
  2.2 Design of FuncyTuner .................................................... 8
    2.2.1 Compiler Flag Space Construction .................................. 8
    2.2.2 Space Search Algorithms ........................................... 9
  2.3 Experimental Design ................................................... 14
    2.3.1 Systems and Benchmarks .......................................... 14
    2.3.2 Compiler Flag Selection .......................................... 15
    2.3.3 Loop Outlining and Caliper Instrumentation ..................... 16
    2.3.4 CV Independence Assumption ...................................... 16
  2.4 Results and Analysis .................................................. 17
    2.4.1 Overall Performance Comparison .................................. 17
    2.4.2 Comparison to the State-of-the-art ................................ 19
    2.4.3 Impact Of Different Inputs ........................................ 21
    2.4.4 Deep Dive: Cloverleaf On Broadwell ................................ 23
  2.5 Related Work ............................................................. 25
  2.6 Conclusion ............................................................... 26

**Chapter 3**  **CodeSeer: Select Input-dependent Code Variants Via Machine Learning** ......................... 27
  3.1 Introduction ............................................................. 28
  3.2 Tackling Type-III Sensitivity .......................................... 31
    3.2.1 Problem Formulation ............................................... 31
    3.2.2 Representative Code Variants Selection ......................... 31
    3.2.3 Algorithm Design .................................................. 33
    3.2.4 Data Labeling ..................................................... 34
  3.3 Experimental Design ................................................... 34
    3.3.1 Benchmarks and Code Variants .................................. 34
    3.3.2 Prepare Input Data Sets .......................................... 36
    3.3.3 Build Machine Learning Models .................................. 36
  3.4 Results on Input Sensitivity ......................................... 37
    3.4.1 Type-I Sensitivity ................................................ 37
    3.4.2 Type-II Sensitivity ................................................ 38
    3.4.3 Type-III Sensitivity .............................................. 39
    3.4.4 Discussion on Type-III Sensitivity ............................... 39
  3.5 Model-based Tuning Results ............................................ 41
LIST OF TABLES

Table 2.1  List of benchmarks. LOC: lines of source code. .......................... 15
Table 2.2  Platform overview, runtime configurations, and benchmark inputs. .... 16
Table 2.3  Performance Statistics on Intel Broadwell ................................. 18
Table 2.4  Performance Statistics on Intel Sandy Bridge ............................. 18
Table 2.5  Performance Statistics on AMD Opteron ................................... 19
Table 2.6  Comparison of optimizations for 5 Cloverleaf kernels on Broadwell. S(Scalar): not vectorized; \{128,256\}: vectorized with \{128,256\}-bit SIMD; unroll\{2,3\}: unroll 2/3 times; IO: instruction reordering; IS: instruction selection; RS: register spilling. ......................................................... 25

Table 3.1  List of benchmarks (LOC: lines of source code) .......................... 35
Table 4.1  Overall results of BARRIERFINDER on SPLASH2 with slicing, intrusive tracing, and interleaving reduction ................................. 61
Table 4.2  Overall results of BARRIERFINDER on the synthetic application suite. Numbers in traces are sync region IDs. ............................... 64
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Combined Elimination does not improve performance significantly.</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Comparison of compilation models. a) A traditional compilation model with the same CV to compile all modules; b) FuncyTuner’s compilation model with different CVs to compile different modules.</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Per-program random search. $C V_k$ and $P_k$ with minimal runtime $T_k$ is its final result.</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Per-function random search. Step 3 is performed 1000 times.</td>
<td>10</td>
</tr>
<tr>
<td>2.5</td>
<td>FuncyTuner per-loop runtime collection framework is a part of both greedy combination and Caliper-guided random search.</td>
<td>12</td>
</tr>
<tr>
<td>2.6</td>
<td>CFR outperforms other methods for most cases: geometric mean of speedups relative to the O3 baseline 9.2%, 10.3%, and 9.4% on Opteron, Sandy Bridge and Broadwell, respectively.</td>
<td>17</td>
</tr>
<tr>
<td>2.7</td>
<td>FuncyTuner provides better performance than all variants of COBAYN(static, dynamic, hybrid), PGO, and OpenTuner.</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>CFR shows little performance sensitivity on small and large inputs with geometric mean of speedup relative to the O3 baseline 12.3% and 10.7% respectively.</td>
<td>22</td>
</tr>
<tr>
<td>2.9</td>
<td>For Cloverleaf on Broadwell, FuncyTuner CFR provides a stable performance benefit than all others while scaling from 100 to 800 time-steps.</td>
<td>22</td>
</tr>
<tr>
<td>2.10</td>
<td>Normalized speedups for top-5 loops of Cloverleaf on Intel Broadwell. Note: COBAYN (static), OpenTuner and Random generate the same code.</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Normalized speedups of 1000 Kripke code variants for a singular, identical tuning input: No performance improvement for any under O3.</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Normalized speedups of two Kripke code variants across 1,000 inputs show 1 type-II input sensitivity: The performance benefit of the same code variant is sensitive to different inputs; and 2 type-III input sensitivity: The best performing code variants depend on sets of many inputs.</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Workflow of CodeSeer</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Normalized speedup distributions for 1000 (top) and 100 (bottom) code variants on a single Kripke input indicating similar results: code variants in excess of the top 100 contribute little to performance (limited diversity).</td>
<td>35</td>
</tr>
<tr>
<td>3.5</td>
<td>All benchmarks expose type-I sensitivity: Performance benefits of a tuned program depends on the tuning input used.</td>
<td>37</td>
</tr>
<tr>
<td>3.6</td>
<td>All benchmarks expose type-II sensitivity: Performance benefits are different for inputs in $D_1$ with $C V_{opt}$; this achieves the best geometric mean speedup on $D_1$.</td>
<td>38</td>
</tr>
<tr>
<td>3.7</td>
<td>Cumulative number of inputs in $D_i$ with different normalized speedup gaps between $C V_{opt}$ and $C V_{opt}'$.</td>
<td>38</td>
</tr>
<tr>
<td>3.8</td>
<td>360.ilbdc, 351.bwaves, and Lulesh expose type-III sensitivity: CV1 and CV2 are the two code variants selected by CodeSeer’s greedy coverage plan algorithm.</td>
<td>40</td>
</tr>
</tbody>
</table>
Figure 3.9 Ratios (for Kripke only) to the per-input best speedup: higher is better. The per-input best speedup is the maximum speedup achieved by 100 code variants for inputs in $D_1$; top-$K$ (1, 2, 5, 10) indicates the best per-input speedup achieved by $K$ code variants identified by CodeSeer’s greedy algorithm.

Figure 3.10 Prediction precision of six different models for the unbalanced (a) and balanced (b) testing datasets: RandomForest (RF) achieves 92% and 90% geometric mean precision, respectively.

Figure 3.11 Speedups normalized of CodeSeer’s high-precision RF model for Kripke. Figures for 351.bwaves, 360.ilbdc and Lulesh are similar (omitted due to space).

Figure 3.12 Speedups normalized to O3 for Kripke $CV_1$, $CV_2$, and the RF model. The results for the balanced test set are similar (omitted).

Figure 3.13 CodeSeer model training overhead for all models on $D_2$: training RandomForest (RF) takes less than 4 minutes for all applications.

Figure 3.14 RandomForest prediction precision converges with 1000 training inputs: test set has 200 samples and is unbalanced.

Figure 4.1 An ad hoc synchronization example formed by $S_2$ and $S_3$. counter and flag are global variables. flag is initialized to true.

Figure 4.2 Extracted code from SPLASH2 LU.

Figure 4.3 The architecture of BARRIERFINDER: IR: interleaving reduction. IA: interleaving avoidance. ET: early termination.

Figure 4.4 Sync Boundary Detection Algorithm.

Figure 4.5 Interleaving enumerations and reduction: An execution state in BARRIERFINDER corresponds to an interleaving.

Figure 4.6 Interpreter-interpretee address space layout in BARRIERFINDER: An application is interpreted in interprettee space while the interpreter itself uses native execution.

Figure 4.7 The effect of interleaving avoidance (IA) and early loop termination (ET) under different iteration counts.
1.1 Motivation

Theoretical analysis and experimental evaluation have been the two traditional methodological pillars to support the development of sciences and engineering since ancient Greece time. The advent of modern electronic computers during World War II has enabled fast and large-scale computation of complex mathematical models. To date, scientific computation and simulations carried out by super-computers and high performance computing (HPC) programs are playing such an essential role that researchers consider it as the third pillar of modern scientific discoveries, advanced manufacturing and engineering. Many super-computers have been built and deployed in US Department of Energy (DOE) laboratories. Scientists run HPC programs to evaluate their ideas on a daily basis. However, these computers usually consume tens of Megawatts power. Such a high cost is overwhelming and needs to be reduced urgently. Moreover, since HPC programs are part of the daily research cycle, their performance is critical. These situations are made worse due to the increasing complexity in modern super-computers, such as complicated hardware features and software stacks. Without careful performance tuning effort, a HPC program would suffer from poor performance compared to its full performance potential. Thus, it would consume more energy, produce fewer scientific results, and slow down scientific exploration.

Program performance engineering is a multi-disciplinary study to address the above inefficiency in computing systems. It involves many topics in computational sciences, computer sciences, and computer engineering. The most prominent tools are profilers [Boe16; Wen18], compiler tool chains [Cor16; Inf15; LA04], debuggers, parallel programming environments [TMF93; DM98] and auto-tuners [Wan19; Hal10; Yi12]. Performance auto-tuners are tools that identify more performant
program configurations [Yu18a; Thi18], algorithmic code variants, compiler optimizations, runtime parameters, and system configurations, etc. The objective is to reduce the end-to-end runtime of a HPC program without compromising correctness. In this dissertation, two frameworks that we propose, i.e., FuncyTuner and CodeSeer, are both auto-tuners. Besides performance concerns, correctness of HPC programs is also critical. Development tools like bug detectors and semantics verifiers are desirable to ease the burden of human programmers. Our third framework, BARRIERFINDER, is a tool that can automatically identify ad hoc barriers and validate their semantics correctness.

1.2 Background

Extracting the peak performance of scientific applications becomes increasingly challenging on modern HPC architectures due to their diverse architecture features, such as deep memory hierarchies and heterogeneous accelerators. The de facto compilation model compiles all source files of a program with a single set of compiler flags, typically O2 or O3. However, it is well-known that O2/O3 may not generate the most performant executables [Che10]. To identify the most performant optimizations for a program, researchers have proposed iterative compilation [Kis00]. Given a program, iterative compilation first generates code variants by compiling the source code with different compiler flags, and then evaluates their performance either by execution [Che10] or prediction [Cav07; Pop16].

Iterative compilation techniques either treat a program as one compilation unit [Ans14; Che10; Ash16; Cav07; Nob16; JK13], or in a fine-grained fashion [PE08; Pop16; Hal10; Yi12]. The former ones lose the opportunity to perform specialized optimization for different program modules while the latter aggressively assume that different modules can be compiled independently without any conflicting interactions. We designed and implemented a per-loop auto-tuning framework, FuncyTuner, which utilizes per-loop profiling information to guide fine-grained random search. Our evaluation on a suite of modern scientific programs shows that FuncyTuner outperforms the state-of-art approaches.

Prior auto-tuning techniques, including ours [Wan19; Thi18], are constrained to tuning for a single inputs, which produces a single optimized executable evaluated on a small set of test inputs to report any performance improvements. As a result, their effectiveness may be limited by the tuning input and any performance tends to be specialized to this input instead of generalizing to a larger set of inputs in production. If an improper input is selected as the tuning input, these optimization techniques may fail to result in executables realizing any performance improvements whatsoever. Furthermore, if different inputs depend on different code variants to achieve peak performance, prior tuning techniques would leave some performance potential on table. As shown in [Wan19], profile-guided optimization (PGO) of Intel compilers rarely improves the performance of a set of modern HPC programs, and there is little practical guidance on how to select characteristic tuning data sets for PGO. We proposed an auto-tuning framework, CodeSeer, to evaluate different types
of program input sensitivities and to perform high-precision low-overhead online code variant selection for each input on the fly.

HPC programs are usually multi-threaded. Threads share the same code (control flows) but operate on different data and often shared which is significantly more complicated than a sequential program. Execution depends on proper synchronizations across threads to ensure correctness and efficiency. While a common set of standard synchronizations, such as muted and condition variable operations, are provided by different languages or libraries, a recent study [Xio10] finds that programmers frequently choose not to use these standard synchronizations but implement their own ad hoc synchronizations for functionality or performance reasons. Researchers were able to find 6 to 83 ad hoc synchronizations in each of the 12 studied program suites [Xio10].

Not only do programmers have difficulty in understanding the intended order relationship by ad hoc synchronizations and validate their correctness [Gu15], but also multi-threaded program development tools, such as data-race detectors [Bes10; Lee12], concurrency-bug finding tools [Par09; Zha10], automated bug-fixing tools [Jin11; Jin12], and synchronization-oriented performance profilers [YP16; CS12], cannot directly use ad hoc synchronization detection results of these existing tools.

State-of-art techniques can only detect sync pairs [Xio10] or simple busy-wait loops [JT14; Yua13], which is insufficient to understand complex ad hoc synchronizations. We propose a framework to automatically identify complex ad hoc synchronizations in full and infer their synchronization relationship for barriers. We instantiate the framework with a tool called BARRIERFINDER, which features various techniques, including program slicing and bounded symbolic execution, to efficiently explore the interleaving space of ad hoc synchronizations within multi-threaded programs with respect to their traces. BARRIERFINDER then uses these traces to recognize ad hoc barriers. Our evaluation shows that BARRIERFINDER is both effective and efficient in recognizing ad hoc barriers automatically.

1.3 Hypothesis

The thesis hypothesis (H) is that HPC programs can benefit in performance and correctness verification from application-specific auto-tuning and validation of ad hoc synchronization constructs, respectively. In particular, H is refined by the following three sub-hypotheses:

H1 Different compilation modules of HPC programs may need different compilation flags to obtain the best performance due to their diverse code structures.

H2 HPC programs may expose various input sensitivity and can benefit from machine learning techniques to realize better per-input performance.

H3 Symbolic execution may automatically detect complex ad hoc synchronizations, validate their semantics correctness, and detect implementation errors for HPC programs.
1.4 Solutions and Contributions

To evaluate our over-arching hypothesis and its three sub-hypotheses, we take an experimental approach to design and implement three frameworks, namely, FuncyTuner (see Chapter 2), CodeSeer (see Chapter 3), and BARRIERFINDER (see Chapter 4). Then, we design and conduct several experiments for each of them and draw conclusion based on our observations.

1. In Chapter 2, we propose a fine-grained compilation framework, FuncyTuner, which utilizes per-loop profiling information to guide the search within the extremely large compilation flag search space. Compared to the state-of-art, FuncyTuner improves performance of modern parallelized scientific programs by 4.5% to 10.7% (geometric mean) relative to the O3 baseline. We also show that greedily picking the best per-loop compiler flags often degrades program performance due to complex inter-module dependencies, and per-function random search without guidance of runtime information does not guarantee performance improvements.

2. In Chapter 3, we propose CodeSeer to evaluate several different types of program sensitivity and build six machine learning models to tackle the challenges presented by type-III sensitivity. Our experimental results show that so-called type-I input sensitivity could have significant performance impacts for certain programs and tuning inputs should be selected carefully. For programs with high type-III sensitivity, CodeSeer predictive models achieve 92% prediction accuracy while introducing less than 0.01 second online prediction overhead.

3. In Chapter 4, we propose a framework to automatically identify complex ad hoc synchronizations in full and infer their synchronization relationships. We instantiate the framework with a tool called BARRIERFINDER, which features various techniques, including program slicing and bounded symbolic execution, to efficiently explore the interleaving space of ad hoc synchronizations within multi-threaded programs for their traces. BARRIERFINDER then uses these traces to characterize ad hoc synchronizations into different types, such as barriers. Our evaluation shows that BARRIERFINDER is both effective and efficient in doing this, and it is also helpful for programmers to understand the correctness of their implementations.

In Chapter 5, we summarize the contributions of this dissertation in the context of our initial hypotheses, discuss the limitations of our work, and outline possible improvements subject to future work.
The de facto compilation model for production software compiles all modules of a target program with a single set of compilation flags, typically O2 or O3. Such a per-program compilation strategy may yield sub-optimal executables since programs often have multiple hot loops with diverse code structures and may be better optimized with a per-region compilation model that assembles an optimized executable by combining the best per-region code variants.

In this paper, we demonstrate that a naïve greedy approach to per-region compilation often degrades performance in comparison to the O3 baseline. To overcome this problem, we contribute a novel per-loop compilation framework, FuncyTuner, which employs light-weight profiling to collect per-loop timing information, and then utilizes a space-focusing technique to construct a performant executable. Experimental results show that FuncyTuner can reliably improve performance of modern scientific applications on several multi-core architectures by 9.2% to 12.3% and 4.5% to 10.7% (geometric mean, up to 22% on certain program) in comparison to the O3 baseline and prior work, respectively.
2.1 Introduction

The de facto compilation model compiles all source files of a program with a single set of compiler flags, typically O2 or O3. However, it is well-known that O2/O3 may not generate the most performant executables [Che10]. This is because optimizations enabled by flags such as O2/O3 are empirically determined to maximize performance for certain benchmark suites, while other programs and architectural platforms may have different characteristics that require other optimizations. As a result, for a given program, other compiler flag combinations may exist that can produce more performant executables than the default O2/O3 choice.

In order to identify the most performant optimizations for a program, researchers have proposed iterative compilation [Kis00]. Given a program, iterative compilation first generates code variants by compiling the source code with different compiler flags, and then evaluates their performance either by execution [Che10] or prediction [Cav07; Pop16].

Several algorithms have been proposed for generating compiler flag combinations to create code variants, e.g., predictive machine learning models [Ash16; Cav07; Nob16; JK13] and optimization flag correlation-based combined elimination [PE08], both of which treat random search as their baselines. They perform compilation on a per-program basis. However, their effectiveness is limited. While machine learning based predictive models [Cav07] perform well on small training datasets, a recent study [Fur15] employing crowd-sourcing compilation shows that their prediction accuracy drops dramatically for large-scale datasets and exhibits close to random behavior.

Combined elimination (CE) [PE08] takes advantage of interactions among compiler flags to find the best combination. However, our evaluation of CE, shown in Fig. 2.1, for three benchmarks on Intel’s Broadwell architecture shows minimal performance benefit in comparison to the O3 baseline for both the GNU C/C++ compiler (GCC release 5.4.0) and the Intel C/C++ compiler (ICC release 17.0.4). A closer inspection of the experiments for CE revealed that it can be limited by the results for local minima. Further, the time complexity of CE is $O(n^2)$, where $n$ is the number of O3 flags. The quadratic complexity limits the use of CE for newer GCC versions such as 5.4.0, which have more than 200 binary optimization flags and 170 multi-valued parameters. In this work, we address the challenge of a large search space by employing a novel search space focusing technique to guide
random search.

Fine-grained per-region compilation techniques [PE08; Pop16; Hal10; Yi12] divide a program into different compilation modules and optimize each separately. Specifically, source-code level auto-tuners [Hal10; Yi12] focus on a single simple computation kernel without considering module interactions in real-world applications. As prior work in compiler flag selection [PE08; Pop16] also assumes that compilation modules are independent, they assemble an optimized executable by greedily picking the best code variant of each module. However, the modules of a program may not be independent due to cross-module interference, such as shared data structures and link-time inter-procedural optimizations across multiple modules. In particular, link-time optimizations can drive optimization decisions, such as loop unrolling, and may invalidate earlier transformations that were made independently for the compilation modules.

Contributions: Based on the observations above, we develop a fine-grained auto-tuning framework, FuncyTuner, that targets modern scientific applications. Our overall objective is to extract the best performance out of an application that is executed repeatedly, such as in high-performance computing (HPC) where scientists test their hypotheses in experiments repeatedly with similar inputs using the same algorithms. To clarify, we neither attempt to derive a better set of optimizations for O3, nor do we attempt to generalize a specific set of optimizations across region boundaries or select different algorithmic code variants according to input characteristics (in contrast to [Li13]). Instead, our objective is to 1) assess whether or not there are module interactions and, if so, 2) understand how to capitalize on such interactions.

Our target HPC applications exploit multi-core parallelism via OpenMP. Their hot-spots consist of OpenMP loops that account for a significant fraction of execution time. FuncyTuner outlines these loops and converts them into individual functions, whose compilation can be auto-tuned. The primary challenge is to identify the best per-loop compilation flags within a significantly larger space than any traditional per-program compilation techniques. To this end, FuncyTuner employs a novel and effective search space focusing technique to guide random search based auto-tuned compilation.

The contributions of this paper are:

• We develop a per-loop compiler flag selection framework, FuncyTuner, that combines program profiling with Caliper [Boe16] and search space focusing algorithms to tune hot loops of modern scientific programs simultaneously without sacrificing the indispensable optimization context for production compilers.

• We demonstrate that FuncyTuner is able to improve program performance for a set of scientific benchmarks by 9.3% to 12.3% in comparison to the O3 baseline and 4.5% to 10.7% relative to prior work, both in geometric mean (up to 22% on certain programs), on several generations of HPC architecture, and thus refreshes the state of the art. Our FuncyTuner further has the potential to improve machine learning based predictive modeling techniques in the future.

• We conduct an in-depth case study for Cloverleaf [UM18] on the Intel Broadwell architecture
Figure 2.2 Comparison of compilation models. a) A traditional compilation model with the same CV to compile all modules; b) FuncyTuner’s compilation model with different CVs to compile different modules.

with the Intel compiler tool chain to elaborate how fine-grained compiler flag selection is affected by inter-module dependencies and demonstrate that it should not be performed greedily, but rather in a focused and targeted manner.

Our FuncyTuner implementation automates these steps, with the exception of Caliper [Boe16] profile instrumentation and collection of timing results, which are manual in our research prototype, but could be automated with further engineering efforts invested (but are of no research value).

2.2 Design of FuncyTuner

This section first defines terminologies and notations used throughout this work and then details the design of the FuncyTuner framework.

2.2.1 Compiler Flag Space Construction

Modern optimizing compilers feature many internal optimization passes and may expose several hundreds of command-line flags to parameterize them. Each flag could either be a binary switch to turn on/off a certain optimization, e.g., loop unrolling and loop tiling, or a multi-valued parametric option to set pass-specific parameters, e.g., thresholds for function inlining and algorithmic variants of register allocation. The set of all flags composes a space called the compiler optimization space (COS, size roughly $2.3 \times 10^{13}$ in this work since there are 33 selected flags while some have multiple values), in which each point is a set of instantiated flags called a compilation vector (CV). Suppose there are $N$ compiler flags, denoted as $F_i (1 \leq i \leq N)$, and suppose $F_i$ has $n_i$ possible values $f_{i1}, f_{i2}, ..., f_{in_i}$. Then a sample CV is represented as $(F_1 = f_{1k_1}, F_2 = f_{2k_2}, ..., F_N = f_{Nk_N})$, where $1 \leq k_i \leq n_i$. Thus, there are in total $C_0 = \prod_{i=1}^{N} n_i$ CVs, each of which could be used to compile all source files of a program in a traditional compilation model (see Fig. 2.2a).

Given a program $P$, a traditional compilation model (see Fig. 2.2a) treats all source files as a single compilation module $M$, within which the source files are compiled with the same CV. In contrast,
as shown in Fig. 2.2b, FuncyTuner divides program $P$ into $J$ compilation modules $M_1, M_2, \ldots, M_J$ ($J$ is program-specific and ranges from 5 to 33 in this work). These modules are created based on the time spent in various regions, in particular loops, of the program $P$. FuncyTuner then compiles $M_j$ with a $CV$, which could be determined independently from other modules and links all object files together to produce the executable. Our hypothesis is that different compilation modules may need different $CV$s to obtain the best performance due to their diverse code structures. However, to identify the best $CV$s, the primary challenge is that the new search space size $COS_{new}$ increases significantly from $C_0$ to $C_1 = C_0^J$. ($J$ is program-specific and ranging from 5 to 33 in this work), since each module can be compiled independently with a different $CV$ in COS. Exhaustive search is not a viable option within such an excessively large space while machine learning-based predictive models require a significant amount of training data and meaningful features to begin with [Fur15]. To address this challenge, we propose several algorithms, as detailed in the rest of this section. We do not differentiate between a program source code module and its corresponding object/binary module. Therefore, $P_k$ and $M_{jk}$ (j-th compilation module of $P_k$) may represent the source code module or compiled program object/binary module, depending on the context.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2_3.png}
\caption{Per-program random search. $CV_k$ and $P_k$ with minimal runtime $T_k$ is its final result.}
\end{figure}

2.2.2 Space Search Algorithms

In this section, we introduce four different search algorithms. Note that per-program random search is a classical algorithm and is used as a reference to understand the other three algorithms.

2.2.2.1 Per-program Random Search

Per-program random search (denoted as Random) does not modify program source code and applies a single $CV$ to all source files of any program $P$ within the traditional compilation model (Fig. 2.2a). As shown in Fig. 2.3, in step ①, 1000 $CV$ samples are randomly selected from $COS$. In step
Algorithm 1: Per-program Random Search

**Input**: COS, K, P, T03

**Output**: speedup, CV

1. \( k = 0, \ T = [], \ \text{CVs} = [] \)
2. \( \text{CVs} = \text{randomSample}(\text{COS}, K) \)
3. while \( k < K \) do
   4. compile \( P \) with \( \text{CVs}[k] \) to generate \( P_k \)
   5. run \( P \) to measure \( T_k \)
   6. \( T[k] = T_k \)
   7. \( k++ \)
   8. \( k = \text{argmin}_k \{ T[k] \mid 1 \leq k \leq K \} \)
9. \( \text{CV} = \text{CVs}[k] \)
10. speedup = \( T_{03}/T[k] \)

, each CV is used to compile \( P \) to obtain a code variant, \( P_k \) (1 \( \leq k \leq 1000 \)). In step (3), runtimes are collected for all code variants. In the end, the code variant with the least runtime is selected as the result. The size of its search space is \( C_0 \), and the pseudo code is presented in Algorithm 1.

2.2.2.2 Per-function Random Search

As shown in Fig. 2.4, per-function random search (denoted as FR) begins with Caliper profiling a program \( P \) to identify hot loops and outlines each of them into a separate source file so that there are \( J \) compilation modules. In step (3), \( J \) CVs are randomly selected from 1000 pre-sampled CVs with replacement. Each selected CV is used to compile one of the \( J \) compilation modules. Note that
the selection of $J$ CVs and compilation is performed 1000 times in step (3) to generate 1000 code variants, which are executed to collect runtimes $T_1, ..., T_{1000}$. FR reports the code variant with the minimum runtime as the best version.

FR uses Caliper [Boe16] profiling only for identifying hot loops, but it does not collect the per-loop runtime information for searching the best per-loop CVs. However, when such information is available, one may choose better CVs for each compilation module. To this end, we introduce FuncyTuner, our per-loop runtime collection framework, shown in Fig. 2.5. As in FR, a program $P$ is first divided into $J$ compilation modules. In step (2), FuncyTuner instruments modules via Caliper’s light-weight [Boe16] APIs to measure per-loop runtimes. Then, in step (4), 1000 pre-sampled CV$_1$, ..., CV$_{1000}$ are used to compile $P$ such that all modules within $P$ are compiled with the same k-th CV to generate $P_k$. In step (5), the generated 1000 code variants are executed to collect per-loop runtimes, denoted as $T_{jk}$ for the module $M_j$ of the code variant $P_k$. This information is utilized in different ways by the next two algorithms, namely greedy combination (see Section 2.2.2.3) and Caliper-guided random search (see Section 2.2.2.4). The detailed algorithm is presented in Fig. 2.

---

**Algorithm 2: Per-function Random Search (FR)**

1. // For FR, CV$_{pruned}$ initially is empty
   
   **Input**: COS, K, P, T$_{O3}$
   
   **Output**: speedup, CV

2. $T = [], \ textCVs = [], CVs = []$

3. // returns K random samples from COS

4. **for** $k = 0; k < K; k++$ **do**

5.   **tempCVs[k][] = randomSample(CVs, J)**

6. **for** $j = 0; j < J; j++$ **do**

7.   compile $M_j$ with tempCVs[k][j]

8. link $M_1, ..., M_J$ to generate $P_k$

9. run $P_k$ to measure $T_k$

10. $T[k] = T_k$

11. $k = \text{argmin}_k \{T[k] \, | \, 1 \leq k \leq K \}$

12. $CV = \text{tempCVs}[k]$

13. speedup = $T_{O3}/T[k]$

---

### 2.2.2.3 Greedy Combination

Greedy combination (denoted as $G$) exploits per-loop runtimes in a straight-forward way. It assembles the final executable by picking the fastest code variant for each module and links them together, assuming that there are no inter-module dependencies such that the greedy composition produces the fastest executable. Formally, $G$ chooses the i-th CV to compile $M_j$ such that
Figure 2.5 FuncyTuner per-loop runtime collection framework is a part of both greedy combination and Caliper-guided random search.

Algorithm 3: Greedy Combination (G)

Input: \(COS, K, P, T_{O3}\)

Output: speedup, CV

1. \(T = [[]], CVs = [], CV_{\text{greedy}} = []\)
2. \(CVs = \text{randomSample} \ (COS, K)\)
3. // FuncyTuner per-loop runtime data collection
4. for \(k = 0; k < K; k++\) do
5. compile \(P\) with \(CVs[k]\) to generate \(P_k\)
6. run \(P\) to measure \(T_{1k}, ..., T_{Jk}\)
7. for \(j = 0; j < J; j++\) do
8. \(T[j][k] = T_{j+k}^1\)
9. //Greedy combination
10. for \(j = 0; j < J; j++\) do
11. \(CV_{\text{greedy}}[j] = \text{argmin}_k \ \{T_{jk}|1 \leq k \leq K\}\)
12. compile \(M_j\) with \(CV_{\text{greedy}}[j]\) for object file \(O_j\)
13. link \(O_j\)s to generate target program \(P_t\)
14. execute \(P_t\) to obtain \(T_t\)
15. \(CV = CV_{\text{greedy}}\)
16. speedup = \(T_{O3}/T_t\)
\( i = \arg \min_k \{ T[jk] \mid 1 \leq k \leq 1000 \} \), and then links all modules to produce the target executable. The pseudo code for both the FuncyTuner data collection framework and Greedy Combination is presented in Algorithm 3.

### 2.2.2.4 Caliper-guided Random Search

**Algorithm 4: Caliper-guided Random Search (CFR)**

**Input**: \( \text{COS}, K, P, X, T_{O3} \)

**Output**: speedup, \( CV \)

1. \( k = 1, K = 1000, T = [], CVs = [], CV_{pruned} = [] \)
2. \( CVs = \text{randomSample} (\text{COS}, K) \)
3. //step-1: FuncyTuner per-loop data collection
4. \textbf{for} \( k = 0; k < K; k++ \) \textbf{do}
5. \( \text{compile } P \text{ with } CVs[k] \text{ to generate } P_k \)
6. \( \text{run } P \text{ to measure } T_{ik}, ..., T_{jk} \)
7. \textbf{for} \( j = 0; j < J; j++ \) \textbf{do}
8. \( T[j][k] = T_{j+1k} \)
9. //prune pre-sampled 1000 CVs
10. \textbf{for} \( k = 0; k < K; k++ \) \textbf{do}
11. \( CV_{pruned}[j] = \{ CVs[i] \mid T[j][i] \text{ is among top–X smallest in } T[j][0], ..., T[j][K-1] \} \)
12. \textbf{for} \( k = 0; k < K; k++ \) \textbf{do}
13. //re-sampling per-loop cv in pruned space.
14. \textbf{for} \( j = 0; j < J; j++ \) \textbf{do}
15. \( \text{tempCVs}[k][j] = \text{randomSample} (CV_{pruned}[j], 1) \)
16. \textbf{for} \( j = 0; j < J; j++ \) \textbf{do}
17. \( \text{compile } M_j \text{ with } \text{tempCVs}[k][j] \)
18. \( \text{link } M_1, ..., M_j \text{ to generate } P_k \)
19. //\( T_k \): end-to-end runtime for \( P_k \) and \( T_k = \sum_{j=1}^{J} T_{jk} \)
20. \( \text{run } P_k \text{ to measure } T_k \)
21. \( T[k] = T_k \)
22. \( k = \arg \min_k \{ T[k] \mid 1 \leq k \leq K \} \)
23. \( CV = \text{tempCVs}[k] \)
24. //\( T_{O3} \): end-to-end runtime for \( P \) compiled with \( O3 \)
25. \( \text{speedup} = T_{O3}/T[k] \)

Similar to \( G \), Caliper-guided random search (denoted as \( CFR \), see Algorithm 4) also relies on per-loop runtime information to make informed selections of CVs. However, \( CFR \) examines more code variants than \( G \) to take potential inter-module dependencies into consideration. In contrast to \( FR \), which does not use any per-loop information, and uniformly performs a random search within the pre-sampled 1000 CVs for each hot loop, \( CFR \) prunes the pre-sampled search space for...
each hot loop before re-sampling per-loop CVs (line 13 of Algorithm 4). The intuition is that more performant CVs, which generate faster per-loop code variants, should be kept in the re-sampling search space, because they may have a higher chance to compose a performant target executable. Within a unified algorithmic framework, G can be considered as only selecting the top-1 CVs, and that FR selects all 1000 or the top-1000 CVs, while CFR selects the top-X (1 \( < X \ll 1000 \)) CVs, all on a per-loop basis.

To summarize, Random is a traditional search algorithm that performs on per-program granularity, while FR, G, and CFR perform on a per-loop basis but with different mechanisms and motivations: FR is to evaluate if random search with per-loop granularity alone is sufficient to achieve the best performance; G is to assess if there are inter-module dependencies by evaluating the effectiveness of greedily combining the best per-loop code variants; and CFR is to see if a focused search space can improves the performance beyond that of Random, FR and G. Note that FR and CFR are proposed by us, while Random and G are based on prior work [PE08; Pop16].

2.3 Experimental Design

In order to evaluate the efficacy of schemes presented in Section 2.2, we have performed experiments for seven modern scientific benchmarks on three architectures. This section describes the setup used in our experiments.

2.3.1 Systems and Benchmarks

We conducted our experiments on three platforms: AMD Opteron, Intel Sandy Bridge, and Intel Broadwell. The architectural details for these systems are provided in Table 2.2. Our benchmark suite (see Table 3.1) consists of seven HPC programs: AMG [Lab18], LULESH [Lab18], Cloverleaf (CL) [UM18], 351.bwaves, 362.fma3d, 363.swim and Optewe [Sou18]. 351.bwaves, 362.fma3d, and 363.swim are from the SPEC OMP 2012 suite, while the rest are widely used HPC proxy applications. These benchmarks have been selected based on two criteria. First, they are written in different languages and exploit multi-core parallelism suitable for HPC via OpenMP pragmas. Second, they feature more than one hot loop, which resembles realistic applications (unlike many other benchmarks with just a single hot loop). While multiple hot loops present difficulties for the compiler in coordinating various loop optimizations, they also provide an opportunity to optimize different parts of the code differently.

All our experiments have been run on CentOS Linux 7.3.1611 and the benchmarks were compiled with the Intel C/C++ Compiler 17.04. OpenMP thread placement has been set to fine, proclist=\([...\] explicit, where proclist is specified in Table 2.2. Details of the OpenMP configurations are presented in Table 2.2. Since scientific codes follow a time-step execution pattern repeatedly performing approximations with decreasing numerical error in an outer loop, we only run for a small number of time-steps (seconds) and then exit prematurely once we have obtained a stable execution time for a time-step. Any optimization then scales up to a full run over all time-steps (hours). To this end,
Table 2.1 List of benchmarks. LOC: lines of source code.

<table>
<thead>
<tr>
<th>Name</th>
<th>Language</th>
<th>LOC</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMG</td>
<td>C</td>
<td>113k</td>
<td>Math: linear solver</td>
</tr>
<tr>
<td>LULESH</td>
<td>C++</td>
<td>7.2k</td>
<td>Hydrodynamics</td>
</tr>
<tr>
<td>Cloverleaf (CL)</td>
<td>C, Fortran</td>
<td>14.5k</td>
<td>Hydrodynamics</td>
</tr>
<tr>
<td>351.bwaves</td>
<td>Fortran</td>
<td>1.2k</td>
<td>Computational fluid dynamics</td>
</tr>
<tr>
<td>362.fma3d</td>
<td>Fortran</td>
<td>62k</td>
<td>Mechanical simulation</td>
</tr>
<tr>
<td>363.swim</td>
<td>Fortran</td>
<td>0.5k</td>
<td>Weather prediction</td>
</tr>
<tr>
<td>Optewe</td>
<td>C++</td>
<td>2.7k</td>
<td>Seismic wave simulation</td>
</tr>
</tbody>
</table>

input sizes and time-steps (iteration counts of simulation outer loop) have been adjusted so that every single run is less than 40 seconds for the O3 baseline compilation. In the first experiments (Section 2.4.1 and Section 2.4.2), we use the same inputs for tuning and testing, whereas we evaluate the impact of different inputs (Section 2.4.3) in later experiments to reflect typical usage patterns of repeated HPC application runs with different scientific inputs. To account for run-to-run variability, each code variant is executed 10 times and their average is presented as the representative runtime.

2.3.2 Compiler Flag Selection

We experiment with 33 optimization-related compilation flags of the Intel compilers. For flags that support any value in a continuous range as input, we discretize the values in the given range. Then, for each flag $F_i$, FuncyTuner selects a value $f_i$ from $f_{i1}, f_{i2}, ..., f_{in_i}$ with equal probability. A CV is constructed by concatenating the selected values for all $F_i$’s. We had to consider several restrictions when selecting the flags. First, a flag must not prevent a program from running successfully on a given target architecture. For example, use of the -fpack flag generates code variants that cause a segmentation fault at runtime and thus -fpack is excluded. Second, for fair performance comparison among different code variants, FuncyTuner enforces strict floating point reproducibility by discarding floating point related optimization flags, and always uses -fp-model source in the presented results. Last, optimized library options, such as Intel MKL and IPP related linkage options, are also excluded since they are not used by our benchmarks.

Also, in order to reach the full optimization potential of the Intel compiler tool chain, according to the Intel optimization note, Intel’s linker xild and library archive tool xiar should be used. We thus modify build systems of all benchmarks accordingly. Processor-specific flags are also considered for the best performance on each architecture. Most flags take effects independent of each other. When indicated by ICC manual, we put flags that can cancel out others in leftmost position, e.g., -finline-function is to the left of -finline-limit since the latter is a modifier of the former.
Table 2.2 Platform overview, runtime configurations, and benchmark inputs.

<table>
<thead>
<tr>
<th>Machine</th>
<th>AMD</th>
<th>Intel</th>
<th>Intel</th>
<th>Intel Broadwell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>Opteron 6128</td>
<td>Xeon E5-2650 v4</td>
<td>Xeon E5-2620 v4</td>
<td></td>
</tr>
<tr>
<td>Sockets</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>NUMA nodes</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Cores/Socket</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Threads/Core</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Core Frequency [GHz]</td>
<td>2.0</td>
<td>2.0</td>
<td>2.1</td>
<td></td>
</tr>
<tr>
<td>processor-specific flag</td>
<td>default</td>
<td>-xAVX</td>
<td>-xCORE-AVX2</td>
<td></td>
</tr>
<tr>
<td>Memory size [GB]</td>
<td>32</td>
<td>16</td>
<td>64</td>
<td></td>
</tr>
<tr>
<td>OpenMP thread count</td>
<td>16</td>
<td>16</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>OpenMP thread proclist</td>
<td>[0-15]</td>
<td>[0-15]</td>
<td>[0-15]</td>
<td></td>
</tr>
<tr>
<td>LULESH: size, steps</td>
<td>120, 10</td>
<td>150, 10</td>
<td>200, 10</td>
<td></td>
</tr>
<tr>
<td>Cloverleaf: size, steps</td>
<td>2000,30</td>
<td>2000,30</td>
<td>2000,60</td>
<td></td>
</tr>
<tr>
<td>AMG: size</td>
<td>18</td>
<td>20</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Optewe: size, steps</td>
<td>320, 5</td>
<td>384, 5</td>
<td>512, 5</td>
<td></td>
</tr>
<tr>
<td>bwaves: input, steps</td>
<td>train, 10</td>
<td>train, 15</td>
<td>train, 50</td>
<td></td>
</tr>
<tr>
<td>fma3d: input</td>
<td>train</td>
<td>train</td>
<td>train</td>
<td></td>
</tr>
<tr>
<td>swim: input</td>
<td>train</td>
<td>train</td>
<td>train</td>
<td></td>
</tr>
</tbody>
</table>

2.3.3 Loop Outlining and Caliper Instrumentation

To identify hot loops that need to be outlined into individual modules, FuncyTuner uses Caliper [Boe16] to profile the target application compiled with `-O3 -qopenmp -fp-model source`. Every loop whose runtime is at the least 1.0% of the baseline’s end-to-end runtime is outlined as an independent compilation module for maximum freedom of CV selection. The runtime for code other than the hot loops (non-loop code) cannot be directly measured because such code tends to be scattered across many source files. Thus, the runtime of non-loop code is derived by subtracting the aggregate runtime of hot loops from the end-to-end runtime for each code variant of a program. Caliper instrumentations generally introduce less than 3% overhead and the per-loop runtimes are sufficiently informative to FuncyTuner so that measurement noise is tolerated with its search algorithms. To evaluate performance, we use `-O3 -qopenmp -fp-model source` as the baseline and report speedups relative to this baseline unless otherwise specified.

2.3.4 CV Independence Assumption

Note that there are two sets of results for the greedy combination. One is obtained by runtime measurement and is denoted as $G$.realized. The other marked $G$.Independent is calculated by summing up the best per-loop and non-loop code runtimes obtained with different CVs. $G$.Independent is used as the hypothetical upper bound for the greedy combination and serves as a reference to assess
Figure 2.6 CFR outperforms other methods for most cases: geometric mean of speedups relative to the O3 baseline 9.2%, 10.3%, and 9.4% on Opteron, Sandy Bridge and Broadwell, respectively.

if there is pairwise independence among different compilation modules.

2.4 Results and Analysis

In this section, we first present results of the four algorithms in Sec. 2.2 on three HPC architectures. We then compare FuncyTuner CFR with prior work, and study their sensitivity on different inputs. To shed light on why CFR performs the best, we conduct a case study on Cloverleaf.

2.4.1 Overall Performance Comparison

Fig. 2.6 compares the performance for the four algorithms in Sec. 2.2, i.e., Random is the classical per-program random search; FR and CFR are the two per-loop algorithms proposed by us; G.realized and G.Independent are results for greedy combination G as explained in Sec. 2.3.4. Note that Random is applied on the original benchmarks while others are all applied on the benchmarks.
Table 2.3 Performance Statistics on Intel Broadwell

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Lulesh</th>
<th>Cloverleaf</th>
<th>AMG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>stdev</td>
<td>Mean</td>
</tr>
<tr>
<td>G.expected</td>
<td>10.9</td>
<td>N/A</td>
<td>5</td>
</tr>
<tr>
<td>G.realized</td>
<td>21.52</td>
<td>1.4</td>
<td>8.92</td>
</tr>
<tr>
<td>O3.origin</td>
<td>13.57</td>
<td>0.04</td>
<td>5.82</td>
</tr>
<tr>
<td>O3.caliper</td>
<td>13.91</td>
<td>0.05</td>
<td>5.91</td>
</tr>
<tr>
<td>O3outlined</td>
<td>13.82</td>
<td>0.04</td>
<td>5.79</td>
</tr>
<tr>
<td>R</td>
<td>12.66</td>
<td>0.07</td>
<td>5.37</td>
</tr>
<tr>
<td>FR</td>
<td>12.76</td>
<td>0.07</td>
<td>5.7</td>
</tr>
<tr>
<td>CFR</td>
<td>12.11</td>
<td>0.42</td>
<td>5.05</td>
</tr>
</tbody>
</table>

Table 2.4 Performance Statistics on Intel Sandy Bridge

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Lulesh</th>
<th>Cloverleaf</th>
<th>AMG</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>stdev</td>
<td>Mean</td>
</tr>
<tr>
<td>G.expected</td>
<td>5.7</td>
<td>N/A</td>
<td>3.7</td>
</tr>
<tr>
<td>G.realized</td>
<td>9.3</td>
<td>0.16</td>
<td>6.38</td>
</tr>
<tr>
<td>O3.origin</td>
<td>7.22</td>
<td>0.05</td>
<td>4.63</td>
</tr>
<tr>
<td>O3.caliper</td>
<td>8.0</td>
<td>0.05</td>
<td>4.76</td>
</tr>
<tr>
<td>O3outlined</td>
<td>7.39</td>
<td>0.08</td>
<td>4.65</td>
</tr>
<tr>
<td>R</td>
<td>6.31</td>
<td>0.17</td>
<td>4</td>
</tr>
<tr>
<td>FR</td>
<td>6.24</td>
<td>0.12</td>
<td>4.54</td>
</tr>
<tr>
<td>CFR</td>
<td>6.19</td>
<td>0.18</td>
<td>3.9</td>
</tr>
</tbody>
</table>

with their hot loops outlined. Moreover, Caliper instrumentations are needed only in the per-loop runtime collection but not in the final optimized executables. For all results over the 7 programs on both training and testing inputs, execution times were between 3 and 36 seconds with a standard deviation of 0.04 to 0.2 (except for two cases with 1.5 and 0.7 for longer LULESH runs) measured over 10 experiments, i.e., results are very uniform with high statistical significance. Detailed performance statistics are presented in Tables 2.3 to 2.5. From these results, we make the following observations.

From these results, we make the following observations.

1. FuncyTuner CFR provides the best performing executables for most scenarios across benchmarks and architectures. It provides 9.2%, 10.3%, 9.4% geometric mean speedups for Opteron, Sandy Bridge and Broadwell, respectively. It also achieves the best case improvement of 18.1% for AMG on AMD Opteron (see Fig. 2.6a) in comparison to the O3 baseline. In contrast, the performance improvement due to Random is only 3.4%, 5.0%, 4.6% on the same respective architectures. In certain cases, Random does not improve performance at all while CFR does much better, e.g., for AMG on Sandy Bridge and Broadwell.

2. G results in significant slowdowns for many benchmark and architecture combinations. Although it improves performance of AMG on Opteron and Sandy Bridge, the improvement
Table 2.5 Performance Statistics on AMD Opteron

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Lulesh</th>
<th></th>
<th></th>
<th>Cloverleaf</th>
<th></th>
<th></th>
<th>AMG</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>stdev</td>
<td></td>
<td>Mean</td>
<td>stdev</td>
<td></td>
<td>Mean</td>
<td>stdev</td>
<td></td>
</tr>
<tr>
<td>G.expected</td>
<td>9.7</td>
<td>N/A</td>
<td></td>
<td>5.73</td>
<td>N/A</td>
<td></td>
<td>9.4</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>G.realized</td>
<td>14.52</td>
<td>0.37</td>
<td></td>
<td>11.01</td>
<td>0.15</td>
<td></td>
<td>9.6</td>
<td>0.07</td>
<td></td>
</tr>
<tr>
<td>O3.origin</td>
<td>11.1</td>
<td>0.08</td>
<td></td>
<td>6.88</td>
<td>0.16</td>
<td></td>
<td>10.27</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>O3.caliper</td>
<td>11.05</td>
<td>0.05</td>
<td></td>
<td>7.38</td>
<td>0.09</td>
<td></td>
<td>10.99</td>
<td>0.62</td>
<td></td>
</tr>
<tr>
<td>O3.outlined</td>
<td>10.84</td>
<td>0.03</td>
<td></td>
<td>6.89</td>
<td>0.14</td>
<td></td>
<td>10.13</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>R.origin</td>
<td>10.87</td>
<td>0.08</td>
<td></td>
<td>6.96</td>
<td>0.12</td>
<td></td>
<td>10.19</td>
<td>0.37</td>
<td></td>
</tr>
<tr>
<td>FR</td>
<td>10.27</td>
<td>0.05</td>
<td></td>
<td>6.7</td>
<td>0.07</td>
<td></td>
<td>9.73</td>
<td>0.92</td>
<td></td>
</tr>
<tr>
<td>CFR</td>
<td>10.16</td>
<td>0.05</td>
<td></td>
<td>6.06</td>
<td>0.08</td>
<td></td>
<td>8.7</td>
<td>0.27</td>
<td></td>
</tr>
</tbody>
</table>

is still inferior to that of FuncyTuner CFR. The huge differences between G.realized and G.Independent substantiate that there are inter-module dependencies for benchmarks in our experiment.

3. FR's performance is inferior to that of CFR and has high variance. For example, it achieves less than a 3% improvement for Cloverleaf on Opteron, Sandy Bridge and Broadwell, while CFR achieves 13.6%, 15.2% and 12.7%, respectively. Such results demonstrate that random search with per-loop granularity alone is insufficient for achieving the best performance and CFR's effective utilization of per-loop runtime information is critical.

2.4.2 Comparison to the State-of-the-art

We first introduce the state-of-art techniques and experimental settings to fairly compare with FuncyTuner CFR on Intel Broadwell. Then, we present the results and our observations that CFR outperforms all of them.

2.4.2.1 The State-of-the-Art

Prior techniques are either search-based [Pop16; Ans14] or predictive modeling-based [Ash16]. In particular, Cere [Pop16] performs a fine-grained compiler flag selection for hot code regions and generates the optimized executable in the same greedy fashion as G in our work. Our results in Fig. 2.6 show that this often degrades performance for the Intel compilers.

OpenTuner [Ans14] performs per-program search with an ensemble of search algorithms, including differential evolution, Torczon hillclimbers, Nelder-Mead and many others. It also employs a meta search (AUC Bandit) technique to coordinate different search algorithms for the best performance. To compare with FuncyTuner CFR, we run OpenTuner with 1000 test iterations using the same CV search space.

COBAYN [Ash16], a state-of-the-art machine learning-based approach, infers performant CVs for a new program by extracting static and dynamic program features and providing them as inputs to a pre-trained Bayesian network. To compare, we first train COBAYN with cBench [Fur18]. Specifically,
we select the top 100 performant CVs out of 1000 random CV samples for each cBench application to extract their static and dynamic features with Milepost-gcc [Fur11] and Mica [HE07]. We then train three models, static, dynamic, and hybrid, using static features, dynamic features and all features, respectively. Since COBAYN can only perform inferences on binary compiler flags, we turn each multi-valued ICC flag into a binary one by allowing it to have two values. Then, we use each of the three models to generate 1000 code variants. The fastest code variant is considered as the result of each model.

Aside from the above meta-compilation techniques, Intel compilers support built-in profile-guided optimization (PGO), which utilizes an instrumentation run of a target program to collect profile information, such as loop trip counts and indirect function call targets. The comparison to PGO offers a perspective to evaluate the trade-off between benefits and complexities for all approaches. We use -qopenmp -fp-model source -prof-gen -prof-dir /app for an instrumented compilation (see recommendations for PGO in Intel’s compiler optimization manual) and then run the programs with tuning inputs in Table 2.2. Afterward, the programs are recompiled with -O3 -qopenmp -fp-model source -prof-use -prof-dir /app.

2.4.2.2 Observations

We make the following observations from the experimental results shown in Fig. 2.7.

1. OpenTuner achieves a 4.9% geometric mean speedup on our benchmark suite and is 4.5% inferior to that of FuncyTuner CFR. In particular, CFR achieves 12.7% speedup over O3 on AMG while OpenTuner is only marginally (1.7%) better than O3. We notice that OpenTuner’s performance benefit increases very slow after tens of test iterations.

2. COBAYN performs similar to OpenTuner and is also inferior to FuncyTuner CFR. Specifically, COBAYN’s static and hybrid models perform 4.6% and 2.1% (geometric mean) better than the O3 baseline, respectively, while COBAYN’s dynamic model is worse than the O3 baseline. In contrast, FuncyTuner CFR improves performance by 9.4% beyond the O3 baseline. The performance of COBAYN’s static model is consistent with the findings in the work that proposed by COBAYN [Ash16] and other previous research [Aga06; Cav07; Fur15]. These related works also
show that a machine learning-based approach is able to reduce search overhead but does not perform better than traditional random search (*Random* in our paper) when the sample size is sufficiently large, e.g., 1000 samples. The poor performance of *COBAYN*’s dynamic and hybrid models may be attributed to limited dynamic features, since *MICA* [HE07] only works with serial code while our target benchmarks are parallel. We also want to point out that FuncyTuner has little overhead to be ported onto different machines, while the re-training overhead can be prohibitively high for *COBAYN*. As an example, extracting dynamic features for cBench takes around 2 days on a 16-node Broadwell cluster.

3. *PGO* results in only minor performance improvements relative to O3 and is not comparable to FuncyTuner *CFR*. While *PGO* is 1.8% better than O3 for AMG, it shows little improvement on six other programs. In fact, *PGO* instrumentation runs fail for LULESH and Optewe.

In brief, FuncyTuner *CFR* delivers significantly better performance than state-of-the-art techniques on our modern scientific simulation codes. It only relies on Caliper light-weight source-code level instrumentation [Boe16], entailing much less engineering complexities than others, especially *COBAYN* and *PGO*. Such simplicity is extremely noticeable when one considers the fact that Intel compilers are industry-quality production compilers and have been tuned for several decades, and *COBAYN* depends on a variety of large tools, such as Milepost GCC [Fur11] and Mica. Nevertheless, their performance is inferior to FuncyTuner *CFR* and their robustness is limited.

### 2.4.3 Impact Of Different Inputs

The over-arching goal of our work is to auto-tune large scientific simulation codes with a given input. Results in Sec. 2.4.1 and Sec. 2.4.2 use the same input as both tuning and test inputs. These inputs capture the typical sizes of application work sets in practice, hence, their performance benefits can generalize to other inputs, e.g., for inputs with the same work-set size but different simulation time-steps. This is shown in Fig. 2.9 for Cloverleaf on Broadwell by varying the number of time-steps as part of the input.

Inputs with different work-set sizes are addressed as follows. We experimented on Broadwell with two sets of inputs that have different input sizes from those in Table 2.2. For 351.bwaves, 362.fma3d, and 363.swim, we use “test” and “ref” as their small and large inputs, respectively. For LULESH, AMG, Cloverleaf, Optewe, their small input sizes are 180, 20, 1000, 384, respectively, while their large input sizes are 250, 30, 4000, 768, respectively.

As shown in Fig. 3.8, we observe little sensitivity for our benchmark applications on their small and large inputs, except that for 351.swim FuncyTuner *CFR* does not perform as well as the other three approaches for its small input. Nonetheless, *CFR* for 351.swim is still 20.6% better than *PGO* and the O3 baseline. We attribute such performance to the fact that the “test” input is so small that each time-step takes less than .01 seconds, which significantly differs from the performance profile of its tuning input in just this one case. FuncyTuner *CFR* achieves 5.5%, 9.5% and 10.7% (all in geometric mean) better performance than *OpenTuner*, *COBAYN*, and *PGO* on large input,
Figure 2.8 CFR shows little performance sensitivity on small and large inputs with geometric mean of speedup relative to the O3 baseline 12.3% and 10.7% respectively.

Figure 2.9 For Cloverleaf on Broadwell, FuncyTuner CFR provides a stable performance benefit than all others while scaling from 100 to 800 time-steps.
respectively. It is notable that the speedup of AMG under CFR over the O3 baseline is 22% while the benefit of other techniques is marginal.

The capability of generalizing performance benefits on tuning inputs to test inputs justifies the tuning overhead of all approaches, which is about 1.5 days for Random/G, 2 days for OpenTuner, 3 days for CFR and 1 week for COBAYN, for each benchmark. Specifically, for CFR's target HPC applications, the overhead is amortized in repetitive production runs. Moreover, the tuning overhead may be dramatically reduced via various techniques [Cav07; Pop16] or by exploiting program-specific CFR convergence trends, i.e., CFR finds the best code variant in tens or several hundreds of evaluations.

2.4.4 Deep Dive: Cloverleaf On Broadwell

Broadly speaking, CFR utilizes per-loop runtime information to direct the fine-grained search towards better performing CVs. Insights about the performance characteristics of different algorithms compared in this paper can be useful for further improving program performance. Specifically, we want to answer the following two questions:

- Q1: Why does G.realized often introduce slowdowns while expected to have performance improvements?
- Q2: Why does CFR perform better than others?

2.4.4.1 Case Study Design

To understand why FuncyTuner CFR performs the best, we selected Cloverleaf to conduct an in-depth case study on Intel Broadwell with two considerations. First, its non-loop code is written in Fortran, and hot loops are in C. This may provide more challenges and opportunities for ICC’s inter-procedural optimizations (IPO). Second, its hot loops have neither complex control flows nor deep loop nests and thus provide more opportunities for ICC to perform optimizations related to loop unrolling and vectorization, which often have a significant impact on program performance and are friendly for manual inspection of generated code to understand how they have been performed. Five hot loops of Cloverleaf are selected since they have comparatively high per-loop runtime ratios (see Table 2.6, others are less than 3.0%) and were found to produce large performance differences (see Fig. 2.10) across different tuning techniques.

To identify performance-critical flags for the best CVs, we design an iterative greedy algorithm to eliminate the flags that have low impact on the program runtime. Each iteration, the algorithm tries to remove one flag for a given loop’s CV (focused CV) while keeping all other CVs intact. If excluding a flag from focused CV does not degrade program performance, the flag is removed; otherwise, it is kept for the current iteration. This process is performed iteratively until no more flags of focused CV can be eliminated. We consider the remaining flags in focused CV as the critical ones for the given loop. Note that we only consider the static COBAYN model, because it is superior to its dynamic and hybrid counterparts for Cloverleaf.
2.4.4.2 Observations

Fig. 2.10 shows the per-loop performance results for the five Cloverleaf hot loops on Broadwell. After greedy elimination, Random, COBAYN and OpenTuner retain \texttt{-qopt-streaming-stores=always -no-ansi-alias -ipo -xCORE-AVX2} as their critical flags; FuncyTuner CFR retains \texttt{-no-vec} for \texttt{dt} and \texttt{mom9} but no special flags for the other three loops; G.realized also has no special flags for mom9. Table 2.6 reveals which critical optimizations, such as loop unrolling and vectorization, are exploited by different algorithms in this case.

We make the following observations:

1. Vectorization is not always profitable. First, \texttt{cell3} and \texttt{cell7} experience a 27.7% and 13.6% slowdown, respectively, when 256-bit vectorization is performed by Random. Other algorithms have similar performance benefits, yet they do not vectorize. O3 uses 128-bit SIMD (single instruction multiple data) instructions. Second, even though Random achieves a 34.8% speedup for \texttt{dt} with 256-bit vectorization, the performance is 12.8% worse than a scalar version. Inspection of assembly code (without access to ICC source code, this is the best that we can do) shows that there are many data permutations and mask operations to handle control flow divergence, which are known to degrade vectorization efficiency. Shorter loop trip counts caused by loop unrolling and OpenMP work sharing also contribute to the inferior performance of the vectorized loops.

2. FuncyTuner CFR has more informed freedom (compared to G, COBAYN and OpenTuner) to select non-conflicting CVs and prioritize performant CVs (compared to FR). For instance, CFR is able to select \texttt{-no-vec} for mom9 to avoid vectorization-induced slowdown.

3. G (see data point marked as G.realized) performs worse than other algorithms and invalidates the assumption that there are no inter-module dependencies. Note that G.realized and G.Independent have the same per-loop CVs. But G.Independent does not practically assemble an executable while G.realized does. The comparison between them demonstrates that there are interference among different modules. For example, G.realized vectorizes mom9 with
Table 2.6 Comparison of optimizations for 5 Cloverleaf kernels on Broadwell. S(Scalar): not vectorized; \{128,256\}: vectorized with \{128,256\}-bit SIMD; unroll[2,3]: unroll 2/3 times; IO: instruction reordering; IS: instruction selection; RS: register spilling.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>dt</th>
<th>cell3</th>
<th>cell7</th>
<th>mom9</th>
<th>acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>G.realized</td>
<td>6.3</td>
<td>2.9</td>
<td>3.5</td>
<td>3.5</td>
<td>4.2</td>
</tr>
<tr>
<td>G.Independent</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>256</td>
<td>256</td>
</tr>
<tr>
<td>O3 baseline</td>
<td>S, unroll2</td>
<td>S</td>
<td>S</td>
<td>128</td>
<td>S, unroll3</td>
</tr>
<tr>
<td>Random</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256, IS</td>
<td>256, IS</td>
</tr>
<tr>
<td>CFR</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S, IS</td>
<td>256</td>
</tr>
</tbody>
</table>

256-bit AVX2 instructions and further unrolls the vectorized loop twice while \(G.Independent\) does not.

4. Other optimizations, such as loop unrolling and instruction selection, also matter, e.g., FuncyTuner CFR and \(G.Independent\) both choose not to vectorize mom9, but their difference in instruction selection results in better performance for CFR.

In summary, we observe that such findings are difficult to derive manually for compiler writers while per-loop auto-tuning with FuncyTuner CFR is able to capitalize on them.

2.5 Related Work

The first order objective of compiler-based auto-tuning techniques is performance, while the second order objectives are code size, power draw, and energy consumption. To achieve peak performance and power efficiency for scientific applications on today's HPC machines, it is necessary to tune various compile-time and runtime configurations. On one side, configurations such as compile-time optimizations, runtime library parameters, and architectural configurations, may have a significant impact \([\text{Kis00}; \text{Che10}; \text{Gho16a}; \text{Tiw12}; \text{Bar16}; \text{Sou17}]\). On the other side, settings effective for one program/machine combination may not achieve peak performance on another, and it is both tedious and challenging to manually find the best configuration \([\text{Par04}]\). To attain the best performance, it is vital to automate the space exploration in an efficient manner. We divide prior work into the following two categories

1. Compiler flag selection techniques \([\text{Kis00}; \text{Cav07}; \text{PE08}; \text{Pop16}; \text{Che10}; \text{Kul04}; \text{Lin08}]\): given a set of compiler flags, the objective is to determine the combination that generates the most performant executable on a given architecture. Our work and many related papers \([\text{Che10}; \text{Pop16}; \text{PE08}]\) belong to this category. \([\text{PE08}; \text{Pop16}]\) also take a fine-grained per-region approach. They select the best code variant for each region in a greedy fashion without considering interactions among different code variants. This is is effective for their case studies but results
in worse performance than random search for OpenMP-based scientific applications. As the state-of-art search-based auto-tuning technique, OpenTuner [Ans14] coordinates many different search algorithms. Furthermore, to reduce the overhead of search-based approaches, researchers have also proposed schemes based on machine learning techniques [Aga06; Cav07; Ash16]. As the state-of-the-art, COBAYN [Ash16] infers compiler flags for a new program by representing them as static/dynamic features to a pre-trained Bayesian network. Our experimentation shows that FuncyTuner CFR outperforms both OpenTuner and COBAYN for Intel compilers while incurring similar cost.

2. Compiler phase ordering techniques [KC12; Kul04; JK13; Ash17]: given a set of compiler optimization passes, there are many valid orders, each of which may generate different runtime performance. Our work focuses on the Intel tool chain, which does not provide command-line flags to perform phase ordering.

2.6 Conclusion

In this work, we presented a fine-grained per-loop compiler flag selection framework, FuncyTuner, that combines program profiling and search space focusing algorithms to improve performance of parallelized scientific programs, in which different code regions/loops may be optimized with different flags. Our experimental evaluation shows that FuncyTuner’s Caliper-guided random search (CFR) effectively utilizes collected per-loop runtimes to focus the search on performant program compilation configurations. FuncyTuner achieves a 9.2% to 12.3% (geometric mean, up to 22% for AMG) performance improvement in comparison to O3 baseline, outperforms the state-of-art search-based technique OpenTuner, machine learning-based approach COBAYN and PGO of Intel compilers by 4.5% to 5.5%, 4.8% to 9.5%, and 10.7%, respectively. We also showed that greedily picking the per-loop best compiler flags often degrades program performance due to complex inter-module dependencies, and per-function random search without guidance of runtime information does not guarantee performance improvements.
CHAPTER 3

CODESEER: SELECT INPUT-DEPENDENT CODE VARIANTS VIA MACHINE LEARNING

In high performance computing (HPC), scientific simulation codes are executed repeatedly with different inputs. The peak performance of these programs heavily depends on various compiler optimizations, which are often selected agnostically on program input or may be selected with sensitivity to just a single input. When subsequently execution, often with different inputs, performance may suffer for all or all but the one input tested, and for the latter potentially even compared to the O3 baseline.

This work proposes a new auto-tuning framework, FuncyTuner, to assess and improve existing input-agnostic or single-input centric rigid application tuning methods. Aided by FuncyTuner, it is observed that modern HPC programs expose different types of input sensitivities, which present a significant challenge for prior work. To tackle this problem, FuncyTuner proceeds with several machine learning models to predict the best per-input code variant on-the-fly. Our evaluation shows that FuncyTuner incurs less than 0.01 second overhead, predicts the best code variant with a geometric mean precision 92% of the time and is capable of improving per-input peak performance to unprecedented levels.
3.1 Introduction

In high performance computing (HPC) environments, scientists tend to test their hypotheses by running large-scale simulation codes repeatedly with different inputs. These applications occupies computing resources for a significant amount of time consuming an equally significant amount of power time and again when executed. Even a 1% performance improvement could translate into significant energy savings in such HPC production runs [Hal15]. To extract program peak performance, various compiler-based auto-tuning technologies have been developed. These techniques are either search-based [Wan19; Ans14; PE08; Hal10; Yi12; Thi18] or machine learning-based [Ash16; Cav07; Nob16; JK13]. They generally are constrained to tuning with a single tuning inputs, which produces a single optimized executable evaluated on a small set of test inputs to report any performance improvements. As a result, their effectiveness may be limited by the tuning input and any performance tends to be localized to this input instead of generalizing to a larger set of inputs in production.

![Figure 3.1](image)

Figure 3.1 Normalized speedups of 1000 Kripke code variants for a singular, identical tuning input: No performance improvement for any under O3.

If an improper input is selected as the tuning input, these optimization techniques may fail to result in executables realizing any performance improvements whatsoever. Consider Fig. 3.1 with the relative performance (y-axis) normalized to O3 over 1,000 different code variants for Kripke [Kun15], a discrete-ordinates solver (see Section 3.3). The results indicate that auto-tuning Kripke with per-program random search [Wan19] generates 1,000 code variants with different compiler flags for Intel C++ compiler. Yet, none of them achieve any performance improvement over the O3 baseline with the selected tuning input. We call this **type-I input sensitivity** (tuning sensitivity): *The performance benefit of auto-tuning may depend on the selection of tuning inputs.* Addressing type-I sensitivity by tuning with many more tuning inputs greatly increases the overhead of auto-tuning techniques,
thus limiting their practicality.

![Graph showing speedups normalized to O3 for two code variants across 1,000 inputs.](image)

**Figure 3.2** Normalized speedups of two Kripke code variants across 1,000 inputs show 1) type-II input sensitivity: The performance benefit of the same code variant is sensitive to different inputs; and 2) type-III input sensitivity: The best performing code variants depend on sets of many inputs.

Fig. 3.2 shows the normalized speedups for two code variants of Kripke across 1,000 inputs. Code variant 1 (cv1) and code variant 2 (cv2) are generated by choosing different compiler flags of the Intel C++ compiler. The results indicate 1) type-II input sensitivity: The significance of a code variant’s performance benefit depends on the chosen test inputs. For instance, cv1 achieves a 49% speedup for input 1,000 but causes a \(\approx 20\%\) slowdown for input 1. Type-II sensitivity demonstrates that we carefully select test inputs to guide performance improvement in program auto-tuning in practice. 2) type-III input sensitivity: The best performing code variants may be differ for diverse test inputs. For example, between cv1 and cv2 always, neither outperforms the other. In fact, there are many cases where one performs more than 10% better than the other, e.g., cv1 is 45.7% better than cv2 for input 998, while cv2 is 28.4% better than cv1 for input 2. Type-III input sensitivity suggests that different code variants are necessary to realize the best per-input program performance. In particular, type-III input sensitivity demonstrates the fundamental limitations of prior auto-tuning techniques in that they only generate a single optimized executable: If a single code variant performs the best for just a subset of inputs, as shown in Fig. 3.2, then the optimized executable is sub-optimal and some performance is left on table.

To account for type-III input sensitivity, modern optimizing compilers, e.g., Intel compilers and the GNU Compiler Collection, feature built-in profile-guided optimization (PGO) \[Int18\]. PGO utilizes program runtime profiling information to generate optimized code for multiple characteristic input data sets. However, a recent study with FuncyTuner \[Wan19\] shows that PGO of Intel compilers rarely improves the performance of a set of modern scientific programs in their work. In addition, there is little practical guidance on how to select characteristic tuning data sets for PGO.
This work proposes an auto-tuning framework, CodeSeer, to evaluate the three types of program input sensitivities, to auto-tune with many inputs to address type-I sensitivity, and to alleviate type-III sensitivity for modern scientific applications. The key insight of CodeSeer is that storage on modern supercomputers is abundant and can be traded off for program performance with fast and accurate machine learning-based prediction models to select the best code variant on a per-input basis. We formulate type-III sensitivity as a classification problem. Given an unseen input, CodeSeer classifies it as one of several representative classes and chooses to execute the corresponding code variants. As shown in Fig. 3.3, CodeSeer proceeds with following steps:

1. It defines the training input datasets \( I \) (of size \( N \)) of a target program and prepares a set of code variants \( C \) (of size \( M \)) by randomly sampling the compiler flag space [Wan19]. \( N \) and \( M \) should be large to represent sufficient diversity in inputs and code variants.

2. It measures end-to-end runtimes for code variants \( C \) on inputs \( I \) to evaluate type-III sensitivity. If a code variant always performs best on all inputs, it will not be further processed; otherwise, it continues with the next step. The overhead to evaluate \( M \times N \) code variants and input combinations can be excessively high. Overhead reduction techniques are critical, as discussed later.

3. It selects several representative code variants according to certain criteria (see Sec. 3.2.4) and uses them as class labels. For each training input \( i \), a label \( c \) is assigned such that the corresponding code variant performs the best for \( i \).

4. It builds the classification models of CodeSeer with labeled training inputs from 3. These models predict the best code variant for unseen test inputs.

5. Accuracy and overhead of the prediction models from 4 are evaluated.

To the best of our knowledge, this work is first to investigate the three types of input sensitivities for modern scientific simulation codes and to propose an ensemble of practical techniques to transparently improve their performance in an input-sensitive manner. In particular, the work makes the following contributions:

- Three types of input sensitivities are evaluated for a set of modern scientific applications, and a case study for Kripke [Kun15] shows that programs with type-III sensitivity need novel tuning techniques to realize best performance.

- A solution to address type-III sensitivity is formulated as a classification problem. This problem of optimally selecting an arbitrary number of code variants to generate the training dataset (called input coverage plan generation) is proved to be NP-complete.
• A prototype machine learning-based framework, CodeSeer, to predict the per-input best code variants for programs with type-III sensitivity is created, and an ensemble of overhead reduction techniques is developed.
• CodeSeer’s models are demonstrated to be effective and efficient in predicting the per-input best code variants for the set of targeted modern scientific applications.

3.2 Tackling Type-III Sensitivity

In this section, a solution to handle the program’s type-III sensitivity first formulated to predict the best per-input code variant. Then, a formal definition of the challenge to select representative code variants is presented and shown to be NP-complete. Two algorithms are proposed to solve the problem in a computationally feasible manner with an analysis and comparison of time complexity.

3.2.1 Problem Formulation

Given a program $P$ with type-III sensitivity, suppose there are $R$ code variants that perform the best, i.e., at least one of $R$ code variants performs the best for any given input $i_n$ of an input dataset $I$ (of size $N$, $1 \leq n \leq N$). We call these $R$ code variants “$P$’s representative code variants” ($RC$ for short) and denote the best performing code variant as $c_l$ ($1 \leq l \leq R$) for $i_n$. Then, we assign the code label $l$ to $i_n$. By labeling each input $i$, we obtain a training data set to build a model that maps (classifies) any unseen input $i_t$ ($t \notin I$) to a code label $l_t$. The code variant corresponding to the label $l_t$ is selected for executing $P$ on $i_t$. To build the classification model, the challenge is to generate the labeled training dataset. First, $RC$ has to be derived with a sufficiently large set of code variants $C$ (of size $M$) and $N$ inputs. If an average program run takes 10 seconds, with $M = 1000$ and $N = 1000$, the evaluation would take about 115 days of machine time. Such high overhead must be addressed to make it practical. Second, the selection of an optimal $RC$ set is at least as hard as NP-complete (discussed next in Sec. 3.2.2), i.e., an efficient, sub-optimal algorithm is required for computational feasibility.

3.2.2 Representative Code Variants Selection

The evaluation of $M$ code variants on $N$ inputs can be presented as an $M \times N$ matrix ($T_{M \times N}$) of run-times. The selection of $RC$ (i.e., the Input Coverage Plan in step 3) encompasses the identification of $R$ ($1 \leq R \leq M$) code variants (corresponding to $R$ rows in $T_{M \times N}$) such that one of $RC$’s elements achieves the best performance for at least one of the $N$ inputs. In principle, an input may have its own unique performant code variant. However, in practice, there tend to be a limited number of code variants vs. an infinite number of inputs. Therefore, we assume that $M$ is smaller than $N$. Moreover, a smaller $R$ is more favorable due to less storage overhead and simpler classification models that are generally more accurate with the same amount of training data. Therefore, it is beneficial to minimize $R$ for a given $T$ while maximizing the geometric mean of speedups across all inputs.
Formally, given a $M \times N$ ($1 < M, N$) matrix $T$ of non-negative numbers, let a row set $S^R$ ($1 \leq R \leq M$) be a set of $R$ rows of $T$, i.e., $S^R = \{ r_1, r_2, ..., r_R \}$ where $\forall s \neq t \land 1 \leq s, t \leq R \land r_s \neq r_t$, where $1 \leq r_s \leq M$. For simplicity, it is required that $\forall s > t \land r_s > r_t$. Given $T$ and $S^R$, a new matrix $T'$ can be constructed by copying row $r_i$ in $T$ into row $i$ of $T'$, where $r_i \in S^R$ and $1 \leq i \leq R$. Therefore, $T'$ is a $R \times N$ sub-matrix of $T$. $T'$ is called the row-set $S^R$-projected matrix of $T$ and, when necessary, $T'$ is denoted as $T(S^R)$ to stress how $T'$ is constructed. Furthermore, the row-sum of a matrix $T$ is defined as $P(T) = \sum_{j=1}^{N} \max_{1 \leq i \leq R} T_{i,j}$. Then, $P(T') = P(T(S^R)) = \sum_{j=1}^{N} \max_{1 \leq i \leq R} T'_{i,j}$ and there are in total $C_M^R = \binom{R}{M}$ ways to construct $S^R$ and $T'$ for a given $R$, each denoted as $S^R_i$ ($1 \leq i \leq C_M^R$). We want to minimize $R$ given by the following objective function (3.1):

$$\arg\min_{R} (\arg\max_{S^R_i} P(T(S^R_i))) \text{ for } 1 \leq R \leq M, 1 \leq i \leq C_M^R \quad (3.1)$$

Let $S^R_g = \arg\max_{S^R_i} P(T(S^R_i))$. Then, minimizing $R$ in (3.1) is straight-forward if the optimization for row-set $S^R_g$ can be solved for a given $R$. It can be shown that the graph independent set problem [Kin19] can be polynomially reduced to a special case of this row-sum optimization problem's decision form: Given a matrix $T$ with $R$ rows satisfying the aforementioned constraints, is there a row-set $S^R$ such that $P(T(S^R))$ is at least as large as an arbitrary (but given) number $C$?

Given a graph $G = (V, E)$ with vertices $V$ and edges $E$, and a constant $k$, a $k$-independent set is a set $V^k$ of $k$ vertices $\{v_{u_1}, v_{u_2}, ..., v_{u_k}\}$ s.t. $(v_{u_i}, v_{u_j}) \notin E \land i \neq j \land 1 \leq i, j \leq k$. Let us construct a special case matrix $T_G$ with elements $\theta_{j,i}$ that has one row for each vertex and in total $|V|$ rows and $|V| + |E|$ columns. Each of the first $|E|$ columns corresponds to exactly one edge $e_t = (v_s, v_t)$, where $1 \leq i \leq |E| \land s \neq t \land 1 \leq s, t \leq |V|$ such that elements $\theta_{s,i} = \theta_{t,i} = 1 \land \theta_{s,s+|E|} = |E| - \sum_{i=1}^{|E|} \theta_{s,i} \land \theta_{j,i} = 0$ for $j \notin \{s, t\}$. As a result, the summation of all elements in any row of $T_G$ must be equal to $|E|$ (property p1).

$$\begin{pmatrix}
    v_1 & \theta_{11} & \theta_{12} & \ldots & \theta_{1|E|} & |E| - \phi_1 & 0 & 0 & 0 \\
    v_2 & \theta_{21} & \theta_{22} & \ldots & \theta_{2|E|} & 0 & |E| - \phi_2 & 0 & 0 \\
    \vdots & \vdots & \vdots & \ldots & \vdots & \vdots & \vdots & \vdots & \vdots \\
    v_{|V|} & \theta_{|V|1} & \theta_{|V|2} & \ldots & \theta_{|V||E|} & 0 & 0 & 0 & |E| - \phi_{|V|}
\end{pmatrix}$$

For a $k$-row set $S^k$, let the number of ones in the same column $c$ be $m_c$. Further, $\forall 1 \leq c \leq |E| \land m_c \in \{0, 1, 2\}$. We can prove that $P(T_G(S^k)) = k|E| \iff \forall 1 \leq s \leq |E| \land m_s \leq 1$ (property p2).

Proof: (1) $\iff$: By p1 and definition of $P(T_G(S^k))$, $P(T_G(S^k)) = k|E|$. (2) $\iff$ by contradiction: Given $P(T_G(S^k)) = k|E|$, assume $\exists m_i = 2$. By definition of $P(T_G(S^k))$, for column $i$, only one of the two ones contributes to $P(T_G(S^k))$ and thus $P(T_G(S^k)) < k|E|$, which contradicts $P(T_G(S^k)) = k|E|$. Therefore, the assumption is false. Therefore, p2 holds. □

A solution of $k$-row sum with $P(T_G(S^k)) = k|E|$ is also a solution for $G$’s $k$-vertex independent
set by $p_2$. Hence, we reduce the NP-complete independent set problem to a special case of the row-sum problem via the construction of $T'_G$. Furthermore, one can verify if a given solution is at least as large as any given number $C$ in $O(k \times N)$ (polynomial) steps, so the row-sum problem in its decision form is also in NP. Therefore, it is at least an NP-complete problem.

To summarize, a special case of the decision form of our row-sum optimization problem is NP-complete. Hence, the original problem must be at least as hard. For all practical purposes, the objective is to find an efficient algorithm to solve this problem.

### 3.2.3 Algorithm Design

For all $1 \leq R \leq M$, the optimal solution $S^R_g$ to Equation (3.1) can be obtained by brute-force (BF) enumeration with time complexity $O(R \times N \times C^R_M)$, since there are $C^R_M$ ways to select $R$ rows to construct $T'$, and each $T'$ needs $R - 1$ comparisons to find $N$ column-wise maximum elements with $O(N)$ time complexity to calculate their sum. In practice, when $R$, $M$ and $N$ are sufficiently small, it is computationally feasible to solve such a problem with brute force.

However, to utilize CodeSeer to generate an input coverage plan with objection function (3.1) that retains a minimal number of code variants while achieving maximal performance benefits, we prefer to begin with a small $R$ but large $N$ and $M$ values. In particular, a large $M$ represents more diversity of code variants while a large $N$ means these code variants are evaluated thoroughly; conversely, a small $R$ indicates fewer class labels for CodeSeer models and less difficulty to predict. Therefore, $M, N \gg R$ for our purpose, such that brute force may be very inefficient. This provides the motivation to design a greedy approximation algorithm (see Algorithm 5) for generating a fast input coverage plan.

The intuition of such an algorithm is to select code variants that achieve the best performance on the most inputs (largest coverage) first. The time complexity of such a selection is $O(M \times N)$ since counting the number of inputs for which a code variant achieves the best performance is iterated over the entire matrix $T_{M \times N}$ ($N$ speedup evaluations for each of $M$ code variants) once. Besides, we rely on the following guideline to evaluate the performance of the approximation:

Suppose $R'$ is the solution to objective function (3.1). Then $S^R_g$ is the optimal row set and $P' = P(T') = P(T(S^R_g))$ is the maximum row-sum. It follows that

$$\forall 1 \leq R_1 < R' < R_2 \leq M \quad P(T(S^R_{R_1})) < P(T(S^R_{R_2})) = P(T(S^R_g)) \quad (3.2)$$

Knowing $P'$ is critical for any approximation algorithms to assess how much performance loss their solutions may incur due to the approximation. Inequation (3.2) indicates that $P'$ may be calculated by considering all $M$ rows and the time complexity to calculate $S^M_g$ is $O(M \times N)$, because it is equivalent to selecting all $M$ rows with $O(1)$ time complexity, identifying all $N$ maximum column elements in $O(M \times N)$ time, and summing them together in $O(N)$ time.
Algorithm 5: CodeSeer Greedy Input Coverage Plan and Data Labeling

**Input**: matrix $T_{M \times N}$ of $M$ code variants for $N$ inputs, $R = \#$ selected code variants

**Output**: row set $S^R$, labels $L_N$

1. $\text{counter}[1...M] = 0$, $L[1...N] = -1$, $S[1...R] = \emptyset$
2. for $i = 1; i < N; i++$
   3. $k = \text{argmin}_k \{T[k][i]\mid 1 \leq k \leq M\}$
   4. $\text{counter}[k]++$
5. //generate coverage plan in row-set $S^R$
6. for $i = 1; i < R; i++$
7. $k = \text{argmax}_k \{\text{counter}[k]\mid 1 \leq k \leq M\}$
8. $S[i] = k$
9. //data labeling in $L_N$
10. for $i = 1; i < N; i++$
11. $L[i] = \text{argmin}_k \{T[k][i]\mid k \in S^R = S[1...R]\}$
12. return $S^R$, $L_N$

3.2.4 Data Labeling

Since we are using classification models to predict the best per-input code variants, a labeled dataset is necessary. Note that in Algorithm 5, each row number in $S^R$ is an identifier of a code variant in $RC$. Therefore, after $S^R$ is calculated for $T$, data labeling (lines 9—11 in Algorithm 5) is performed such that each of the $N$ inputs is labeled with the identifier of its corresponding best code variant.

3.3 Experimental Design

This section first describes the benchmark applications and our experimental platform. It then discusses how training datasets are prepared for the evaluation of type-III sensitivity. Also presented are the results of type-III sensitivity followed by an analysis of the root causes via a case study on Kripke [Kun15].

3.3.1 Benchmarks and Code Variants

Our benchmark applications are ten modern HPC codes for scientific simulation (see Table 3.1): Kripke [Kun15], LULESH [Kar13], Cloverleaf [UM18], Optewe [Sou17], miniFE [UM19], 351.bwaves, 357.bt331, 360.ilbdc, 363.swim and 370.mgrid331. The latter five applications are from the SPEC OMP 2012 suite [Mül12]. These benchmarks are written in different languages, exploiting multi-core parallelism of modern HPC architectures via OpenMP pragmas. They are designed to resemble the properties of large-scale applications with the same core algorithms, and are widely used in program performance/energy studies.

Code variants are generated with different compilation flags of Intel compiler tool chain (version 17.0.4) to achieve high performance. As suggested by Intel [Cor18], the original build systems are modified to use Intel’s compilers, linker (xild) and archiver (xiar). The compilation flags are selected and combined with a classic random algorithm, similar to prior work [Che10; Wan19]. We also enforce strict floating-point reproducibility by always including `-fp-model source` in the
Table 3.1 List of benchmarks (LOC: lines of source code)

<table>
<thead>
<tr>
<th>Name</th>
<th>Language</th>
<th>LOC</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kripke</td>
<td>C++</td>
<td>7.2k</td>
<td>Discrete-ordinates solver</td>
</tr>
<tr>
<td>Optewe</td>
<td>C++</td>
<td>2.7k</td>
<td>Seismic wave simulation</td>
</tr>
<tr>
<td>LULESH</td>
<td>C++</td>
<td>7.2k</td>
<td>Hydrodynamics</td>
</tr>
<tr>
<td>Cloverleaf (CL)</td>
<td>C, Fortran</td>
<td>14.5k</td>
<td>Hydrodynamics</td>
</tr>
<tr>
<td>miniFE</td>
<td>C++</td>
<td>24.5k</td>
<td>Finite Element</td>
</tr>
<tr>
<td>351.bwaves</td>
<td>Fortran</td>
<td>1.2k</td>
<td>Computational fluid dynamics</td>
</tr>
<tr>
<td>357.bt331</td>
<td>Fortran</td>
<td>5.4k</td>
<td>Computational fluid dynamics</td>
</tr>
<tr>
<td>360.ilbdc</td>
<td>Fortran</td>
<td>1.3k</td>
<td>Lattice Boltzmann</td>
</tr>
<tr>
<td>363.swim</td>
<td>Fortran</td>
<td>0.5k</td>
<td>Weather prediction</td>
</tr>
<tr>
<td>370.mgrid331</td>
<td>Fortran</td>
<td>1.7k</td>
<td>multigrid solver</td>
</tr>
</tbody>
</table>

compilation flags. The flags for baseline code variants are `-O3 -qopenmp -fp-model source`. End-to-end runtimes are measured on a HPC cluster with 16 computation nodes featuring two Intel Broadwell Xeon E5-2620 v4@2.1GHz processor with 16 cores (hyper-threading on) each in a two-node NUMA configuration of 64GB DRAM, running CentOS Linux release 7.5.1804. OpenMP thread placement is set to \texttt{granularity= fine, proclist=[0-15], explicit}.

We used 100 code variants in all experiments to make the runtime overhead for our benchmarks feasible, unless otherwise specified. Fig. 3.4 shows that the normalized speedup distributions for 1000 (top) and 100 (bottom) Kripke code variants are similar. This indicates that using 100 code variants may be sufficient to capture most of the performance diversities for a given input (as similar patterns are seen for many other inputs for Kripke). Prior work [Ash16; Aga06] also suggests that the best performance of code variants for any specific tuning input is usually found within tens to hundreds of iterations by random search within compilation flag space.

Figure 3.4 Normalized speedup distributions for 1000 (top) and 100 (bottom) code variants on a single Kripke input indicating similar results: code variants in excess of the top 100 contribute little to performance (limited diversity).
3.3.2 Prepare Input Data Sets

Inputs for these benchmarks can be set either input files or command line options. To generate a large set of inputs, we consulted the respective user guide of programs to select tunable input variables. Input variables tend to be continuous integer value ranges expressed as variable_name [min_value, max_value, step_size] or [formula (constraint)]. Variables and their ranges are independent of one another unless otherwise mentioned. Kripke [Kun15] inputs are groups [2^k (0 ≤ k ≤ 6)], gset [2^k (0 ≤ k ≤ 5)], legendre [0, 9, 1], quad [32k (1 ≤ k ≤ 6)], dset [8k (1 ≤ k ≤ 8)], all three sub-dimensions of zones [16k (1 ≤ k ≤ 4)], all three sub-dimensions zset [2^k (0 ≤ k ≤ 8)], nest (G Z D). Moreover, groups and quad should divide gset and dset, respectively. LULESH [Kar13] inputs are size (30, 420, 30), i (1, 100, 10), r (5, 50, 5), c (5, 60, 5), b (0, 9, 1). Cloverleaf [UM18] inputs are x and y (100, 18000, 200), end_step (10, 100, 10) Optewe [Sou17] inputs are x, y and z (32, 1000, 32), nt (1, 40, 3), s (1, 3, 1). miniFE [UM19] inputs are nx, ny, nz (64, 328, 4). 351.bwaves inputs are nx, ny, nz (64, 328, 4), method (0, 1, 1), configuration (0, 1, 1), time steps (5, 40, 5). 357.bt331 inputs are x, y, z (64, 408, 4), steps (2, 20, 2). 360.ilbdc inputs are x, y, z (32, 1000, 5), GEONME is #channel or #packing or #synpor. 363.swim inputs are line1 (2, 20, 2), line6 (30, 1000, 50), line8, line9 (512, 7701, 256). 370.mgrid331 inputs are x, y, z (64, 1200, 14), steps (10, 100, 10).

The input space created by ranges of all input variables are randomly sampled for 10^4 inputs without replacement. These inputs are first evaluated with the target program’s O3 baseline compilation for 5 consecutive runs. Then, 4,000 of them are selected according to two criteria: (1) The end-to-end runtimes need to be within one to 200 seconds. Inputs with too short a runtime do not contribute significantly to performance (geometric mean) and are prone to measurement noise, while those with too long runtimes are prohibitively expensive for the experiments due to the number of samples taken, i.e., runs executed. (2) To reduce the overhead for evaluation, we restrict program inputs to those with less than 1% variations under O3 so that one run is sufficiently reliable to evaluate each code variant on these inputs.

The pre-selected 4,000 inputs are divided into three subsets, D_1, D_2, and D_3, by random sampling without replacement. They are used for different steps in Fig. 3.3. D_1 includes 1,000 inputs and is used in step ② to evaluate program input sensitivity. The evaluation is done on 100 program code variants. If a program exposes type-III sensitivity, CodeSeer uses D_1 to generate an input coverage plan that identifies a subset of 100 code variants (denoted as CV_T) as the prediction targets for data labeling in step ③. Another 2,800 inputs are evaluated on CV_T to generate more labeled inputs. Therefore, D_2 includes D_1 and these 2800 inputs. After data labeling, D_2 is used as the training dataset in step ④ to build six machine learning models. D_3 contains the remaining 200 inputs and represents the testing dataset in ⑤ for evaluation.

3.3.3 Build Machine Learning Models

For each program that has type-III sensitivity, we prototype six classification models in CodeSeer with Scikit-learn [com19a] since this model collection covers different trade-offs between prediction...
precision, training overhead, and online prediction speed. These models include support vector machine (SVM), decision tree (DT), random forest (RF), logistic regression (LR), neural network (NN) and K-nearest neighbors (KNN). For a target program, we first applied feature scaling to each dimension $x$ of the training inputs ($D_2$) by $(x - u)/s$, where $u$ is the mean and $s$ is the standard deviation of $x$. We then employ 5-fold cross-validation with $D_2$ for each model to avoid over-fitting and use randomized grid search to tune their hyper-parameters with weighted F1-score as the guiding performance metric. We evaluate these models on $D_3$ and report their weighted precision.

### 3.4 Results on Input Sensitivity

#### 3.4.1 Type-I Sensitivity

![Figure 3.5](image)

*Figure 3.5* All benchmarks expose type-I sensitivity: Performance benefits of a tuned program depends on the tuning input used.

For any input $i$ ($1 \leq i \leq 1000$) in $D_1$, there is one code variant $C_{i}^{opt}$ ($1 \leq opt \leq 100$) that achieves the best speedup relative to the $O3$ baseline. We calculated the geometric mean speedups ($S_i$) over $O3$ for $C_{i}^{opt}$ across all inputs in $D_1$ and show their probability densities in Fig. 3.5. We make the following observations:

1. All applications expose type-I sensitivity since the speedups differ significantly for input $i$ in $D_1$. Since $C_{i}^{opt}$ would be the final optimized executable for any traditional approach [Wan19; PE06; Ans14; Ash16], Fig. 3.5 demonstrates that their tuning input must be chosen carefully. If a bad input is chosen, they will not be able to find any performance improvement.

2. Applications have very different type-I sensitivity. For example, all inputs for 357.bt always lead to performance improvement while any other benchmark requires more effort to identify the best tuning input. In contrast, CodeSeer considers input sensitivity in its design pitfall by evaluating more tuning inputs, thereby avoiding a single-input bias.
3.4.2 Type-II Sensitivity

![Figure 3.6](image)

**Figure 3.6** All benchmarks expose type-II sensitivity: Performance benefits are different for inputs in $D_1$ with $CV_{opt}$; this achieves the best geometric mean speedup on $D_1$.

For a target program, let $CV_{opt}$ be the code variant that achieves the highest geometric mean speedups across all inputs in $D_1$. The distribution of speedups achieved by $CV_{opt}$ for inputs in $D_1$ is shown in Fig. 3.6. We observe that all applications expose type-II sensitivity: $CV_{opt}$ achieves very different speedups for inputs in $D_1$. Prior work [PE06; Wan19; Ans14; Ash16] evaluated performance of optimized executables on a limited number of testing inputs resulting in performance benefits that may not generalize to other inputs. In the worst case, their optimized executables may lead to degradation, sometimes as significant as miniFE and 360.ilbdc with a minimum of 0.72 and 0.82, respectively.

![Figure 3.7](image)

**Figure 3.7** Cumulative number of inputs in $D_1$ with different normalized speedup gaps between $CV_{opt}$ and $CV_{opt}'$.
3.4.3 Type-III Sensitivity

Type-II sensitivity suggests that $C_{Vopt}$ may not achieve the best per-input speedups. In fact, the cumulative number of inputs with different speedup gaps between the $C_{Vopt}$ and the best code variant ($C_{V^i_{opt}}$) for input $i$ are shown in Fig. 3.7. We observe that $C_{Vopt}$ is not always the best code variant per input (type-III sensitivity): For Cloverleaf, $C_{Vopt}$ is the best code variant per input, but Kripke under $C_{Vopt}$ only achieves the best per-input speedups for about 52% of the inputs in $D_1$. All other applications fall in between these two extremes. In fact, $C_{Vopt}$ for Optewe, 370.mg, miniFE, 363.swim and 357.bt differs by less than 2% (performance loss) from $C_{V^i_{opt}}$ for around 8%, 1%, 10%, 1%, 3% and 5% of the inputs in $D_1$, respectively; $C_{Vopt}$ for Lulesh, 351.bwaves and 360.ilbdc is over 3% inferior to more than 10% inputs, respectively.

Results for Kripke (Fig. 3.2), 351.bwaves, 360.ilbdc and Lulesh (Fig. 3.8) show that there are two code variants that perform best for different subsets of $D_1$. These two code variants happen to be identified by our greedy algorithm (see Algorithm. 5), while the problem in general is at least NP-complete. We observe that type-III sensitivity introduces an inherent limitation for prior auto-tuning approaches that yield only one optimized executable for each target program, e.g., 10% of $D_1$ experience significant degradation relative to their $O3$ baseline while the performance of more than 10% of inputs for Kripke $C_{Vopt}$ results in a 10% to 52.8% gap relative to peak performance. This reinforces the need for low-overhead high-precision predictive models.

3.4.4 Discussion on Type-III Sensitivity

Type-III sensitivity may result from two scenarios. First, many compiler optimizations, e.g., loop unrolling and vectorization, do not always improve performance. For this reason, they usually rely on heuristic cost models to decide whether or not to perform the optimizations at compile time, and/or subsequently decide whether or not to take the optimized code paths at runtime. However, the cost model may mispredict. For the two code variants ($C_{V_1}, C_{V_2}$) of Kripke in Fig. 3.2, we profiled a hot loop (LTimes) with Caliper [Boe16] and Likwid [Hag10] for two inputs, $i_1$ and $i_2$. We observed that:

1. $C_{V_1}$ is 41% faster than $C_{V_2}$ on $i_1$, but 32% slower on $i_2$;
2. LTimes is vectorized in $C_{V_1}$ but unvectorized in $C_{V_2}$.

These observations demonstrate that Intel’s C compiler has a cost model that fails for these inputs. Second, runtime ratios of hot loops may change due to different input sizes. Even if the relative loop speedups in different code variants stay the same, such changes may favor another code variant for a new input. In general, our experiences suggest that it is challenging to understand the performance discrepancies between code variants on a per-input basis: Differences may be subsumed by cache behavior or memory bandwidth utilization. In contrast to past work, CodeSeer utilizes machine learning models to capture the complex relationship between program inputs and code variants resulting from different optimizations.
Figure 3.8 360.ilbdc, 351.bwaves, and Lulesh expose type-III sensitivity: CV1 and CV2 are the two code variants selected by CodeSeer’s greedy coverage plan algorithm.
3.5 Model-based Tuning Results

3.5.1 Precise Prediction

3.5.1.1 Deciding Target Code Variants

CodeSeer relies on its greedy algorithm (see Algorithm 5) to identify $R$ ($1 \leq R \leq 100$) code variants as the input coverage plan for $D_1$. Fig. 3.9 shows that the benefit of a model with more target classes, assuming perfect prediction, diminishes very fast and that a model with two code variants is able to harvest most of the performance potential. Moreover, since there is a one-to-one mapping from these $R$ code variants to $R$ target classes, when the training set size is fixed, having more classes means that there will be fewer samples in each class and may produce less precise models. These observations were the motivation to use only two target classes ($C_{VT}$) for CodeSeer models in our work.

![Figure 3.9](image)

**Figure 3.9** Ratios (for Kripke only) to the per-input best speedup: higher is better. The per-input best speedup is the maximum speedup achieved by 100 code variants for inputs in $D_1$; top-$K$ (1, 2, 5, 10) indicates the best per-input speedup achieved by $K$ code variants identified by CodeSeer’s greedy algorithm.

3.5.1.2 Precision Results

Inputs in $D_2$ (excluding those in $D_1$), are evaluated on the two code variants in $D_1$’s coverage plan. After data labeling (see Algorithm 5), $D_2$ is used for training CodeSeer’s models and $D_3$ (see 3.3.2) is used as the test set. $D_3$ approximates the class imbalance in $D_2$ and is an unbalanced test set. In contrast, a balanced test set has the same number of inputs in each target class, which is useful to evaluate how well CodeSeer’s models can generalize to other datasets. To this end, we upsampled the input space for a few more inputs to enlarge the smaller class up to 100 inputs, and conversely downsampled $D_3$ to reduce the larger class down to 100 inputs.
Fig. 3.10 shows CodeSeer's prediction results. We make the following observations:

1. Random forest (RF) achieves the best geometric mean precision, 92% and 90% on the unbalanced and balanced datasets, respectively. The high precision on the balanced dataset shows that RF is able to generalize well.

2. Precision varies for different applications. 351.bwaves experiences 97% precision on both test sets; precisions for Kripke and 360.ilbdc are around 95%; Lulesh's precision is about 80%, which results from mispredicting the best code variant in absolute terms, but only to choose another code variant that is only 1% less performant for these inputs, i.e., results are nearly indistinguishable. If precision was redefined to allow variations of up to 1%, prediction precision for RF would increase to 95% in the figure.

3.5.1.3 Performance Benefits

As mentioned in Section 3.4.1, CodeSeer first handles type-I sensitivity by tuning with many more inputs and identifies \( C V_{opt} \) with its greedy algorithm. CodeSeer's high-precision prediction model, random forest (RF), then improves the per-input performance to an unprecedented level. This is illustrated by Fig. 3.11, which plots the difference in performance between RF and the respective
code variants (balanced/unbalanced CVs) for all inputs. RF outperforms any CV a positive gap, which is the case for almost all inputs, sometimes by as much as 43%. (Speedup gaps are sorted independently in monotonically increasing order for each data set.) Fig. 3.12 shows that the ensemble RF model provides the best result in terms of its speedup factor over O3, up to a 45% improvement with benefits for 93% of the input. In contrast, among the code variants $C V_1$ and $C V_2$, one may provide higher performance for some input while the inverse is the case for other input. Again data sets are sorted independently. For unsorted (same input) plots, scatter graphs would results (similar to Fig. 3.1), i.e., the RF model has to compensate for complex input sensitivities Overall, we observed that:

(1) CodeSeer’s correct RF predictions translate into a performance benefit for all applications on both unbalanced and balanced test sets exceeding those of $C V_{opt}$. For Kripke, around 20% and 25% of the inputs benefit from more than 5% more speedups (relative to the O3 baseline) compared to $C V_1$ and $C V_2$, respectively.

(2) CodeSeer’s RF mispredictions have limited negative impacts, since the rate is low (Kripke, 351.bwaves) or the speedup differences for mispredicted inputs are small (mostly within 1% for Lulesh). For 360.ilbdc, the training set is highly unbalanced against $C V_2$, so the misprediction rate for $C V_2$ is higher than $C V_1$ on the balanced test set.

3.5.2 Overhead Analysis

On our 16 node Broadwell cluster, it takes about one week per program to evaluate their input sensitivity with several optimizations, e.g., using 100 code variants (see Sec. 3.3.1). Evaluation of 3,000 more inputs in $C V_{opt}$ takes less than one day, which may be further reduced. In Fig. 3.14, we observed that random forest converges with about 1,000 training inputs for an unbalanced test set (with similar results for a balanced test set). CodeSeer’s model training overhead, shown in Fig. 3.13,
amounts to less than 4 minutes to train RF for each application. Since CodeSeer’s model is designed to perform online prediction at the time that the target program is being launched, fast inference is critical. We observed that inference overhead is around 0.01 second for CodeSeer’s RF models after being translated from a Python implementation in Scikit-learn [com19a] to a C implementation with a model translator [com19b]. Such a low overhead makes CodeSeer’s random forest model viable for per-input code variant selection in HPC environments where programs run for hours or days.
3.6 Related Work

The most closely related work to ours focuses on program performance with iterative compiling techniques. These approaches are either search-based [Ans14; PE08; Pop16; Hal10; Yi12; Thi18; Che10] or machine learning-based [Ash16; Cav07; Nob16; JK13]. FuncyTuner [Wan19] is the state-of-the-art approach for optimizing modern scientific applications with caliper-guided [Boe16] fine-grained random search. FuncyTuner optimizes multiple hot loops simultaneously, while others [Hal10; Yi12] focus on a single loop at a time. Machine learning-based approaches [Ash16; Cav07; Nob16; JK13] aim to reduce the tuning overhead of search-based approaches while striving to achieve similar performance. However, these techniques usually evaluate the impact of program input sensitivity with only a handful of inputs, which limits their effectiveness as our work demonstrates. Moreover, Yang Chen et al. [Che10] studied program input sensitivity with 1000 data sets for a suite of multimedia benchmarks with the objective to determine a single code variant for a given embedded application. They observed that a program-specific compiler flag combination can achieve most of the best performance in their experiments. However, our evaluation shows that this may not be the case for modern scientific applications. Such applications usually have many more input parameters and runtime configurations. To alleviate their type-III input sensitivity, prediction-based code variant selection is critical to achieve the best performance across inputs.

Prior work also focuses on mapping program inputs to the best algorithmic variants or program configurations at runtime. Jae-Seung Yeom et al. [Yeo16] and Jiajia Li et al. [Li13] used machine learning models to learn such mappings from features of matrices to characterize solvers or algorithmic kernels. Frigo [Fri99] showed that a special-purpose compiler can generate specialized DFT kernels for a given input. These techniques are tied to a specific algorithm while CodeSeer has no assumption on the tuning target and is much more general. Other research [Mag13; TC17;
Mur14; ZM12; MZ13; Mur16] focuses on auto-tuning HPC kernels for modern GPUs or optimal runtime configurations [Yu18b]. Magni et al. [Mag13] propose to choose the best parameters for transforming sequential OpenACC loops into parallel threaded codes while considering program input sizes. Tillet et al. [TC17] build a multi-layered perceptron model to predict the performance of code variants with different tuning parameters. The model incorporates the impact of inputs and is able to guide the construction of the per-input best code variant at runtime. Muralidharan et al. [Mur14; Mur16] expose tunable parameters to programmers for code variant generation and construct a machine learning model for per-input code variant selection. These works focus on single GPU kernel selection at runtime while CodeSeer tackles type-III sensitivity for programs with multiple kernels at launch-time. Zhang et al. [ZM12; MZ13] propose an auto-tuner for 3D stencil code generation on several heterogeneous GPUs. They exhaustively search a space composed of GPU thread block dimensions and memory placements but do not consider the per-input optimal code variant selection.

As we approach exascale computing systems in HPC, power efficiency becomes more desirable due to high peak power consumption [Hal15]. Gholkar et al. [Gho16b] proposed techniques to improve HPC power efficiency at both the machine level and the job level. Bari et al. [Bar16] developed a framework called ARCS to automatically select the best runtime configurations for each OpenMP parallel region at a given power level. Marathe et al. [Mar17] modeled the interactions between an application's input parameters and system constraints such as power usage with deep transfer learning so that the best performing configuration can be identified quickly. Sourouri [Sou17] enforced fine-grained dynamic voltage and frequency scaling (DVFS) and OpenMP configurations to reduce energy consumption. These techniques emphasize the importance of runtime configurations to program performance and power efficiency, which are orthogonal to CodeSeer's compilation-time approach.

### 3.7 Conclusion

We proposed a novel auto-tuning framework, CodeSeer, that discovers program input sensitivity for modern HPC applications and performs per-input code variant selection via fast and precise machine learning models. With CodeSeer, we identified three types of program sensitivities. We showed that (1) the optimized executable produced for a target program by prior auto-tuning techniques remains sensitive to the tuning input (type-I sensitivity); (2) the performance benefit of the optimized executable reported by prior techniques may not generalize to many unseen inputs in practice (type-II sensitivity); and (3) the per-input best speedup may depend on different code variants (type-III sensitivity).

In contrast to prior auto-tuning techniques, CodeSeer address the fundamental challenges of type-I and type-III sensitivity by optimizing programs for many more inputs than prior work and by generating machine learning models to automatically perform per-input code variant selection at program launch-time. Evaluations show that CodeSeer's models achieved 90% and 92% precision...
(geometric mean) on balanced and unbalanced testing sets, respectively, while the prediction overhead is around 0.01 seconds. CodeSeer contributes a novel, computationally viable technique for improving performance for HPC programs with type-III sensitivity to an unprecedented level.
Ad hoc synchronizations are pervasive in multi-threaded programs. Due to their diversity and complexity, understanding the enforced synchronization relationships of ad hoc synchronizations is challenging but crucial to multi-threaded program development and maintenance. Existing techniques can partially detect primitive ad hoc synchronizations, but they cannot recognize complete implementations or infer the enforced synchronization relationships. In this paper, we propose a framework to automatically identify complex ad hoc synchronizations in full and infer their synchronization relationships. We instantiate the framework with a tool called BARRIERFINDER, which features various techniques, including program slicing and bounded symbolic execution, to efficiently explore interleaving space of ad hoc synchronizations within multi-threaded programs for their traces. BARRIERFINDER then uses these traces to characterize ad hoc synchronizations into different types, such as barriers. Our evaluation shows that BARRIERFINDER is both effective and efficient in doing this, and is also helpful for programmers to understand the correctness of their implementations.

4.1 Introduction

4.1.1 Motivation

In the current multi-core era [Her05; SL05], multi-threaded programming has become imperative to leverage the full power of modern CPUs. As multi-threaded programs share resources across
threads, programmers rely on proper synchronizations to ensure program correctness and efficiency. While a common set of standard synchronizations, such as mutex and condition variable operations, are provided by different languages or libraries, a recent study [Xio10] finds that programmers frequently choose not to use these standard synchronizations but implement their own ad hoc synchronizations for functionality or performance reasons. Researchers were able to find 6 to 83 ad hoc synchronizations in each of the 12 studied program suites [Xio10].

Because of the critical role that synchronizations play in multi-threaded programs, it is important to have an accurate understanding of the semantics of synchronizations and their enforced synchronization relationships. While standard synchronizations are easy to recognize and understand, ad hoc synchronizations have unmodularized implementations and enforce diverse synchronization relationships, making synchronization understanding a challenging task.

Fig. 4.1 shows an example to illustrate the basic concepts of ad hoc synchronizations. The ad hoc synchronization in Fig. 4.1 is formed by $S_2$ and $S_3$, where the shared variable $\text{flag}$ is called a sync variable, the while loop in $S_3$ is a sync loop, $S_2$ is a sync write, and the sync loop and sync write compose a sync pair. In this illustrating example, the sync pair formed by $S_2$ and $S_3$ enforces an order relationship between $S_1$ and $S_4$ that $S_1$ happens before $S_4$.

//Thread 1
counter = 5; //S_1
flag = false; //S_2

//Thread 2
while (flag); //S_3
counter++; //S_4

Figure 4.1 An ad hoc synchronization example formed by $S_2$ and $S_3$. counter and flag are global variables. flag is initialized to true.

To detect ad hoc synchronizations, researchers have proposed various techniques [JT10; JT14; Tia08; Tia09; Xio10; Yin13; Yua13]. However, existing techniques only detect sync pairs, i.e., sync loops and their corresponding writes, but they do not further infer synchronization relationships being enforced. This is problematic, as a sync pair can implement a mutual-exclusion relationship or different types of order relationships. For example, Fig. 4.2 shows another ad hoc synchronization with the sync pair in lines 23 and 28 labeled, but it implements a barrier.

Not only do programmers have difficulties in understanding the intended order relationship by the sync pairs and verify their correctness [Gu15], but also multi-threaded program development tools, such as data-race detectors [Bes10; Lee12], concurrency-bug finding tools [Par09; Zha10], automated bug-fixing tools [Jin11; Jin12], and synchronization-oriented performance profilers [YP16; CS12], cannot directly use the ad hoc synchronization detection results of these existing tools. For example, SyncFinder [Xio10], which is the state-of-the-art tool for detecting ad hoc synchronizations, can detect the sync pair in Fig. 4.1, but it does not determine the order relationship enforced by $S_2$ and $S_3$. As a result, race detectors need to conduct further analysis on top of SyncFinder results. Otherwise, they could conclude that $S_1$ and $S_4$ constitute a data race on the shared variable counter, resulting in a false positive. To determine the order relationship in Fig. 4.1, one could enumerate all
1 int gsense = 1, gcount = 0, P = ...; // input
2 main() {
3     for (i=1; i<P; i++)
4         pthread_create(SlaveStart, ...);
5 ...  
6     SlaveStart();
7 }
8 SlaveStart() {
9     ... // computation and two barriers
10    for (...) {
11       ... // computation
12       { // barrier begin
13          int lsense = gsense;
14          while (1) {
15              int oldcount = gcount;
16              int newcount = oldcount + lsense;
17              // atomic compare exchange using assembly
18              int updatedcount = CmpXchg(&gcount,
19                                              oldcount, newcount);
20              if (updatedcount == oldcount) {
21                  if ((newcount == P && lsense == 1)
22                      || (newcount == 0 && lsense == -1)) {
23                      gsense = -lsense; // the sync write
24                  }
25              }
26          }
27          while (gsense == lsense); // the sync loop
28       } // barrier end
29     } // computation and one barrier
30 }
31 ... // computation and one barrier
32 }

Figure 4.2 Extracted code from SPLASH2 LU

possible interleavings and see the temporal invariant that $S_1$ will always happen before $S_4$, due to the simplicity of this example.

However, inferring the synchronization relationship after detecting sync pairs is not always as easy as the one in Fig. 4.1, and sometimes it can be very challenging. The ad hoc barrier in Fig. 4.2 exemplifies the two major challenges:

- First, a sync pair, which is the only information reported by existing ad hoc synchronization detectors, may be only a part of an ad hoc synchronization. Without considering extra code, it may be impossible to infer the enforced synchronization relationship. For example, the sync pair in Fig. 4.2, is only a portion of the complete ad hoc synchronization implementing a barrier. To recognize the ad hoc barrier, all the code from line 12 to 29 needs to be considered, in addition to their threading context from line 3 to line 6. In this example, the static control flow is already complex, and determining the threading context involves non-trivial interprocedural analysis.
• Second, there can be an excessive number of feasible thread interleavings to consider for inferring synchronization relationships and verifying their correctness. Although the example in Fig. 4.1 has a small interleaving space and the synchronization relationship can be inferred with ease, the example in Fig. 4.2 has a much larger interleaving space, the complexity of which will be detailed in Sec. 4.2.3. Without a thorough exploration or a proof, one cannot be sure what synchronization relationship is enforced by a sync pair and relevant code constructs or if the implementation is correct.

To sum up, understanding ad hoc synchronizations in terms of their enforced semantics and correctness is an important but challenging task that has not been addressed. Techniques to bridge the gap, anywhere between existing ad hoc synchronization detection tools and various multi-threaded program development tools, are in a great need to make the results from the former to be more useful for the latter.

4.1.2 Contribution

To tackle these challenges and bridge the gap, we propose an ad hoc synchronization analysis framework to (1) automatically recognize complex ad hoc synchronizations beyond simple sync pairs, and (2) efficiently infer the enforced synchronization relationships without repetitively examining equivalent interleavings. To the best of our knowledge, no existing technique has accounted for such complexity.

We currently instantiate the framework for automatic recognition of ad hoc barriers and present BARRIERFINDER. We choose to focus on ad hoc barriers because they are both common and beneficial to be recognized. The ad hoc synchronization study [Xio10] reported that barriers are a common type of synchronizations with ad hoc implementations. Further, a recent work also shows that the recognition of barriers can reduce the complexity of many multi-threaded program analyses and improve many development tools [Das15]. Our approach capitalizes on the intuition that all ad hoc barriers enforce a temporal invariant among different thread interleavings. Specifically, the temporal invariant requires that no participating threads can proceed beyond a program point (blocking point) before the last participant has reached a specific program point (releasing point), so that in effect computation prior to the blocking point are finished in all threads before computation after the blocking point can be executed in any thread. As shown in Fig. 4.3, BARRIERFINDER takes

![Figure 4.3](image-url) The architecture of BARRIERFINDER. IR: interleaving reduction. IA: interleaving avoidance. ET: early termination.
program source code and sync pairs detected by SyncFinder [Xio10] as inputs, and it proceeds in four steps to determine whether each sync pair and any relevant code compose an ad hoc barrier:

1. As a sync pair is seldom a complete ad hoc synchronization itself, we first slice the program with sync pairs as the slicing criteria. This helps us identify program constructs that are also integral parts of ad hoc barriers.

2. We then analyze and instrument the program slices with auxiliary APIs, such as scheduling and tracing APIs. These APIs are directives to examine the temporal invariant of the sliced program constructs by efficient interleaving enumeration.

3. We further symbolically execute the program to exhaustively enumerate nonequivalent interleavings and to generate traces representing these interleavings.

4. Finally, we mine the interleaving traces to find predefined temporal patterns and infer the synchronization relationship. Since BARRIERFINDER focuses on ad hoc barriers, we define patterns for barriers. BARRIERFINDER reports whether a sync pair is part of an ad hoc barrier. If that is the case, it reports the complete barrier implementation. Otherwise, it reports the context of the violation.

Overall, this paper makes the following contributions:

- We propose a framework to infer the synchronization relationship enforced by ad hoc synchronizations. To our knowledge, we are the first to analyze ad hoc synchronizations beyond recognizing sync pairs.

- We instantiate our framework for ad hoc barriers and implement BARRIERFINDER with several novel techniques to account for interleaving space blow-up and to boost its execution efficiency.

- We evaluate BARRIERFINDER on both real-world programs and synthetic benchmarks. Results suggest that our approach is efficient and effective in recognizing different ad hoc synchronizations and can also help programmers understand the correctness of ad hoc synchronizations.

- We demonstrate how BARRIERFINDER’s result can be complementary generalized to an unlimited number of concurrent threads with a proof sketch.

4.2 BARRIERFINDER Design

BARRIERFINDER employs various static and dynamic techniques to support efficient interleaving space enumeration. As shown in Fig. 4.3, there are 3 pipelined steps in BARRIERFINDER. The front end performs compile-time inter-procedural program slicing, sync region boundary analysis, and instrumentation on program LLVM IRs. The middle end symbolically executes the preprocessed
program to collect sync traces with 3 critical techniques to tackle the challenge of efficient inter-leaving enumeration. Barri erFinder’s back end analyzes sync traces and reports synchronization relationships for sync regions and their source code context information or the violation context. In this section, we elaborate on its design considerations and discuss alternatives when applicable.

4.2.1 Front End Preprocessing

4.2.1.1 Interprocedural Slicing

BarrierFinder employs program slicing [Wei81; Wei84] with the objectives to (1) reduce code size and (2) preserve the execution context for sync regions. To recognize ad hoc barriers beyond sync pairs, such as those in Fig. 4.2, any viable approach has to capture the execution context of the sync region, including the initial values of sync variables, the number of participating threads, and sync region boundaries. Moreover, computation code is irrelevant in general for inferring the semantics of ad hoc synchronizations and may be sliced away to improve BarrierFinder’s efficiency.

BarrierFinder’s slicing step leverages LLVM Slicer [Sla15], implementing Anderson’s algorithm with field-sensitivity. We further enhance LLVM Slicer to address the challenge of concurrent program slicing as follows.

- We enhance the original slicing algorithm in LLVM Slicer [Sla15] by constraining the target programs to those based on the Pthread standard, so that Pthread API calls are always preserved in the slice.

- We preserve program concurrency constructs in two ways. First, the slicer leaves all well-defined sync constructs intact. Second, all potential accesses to sync variables are properly marked as sync writes and reads in sync loops based on SyncFinder results. We can thus guarantee the equivalence of concurrent events between the original program and its sliced counterpart.

4.2.1.2 Sync Boundary Detection

Sync region boundary information marks the boundaries between computation and sync regions. Such boundary information is critical to generating separable traces for consecutive ad hoc synchronizations. Since there may be multiple ad hoc synchronizations in a program, e.g., the LU code in Fig. 4.2 has 5 barriers, BarrierFinder needs sync region boundary information to instrument trace separators, which are used to distinguish consecutive synchronizations during the trace analysis stage.

A natural boundary is the first instruction sliced away with respect to the slicing criteria, and indicates the ending point of computation code prior to a sync region. To detect boundaries, BarrierFinder relies on a heuristic based on the observation that a sync region tends to have data dependencies only on sync variables, which are global or on the heap. BarrierFinder realizes this heuristic in a classical backward data flow analysis algorithm. This algorithm identifies (1) the
1 Input: CFG of the Function F where sync code resides,
2 worklist={basic blocks in the read-side loop}
3 Output: the boundary instruction, bi
4 Steps:
5 do{
6   do{
7     for bb in worklist
8       // reversely traversal of instructions in bb
9       for ins in bb{
10       // In[ins]/Out[ins] is In/Out set for ins
11       calculate In[ins], Out[ins];
12       if(worklist.size() == 1 && In[ins].localSet.size() == 0)
13       return ins;
14     }
15   }while(if any bb’s In set changed);
16   /* compute new worklist for bb’s whose successors have been visited. */
17   for bb in worklist
18     push bb’s unvisited predecessors and all its visited successors into newlist
19   for bb in newlist
20     compute bb’s In Sets;
21   clear worklist;
22   copy newlist into worklist;
23 }while(worklist is not empty);

Figure 4.4 Sync Boundary Detection Algorithm

first instruction accessing a sync variable as the start of a region and (2) the first instruction in the immediate post dominator of a sync loop as the ending boundary. The algorithm in Fig. 4.4 utilizes classical backward data flow analysis. Gen/Kill sets are computed implicitly for each instruction traversed. For example, the load access of a global variable into a local variable would kill the local variable, and record the address of global variable as a live variable. In/Out sets are computed for the ending and beginning instruction of the analyzed basic block, respectively. The meet/join operator is the set union of the successors’ Out sets. To account for the existence of loops, a fixed-point algorithm is applied. The algorithm terminates when the set of local variables is empty for the current instruction (line 14 in Fig. 4.4).

4.2.2 Instrumentation

After detecting the boundary of a sync code region, the BARRIERFINDER frontend instruments the region with special instructions to be interpreted in Cloud9. There are several different instructions to be inserted.
4.2.2.1 Trace API Instrumentation

There are several considerations while deciding where to instrument trace APIs. First, we have to distinguish sync writes and sync loops by different tracing events. Second, we want to insert a minimal but sufficient number of trace APIs. Since trace generation is interpreted in BARRIERFINDER, it is necessary to minimize its performance overhead. Moreover, if too many runtime events are traced, temporal-invariant mining would suffer from unnecessary overhead within the BARRIERFINDER step. BARRIERFINDER relies on the following rules to satisfy these constraints:

Rule 1 Insert a trace API call that generates a character ‘R’ right after a sync loop.
Rule 2 Insert a trace API call that generates a character ‘W’ right before a sync write.
Rule 3 Insert a trace API call that generates a monotonically increasing separator as an integer counter at the beginning boundary of a sync region.

With Rules 1 and 2, one character that corresponds to one access to a sync variable will be generated, and these characters allow us to distinguish sync writes and sync loops. With Rule 3, we expect a separator to facilitate the distinction of traces over the detected ad hoc synchronizations. With these rules, we obtain traces like 11WRR22WRR for two consecutive barriers with 2 threads. It is then straightforward for BARRIERFINDER’s trace analyzer to separate traces into several independent ones and to correlate them to sync regions.

4.2.2.2 Scheduling API Instrumentation

To guide the symbolic engine to enumerate different interleavings, BARRIERFINDER further instruments the sliced program with scheduling APIs. To decide where to instrument scheduling APIs, we divide all instructions within a sync region into two categories.

1. Instructions accessing global variables or heap variables, which are visible to all threads, denoted as IG. IGs impose side-effects across threads and affect their execution.
2. Instructions other than IG, such as accesses to thread-local variables, denoted as IL. IL can neither “import” side-effects from other threads affecting its own execution nor “export” side-effects to affect other threads.

As different interleavings of instructions in IL do not change the global program state, it is sufficient to instrument a scheduling API after each instruction in IG.

The scheduling APIs guide BARRIERFINDER’s symbolic execution engine to decide when to explore new interleavings by forking new execution states. BARRIERFINDER’s symbolic execution engine represents an interleaving with an execution state, and there is a one-to-one mapping between execution states and interleavings. As shown in Fig. 4.5, there is only one execution state initially. When BARRIERFINDER’s symbolic execution engine sees a scheduling API, it forks new execution states for each possible interleaving. Suppose there are t threads, denoted as Ti (1 ≤ i ≤ t), ready to run in execution state ES0, excluding the currently running thread T0. When BARRIERFINDER interprets a scheduling API, it first makes t copies of ES0, denoted as ESi (1 ≤ i ≤ t). It then schedules Ti (1 ≤ i ≤ t) as the running thread for ESi, as shown in the middle of Fig. 4.5. As a result, newly
forked execution states $ES_i$ ($1 \leq i \leq t$) and $ES_0$ only have different running threads. All the other execution contexts and resources, such as their registers, memory contents and data layouts, are exactly the same. Afterwards, each execution state is executed independently to enumerate all interleavings within a sync region with repeatedly forked states at scheduling instructions until the program exits.

The running thread
A suspended thread
Before scheduling
After scheduling 
& before reduction
After reduction

Figure 4.5 Interleaving enumerations and reduction: An execution state in BARRIERFINDER corresponds to an interleaving.

### 4.2.3 Tackling Space Explosion in the Middle End

To tackle the challenge of interleaving space explosion, BARRIERFINDER’s middle end relies on three techniques, namely interleaving reduction (IR), interleaving avoidance (IA), and early loop termination (ET). These techniques exploit interleaving equivalence to shrink the exponential interleaving space and enable BARRIERFINDER’s efficient interleaving enumeration within sync regions.

#### 4.2.3.1 Interleaving Equivalence Test – The Foundation

If two interleavings produce the same sync traces on all future occasions, BARRIERFINDER considers them to be equivalent. Specifically, if BARRIERFINDER finds two interleavings with the same execution context, i.e., program pointers, calling stacks, and memory contents, across threads, it can guarantee their equivalence and reduce them to one without missing any distinctive future sync traces. This is because program execution only depends on the current execution context
but not their past interleavings. However, this may be too restrictive for barriers, where thread identity does not matter. Taking this into consideration, BARRIERFINDER excludes identifiers of participating threads from the equivalence test. If two interleavings are equivalent, we call either one of them an interleaving invariant with respect to a program point $P$ at run time $T$. $P$ is also called an invariant derivation point, because the interleaving invariant is derived in this place. $T$ is called the interleaving reduction time since the reduction of equivalent interleavings happens then. To identify $P$ and $T$, BARRIERFINDER relies on heuristics, which are elaborated for interleaving reduction and interleaving avoidance in Sec. 4.2.3.2 and Sec. 4.2.3.3, respectively.

4.2.3.2 Interleaving Reduction – The Enabling Technique

Interleaving reduction (IR) reduces redundant exploration among equivalent interleavings. The program point $P$ and reduction time point $T$ to perform interleaving reduction is identified with the following heuristics:

- $P$ should be within a post-dominator basic block of sync region ending boundary that all participating threads execute. If only a subset of participating threads execute $P$, BARRIERFINDER might miss certain invariants.
- $T$ should be chosen such that it is likely for interleavings to be equivalent, for example, when the first or the last participating thread passes $P$.

BARRIERFINDER selects the first instruction in a sync region’s immediate post-dominator basic block as $P$, relative to the sync region’s ending point. $T$ is selected as the time when the last thread passes $P$. We implement a classical reference counter to maintain the number of threads entering and exiting a sync region. The counter is initialized to zero indicating that no thread is currently in the sync region. On entry, the counter is incremented; at the exit, it is decremented. When the counter is zero again, the runtime library knows that the last thread has just passed through the region such that its time $T$ and the interleaving invariants can be derived.

Upon the first interleaving reaching $T$, there is no interleaving invariant yet. Hence, BARRIERFINDER adds this interleaving to the invariant set (IS), suspends its execution and schedules other interleavings for execution. For any subsequent interleaving reaching $T$, if an equivalent interleaving is found in IS, the new interleaving is terminated and all its resources are released immediately; otherwise, BARRIERFINDER adds it to IS as a new invariant. This process continues until all interleavings have been enumerated and executed. Then, BARRIERFINDER schedules and executes all invariant interleavings in IS. As illustrated in Fig. 4.5, 3 interleavings are assumed to be equivalent after they are forked and further explored independently. With interleaving reduction, they are reduced into one representative while the other two are terminated.

4.2.3.3 Interleaving Avoidance – Loop-centric Technique One

If a new interleaving to be explored has the same execution context as another interleaving that has already been explored in previous loop iterations, BARRIERFINDER can avoid exploring this new interleaving. Interleaving avoidance (IA) targets such opportunities for sync regions that are
executed multiple times, especially those in loops. In contrast to interleaving reduction, interleaving avoidance (IA) is performed before a sync region is entered.

Specifically, **BarrierFinder IA** selects program point $P$ as the first instruction in the immediate dominator basic block of a sync region's beginning boundary and time point $T$ as when the first thread executes $P$. **BarrierFinder IA** adds the first interleaving hitting $T$ to invariant set $IS$, snapshots its execution context, and continues its exploration as normal. When a subsequent interleaving hits $T$, **BarrierFinder** checks whether it is equivalent to any invariant in $IS$ with the equivalence test. If successful, interleaving enumeration for this new interleaving is avoided. Otherwise, a new interleaving is identified that has not been explored before and the algorithm proceeds as normal. In this way, IA prevents redundant interleaving enumerations from being considered at all across loop iterations.

### 4.2.3.4 Early Loop Termination – Loop-centric Technique Two

We observe that there is usually left-over computation code after slicing. Such code is not sliced away due to limitations of interprocedural slicing. If such code is within loops, it may introduce a high execution overhead. Moreover, sync regions do not expose new traces after several loop iterations in practice. Early loop termination (ET) takes advantage of these observations by breaking out the innermost loop (ET-loop), which contains the sync regions being explored after the ET-loop fixed-point state is established. A fixed-point state for each sync region is established after a sync region has been explored under the same execution context twice. The fixed-point state for the entire loop is established when all encompassed sync regions have seen a fixed-point state at least once, which is also the point in time for ET to be applied. To terminate ET-loop early, ET sets the program counters of all threads to the first instruction in the immediate post-dominator basic block of the current ET-loop.

### 4.2.4 Optimizations for Execution Efficiency

We choose the state-of-the-art symbolic execution engine Cloud9 [Buc11] as the infrastructure for **BarrierFinder**'s middle end. The execution engine is based on interpretation, which is slow compared to native execution. Therefore, it is critical to minimize the overhead related to interleaving enumeration.

#### 4.2.4.1 Incremental Snapshotting

**BarrierFinder** employs a snapshot mechanism to capture execution contexts, which are used in the interleaving equivalence test 4.2.3.1. To reduce the cost, **BarrierFinder** takes a minimal snapshot as follows. Let's suppose $SR$ is a sync region being explored. For interleavings that are created when the first scheduling point within $SR$ is hit at time $c_0$, their execution contexts (except for their thread state, i.e., suspended/running, etc.) are the same. As execution progresses, other
execution contexts, e.g., global variables, may diverge. Therefore, it is sufficient to only snapshot those variables that are updated since $c_0$ and are still alive when the equivalence test is performed.

BARRIERFINDER's front end is responsible for liveness analysis and snapshot API instrumentation. Snapshotting is incrementally performed within the BARRIERFINDER middle end during sync region exploration. This minimizes a snapshot.

### 4.2.4.2 Intrusive Tracing and Beyond

Fig. 4.6 shows the address space layout for BARRIERFINDER. BARRIERFINDER symbolic execution engine (interpreter) crosses the interpreter-interpretee address space boundary for native execution if possible. First, per-interleaving traces are initially generated and stored in a trace buffer within the interpretee's address space. BARRIERFINDER accesses and dumps its trace buffer into a file without the interpretee's involvement when an interleaving terminates. We call this technique intrusive tracing. Similarly, instead of relying on the interpretee to capture its execution context to a file, BARRIERFINDER snapshots a per-interleaving execution context in buffers within the interpretee's address space. BARRIERFINDER's middle end later reads these buffers directly, performs the equivalence test, and installs a new invariant in the interpreter's address space if necessary. All these happen in native mode without file I/O.

### 4.2.5 Mining Sync Traces for Knowledge in the Back End

BARRIERFINDER's back end analyzes sync traces to derive critical information for sync region understanding. Let's take a sample trace $11WRR$ as an example. First, BARRIERFINDER extracts the sync region ID 1 and use it to separate sync traces to two parts. It then checks whether the sub-trace is self-consistent, i.e., the number of 1s should be equal to the number of Rs since there is one trace point for each of them at the beginning and ending boundary of a sync region respectively. If self-consistency is confirmed, then BARRIERFINDER tries to match $WRR$ with our pre-defined invariant for barriers $WR^N$. Since it succeeds for this trace example, BARRIERFINDER uses the extracted ID 1 as the key to retrieve context information, such as source code line numbers for the sync region and reports the code region as a barrier.
4.3 Experimental Evaluation

4.3.1 Prototype

We implemented BARRIERFINDER’s front end on top of LLVM Slicer [Sla15], and the boundary analysis and instrumentation pass is implemented as a sub-pass inside the slicer. BARRIERFINDER’s middle end is built on top of Cloud9 [Buc11], for its flexible interpretation and symbolic execution capabilities. The back end is a stand-alone python package, which separates collected sync traces into independent ones, according to sync region IDs, and also maps them back to their corresponding program source code contexts. This way, BARRIERFINDER reports the number of recognized ad hoc barriers and their respective source code ranges. If a sync pair is not a part of an ad hoc barrier, BARRIERFINDER will report where a violation of the expected runtime invariants is found.

4.3.2 Methodology and Experimental Settings

We conduct empirical experiments to evaluate the efficiency and effectiveness of BARRIERFINDER on the SPLASH2 [Woo95] suite and a synthetic benchmark suite. All measurements are conducted on a machine with Intel Core i7-4790 @ 3.60 GHz (hyper-threading enabled), [Cor14], 16GB DDR3@1600 MHz main memory, and Ubuntu 15.10 as the operating system.

We use applications from the SPLASH2 suite to evaluate the efficiency of BARRIERFINDER, and we pay special attention to how different techniques speed up the interleaving enumeration process. SPLASH2 is used in Xiong et al. [Xio10]’s ad hoc synchronization study as a representative suite for scientific applications. These programs contain complex control flows and many functions in order to show the intellectual merits of BARRIERFINDER, since they present consecutive barriers, including barriers within loops, to show the efficiency of BARRIERFINDER, and also stress BARRIERFINDER’s interprocedural analyses, such as slicing.

We use a synthetic benchmark suite to evaluate the effectiveness of BARRIERFINDER, i.e., whether it can correctly differentiate ad hoc synchronizations that implement barriers from those do not. Our synthetic benchmarks have a similar structure. The main thread first creates one child thread, and all threads execute twice the same sequence of code, which is a variant of barrier based on the textbook by Taubenfeld [Tau06]. In total, we have eight different variants. They are all counter based [Tau06; MB14], just as the one in Fig. 4.2. Among the eight variants, two are correct ad hoc barriers, where one of them is the same as the ad hoc barrier in Fig. 4.2, and the other mainly differs in that it will reset gcount to 0 before line 23 and always increment oldcount by 1 in line 16. The other six are all wrong implementations because of different reasons, e.g., different initial values for sync variables, different orders of certain statements, and whether or not atomic instructions are used. Because of these differences, these six wrong implementations violate the semantics of barriers or may result in deadlocks.

To show that our proposed framework is versatile, we also include one benchmark implementing an allAB [Jin12] relationship extracted from MySQL in our synthetic benchmark suite. The allAB
relationship in our benchmark requires that the main thread executes the B operation cannot proceed until both child threads cannot execute more A operations, which can be viewed as a variant on barriers.

Since these synthetic benchmarks stress less on interleaving-space enumeration, we focus on whether or not BarrierFinder can correctly recognize different ad hoc synchronizations and omit the performance measurements.

BarrierFinder takes sync pairs as input. Since SyncFinder [Xio10] is no longer maintained by the original authors and the code is not available to us, sync-pair annotations are manually inserted. Note that sync pairs are low-level primitive synchronization constructs that they are just busy-wait loops and write accesses to shared variables. They are neither complete implementations of ad hoc synchronizations nor do they indicate the enforced synchronization relationships.

### 4.3.3 Efficiency Results on Real-World Benchmarks

Tab. 4.1 shows the results for the six SPLASH2 benchmarks currently supported by BarrierFinder. Column “LOC” lists the number of lines of C source code, and column “LOB” lists the number of lines of LLVM bitcode. We then show the slicing time of BarrierFinder’s front end and the number of ad hoc barriers (column “Patterns (#)”). We next show the runtime of BarrierFinder to exhaustively enumerate the interleavings with the number of threads bounded to 2. For the remaining columns, subscripts r, s, and t represent interleaving reduction, program slicing, and intrusive tracing, respectively. Different subscript combinations show the runtimes consumed by BarrierFinder’s middle end with different optimizations enabled. For example, $T_{str}$ is the runtime with all three optimizations enabled, while $T_{sr}$ is the runtime with slicing and intrusive tracing enabled but interleaving reduction disabled. N/A indicates benchmark crashes, and OOR indicates the execution runs out of memory. Runtimes (in seconds) are averaged for 10 runs, with their standard deviations in parentheses. To handle the default trip count of 32 interactions in LU requires interleaving avoidance and early loop termination, the results for LU in Tab. 4.1 are measured with trip count as 1 to show the benefits of slicing, intrusive tracing, and interleaving reduction on LU.

**Table 4.1** Overall results of BarrierFinder on SPLASH2 with slicing, intrusive tracing, and interleaving reduction

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>LOC.</th>
<th>LOB.</th>
<th>Slicing time</th>
<th>Patterns (#)</th>
<th>$T_r$</th>
<th>$T_s$</th>
<th>$T_t$</th>
<th>$T_{str}$</th>
<th>$T_{sr}$</th>
<th>$T_{st}$</th>
<th>$T_{str}$</th>
<th>$T_{str}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT</td>
<td>1.2k</td>
<td>4679</td>
<td>0.2 (0.001)</td>
<td>barriers (7)</td>
<td>OOR</td>
<td>OOR</td>
<td>57.6 (0.44)</td>
<td>OOR</td>
<td>17.4 (0.1)</td>
<td>1.3 (0.06)</td>
<td>13.4</td>
<td></td>
</tr>
<tr>
<td>Cholesky</td>
<td>6.1k</td>
<td>2679</td>
<td>94.8 (0.17)</td>
<td>barriers (4)</td>
<td>N/A</td>
<td>N/A</td>
<td>24 (0.3)</td>
<td>N/A</td>
<td>2.5 (0.06)</td>
<td>9.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Raytrace</td>
<td>11k</td>
<td>24173</td>
<td>15.8 (0.04)</td>
<td>barriers (1)</td>
<td>N/A</td>
<td>N/A</td>
<td>8.6× (0.06)</td>
<td>17.4×(0.06)</td>
<td>8.3×(0.08)</td>
<td>2.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Radix</td>
<td>1.2k</td>
<td>3856</td>
<td>0.1 (0.02)</td>
<td>barriers (7)</td>
<td>OOR</td>
<td>OOR</td>
<td>108.8 (1.0)</td>
<td>OOR</td>
<td>4.5 (0.17)</td>
<td>24.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LU</td>
<td>1.1k</td>
<td>4555</td>
<td>0.53 (0.001)</td>
<td>barriers (5)</td>
<td>N/A</td>
<td>N/A</td>
<td>31.3 (0.2)</td>
<td>N/A</td>
<td>1.7 (0.01)</td>
<td>18.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FMM</td>
<td>5k</td>
<td>16583</td>
<td>18.2 (0.1)</td>
<td>barriers (10)</td>
<td>OOR</td>
<td>OOR</td>
<td>355.4 (7.8)</td>
<td>OOR</td>
<td>333.5 (1.6)</td>
<td>12.3 (0.08)</td>
<td>27.1</td>
<td></td>
</tr>
</tbody>
</table>
4.3.3.1 Observations

We make the following observations from our results:

1. **BarrierFinder** is effective in detecting different numbers of ad hoc barriers in these benchmarks, and we manually confirmed that **BarrierFinder** detects all the barriers in each benchmark. To the best of our knowledge, **BarrierFinder** is the first tool to have such a capability. No prior work, including SyncFinder [Xio10], can detect any of these ad hoc barriers in whole. The trace generated for two consecutive barriers in LU is $11\text{WRR}22\text{WRR}$. **BarrierFinder**’s back end divides such a string by considering $11$ and $22$ as separators. The two characteristic sub-traces $\text{WRR}$ match our predefined temporal invariant for barriers, and their corresponding sync regions are accordingly reported as barriers. The sync regions contain both the upper and lower loops in Fig. 4.2. The detected pattern and sync region reports show that **BarrierFinder** is able to detect the entire code construct of ad hoc barriers and recognize their barrier semantics. Note that it is possible for **BarrierFinder** to report false positives since it can only exhaustively enumerate the interleaving space when there are only two participating threads. However, there is no such case in our evaluation and we are unaware of such code in practice. False negatives cannot occur solely due to the bounded number of threads, because an ad hoc barrier has to enforce the temporal invariant even if there are only two participating threads.

2. **BarrierFinder** is efficient in recognizing ad hoc barriers. Specifically, column “$T_{str}$” in Tab. 4.1 shows the time spent in the middle end, which is usually less than 10 seconds when there are two participating threads. This shows our optimization techniques, combined together, make our approach quite efficient.

3. Interleaving reduction is the critical technique that enables **BarrierFinder** to efficiently enumerate the interleaving space of ad hoc barriers. Column “$T_{st}$” shows the runtimes of the middle end with slicing and intrusive tracing enabled but without interleaving reduction (IR). Except for Raytrace that contains only one barrier, all benchmarks run out of memory resources in less than two minutes and progress very slowly after that. In comparison, runtimes in column “$T_{str}$” show that IR is critical for **BarrierFinder**’s efficiency.

4. Slicing is critical for **BarrierFinder**’s middle end to succeed in analyzing the benchmarks. As shown in column “$T_{r}$”, without slicing, **BarrierFinder**’s middle end crashes for Cholesky, Raytrace, and LU. The cause is rooted in Cloud9, but all benchmarks succeed with slicing enabled. The slicing overhead for FFT, Radix, and LU is small, but it is higher for Cholesky and Raytrace. The general trend is that larger benchmarks incur higher slicing overhead. The benefit of slicing is that it eliminates code that is irrelevant to synchronization explorations and improves middle-end efficiency, which is substantiated by comparing $T_{str}$ and $T_{r}$ for FFT. Without slicing, the runtime for Radix is also prohibitively high as its computation exhausts main memory quickly.

5. Intrusive tracing boosts **BarrierFinder**’s middle-end performance by up to 27X. Column “$\frac{T_{str}}{T_{sr}}$” in Tab. 4.1 indicates a significant speedup due to our trace optimization technique, which crosses the interpreter-interpretee boundary.
4.3.3.2 The Benefits of IA and ET

In our current benchmarks, only LU, Radix, and FMM have loops that encompass sync regions. We manually adapt their loop trip counts to demonstrate the effectiveness of interleaving avoidance (IA) and early loop termination (ET). Specifically, we measure the runtime for the following configurations: (1) Interleaving reduction only (IR), (2) IR with IA but without ET (IR-IA), and (3) IR-IA with ET (IR-ET). Figs. 4.7a, 4.7b, and 4.7c show the performance results for Radix, FMM, and LU, respectively. The x-axis indicates the loop trip count while the y-axis depicts BARRIERFINDER’s middle-end runtime in seconds averaged over 10 consecutive runs. Since performance variability is small (see standard deviations in Tab. 4.1), other runtime statistics are omitted to save space. Note that results for IR are provided only if BARRIERFINDER does not exhaust the 16GB memory under that configuration.

![Graphs showing performance results for Radix, FMM, and LU](a) Radix (b) FMM (c) LU

**Figure 4.7** The effect of interleaving avoidance (IA) and early loop termination (ET) under different iteration counts
Table 4.2 Overall results of BARRIERFINDER on the synthetic application suite. Numbers in traces are sync region IDs.

<table>
<thead>
<tr>
<th>ID</th>
<th>Actual</th>
<th>Characterized</th>
<th>Deadlock trace</th>
<th>Violation sequence ID</th>
<th>Atomic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>barrier</td>
<td>barrier</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
<td>A reusable barrier</td>
</tr>
<tr>
<td>2</td>
<td>barrier</td>
<td>barrier</td>
<td>No</td>
<td>N/A</td>
<td>Yes</td>
<td>A reusable barrier</td>
</tr>
<tr>
<td>3</td>
<td>bad barrier</td>
<td>bad barrier</td>
<td>Yes</td>
<td>11WRR</td>
<td>3320</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>bad barrier</td>
<td>bad barrier</td>
<td>No</td>
<td>11WR2RR2WR</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>bad barrier</td>
<td>bad barrier</td>
<td>No</td>
<td>11WR2RR2WR</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>bad barrier</td>
<td>bad barrier</td>
<td>Yes</td>
<td>11WR2RR2WRW</td>
<td>18</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>bad barrier</td>
<td>bad barrier</td>
<td>Yes</td>
<td>11WR2</td>
<td>0</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>bad barrier</td>
<td>bad barrier</td>
<td>Yes</td>
<td>11WR2</td>
<td>0</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>allAB</td>
<td>allAB</td>
<td>No</td>
<td>N/A</td>
<td>N/A</td>
<td>allAB</td>
</tr>
</tbody>
</table>

The results show IA and ET are the enabling techniques for exploring sync regions in loops. As we see in these figures, ET and IA significantly improve BARRIERFINDER’s efficiency compared to IR only. Such a performance improvement is critical for LU, which otherwise cannot be explored without reduction on its loop trip count when only IR is enabled. Also, IA and ET enable BARRIERFINDER to explore two orders of magnitude more iterations than IR-only for Radix and FMM.

4.3.4 Effectiveness Results on Synthetic Benchmarks

Our synthetic benchmark suite contains eight variants of barriers and one allAB. The expected trace patterns without sync region IDs are WRRWRR and WWR for barriers and allAB, respectively. BARRIERFINDER exits once a violation to either pattern or a deadlock is found.

Tab. 4.2 shows the effectiveness results of BARRIERFINDER on our synthetic benchmark suite. “Deadlock” indicates if a deadlock occurs; “Violation trace” is the first trace violating expected trace patterns while “Violation sequence ID” shows how many valid traces have been generated before the violation trace; “Atomic” indicates whether or not the barrier counter, e.g., gcount in Fig. 4.2 is accessed and checked atomically, and “Description” provides a short description for each benchmark’s detection result.

We make the following observations from results in Tab. 4.2:

1. BARRIERFINDER has neither false positives nor false negatives for the synthetic application suite (“Actual” and “Characterized” are the same), since it enumerates all non-equivalent feasible interleavings.

2. BARRIERFINDER reliably differentiates correct barrier implementations from wrong ones. Barriers 1 and 2 are correct while others have different problems. For a violation, BARRIERFINDER not only reports the violation trace but also produces the contexts, e.g., thread scheduling status and call stacks, which helps programmers understand the root causes. As shown by “Description”, incorrect barriers encounter different problems.
3 \textsc{BarrierFinder} recognizes allAB, which can be regarded as a different variant of barriers. This also shows the potential for further generalization of the framework to other ad hoc synchronizations.

4.3.5 Generalization to Any Number of Threads

\textsc{BarrierFinder} successfully enumerates the interleaving space of ad hoc barriers when the number of participating threads is small, which is critical for automatic recognition of synchronizations and understanding of their correctness. To go beyond that, we give an inductive proof over the number of participating threads, \( n \). Consider the following invariants for the algorithm in Fig. 4.2: (1) At line 28, which we refer to in the following as the program blocking point, \( P_b \), the invariant \( I_b(i) : \text{newcount} = i \) holds for all threads \( 1..n-1 \) and these threads will busy wait at \( P_b \) as long as \( g\text{sense} = l\text{sense} \). (2) At line 23, \( I_r(i) : \text{newcount} = i \) holds for thread \( i = n \), and the postcondition of \( I_r \), is \( g\text{sense} \neq l\text{sense} \), which will subsequently cause threads \( 1..n-1 \) to proceed past \( P_b \) by exiting the loop. In conjunction, \( I_b \) and \( I_r \) establish the barrier semantics, where \( 1..n-1 \) threads wait until thread \( n \) releases the others and then proceeds to \( P_b \) itself, where it does not enter the loop as \( g\text{sense} \neq l\text{sense} \). Notice that \( l\text{count} \) and \( \text{newcount} \) are local variables with thread-specific values while \( g\text{sense} \) is a global variable shared between threads.

\textbf{Base:} Let us assume that \( l\text{sense} = 1 \). For \( n = 2 \), thread 1 is first to successfully execute CmpXchg (such that \( \text{updatecount} = \text{oldcount} \)), i.e., its \( \text{newcount} = 1 = I_b(1) \) (as \( g\text{count} \) is incremented by 1) such that it proceeds to \( P_b \) eventually. Thread 2 will succeed in CmpXchg later so that its \( \text{newcount} = 2 = n = I_r(2) \), i.e., it will get to \( P_r \) and set \( g\text{sense} \) to the inverse of \( l\text{sense} \). This releases thread 1, which will eventually proceed past \( P_b \) and allows thread 2 to bypass the loop at \( P_b \) so that both thread exit the barrier. (The argument for \( l\text{sense} = -1 \) is symmetrical with decrements over \( g\text{count} \), where thread 2 eventually reaches \( P_r \) with \( \text{newcount} = 0 \).) In fact, \textsc{BarrierFinder} has already proved that these invariants hold as part of the states at \( P_b, P_r \) considered during execution interleaving, including the correct barrier semantics of leaving only after all threads have arrived, by exhaustive state enumeration.

\textbf{Hypothesis:} For \( n \) threads, let us assume that \( I_b(i) \) holds for all \( i = 1..n-1 \) threads and \( I_r \) holds for thread \( n \), including correct barrier semantics upon proceeding past these program points.

\textbf{Step:} For \( n+1 \) threads (and \( g\text{sense} = 1 \)), consider two cases.

1. Let the last one to succeed with the CmpXchg be thread \( n+1 \). For threads \( 1..n-1 \), the hypothesis established \( I_b(n) \) at \( P_b \) with increasing \( \text{newcount} \) values as \( g\text{sense} = 1 \). Thread \( n \) is now the second to last one to succeed in CmpXchg, so \( \text{newcount} = n = I_r(n) \), which causes this thread to bypass \( P_r \) and proceed toward \( P_b \), where it would busy wait due to \( g\text{sense} = l\text{sense} \). And for thread \( n+1 \), as the last one to succeed in CmpXchg, \( \text{newcount} = n = I_b(n+1) \) with \( l\text{sense} = g\text{sense} = 1 \), reaching \( P_r \) to invert \( g\text{sense} \) before reaching \( P_b \) without entering the loop \( g\text{sense} \neq l\text{sense} \).

2. Let thread \( n+1 \) be any base the last thread to succeed in CmpXchg. Without loss of generality, let thread \( n+1 \) succeed as the \( m \)-th thread in CmpXchg. Then there are threads \( i = 1..m-1 \) who succeeded in CmpXchg before and, with increasing \( l\text{sense} \), are proceeding to \( P_b \) under \( I_b(i) \) by the
hypothesis. For thread \( n+1 \), \( newcount = m = I_b(m) \), so it proceeds toward \( P_b \). Threads \( j = m+1..n \) succeed in CmpXchg in the respective indexed order next, i.e., their respective \( newcount = j = I_r(j) \). The last one to succeed in CmpXchg, say thread \( l \), has \( newcount = n + 1 = I_r(n+1) \) and proceeds to \( P_r \) inverting \( gsense \) and then to \( P_b \) bypassing the loop as per hypothesis.

This establishes the correct barrier semantics upon continuing past \( P_b \) for all threads. The cases for \( gsense = -1 \) are symmetrical (with decrements per thread succeeding in CmpXchg). Furthermore, alternating \( gsense \) signs upon successive barriers of \( n+1 \) threads establish the same barrier semantics as for \( n \) threads, i.e., only after \( P_r \) is reached by the last thread in the previous barrier may all threads proceed to enter the next barrier, where they then enter in increasing/decreasing \( newcount \) order for \( gsense = 1/gsense = -1 \). Any thread still at \( P_b \) of the previous barrier may proceed as their local \( lsense \neq gsense \) while other threads already in the next barrier set \( lsense \) to \( gsense \) (line 13), so that they eventually spin in line 28 at \( P_b \), other than the last thread.

### 4.4 Related work

#### 4.4.1 Synchronization Characterization and Detection

Several empirical studies related to synchronizations have been performed. Xiong et al. [Xio10] characterize ad hoc synchronizations of representative open-source applications and find they are pervasive. Pinto et al. [Pin15; Wu16] survey real-world C++ and Java programs to assess how programs are synchronized in practice with concurrent language features, concurrent libraries, or concurrent data structures. Gu et al. [Gu15] investigate how programmers change program synchronizations and their relation to concurrency bugs. The results of these studies motivate us to work on accurate synchronization understanding and guide the design of our approach.

Specifically, on ad hoc synchronizations, existing techniques [JT10; JT14; Tia08; Tia09; Xio10; Yin13; Yua13] use either static or dynamic approaches to detect sync pairs. We proceed further to detect complete synchronizations and recognize enforced synchronization relationships.

#### 4.4.2 Runtime Invariant Detection

Our approach recognizes ad hoc barriers by mining execution traces for temporal invariants. Invariant mining is a technique pioneered by Daikon [Ern07], and our approach shares many common elements with Daikon, e.g., generating concrete traces and mining traces for invariants. Similar to our work, researchers have also explored temporal invariant mining for different purposes. Beschastnikh et al. [Bes11] propose techniques to mine temporal invariants based on partially ordered logs, and CSight [Bes14] further uses temporal invariants to model concurrent systems. CloudSeer [Yu16] uses temporal invariants to model the workflow of cloud systems and then uses the models for monitoring purposes. Instead, we focus on inferring the synchronization relationship of ad hoc synchronizations.
4.5 Conclusion

This paper contributes BARRIERFINDER, a pipelined framework to automate the recognition of complex ad hoc synchronizations that realize barriers. The experimental evaluation shows that BARRIERFINDER is able to detect barriers in 6 SPLASH2 benchmarks efficiently. BARRIERFINDER is also helpful for programmers to verify the correctness of counter-based barriers and is able to characterize other ad hoc synchronizations like allAB.
5.1 Conclusions

Extracting the peak performance of scientific applications becomes increasingly challenging on modern HPC architectures. Novel auto-tuning technologies that can improve HPC program performance and synchronization validation techniques that can detect and verify the correctness of ad hoc synchronization constructs within HPC programs are in high demands at DOE labs.

In this work, we first elaborated our thesis hypothesis, \( H \), and its three sub-hypotheses, \( H_1, H_2, H_3 \), in Chapter 1.3. Then, we took an experimental approach to verify the validity for each of them. Specifically, we made the following findings:

1. We designed and implemented a per-loop auto-tuning framework, FuncyTuner, which utilizes per-loop profiling information to guide fine-grained random search (CFR). Our extensive evaluation on a suite of modern scientific programs shows that FuncyTuner outperforms the state-of-art approaches by 4.5% to 10.7% (geometric mean) relative to O3 baseline. This shows \( H_1 \)

2. We designed and implemented a novel multi-input auto-tuning framework, CodeSeer, that discovers three types of input sensitivity for our benchmark HPC programs: (1) the optimized executable produced for a target program by prior auto-tuning techniques remains sensitive to the tuning input (type-I sensitivity); (2) the performance benefit of the optimized executable reported by prior techniques may not generalize to unseen inputs in practice (type-II sensitivity); and (3) the per-input best speedup may depend on different code variants (type-III sensitivity). The CodeSeer machine learning model is capable of dynamically selecting the
best per-input code variant with 90% and 92% precision (geometric mean) on balanced and unbalanced testing sets, respectively, while the prediction overhead is less than 0.01 seconds. This shows H2.

3. We built a pipelined framework, BarrierFinder, to automate the recognition of complex ad hoc synchronizations that realize barriers. The experimental evaluation shows that BarrierFinder is able to effectively detect barriers in 6 HPC benchmarks in SPLASH2 with high efficiency. BarrierFinder is also helpful for programmers to verify the correctness of counter-based barriers and is able to characterize other ad hoc synchronizations like allAB. This shows H3.

In conclusion, we showed that our over-arching hypothesis (see H in Chapter 1.3) holds by showing that its three sub-hypotheses hold.

### 5.2 Future Work

In this work, we established the new state of the art in program performance auto-tuning technologies. We also provided innovations for the practical techniques in detecting and validating complex ad hoc synchronizations. However, there are still limitations that deserve extensive future investigations:

1. **Overhead reduction techniques for FuncyTuner and CodeSeer:** The overhead in our experiments may be too high for wide adoption in practice. Overhead reduction via machine learning techniques may not only benefit many other HPC applications and larger scale experimental exploration for FuncyTuner and CodeSeer, but also make it possible to try other tuning approaches, such as genetic algorithms [Lin08]. Moreover, to address type-I sensitivity, CodeSeer evaluates many tuning inputs, but it may be possible to greatly reduce this overhead by looking into static features of hot computation kernels.

2. **FuncyTuner and CodeSeer auto-tune all the kernels of HPC programs simulataneously.** However, compared to other work that focuses on tuning a single kernel [Hal10; Yi12; Bag19], it lacks more flexibility to take advantage of aggressive source code transformations. Furthermore, compared to other low-level approaches [Wen18; DB04] and compiler optimizations [Ste03; BA06; Sch14; Pho16], our approach does not utilize any architectural information. These performance engineering technologies are competing with each other, and it is interesting to see how they interact and generalize to other modern accelerator architectures [Che18; TC17].

3. **BarrierFinder** currently employs exhaustive enumeration for synchronization validation and can detect deadlocks of erroneous barrier implementations. BarrierFinder could be extended to detect data races in OpenMP-based HPC program [Lia17] and other bugs.
BIBLIOGRAPHY


---

75


