ABSTRACT

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 naive 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 lightweight 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.

KEYWORDS
per-loop, fine-grained, auto-tuning, ICC, compiler, optimization, profile, OpenMP, HPC, scientific simulation

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 [7]. 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 [14]. 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 [7] or prediction [6, 22].

Several algorithms have been proposed for generating compiler flag combinations to create code variants, e.g., predictive machine learning models [4, 6, 13, 20] and optimization flag correlation-based combined elimination [21]. They perform compilation on a per-program basis. However, their effectiveness is limited. While machine learning based predictive models [6] perform well on small training datasets, a recent study [10] shows that their prediction accuracy drops dramatically for large-scale datasets and exhibits close to random behavior.

Combined elimination (CE) [21] takes advantage of interactions among compiler flags to find the best combination. However, our evaluation of CE, shown in Fig. 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.

Fine-grained per-region compilation techniques [11, 21, 22, 25] divide a program into different compilation modules and optimize...
each separately. Specifically, source-code level auto-tuners [11, 25] focus on a single simple computation kernel without considering module interactions in real-world applications. As prior work in compiler flag selection [21, 22] 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 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. 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 [5] 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.
- We conduct an in-depth case study for Cloverleaf [24] on the Intel Broadwell architecture 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 [5] 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 DESIGN OF FUNCYTUNER

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

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 2.3e13 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 $CV = (f_1 = f_{k1}, f_2 = f_{k2}, ..., f_N = f_{kn_N})$, 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.

Given a program $P$, a traditional compilation model treats all source files as a single compilation module $M$, within which the source files are compiled with the same $CV$. In contrast, FuncyTuner divides program $P$ into $J$ compilation modules $M_1, M_2, ..., M_J$ ($J$ 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 CVs to obtain the best performance due to their diverse code structures.

However, to identify the best CVs, the primary challenge is that the new search space size $COS_{new}$ increases significantly from $C_0$ to $C_1 = C_0^J$. 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 [10]. 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.

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.

2.2.1 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 $P$. As shown in Fig. 2, in step ①, 1000 CV samples are randomly selected from COS. In step ②, each CV is used to compile $P$ to obtain a code
which are executed to collect runtimes $T_{1000}$ times. In step 5, 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 $C_0$.

Figure 2: Per-program random search. $CV_k$ and $P_k$ with minimal runtime $T_k$ is its final result.

2.2.2 Per-function Random Search. As shown in Fig. 3, 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 $T_1, \ldots, T_{1000}$. FR reports the code variant with the minimum runtime as the best version.

FR uses Caliper [5] 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. 4. As in FR, a program $P$ is first divided into $J$ compilation modules. In step 2, FuncyTuner instruments modules via Caliper’s light-weight [5] APIs to measure per-loop runtimes. Then, in step 3, 1000 pre-sampled $CV_1, \ldots, CV_{1000}$ 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 2.2.3) and Caliper-guided random search (see Section 2.2.4).

Figure 3: Per-function random search. Step 3 is performed 1000 times.

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 $M_i$ such that $i = \text{argmin}_k \{ T_{jk} | 1 \leq k \leq 1000 \}$, and then links all modules to produce the target executable.

2.2.4 Caliper-guided Random Search. Similar to $G$, Caliper-guided random search (denoted as CFR, see Algorithm 1) 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, CFR prunes the pre-sampled search space for each hot loop before re-sampling per-loop CVs (line 13 of Algorithm 1). 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 ($1 < X <= 1000$) 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 improve 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 [21, 22].

3 EXPERIMENTAL DESIGN

In order to evaluate the efficacy of schemes presented in Section 2, we have performed experiments for seven modern scientific benchmarks on three architectures. This section describes the setup used in our experiments.
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. Our benchmark suite consists of seven HPC programs: AMG [17], LULESH [17], Cloverleaf (CL) [24], 351.bwaves, 362.fma3d, 363.swim and Optewe [23]. 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. Details of the OpenMP configurations are presented in Table 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, input sizes and time-steps have been adjusted so that every single run is less than 40 seconds for the O3 baseline compilation. In the first experiments (Section 4.1 and Section 4.2), we use the same inputs for tuning and testing, whereas we evaluate the impact of different inputs (Section 4.3) in later experiments to reflect typical usage patterns of repeated HPC application runs with different scientific inputs.

### 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 $f_{i1}, f_{i2}, \ldots, f_{in_i}$ 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.

### Table 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>Fortran</td>
<td>14.3k</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>6.2k</td>
<td>Mechanical simulation</td>
</tr>
<tr>
<td>363.swim</td>
<td>Fortran</td>
<td>8.3k</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>

### Table 2: Platform overview, runtime configurations, and benchmark inputs.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Processor</th>
<th>AMD</th>
<th>Intel Sandy</th>
<th>Intel Broadwell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opteron</td>
<td></td>
<td>6128</td>
<td>2650</td>
<td></td>
</tr>
<tr>
<td>Xeon E5-2620 v4</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Sockets</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>NUMA nodes</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Cores/Socket</td>
<td>16</td>
<td>16</td>
<td>16</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>2.60</td>
<td>2.60</td>
<td>2.1</td>
<td></td>
</tr>
<tr>
<td>Memory size [GB]</td>
<td>4</td>
<td>4</td>
<td>8</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 procl</td>
<td>[0-15]</td>
<td>[0-15]</td>
<td>[0-15]</td>
<td></td>
</tr>
<tr>
<td>LULESH: sizes, steps</td>
<td>200, 30</td>
<td>200, 30</td>
<td>200, 60</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>18</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Optewe: sizes, 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>
3.3 Loop Outlining and Caliper Instrumentation
To identify hot loops that need to be outlined into individual modules, FuncyTuner uses Caliper [5] 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.

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 if there is pairwise independence among different compilation modules.

4 RESULTS AND ANALYSIS
In this section, we first present results of the four algorithms in Sec. 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.

4.1 Overall Performance Comparison
Fig. 5 compares the performance for the four algorithms in Sec. 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. 3.4. Note that Random is applied on the original benchmarks while others are all applied on the benchmarks 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. 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. 5a) 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 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.

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.

4.2.1 The State-of-the Art. Prior techniques are either search-based [2, 22] or predictive modeling-based [4]. In particular, Cere [22] 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. 5 show that this often degrades performance for the Intel compilers.

OpenTuner [2] 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 [4], 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 [8]. Specifically, we select the top 100 performant CV’s out of 1000 random CV samples for each cBench application to extract their static and dynamic features with Milepostge [9] and Mica [12]. 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
Figure 5: 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.

Figure 6: FuncyTuner provides better performance than all variants of COBAYN (static, dynamic, hybrid), PGO, and OpenTuner.

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 `-openmp -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. Afterward, the programs are recompiled with `-O3 -openmp -fp-model source -prof-use -prof-dir/app`.

4.2.2 Observations. We make the following observations from the experimental results shown in Fig. 6.
(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** [4] and other previous research [1, 6, 10]. 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 [12] only works with serial code while our target benchmarks are parallel.

(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 [5], 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 [9] and Mica. Nevertheless, their performance is inferior to **FuncyTuner CFR** and their robustness is limited.

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. 4.1 and Sec. 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. 8 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. 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. 7, 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 0.1 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, 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 [6, 22] or by exploiting program-specific **CFR** convergence trends, i.e., **CFR** finds the best code variant in tens or several hundreds of evaluations.

4.4 **Deep Dive: Cloverleaf On Broadwell**

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. Five hot loops of Cloverleaf are selected since they have comparatively high per-loop runtime ratios (see Table 3, others are less than 3.0%) and were found to produce large performance differences (see Fig. 9) 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.

4.4.2 **Observations**

Fig. 9 shows the per-loop performance results for the five Cloverleaf hot loops on Broadwell. After greedy elimination, **Random**, **COBAYN** and **OpenTuner** retain -gopt-streaming-stores=always -no-ansi-alias -ipo -xCORE-AVX2 as their critical flags; **FuncyTuner CFR** retains -no-vec for *d* and mom9 but no special flags for the other three loops; *G.realized* also has no special flags for mom9. Table 3 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, cell3 and 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 *d* with 256-bit vectorization, the performance is 12.8% worse than a scalar version.
Figure 7: 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 8: For Cloverleaf on Broadwell, FuncyTuner CFR provides a stable performance benefit than all others while scaling from 100 to 800 time-steps.

Figure 9: Normalized speedups for top-5 loops of Cloverleaf on Intel Broadwell. Note: COBAYN (static), OpenTuner and Random generate the same code.

Inspection of assembly code shows that there are many data permutations and mask operations to handle control flow divergence, which are known to degrade vectorization efficiency.

(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 -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 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.

Table 3: 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>d1</th>
<th>cell3</th>
<th>cell7</th>
<th>mom9</th>
<th>acc</th>
</tr>
</thead>
<tbody>
<tr>
<td>G.realized</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>256</td>
</tr>
<tr>
<td>G.Independent</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>256</td>
</tr>
<tr>
<td>O3 baseline</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>256</td>
</tr>
<tr>
<td>Random</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td>256</td>
<td></td>
</tr>
<tr>
<td>CFR</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td>IS</td>
<td>256</td>
</tr>
</tbody>
</table>
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.

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. We divide prior work into the following two categories:

(1) Compiler flag selection techniques [6, 7, 14, 15, 19, 21, 22]: 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 [7, 14, 15, 22] belong to this category. [21, 22] 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 effective for their case studies but results in worse performance than random search for OpenMP-based scientific applications. As the state-of-the-art search-based auto-tuning technique, OpenTuner [2] coordinates many different search algorithms. Furthermore, to reduce the overhead of search-based approaches, researchers have also proposed schemes based on machine learning techniques [1, 4, 6]. As the state-of-the-art, COBAYN [4] 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 [3, 13, 15, 16]: given a set of compiler optimization passes, there are many viable 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.

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 region/sloops 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.

ACKNOWLEDGMENTS

This work was funded in part by NSF grants 1058779, 1217748 and 1525609, by US Air Force, Office of Scientific Research grants AFOSR-F9550-12-1-0442 and AFOSR-F9550-17-1-0205, and under the auspices of the U.S. Department of Energy by Sandia National Laboratories subcontract DOE-1403482 and Lawrence Livermore National Laboratory (LLNL) under contract DE-AC52-07NA27344.

REFERENCES