Exploiting Community Structure for Floating-Point Precision Tuning

Hui Guo
Department of Computer Science
University of California, Davis, USA
higuo@ucdavis.edu

Cindy Rubio-González
Department of Computer Science
University of California, Davis, USA
crubio@ucdavis.edu

ABSTRACT

Floating-point types are notorious for their intricate representation. The effective use of mixed precision, i.e., using various precisions in different computations, is critical to achieve a good balance between accuracy and performance. Unfortunately, reasoning about mixed precision is difficult even for numerical experts. Techniques have been proposed to systematically search over floating-point variables and/or program instructions to find a faster, mixed-precision version of a given program. These techniques, however, are characterized by their *black box* nature, and face scalability limitations due to the large search space. In this paper, we exploit the community structure of floating-point variables to devise a scalable hierarchical search for precision tuning. Specifically, we perform dependence analysis and edge profiling to create a weighted dependence graph that presents a network of floating-point variables. We then formulate hierarchy construction on the network as a community detection problem, and present a hierarchical search algorithm that iteratively lowers precision with regard to communities. We implement our algorithm in the tool HiFPTuner, and show that it exhibits higher search efficiency over the state of the art for 75.9% of the experiments taking 59.6% less search time on average. Moreover, HiFPTuner finds more profitable configurations for 51.7% of the experiments, with one known to be as good as the global optimum found through exhaustive search.

CCS CONCEPTS

• **Software and its engineering** → *Dynamic analysis*; *Software notations and tools*; *Software verification and validation*;

KEYWORDS

floating point, precision tuning, dependence analysis, community detection, hierarchical search

ACM Reference Format:
Hui Guo and Cindy Rubio-González. 2018. Exploiting Community Structure for Floating-Point Precision Tuning. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3213846.3213862

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213862

1 INTRODUCTION

The use of numerical software has grown rapidly over the past few years. From computer games to safety-critical systems, a large variety of applications today make extensive use of floating point. Unfortunately, floating-point computation is notoriously unintuitive and floating-point programs suffer from a large variety of accuracy problems including extreme sensitivity to large roundoff errors and incorrectly handled numerical exceptions [20]. This has led to software bugs that have caused catastrophic failures [6, 7, 34]. To strengthen the accuracy of floating-point programs and therefore evade potential failures, a common practice is to use the highest available precision. Higher precision provides higher magnitude of representable numbers with increased precision. However, using the highest precision uniformly through the program can have a negative effect in performance.

Techniques [16, 19, 24, 30, 31] have been proposed to assist programmers in exploring the trade-off between floating-point accuracy and performance. Their main goal is to improve performance by reducing precision with respect to an accuracy constraint. With regards to attaining optimally decreased precision, such techniques are divided into two categories: (1) using a static performance and accuracy model to formulate precision tuning as an optimization problem, and (2) dynamically searching through different precisions to find a local optimum. Because of the limitation of error-analysis techniques required in the model, static approaches aim at tuning floating-point expressions rather than programs as a whole. On the other hand, dynamic approaches are able to tune medium-sized programs, but require to explore large search spaces, and the solutions can deviate significantly from the global optima.

In this paper, we present a community detection method for floating-point variables, and exploit it on dynamic precision tuning to improve both scalability of the search, and the quality of the proposed solutions. Unlike existing dynamic tuning approaches, which treat the program as a *black box*, we analyze the source code and its runtime behaviors to identify dependencies among floating-point variables, and to provide a customized hierarchical search for each program under analysis.

Our algorithm improves scalability by reducing *search effort*. We refer to search effort as the number of configurations that the search algorithm explores during precision tuning. Because each configuration requires to run the program to test accuracy and measure performance, reducing the number of configurations leads to a more scalable search algorithm. The *quality of a configuration* is measured in terms of the degree to which precision is decreased, and the program speedup achieved after tuning. A common mistake is to regard these two as equivalent when in reality program speedup can be affected by a variety of factors. One such factor is the number

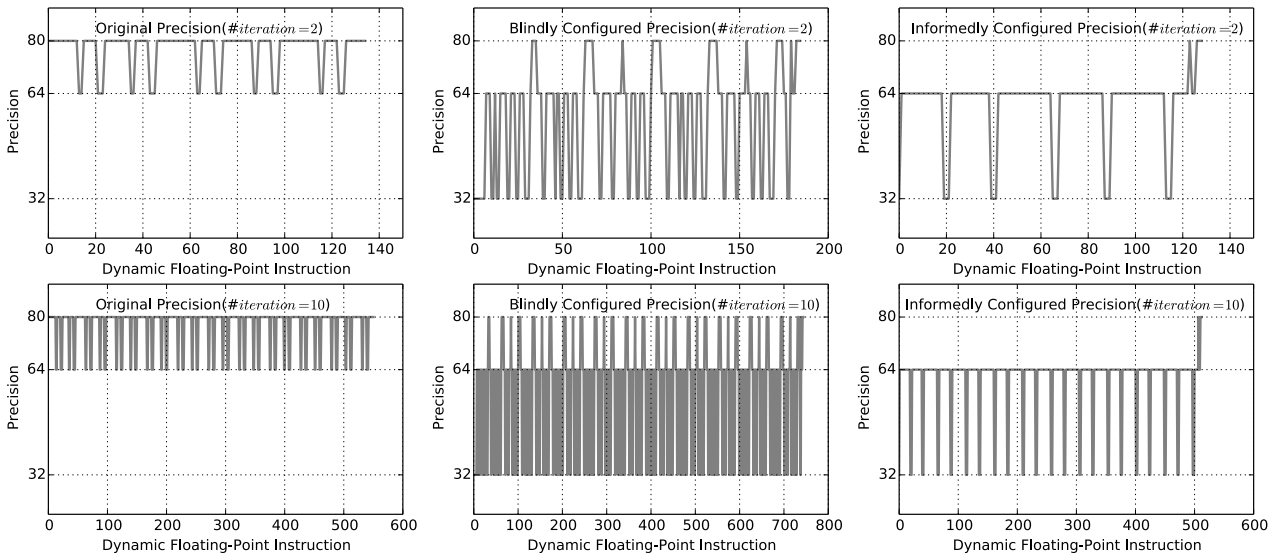


Figure 1: Frequency of precision shifting during the execution of a floating-point program implementing the Simpson's rule (left column), a tuned version of the program produced by a blind search (middle column), and a tuned program version produced by an informed search (right column). The two graphs in each column correspond to the same program when repeating a loop 2 times (top) and 10 times (bottom).

of type casts introduced when variables used in an expression have different floating-point types. In this paper, we focus on program speedup when evaluating precision configurations.

The main insight is that avoiding frequent shifts in precision at runtime contributes to better overall performance. By doing so, we amortize the performance overhead imposed by mixed precision due to type castings, and potentially enable optimizations that target uniform precision such as vectorization. To achieve this, we analyze the dependencies among variables (read-after-write dependencies), and gradually leverage these dependencies to guide the search. We refer to this as an *informed* search. Conversely, a *blind* search does not take variable dependencies into consideration, thus proposing configurations that may not improve performance as much.

As an example, Figure 1 plots the precision of executed instructions in three programs: a sample floating-point program that approximates the definite integral of $sin(\pi x)$ on interval $(0, 1)$ using the Simpson's rule [25], a tuned version of this program produced by a blind-search approach, and a tuned version for which an informed search is used. The initial program uses *long double* (80 bits) and *double* (64 bits) precision, and contains a loop. The left column in Figure 1 shows two graphs for the initial program when the loop iterates 2 times (top row) and 10 times (bottom row), respectively. As the number of iterations increases, we observe more shifting in precision. The middle column shows that the tuned program produced by the *blind* search uses lower precision — *float* (32 bits) — for some instructions, but the shifting in precision increases significantly. On the other hand, the tuned program produced by the *informed* search, shown in the right column, shifts precision less often during execution. Moreover, the program avoids the use of *long double* precision except for two cast instructions introduced by a long-double format specifier in a `print` statement at the end of the execution. Overall, the informed search proposes tuned programs that use precision in a more consistent manner, i.e., reducing the frequency of precision shifting at runtime.

In this paper, we present the first informed search for floating-point precision tuning. Specifically, we create a variable community hierarchy based on the dependence analysis for the target program, which groups dependent variables that may require the same level of precision. A hierarchical search is deployed on the community structure to informatively prune the search space, and explore promising precision configurations. We implement our hierarchical search in a tool named HiFPTuner, short for Hierarchical Floating-Point precision tuner. We evaluate HiFPTuner on nine floating-point programs and compare it against the state of the art. The experimental results show that HiFPTuner is effective at finding precision configurations in significantly fewer runs. In particular, we find that for 22 out of 29 experiments, HiFPTuner finds configurations that lead to equivalent or larger program speedups than the state of the art while exploring 59.6% fewer configurations. Furthermore, HiFPTuner finds more profitable configurations for 51.7% experiments. This paper makes the following contributions:

- We present an algorithm to explore the community structure of variables in floating-point programs (Section 3).
- We investigate the use of community structures for precision tuning, and present a hierarchical search algorithm that exploits these community structures (Section 3).
- We implement our algorithm in the tool HiFPTuner, and demonstrate its effectiveness in terms of search efforts and quality of proposed configurations (Section 4).
- We provide a detailed comparison of HiFPTuner and the state of the art (Section 4).

We present our motivation in Section 2, discuss the limitations in Section 5 and related work in Section 6, and conclude in Section 7.

2 MOTIVATION

In this section, we present an illustrative example to describe the limitations of state-of-the-art search-based precision tuners and motivate our hierarchical search algorithm based on the community structure of floating-point variables.

2.1 An Illustrative Example

Precision tuning considers all floating-point variables declared in functions defined in the program, as well as calls to library functions for which lower-precision implementations exist. For example, program *simpsons* in Figure 2 (first discussed in Section 1) declares nine *long double* variables — three variables in function `fun` (p, pi, and q) and six variables in function `main` (a, b, h, s, x, and $fuzz$) — and makes two calls to *double* precision library functions `acos` and `sin`, for which there exist single-precision implementations `acosf` and `sinf`, respectively.

```
1  long double fun(long double p) {
2     long double pi = acos(-1.0);
3     long double q = sin(pi * p);
4     return q;
5  }
6
7  void main() {
8     long double a, b;    // endpoints of the interval
9     // subinterval length, integral approximation, x
10    long double h, s, x;
11    const long double fuzz = 1e-26; // tolerated error
12    const int n = 2000000;
13    a = 0.0;
14    b = 1.0;
15    h = (b - a) / n;
16    x = a;
17    s = fun(x);
18  L100:
19    x = x + h;
20    s = s + 4.0 * fun(x);
21    x = x + h;
22    if (x + fuzz >= b) goto L110;
23    s = s + 2.0 * fun(x);
24    goto L100;
25  L110:
26    s = s + fun(x);
27    printf("%1.16Le\n", (long double)h * s / 3.0);
28 }
```

Figure 2: Sample program *simpsons*.

Configuration Space. Given the type candidate set {*float*, *double*, *long double*}, the size of the configuration space for *simpsons* is $3^9 \times 2^2 = 78,732$. If we assume that the average runtime of each configuration is 1 second, then evaluating all configurations would take 21.9 hours. While the number of configurations increases exponentially with the number of precision allocations (floating-point variable declarations and library calls), it would take about 110 years to evaluate all the possible configurations for a program with 20 floating-point variables in *long double*. Even when parallelizing this task, the search space is just too large.

A precision configuration is evaluated by executing the resulting program. Only if the program satisfies a given accuracy constraint and has better performance than the original program, then the configuration is evaluated as *valid*. The accuracy constraint, e.g., error threshold 10^{-4}, indicates the largest absolute value of the relative error of the result with respect to the result produced by the original program. For our example, the original program produces the result 6.3661977236756841e−01. A *valid* configuration that satisfies the accuracy constraint requires to be in range (6.365561103903317e−01, 6.372563921399359e−01).

Search-Based Precision Tuning. While evaluating every possible configuration in the search space is expensive, or even infeasible, the state of the art in precision tuning applies search-based strategies to tune floating-point programs. The search strategies currently adopted are general strategies such as random search and binary search. Floating-point variables and function calls for which alternative lower-precision implementations exist are listed in a randomly chosen order. The search then blindly explores the configuration space on the list to find a local minimum.

As an example, we show how a variant of binary search is used to tune the *simpsons* program. First, variable declarations and function calls are collected into a list:

```
p  pi  q  acos  sin  a  b  s  h  x  fuzz
```

While lowering precision for all elements leads to an inaccurate program (thus *invalid* configuration), the tuner divides the list into two partitions and lowers the precision for each partition at a time.

```
p  pi  q  acos  sin a | b  s  h  x  fuzz
p  pi  q  acos  sin a | b  s  h  x  fuzz
```

If either configuration satisfies the accuracy and performance constraints, the corresponding partition whose precision was lowered is temporarily removed from the search space (the variables are set to use the newly assigned precision), and the search starts again with the remaining elements. Otherwise, the search iteratively divides each partition until no partition is divisible. In our example, the search iteratively divides the list until there are 8 partitions, and lowering the 4^{th} partition leads to a *valid* configuration.

```
p  pi  q | acos  sin a | b  s  h | x  fuzz
...
p  pi | q | acos | sin  a | b  s | h | x | fuzz
.
p  pi | q | acos | sin  a | b  s | h | x | fuzz
```

In the configuration above, the precision of `sin` and variable a is lowered to *float* and *double*, respectively, and the configuration satisfies accuracy and performance constraints. Thus, we then attempt to further lower the precision of this new configuration. We set `sin` to *float*, and a to *double*, temporarily remove both items from the search space, and continue the search on the remaining elements:

```
p  pi  q  acos  b  s  h  x  fuzz
```

The list is minimized as elements are lowered and removed from the search space. A search phase terminates when none of the remaining elements can be lowered:

```
q  acos
```

Variable q and `acos` require their original precisions. Up to this point, the configuration is as follows:

```
        p  pi  q  acos  sin  a  b  s  h  x  fuzz
CONFIG: d  d   ld   d    f   d  d  d  d  d   d
```

Since there are variables in *double* precision that could potentially be lowered to *float*, we run a second search phase to continue the exploration. The final configuration for *simpsons* is:

```
         p  pi  q  acos  sin  a  b  s  h  x  fuzz
CONFIG1: f  d   ld   d    f   f  f  d  f  d   f
```

Figure 3: HiFPTuner overview.

CONFIG1 is the configuration applied to the tuned program whose plot is shown in the middle column of Figure 1. A speedup of 78.7% is observed as most of the *long double* variables are lowered to *double* or *float*. However, we also observe that by randomly reordering the initial list of elements, we find distinct final configurations that lead to higher speedups:

```
          acos p  sin  x  b  fuzz  a  q  h  pi  s
CONFIG2:   d   f   f   d  f   f     f  f  f  d   d

          p  h   s  x  sin  b  fuzz  q  a  pi  acos
CONFIG3:  d  d  ld  d   f   d   d    d  d  d   f
```

CONFIG2 has a speedup of 84.8% while CONFIG3 has a speedup of 85.7%. The ordering of variables affects how elements are grouped during the search, and thus the final result.

2.2 Hierarchical Search

Instead of blindly searching over a sequence of variables, our search algorithm uses a *hierarchy*, i.e., a community structure of variables, which groups variables likely to require the same precision based on their usage. In the hierarchy, a node on level i denotes a group of separate nodes on level $i-1$, therefore, a lower level describes the partition of the nodes on its upper level. A sample hierarchy for program *simpsons* (height = 3) is shown below:

```
    p pi q acos sin  | a b h x s fuzz       -- L2
  p pi q  |acos|sin| a b h x   |s|fuzz      -- L1
p | pi | q | acos | sin | a | b | h | x | s | fuzz  -- L0
```

As we can see in the example, level L2 of the hierarchy (top level) provides a grouping of the elements from level L1. Similarly, level L1 presents a grouping of the elements from level L0 (bottom level). The search starts from the top level of the hierarchy, explores distinct precision alternatives for each node (group) in the level as a whole, and then descends until reaching the bottom. The search is guided by the given groupings. The configuration space is reduced both at the top of the hierarchy (fewer groups), and as the search moves down the hierarchy. For the latter, if group $\{p, pi, q\}$ on level L1 is assigned *float*, for example, then each corresponding group $\{p\}$, $\{pi\}$, and $\{q\}$ on level 0 will be initialized as *float* before conducting the search. Therefore, other precision combinations for these groups, e.g., `f d d`, will not be explored during the search.

The intuition is that by applying a hierarchical search, we are in favor of assigning identical precision to a group of dependent variables, and groupings can be explicitly specified according to the program and a user's intention. To provide a hierarchy of variable groupings, we leverage the semantics of the program by analyzing the dependencies among floating-point variables. Frequently dependent variables are grouped early during the hierarchy construction, i.e., the groupings appear across most levels of the hierarchy. Thus, these highly dependent variables are more likely to be assigned the same precision as the search moves down the hierarchy. The hierarchical search based on the community structure above produces the CONFIG4 below[1]. This is the configuration applied to the tuned program whose plot is shown in the right column of Figure 1. As one can observe, precision shifting is less frequent, and a higher speedup of 90.0% is achieved. *Indeed, this is the best configuration we find through exhaustive search.* This is achieved by exploring only 24 configurations.

```
          p  pi  q  acos  sin  a  b  s  h  x  fuzz
CONFIG4:  d  d   d   d     f   d  d  d  d  d   f
```

3 TECHNICAL APPROACH

We present our hierarchical search algorithm for precision tuning. As shown in Figure 3, we first perform dependence analysis and edge profiling to create a weighted dependence graph of the program. Second, we apply community detection iteratively to construct a hierarchy from the weighted dependence graph. Finally, we apply our hierarchical search algorithm to tune the program. The rest of this section describes each step in more detail.

3.1 Dependence Analysis and Edge Profiling

We perform dependence analysis and edge profiling to create a weighted dependence graph. *Dependence analysis* summarizes the static floating-point arithmetic assignments as a dependence graph of variables, and *edge profiling* generates a weight for each dependency via program instrumentation.

Definition 3.1. A *weighted dependence graph* for a program is a directed graph $G(V, E, W)$ where the set of vertices V represents the floating-point variables in the program and an edge $u \to v \in E$ denotes that the value of u is used to compute the value of v in at least one arithmetic assignment, while $W : E \to \mathbb{N}$ implies how many times the assignments associated with the edge are executed.

Consider program *simpsons* in Figure 2. Statement 3 in function `fun`, and statements 15-17, 19-21, 23 and 26 in function `main` are analyzed and instrumented to construct the dependence graph. Two dependence edges, from variable *pi* to *q* and variable *p* to *q*, respectively, are added based on statement 3, and their weights are the number of times the statement is executed. Figure 4 shows the weighted dependence graph for program *simpsons*. Each subgraph describes a function in the program. Note that the analysis is intraprocedural—we do not connect actual parameter variables and formal parameter variables in a function call statement, e.g., variable *x* in statement 20 and its corresponding formal parameter variable *p* in function `fun`. The generated dependence graph does not include library function calls.

[1]The hierarchy used in the experiments to produce CONFIG4 consists of levels L1 and L0. Here we add an auxiliary level L2 just for illustration. Note that this additional level does not affect the final result.

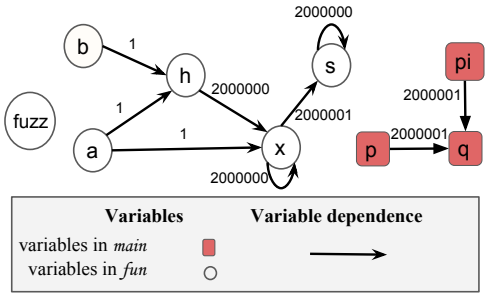
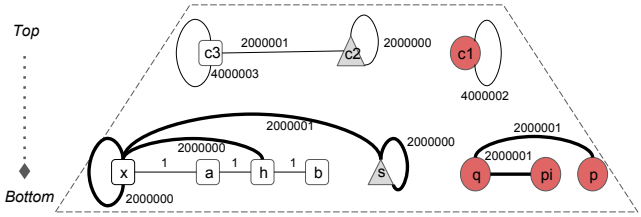
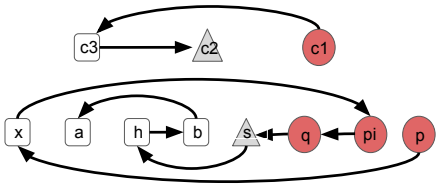


Figure 4: Weighted dependence graph for program *simpsons*.



(a) Hierarchy structure.



(b) Ordered hierarchy structure.

Figure 5: Hierarchy structures for program *simpsons*. Figure 5a shows the hierarchy structure. The bottom level has eight nodes, each representing a separate floating-point variable in the program. Each of the three nodes in the top level represents a community from the bottom level (shown by node shape and color). Edge labels indicate weights. Figure 5b shows the corresponding ordered hierarchy. For simplicity, we omit the unconnected variable *fuzz*.

3.2 Hierarchy Construction

A *hierarchy* is the structure used to leverage the various degrees of dependence among variables. This structure provides hierarchically pre-grouped variables for the search algorithm to informatively explore during precision tuning. The rest of this subsection describes the two steps followed to construct a hierarchy based on the weighted dependence graph.

3.2.1 Iterative Community Detection. We formulate the problem of grouping variables as a community detection problem [22, 29] in networks. A *community* is a subgraph containing nodes that are more densely linked to each other than to the rest of the graph. We use communities to represent sets of floating-point variables that are likely to require identical precision. The weighted dependence graph, extracted from the program and its runtime behaviors, provides a network of floating-point variables in which an edge and its weight indicate the degree of dependence between two variables. By iteratively applying community detection, the dependence among variables is leveraged into a hierarchy.

The hierarchy is constructed from the weighted dependence graph without considering the direction of the edges. The construction starts with a *bottom level* in which each variable is in a separate group, and iteratively adds new levels by (1) applying community detection on the current top level's dependence graph, and (2) using the community detection results to reconstruct the dependence graph for the new top level. The dependence graph is reconstructed by collapsing the nodes within a same community.

The problem of community detection is computationally hard given that the number of communities at each level is unknown, and that the communities can be in various densities and sizes. This problem has been widely explored in the field of networks and is not the focus of this paper. We borrow an existing algorithm among the most popular approaches today, modularity maximization [26, 27], to solve our problem. Modularity Q measures the density of edges within communities as compared to the expected density if the edges were distributed at random. It is defined as,

$$Q = \frac{1}{2m}\sum_{i,j}(A_{ij} - \frac{k_i k_j}{2m})\delta(c_i, c_j) \quad (1)$$

where A_{ij} represents the weight of the edge between i and j, and k_i, k_j are degrees of nodes i and j, $m = \frac{1}{2}\sum_i k_i$ is the total number of edges in the network, thus $\frac{k_i k_j}{2m}$ is the expected weight of the edge between nodes i and j if edges are randomly distributed, c_i is the community to which node i belongs, and the δ function $\delta(x, y)$ returns 1 if $x = y$ and 0 otherwise. In the modularity maximization algorithm, different community assignments are explored in order to obtain maximum modularity. In this paper, we use an existing implementation of this algorithm, *Fast Unfolding* [14].

Consider the dependence graph for *simpsons* in Figure 4. Initially, each of the 9 variables is placed in a distinct group.

```
p | pi | q | a | b | s | h | x | fuzz
```

We compute the value of Q using the current groups and attempt to optimize it by moving one variable from its current group to the group of one of its neighbors in the dependence graph. For example, as variable q is linked to variables p and pi in Figure 4, we randomly select p, and move q to p's group.

```
p  q | pi | a | b | s | h | x | fuzz
```

This increases the value of Q, and we keep moving distinct variables around until no movement of a variable can further optimize Q. When community detection is finished, the identified communities are added to the top of the hierarchy to form a new level $i + 1$. This new level captures the communities identified in level i, and their reconstructed dependence graph. Community detection is iteratively applied on the current top level of the hierarchy until a maximum of modularity is attained. In our example, this is achieved with two levels, and the communities at the top level are:

```
p  q  pi | a  b  h  x | s | fuzz
```

Figure 5a shows a graphical illustration of the hierarchy for *simpsons*. Note that the dependence graph has been reconstructed in the top level, and weights have been updated accordingly.

3.2.2 Hierarchy Ordering. Hierarchy ordering consists of ordering the items in each level of the hierarchy to follow the dependence

Algorithm 1 Hierarchical Search Algorithm

Require: Program P, Hierarchy of variables $H[\,]$, Initial Config C_p
Ensure: Find local minimum configuration C_{opt}

```
1.  Search space S ← SearchSpace(C_p)
2.  Result r_0, Performance p_0 ← RunConfiguration(C_p, P)
3.  C_opt ← C_p
4.  p_opt ← p_0
5.  for i ← len(H) − 1 to 0 do
6.     C_opt_i, p_i ← LevelSearch(H[i], C_opt, S, r_0, p_0, P)
7.     if p_i > p_opt then
8.        C_opt ← C_opt_i
9.        p_opt ← p_i
10.    end if
11. end for
12. return C_opt
13. procedure LevelSearch (communities, C, S, r_0, p_0, P)
14.    Community list Tu ← [ ]
15.    Search range list Sr ← [ ]
16.    for i in communities do
17.       Tu ← Tu+ unfold(i, C, S)
18.       Sr ← Sr+ unfold(i, C, S)
19.    end for
20.    Tu, Sr ← Tu, Sr+ Library function calls
21.    C_opt ← BaseSearch(Tu, Sr, r_0, p_0, P)
22.    return C_opt
23. end procedure
```

flow. For the bottom level, we find the topological order of the variables in the initial dependence graph generated by the dependence analysis, and edge profiling. We permute the nodes in the bottom level accordingly. For higher levels we create the corresponding community dependence graphs. The community dependence graph of level i is constructed by merging the nodes of the dependence graph at level $i-1$. Edges inside a community are removed. The ordered hierarchy for *simpsons* is shown in Figure 5b.

3.3 Hierarchical Precision Tuning Algorithm

Based on the ordered hierarchy, our algorithm starts searching from the top of the hierarchy down to the bottom. The resulting precision configuration at hierarchy level $i+1$ becomes the initial configuration for level i, thus reducing the search space. Configurations are evaluated in terms of the performance of the resulting programs. If no configuration at level i has better performance than its initial configuration, then the initial configuration is passed down as the configuration of level i. In such situations, the search performed at level i has no effect in the result; the configuration found at level $i+1$ is used instead. From our experiments, we observe than searching the bottom level rarely produces a better configuration. Thus, in practice, skipping the search at the bottom level can effectively improve search efficiency at the cost of missing further refinements on occasion. More details on this are discussed in our experimental evaluation in Section 4.

As described in Algorithm 1, we start by creating the search space S based on the initial configuration C_p of program P (line 1). We then apply the configuration C_p to program P, run the resulting program, and log both the result r_0 and its performance p_0 (line 2). These two values are used by `BaseSearch` (line 21) to evaluate whether candidate configurations satisfy the accuracy constraint and improve performance with respect to r_0 and C_p, respectively. We proceed through each level of the hierarchy (lines 5-11) to produce a level configuration by invoking `LevelSearch`, defined on lines 13-23. This procedure creates the community list and its corresponding search range list with respect to the communities on the hierarchy level (lines 14-20). Note that a community may contain variables in distinct search ranges (e.g., variable a in search range $\{f, d\}$, variable b in $\{f*, d*, ld*\}$). To simplify the search, communities are *unfolded* (lines 17-18). This means that a community can be broken down into two or more communities based on the search ranges of its elements. We add library function calls separately, i.e., we create a community for each of these calls (line 20).

Finally, `BaseSearch` (line 21) is applied on the list of communities and search ranges for each level. The base search algorithm iteratively proposes candidate configurations with lower precision, which are evaluated by `RunConfiguration` on accuracy and performance. The configuration with the best performance is returned by the algorithm. Similarly, the best configuration found across hierarchy levels is returned by the hierarchical search algorithm (see lines 7-10, 12). Note that the base search strategy is independent of the hierarchical algorithm, and can be interchangeable. Sample search strategies include random search, binary search, and genetic search. In the rest of this paper we adopt binary search to facilitate comparison with the state of the art.

4 EXPERIMENTAL EVALUATION

The main goal of this experimental evaluation is to answer the following research questions:

RQ1. How *efficient* is hierarchical search for precision tuning in comparison to the state of the art?
RQ2. How *effective* is hierarchical search in finding higher quality configurations than the state of the art?

We compare HiFPTuner against a dynamic non-hierarchical search algorithm, as implemented in the tool Precimonious [31].

4.1 Experimental Setup

We implement dependence analysis and edge profiling using the LLVM compiler infrastructure [4]. Hierarchy construction takes as input the weighted program dependence graph, and generates its hierarchy structure using the NetworkX [5] Python package[2]. The hierarchical search algorithm is implemented in Python, and uses Precimonious' program transformation LLVM passes to change the precision of a program given a configuration.

We evaluate HiFPTuner on three long-double programs: *simpsons* — an implementation of the widely used Simpson's rule [25] for integration in numerical analysis, *arclength* — a program first introduced by [11] and used as a benchmark for precision tuning in prior work [30, 31], and *piqpr* — an implementation of the "Bailey-Borwein-Plouffe" (BBP) algorithm [10] for computing digits of π beginning at the position of $0.3 * 10^8$ [1]. We also evaluate HiFPTuner on four double precision routines (*fft*, *gaussian*, *sum*, and *bessel*) from the GNU Scientific Library (GSL) [3], and two programs (*ep* and *cg*) from the NAS Parallel Benchmarks [12]. Table 1 describes each program in terms of lines of code (LOC), the initial

[2]NetworkX uses *Fast Unfolding* as the community detection module [2].

Table 1: Benchmark statistics.

Program	LOC	Initial L	D	F	C	Inputs
simpsons	40	9	0	0	2	Input-free
arclength	45	8	0	0	3	Input-free
piqpr	125	17	0	0	0	Input-free
fft	264	0	22	1	2	Coverage inputs
gaussian	461	0	56	0	2	Coverage inputs
sum	405	0	34	0	2	Coverage inputs
bessel	148	0	24	0	5	Coverage inputs
ep	260	0	13	0	4	Class A
cg	880	0	32	0	3	Class A

precision configuration, and testing inputs. The initial configuration of each program is in the form "L D F C", where L, D, and F correspond to the number of *long double*, *double*, and *float* variables declared in the original program, and C is the number of function calls considered for precision tuning. Programs *simpsons*, *arclength* and *piqpr* are input-free, the GSL benchmarks are tuned and tested over an input set that maximizes code coverage [31], and we use the provided input Class A for the NAS Parallel Benchmarks.

We annotate the programs with calls to utility functions to log results and measure performance, except for the NAS Benchmarks, which already include built-in accuracy checking routines. We use four error thresholds, 10^{-4}, 10^{-6}, 10^{-8}, 10^{-10} for tuning, which bound the relative error of the result. Because *ep* originally uses error threshold 10^{-8}, we tune the precision using equal or larger thresholds than 10^{-8}. Thus, we run a total of 35 experiments: 8 programs $\times$ 4 error thresholds + 1 program $\times$ 3 error thresholds. We run the experiments on a 2-core Intel E5-2676 2.4 GHz processor, 4GB RAM workstation with Ubuntu 16.04 LTS. Finally, we use the clang compiler with optimization level -O2 for performance evaluation, which is the default optimization used by our benchmarks.

We compare HiFPTuner against the state-of-the-art precision tuner Precimonious in terms of the efficiency of the search, and the quality of the proposed configurations. Precimonious performs a variant of binary search known as delta debugging [35], which guarantees to find a local 1-minimum if one exists. A configuration is said to be 1-minimal if lowering any additional variable (or function call) leads to a configuration that produces an inaccurate result, or is not faster than the original program. The algorithm's average time complexity is $O(n \log n)$, and worst case $O(n^2)$, where n is the number of elements to be tuned. For a fair comparison and to better evaluate the benefit of a hierarchical search over a non-hierarchical approach, we choose to use the same delta-debugging search as our base search algorithm. As noted in Section 3, the base search algorithm can be replaced by other search strategies.

4.2 Experimental Results

In search-based precision tuning, the number of configurations explored, i.e., run, by the search algorithm along with the total search time demonstrate the algorithm's efforts in finding a solution. However, the total search time is mainly dominated by the execution time of each explored configuration. Unfortunately, configuration runtime vary widely due to the performance disparity of instructions in different floating-point precisions. To preclude the effects of such runtime disparity, we simply use the number of explored configurations to represent the search efforts of the algorithm.

In Table 2, "Configs" gives the total number of configurations run by each tool to find the final configuration. For example, HiFPTuner runs a total of 24 configurations for program *simpsons* 10^{-10} while Precimonious runs 124 configurations. For HiFPTuner, we run a thorough search to the bottom of the hierarchy for the sake of completeness and provide the breakdown of configurations run at each level. However, as described in Section 3, exploring the bottom level rarely produces a better configuration, and thus can be skipped in general. In the case of *simpsons* 10^{-10}, 24 and 48 configurations are run at levels L1 (top) and L0 (bottom), respectively, and the top level proposes the final configuration (searching the bottom level does not produce a better configuration) as indicated by "Configs". Finally, Table 2 lists the final configurations in the form "L D F S", where L, D, and F are the number of *long double*, *double*, and *float* variables to declare in the tuned programs, and S is the number of function calls to switch to lower precision. "Speedup" gives the speedup observed for the tuned programs. We report the average speedup across five runs.

For 6 out of 35 experiments, *bessel* with error thresholds 10^{-10} through 10^{-4} and *fft* 10^{-6} to 10^{-4}, the programs satisfy both accuracy and performance constraints with the lowest precision and thus a search is not necessary. Table 3 shows the initial and final configurations for these programs, as well as the speedup observed. For the remaining experiments, HiFPTuner runs fewer configurations than Precimonious for the majority (24 out of 29) of the experiments. In 22 of these cases, HiFPTuner finds comparable or better configurations (see bold numbers under HiFPTuner "Configs" in Table 2). Moreover, HiFPTuner finds better configurations for 3 additional experiments beyond the 22 (e.g., *sum* 10^{-8} with 18% vs. 0% speedup). We use the following definitions to summarize HiFPTuner's results.

Definition 4.1 (Configuration Quality)**.** A configuration C_1 is better than a configuration C_2 if C_1 leads to a program with higher speedup than C_2.

Definition 4.2 (Search Efficiency)**.** A search strategy S_1 is more efficient than search strategy S_2 if S_1 finds an equivalent or a better configuration than S_2 faster.

Search Efficiency Result: HiFPTuner exhibits higher search efficiency over Precimonious for 75.9% (22 out of 29) of the experiments that require fine tuning (see bold numbers under HiFPTuner "Configs" in Table 2).

Definition 4.3 (Search Effectiveness)**.** A search strategy S_1 is more effective than search strategy S_2 if S_1's final configuration is better than S_2's final configuration.

Search Effectiveness Result: HiFPTuner finds better configurations for 51.7% (15 out of 29) of the experiments that require fine tuning compared to Precimonious (see bold numbers under HiFPTuner "Speedup" in Table 2).

4.2.1 Search Efficiency. HiFPTuner improves search efficiency by 59.6% on average over Precimonious for 22 out of 29 experiments that require fine tuning (see bold numbers under HiFPTuner

Table 2: Precision tuning results for error thresholds 10^{-10}, 10^{-8}, 10^{-6}, and 10^{-4}. The initial configuration is shown in the form "L D F C" where L, D, and F denote the number of *long double*, *double* and *float* variables declared in the original program, and C is the number of function calls to be tuned (e.g., *sin*). "Configs" show the *total* number of configurations run by each tool to find the final configuration. The columns under "Configs/Level" — L2, L1, L0— show the number of configurations run at each level of the hierarchy. "L D F S" gives the final configuration proposed by each tool. We modify each program according to the final configuration, and report the corresponding speedup in columns "Speedup". Note that the top level for programs *simpsons*, *arclength*, and *piqpr* is "L1". We use '-' as final configuration and speedup when a tool does not find a configuration that satisfies the accuracy constraint, and improves performance. We show in bold the number of configurations explored by the most efficient tool.

$E10^{-10}$	Initial				HiFPTuner									Precimonious					
						Configs/Level			Final Configuration						Final Configuration				
Program	L	D	F	C	Configs	L2	L1	L0	L	D	F	S	Speedup	Configs	L	D	F	S	Speedup
simpsons	9	0	0	2	**24**	n/a	24	48	0	8	1	1	**90.0%**	124	1	4	4	1	80.0%
arclength	8	0	0	3	**30**	n/a	30	39	0	7	1	1	4.9%	142	0	7	1	1	4.9%
piqpr	17	0	0	0	**66**	n/a	66	189	-	-	-	-	-	212	-	-	-	-	-
fft	0	22	1	2	**79**	79	92	98	0	21	2	0	**0.8%**	103	-	-	-	-	-
gaussian	0	56	0	2	**79**	79	94	232	-	-	-	-	-	675	-	-	-	-	-
sum	0	34	0	2	**351**	351	122	290	0	13	21	2	**2.5%**	379	-	-	-	-	-
cg	0	32	0	3	91	91	288	574	-	-	-	-	-	1027	0	25	7	2	**2.7%**
$E10^{-8}$	Initial				HiFPTuner									Precimonious					
						Configs/Level			Final Configuration						Final Configuration				
Program	L	D	F	C	Configs	L2	L1	L0	L	D	F	S	Speedup	Configs	L	D	F	S	Speedup
simpsons	9	0	0	2	**24**	n/a	24	58	0	8	1	1	**90.0%**	116	1	3	5	1	78.7%
arclength	8	0	0	3	**30**	n/a	30	39	0	7	1	1	4.9%	142	0	7	1	1	4.9%
piqpr	17	0	0	0	**52**	n/a	52	76	3	14	0	0	**41.2%**	164	3	13	1	0	0.3%
fft	0	22	1	2	**43**	43	52	100	-	-	-	-	-	297	0	21	2	0	0%
gaussian	0	56	0	2	**211**	211	132	74	0	10	46	2	0%	275	-	-	-	-	-
sum	0	34	0	2	533	261	64	208	0	10	24	2	**18.0%**	433	-	-	-	-	-
ep	0	13	0	4	**45**	45	38	76	-	-	-	-	-	77	-	-	-	-	-
cg	0	32	0	3	**497**	497	116	232	0	24	8	3	**3.3%**	735	-	-	-	-	-
$E10^{-6}$	Initial				HiFPTuner									Precimonious					
						Configs/Level			Final Configuration						Final Configuration				
Program	L	D	F	C	Configs	L2	L1	L0	L	D	F	S	Speedup	Configs	L	D	F	S	Speedup
simpsons	9	0	0	2	**22**	n/a	16	6	0	2	7	2	82.0%	24	0	2	7	2	82.0%
arclength	8	0	0	3	**16**	n/a	16	27	0	2	6	2	**44.7%**	94	-	-	-	-	-
piqpr	17	0	0	0	54	n/a	54	76	-	-	-	-	-	216	3	10	4	0	**0.7%**
gaussian	0	56	0	2	**69**	69	84	538	0	54	2	2	**0.4%**	361	-	-	-	-	-
sum	0	34	0	2	719	272	149	298	0	10	24	2	18.0%	**509**	0	10	24	2	18.0%
ep	0	13	0	4	**17**	17	4	92	0	10	3	4	**45.0%**	95	0	6	7	4	33.0%
cg	0	32	0	3	385	385	54	336	0	21	11	3	**5.6%**	361	0	25	7	3	0%
$E10^{-4}$	Initial				HiFPTuner									Precimonious					
						Configs/Level			Final Configuration						Final Configuration				
Program	L	D	F	C	Configs	L2	L1	L0	L	D	F	S	Speedup	Configs	L	D	F	S	Speedup
simpsons	9	0	0	2	**16**	n/a	16	6	0	5	4	2	82.0%	24	0	2	7	2	82.0%
arclength	8	0	0	3	**10**	n/a	10	15	0	6	2	3	**85.8%**	26	0	5	3	3	83.6%
piqpr	17	0	0	0	**22**	n/a	22	119	5	4	8	0	**0.7%**	134	-	-	-	-	-
gaussian	0	56	0	2	**147**	147	306	494	-	-	-	-	-	233	-	-	-	-	-
sum	0	34	0	2	195	195	156	578	-	-	-	-	-	**179**	-	-	-	-	-
ep	0	13	0	4	**17**	17	4	70	0	10	3	4	**45.0%**	113	0	5	8	4	43.0%
cg	0	32	0	3	197	157	20	20	0	3	29	3	**10.4%**	97	0	2	30	3	6.8%

Table 3: Benchmarks that can use uniform single precision.

		Initial			Final Configuration			
Program	Thresholds	D	F	C	D	F	S	Speedup
bessel	all	24	0	5	0	24	5	11.5%
fft	10^{-6}, 10^{-4}	22	1	2	0	23	2	43.4%

"Configs" in Table 2). In other words, HiFPTuner explores 59.6% fewer configurations on average for 22 out of the 29 experiments while finding equivalent or better configurations than Precimonious. Such improvement, which indicates a saving of more than half of the search time without affecting configuration quality, is significant, especially for long-running programs. This improvement is due to the reduction in the number of tunable items at the top level of the hierarchy compared to the number of variables and

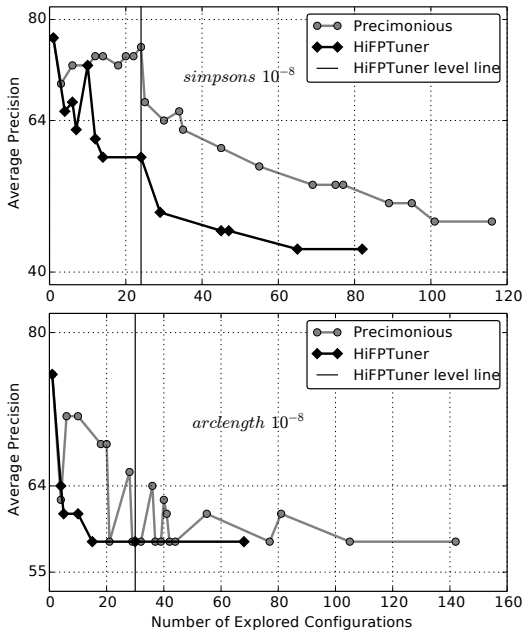


Figure 6: Search efficiency of HiFPTuner and Precimonious. The average precision of a configuration is calculated based on the number of bits used for floating-point variable allocations, e.g., the average precision is 32 if all allocations are in *float*. Each subfigure plots configurations whose output satisfy the specified error threshold 10^{-8}. HiFPTuner converges 1.4× and 2× faster than Precimonious for *simpsons*, and *arclength*, respectively.

function calls considered by Precimonious. For example, HiFPTuner only explores 6 communities at the top level for *simpsons* while Precimonious searches over 11 items (individual variables and function calls). The construction of the communities plays a critical role in improving search efficiency. Constructing these communities "blindly" can result in communities that cannot be assigned uniform precision to meet the accuracy and performance constraints, which eventually leads to worse configurations.

To further examine the improvement of HiFPTuner on search efficiency, we visualize the tuning processes of HiFPTuner and Precimonious in two experiments and observe their convergence rates. More specifically, we collect the configurations evaluated as *valid* subject to the accuracy constraint during the search (as a representative of all the explored configurations) to observe the decrease of the average precision of the explored configurations. Figure 6 plots the precision variation of the configurations with respect to the number of explored configurations for the programs *simpsons* 10^{-8} and *arclentgh* 10^{-8}. We observe that, overall, HiFPTuner converges 1.4× and 2× faster than Precimonious on *simpsons* and *arclength*, respectively. The final *valid* configurations for each tool are very close on the average precision. Additionally, "HiFPTuner level line" marks the point at which HiFPTuner moves from the top to the bottom level. As it can be observed in the plot of *simpsons*, the top-level final *valid* configuration is further improved on precision at the bottom level, however, it is found to be faster than all the configurations explored at the bottom level. For *arclength*, the top-level final configuration remains constant after the exploration descends to the bottom level. Both experiments show that it is unnecessary to explore the bottom level, without which, HiFPTuner converges 4.8× and 4.7× faster than Precimonious.

Table 4: Hierarchy construction results.

Program	Elements	h:mm:ss	Communities		
			Level 2	Level 1	Level 0
simpsons	11	0:02:53	n/a	6	11
arclength	11	0:00:56	n/a	7	11
piqpr	17	1:52:10	n/a	6	17
fft	25	0:14:12	11	14	25
gaussian	58	0:10:44	18	22	58
sum	36	0:05:48	23	24	36
bessel	29	0:04:44	11	14	29
ep	17	0:17:25	9	9	17
cg	35	0:11:39	21	24	35

4.2.2 Search Effectiveness. HiFPTuner proposes a configuration that improves performance for 20 out of 29 experiments. For the remaining 9 experiments, either a new configuration is proposed that does not lead to speedup, or the original configuration is returned (marked with '-'). Compared to Precimonious, HiFPTuner succeeds to tune 6 more programs (in other words, it finds a better configuration than the initial configuration) and reports a configuration that leads to larger speedups than Precimonious's for 15 programs in total.

To shed some light on the search effectiveness of HiFPTuner, we further examine the final configurations. As we can observe from Table 2, most of the configurations found by HiFPTuner are from the top level (25 out of 29), which indicates that variables within a given community are successfully assigned the same precision. In other words, the communities effectively guide the search (HiFPTuner) to a configuration that leads to larger speedups compared to a blind search (Precimonious) that explores too widely. This observation, i.e., reducing the large search space is useful to find better configurations, can also be learned from another point of view. Consider the program *sum* as an example, for error threshold 10^{-6}, both HiFPTuner and Precimonious find a configuration that leads to 18% speedup, however for the larger error threshold 10^{-4}, neither of them succeeds to find a configuration even though the configuration found for error threshold 10^{-6} surely satisfies error threshold 10^{-4}. In this case, a tighter error threshold successfully reduced the search space to a smaller space from which it was easier to find a profitable configuration.

4.3 Hierarchy Construction Results

We show the cost of hierarchy construction in Table 4. Hierarchy construction is relatively cheap for *simpsons* and *arclength*, taking 2 min 53 sec, and 56 sec, respectively. This is not the case for *piqpr*, for which hierarchy construction takes almost 2 hours. For the rest of the programs, hierarchy construction takes from 4 min to 17 min. Note that we only have to construct the hierarchy once per program despite the number of error thresholds to explore. Finally, we find that most of the time in constructing the hierarchy is spent in edge profiling. This task could be further optimized in the future.

Table 4 also lists the total number of elements (variables and function calls) for which community detection is performed, as

well as the number of communities detected at each hierarchy level. The long-double programs *simpsons*, *arclength*, and *piqpr* only have two hierarchy levels. Note that the resulting communities are further unfolded based on the initial precision of the variables within a community, therefore, a higher level can contain as many communities as its lower level, e.g., program *ep*.

5 LIMITATIONS AND FUTURE WORK

As with other dynamic tuners, HiFPTuner's configurations are dependent on the tuning inputs, and no accuracy guarantee is provided for untested inputs. We performed additional experiments that show that our configurations satisfy accuracy constraints for a large number of untested inputs, however it remains future work to use complementary input generation tools to ensure that the tuning input set is as representative as possible. On the other hand, HiFPTuner can be used to tune mid-size programs, and advances the state of the art in dynamic precision tuning by enabling more effective and efficient tuning. Given our promising results for the mid-size programs, we plan to expand our evaluation in the future to larger programs for which we expect to see even larger benefits.

Our dependence analysis focuses on floating-point variables. While there are library function calls such as `sin` and `cos` in the search space, we would like to include these in the dependence analysis. We would also like to investigate whether the configurations proposed by HiFPTuner enable additional floating-point optimizations, and/or vectorization. Another future direction is to determine the effectiveness of various community detection algorithms for precision tuning. Finally, we believe that the idea of leveraging program information to guide the search could be adopted in other settings that involve large search spaces, such as performance and parameter tuning, which are out of the scope of this paper.

6 RELATED WORK

Floating-Point Precision Tuning. Precimonious [31] is a dynamic analysis that finds a type configuration for floating-point variables that satisfies a given accuracy constraint, and improves performance. It uses the delta-debugging algorithm to search over the types of floating-point variables, in their order of declaration. As noted in this paper, the order of the variables has an impact on the effectiveness of the search, and the quality of the configurations. Instead, we propose a hierarchical search that exploits the community structure of variables to improve the effectiveness of the search. Blame analysis [30] performs shadow execution to identify variables that are numerically insensitive, which can be excluded from the search space before tuning. This technique is complementary to our work, and it can be used to preprocess the search space before applying HiFPTuner.

Chiang et al. [16] present a static approach to precision allocation based on formal analysis via Symbolic Taylor Expansions, and error analysis based on interval functions. Unlike dynamic tools, the precision allocation guarantees to meet the error target across all program inputs in an interval. However, this approach is limited to conditional-free expressions. Darulova et al. [19] propose a technique to rewrite programs by adjusting the evaluation order of arithmetic expressions prior to tuning. While sound, the technique is limited to relatively small programs that can be verified statically.

Lam et al. [24] build mixed-precision binaries via instrumentation. A brute-force algorithm is used to find double precision instructions that can be replaced with single precision, and later extend this work to visualize the program's precision sensitivity into histograms [23]. Schkufza et al. [33] develop a stochastic search method to optimize floating-point programs. The algorithm applies a variety of program transformations, trading bit-wise precision for performance to enhance compiler optimization on floating-point binaries. Instead, HiFPTuner assists programmers in finding a mixed-precision program at the source level whose performance is improved. *FloatWatch* [15] identifies instructions that can be run in a lower precision by computing the overall range of values for each instruction. All the above methods suffer from scalability limitations, and do not leverage community structure to guide the search. Finally, a number of tools (e.g., [8, 13, 28, 32]) have been proposed to find accuracy problems, and/or improve accuracy. On the other hand, HiFPTuner reduces precision to improve performance.

Floating-Point Code Testing. Testing tools for floating-point programs could provide support for precision tuners to evaluate configurations. So far, much effort has been put into the automatic generation of program inputs. Chiang et al. [18] guide a random search on the input domain to automatically locate inputs that trigger high numerical errors in the results. In [17], the authors discover inputs for divergence detection. Bagnara et al. [9] address constraint solving over floating-point numbers and generate path-oriented test inputs for floating-point programs. Fu et al. [21] applies unconstrained programming to identify inputs that can cover a new branch, and therefore achieve high coverage. The above tools could be combined with HiFPTuner to further strengthen its robustness.

7 CONCLUSION

We presented HiFPTuner, a semantic-based search algorithm to hierarchically guide the tuning of floating-point programs to improve performance with respect to numerical accuracy. Unlike other search-based tuners that treat the program as a black box, we analyze the code and its runtime behaviors to create a weighted dependence graph used to detect communities of variables and construct a hierarchy. We evaluated HiFPTuner on nine numerical programs, and compared its efficiency and effectiveness against the state-of-the-art tuner Precimonious. HiFPTuner exhibits higher search efficiency over Precimonious for 75.9% of the experiments that require fine tuning, taking 59.6% less search time on average. Moreover, HiFPTuner finds configurations with larger speedup than Precimonious's for 51.7% of the experiments. In one case, HiFPTuner was able to find a configuration that lead to a speedup as large as the global optimum configuration's found through exhaustive search (*simpsons* with error threshold 10^{-8}). Based on our observation, the communities constructed from the dependence graph successfully narrow down the search space and lead to a more scalable and effective search in precision tuning.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers, and our shepherd Sasa Misailovic, for their insightful comments and suggestions. This work was supported in part by NSF grant CCF-1750983.

REFERENCES

[1] Bbp code directory. http://www.experimentalmath.info/bbp-codes/.
[2] Community detection for networkx documentation. http://perso.crans.org/aynaud/communities/index.html.
[3] GSL - GNU scientific library - GNU project - free software foundation (FSF). http://www.gnu.org/software/gsl/.
[4] The llvm compiler infrastructure. http://llvm.org/.
[5] Networkx, high-productivity software for complex networks. https://networkx.github.io/.
[6] The Explosion of the Ariane 5. https://www.ima.umn.edu/~arnold/disasters/ariane.html.
[7] Toyota: Software to blame for Prius brake problems. http://www.cnn.com/2010/WORLD/asiapcf/02/04/japan.prius.complaints/index.html.
[8] D. An, R. Blue, M. Lam, S. Piper, and G. Stoker. Fpinst: Floating point error analysis using dyninst, 2008.
[9] R. Bagnara, M. Carlier, R. Gori, and A. Gotlieb. Symbolic path-oriented test data generation for floating-point programs. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation, ICST*, 2013.
[10] D. H. Bailey. A compendium of bbp-type formulas for mathematical constants. *Preprint, http://crd.lbl.gov/dhbailey/dhbpapers/index.html*, 2000.
[11] D. H. Bailey. Resolving numerical anomalies in scientific computation. *LBNL Report #: LBNL-548E*, 2008.
[12] D. H. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0, 1995.
[13] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2012.
[14] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008.
[15] A. W. Brown, P. H. J. Kelly, and W. Luk. Profiling floating point value ranges for reconfigurable implementation. In *Proceedings of the 1st HiPEAC Workshop on Reconfigurable Computing*, 2007.
[16] W.-F. Chiang, M. Baranowski, I. Briggs, A. Solovyev, G. Gopalakrishnan, and Z. Rakamarić. Rigorous floating-point mixed-precision tuning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL*, 2017.
[17] W.-F. Chiang, G. Gopalakrishnan, and Z. Rakamarić. Practical floating-point divergence detection. In *International Workshop on Languages and Compilers for Parallel Computing*, 2015.
[18] W. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev. Efficient search for inputs causing high floating-point errors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*, 2014.
[19] E. Darulova, E. Horn, and S. Sharma. Sound mixed-precision optimization with rewriting. *arXiv preprint arXiv:1707.02118*, 2017.
[20] A. D. Franco, H. Guo, and C. Rubio-González. A comprehensive study of real-world numerical bug characteristics. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE*, 2017.
[21] Z. Fu and Z. Su. Achieving high coverage for floating-point code via unconstrained programming. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2017.
[22] M. Girvan and M. E. Newman. Community structure in social and biological networks. *Proceedings of the national academy of sciences*, 2002.
[23] M. O. Lam and J. K. Hollingsworth. Fine-grained floating-point precision analysis. *The International Journal of High Performance Computing Applications*, 2016.
[24] M. O. Lam, J. K. Hollingsworth, B. R. de Supinski, and M. P. LeGendre. Automatically adapting programs for mixed-precision floating-point computation. In *Proceedings of the 27th international ACM conference on International conference on supercomputing, ICS*, 2013.
[25] W. M. McKeeman. Algorithm 145: Adaptive numerical integration by simpson's rule. *Commun. ACM*, 1962.
[26] M. E. Newman. Fast algorithm for detecting community structure in networks. *Physical review E*, 2004.
[27] M. E. Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 2006.
[28] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2015.
[29] F. Radicchi, C. Castellano, F. Cecconi, V. Loreto, and D. Parisi. Defining and identifying communities in networks. *Proceedings of the National Academy of Sciences of the United States of America*, 2004.
[30] C. Rubio-González, C. Nguyen, B. Mehne, K. Sen, J. Demmel, W. Kahan, C. Iancu, W. Lavrijsen, D. H. Bailey, and D. Hough. Floating-point precision tuning using blame analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE*, 2016.
[31] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, 2013.
[32] A. Sanchez-Stern, P. Panchekha, S. Lerner, and Z. Tatlock. Finding root causes of floating point error. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2018.
[33] E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*, 2014.
[34] R. Skeel. Roundoff error and the patriot missile. *SIAM News*, 1992.
[35] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE*, 1999.