Precimonious & HiFPTuner

Tuning Assistant for Floating-Point Precision

Ignacio Laguna, Harshitha Menon
Lawrence Livermore National Laboratory

Michael Bentley, Ian Briggs, Pavel Panchekha, Ganesh Gopalakrishnan
University of Utah

Hui Guo, Cindy Rubio González
University of California at Davis

Michael O. Lam
James Madison University

http://fpanalysis.tools.org/

This work was supported by through the X-Stack program funded by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under collaborative agreement SC0008699, NSF grant 1750983, and a gift from Oracle.
Floating-Point Precision Tuning

- Floating-point (FP) arithmetic used in variety of domains

- Reasoning about FP programs is difficult
  - Large variety of numerical problems
  - Most programmers are not experts in FP

- Common practice: use highest available precision
  - Disadvantage: more expensive!

- Goal: automated techniques to assist in tuning floating-point precision
Example: Arc Length

Consider the problem of finding the arc length of the function

\[ g(x) = x + \sum_{0 \leq k \leq 5} 2^{-k} \sin(2^k x) \]

Summing for \( x_k \in (0, \pi) \) into \( n \) subintervals

\[ \sum_{k=0}^{n-1} \sqrt{h^2 + (g(x_{k+1}) - g(x_k))^2} \]

with \( h = \pi / n \) and \( x_k = k h \)

<table>
<thead>
<tr>
<th>Precision</th>
<th>Slowdown</th>
<th>Result</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>double-double</td>
<td>20X</td>
<td>5.795776322412856</td>
<td>✔</td>
</tr>
<tr>
<td>double</td>
<td>1X</td>
<td>5.795776322413031</td>
<td>✗</td>
</tr>
<tr>
<td>mixed precision</td>
<td>&lt; 2X</td>
<td>5.795776322412856</td>
<td>✔</td>
</tr>
</tbody>
</table>
Example: Arc Length

```c
long double g(long double x) {
    int k, n = 5;
    long double t1 = x;
    long double d1 = 1.0L;
    for(k = 1; k <= n; k++) {
        ...
    }
    return t1;
}

int main() {
    int i, n = 1000000;
    long double h, t1, t2, dp; // dp = double precision
    long double s1;
    ...
    for(i = 1; i <= n; i++) {
        t2 = g(i * h);
        s1 = s1 + sqrt(h*h + (t2 - t1)*(t2 - t1));
        t1 = t2;
    }
    // final answer stored in variable s1
    return 0;
}
```

Mixed Precision Program
Precimonious

“Parsimonious or Frugal with Precision”

Dynamic Analysis for Floating-Point Precision Tuning

Annotated with error threshold

Less Precision

Modified program in executable format

Speedup
Challenges for Precision Tuning

● Searching efficiently over variable types and function implementations
  ○ Naïve approach -> exponential time
  ○ 19,683 configurations for arclength program ($3^9$)
  ○ 11 hours 5 minutes
  ○ Global minimum vs. Local minimum

● Evaluating type configurations
  ○ Less precision not necessarily faster
  ○ Based on runtime, energy consumption, etc.

● Determining accuracy constraints
  ○ How accurate must the final result be?
  ○ What error threshold to use?
Searching for Type Configuration

- **Double Precision**
  - 10 red circles
  - Checkmark

- **Single Precision**
  - 10 blue circles
  - Cross
Searching for Type Configuration

double precision

single precision
Searching for Type Configuration

double precision

single precision

http://fpanalysis-tools.org/
Searching for Type Configuration

double precision

single precision

http://fpanalysistools.org/
Searching for Type Configuration

double precision

single precision

http://fpanalysistools.org/
Searching for Type Configuration

double precision

single precision
Searching for Type Configuration

- **Double Precision**: All configurations marked as invalid.
- **Proposed Configuration**: Valid configuration.
- **Failed Configurations**: Multiple invalid configurations.
- **Single Precision**: All configurations marked as invalid.
Source code available:
https://github.com/plse/precimonious

Questions?
Directory Structure

/$HOME

|--/Module-Precimonious
|   |--/exercise
|   |--/exercise-2

|--/Module-HiFPTuner
|   |--/exercise
|   |--/exercise-2
Exercise

$ cd Module-Precimonious
Step 1: Build Precimonious

- Open setup.sh file
- Precimonious uses LLVM and is built using scons
- Execute:
  - $ ./setup.sh

Success building and running tests
Step 2: Annotate Program (already done)

- Execute:
  - $ cd exercise
  - $ ls

- Open `simpsons.c` file

The program we will tune:

```bash
hiusr@ip-172-31-8-101:~/Module-Precimionous/exercise$ ls
Makefile include.txt run-analysis.sh
exclude.txt include_global.txt run-config.sh
exclude_local.txt reference run-dependencies.sh

$ cd exercise
$ ls

FILE* file;
file = fopen("score.cov", "w");
fprintf(file, "%ld\n", diff);
fclose(file);

****** BEGIN PRECIMIONOUS ACCURACY CHECKING AND LOGGING ******/
threshold = pow(10, epsilon)*sl;

// cov_spec_log("spec.cov", threshold, 1, (long double)s1);
cov_log("result", "log.cov", 1, (long double) s1);
cov_check("log.cov", "spec.cov", 1);

****** END PRECIMIONOUS ACCURACY CHECKING AND LOGGING ******/
```

Accuracy logging & checking

Performance logging
Step 3: Compile Program with Clang

- **Execute:**
  - `$ make clean`
  - `$ make`

- Creates LLVM bitcode file and optimized executable for later use
Step 4: Run Analysis on Program

Sample output:

- Execute:
  - $ ./run-analysis.sh simpsons

Type changes are listed for each explored configuration

Suggested type configuration

Number of explored configurations

** Exploring configuration #109
** Changing precision of variables
  Variable x: x86_fp80 -> float
  Variable pi: x86_fp80 -> double
  Variable a: x86_fp80 -> float
  Variable b: x86_fp80 -> float
  Variable s1: x86_fp80 -> double
  Variable h: x86_fp80 -> float
  Variable fuzz: x86_fp80 -> float
  Variable x: x86_fp80 -> double

** Replacing function calls
  Function call: sin -> sinf

** Result is within error threshold

Check dd2_valid_simpsons_bc.json for the valid configuration file

Number of configurations explored by Precimonicous:

TOTAL: 110
--VALID 18
--INVALID 92
--FAILED 0
Step 4: Run Analysis – Configuration File

- Open `config_simpsons.json`
- Original type configuration

```json
"config": [
  {"localVar": {
    "function": "fun",
    "name": "x",
    "type": "longdouble"
  },
  {"localVar": {
    "function": "fun",
    "name": "pi",
    "type": "longdouble"
  },
  {"localVar": {
    "function": "fun",
    "name": "result",
    "type": "longdouble"
  },
  {"call": {
    "id": "4",
    "function": "fun",
    "name": "acos",
    "switch": "acos",
    "type": ["double","double"]
  }}
]"```
Step 4: Run Analysis – Search File

- Open `search_funarc.json`
- Search space file
- To exclude functions edit `exclude.txt`
- To exclude variables edit `exclude_local.txt`
- Or you can directly edit search file prior to analysis
Step 4: Run Analysis – Output Files

- Execute:
  - $ cd results
  - $ ls
Step 4: Run Analysis – Output Files

- Open dd2_valid_funarc.bc.json: suggested configuration file in JSON format
- Open dd2_diff_funarc.bc.json: summary of type changes

```plaintext
localVar: x at fun longdouble -> float
localVar: pi at fun longdouble -> double
call: sin at funsinf -> sinf
localVar: a at main longdouble -> float
localVar: b at main longdouble -> float
localVar: s1 at main longdouble -> double
localVar: h at main longdouble -> float
localVar: fuzz at main longdouble -> float
localVar: x at main longdouble -> double
```
Step 5: Apply Result Configuration & Compare Performance

- Execute:
  - $ cd ..
  - $ ./run-config.sh simpsons

- Execute:
  - $ time ./original_simpsons.out
  - $ time ./tuned_simpsons.out

```bash
hiusr@ip-172-31-8-101:~/Module-Precimonious/exercise$ ./run-config.sh simpsons
** Applying precimonious configuration
** Changing precision of variables
  Variable x: x86_fp80 -> float
  Variable pi: x86_fp80 -> double
  Variable a: x86_fp80 -> float
  Variable b: x86_fp80 -> float
  Variable s1: x86_fp80 -> double
  Variable h: x86_fp80 -> float
  Variable fuzz: x86_fp80 -> float
  Variable x: x86_fp80 -> double
** Replacing function calls
  Function call: sin -> sinf
** Result is within error threshold

Run the following to compare performance:
time ./original_simpsons.out
time ./tuned_simpsons.out
```
Exercise 2: Run Precimionous on funarc program

- Open exercise-2/funarc.c to see annotated program

- Execute:
  - cd ../exercise-2
  - make clean
  - make
  - ./run-analysis.sh funarc
  - ./run-config.sh funarc

- Open results/dd2_valid_funarc.bc.json to see configuration in JSON format
- Open results/dd2_diff_funarc.bc.json to see difference between original program and proposed configuration
Limitations of Precimonious

● Type configurations rely on inputs tested
  ○ No guarantees if worse conditioned input
  ○ Could be combined with input generation tools (e.g., S3FP)

● Getting trapped in local minimum

● Analysis scalability
  ○ Approach does not scale well for long-running applications
  ○ Need to reduce search space and reduce number of runs
  ○ Check out our follow up work on Blame Analysis (ICSE’16)

● Analysis effectiveness
  ○ Approach does not exploit relationship among variables
  ○ Check out our follow up work on HiFPTuner (ISSTA’18)
HiFPTuner: exploiting the community structure of the variables in precision tuning
Search Faster and Reach Better Configurations

Same type for variables in one community
- Decreased search space - only exploring the configurations which satisfy the community structure of the variables
- Better configurations for speed-up - dependent variables are assigned with the same type which avoids type casts

One type per variable
- Exponential number of type configurations with regard to the number of variables – large search space
- Trapped in local optimum introducing many type casts
HiFPTuner
Hierarchical Floating-Point Precision Tuning

1. Dependence analysis
2. Community detection
3. Hierarchical Search
   - can be combined with any base search algorithm such as binary search or delta-debugging algorithm

Source:
https://github.com/ucd-plse/HiFPTuner
Exercise

$ cd Module-HiFPTuner
Build HiFPTuner

$ source ./setup.sh

Check the environment variable

$ echo $LIBRARY_PATH

hiusr@ip-172-31-8-101:~/Module-HiFPTuner$ echo $LIBRARY_PATH
/home/hiusr/Module-HiFPTuner/HiFPTuner/precimoniouas/logging:
Step 1: Annotate Program and Compile it to *bitcode* File

Source: `simpsons.c` (annotated with accuracy logging/checking functions and timing code shown before)

Compile `simpsons.c` to LLVM *bitcode* file

It generates `simpson.bc` and the executable `original_simpsons.out`

Note: `original_simpsons.out` will be used later for performance comparison
Step 2: Run HiFPTuner

Run HiFPTuner on simpsons.bc

$ ./run-hifptuner.sh simpsons

Output files:

./results-hifptuner
- result file
- dd2_valid_simpons.bc.json: the precision configuration file

log files
- log.txt, log.dd: search log of HiFPTuner
- sorted_partition.json: the community structure of floating-point variables
- auto-tuning_analyze_time.txt: dependence analysis time
- auto-tuning_config_time.txt: community detection time
**Step 2: Run HiFPTuner – community detection**

**Input**: `varDepPairs_pro.json, edgeProfilingOut.json`

**Output**: `sorted_partition.json`

```
[{
  "fun.x": 0,
  "main.x": 1,
  "fun.pi": 0,
  "main.threshold": 2,
  "fun.result": 0,
  "main.s1": 2,
  "main.h": 1,
  "main.b": 1,
  "main.a": 1,
  "main.epsilon": 2
}]
```

Hierarchy height : 1
Step 2: Run HiFPTuner – community detection

Input : varDepPairs_pro.json, edgeProfilingOut.json
Output : sorted_partition.json

Hierarchy height : 1
Community number
(sorted in the topological order of dependence)
Step 2: Run HiFPTuner – hierarchical search

Input : simpsons.bc,
       search_simpsons.json, config_simpsons.json,
       sorted_partition.json
Output : dd2_valid_simpsons.bc.json
Step 3: Tuned Program VS Original Program

Generate tuned executable `hifptuner_tuned_simpsons.out`

```
$ cd ..
$ ./run-config.sh simpsons
```

Time the execution of the original program and the tuned program and compare the execution time

```
$ time ./original_simpsons.out
$ time ./hifptuner_tuned_simpsons.out
```

0m1.710s VS 0m0.951s
Step 4: HiFPTuner VS Precimonious

- Tuned program: which is faster?
- Search time: which explored less configurations?
Step 4: HiFPTuner VS Precimonious: tuned program

Time the execution of the tuned programs of Precimonious and HiFPTuner, and compare the execution time

```
$ time ../../Module-Precimonious/exercise/tuned_simpsons.out
$ time ./hifptuner_tuned_simpsons.out
```

0m1.191s VS 0m0.951s: HiFPTuner found better configuration.
Step 4: HiFPTuner VS Precimonious: search time

Compare the search effort of Precimonious and HiFPTuner:

$ cat ../../Module-Precimonious/exercise/results/log.txt

$ cat results-hifptuner/log.txt

<table>
<thead>
<tr>
<th>Number of configurations explored by Precimonious:</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL: 110</td>
</tr>
<tr>
<td>--VALID 18</td>
</tr>
<tr>
<td>--INVALID 92</td>
</tr>
<tr>
<td>--FAILED 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of configurations explored by HiFPTuner:</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL: 20</td>
</tr>
<tr>
<td>--VALID 6</td>
</tr>
<tr>
<td>--INVALID 14</td>
</tr>
<tr>
<td>--FAILED 0</td>
</tr>
</tbody>
</table>

HiFPTuner is more efficient.

VALID configuration: accuracy check ✔
INVALID configuration: accuracy check X
FAILED configuration: it crashes
Exercise 2: Run HiFPTuner on funarc program

- Open exercise-2/funarc.c to see annotated program

- Execute:
  - cd ../exercise-2
  - make clean
  - make
  - ./run-hifptuner.sh funarc
  - ./run-config.sh funarc

- Open results-hifptuner/dd2_valid_funarc.bc.json to see configuration in JSON format

- Open results-hifptuner/dd2_diff_funarc.bc.json to see difference between original program and proposed configuration
Run Precimonious or HiFPTuner on Your Program

- Annotate your program with our utility functions
  - **Accuracy log and check**
    - `cov_spec_log`: log the accurate result yielded by original precision to file “spec.cov”
    - `cov_log`: log the result of the program in each execution to file “log.cov”
    - `cov_check`: check whether the result in current execution satisfies the accuracy criterion
  - **Performance log**
    - log the execution time of the code of interest to the file: “score.cov”

- Compile your program to LLVM *bitcode*
  - **WLLVM**, [https://github.com/travitch/whole-program-llvm](https://github.com/travitch/whole-program-llvm) - for large projects

- Download the precision tuning docker image
Collaborators

University of California, Berkeley

Cuong Nguyen
Diep Nguyen
Ben Mehne
James Demmel
William Kahan
Koushik Sen

Lawrence Berkeley National Lab

Costin Iancu
David Bailey
Wim Lavrijsen

Oracle

David Hough

http://fpanalysistools.org/
Precimonious:  
https://github.com/ucd-plse/precimonious

HiFPTuner:  
https://github.com/ucd-plse/HiFPTuner

Hui Guo <hguo@ucdavis.edu>
Cindy Rubio-Gonzalez <crubio@ucdavis.edu>

Questions?