The Road Not Taken: Exploring Alias Analysis Based Optimizations Missed by the Compiler (Artifact)

1 Getting Started

SCOUT was evaluated on a 3.60GHz Intel(R) Core(TM) i9-9900K CPU 8 core machine with an x86_64 architecture and 32 GB primary memory, with a 64-bit Ubuntu 20.04.2 LTS operating system. We implemented SCOUT as a part of the LLVM infrastructure version 10.0.0. We extended the widely used production allocator JEMALLOC-5.2.1 to implement our custom allocator. We have used the Polybench (version 4.2.1) and CPU SPEC 2017 benchmarks to evaluate SCOUT.

1.1 Setting up the virtual machine

1. Install Virtual Box on Ubuntu

   ```
   sudo apt-get update
   sudo apt-get install virtualbox
   sudo apt-get install virtualbox-ext-pack
   ```

2. We have successfully installed and compiled all the required libraries, SCOUT, native and modified JEMALLOC allocator, LLVM version 3.6.0 and the open source Polybench benchmark on the virtual machine. As CPU SPEC 2017 is a licensed benchmark suite, we have included the config files and scripts to build and execute the CPU SPEC 2017 benchmarks. We discuss the steps to set up CPU SPEC 2017 in Section 1.2 using the available ISO image. The virtual machine image (.ova) can be downloaded using the following link: `https://doi.org/10.5281/zenodo.6817675`
3. Importing .ova file in Virtual Box

   ```
   #Open virtual Box.
   #Select File > Import Appliance...
   #Browse through the files and select the downloaded
   #.ova file.
   #Click import at the bottom.
   #It may take a while to import the .ova file.
   ```

4. Now, start the imported virtual machine. If you see the Ubuntu start screen, the virtual machine has booted successfully. `Password: scout`

1.2 Setting up the CPU SPEC 2017 benchmarks

As the CPU SPEC 2017 benchmark suite is licensed, it cannot be distributed publicly. Therefore, in this section, we list the steps to set up the CPU SPEC 2017 benchmarks, assuming availability of the ISO image.

1. Create CPU SPEC 2017 installation directory.

   ```
   cd /home/oopsla22-scout/Scout/
   mkdir SPEC
   ```

2. Mount the ISO image by right-clicking on the .iso file and selecting "Open with Disk Image Mounter".
3. Open a terminal in the mounted file system and type,

   ```
   ./install.sh -d /home/oopsla22-scout/Scout/SPEC/
   ```

4. Confirm the installation directory.

   ```
   #It may take a while to install the benchmarks.
   ```

5. Open a terminal and type,

   ```
   cd /home/oopsla22-scout/Scout/SPEC/
   source shrc
   #Alternatively, run source cshrc
   ```

6. To test the installation, check the version,

   ```
   runcpu --version
   #This will display the summary of the versions of
   #the utilities.
   ```

7. Copy the config files and the scripts to run CPU SPEC 2017 benchmarks with Scout,

   ```
   cp /home/oopsla22-scout/Scout/CPUSPEC2017/config/* \
   /home/oopsla22-scout/Scout/SPEC/config/
   cp /home/oopsla22-scout/Scout/CPUSPEC2017/scripts/* \
   /home/oopsla22-scout/Scout/SPEC/
   ```

NOTE: It is possible that some of the installed CPU SPEC 2017 benchmarks are not updated. In that case, the build phase for some benchmarks (such as `510.parest_r`) might fail. To update the CPU SPEC 2017 benchmarks, run the following commands,

```
cd /home/oopsla22-scout/Scout/SPEC/
source shrc
runcpu --update
#This may take a while.
#Comment the Wl-stack flag in clang.xml file.
```

1.3 Building from the scratch (Optional)

This section describes the steps to manually compile SCOUT, allocator and LLVM-3.6.0 from the source code, if the user wants to build it from scratch. Otherwise, we have already compiled SCOUT, native and custom allocator, and LLVM-3.6.0 in the virtual machine. We have also installed all the required libraries in the virtual machine.

1. CMake
   ```
   sudo apt-get install cmake
   ```
2. Autoconf
   ```
   sudo apt-get install autoconf
   ```
3. Libffi-dev
   ```
   sudo apt-get install -y libffi-dev
   ```
4. Ninja
   ```
   sudo apt install ninja-build
   ```
5. Compiling SCOUT
   ```
   cd /home/oopsla22-scout/Scout/build/
   ./build.sh
   ninja
   ```
6. Compiling LLVM-3.6.0
   ```
   cd /home/oopsla22-scout/Scout/build_3.6.0/
   ./build.sh
   ninja
   ```
7. Compiling native JEMALLOC allocator
   ```
   cd /home/oopsla22-scout/Scout/jemallocn/
   ./autogen.sh
   make
   ```
8. Compiling custom JEMALLOC allocator
   ```
   cd /home/oopsla22-scout/Scout/jemalloc2k/
   ./autogen.sh
   make
   ```

2 Step by Step Instructions

This section provides step by step instructions to regenerate the results presented in the Evaluation section (Section 5) of the paper. We have provided the scripts to get the results presented in Tables 1, 2, 3, 4 and 5 of the Evaluation section. This section is organized as follows:

1. The first section lists the steps to regenerate the results for the Polybench benchmarks (Table 2 and 3).
2. The second section lists the steps to regenerate the results for the CPU SPEC 2017 benchmarks (Table 1, 4 and 5).

2.1 Generating Polybench results

Tables 2 and 3 in the Evaluation section present the results for the Polybench benchmarks. These benchmarks are present in the folder `/home/oopsla22-scout/Scout/polybench-c-4.2.1-beta/` of the virtual machine. The build scripts are present in the folder `/home/oopsla22-scout/Scout/polybench-c-4.2.1-beta/scripts/`. The executable for the benchmarks are by default created in the folder `/home/oopsla22-scout/Scout/polybench-c-4.2.1-beta/exe/`.

We have evaluated SCOUT on the default input set (`LARGE`) of Polybench. The default number of iterations is set to 5. However, this value can be manually set by updating the variable `iter` in the build scripts. The variable `BENCHMARK` can also be modified to manually specify the benchmark names.

2.1.1 Generating the results presented in Table 2

This section describes how results shown in Table 2 of the paper can be reproduced. Launch the script table2.sh.

```
cd /home/oopsla22-scout/Scout/polybench-c-4.2.1-beta/ \
scripts/
./table2.sh
#This should not take more than 4-5 minutes to execute.
```

The output of this script is divided into three parts. The first part presents the total number of loops versioned by SCOUT for the six benchmarks, $\#\boldsymbol{L}_a$. The second part presents the performance benefits when the benchmarks are compiled using SCOUT, $\boldsymbol{P}_a$. The third part presents the performance benefits when the benchmarks are compiled with the `restrict` keyword, $\boldsymbol{P}_r$. The performance benefits can possibly vary for different hardware configurations. However, the other statistics will remain the same for the same version of benchmarks, as this number is obtained during the compilation step.

2.1.2 Generating the results presented in Table 3

This section describes how the results shown in Table 3 of the paper can be reproduced. Run the script table3.sh.

```
cd /home/oopsla22-scout/Scout/polybench-c-4.2.1-beta/ \
scripts/
./table3.sh
#This should not take more than 5-6 minutes to execute.
```

The output of this script is divided into three parts. The first part presents the speedups when the benchmarks are compiled with the `restrict` keyword using LLVM-3.6.0, $\boldsymbol{S}_{ra}$. The second part presents the speedups when the benchmarks are compiled using SCOUT, $\boldsymbol{S}_s$. The third part presents the speedups when the benchmarks are compiled with the `restrict` keyword using LLVM-10, $\boldsymbol{S}_{rb}$. The speedups can possibly vary for different hardware configurations. The statisticts provided in the first column of Table 3, $\boldsymbol{S}_h$ are taken from Figure 17 [1].

2.2 Generating CPU SPEC 2017 results

Tables 1, 4 and 5 in the Evaluation Section (Section 5) of the paper present the CPU and memory overhead along with the performance improvement obtained with and without the profiler for the CPU SPEC 2017 benchmarks. In this section, we describe how to reproduce the presented results.

We have evaluated SCOUT on the `reference` input set of the CPU SPEC 2017 benchmarks. The execution times of the benchmarks lie in the range of 130-340 seconds on the system that was used for evaluation. A smaller input set called `test`, for which the execution times of the benchmarks lie in the range of 0.01-22 seconds, is also available. The build scripts that we have provided execute the `reference` input set by default. However, if you want to use the `test` input set for evaluation, you can do so by updating the variable `SIZE` in the scripts. The performance benefits might vary with the input set.

The default number of iterations is set to 5. However, this value can be modified by updating the variable `ITER` in the build scripts. The variable `BENCHMARK` can also be modified to manually specify the benchmark names.

2.2.1 Generating the results presented in Table 1

The results presented in Table 1 can be reproduced by executing the table1.sh script.

```
cd /home/oopsla22-scout/Scout/SPEC/
./table1.sh
```

The first part of the output represents the memory overhead of the custom allocator **MO** for each benchmark. The second part represents the CPU overhead of the allocator **CO** for each benchmark. The overheads can possibly vary for different hardware configurations.

2.2.2 Generating the results presented in Table 4

The results presented in Table 4 can be obtained by executing the table4.sh script.

```
cd /home/oopsla22-scout/Scout/SPEC/
./table4.sh
```

The first part of the output represents the number of versioned loops executed $\#\boldsymbol{L}_e$. The second part of the output represents the number of loops versioned by SCOUT $\#\boldsymbol{L}_a$ and the number of versioned loops that are vectorizable $\#\boldsymbol{L}_b$. The third part represents the performance benefits obtained when the benchmarks are compiled with SCOUT $\boldsymbol{P}_a$. The performance benefits can possibly vary for different hardware configurations. However, the other statistics will remain the same for the specified versions of the CPU SPEC 2017 benchmarks.

2.2.3 Generating the results presented in Table 5

The results presented in Table 5 can be obtained by launching the table5.sh script.

```
cd /home/oopsla22-scout/Scout/SPEC/
./table5.sh
```

This script executes the profiler on the benchmarks to collect the time statistics, as mentioned in Section 3.5 of the paper. This script further compiles the benchmarks using the profiler's output and SCOUT. The output of this script represents the number of versioned loops based on the profiler's feedback $\#\boldsymbol{L}_p$, number of versioned loops that are vectorizable $\#\boldsymbol{L}_b$ and the performance benefits $\boldsymbol{P}_a$ for different thresholds (of the loop's execution time improvement) when the benchmarks are compiled using SCOUT and the profiler's feedback. Currently, this script reports the results for threshold 0, 10, 20 and 30, as discussed in the paper. However, the user can customize the thresholds by updating the variable `VAL` in the build scripts.

References

1. Alves, P., Gruber, F., Doerfert, J., Lamprineas, A., Grosser, T., Rastello, F., Pereira, F.M.Q.a.: Runtime pointer disambiguation. In: Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. p. 589–606. OOPSLA 2015, Association for Computing Machinery, New York, NY, USA (2015). https://doi.org/10.1145/2814270.2814285