Rapid: Region-based Pointer Disambiguation (Artifact)

1 Introduction

RAPID was evaluated on an Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz 12-core machine with x86_64 architecture, 32 GB of primary memory, and the Ubuntu 20.04.1 LTS operating system. We implemented RAPID as part of the LLVM infrastructure (v10.0.0) and modified Mimalloc (v2.0.6) to implement our region-based memory allocator. Hyper-threading was disabled during the evaluation. We evaluated RAPID using Polybench (version 4.2.1) and CPU SPEC 2017 benchmarks.

2 Directory structure

The artifact for RAPID is divided into four parts:

1. RAPID: The source code of RAPID belongs to the folders `llvm_rapid`, `clang_rapid`, `mimalloc_prof` and `mimalloc_region`. Folders `llvm_rapid` and `clang_rapid` comprise the LLVM/clang implementation of RAPID. Folders `mimalloc_prof` and `mimalloc_region` constitute the implementation of the custom allocator. Section 3 describes the manual steps to build RAPID. Folder $HOME/stats/ stores the log files.
2. Scout: The source code of Scout belongs to the folders `llvm_scout`, `clang_scout`, `jemallocn` and `jemalloc2k`. Folders `llvm_scout`, `clang_scout` consist of the LLVM/clang implementation of Scout. Folders `jemallocn` and `jemalloc2k` constitute the implementation of the native and the custom allocator. Section 3 describes the manual steps to build Scout.
3. LLVM-3.6.0: The source code of LLVM-3.6.0 belongs to the folders `llvm-3.6.0`, and `clang_3.6.0`. Section 3 describes the steps to build LLVM-3.6.0 manually.
4. Testcases and the benchmarks: Folder `Testcase` consists of testcases to test the setup of RAPID and run it. This folder also contains a script to compile and run the testcases with RAPID. Section 3 describes the steps to run it. Folders `polybench-c-4.2.1-beta` and `polybench-c-4.2.1-beta_1` contain the source code of Polybench benchmarks. Section 4 describes the steps to run the benchmarks and regenerate the results reported in the paper. Folder `spec` contains the config files and the scripts to run CPU SPEC 2017 benchmarks. Sections 3 and 4 describe the steps for setting up the CPU SPEC 2017 benchmarks and regenerating the results reported in the paper.

NOTE: All the folders contain the required build files and the scripts.

3 Getting Started

3.1 Setting up Rapid

This subsection describes the steps to compile and set up all the artifacts mentioned in Section 1 to reproduce the results reported in the paper. It also provides instructions on running a few test cases with Rapid to verify the setup. Additionally, it describes the steps followed by the setup script.

1. Download all the required artifacts for evaluating Rapid using the following command:

   ```
   cd $HOME/

   #Using git.
   #Install git if not already present.
   sudo apt-get install git

   git clone \
   https://github.com/khushboochitre/artifact_rapid.git

   #Using zenodo.
   #Download the artifact using the following command:
   wget https://doi.org/10.5281/zenodo.8141381

   tar -xvf 475_oopsla23.tar.xz
   ```

2. Compile and set up all the artifacts:

   ```
   cd $HOME/artifact_rapid/

   ./setup.sh
   ```

   The setup process will take approximately 30-45 minutes. The setup script installs all the necessary dependencies and compiles the source code for Rapid, region-based allocator, Scout, segment-based allocator, and LLVM-3.6.0. This completes the setup of Rapid. Now, we need to test whether the installation was successful.

3. Test the setup of Rapid:

   ```
   cd Testcase/

   ./run.sh
   ```

   The `Testcase` folder contains three test cases:

   (a) `test1.c` depicts memory allocations using standard APIs to demonstrate our default approach.

   (b) `test2.c` depicts memory allocations using wrapper functions to demonstrate the handling of wrappers in our approach (Section 4.0.5 in the paper).

(c) `test3.c` depicts memory allocations inside a loop to demonstrate the handling of conflicting allocation sites in our approach (Section 3.3.2 in the paper).

The script generates log files for all three test cases in the `$HOME/stats` folder. It then uses these log files for final code generation for the test cases. The script outputs the execution times when compiled with native configurations and Rapid. The performance benefits can be computed by calculating the percentage decrease in the execution time compared to native execution.

3.2 Setting up CPU SPEC 2017

The CPU SPEC 2017 benchmark suite is licensed, so it cannot be distributed publicly. The cpu2017-1_0_5.iso needs to be purchased. This section describes the steps to set up CPU SPEC 2017 benchmarks, assuming the availability of cpu2017-1_0_5.iso.

1. Create an installation folder:
   ```
   cd $HOME/artifact_rapid

   mkdir cpuspec
   ```
2. Mount the ISO image by right-clicking on the .iso file and choosing "Open with Disk Image Mounter".
3. Open a terminal in the mounted file system:
   ```
   ./install.sh -d $HOME/artifact_rapid/cpuspec
   ```
4. Confirm the installation directory. It may take 2-3 minutes to install the benchmarks.
5. Open a terminal:
   ```
   cd $HOME/artifact_rapid/cpuspec

   source shrc
   ```
6. To verify the installation:
   ```
   runcpu --version
   ```
7. Copy the scripts and config files to the installation folder:
   ```
   cp $HOME/artifact_rapid/spec/config/* \
   $HOME/artifact_rapid/cpuspec/config/

   cp $HOME/artifact_rapid/spec/scripts/* \
   $HOME/artifact_rapid/cpuspec/
   ```
8. Update the 510.parest_r benchmark:
   ```
   runcpu --update

   #Comment the Wl-stack flag in clang.xml file.
   ```

The setup and basic testing are now complete. You can proceed to the next section (Section 4) to reproduce the results reported in the paper.

3.3 Steps followed by the setup.sh script

1. Installing dependencies:

```
sudo apt-get install build-essential

sudo apt-get install cmake

sudo apt-get install autoconf

sudo apt-get install -y libffi-dev

sudo apt install ninja-build
```

2. Moving the clang folders to the llvm directories:

```
cd artifact_rapid

cp -r $HOME/artifact_rapid/clang_rapid \
$HOME/artifact_rapid/llvm_rapid/tools/clang

cp -r $HOME/artifact_rapid/clang_scout \
$HOME/artifact_rapid/llvm_scout/tools/clang

cp -r $HOME/artifact_rapid/clang_3.6.0 \
$HOME/artifact_rapid/llvm-3.6.0/tools/clang
```

3. Setting up Rapid:

```
cd $HOME/artifact_rapid/llvm_rapid/build/

./build.sh

ninja
```

4. Setting up the region-based allocator:

```
cd $HOME/artifact_rapid/mimalloc_prof/

cmake .

make

cd $HOME/artifact_rapid/mimalloc_region/

cmake .

make
```

5. Setting up Scout:

```
cd $HOME/artifact_rapid/llvm_scout/build/

./build.sh

ninja
```

6. Setting up the segment-based allocator:

```
cd $HOME/artifact_rapid/jemallocn/

./autogen.sh

make -j

cd $HOME/artifact_rapid/jemalloc2k/

./autogen.sh

make -j
```

7. Setting up LLVM-3.6.0:

```
cd $HOME/artifact_rapid/llvm-3.6.0/build/

./build.sh

ninja
```

3.4 Flags description

This section specifies the flags required to compile the applications with RAPID for different configurations. The user can use these flags to compile RAPID manually.

1. To generate the profile files (Section 5.2.4 of the paper):
   "`-O3 -g -mllvm -enable-wrapper-annotation -mllvm -instrument-dynamic-checks -mllvm <log_file_name> -L$HOME/artifact_rapid/mimalloc_prof -lmimalloc1 -lm`"
2. To get the wrapper functions (Table 1 of the paper):
   "`-identify-wrappers -disable-additional-vectorize`"
   NOTE: This pass works with the OPT tool.
3. To execute the native version of the application (Figure 7, Figure 9, Table 4, and Table 6):
   "`-O3 -mllvm -enable-native -mllvm -region-wrapper-annotation -mllvm -disable-additional-vectorize -g -L $HOME/artifact_rapid/mimalloc_region -mllvm -use-check-info -mllvm <log_file_name> -lmimalloc -lm`"

4. To execute with the custom allocator without dynamic checks and loop-versioning (Figure 6, Figure 7, Figure 9, and Table 2):
   ”`-O3 -mllvm -region-wrapper-annotation -mllvm -region-disable-checks -mllvm -disable-additional-vectorize -g -L $HOME/artifact_rapid/mimalloc_region -mllvm -use-check-info -mllvm <log_file_name> -lmimalloc -lm`”

5. To execute with Rapid (Figure 6, Figure 9, and Table 2):
   ”`-O3 -mllvm -stats -mllvm -region-wrapper-annotation -mllvm -disable-additional-vectorize -g -L $HOME/artifact_rapid/mimalloc_region -mllvm -use-check-info -mllvm <log_file_name> -lmimalloc -lm`”
   NOTE: ”`-mllvm -stats`” flag reports the number of loops versioned and the number of required regions.

6. To execute with Rapid when the dynamic checks always fail (Figure 6 and Figure 9):
   ”`-O3 -mllvm -stats -mllvm -region-check-overhead -mllvm -region-wrapper-annotation -mllvm -disable-additional-vectorize -g -L $HOME/artifact_rapid/mimalloc_region -mllvm -use-check-info -mllvm <log_file_name> -lmimalloc -lm`”

7. To execute with Rapid handling allocations inside the loops (Table 2 and Table 9):
   ”`-O3 -mllvm -region-threshold -mllvm <int_threshold> -mllvm -stats -mllvm -region-wrapper-annotation -mllvm -disable-additional-vectorize -g -L $HOME/artifact_rapid/mimalloc_region -mllvm -use-check-info -mllvm <log_file_name> -lmimalloc -lm`”
   NOTE: ”`-mllvm -region-threshold -mllvm <int_threshold>`” flag is used to specify the maximum number of regions available.

8. To execute with Rapid handling allocations inside the loops and loop filtering mechanism (Table 6):
   ”`-O3 -mavx -mllvm -region-loop-iter-threshold -mllvm <int_loopiter_threshold> -mllvm -region-threshold -mllvm <int_threshold> -mllvm -stats -mllvm -region-wrapper-annotation -mllvm -disable-additional-vectorize -g -L $HOME/artifact_rapid/mimalloc_region -mllvm -use-check-info -mllvm <log_file_name> -lmimalloc -lm`”
   NOTE: ”`-mllvm -region-loop-iter-threshold -mllvm <int_loopiter_threshold>`” flag is used to specify the number of iterations for the loop filtering mechanism.

4 Step-by-Step Instructions

This section outlines the steps to reproduce the results reported in the Evaluation section (Section 5) of the paper. The evaluation was conducted on Polybench and CPU SPEC 2017 benchmarks using the `large` and `reference` input sets, respectively. The reported execution times are the arithmetic mean of five runs.

For the `large` input sets, a single iteration of Polybench benchmarks takes approximately three minutes. The variables `iter` and `BENCHMARK` can be modified in the provided scripts to update the number of iterations and benchmarks to execute. The default value of `iter` is set to five.

For the `reference` input set, a single iteration of CPU SPEC 2017 benchmarks takes approximately two hours. The smaller input set, `test`, can be selected by updating the variable `SIZE` in the scripts. The number of iterations can be modified by updating the variable `ITER` in the scripts. The maximum number of regions for handling allocations inside the loops can be adjusted using the variable `THRESH` in the scripts, and the loop filtering mechanism's threshold can be modified using the variable `ITERVAL`.

The following subsections describe the steps to reproduce the tables presented in Section 5 of the paper.

4.0.1 Generating the Profile Files

To obtain the log files, run the following commands:

```
cd $HOME/artifact_rapid/polybench-c-4.2.1-beta

./polybench_rapid_profiler.sh

#This process takes approximately 3 minutes to execute.

cd $HOME/artifact_rapid/cpuspec

./profile.sh

#This process takes approximately 5 hours to execute.
```

These commands generate log files containing the required dynamic checks for the benchmarks. The log files will be stored in the `$HOME/stats` folder.

4.0.2 Generating the Results presented in Figure 6

To obtain the results presented in Figure 6, run the following command:

```
cd $HOME/artifact_rapid/polybench-c-4.2.1-beta

./figure6.sh

#This process takes approximately 3-4 minutes to execute.
```

This script outputs the performance benefits achieved using the `restrict` keyword, Scout, and RAPID. It also provides the number of regions required by RAPID and the number of loops versioned by Scout and RAPID. Additionally, it reports the worst-case performance for RAPID. The statistics for Scout are obtained from the paper [2].

NOTE: For `jacobi-1d`, the percentage change in the execution time might vary, as the benchmark's execution time is quite small. To mitigate this, it is recommended to run the experiment multiple times for this benchmark.

4.0.3 Generating the Results presented in Table 2

To obtain the results presented in Table 2, run the following command:

```
#Update GCC versions by updating the
#variables $GCC_PATH and
#$COMPILER_PATH in the script
#polybench_restrict_speedup_3.sh.

cd $HOME/artifact_rapid/polybench-c-4.2.1-beta

./table2.sh

#This process takes approximately 4-5 minutes to execute.
```

This script outputs the speedups achieved using LLVM-3.6.0, LLVM-10, the hybrid approach [1], Scout, and RAPID. The statistics for Scout are obtained from the paper [2].

4.0.4 Generating the Results presented in Figure 7

To obtain the results presented in Figure 7, run the following command:

```
cd $HOME/artifact_rapid/cpuspec

./figure7.sh

#This process takes approximately 22 hours to execute
the five iterations for each setting.
```

This script outputs the CPU overhead of the custom allocator used by RAPID and Scout. The statistics for Scout are obtained from the paper [2].

4.0.5 Generating the Results presented in Figure 8

To obtain the results presented in Figure 8, run the following command:

```
cd $HOME/artifact_rapid/cpuspec

./figure8.sh
```

```
#This process takes approximately 4 hours to execute
one iteration.
```

This script outputs the memory overhead of the custom allocator used by Rapid and Scout. The statistics for Scout are obtained from the paper [2].

4.0.6 Generating the Results presented in Figure 9

To obtain the results presented in Figure 9, run the following command:

```
cd $HOME/artifact_rapid/cpuspec

./figure9.sh

#This process takes approximately 20 hours to execute
the five iterations for each setting.
```

This script outputs the performance benefits achieved using Scout and Rapid, with and without the allocator's overhead. It also provides the number of regions required by Rapid and the number of loops versioned by Scout and Rapid. Additionally, it reports the worst-case performance for Rapid and Scout. The worst-case performance column for Scout represents the performance of Scout without the profiler. The statistics for Scout are obtained from the paper [2].

4.0.7 Generating the Results presented in Table 4

To obtain the results presented in Table 4, run the following command:

```
cd $HOME/artifact_rapid/cpuspec

./table4.sh

#This process takes approximately 10 hours to execute
the five iterations for each setting.
```

This script outputs the CPU overhead, and memory overhead for the three benchmarks with memory allocations inside the loops. The experiments used five thresholds (8, 16, 32, 128 and 256) for the maximum available regions.

4.0.8 Generating the Results presented in Table 5

To obtain the results presented in Table 5, run the following command:

```
cd $HOME/artifact_rapid/cpuspec

./table5.sh

#This process takes approximately 20 hours to execute
the five iterations for each setting.
```

This script outputs the performance benefits for the three benchmarks with memory allocations inside the loops. The experiments used five thresholds (8, 16, 32, 128 and 256) for the maximum available regions. The script also reports the worst-case performance for these benchmarks across all thresholds.

4.0.9 Generating the Results presented in Table 6

To obtain the results presented in Table 6, run the following command:

```
cd $HOME/artifact_rapid/cpuspec

./table6.sh

#This process takes approximately 20 hours to execute
the five iterations for each setting.
```

This script outputs the performance benefits achieved using the loop filtering mechanism. A threshold of 64 is considered to filter out the loops. The output includes the performance benefits with and without the allocator's overhead for six thresholds five thresholds (0, 8, 16, 32, 128 and 256) for the maximum available regions.

4.0.10 Generating the Results presented in Figures 10-13

To obtain the results presented in Figures 10-13, run the following command:

```
cd $HOME/artifact_rapid/cpuspec

./perf_stats.sh

#This process takes approximately 6 hours to execute
one iteration for each setting.
```

This script outputs L1-dcache-loads (L1-dcache-loads), L1-dcache-load-misses (L1-dcache-load-misses), L1-icache-loads (icache_64b.iftag_hit + icache_64b.iftag_miss) and L1-icache-load-misses (percentage of icache_64b.iftag_miss w.r.t L1-icache-loads). These statistics are generated for native, custom allocator and Rapid. The script logs the output for the three cases and stores the files in the folder $HOME/stats/.

4.0.11 Identifying wrapper functions

```
cd $HOME/artifact_rapid/polybench-c-4.2.1-beta

./wrapper.sh

cd $HOME/artifact_rapid/cpuspec
```

```
cp $HOME/artifact_rapid/cpuspec/benchspec/
\ Makefile.defaults $HOME/artifact_rapid/cpuspec/ \
benchspec/Makefile.defaults.orig

cp $HOME/artifact_rapid/spec/Makefile.defaults \
$HOME/artifact_rapid/cpuspec/benchspec/

./wrapper.sh

cp $HOME/artifact_rapid/cpuspec/benchspec/ \
Makefile.defaults.orig $HOME/artifact_rapid/cpuspec/ \
benchspec/Makefile.defaults
```

The names of the wrapper functions appear in the `make.out` file for the corresponding CPU SPEC 2017 benchmarks.

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
2. Chitre, K., Kedia, P., Purandare, R.: The road not taken: Exploring alias analysis based optimizations missed by the compiler. Proc. ACM Program. Lang. **6**(OOPSLA2) (oct 2022). https://doi.org/10.1145/3563316, `https://doi.org/10.1145/3563316`