1. Getting Started Guide

DiscoPoP is an open-source tool that helps software developers parallelize their programs with threads. It is a joint project of Technical University of Darmstadt and Iowa State University.

1.1. Prerequisites

DiscoPoP is built on top of LLVM. Before doing anything, you'll need a basic development setup. We have tested DiscoPoP on Unbuntu and following packages should be installed before installing the profiler:

```
sudo apt-get install git build-essential cmake
```

Additionally, you need LLVM installed on your system. Currently, DiscoPoP supports LLVM 8.0.1. You can download it from the following link:

https://releases.llvm.org/download.html#8.0.1

Lower LLVM versions are not supported, due to API changes which lead to the compilation failure. Please follow the installation tutorial to install LLVM:

https://llvm.org/docs/GettingStarted.html

1.2. Alternatives to skip LLVM installation

If you could install LLVM successfully on your machine, please continue to Section 1.3. Otherwise, we have installed LLVM, DiscoPoP, and our hybrid approach on a docker and our test system. You can use either of them to bypass LLVM installation and evaluate the artifact.

1.2.1. Docker image

We have created a docker image that you can simply log in and run the test programs.

Folder link: https://1drv.ms/u/s!ApjkbbTxJW6bmOJTlv5ZzfU_miqTZg?e=prnEQ1
Password: europar2020

Then, please follow the instruction below to run the docker and consequently the benchmarks.

1. Download the docker image
2. Load the docker image (can take several minutes):

```
docker load –input europar-artifact-105-image.tar
```

3. Run a container from the loaded image:

```
docker run --name=artifact-105 -it europar-artifact-105 bash
```

4. Navigate to benchmarks directory:

```
cd /root/benchmarks
```

5. Please continue to Section 2 to analyze benchmarks with DiscoPoP and our hybrid approach.

NOTE: To start the same container at a later point, run

```
docker start artifact-105
```

and execute an interactive shell in it:

```
docker exec -it artifact-105 bash
```

1.2.2. Our test system

Below are the access information to our evaluation system:

The access info has been deleted. Please contact Mohammad Norouzi (norouzi@cs.tu-darmstadt.de), if you encountered problems.

We have already installed DiscoPoP and our approach on the test system. Therefore, you can continue to Section 2 to try DiscoPoP and our hybrid approach.

1.3. Installation of DiscoPoP and our approach

Once you have installed LLVM, the installation of DiscoPoP profiler and our approach is as easy as running cmake and make!

You can find the data-dependence profiler of DiscoPoP in "DPInstrumentation" directory. The source code for our approach which is proposed in the paper is in "DPInstrumentationOmission".

For the installation, first create a build directory:

```
mkdir build; cd build;
```

Next configure cmake. The preferred LLVM installation path for DiscoPoP can be set using the -DLLVM_DIST_PATH=<PATH_TO_LLVM_BUILD_FOLDER> cmake variable.

```
cmake -DLLVM_DIST_PATH=<PATH_TO_LLVM_BUILD_FOLDER> ..
```

Once the configuration process is successfully finished, run `make` to compile the profiler to obtain the profiling libraries.

2. Step-by-Step Instructions on how to reproduce the results

We compiled the benchmarks using clang 8.0.1, which is also used by the data-dependence profiler of DiscoPoP. We ran the benchmarks on an Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz with 64Gb of main memory, running Ubuntu 14.04 (64-bit edition). We profiled the benchmarks using the inputs packaged with the programs. You can find the benchmarks that we used for the evaluation in test/testedBenchmarkSuites.

Please find the execution times that we recorded in the excel sheet in the root directory (Statistics.xlsx).

2.1. General instructions to use DiscoPoP and our approach

In the following, we will explain how to run the data-dependence profiler of DiscoPoP and our approach. However, before the execution, run the `dp-fmap` script in the root folder of the target application to obtain the list of files. The output will be written in a file named `FileMapping.txt`.

```
<DISCOPOP_PATH>/scripts/dp-fmap
```

Please use the exact compiler flags that we used. Otherwise, you might not get the correct results, or the analysis might fail.

2.1.1. Profiling with DiscoPoP

To obtain data dependences, we need to instrument the target application:

```
clang -g -O0 -S -emit-llvm -fno-discard-value-names -Xclang -load -Xclang <PATH_TO_DISCOPOP_BUILD_DIR>/libi/LLVMDPInstrumentation.so -mllvm -fm-path -mllvm ./FileMapping.txt -c <C_File> -o out.ll
```

If your test project has multiple files, please use llvm-link to link multiple .ll files into one:

```
llvm-link out.ll other.ll -o out.ll
```

Make the application executable:

```
clang++ out.ll -L<PATH_TO_DISCOPOP_BUILD_DIR>/rtlib -lDiscoPoP_RT -lpthread -o <APP_NAME>
```

Running the application will result in a text file containing all the dependences located in the present working directory

```
./<APP_NAME>
```

2.1.2. Profiling with our hybrid approach

You may run the DPInstrumentationOmission pass to omit non-essential instructions from profiling to improve the performance without harming the accuracy of the extracted data dependences.

First, we need to instrument the code:

clang -g -O0 -emit-llvm -fno-discard-value-names -Xclang -load -Xclang <PATH_TO_DISCOPOP_BUILD_DIR>/libi/LLVMDPInstrumentation.so -mllvm -fm-path -mllvm ./FileMapping.txt -c <C_File> -o out.ll

Then, we eliminate instrumenting memory-access instructions whose data dependences can be identified statically. Not to miss any data dependences, we extract the data dependences statically. Please run the following command to extrac data dependences statically and eliminate the instrumentations:

opt -S -load=<PATH_TO_DISCOPOP_BUILD_DIR>/libi/LLVMDPInstrumentationOmission.so -dp-instrumentation-omission -stats out.ll -o out.ll

Finally, make the application executable:

clang++ out.ll -L<PATH_TO_DISCOPOP_BUILD_DIR>/rtlib -lDiscoPoP_RT -lpthread -o <APP_NAME>

Running the application will result in a text file containing all the dependences located in the present working directory

./<APP_NAME>

2.2. Walk-through example

The following walk-through example demonstrates how to use DiscoPoP and our approach to analyze a sequential sample application. In this example, we use the program fib. We assume that you have successfully installed DiscoPoP and our approach. We have created a build directory which contains the installation files of DiscoPoP and our approach.

First, switch to the test/fibonacci folder which contains the program fib.c. Then, please run the following commands step-by-step to obtain the desired results.

Run the dp-fmap script to obtain the list of files. The output will be written in a file named FileMapping.txt.

../../scripts/dp-fmap

To obtain data dependences, we need to instrument the application and run it.

clang -g -O0 -fno-discard-value-names -emit-llvm -Xclang -load -Xclang ../../build/libi/LLVMDPInstrumentation.so -mllvm -fm-path -mllvm ./FileMapping.txt -c fib.c -o fib-discopop.ll

clang++ fib-discopop.ll -L../../build/rtlib -lDiscoPoP_RT -lpthread -o fib-discopop

With the following command, we run the program to obtain data dependences with DiscoPoP profiler. We used time command in Linux to get the profiling time. Please feel free to use any other time measurement commands/tools etc.

```
time ./fib-discopop
```

The output is a text file that contains all the dependences. For the example, you can find the data dependences in test/fibonacci/fib-discopop_dep.txt. A data dependence is represented as a triple <sink, type, source>. type is the dependence type (RAW, WAR or WAW). Note that a special type INIT represents the first write operation to a memory address. source and sink are the source code locations of the former and the latter memory access, respectively. sink is further represented as a pair <fileID:lineID>, while source is represented as a triple <fileID:lineID|variableName>. The keyword NOM (short for "NORMAL") indicates that the source line specified by aggregated sink has no control-flow information. Otherwise, BGN and END represent the entry and exit points of a control region, respectively.

Further, we use our approach to remove the instrumentation of memory-access instructions whose data dependences can be identified statically by our approach. Again, we need to execute the instrumentation step on the fib program. Please note that we could reuse the instrumented file in the previous step. However, we repeated it for the sake of clarity.

```
clang -g -O0 -fno-discard-value-names -emit-llvm -Xclang -load -Xclang ../../build/libi/LLVMDPInstrumentation.so -mllvm -fm-path -mllvm ./FileMapping.txt -c fib.c -o fib-our-approach.ll
```

```
opt -S -load=../../build/libi/LLVMDPInstrumentationOmission.so -dp-instrumentation-omission -stats fib-our-approach.ll -o fib-our-approach.ll
```

```
clang++ fib-our-approach.ll -L../../build/rtlib -lDiscoPoP_RT -lpthread -o fib-our-approach
```

```
time ./fib-our-approach
```

The output is again a txt file (i.e., ./test/fibonacci/fib-our-approach_dep.txt) which contains data dependences extracted by our approach.

Please compare the profiling time with and without our approach (i.e., the execution time of "fib-discopop" and the execution time of "fib-our-approach"). They are significantly different (around 80% reduction in the profiling time).

Also, please compare the data dependences identified by both tools. Our approach identifies two more data dependences:

> 1:19 NOM RAW 1:16|n RAW 1:18|res

Both of them are transitive data dependences as explained in the paper.

2.3. Testing the evaluated benchmarks in the paper

This section describes how to run the tests which provided the results of our paper. Three benchmark suites were used: BOTS, Polybench, and NPB. We have prepared scripts to facilitate the evaluation process. To use the scripts, you need to set environment variables (Section 2.3.1). You can find instructions to instrument memory accesses in Section 2.3.2, and profile the benchmarks with DiscoPoP and our approach in Section 2.3.3. In the end, you can compare the data dependences identified by each method with the script provided in Section 2.3.4.

2.3.1. Prepare environment variables (Can be skipped if using Docker Image)

Store the path to both the DiscoPoP and the LLVM build directories in the following environment variables:

```
export LLVM_BUILD=<PATH_TO_YOUR_LLVM_BUILD>
export DP_BUILD=<PATH_TO_YOUR_DP_BUILD>
```

Note: Please make sure that you correctly set the variables. Otherwise, you will run into segmentation fault errors most probably.

2.3.2. Instrument memory accesses in the benchmarks

2.3.2.1. BOTS

Please run the following commands:

```
cd bots
<PATH_TO_CLONED_DP_REPO>/scripts/dp-fmap .
./configure
##Select “dp”
cd serial
make
cd ..
```

2.3.2.2. Polybench

```
cd Poly
<PATH_TO_CLONED_DP_REPO>/scripts/dp-fmap .
./compile-all.sh
# Select “dp”
```

2.3.2.3. NAS

```
cd NAS
<PATH_TO_CLONED_DP_REPO>/scripts/dp-fmap .
./make-all.sh <CLASS>
```

Where <CLASS> is one of the following:
* S: small for quick test purposes
* W: workstation size (a 90's workstation; now likely too small)
* A, B, C: standard test problems; ~4X size increase going from one class to the next
* D, E, F: large test problems; ~16X size increase from each of the previous classes

Two programs (i.e., DC and UA) in the benchmark suite are not evaluated. They are not compilable on our test system and we did not claim/provide timings for them.

2.3.3. Profile benchmarks with DiscoPoP and our approach

For all 3 benchmark suites, the process to run the DiscoPoP Profiler and our approach is the same and is described in the steps below.

Note: Please cd into the respective benchmark directory before running the commands below.

2.3.3.1. Profiling with DiscoPoP

```
./make-all-runable.sh
## Select “dp”
./run-all.sh <N> DEPS_VANILLA
```

Replace \<N> with the desired number of runs per program. The identified dependences will be in the `<Benchmark Suite>/DEPS_VANILLA` folder and the performance results will be under `<Benchmark Suite>/DEPS_VANILLA/run-all.log`

NOTE: Profiling the following programs in Polybench takes a long time:

```
fdtd-2d, fdtd-apml, gemm, gemver, gesummv, syrk, trmm
```

2.3.3.2. Profiling with our hybrid approach

```
./make-all-runable.sh
## Select “dp+SPA”
./run-all.sh <N> DEPS_OMISSIONS
```

Replace \<N> with the desired number of runs per program. The identified dependences will be in the `<Benchmark Suite>/DEPS_OMISSIONS` folder and the performance results will be under `<Benchmark Suite>/DEPS_OMISSIONS/run-all.log`

NOTE: You can modify the list of programs which are run directly in the `run-all.sh` script in the directory of each benchmark suite.

NOTE: You can run a single target program \<N> times by runnning:

```
./run-n-times <N> run_custom/run-<TARGET>.sh
```

NOTE: You may observe a significant difference in the reported benchmarks run times and the actual wall-clock times. It is because the wall-clock time includes the waiting time for system resources. At the end of the profiling time, DiscoPoP spends some time dumping the dependences in a file. We do not consider the dependence-dumping time because it is not the time which is spent on profiling the data dependences.

2.3.4. Compare dependences identified by DiscoPoP and our approach

You can use our comparison script to facilitate the comparison of dependence files generated by DiscoPoP profiler and our approach. In each of the benchmark folders, after following the instructions to profile the programs with both methods, run the following script:

```
./compare-all-deps.sh DEPS_VANILLA DEPS_OMISSIONS
```

The script will compare all of the dependences identified for each program in each of the folders DEPS_VANILLA and DEPS_OMISSIONS. It can identify most transitive dependences and will not print these.

Note that the compare script can identify some but not all transitive dependencies. The results of the comparison usually still contains other transitive dependences.

License