A. Artifact Appendix

A.1 Abstract

This artifact description contains information about the complete workflow required to set up and reproduce experiments in ALDA. We describe how the software can be obtained and the build process as well as necessary preprocessing steps to generate the test program and baseline to run. All the programs and benchmarks are publicly available except for the SPEC 2006 benchmark.. In addition, we provide a VM with all the programs and input data prepared and as well as instructions on how to build such a VM.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Compiler optimizations like common sub-expression elimination and special optimizations based on ALDA like map fusion.
- **Program:** LLVM dev tools, wllvm, cmake, make.
- **Compilation:** LLVM 6.0, clang-6.0 and Python3 under Ubuntu 18.04 Desktop LTS
- **Transformations:** LLVM IR instrumentation implemented as LLVM pass
- **Binary:** LLVM-dev tools, wllvm, SPLASH2, SPEC2006, memcached, ffmpeg, sort, nginx
- **Hardware:** For CPU, we recommend to use Intel(R) Xeon(R) CPU E5-2630 v4 or any Intel Xeon later than this with at least four cores. For memory, we need at least 32GB memory.
- **Execution:** Linux shell
- **Metrics:** Use the modified version divides the original version to get the normalized overhead
- **Output:** Wall time of each program execution.
- **How much disk space required (approximately)?:** At least 256GB
- **How much time is needed to prepare workflow (approximately)?:** At least 24 hours to finish subsection A.5.
- **How much time is needed to complete experiments (approximately)?:** Each experiments are divided into 3 components, each component requires at least 24 hours, usually less than 48 hours to finish its six times execution.
- **Publicly available?:** See subsubsection A.4.2
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.5748339

A.3 Importing Virtual Machine

Remember to increase the RAM size of the VM to at least 32GB and ensure there are at least 4 cores. This VM is created by laptop so it default RAM size is only 4GB. The username of the VM is **anony** and the password is **password**.

A.4 Description

A.4.1 Content

We provide following 3 ways to evaluate:

1. If you want to use the VM to rerun the experiment, you can import the VM to your machine. Remember to add more disk space and memory to meet our basic requirement. Then you only need to read subsection A.5 and then you can jump to subsection A.9.
2. If you have your own machine (meet our requirements), you can download the experiments.zip in our submission and decompress it. It contains all the executable and bitcode files for the experiments. Then you only need to finish subsection A.5 and jump to subsection A.9.
3. If you want to fully reproduce the experiment from a Ubuntu machine, please follow all the steps starting subsection A.5.

Please select your way to reproduce from above 3 options. If you pick up the 3rd option, we strongly recommend you to build a folder with the same structure as our experiments folder and copy all our inputs and scripts into it.

A.4.2 How to access

Below we make a list of all the experiment benchmarks and programs needed to reproduce the result:

- **ALDAcc**
  https://github.com/CXWorks/alda-analysis
- **Hand written Eraser**
  https://github.com/CXWorks/hand_eraser
- **SPLASH2**
  https://github.com/staceyson/splash2
- **Memcached-1.6.2**
  http://www.memcached.org/files/memcached-1.6.2.tar.gz
- **Memtier_benchmark-1.3.0**
  https://github.com/RedisLabs/memtier_benchmark/releases/tag/1.3.0
- **Nginx-1.7.9**
  http://nginx.org/download/nginx-1.7.9.tar.gz
- **FFmpeg-4.2.2**
  https://ffmpeg.org/releases/ffmpeg-4.2.2.tar.gz
- **Test video**
  http://samples.ffmpeg.org/V-codecs/h264/HD-h264.ts
- **GNU coreutils sort-8.32**
  https://github.com/coreutils/coreutils/releases/tag/v8.32

A.4.3 Hardware dependencies

We don't have specific feature requirements for CPU, but we recommend to use Intel(R) Xeon(R) CPU E5-2630 v4 or any Xeon later than this with at least four cores. And we require at least 32GB memory and 256GB disk to run the experiments.

A.4.4 Software dependencies

Except for the programs/benchmarks listed above, we have following software dependencies. They can be installed automatically through package management tools under Ubuntu 18.04 Desktop LTS.

1. Flex 2. Bison 3. CMake 4. C++ Boost 5. Python3 6. Pythoin-pip3 7. Python-fire 8. Python-Cheetah 9. Python-wllvm

A.5 Basic Installation

A.5.1 Install apt packages

```
apt update && apt install git flex bison
    cmake llvm-dev clang build-
   essential autoconf autopoint texinfo
    gperf subversion g++ python-dev
   autotools-dev libicu-dev libbz2-dev
   libboost-all-dev libssl-dev libevent
   -dev apache2-utils python3-venv
   python3-pip git
```

Note: Some ubuntu 18.04s' default LLVM version is 6.0.0, which is also accetptale.

A.5.2 Install Python related packages

```
pip3 install fire
pip3 install Cheetah3
```

A.5.3 Install ALDAcc

```
git clone https://github.com/CXWorks/
    alda-analysis.git
```

Then rename it:

```
mv alda-analysis auto-analysis
```

Note: Our test driver assume ALDAcc is installed in the home folder of current user.

A.5.4 Install Hand written Eraser

```
git clone https://github.com/CXWorks/
    hand_eraser.git
```

Note: Our test driver assume hand written Eraser is installed in the home folder of current user.

A.5.5 Install Nginx Workspace

Note: If you are using the VM provided by us, you can skip this.

The whole working directory is under experiments/input/nginx. When compiling under VM, we use a fixed work path as /home-/anony/nginx as its workspace. So you need to create the same folder under your machine and either use a softlink to link the content or copy everything from experiments/input/nginx into your local /home/anony/nginx folder.

!!Checkpoint!! Now you should be able to run nginx under experiments/origin/nginx. After launching the server, you should be able to visit http://127.0.0.1:8020 in your machine.

A.6 Install Original Benchmark

This section shows the steps how to compile and run the benchmarks without any modification. This original version program will be used as the unit in our following normalization.

A.6.1 Install SPLASH2

```
git clone https://github.com/staceyson/
    splash2.git
```

- **Setup build tools** Our experiment programs are in codes/kernel or codes/apps folders. You need to manually navigate into each program's folder(e.g. codes/kernels/fft) and delete an empty file named *makefile* (all in lower case) while keeping the *Makefile*.

- **Setup compiler flags** Navigate to codes/Makefile.config, change the item in line 13 named *BASEDIR* to the correct path (**Attention: must use absolute path**). And change the *CFLAGS* to optimization level 2 (-O2). Finally, we need to remove -static flags in both CFLAGS and LDFLAGS.

- **codes/apps/volrend** Navigate to the folder and decompress the libtiff.tar.Z. Goto the decompressed folder and open Makefile, for *CONF_LIBRARY* add four more lines below:

```
  -Du_char="unsigned char" \
  -Du_int="unsigned int" \
  -Du_short="unsigned short" \
  -Du_long="unsigned long" \
```

Then under the libtiff folder, you should be able to compile this dependent library.

- **codes/apps/radiosity** Navigate to the folder and find two folders named glibps and glibdumb. Go to each folder and simply run make to build these dependencies.

!!Checkpoint!! Now you should be able to compile all the programs in this SPLASH2 benchmark and since we don't make any modifications to the source code or IR, this will be used as our "original" version of programs.

A.6.2 Install SPEC 2006 Int

Due to license limitation, we didn't include the source code of SPEC 2006 into VM, but here we give instructions on how to prepare experiment environment for SPEC 2006 Int benchmark after you successfully installed it.

- **Setup compile configuration**

```
cp config/Example-linux64-amd64-gcc43+.
    cfg config/default.cfg
```

Note: If your gcc version is old or you can't find the cfg file, you can also use Example-linux64-amd64-gcc43.cfg.

Then use editor to open config/default.cfg and navigate to line 79, change default C compiler to clang and default C++ compiler to clang++.

- **Setup compile flag**

Use editor open benchspec/Makefile.defaults and find line 46, line 56. Add "-std=gnu89" option for line 46 and "-std=gnu++98" option for line 56.

!!Checkpoint!! Now you should be able to run SPEC Int "original" programs without any modifications:

```
source shrc && runspec -a run int
```

A.6.3 Install Memcached

Use following commands:

```
wget http://www.memcached.org/files/
    memcached-1.6.2.tar.gz && tar zxf
    memcached-1.6.2.tar.gz && cd
    memcached-1.6.2/ && CC=clang ./
    configure && make
```

!!Checkpoint!! Now you should have an executable memcached program in the folder.

A.6.4 Install Memtier Benchmark

```
wget https://github.com/RedisLabs/
    memtier_benchmark/archive/refs/tags
    /1.3.0.tar.gz && tar zxf 1.3.0.tar.
    gz && cd memtier_benchmark-1.3.0/ &&
     autoreconf -ivf && ./configure &&
    make && make install
```

!!Checkpoint!! Now you should have *memtier_benchmark* in your shell and you can launch memcached in one terminal and use memtier benchmark to test it by:

```
memtier_benchmark -p 11211 -P
    memcache_text
```

A.6.5 Install Nginx

```
wget http://nginx.org/download/nginx
    -1.7.9.tar.gz && tar zxf nginx
    -1.7.9.tar.gz && cd nginx-1.7.9/ &&
    autoreconf -ivf && CC=clang ./
    configure --prefix=$HOME/nginx &&
    make
```

Note: Sometimes the *autoreconf* may fail, you can continue building if there is *configure* file around.

Here we recommend you set prefix to any folder under current user instead of using the default value.

!!Checkpoint!! Now you should have an executable nginx server under objs folder. To launch it, you need to create folders and configuration files in your prefix path folder. These files includes the testing html can be found in VM.

A.6.6 Install FFmpeg

```
wget https://ffmpeg.org/releases/ffmpeg
    -4.2.2.tar.gz && tar zxf ffmpeg
    -4.2.2.tar.gz && cd ffmpeg-4.2.2/ &&
     ./configure --cc=clang --disable-
    x86asm && make && wget http://
    samples.ffmpeg.org/V-codecs/h264/HD-
    h264.ts
```

!!Checkpoint!! Now you should have an executable ffmpeg decoder with a sample video HD-h264.ts. You can run following command to transform it:

```
./ffmpeg -i HD-h264.ts -f rawvideo -
    threads 4 -y -target ntsc-dv /dev/
    null 2>&1
```

A.6.7 Install GNU coreutils sort

```
git clone https://github.com/coreutils/
    coreutils.git && cd coreutils && git
     checkout v8.32 && ./bootstrap  &&
    CC=clang ./configure && make
```

!!Checkpoint!! Now you should have an executable sort prohram in objs folder. To you can find a randomly generated file in VM and use following command to run it

```
./sort  -n --parallel=4 rand.txt > /dev/
    null
```

A.7 Build MSan Version Benchmark

This section shows the step to compile the benchmarks with LLVM MemorySanitizer. If you followed the steps in subsection A.6, you need to **clean up all the previous outputs and flag settings** and rebuild the program.

A.7.1 Build MSan version of SPLASH2

In ALDApaper, one important experiments is the performance comparison with MemorySanitizer. To reproduce the result, we need to build a MSan verison of above programs. According to MemorySanitier's instruction in https://github.com/google/sanitizers/wiki/MemorySanitizer#using-memorysanitizer and in our paper's page 9 footnote 5, we need to add following compiler flags to the *CFLAGS* in codes/Makefile.config:

```
-Qunused-arguments -fPIE -pie -fsanitize
    =memory -mllvm -msan-handle-icmp=0 -
    mllvm -msan-check-access-address=0 -
    fno-omit-frame-pointer
```

!!Checkpoint!! Now you should be able to successfully build each program and if you run with following environment, it will print help info in the shell.

```
MSAN_OPTION=help=1 ./FFT
```

A.7.2 Build MSan version of SPEC 2006 Int

Use editor to open benchspec/Makefile.defaults and add following flags for EXTRA_CFLAGS and LDCFLAGS.

```
-fPIE -pie -fsanitize=memory -mllvm -
    msan-handle-icmp=0 -mllvm -msan-
    check-access-address=0 -fno-omit-
    frame-pointer
```

!!Checkpoint!! Now you should be able to successfully build each program. However, since SPEC changed the program's default IO, we can't see MSan's help info print.

A.7.3 Build MSan version of Memcached

In the configure step, use following command to run and then rebuild the project:

```
CC=clang CFLAGS="-O2 -Qunused-arguments
    -fPIE -pie -fsanitize=memory -mllvm
    -msan-handle-icmp=0 -mllvm -msan-
    check-access-address=0 -fno-omit-
    frame-pointer" ./configure
```

!!Checkpoint!! Now you should be able to successfully build each program. However, use objdump can see there are many msan related function calls in the binary.

A.7.4 Build MSan version of Nginx

In the configure step, use following command to run and then rebuild the project:

```
CC=clang CFLAGS="-O2 -Qunused-arguments
    -fPIE -pie -fsanitize=memory -mllvm
    -msan-handle-icmp=0 -mllvm -msan-
    check-access-address=0 -fno-omit-
    frame-pointer" ./configure --prefix
    =/home/anony/nginx
```

- **Linking Nginx with MSan** Unfortunately, nginx's configure script doesn't support adding linker flags, so it's an expected error here for the last linking step and we need to manually run it. Noted the final link command will be printed like below and it contains many lines:

```
clang -o objs/nginx \
objs/src/core/nginx.o \
 ...(omit)
objs/ngx_modules.o \
-lcrypt -lpcre -lcrypto -lcrypto -lz
```

We need to modify this command like this and rerun it to build our program:

```
clang -fsanitize=memory \
-mllvm  -msan-handle-icmp=0 \
-mllvm -msan-check-access-address=0 \
-fno-omit-frame-pointer \
-o objs/nginx \
objs/src/core/nginx.o \
 ...(omit)
objs/ngx_modules.o \
-lcrypt -lpcre -lcrypto -lcrypto -lz \
-lpthread
```

!!Checkpoint!! Now you should be able to successfully build nginx. Besides, use objdump can see there are many msan related function calls in the binary.

A.7.5 Build MSan version of FFmpeg

Because FFmpeg uses a special test for compiler, MSan can't pass it during "configure" phrase. So we need to use following command to generate the Makefile and modify it directly.

```
./configure --cc=clang --disable-x86asm
```

Then use editor to open ffbuild/config.mak and add below flags to the beginning of line 48 *CFLAGS*.

```
-O2 -fPIE -pie -Qunused-arguments -
    fsanitize=memory -mllvm -msan-handle
    -icmp=0 -mllvm -msan-check-access-
    address=0 -fno-omit-frame-pointer
```

and for line 73 *LDFLAGS* add following flags then build the program:

```
-fsanitize=memory -mllvm -msan-handle-
    icmp=0 -mllvm -msan-check-access-
    address=0 -fno-omit-frame-pointer
```

!!Checkpoint!! Now you should be able to successfully build ffmpeg. Besides, use objdump can see there are many msan related function calls in the binary.

A.7.6 Build MSan version of GNU coreutils

In the configure step, use following command to run and then rebuild the project:

```
CC=clang CFLAGS="-O2 -Qunused-arguments
    -fPIE -pie -fsanitize=memory -mllvm
    -msan-handle-icmp=0 -mllvm -msan-
    check-access-address=0 -fno-omit-
    frame-pointer" ./configure
```

!!Checkpoint!! Now you should be able to successfully build sort under src/ folder. Besides, use objdump can see there are many msan related function calls in the binary.

A.8 Extract LLVM IR Version of Benchmark

This section shows the step to compile the benchmarks into LLVM IR format. If you followed the steps in subsection A.6 or subsection A.7, you need to **clean up all the previous outputs and flag settings** and rebuild the program.

A.8.1 Install wllvm

Use following command to install(for Ubuntu 18.04 LTS, you need to manually install it, not through pip3):

```
git clone https://github.com/travitch/
    whole-program-llvm && cd whole-
    program-llvm && sudo pip3 install -e
     .
```

!!Checkpoint!! Open a shell and run wllvm, it should print a CRITICAL error message.

From above sections, we already get the original executable programs and MSan instrumented program. Now we need to extract an IR version of the program for ALDAcc. For most cases, wllvm can help us to extract a single bitcode(bc) file from a project. The instructions on how to use it can be found in https://github.com/travitch/whole-program-llvm. In a nutshell, the following 5 steps are needed:

1. export LLVM_COMPILER=clang in the shell environment
2. Set the default compiler to wllvm instead of clang
3. Refer to our paper sec 5.6.1, we need to add two more compiler flags "-fno-vectorize -fno-slp-vectorize" and ensure the same opt level 2 for this build
4. Build the project as previous instructions
5. Navigate the the same folder of executables, run extract-bc with program name to get the bitcode file

!!Attention!! Remember to remove the MSan related compiler flags when you do this!

A.8.2 Extract IR form of SPLASH2

Use editor to open codes/Makefile.config and make following changes:

1. Set default CC to wllvm
2. Add compiler flag "-fno-vectorize -fno-slp-vectorize"

- **Extract IR** Then goto each program folder and build it. After that, run "extract-bc $(programname)" to get the IR file of program.

!!Checkpoint!! For the IR file you get, they can be easily linked to be executables by running(take FFT as example):

```
clang -O2 FFT.bc -lpthread -lm
```

A.8.3 Extract IR form of SPEC 2006 Int

1. Navigate to file config/default.cfg line 79, set the default CC to wllvm
2. Navigate to benchspec/Makefile.defaults line 46 and add compiler flag "-fno-vectorize -fno-slp-vectorize"
3. Goto each program's build folder(e.g. bzip2's build folder is benchspec/CPU2006/401.bzip2/build/build_base_gcc43-64bit.0000/) and run extract-bc to get the IR file.

!!Checkpoint!! For the IR file you get, they can be easily linked to be executable(with corresponding libraries) by running(take bzip2 as example):

```
clang -O2 bzip2.bc -lpthread -lm
```

A.8.4 Extract IR form of Memcached

Run following command to get the IR file(remember to export LLVM_COMPILER variable first):

```
CC=wllvm CFLAGS="-O2 -Qunused-arguments
    -fPIE -pie -fno-vectorize -fno-slp-
    vectorize" ./configure && make &&
    extract-bc memcached
```

!!Checkpoint!! For the IR file you get, they can be easily linked to be executable(with corresponding libraries) by running:

```
clang -O2 memcached.bc -lpthread -levent
```

A.8.5 Extract IR form of Nginx

Run following command to get the IR file:

```
CC=wllvm CFLAGS="-O2 -Qunused-arguments
    -fPIE -pie -fno-vectorize -fno-slp-
    vectorize" ./configure --prefix=/
    home/anony/nginx && make && extract-
    bc objs/nginx
```

!!Checkpoint!! For the IR file you get, they can be easily linked to be executable(with corresponding libraries) by running:

```
clang -O2 objs/nginx.bc -lcrypt -lpcre -
    lcrypto -lz -lpthread
```

A.8.6 Extract IR form of Nginx

Run following command to get the IR file:

```
CC=wllvm CFLAGS="-O2 -Qunused-arguments
    -fPIE -pie -fno-vectorize -fno-slp-
    vectorize" ./configure --prefix=
    $HOME/nginx && make && extract-bc
    objs/nginx
```

!!Checkpoint!! For the IR file you get, they can be easily linked to be executable(with corresponding libraries) by running:

```
clang -O2 objs/nginx.bc -lcrypt -lpcre -
    lcrypto -lz -lpthread
```

A.8.7 Extract IR form of FFmpeg

Run following command to get the IR file:

```
CFLAGS="-O2 -Qunused-arguments -fPIE -
    pie -fno-vectorize -fno-slp-
    vectorize" ./configure --cc=wllvm --
    disable-x86asm && make && extract-bc
     ffmpeg
```

!!Checkpoint!! For the IR file you get, they can be easily linked to be executable(with corresponding libraries) by running:

```
clang ffmpeg.bc -lpthread -lz -lm -lbz2
```

A.8.8 Extract IR form of GNU coreutils sort

Run following command to get the IR file:

```
CC=wllvm CFLAGS="-O2 -Qunused-arguments
    -fPIE -pie -fno-vectorize -fno-slp-
    vectorize" ./configure && make &&
    extract-bc src/sort
```

!!Checkpoint!! For the IR file you get, they can be easily linked to be executable(with corresponding libraries) by running:

```
clang -O2 src/sort.bc -lpthread
```

A.9 Experiment workflow

We give the methods to build 3 different types of benchmark programs: origin, MSan and IR version. The whole workflow of our experiments are listed below:

For figure 3, in our paper page 9, we need to first run the origin program, then MSan program and finally apply ALDA MSan on IR and run them. We use the later two runs' result divides the origin programs' runtime to get the normalized overhead, then we can draw the graph. **The expected result of this graph is that ALDA MSan achieves the same or slightly faster performance compared with MSan.**

For figure 5, we need ALDAcc to first apply Eraser, Fasttrack, Use-after-free, Index taint tracking individually on IR version of the program, then run a collected version of all the analyses. We can use the same origin programs' time in figure 3 and compute the normalized overhead. **The expected result of this graph is that the combined analysis's execution time should show a 44.9% speed up compared with the sum of all individual analyses.**

Finally for figure 4, in our paper page 10, we run origin program and then ALDA Eraser and hand-written Eraser. Then we compute the normalized overhead and draw the graph. To save time, we recommend to use the result from previous experiments as ALDA's result and only run the hand version. **The expected result of this graph is that ALDA Eraser achieves the same or slightly faster performance compared with hand written version.**

Program	Command	Notes
fft	./fft -p2 -m26	
lu_c	./lu -p2 -n4096	
lu_nc	./lu -p2 -n4096	
radix	./radix -p2 -n268435456	
cholesky	./cholesky -p2 inputs/tk29.O	Decompress the file first
barnes	./barnes <input	Modify input file: line 2 : 16384 ->131072 line12: 1 ->2
fmm	./fmm <input/input.16384	Modify input.16384 file: line 3: 16384 ->131072 line 5: 1 ->2
ocean_c	./ocean -p2 -n4098	
raytrace	./raytrace -p2 inputs/balls4.env	Decompress the file first
water_ns	./water <input	Modify input file: line 1 : 3 ->210 line3: 1 ->2, 6.212752 ->0
volrend	./volrend 2 inputs/head	Decompress the file first
radiosity	./radiosity -p 2 -bf 0.005 -batch -room -largeroom	

Table 1. Input argumnets for SPLASH2 benchmark

Program	Command
memcached	./memcached -l 0.0.0.0 & memtier_benchmark -p 11211 -P memcache_binary
nginx	./nginx & ab -n 1000000 -c 16 http://127.0.0.1:8020/
ffmpeg	./ffmpeg -nostdin -loglevel panic -i HD-h264.ts \ -f rawvideo -threads 4 -y -target ntsc-dv /dev/null 2>&1
sort	./sort -n –parallel=4 rand.txt >/dev/null

Table 2. Input argumnets for applications

A.10 Input arguments

In this section we list all the input arguments for each benchmark programs. The program's inputs remain the same across all the experiments. For SPEC 2006 Int, we use the standard run arguments in their benchmark (e.g. bizp2's input data can be found under benchspec/CPU2006/401.bzip2/data folder). For SPLASH2 benchmark, we list the arguments in Table 1. Besides, the input arguments for applications are listed in Table 2.

A.10.1 Nginx Configuration and Workspace

In the previous build process, we have assigned a workspace for Nginx. Please create that folder and move the files under experiments/input/nginx in our VM to that folder, then you should be able to visit our test page in your localhost with port number 8020.

A.10.2 MSan Input Environment

Refer to our paper page 9 footnote 5, we need to add following environment before every MSan run:

```
MSAN_OPTIONS=strict_string_checks=1:
    strict_memcmp=1:handle_ioctl=0:
    allocator_may_return_null=1
```

Note: For MSan version of *sort* program, you need to add one more option as **allocator_may_return_null=1** to avoid panic after one allocation failed.

A.10.3 Timer

For the timing of our experiment, SPLASH2 benchmark already has its built-in timer, we use that for SPLASH2. For Nginx and Memcached, the benchmark tools will generate the running speed, we will normalize it to its runtime. For other benchmarks or applications, we use **time** linux command to get the wall time (*real* time of the three output) as the program's performance.

A.11 Evaluation and Interpret Expected Results

A.11.1 Correction

We need to correct one experiment result for MSan (figure 3), which is program **fmm** in SPLASH2. Due to the same reseaon as **barnes** in table 3, LLVM MSan will generate error report without adding special handler(and ALDA MSan already has the handler). We will remove this result from figure 3 to table 3. And this correction actually enhanced our result.

A.11.2 Reproduce ALDA MSan Experiment (Figure 3)

- **Reproduce in Option 1 & 2** In the VM, goto `~/experiments` and run (count is the number of times to run each programs, we recommend use at least 3):

```
./auto_msan.sh $count > msan.txt 2>&1
python3 alda_msan.py
```

- **Reproduce by yourself** To reproduce, you need to run SPLASH2, SPEC Int and 4 applications. The input can be found in Table 1 and Table 2. Specifically, you need to run their original program, LLVM MSan version and ALDA MSan version. Then use MSan versions divide original version to get the normalized overhead.

A.11.3 Reproduce ALDA Combined Analysis Experiment (Figure 5)

- **Reproduce in Option 1 & 2** In the VM, goto `~/experiments` and run (count is the number of times to run each programs, we recommend use at least 3):

```
./auto_combined.sh $count > combined.txt
    2>&1
python3 alda_combined.py
```

- **Reproduce by yourself** To reproduce, you need to run SPLASH2, SPEC Int and three applications. The input can be found in Table 1 and Table 2. To generate the graph, you need to run Eraser, Fasttrack, Use-after-free and Index-taint-tracking and the combined version. Then you can compare the sum of individual analyses with their combined version to generate the graph and borrow the orginal program runtime in subsubsection A.11.2.

A.11.4 Reproduce Eraser Experiment (Figure 4)

- **Reproduce in Option 1 & 2** In the VM, goto `~/experiments` and run (count is the number of times to run each programs, we recommend use at least 3):

```
./auto_eraser.sh $count > hand.txt 2>&1
python3 alda_eraser.py
```

- **Reproduce by yourself** To reproduce, you need to run SPLASH2 in with ALDA Eraser and hand written Eraser. The input can be found in Table 1. Specifically, you need to run the original program, hand written Eraser version and ALDAEraser version. Then use two Eraser versions divide original version to get the normalized overhead.

A.12 Experiment customization

All the program's workload can be changed. Some might require larger memory when running combined analysis, but most are fine with current settings. The number of threads to run can't be changed.