Compacting Points-To Sets Through Object Clustering (Artifact)

MOHAMAD BARBAR, University of Technology Sydney, Australia and CSIRO's Data61, Australia
YULEI SUI, University of Technology Sydney, Australia

1 OVERVIEW

This artifact is for *Compacting Points-To Sets Through Object Clustering* by Mohamad Barbar and Yulei Sui published at OOPSLA 2021. It is packaged as a Docker image and can reproduce the tables in Section 5 (Evaluation) of the paper.

Section 2 of this document provides instructions on setting up the environment and running a quick experiment to verify everything works. Section 3 provides instructions on reproducing our evaluation and varying the experiment through changes to source code, limits, and compiling new programs for analysis. Specifically, Section 3.2 shows how our artifact supports our claims.

1.1 Dependencies

This artifact requires Docker (tested with version 19.03.6 on an Ubuntu machine) and the initial instructions for loading the image and starting a container assume a UNIX shell. An AMD64/x86-64 machine is also required. In order of preference, the CPU should be able to run programs compiled for the Knights Landing microarchitecture (`-march=knl`), the Haswell microarchitecture (`-march=haswell`), or otherwise, in which case any relatively modern AMD64/x86-64 CPU is fine. This is because our work uses some SIMD and Knights Landing provides AVX-512. Most users should have no trouble running code targetting Haswell but processors supporting Knights Landing may be less common.

1.2 Container Layout

Within the container (Section 2.1), from `$HOME` (`/root/`), the container has the following structure (some directories irrelevant to our purposes are omitted):

```
.
|-- bench
|   |-- *.bc           # Bitcode we will analyse.
|   |-- bench.sh       # Benchmarking scripts -- run analyses
|   |-- table.awk      #   and produce tables.
|-- qbe                # Example code we will produce analysable bitcode from.
`-- svf                # SVF source and build tree.
    |-- knl-build      # Build for Knights Landing microarchitecture.
        |-- bin/wpa    # SVF binary (similarly for the following two builds).
    |-- haswell-build  # Build for Haswell microarchitecture.
    |-- base-build     # More portable AMD64/x86-64 build.
    `-- * (remainder)  # SVF source.
```

2 GETTING STARTED GUIDE

In this section, we first load the Docker image and run a container, and then perform a simple experiment to verify things work.

2.1 Environment Setup

First, we must load the Docker image and start a container to run the benchmarks in:

(1) `docker load < compacting-points-to-sets.tar.gz`
(2) `docker run -it compacting-points-to-sets bash`

Now, we are in a Debian container, ready to run experiments. For convenience, some tools like `vim`, `git`, and `nano` are available. All instructions should be performed within this container unless otherwise specified.

2.2 Simple Experiment

We will run our benchmarking script on the first program, `dhcpcd`:

(1) `cd $HOME/bench`
(2) `./bench.sh $HOME/svf/base-build/bin/wpa 1 1 1 dhcpcd.bc`

This will perform 1 round of all necessary analyses (5; see columns of Tables 6 and 7) on `dhcpcd` with a time-limit of 1 hour and a memory-limit of 1 gigabyte. This should take less than 10 minutes and will print Tables 4 to 7 from the paper with a single data row.

NOTE: `base-build` should be replaced with `knl-build` on machines which support running programs compiled for Knights Landing (`-march=knl`). On machines which do not support such but can run programs compiled for Haswell (`-march=haswell`), `base-build` should be replaced with `haswell-build`. In short, the preference is to run the Knights Landing build. If that is not possible, then the Haswell build. If that is not possible, then the generic base build. We use the Knights Landing build in our paper. **For the remainder of these instructions, we will use "`march-build`" to refer to whichever is most appropriate for the user's machine**, for example, `$HOME/svf/march-build/bin/wpa`.

2.3 Tracking Progress

The benchmarking script will print the currently running analysis, for example:

```
= running: /root/svf/haswell-build/bin/wpa ... -cluster-fs ...
-staged-pt-type=sbv dhcpcd.bc
```

We have omitted some options for readability. Most notably, this line means we are analysing `dhcpcd` with our clustering (`-cluster-fs`), and using sparse bit-vectors (`-staged-pt-type=sbv`). An absence of the `cluster-fs` option indicates the analysis is without our clustering, and the possible values for the `-staged-pt-type` option are `sbv` (sparse bit-vectors), `cbv` (core bit-vectors), and `bv` (bit-vectors).

3 STEP-BY-STEP INSTRUCTIONS

This section details usage of the benchmarking script and how to reproduce our results, how to build new bitcode for analysis, and how to modify the source code.

3.1 The Benchmarking Script

The benchmarking script `bench.sh` runs analyses and calls to a tabling script to produce tables from the results. It is to be used as follows:

```
usage: ./bench.sh SVF_BIN NUM_RUNS TIME_LIMIT MEM_LIMIT BITCODE...
     : TIME_LIMIT is in hours
     : MEM_LIMIT is in GB
```

where,

- `SVF_BIN` refers to the SVF binary and 3 are available at `$HOME/svf/march-build/bin/wpa` where `march-build` is replaced as described at the end of Section 2.2.
- `NUM_RUNS` refers to the number of rounds of analyses to be performed. The script will then take a median of the values produced by the runs. The best candidate (bold in Tables 4 and 5) is determined by choosing the most common best candidate. If required, the median values for the number of words (Tables 4 and 5) is rounded up. If in analysing a program the analysis runs out of time or memory once, we consider that to be the result for that program/analysis.
- `TIME_LIMIT` (hours) refers to the time-limit given to each run, excluding anything not part of the flow-sensitive analysis (reading bitcode, running auxiliary analysis, producing statistics, etc.).
- `MEM_LIMIT` (GB) refers to the memory-limit given to the entire SVF process.
- `BITCODE...` is a list of LLVM bitcode files to analyse.

Note: `bench.sh` must be run from the `$HOME/bench` directory, and bitcode files must live within `$HOME/bench`.

3.1.1 Bitcode Profiles. To make listing the desired bitcode files for the `BITCODE...` option simpler, we have added environment variables `W_8GB`, `W_16GB`, `W_32GB`, and `W_64GB`. `W_XGB` expands to a list of bitcode files in the paper which can be analysed with clustering and CBVs (6th column of Table 7) using `X` GB of memory. For example, `$W_8GB` expands to `dhcpcd.bc gawk.bc bash.bc mutt.bc keepassxc.bc`. We have also included variable `ALL_BITCODE` which is equivalent to `W_64GB`, i.e., includes all the programs.

3.2 Reproducing our Results

Knowing how `bench.sh` works, our results can be reproduced by

```
./bench.sh $HOME/svf/knl-build/bin/wpa 3 24 100 $ALL_BITCODE
```

That is, for all bitcode, we performed 3 rounds, capped time at 24 hours, memory at 100 GB, and used a Knights Landing build. We used an Intel Xeon 6132 processor.

For Tables 4 and 5, the values and the final column should be similar (not precisely the same as there exists some level of indeterminism that may affect internal data structures but not results). For Table 6, the final two columns should be similar to the paper, but may differ slightly due to CPU, build, load, some degree of indeterminism, etc. For Table 7, the final two columns should be extremely similar.

3.3 Building Bitcode for Analysis

3.3.1 Rebuilding our Bitcode. Our programs were built using the crux-bitcode project (https://github.com/mbarbar/crux-bitcode). Outside the Docker container, it can be used to produce bitcode which can be copied into the container with `docker cp`. To build our programs in the same way we did, outside the container (this may take a while as a Docker container is spun up to compile the programs and their dependencies):

(1) `git clone https://github.com/mbarbar/crux-bitcode`
(2) `cd crux-bitcode`
(3) `git checkout f581decd05`
(4) Edit `pkgmk.conf` to be:

```
export CFLAGS="-O3 -g -fno-discard-value-names -fno-lto"
export CXXFLAGS="-O3 -g -fno-discard-value-names -fno-lto"
export JOBS=$(nproc)
export MAKEFLAGS="-j $JOBS"
```

(5) Edit `pkgs.txt` to be:

```
dhcpcd
gawk
bash
mutt
lynx
xpdf
static-ruby
keepassxc
```

(6) `./build-bitcode.sh`

3.3.2 Building Bitcode Manually. We can use Whole Program LLVM (https://github.com/SRI-CSL/gllvm) to build bitcode by using `gclang` or `gclang++` as the compiler (this is what `crux-bitcode` attempts to automate). Note: it is not easy to substitute the compiler for all projects. For this example, we will build bitcode for `qbe`, a small compiler backend. The source is included in the container and does not need to be downloaded.

(1) `cd $HOME/qbe`
(2) `CC=gclang make` (build with `gclang`.)
(3) `cd obj`
(4) `get-bc qbe` (extract bitcode from built binary.)
(5) `cp qbe.bc $HOME/bench`

Now, following previous instructions, `qbe.bc` can be used with the `bench.sh` script.

3.4 Source Code

We have modified SVF at `$HOME/svf`. Our most important changes are available in the files (rooted at `$HOME/svf`):

- `include/Util/NodeIDAllocator.h` and `lib/Util/NodeIDAllocator.cpp`: methods in class `NodeIDAllocator::Clusterer` are called to perform clustering.
- `include/Util/CoreBitVector.h` and `lib/Util/CoreBitVector.cpp`: implementation of the core bit-vector.
- `include/Util/PointsTo.h` and `lib/Util/PointsTo.cpp`: makes it easy to switch the points-to set data structure on-the-fly (largely boilerplate).

After some clean up, we plan to include our work upstream. Progress on this can be seen at https://github.com/SVF-tools/SVF/wiki/Object-Clustering.

3.4.1 Modifying Source Code. Source code in `$HOME/svf` can be modified and rebuilt:

(1) `cd $HOME/svf`
(2) Modify source
(3) `cd $HOME/svf/march-build`
(4) `ninja`

Upon successful compilation, `$HOME/svf/march-build/bin/wpa` will reflect the modifications.

ACKNOWLEDGMENTS

For most of the duration of this project, the first author was supported by a PhD scholarship funded by CSIRO's Data61. This research is supported by Australian Research Grants DP200101328 and DP210101348.