A. Artifact description

A.1 Abstract

The present artifact supports the results presented in the paper "Trace-based Affine Reconstruction of Codes". We provide a Python implementation of the reconstruction engine described in the paper. The implementation requires Python 2.6 or higher and NumPy 1.8.1 or higher in order to run. The reviewer is expected to load from disk/generate a memory reference trace and run it through the `extract()` function, which will construct an affine loop that generates the input trace using the techniques introduced in the paper. In order to save reviewing time, we have included a "trace library" (`generate_trace()` function), allowing to synthetically generate a trace equivalent to any PolyBench one but in a much faster way than the time required to download, decompress and read the trace from disk. In order to assess the correction of the synthetically generated traces, the entire set of PolyBench C/3.2 traces, pre-processed in the manner described in the paper and ready to use as input for the engine, is made available online for the reviewer to double-check synthetic results at their discretion.

A.2 Description

A.2.1 Check-list (artifact meta information)

- **Algorithm:** we present a new algorithm to reconstruct affine references from their memory traces. Its pseudo-code can be found in the paper in Algorithms 1 and 2.
- **Program:** For validation, we use the publicly available PolyBench C/3.2 benchmarks. Since the traces generated by PolyBench need to be preprocessed as described in the paper before being fed to the reconstruction engine, we recommend to download the processed traces that we have made available. In order to save time, it is also possible to synthetically generate the traces using the provided facilities in the artifact code. The artifact also includes an annotated version of the PolyBench/C 3.2 codes, containing the correspondence between preprocessed/synthetically generated traces and references in the code.
- **Data set:** we use memory traces from PolyBench/C 3.2 which can be downloaded together with the artifact.
- **Run-time environment:** the provided code should be run with a Python 2.x interpreter, specifically $\geq$ 2.6.6. It also needs NumPy 1.8.1. iPython 2.x is required to inspect some examples interactively, although the same examples can be read using a web browser. No root access is needed to run the code.
- **Hardware:** the provided implementation is not optimized from the memory point of view, and needs the entire trace to be loaded into memory. A machine with 64GB of RAM is recommended to run the largest experiments.
- **Execution:** in order to achieve the performance obtained in the tests, the reviewer should not execute any other computationally expensive task while validating the tool.
- **Output:** the output is given through the interpreter in text mode.
- **Experiment workflow:** an experiment consists of: 1) launching the interpreter and loading the provided library; 2) loading or generating the target trace; 3) running the `extract()` function on the first-order differences of the trace.

A.2.2 How delivered

The present artifact is composed of a `.tgz` file and a set of compressed traces that can be optionally used as input to validate synthetic ones. Since these traces are very large, they are not packed together with the executable and documentation for the artifact. All the files and data can be downloaded from http://gac.udc.es/~grodriguez/CGO2016-AE/.

The main downloadable file, a `.tgz` archive, contains the following files:

- `trace_ae.py`: source code of the reconstruction engine. It has been annotated with meaningful information for the artifact evaluation process. It contains all the executable parts of the artifact.
- Folder `polybench-c-3.2`: contains annotated PolyBench source, to give insight about how the trace library included in the Python file corresponds to memory references in PolyBench codes. These can be used to generate evaluation traces at the reviewers' discretion.
- `workflow.ipynb`: iPython Notebook containing an example session using the tool in an interactive way.
- `workflow.html`: same iPython Notebook in HTML version.
- `cholesky.4.trace.bz2`: trace file used in the example notebook.

A set of pre-processed input traces, suitable for validating synthetically generated ones, can be obtained from the artifact home page. As detailed in the paper, these contain isolated memory traces generated by single references in the PolyBench codes. As their combined size exceeds 20GB, they have not been packed inside the downloadable `.tgz`.

A.2.3 Hardware dependencies

A machine with 64GB of RAM is recommended in order to process the largest trace examples. The performance numbers in the paper were obtained on Intel Xeon E5-2660 Sandy Bridge 2.20 Ghz nodes.

A.2.4 Software dependencies

The code requires a Python 2 interpreter to be executed, specifically Python $\geq$ 2.6.6. It also requires NumPy 1.8.1. In order to interactively inspect the iPython Notebook included in the artifact as an usage example, iPython 2.x must also be installed. Note that this usage example can be also accessed (although in a non-interactive way) using any web browser.

A.2.5 Datasets

The more convenient way to test the tool is through synthetically generated traces readily available by executing the `generate_trace()` function included in the Python code (see Section A.4). However, in order for the reviewer to double-check the accuracy of the synthetic traces, original traces generated by PolyBench C/3.2 executions are available for download at the artifact home page. These are very large and therefore not included in the packed artifact. Alternatively, the reviewer may decide to locally generate their own traces by running PolyBench codes. This is covered in more detail in Section A.4.

A.3 Installation

Installation of the tool is straightforward. It is implemented as a Python module, and therefore it is only necessary to extract the `.py` file and load it into an interpreter using `import`. Instructions are provided in Section A.4 to describe how to use the tool through an interactive Python session.

A.4 Experiment workflow

This section describes how to use the tool in four different use cases: manual reconstruction of synthetically generated traces, usage of automatic test features included in the tool to reconstruct synthetically generated traces, reconstruction of pre-processed traces downloaded from the artifact page, and local generation of custom traces by the reviewer. Note that this section is also available as an iPython Notebook in files `workflow.ipynb` and `workflow.html`, packed in the artifact.

A.4.1 Reconstruction of synthetically generated traces

Besides the reconstruction engine, `trace_ae.py` includes a trace library including all the PolyBench/C 3.2 memory references. The `generate_trace()` function can be used to synthetically generate a trace equivalent to any PolyBench one in the following way:

```
>>> import numpy as np
>>> import trace_ae
>>> (a,c,d)=trace_ae.generate_trace(
                              test="cholesky",
                              access=4,
                              size="standard" )
```

The previous command will generate a trace equivalent to `cholesky.4`, with only two differences:

- The base address of the access (first referenced address) is 0.
- The base stride (data size) is 1, instead of the size in bytes of the base array data type.

As mentioned in the last paragraph of Section 3.4 in the paper, these are not relevant for the reconstruction process. The `generate_trace()` function returns four values:

- `a`: NumPy array containing the synthetic trace.
- `c`: NumPy array containing the first observed stride, used to start the reconstruction process, and calculated as:

  ```
  >>> c = np.array( [a[1]-a[0]] )
  ```

- `d`: NumPy array containing the first order differences of trace `a`, calculated as:

  ```
  >>> d = a[2:] - a[1:-1]
  ```

Once the synthetic trace is generated, the reconstruction process is started by calling:

```
>>> trace_ae.extract( d, c, cut=3 )
```

In the previous call, the parameter `cut` indicates the depth threshold for the maximum solution size to try. Since the simplest representation of the `cholesky.4` reference is a 3-level loop, using `cut` below 3 will not return a valid reconstruction. This is a very fast process. Using `cut` above 3 will return a valid reconstruction, but it may not be a minimal one from the structural point of view (i.e., it may use loops of depth greater than 3). Besides, the number of explored branches grows exponentially with loop depth, and therefore it is not desirable to try a reconstruction with a depth greater than strictly necessary. For this reason, in order to guarantee a breadth-first traversal of the solution space, in our tests `extract()` is called with incremental values of `cut` until a solution is found (see Section A.4.2).

Upon completion, `extract()` returns the components of the solution, if it has been found:

- Coefficients vector $\vec{c}$.
- Iteration matrix $\mathbf{I}$, containing a subset of the iteration indices in the reconstruction.
- Bounds matrix $\mathbf{U}$.
- Bounds vector $\vec{w}$.

The Artifact Evaluation version also prints two values necessary to assess the experiments carried out in the paper:

- Number of references which were predicted by $\vec{\gamma}$ versus total number of references in the trace.
- Maximum memory usage for storing the solution elements and backtracking stack during the reconstruction process.

Note that this manual way of reconstructing a synthetic trace is not convenient for performance (runtime) measurements, which are covered in Section A.4.2.

A.4.2 Automatic reconstruction using system tests

In order to avoid the manual process of constructing a synthetic trace through `generate_trace()`, the Python code includes three convenience functions:

- To run the reconstruction on a single reference of a benchmark, run:

  ```
  >>> trace_ae.system_tests(
              test="cholesky",
              access=4,
  ```

```
                    size="standard",
                    timing=True,
                    debug=True )
```

This call will generate and reconstruct reference `choles-ky.4`. It will show the time for the reconstruction process and, after it ends, will check that running the generated loop rebuilds the exact memory trace used as input. Note that the reconstruction process will first try to reconstruct the trace using `cut=1`, then `cut=2` and finally `cut=3` before it suceeds. Reported time includes all three calls to `extract()`. Also note that the debugging process is very costly: reconstructing the original trace from **U**, $\overrightarrow{w}$ and $\overrightarrow{c}$ and checking that it is exactly equal to the original one takes longer than the reconstruction process itself (although, admittedly, this process is not as optimized as the reconstruction which is the subject of the paper).

- To run the reconstruction of all the references of a benchmark, run:

```
>>> trace_ae.all_accesses(
                    test="cholesky",
                    size="standard",
                    timing=True,
                    debug=True)
```

This call invokes `system_tests()` for each reference belonging to benchmark `cholesky` in the trace library . Reconstruction times are reported per reference.

- To run the full system tests, reconstructing all references of all PolyBench benchmarks, run:

```
>>> trace_ae.all_tests(
                    size="standard",
                    timing=True,
                    debug=True)
```

Note that this is a very lengthy process (it may take 24-48h, depending on the hardware used for the tests). Alternatively, the reviewer may use `size="small"` to generate PolyBench traces equivalent to those obtained when compiling PolyBench using `-DSMALL_DATASET`. This process will only take a few minutes, and is enough for assessing the reconstruction from the qualitative point of view.

A.4.3 Reconstruction of downloaded traces

The reconstruction process for downloaded traces is straightforward:

1. Download the chosen trace from the artifact home page.
2. Uncompress it using `bunzip2`.
3. Read the trace into memory using the following command:

```
>>> (a,c,d) = trace_ae.read_trace(
                        "cholesky.4.trace" )
```

Where `read_trace()` expects a full or relative path to the trace file. The function returns `a`, `c` and `d` with the same meanings as when generating the trace synthetically using `generate_trace()`.

4. Start the reconstruction process as before:

```
>>> trace_ae.extract(d, c, cut=3)
```

The solution must be structurally equal to the one obtained using synthetic traces. The only differences should reside in the base address, and in $\overrightarrow{c}$ which will be multiplied in this case by the data size of the base array.

A.4.4 Local generation of custom traces by the reviewer

The reviewer may easily generate a single reference trace from the PolyBench sources packed with the artifact:

1. Choose a benchmark and reference, e.g., `cholesky.4`.
2. Edit the source code of the `cholesky` benchmark, found in `polybench-c-3.2/linear-algebra/kernels/cholesky/cholesky.c`.
3. In the `scop` section of the code, look for the access annotated with the number `4`. In particular, `cholesky.4` appears in line 82:

```
x = x - A[j][k] * A[i][k];
// 4 -> A[j][k] (read)
```

4. Insert a `printf()` immediately after the reference in the code:

```
printf( "4 %lx\n", &A[j][k] );
```

5. Run the code and store the output of the `printf()` into a file `cholesky.4.trace`.
6. Use the reconstruction engine to process the generated trace.

A.5 Evaluation and expected result

There are four evaluation dimensions that the reviewer should take into account:

1. Qualitative evaluation: the reconstruction process should return **U**, $\overrightarrow{w}$ and $\overrightarrow{c}$ such that they reconstruct the exact affine input trace.
2. Performance evaluation: the reconstruction process should finish in a time that conforms to the findings in the paper.
3. Evaluation as a predictor: the reconstruction process should predict the number of accesses shown in Table 2 in the paper.
4. Memory consumption: Section 4 gives the lower and upper bounds on memory consumption. The code in `trace_ae.py` automatically measures this consumption for each invocation of `extract()`.