README.md

Guided Linking: Dynamic Linking Without the Costs

Getting started guide

System requirements

The minimum requirements are an x86-64 CPU core, 8-12GB RAM, and 30-50GB hard disk space. But with only one core, reproducing the results could take more than a week. We recommend using as many cores as possible, as long as there is at least 8GB RAM per core.

Running the VM

The artifact is provided as a virtual machine in OVA format, designed to work with VirtualBox. It has been tested to work with VirtualBox 6.1.10, but other versions are also likely to work. To run the VM, follow these steps:

1. Download the artifact file.
2. Download, install, and start VirtualBox.
3. Choose File > Import Appliance.
4. Select the artifact OVA file.
5. On the next page, change settings for CPU count and RAM size. Other settings can be left at their defaults.
6. Import the VM.
7. Start the VM.

Working in the VM

When the VM starts up, you should be logged in automatically. If not, you can log in with username `demo` and password `demo`.

The VM is based on the NixOS Linux distribution, which the reviewers may not be familiar with. In particular, installing software packages works somewhat differently in NixOS than in other distributions. After opening a terminal, packages can be found with `nix search` and installed with `nix-env -iA` as shown below. Note that these commands are **not** run as root. If a GUI program is installed, it will appear in the launch menu after a minute or two.

```
[demo@nixos:~]$ nix search emacs
* nixos.emacs (emacs)
  The extensible, customizable GNU text editor
...

[demo@nixos:~]$ nix-env -iA nixos.emacs
installing 'emacs-26.3'
...
created 2082 symlinks in user environment
```

If desired, files can be shared between the VM and the host computer. When the VM is stopped, open the VM settings, choose "Shared Folders", and add a folder of your choice with "auto-mount" enabled and `/mnt` as the mount point. After you start the VM, the shared folder will appear under `/mnt`.

Running an experiment

Open a terminal, and run:

```
[demo@nixos:~]$ cd ~/evaluation
[demo@nixos:~/evaluation]$ ./lzip-experiment.sh
...
/nix/store/wdv36yk7wa1jjycs603m63npl36rgwkg-lzip-links
lto
real    0m3.755s
user    0m3.678s
sys     0m0.048s

closed
real    0m3.676s
user    0m3.621s
sys     0m0.050s

compatible
real    0m3.754s
user    0m3.699s
sys     0m0.052s

interposable
real    0m3.623s
user    0m3.591s
sys     0m0.030s

open
real    0m3.336s
user    0m3.297s
sys     0m0.036s

open-spurious
real    0m3.554s
user    0m3.506s
sys     0m0.044s
```

If the final output looks like this, congratulations! You've successfully applied guided linking to lzip in several different configurations, compiled to machine code, and benchmarked the final executable. The exact numbers aren't important here.

Reproducing results (step by step instructions)

1. Understanding the artifact

The artifact consists of three folders under `/home/demo`:

- `nixpkgs-bitcode` is a version of the Nixpkgs package collection. It has been modified so it can build thousands of Linux packages with the Clang compiler and the `-fembed-bitcode` flag, which causes the compiler's intermediate representation (LLVM IR, aka "bitcode") to be saved in the final ELF executable/library. We're using `nixpkgs-bitcode` to get LLVM IR for all the packages we want to optimize with guided linking.
- `bcdb` is an implementation of a "Bitcode Database", which can store the bitcode from thousands of different packages in a single database. Guided linking is implemented as a tool that works with the database, called `bcdb mux2`. (Not to be confused with a different tool called `bcdb mux`.)
- `evaluation` contains the scripts used to evaluate guided linking.

Basically, the evaluation of a particular package set works like this:

1. `nixpkgs-bitcode` is used to build each package in the form of ELF files with embedded bitcode.
2. Tools like `bc-imitate`, `bcdb init`, and `bcdb add` (explained later) are used to extract the bitcode from all the packages and add it to a single BCDB file.
3. `bcdb mux2` is used to perform guided linking on the packages, producing a set of optimized bitcode files.
4. The optimized bitcode files are optimized and compiled to machine code using the standard LLVM tools.

2. What to expect from the evaluation scripts

Each of the experiment scripts, like `lzip-experiment.sh`, starts by running a `nix-build` command that invokes all four of the steps listed above. The VM image already has a cached copy of the `nixpkgs-bitcode` results, so step 1 will be skipped unless you change something, but the other three steps still need to be performed.

After finishing the steps, the experiment script then benchmarks the final machine code, comparing the guided linking version against a baseline LTO version. The results are saved in a file like `lzip-result.txt`. These files reproduce all the results in the paper, particularly figures 8 and 9 and table 2.

Along the way, lots of log messages will be printed. A few of these might look concerning, but are actually normal:

- The message `No bitcode found in ...` means that one of the packages depends on a library that was compiled without bitcode; this is intentional, as some packages aren't compatible with Clang or use assembly files, which aren't supported yet. These libraries won't be optimized using guided linking, but the packages that use them will still work.
- The message `WARNING: no isolated CPUs detected` mean that the VM isn't configured optimally for reproducible performance benchmarks. Unfortunately, due to time limitations, we could not provide instructions for using CPU isolation like we did in the paper.
- Some warnings will be printed about the standard deviations of results being too high. This is another side effect of the lack of CPU isolation.

3. Build everything at once (optional)

For optimal parallelism, it's possible to build all the optimized programs at once, and then run the experiments sequentially:

```
[demo@nixos:~/evaluation]$ ./all-experiments.sh
```

If you run this, you can skip the `*-experiment.sh` scripts below.

4. Evaluating Boost and Protobuf

```
[demo@nixos:~/evaluation]$ ./boost-experiment.sh
...
[demo@nixos:~/evaluation]$ cat boost-result.txt
LTO size:            402766912 bytes
Guided linking size: 175857088 bytes
[demo@nixos:~/evaluation]$ ./protobuf-experiment.sh
...
[demo@nixos:~/evaluation]$ cat protobuf-result.txt
LTO size:            91112400 bytes
Guided linking size: 62005240 bytes
```

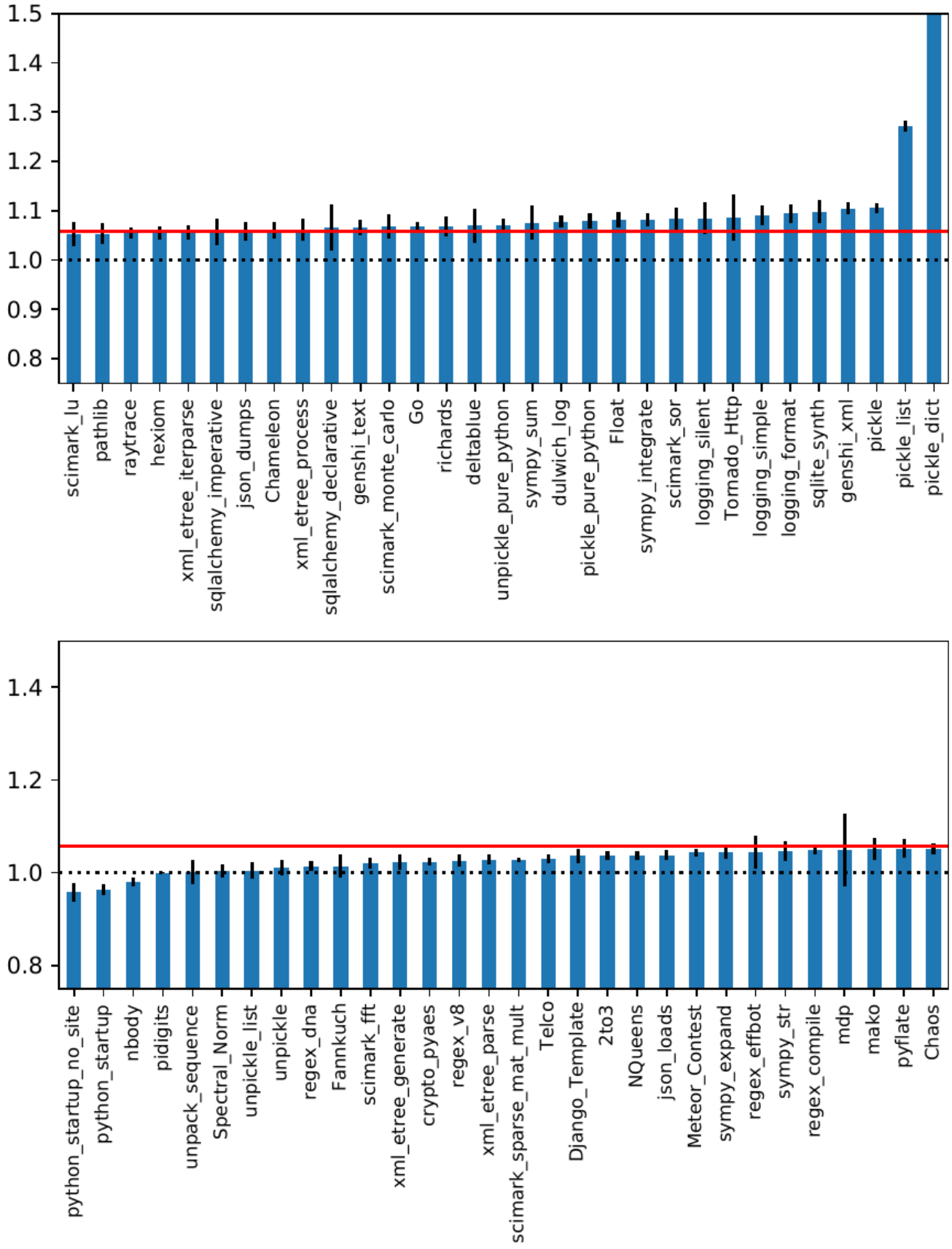
This compiles several versions of Boost and Protobuf and measures the total stripped ELF file size with and without guided linking, as shown in Table 2 on page 20 of the paper. Note that the numbers shown in the paper are mibibytes, not megabytes; the results will also be slightly different from the ones in the paper due to code refactoring.

5. Evaluating Python

```
[demo@nixos:~/evaluation]$ ./python-experiment.sh
...
[demo@nixos:~/evaluation]$ cat python-result.txt
[demo@nixos:~/evaluation]$ xdg-open python-result-figure0.png
[demo@nixos:~/evaluation]$ xdg-open python-result-figure1.png
```

This compiles Python with guided linking and evaluates it on the pyperformance benchmarks. Because of the differences between the VM and our benchmark machine, performance results may vary significantly.

NOTE: figure 9 in the paper is incorrect. We accidentally used a GCC-compiled version of Python as our baseline, when we should have used an LLVM-compiled version. When we re-ran the evaluation with the correct baseline, our average speedup over the baseline is slightly smaller (5.8% instead of 6.6%). We mentioned this problem to the paper reviewers, and we will correct it when revising the paper. This is the corrected version of figure 9:



6. Evaluating Clang

```
[demo@nixos:~/evaluation]$ ./clang-experiment.sh
...
[demo@nixos:~/evaluation]$ cat clang-result.txt
```

This compiles Clang with guided linking in several configurations, and benchmarks the time taken to run `clang -O1` to compile SQLite. This corresponds to figure 8 in the paper. Note that the file sizes given in the paper are mibibytes, not megabytes. The speed results may vary significantly from the paper due to the use of a VM, but the size results should be very similar.

Adding new experiments (optional)

New experiments can be created by adding them to the file `~/evaluation/experiments.nix`. They should be defined analogously to the existing experiments; it may be helpful to consult the [https://nixos.org/nix/manual/#ch-expression-language](Nix language reference).

Note that the `known-rtld-local` option, in the source, refers to applying the "no scope conflict" constraint by default.

When choosing the `configurations` and (if necessary) the `symbol-list`, reviewers should pay close attention to section 4 of the paper in order to determine which constraints can be safely applied. The current implementation of guided linking may silently produce incorrect output of the constraints are violated; we plan to address this in the future so that constraint violations are detected at run time and cause an error.

Not all packages provided by Nixpkgs actually build correctly with bitcode. A list of packages known to build successfully can be found in `~/nixpkgs-bitcode/pkgs/top-level/release-bitcode.nix`, although most of these packages have not actually been tested in bitcode form.

Once a new experiment has been defined, it can be compiled like so:

```
[demo@nixos:~/evaluation]$ nix-build -A foo.everything -o result-foo
[demo@nixos:~/evaluation]$ ls result-foo/
bcdb-file  lto  lto-packages  mux2  mux2-bitcode  mux2-packages  original
```

The most important parts of the result are `lto`, which holds the compiled baseline with LTO but without guided linking, and `mux2`, which holds the compiled guided linking result.

Using the tool manually (optional)

This section provides an example of using the BCDB without relying on Nix. Also take a look at `~/bcdb/README.md`.

1. Build with bitcode

```
nix-env -iA nixos.clang_10
git clone https://github.com/lz4/lz4
cd lz4
CC=clang make CFLAGS=-fembed-bitcode
```

2. Extract the bitcode

This extracts the bitcode that was saved with `-fembed-bitcode`, and also adds some extra information, like the list of libraries that `lz4` is dynamically linked to.

```
bc-imitate annotate --binary=lz4 > lz4.bc
```

3. Initialize the BCDB file

```
export BCDB_URI=sqlite:lz4.bcdb
bcdb init
bcdb add -name lz4 lz4.bc
```

Behind the scenes, the `lz4.bc` bitcode module is split into separate functions, and syntactically identical functions are deduplicated.

4. Explore the BCDB file

```
bcdb list-modules
bcdb list-function-ids
bcdb refs 99
bcdb get-function --id=99 | llvm-dis
```

5. Perform guided linking

```
bcdb mux2 --known-rtld-local --known-dynamic-uses lz4 --muxed-name muxed -o out
llvm-dis < out/lz4
llvm-dis < out/muxed
```

6. Compile to machine code

We use `bc-imitate` to figure out any extra linker options we need to add; for instance, if the original `lz4` file linked against a library `libfoo.so`, `bc-imitate clang-args out/lz4` will output `-lfoo`.

```
clang -flto out/muxed -o out/muxed.so $(bc-imitate clang-args out/muxed)
clang -flto out/lz4 out/muxed.so -o out/lz4.exe $(bc-imitate clang-args out/lz4)
out/lz4.exe --help
```

Exploring and changing the code (optional)

The main body of the guided linking code is in `~/bcdb/lib/BCDB/Mux2.cpp`. It's built on `~/bcdb/lib/BCDB/Merge.cpp`, which is used to merge modules together and handle symbol conflicts.