Appendix

A. Abstract

This artifact provides the source code that implements F3M, the contribution of this paper, and HyFM, the state-of-the-art function merging approach. Both optimizations are implemented on top of LLVM-14.0.0. Also included are scripts that install the necessary dependencies, run all the experiments, and reproduce the figures in this paper. We finally include pre-compiled bitcode files for 40 benchmarks. The artifact will build all of them under four strategies: no function merging, HyFM, F3M, and adaptive F3M. It will then generate plots comparing their code size reduction and compilation overhead. It will also run experiments measuring the effect of various F3M parameters such as fingerprint size, number of LSH rows, similarity threshold, bucket search cap. Finally it will produce two heatmap plots showing how well fingerprint similarity and merging quality correlate with each other under HyFM and F3M. The scripts can be easily extended to handle more benchmarks. The artifact has a minimal set of requirements that should be already met in most development systems running a modern version of Linux. We provide installation scripts to handle dependencies automatically.

B. Artifact check-list (meta-information)

- **Algorithm:** Fast Focused Function Merging algorithm
- **Program:** SPEC CPU 2006, SPEC CPU 2017, gcc-11.2.0, Google Chrome, LibreOffice 7.2, Linux kernel 5.11, clang-14. Non-SPEC benchmarks are included as precompiled bitcodes. SPEC benchmarks need to be obtained independently.
- **Compilation:** GCC-5.1 or above, Clang 3.5 or above.
- **Run-time environment:** Any relatively recent Linux system. Minimal dependencies should be already met. Otherwise sudo might be required. If not present, Python3 and cmake will be installed locally.
- **Hardware:** Any x86-64 machine with at least 32GB memory, preferably 64GB
- **Metrics:** Compilation time, Function Merging time, and object file size
- **Output:** A set of plots (.pdf). Processed data in CSVs
- **Experiments:** One bash script to run all experiments, another one to produce all plots
- **How much disk space required (approximately)?:** 15 GB (another 55GB for experiment 3)
- **How much time is needed to prepare workflow (approximately)?:** ˜30-60 minutes
- **How much time is needed to complete experiments (approximately)?:** Full experiments 2 weeks, main results 5 days, minimal set 1 day.
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache License 2.0
- **Archived (provide DOI)?:** 10.48420/17041502

C. Description

1) How delivered: The artifact is publicly available. It can be downloaded as a tar.xz archive from https://figshare.com/s/ae12c9bbac2d3157d0f6. After unpacking, the artifact occupies 3.5 GB of hard disk space. We provide bash scripts that automate the installation and use of this artifact.

2) Hardware dependencies: Any dedicated 64-bit x86 server running Linux should be acceptable as long it has at least 32GB of memory. Compiling the largest benchmarks though requires at least 64GB.

3) Software dependencies: A modern Linux installation is assumed. Most of the hard prerequisites below will be already installed on most development systems. If not, you will have to install them using the package manager, either manually or through `install.sh`.

Hard prerequisites:

- GCC-5.1+ or Clang-3.5+
- 32-bit libgcc (gcc-multilib)
- GNU Make 3.79+
- wget and git

Extra software is needed to build LLVM and run the experiments but the installation scripts will install them locally if needed.

Soft prerequisites:

- Python3 3.6+ and pip3
- CMake 3.13.4+
- numpy, matplotlib, and statsmodel

4) Data sets:

- cc1plus from GNU gcc-11.2.0
- Google Chrome (Cloned from repo on 2021-08-10)
- Libreoffice-7.2.0
- Linux-5.11 (using Sami Tolvanen's LTO branch)
- clang-14.0
- C/C++ benchmarks from SPEC 06/17

All of them are included as pre-compiled bitcodes. They were compiled for size (with the process described in their respective folders in this artifact) and their intermediate bitcode files combined into a monolithic bitcode using llvm-link.

D. Installation

Download our artifact (f3m-cgo22-artifact.v4.tar.xz) from the archive and then untar it:

```
$ tar -xf f3m-cgo22-artifact.v4.tar.xz
$ cd f3m_exp
```

Execute the `install.sh` bash script which automates installing all dependencies and building LLVM:

```
$ sudo bash ./install.sh
```

The script requires `sudo` rights for using the system's package manager. If the user wants to install dependencies manually (or if they are already installed), they can directly invoke `setup.sh` which prepares the experimental environment:

```
$ bash ./setup.sh
```

or for CentOS 7:

```
$ scl enable devtoolset-7 "bash ./setup.sh"
```

The user might want to change the default number of make jobs (`NJOBS` variable in `setup.sh`). Also, if the user has the Ninja build system already installed, they might prefer to replace the last two lines of the script with their commented versions using ninja. When the setup is complete, the folder llvm-project/build/bin should contain `opt`, `clang`, `ld.lld`, and `llvm-link`.

E. Experiment workflow

The artifact reproduces the results in Section IV, as well as Figures 3, 4, 6, and 9 in earlier Sections. The general workflow of each experiment is the following:

1) A top-level script selects a set of benchmarks, combinations of flags controlling the behavior of F3M and HyFM, and upper limits for the number of repeated evaluations and runtime.
2) The script then calls main() in run_exps.py which iterates over all benchmarks and flag combinations. In each iteration:
   a) If no pre-compiled bitcode file exists, we compile the benchmark's source files to bitcode and combine them with `llvm-link`.
   b) We use `opt` to apply the selected function merging technique with the provided compilation flags.
   c) We compile the optimized bitcode file to a binary object with clang and then link it with ld.lld.

d) Information about the time it took to run each stage and what F3M and HyFM did is saved to a log file.

e) The size of the object file, the location of the log file, and some other meta-information is stored in results.db.

f) If chosen, the scripts execute the benchmark binary and store its runtime in the database.

3) The plotting scripts read the database and the logs to get all the relevant data.

There are two different workflows that can be used, one that reproduces all the results but takes a very long time and one that reproduces the results only partially but can be run within a day.

1) Full Set: `$ bash run_full.sh`

This will take approximately two weeks. It is a good idea to run the experiments with an *appropriate virtual memory ulimit set*, to kill experiments early if they are going to fail due to the limited amount of memory.

The script will execute the following experiments:

Comparison of merging techniques: This is the main result presented in the paper. The experiment is implemented via `exp.1.main.py`. It compares F3M, adaptive F3M, HyFM, and no merging in terms of object file size and compilation time using all available benchmarks. It takes approximately five days for all benchmarks.

Exploration of LSH parameters: This experiment covers Section IV-D and is implemented via `exp.2.parameter_expl.py`. It uses all benchmarks except for the three largest ones (llvm, libreoffice, and chrome), compiling them for various different numbers of LSH rows, fingerprint sizes, and similarity thresholds. It also explores the effect of the bucket search cap but only on Linux. Like with the previous experiment, this takes a long time, approximately 10 days.

Correlation of fingerprints with alignment quality: This experiment produces Figures 4 and 10. It processes Linux without actually compiling it. It only outputs alignment ratios and fingerprint distances for all function pairs. The experiment is implemented in `exp.3.fingerprint_quality.py`. It takes around 20 hours.

2) Minimal Set: `$ bash run_minimal.sh`

This workflow only executes the main experiment (comparison of merging techniques), without the most time consuming benchmark (chrome), and without repeated measurement, so the results will suffer from noise. The experimental configuration is described by `exp.1.main.fast.py`. This should take only 16-24 hours.

F. Evaluation and expected result

1) Full Set: `$ bash plot_all.sh`

Alternatively, execute one by one the plotting scripts corresponding to each experiment (plot.*.py). This will generate a set of CSV files:

- data.1.main.csv with the main results
- data.2.geometry.csv for the exploration of fingerprint size and row number
- data.2.threshold.csv for the exploration of the similarity threshold
- data.2.cap.csv for the exploration of bucket size cap

The scripts will also generate a large number of plots. Some of them are used in the paper:

- Fig. 3: breakdown_{400.perlbench,linux,chrome}.pdf
- Fig. 4: correlation_linux_fq.pdf
- Fig. 6: merge_success_linux_hyfm.pdf
- Fig. 9: cost_benefit_linux_f3m.pdf
- Fig. 10: correlation_linux_mh.pdf
- Fig. 11: code-size-reduction.pdf
- Fig. 12: compile-time-increase.pdf
- Fig. 13: merging-breakdown.pdf
- Fig. 14: threshold_average.pdf
- Fig. 15: geometry_{ttime,size}_average_all.pdf
- Fig. 16: bucket_cap.pdf

These figures should be similar to the ones in the paper but not necessarily identical. Timing data might differ significantly, though the relative speedups/slowdowns between experiments performed on the same machine should be similar. Object file sizes for the baseline and HyFM should be nearly identical to those reported in the paper. Small differences can be attributed to different versions of libc, libstdc++, and libgcc. Object file sizes for F3M differ from execution to execution due to sources of randomness in the algorithm. This means that size results might differ from the ones reported. If each experiment is repeated enough times, the difference between the reported values and the ones generated should be smaller than the confidence interval for most cases.

2) Minimal Set: `$ bash plot_minimal.sh`

This will be useful if you have used the minimal workflow or you only executed `exp.1.main.py`. It will generate only one csv file (data.1.main.csv) and the plots for figures 3, 6, 9, 11, 12, and 13.

G. Experiment customization

Creating new experiments is relatively straightforward. Just define a list of compilation flag combination that you want to evaluate, a set of benchmarks, and call run_exps.main(). You can find a complete list of supported compilation flags and their meaning in flags.py (`flags._DEFAULT`).

To add a new benchmark, create a folder for it in `benchmarks`. The experimental scripts expect two things in the benchmark's folder. The first is a Makefile which references Makefile.config in `benchmarks/` and then defines benchmark specific compilation flags and rules. You can copy one of the existing Makefiles and adapt it. The other thing is either the source in a `src` subdirectory or a pre-compiled bitcode file.

You also need to define the benchmark in `config.BMARK_SUITES`. Each entry there defines either an individual benchmark, if the `pattern` list is empty, or a benchmark suite with the `pattern` list defining the naming patterns of the subdirectories corresponding to each benchmark. The `name` property needs to match the name of the benchmark's folder.

Finally, `config.py` defines the locations of the llvm binaries, the log files, the benchmarks, and the results database. Change them to suite your setup.