We develop a new technique to improve the compilers capability of optimizing innermost loops using the LLVM compiler infrastructure. In particular by the overcoming the limitations of static alias analyses by using profiling feedback to guide optimization. We combine a hybrid alias analysis scheme with loop cloning in order to expose instruction level parallelism and therefore allow the compiler to apply more agressive optimizations (eg. vectorization) when possible.

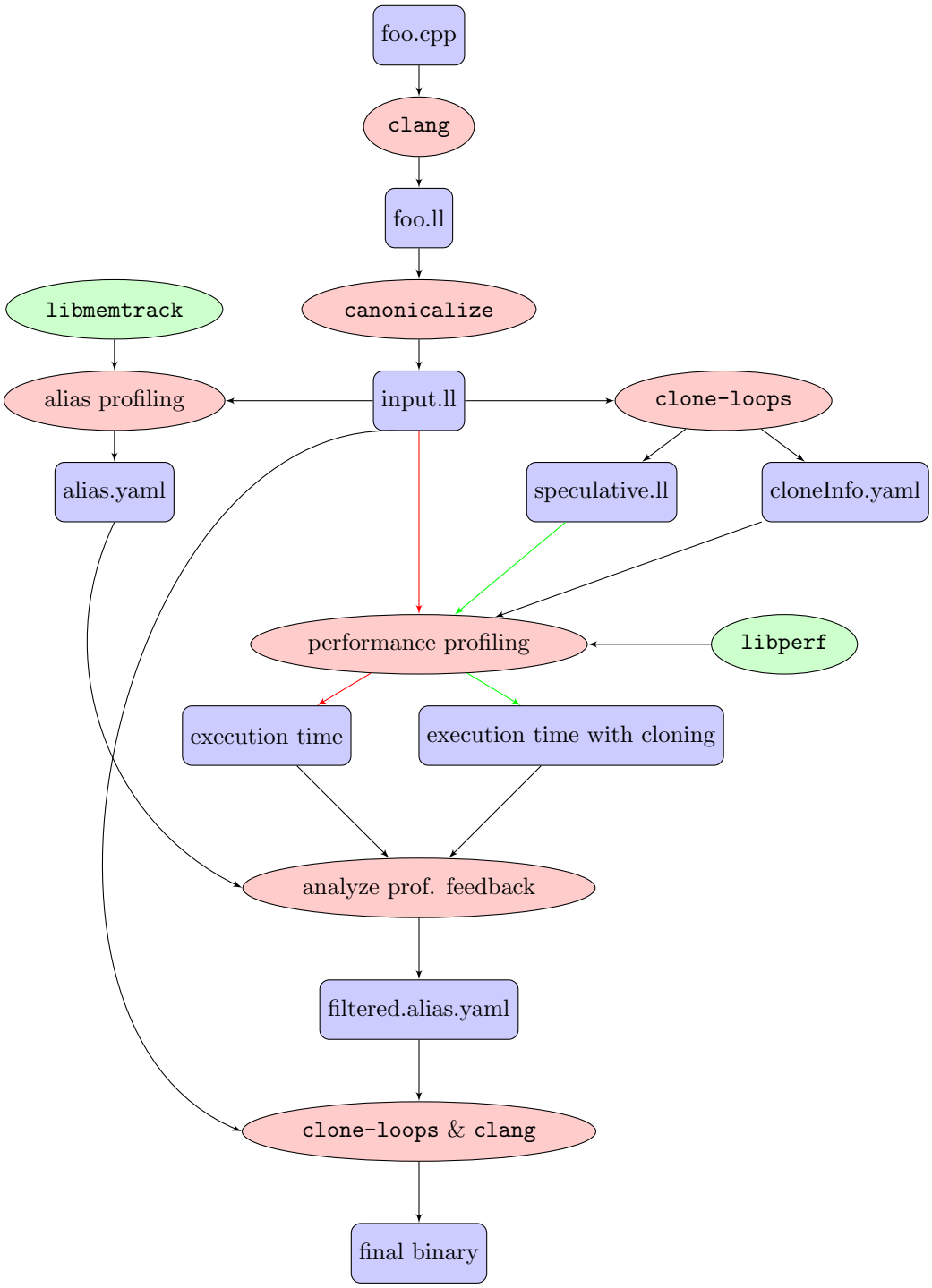
First we identify the innermost loops of the program, they are the candidates for cloning. For each innermost loop we instrument all the instructions (loads and stores) that may alias in order to achieve a more precise alias analysis. We insert runtime checks to see if two "may-alias" memory locations point inside the same memory region of the heap. If they do, we cannot tell whether they actually alias or not. If they point to different memory regions, then we know that they do not alias.

Each memory location can have more than one base pointers. We examine all the combinations of base pointers of the two interesting memory locations. The checks are hoisted as much as possible. In case of globals or function arguments a check is inserted in the first basic block of the function. Otherwise the check is inserted right after the instruction that defines the base pointer. The base pointers are quaranteed to dominate the loop preheader, thus no checks are inserted in the loop body. We do a bitwise AND of the checks and depending on the result the program either branches to the original loop body, or to a cloned version of it. The cloned loop contains "no-alias" annotations so that the compiler will be able to optimize it further.

We implement the module as a sequence of transformation passes. First we insert all possible checks and execute the program. During execution, the program creates a trace file containing the actual aliasing information regarding the interesting instructions that we want to profile. Then we compile the module again with this profiling information and we only keep the checks that resulted in "no-alias" most of the time. A third pass surrounds the cloned loops with timers and compares the execution time with the original program. If a loop cloning was not profitable then it is discarded. The timers take into account the runtime overhead of the checks. The "no-alias" frequency threshold and the minimum accepted speedup are configurable.

We assume that heap allocation is done using libc `malloc`. We overwrite `__malloc_initialize_hook`, `__malloc_hook`, `__memalign_hook` and `__free_hook` to keep metadata regarding each memory allocation. Upon initialization, we create a C++ `std::map` object using a custom memory allocator. The allocator uses `mmap` to avoid recursive calls to `malloc`. The map is sorted by the keys in descending order. Upon a `malloc` call we insert a mapping M such that M.key is a pointer to the start of the allocated region, and M.value is a pointer to the end of the allocated region. Tracking an arbitrary pointer is done by calling the `lower_bound` method on the map. This returns the first mapping with a key smaller than the given pointer. If the pointer is smaller than the value of the mapping, then it points inside this memory region. Upon a free call we erase the corresponding mapping.

Usage


foo.cpp
clang
foo.ll
libmemtrack
canonicalize
alias profiling
input.ll
clone-loops
alias.yaml
speculative.ll
cloneInfo.yaml
performance profiling
libperf
execution time
execution time with cloning
analyze prof. feedback
filtered.alias.yaml
clone-loops & clang
final binary

The toolchain consists of six tools and two dynamic libraries, it furthermore makes use of `clang` and `opt` from the LLVM project. The relationship of these tools is shown in the above figure.

- `canonicalize`

  - Input: A LLVM IR file (`.ll`).
  - Output: A LLVM IR file.

  This tool applies some transformations, such as loop simplification and memory to register promition. Any LLVM IR file that is passes to a later stage must first be processed by this tool. In order for the hybrid alias analysis to take full effect the input file should be compiled with either `-O0` or with `-fno-vectorize`, the final IR file can then be compiled with full optimizations enabled.

- `alias-instrument`

  - Input: A LLVM IR file.
  - Output: A LLVM IR file.

  This tool inserts calls to alias profiling code. Programs that have been processed by this tool must be linked with `libmemtrack`. Running the linked executable will create an alias trace file.

- `aggregate-alias-trace`

  - Input: An alias trace file and path for the output YAML file
  - Output: A YAML file containing the aggregated information from the alias trace

  This file aggragegates the raw alias trace into a YAML file. With the `--should-not-alias-threshold` option you can control at which probability two pointers are considered for runtime checks. The value should be a percentage as an integer between 1 and 100, and defaults to 5.

- `clone-loops`

  - Input: A LLVM IR file, the aggregated alias information YAML, and the desired path for cloned loops file prodcued by this tool.
  - Output: A LLVM IR file and a YAML file containing a list of cloned loops

  This tool uses the alias information gathered in a previous profiling run and uses it to perform loop cloning and insert speculative alias checks.

- `perf-instrument`

  - Input: Two LLVM IR files, first the file that was fed into `clone-loops` and then the output IR of `clone-loops`, and the cloned loops file.
  - Output: Two LLVM IR files with performance profiling instrumentation.

  This pass adds performance profiling instrumentation so the profitability of loop cloning can be evaluated. Programs processed by this tool have to be linked with the `libperf` library and running them creates a performance trace file.

- `aggregate-time-and-alias-trace`

  - Input: The alias YAML file, the YAML from the original programs performance profiling run, the YAML from the cloned programs perf profiling run and the desired path for the output file.
  - Output: A YAML file containing the aggregated information from the alias trace

  This file filters the alias YAML file with the information from the performance profiling run and removes loops for which cloning was not profitable. The resulting YAML can be used as input for `clone-loops` With the `--speedup-threshold` option you can control at which speedup cloning of a loop is considered profitable. The value should be a percentage as an integer, and defaults to 5.

- `libmemtrack`
  This library hooks into libc's `malloc` to facilitate our alias checks. It furthermore contains helper functions for alias profiling. There are two environment variables that control the behaviour of the tracing functions. `TRACE_FILE` controls the location of the trace file that is written, if not set it defaults to `alias.trace`. `SAMPLING_RATE` controls the sampling rate for tracing, it must contain a positive integer, higher numbers mean the profiling will be less accurate but also reduces the performance overhead of profiling.

- `libperf`
  This library contains helper functions for performance profiling. There is one environment variables that controls the behaviour of the tracing functions. `TRACE_FILE` controls the location of the trace file that is written, if not set it defaults to 'name of module' + `.cputime`.