"Header Time Optimization": Cross-Translation Unit Optimization via Annotated Headers

William S. Moses (wmoses@mit.edu), Johannes Doerfert (jdoerfert@anl.gov)
MIT CSAIL, Argonne National Lab

Writing Optimizable Code is Hard

How do we ensure that *norm* is hoisted outside the loop (and *normalize* vectorized)?
\[
\text{double norm} = \text{double}_{-\text{A}}(\text{int}\ n);
\]
\[
\text{void normalize} = \text{double}_{-\text{out}} + \text{double}_{-\text{in}}(\text{int}\ n) \\
\text{for}\ i = 0; i < n; ++i \\
\text{out}[i] = \text{in}[i] / \text{norm}(\text{in}, n);
\]

We could try adding: restrict type, const type, pure attribute, #pragma vectorize(enable), #pragma interleave(enable), #decispec((noalias)).

None of these work.

What we really want are two LLVM attributes:

- \text{attribute}({\text{fn_attr("readonly")}, \text{fn_attr("argmemonly")}})
- \text{double norm}(\text{double}_{-\text{A}}, \text{int}\ n);

void normalize = \text{double}_{-\text{restrict}} \text{out}, \text{double}_{-\text{restrict}} \text{in}, \text{int}\ n;

This is a problem in real programs! In the DOE RSbench benchmark [2] adding "read-\text{none}" to \text{fast\_exp} gives a 7% improvement to the entire program (with another 1% for "un\text{wind}").

Automatically Making Code Optimizable

LLVM automatically derives these attributes as part of the compilation process, then throws it away when it's done.

Let's ensure this information is accessible across translation units.

Why not always use LTO?

- Running LTO (even ThinLTO [3]) is a burden on compile times
- LTO may not be available in your build / operating system
- It's often impossible to run LTO on your entire program (e.g. using an external library)

Also, it's interesting to see how much of LTO's speedups come from "easily fixable" mechanisms and provide users the agency to fix them in source code (making the speedups available to everyone independent from compiler/linker used)

**Header Files**

HTO creates new files in a given directory that can be included in any C/C++ program (chosen for easiest experimentation).

Not all LLVM attributes are representable with existing Clang attributes. We created a generic way to represent LLVM attributes in Clang (shown below).

```c
struct Vector; struct Matrix;
\text{__attribute__((fn_attr("readonly")), arg_attr(0, "readonly"), ret_attr(_noalias))}
\text{Vector\_matevec(Vector\_M, Vector\_R)}
```

**Introducing "Header Time Optimization"**

At the end of the compilation process, denote what derived attributes can be safely added to functions using LLVM's existing analyses and Attributor [1].

Header time optimization has three modes of operation: remark mode (Figure 1), pipeline mode (Figure 2, 3), and diff mode (in progress) where we create a diff for original source tree.

**Demonstrating Header Time Optimization**

```c
\text{Figure 1. Remark Mode: print out optimization remarks for attributes that should be added to functions}

\text{Figure 2. Pipeline Mode: automatically generate a new header file with this new information, then use this header to recompile the source with this information. Often this doesn't even require an extra compilation (for example the HTO flag can be passed on a first build for profile guided optimization).

\text{Figure 3. Pipeline mode for a library. The annotated header is shipped with the library and used to compile user code.}
```

**Present Limitations & Future Work**

We currently don't generate annotations for functions with anonymous structs (we have a script to automatically generate random names), C++ member functions (since they can't forward declare), array type of structs/classes (type mystruct[3] is incompletely defined at this time).

When we allow users to output a diff (easier for integration) rather than pipeline (easier for experiments), these limitations are resolved and we get more performance gains.

In the future we plan to generate standard C/C++ attributes when they exist.

**Experiments**

Ran multi-source benchmarks in LLVM test suite Annotated headers allow more LLVM optimizations to perform better optimizations: 165% increase in mem2reg promotions, 33% increase in correlated value propagations, 28% increase common subexpression elimination, etc.

HTO was able to find significant speedups for many programs. Comparing with LTO we find that there are three places of interest: where neither found a speedup, where LTO found a speedup HTO didn't and where both HTO and LTO found a speedup.

**Acknowledgements & References**

William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DE-SC0019523, Google Summer of Code, NSF Grant 1533604, and 1533606, ANL grant 5177131, and IBM grant W1771446. Johannes Doerfert was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation’s exascale computing imperative.

