"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)?

```
double norm(double *A, int n);

void normalize(double *out, double *in, int n) {
  for (int i = 0; i < n; ++i)
    out[i] = in[i] / norm(in, n);
}
```

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

None of those work.

What we really want are two LLVM attributes:

```
__attribute__((fn_attr("readonly"), fn_attr("argmemonly")))
double norm(double *A, int n);

void normalize(double *restrict out, double *restrict in, int n);
```

This is a problem in real programs! In the DOE RSBench benchmark [2] adding "readnone" to `fast_cexp` gives a 7% improvement to the entire program (with another 1% for "unwind").

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 user's 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).

```
struct Vector; struct Matrix;

__attribute__((fn_attr("readonly"), arg_attr(0, "readonly"),
               ret_attr("noalias")))
Vector* matvec(Matrix *M, Vector *B);
```

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.

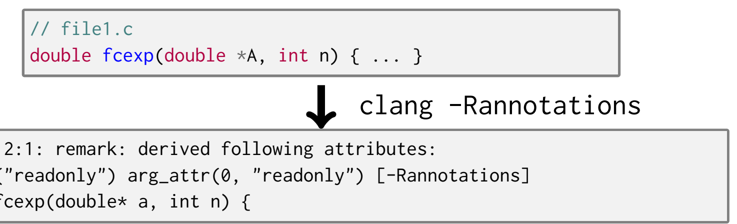

Figure 1. Remark Mode: print out optimization remarks for attributes that should be added to functions

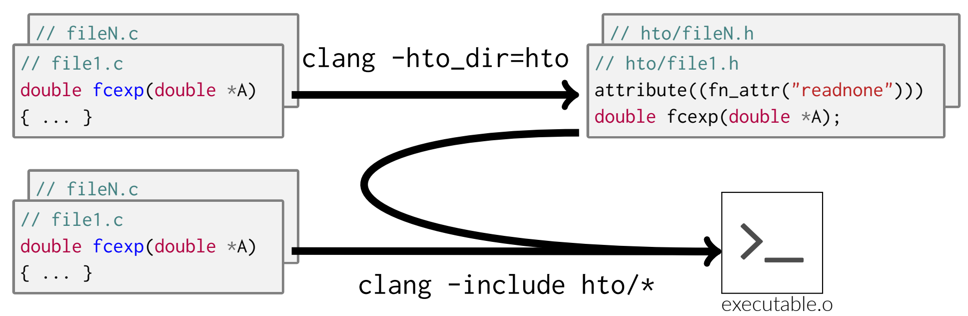

Figure 2. Pipeline Mode: automatically generate a new header file with this new information, then use this header to recompile the source with this information. Often this doesn't even require an extra compilation (for example the HTO flag can be passed on a first build for profile guided-optimization).

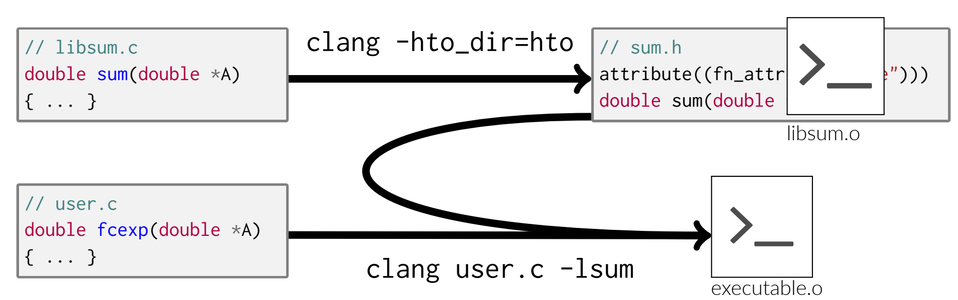

Figure 3. Pipeline mode for a library. The annotated header is shipped with the library and used to compile user code.

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 declared), array type of struct/classes (type mystruct[3] is incomplete ahead of 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 sigificnat 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.

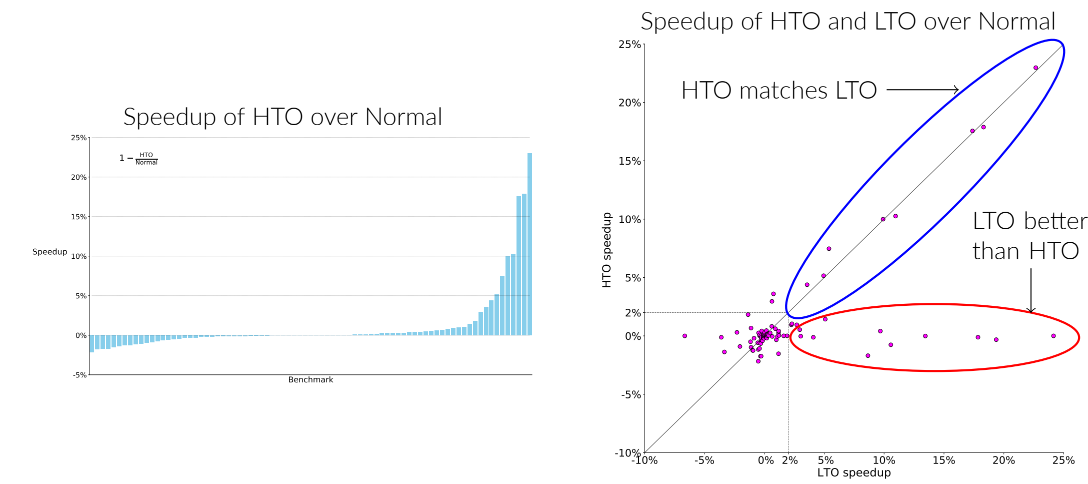

Figure 4. Speedups of HTO and LTO on the LLVM multisource test suite

Let's now look at the benchmarks where either LTO or HTO found a speedup. We see that for more than half of the LTO speedups can be simply derived by function annotations/HTO alone. For the other half of the speedups, LTO takes sigificantly longer to compile, implying that inlining/IPO is necessary.

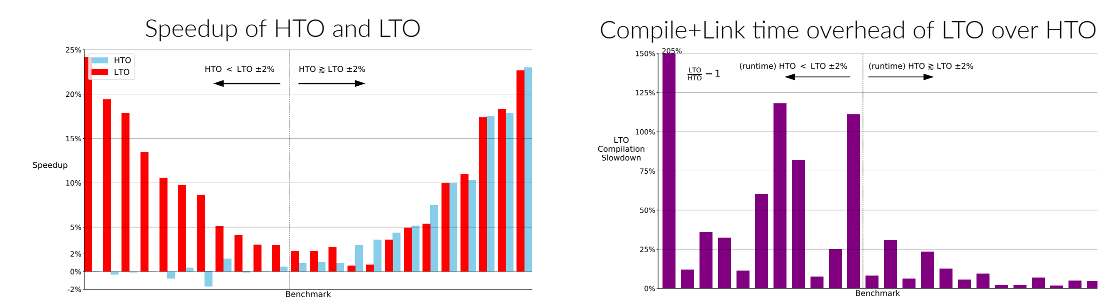

Figure 5. Comparison between LTO and HTO on codes where a speedup exists.

Acknowledgements & References

William S. Moses was supported in part by a DOE Computational Sciences Graduate Fellowship DE-SC0019323, Google Summer of Code, NSF Grant 1533644 and 1533644, LANL grant 531711, and IBM grant W1771646. 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.

[1] J. Doerfert, H. Ueno, and S.0 Stipanovic: The Attributor: A Versatile Inter-procedural Fixpoint Iteration Framework. US LLVM Dev Meeting, 2019.

[2] Johannes Doerfert, Brian Homerding, and Hal Finkel. Performance exploration through optimistic static program annotations. In *International Conference on High Performance Computing*, pages 247–268. Springer, 2019.

[3] Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: scalable and incremental lto. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 111–121. IEEE, 2017.