Improve nothrow detection in GCC

Abstract:

Exception handling in GCC follows the Itanium ABI, particularly the C++ exception handling ABI. This process involves creating, throwing, and finally destroying the exception object. The strategy employed is known as the Zero Cost strategy, which aims to minimize the impact on the main program execution path. It achieves this by pushing all exception handling operations into separate side tables, thus avoiding any potential negative effects on the instruction cache.

When an exception is caught, GCC utilizes DWARF information to unwind the stack, allowing the program to resume execution at the appropriate point. However, one current limitation is that GCC does not track the specific type of exception that is thrown.

This project aims to improve the nothrow detection in GCC which will allow to eliminate the dead exception handling regions and thus optimize the intermediate code generated.

Current State of Project:

The current implementation of exception handling in GCC works in a way that it marks statements that can possibly throw (these can be calls or non-call exceptions) and assigns them to exception handling regions. Exception handling regions are organized into a tree structure which describes what types are caught and handled (directly quoted by Jan Hubicka).
Two predicates, can_throw_internal and can_throw_external, are used in optimisation to identify nothrow functions. If a function is nothrow, EH tables can be saved, and EH handling code—particularly EH cleanup sections that call frequently occurring implicit destructors—can be optimized.

1) The predicate can_throw_internal takes a constant RTX instruction as input and returns a boolean value that is used to determine whether a function has the potential to throw exceptions to an internal scope. It does so by checking if the function has an associated landing pad.

2) The predicate can_throw_external checks if the instruction is a non-jump instruction, the function iterates over the elements of the sequence. For each element, it recursively calls

can_throw_external to check if it can potentially throw an external exception. If any element can throw externally, the function returns true. There are parameter checks and also checks for internal landing pad and if the instruction is within an EH region. The return value of this predicate is “false” if it is a nothrow function.

The nothrow discovery currently lives in pure-const pass and is very simple minded: if something in function passes can_throw_external then function can throw.

However, since exception types are not matched in the function, there are test cases where there is a redundant catch which is not optimized after the IR generation stage, one test case being given in the problem statement itself. The dumps for the exception handling (GIMPLE SSA) between all passes can be seen by adding the option -fdump-tree-all-details.

Project Goals:

The goal of this project is to make the middle end of GCC (GIMPLE) aware of the nothrow functions so as to optimize out the redundancies in the program code.
Currently, GCC doesn't have the ability to track the type of a given exception and detect that the given function handles all exceptions that it can possibly receive.

In exception handling, after constructing the exception object with the throw argument value, the generated code calls the __cxa_throw runtime library routine. We intend to use the arguments of the __cxa_throw function to help us know the types of exceptions that are being handled in the function.
As far as I have learnt by going through the codebase (eh_throw.cc), there are functions for throwing as well as rethrowing (used for providing helpful information to higher level catch blocks), which are structured as:

void __cxa_throw (void *thrown_exception, std::type_info *tinfo, void (*dest) (void *));

We will be using the second argument of this function in our solution for the project-

- A std::type_info pointer, giving the static type of the throw argument as a std::type_info pointer, used for matching potential catch sites to the thrown exception.

Implementation:

1) I plan to extend except.cc, possibly the can_throw_external function to take into account the types of exceptions getting handled in a function and say for sure whether the function is a nothrow.

2) Include propagation to determine the sorts of exceptions that are thrown again, enabling us to generate a list of all possible exception types that the function may throw.
3) Handle the GIMPLE SSA dumps for testing and debugging of the methods implemented.

NOTE: As suggested by Jan Hubicka (seems reasonable to me too) I will first
implement the propagation to make GIMPLE aware of the types of exceptions for the nothrow detection of a function.

Timeline Of the Project:

My college final exams will end on April 25 and I don't have any internship/projects during upcoming summer break, so I will be working full-time with GCC if selected. Thus, I will be able to devote 7-8 hours per day (40+ hours per week) to this project.
I have already built and tested GCC and have a basic understanding of except.cc, except.h and eh_throw.cc. Hence, it wouldn't require me to devote a lot of time in understanding the code. Although I have read the contribution guidelines and David's newbie guide, I will be spending my free time in April to get a good grasp on those aspects.

I plan to complete this project in four phases -

Here is the detailed timeline:

Phase 0 (Pre GSoC period till May 1) :

1) Get familiar with the exception handling of GCC, its data structures and the middle end of GCC (GIMPLE) and how it performs optimizations (SSA).
2) Read more about testing GCC, Contribution guidelines, making a patch etc.

Phase 1 :

1) Understanding the codebase in more detail.
2) Understand the semantics of the __cxa_throw call better and also look at the structure and working of exception handling tables.
This will require roughly 2-3 weeks of time.

Phase 2 :

1) Understand the current GCC EH optimizations and eh_frame structure in detail.

2) Extending the except.cc to do type sensitive propagation of thrown exceptions using type_info from __cxa_throw.
This will require roughly 3-4 weeks of time.

Phase 3 :

1) Finally test all implementations on a comprehensive GCC test suite.
2) Writing documentation as per need.
This will require roughly 1-2 weeks of time. (Testing won't require a lot of time but debugging will)
I have kept a buffer of approximately 2-3 weeks if things go south or maybe debugging a specific portion takes a lot of time.

About Me:

- Name - Pranil Dey
- University - Indian Institute of Technology Kharagpur
- Email - mkdeyp@gmail.com
- GitHub username: satansunny47
- Time Zone - IST (GMT + 05:30)
- Prefered Language for communication: English

I am a fourth-year undergraduate student pursuing a dual degree in the Department of Computer Science & Engineering at the Indian Institute of Technology Kharagpur. Since my first year in 2020, I have been programming in C/C++. I wish to work with GCC this summer as a GSoC student and beyond. I believe I have the necessary skills to undertake this project.
I've finished a number of C/C++ programming projects and assignments over the past three years, the most of which are accessible in my open github repository.
As part of the curriculum at my institute, I have also completed laboratory courses and compiler design theory.
In the laboratory, we designed a tiny-c compiler (a subset of GCC, GitHub link: click here). In theory, I learned about different phases of compilations, various optimization techniques, etc. Please click here to go to my course website for a detailed overview. My interest in compiler development was piqued by this course, and I can't wait to expand it by contributing significantly to GCC. I am also familiar with assembly language as I have taken courses on operating systems and computer architecture. This summer and in the future, I hope to contribute significantly to GCC.

My experience with GCC:

I've been using GCC for the past three years. I came here because I was curious to see how the GCC worked internally after taking a compiler course. I gained a solid understanding of how to build, test, and debug GCC by beginning with David's Beginner's Guide.
Also, I have added a custom diagnostic that says “hello world” and emits the function name for each function that is compiled in the source file.
Up until now, I have a good understanding of exception handling and I am excited to learn more about the various optimization techniques and implement the same. I have read about different contribution guidelines like coding styles, and how to make a patch and proper documentation techniques.
I have really enjoyed learning about the working of GCC EH, as GCC is something which I use frequently in programming and it brings me great joy to know that I have a chance to contribute to this impressive codebase.

Post GSOC:

As I have mentioned multiple times earlier, I am genuinely interested in compiler development, and I will always keep up with GCC's development and make my best effort to contribute. Furthermore, I will always be available for any future changes or extensions in this project.

References:
I have added links to wherever needed above, these are additional references.
https://gcc.gnu.org/pipermail/gcc/2024-March/243512.html
http://www.hexblog.com/wp-content/uploads/2012/06/Recon-2012-Skochinsky-Compiler-Internals.pdf
https://refspecs.linuxfoundation.org/abi-eh-1.22.html#cxx-abi
https://gcc-newbies-guide.readthedocs.io/en/latest/index.htm
Contributing to GCC - GNU Project