ЛАБОРАТОРНАЯ РАБОТА №2.
«ИЗУЧЕНИЕ ОПТИМИЗИРУЮЩЕГО КОМПИЛЯТОРА»

Цели работы

1. Изучение основных функций оптимизирующего компилятора, и некоторых примеров оптимизирующих преобразований и уровней оптимизации.
2. Получение базовых навыков работы с компилятором GCC.
3. Исследование влияния оптимизационных настроек компилятора GCC на время исполнения программы.

1. ОПТИМИЗИРУЮЩИЙ КОМПИЛЯТОР. ОСНОВНЫЕ ФУНКЦИИ И ХАРАКТЕРИСТИКИ

Программист при написании программ практически всегда пользуется языками программирования высокого уровня. В результате работы программиста создается исходный текст программы, который хранится в одном или нескольких текстовых файлах. Для того чтобы получившуюся программу можно было исполнить на компьютере, ее необходимо преобразовать из исходного представления на языке высокого уровня в бинарное представление, содержащее команды, которые может исполнить центральный процессор компьютера. Это преобразование производится с помощью программ, называемых компиляторами. Получая на входе один или несколько файлов с исходным текстом программы, компилятор создает бинарное представление и сохраняет его в бинарные исполняемые файлы, которые могут загружаться главной управляющей программой компьютера (операционной системой) и исполняться на компьютере.

Основные характеристики компилятора – входные языки высокого уровня, которые он может обрабатывать, и целевая архитектура (в первую

очередь – система команд процессора), для которой он может генерировать бинарный код.

Помимо главной функции – преобразования исходного текста программы в бинарный исполняемый код – компилятор обеспечивает проверку корректности синтаксиса программы, вывод сообщений о синтаксических и семантических ошибках пользователю, и оптимизацию кода. Существует много критериев оптимизации, но оптимизация кода в компиляторах, как правило, заключается в уменьшении размера бинарного кода и/или уменьшении времени исполнения программы.

Компилятор может быть как отдельной утилитой, запускаемой из командной строки, так и частью интегрированной среды разработчика, которая позволяет редактировать, компилировать и исполнять программы. В этой лабораторной работе рассматривается компилятор как отдельная утилита на примере компилятора GNU Compiler Collection (GCC). GCC позволяет компилировать программы, написанные на ряде языков высокого уровня, включая Си и Си++, в бинарный код для разных архитектур, в том числе 32- и 64-битных архитектур Intel (x86, x86_64). Управление настройками компилятора можно осуществлять с помощью ключей в командной строке, а также с помощью директив и атрибутов, указываемых в исходном тексте программы.

2. ОСНОВНЫЕ ГРУППЫ КЛЮЧЕЙ КОМПИЛЯТОРА GNU COMPILER COLLECTION

1. Глобальные настройки, например, задающие режим работы компилятора (например, делать ли препроцессинг, компиляцию, линковку), выбор имени генерируемого исполняемого файла и т.д.
2. Настройки, специфичные для Си/Си++, например, выбор диалекта языка программирования.

3. Настройки сообщений об ошибках и предупреждениях, например, нужно ли выводить предупреждения об объявленных, но неиспользуемых переменных.
4. Настройки отладки, например, нужно ли включать отладочную информацию в генерируемый исполняемый файл.
5. Настройки оптимизации (явное указание необходимых оптимизирующих преобразований и задание так называемых уровней оптимизации, рассматриваемых далее).

3. УРОВНИ ОПТИМИЗАЦИИ GCC

Число оптимизирующих преобразований в современном компиляторе велико. Явное задание всех необходимых оптимизирующих преобразований было бы громоздким. Поэтому вводится понятие уровня оптимизации как множества используемых оптимизирующих преобразований. Как правило, компиляторы имеют несколько уровней оптимизации. Например, в GCC есть следующие уровни.

1. На уровне **O0** почти все оптимизации отключены. Компиляция выполняется быстрее, чем на любом другом уровне оптимизации. При необходимости производить отладку программы или изучать ассемблерный листинг сгенерированного кода данный уровень оптимизации предпочтителен, так как получаемый листинг проще в понимании по сравнению с листингами для других уровней оптимизации.
2. На уровне **O1** включены оптимизации для уменьшения размера бинарного исполняемого файла и такие оптимизации, уменьшающие время работы программы, которые не сильно замедляют работу компилятора.
3. На уровне **O2** включены практически все доступные оптимизации, кроме тех, что ускоряют вычисления за счет увеличения размера кода.

4. В уровне **O3** включены все оптимизации из уровня **O2**, к ним добавлены оптимизации времени работы программы, которые могут приводить к увеличению размера бинарного исполняемого файла.
5. Уровень **Os** служит для оптимизации размера программ, в него включено подмножество оптимизаций из уровня **O2**.
6. Уровень **Ofast** включает все оптимизации уровня **O3**, а также ряд других, таких как использование более быстрых и менее точных математических функций.
7. Уровень **Og** производит все оптимизации, которые сохраняют возможность просмотра стека вызовов, фрагментов исходного текста программы, относящихся к разным уровням этого стека, и возможность приостановки программы для каждой строки исходного текста, содержащей операторы. Для многих программ оптимизация следующего уровня не дает выигрыша по скорости в сравнении с предыдущим. В ряде случаев использование уровня оптимизации **O3** приводит к генерации более медленной программы по сравнению с уровнем оптимизации **O2.** Общий подход к выбору уровня сводится к замерам времени исполнения программы, скомпилированной для каждого из этих уровней.

4. ПРИМЕРЫ ОПТИМИЗИРУЮЩИХ ПРЕОБРАЗОВАНИЙ В GCC

Все оптимизирующие преобразования можно разбить на две группы: платформенно независимые и специфичные для конкретной платформы. Если известна архитектура компьютера, на котором будет запускаться программа, то можно включить оптимизацию под эту конкретную архитектуру. Компилятор будет использовать дополнительные команды и другие возможности этой архитектуры, а также учитывать её особенности для получения более эффективного кода. В обычном режиме компилятор не может этого делать из соображений совместимости.

Список всех ключей оптимизации с аннотацией можно напечатать с помощью ключа ***-- help=optimizers***. Рассмотрим примеры некоторых оптимизирующих преобразований, применяемых в GCC.

Удаление мертвого кода (dead code elimination) – преобразование, удаляющее фрагменты кода, которые не влияют на результат программы. К мертвому коду относят код, который не исполняется ни при каких условиях, и код, изменяющий значения переменных, которые никогда не используются. Это преобразование уменьшает размер исполняемого кода и иногда уменьшает время исполнения, так как исключает выполнение команд, не влияющих на результат. Преобразование включается ключами ***-fdce, -fdse, -ftree-dce, -ftree-builtin-call-dce***, последнее из которых активно на уровнях оптимизации **O2**, **O3**, а остальные – на всех уровнях оптимизации кроме **O0**.

Отображение переменных на регистры процессора. При отсутствии оптимизаций компилятор отображает данные программы в оперативную память. В таком случае для каждого их чтения или записи происходит доступ к памяти. Если же данные имеют небольшой размер, то компилятор может отобразить их на регистры. Компилятор GCC отображает локальные переменные на регистры. Компилятор Compaq C Compiler для архитектуры Alpha может отображать на регистры небольшие массивы. Преобразование доступно на уровнях **O1**, **O2**, **O3**.

Раскрутка циклов включается ключами GCC ***-funroll-loops, -funroll-all-loops.*** Исходный цикл преобразуется в другой цикл, в котором одно тело цикла содержит несколько тел старого цикла. При этом счетчик цикла меняется соответственно. Эта оптимизация может уменьшать время исполнения за счет того, что уменьшается количество команд проверки условия выхода из цикла и команд условного перехода, которые могут приводить к приостановке конвейера команд. Однако, иногда раскрутка цикла приводит к увеличению времени исполнения программы. Другой недостаток преобразования – увеличение размера результирующего кода.

Встраивание функций. При использовании этого преобразования вместо вызова функции в код встраивается тело функции. При этом ценой разросшегося кода устраняются расходы на вызов функции и передачу аргументов. Встраивание функций, размер кода которых меньше или приблизительно равен размеру кода их вызова, включается ключом ***-finline-small-functions*** (включено по умолчанию на уровнях оптимизации **O2**, **O3**). Оптимизация по встраиванию более крупных функций включается с помощью ключей ***-finline-functions*** (включено на **O3**), ***-finline-functions-called-once*** (не включено только на **O0**), ***-findirect-inlining*** (включено на **O2**, **O3**). Кроме этих основных ключей есть дополнительные ключи для настройки параметров преобразования. Например, ***-finline-limit*** задает максимальный размер функций, которые следует встраивать.

Переупорядочивание команд. Команды, если это не нарушит информационных зависимостей, переупорядочиваются таким образом, чтобы более равномерно и полно загружать вычислительные устройства процессора.

Использование расширений процессора – группа специфичных для платформы преобразований. При генерации кода используются дополнительные команды, специфичные для данной архитектуры. В результате код может получиться более быстрым, особенно при векторизации вычислений, однако может потерять переносимость, т.е. не будет функционировать на процессорах других версиях архитектуры.

Вынос инвариантных вычислений за циклы. Если в цикле присутствуют вычисления, которые не зависят от итерации цикла, то они выносятся за цикл и тем самым многократно не повторяются.

Перепрыгивание переходов. Если в программе имеется цепочка последовательных переходов (условных или безусловных), она заменяется на единственный переход, который ведет сразу в окончательный пункт назначения, минуя промежуточные переходы. Преобразование включается ключом ***-fcrossjumping*** и активно на уровнях **O2**, **O3**.

Устранение несущественных проверок указателей на NULL. Считается, что обращение по нулевому указателю всегда приведёт к исключению (и аварийной остановке программы). Поэтому, если в коде встречается проверка указателя на ноль после обращения по этому адресу, то такая проверка из кода исключается, так как указатель заведомо не нулевой, если исполнение дойдёт до этой точки. Оптимизирующее преобразование включается ключом ***-fdelete-null-pointer-checks*** и выключается ***-fno-delete-null-pointer-checks***. Для платформ x86, x86_64 преобразование активно на всех уровнях, включая **O0**.

5. ЗАДАНИЕ ОПТИМИЗАЦИОННЫХ НАСТРОЕК GCC

Существует несколько способов указания требуемых оптимизирующих преобразований и уровней оптимизации.

- ***Ключи* компилятора.** Для компиляции программы с указанием уровня оптимизации (например, **O3**) используется команда:

  gcc -O3 lab2.c -o lab2.bin –Wall

  Регистр ключей в GCC имеет значение. Например, ключи –l и –L имеют разный смысл.

- **Директивы компилятора GCC** можно использовать для версии 4.4 и новее**.** Фрагмент исходного текста, для которого нужно задать определенный уровень оптимизации (например, **O3**) помечается директивами:

  #pragma GCC push_options

  #pragma GCC optimize("O3")

  // …текст программы будет оптимизироваться на уровне O3 …

  #pragma GCC pop_options

- **Атрибуты GCC** позволяют для отдельной функции можно указать настройки оптимизации с помощью атрибутов. Например, определим функцию, которую необходимо оптимизировать на уровне **O2**:

__attribute__((optimize("O2"))) f_O1(){…}

6. ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Написать программу на языке C или C++, которая реализует выбранный алгоритм из задания.
2. Проверить правильность работы программы на нескольких тестовых наборах входных данных.
3. Выбрать значение параметра N таким, чтобы время работы программы было порядка 30-60 секунд.
4. Программу скомпилировать компилятором GCC с уровнями оптимизации **-O0**, **-O1**, **-O2**, **-O3, -Os, -Ofast, -Og** под архитектуру процессора x86.
5. Для каждого из семи вариантов компиляции измерить время работы программы при нескольких значениях N**.**
6. Составить отчет по лабораторной работе. Отчет должен содержать следующее.
   1. Титульный лист.
   2. Цель лабораторной работы.
   3. Вариант задания.
   4. Графики зависимости времени выполнения программы с уровнями оптимизации **-O0**, **-O1**, **-O2**, **-O3, -Os, -Ofast, -Og** от параметра N.
   5. Полный компилируемый листинг реализованной программы и команды для ее компиляции.
   6. Вывод по результатам лабораторной работы.

7. ВАРИАНТЫ ЗАДАНИЙ

Варианты заданий взять из лабораторной работы №1.

8. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Каковы главные функции оптимизирующего компилятора?
2. Приведите примеры характеристик программы, по которым осуществляется оптимизация?
3. Какие бывают примеры оптимизирующих преобразований, что они оптимизируют, в чем их суть?
4. Всегда ли оптимизирующая компиляция позволяет уменьшить время работы программы?
5. Чем отличается общая оптимизация от оптимизации под архитектуру?
6. Какие имеются группы ключей в GCC?
7. Какие уровни оптимизации есть в GCC, и чем они характеризуются?