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).
6. **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.

1. **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, -funrollall-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 -finlinesmall-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-functionscalled-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-deletenull-pointer-checks. Для платформ x86, x86\_64 преобразование активно на всех уровнях, включая O0.

1. **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(){…}

8. КОНТРОЛЬНЫЕ ВОПРОСЫ

**1. Каковы главные функции оптимизирующего компилятора?**

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.

**2.Приведите примеры характеристик программы, по которым осуществляется оптимизация?**

• Скорость выполнения: Компилятор может оптимизировать код для более быстрого выполнения.

• Использование памяти: Компилятор может оптимизировать код для минимизации потребления памяти.

• Размер исполняемого файла: Компилятор может оптимизировать код для создания меньшего исполняемого файла.

• Энергопотребление: Компилятор может оптимизировать код для снижения энергопотребления.

**3. Какие бывают примеры оптимизирующих преобразований, что они оптимизируют, в чем их суть?**

4 пункт

**Суть оптимизирующих преобразований:**

Цель оптимизирующих преобразований — найти более эффективные способы выполнения той же задачи. Они могут изменять порядок выполнения инструкций, удалять ненужный код, изменять структуру данных, использовать более эффективные алгоритмы, и т.д.

**4. Всегда ли оптимизирующая компиляция позволяет уменьшить время работы программы?**

Нет, оптимизирующая компиляция не всегда позволяет уменьшить время работы программы.

• Специфика алгоритма: Если алгоритм программы уже оптимален, то компилятор не сможет существенно сократить время выполнения.

• Оптимизация для конкретных случаев: Некоторые оптимизации эффективны только для определенных типов задач или архитектур процессоров. Если компилятор оптимизирует код для случая, который не является доминирующим, то это может даже увеличить время работы.

• Накладные расходы на оптимизацию: Сама оптимизация кода требует времени и ресурсов. В некоторых случаях оптимизация может привести к увеличению размера исполняемого файла или времени компиляции, что может компенсировать прирост производительности.

• Сложные оптимизации: Некоторые виды оптимизаций могут быть настолько сложными, что компилятор не всегда может найти оптимальное решение.

**5. Чем отличается общая оптимизация от оптимизации под архитектуру?**

Оптимизация компилятора — это процесс преобразования исходного кода программы в более эффективный машинный код. Существуют два основных типа оптимизации: общая оптимизация и оптимизация под архитектуру.

Общая оптимизация направлена на повышение производительности программы независимо от архитектуры процессора. Она применяет преобразования, которые эффективны для большинства процессоров. Примеры общей оптимизации:

• Устранение мертвого кода: Удаление кода, который не влияет на результат программы.

• Развертывание циклов: Развертывание тела цикла, чтобы избавиться от проверок условий.

Оптимизация под архитектуру направлена на использование особенностей конкретной архитектуры процессора для увеличения производительности программы. Такие оптимизации могут быть неэффективны или даже вредны на других архитектурах. Примеры оптимизации под архитектуру:

• Использование инструкций SIMD: Использование специализированных инструкций SIMD (Single Instruction, Multiple Data) для выполнения операций над множеством данных одновременно.

• Векторизация: Преобразование циклов для использования инструкций SIMD.

• Оптимизация доступа к кэшу: Перестановка инструкций, чтобы улучшить использование кэша данных и инструкции, учитывая структуру кэша конкретного процессора.

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.

• Область применения: Общая оптимизация применяется для всех архитектур, оптимизация под архитектуру — только для определенной.

• Цель: Общая оптимизация направлена на повышение общей производительности, оптимизация под архитектуру — на максимальное использование особенностей конкретного процессора.

• Сложность: Оптимизация под архитектуру обычно более сложна, так как требует знания архитектуры процессора.

**6. Какие имеются группы ключей в GCC?**

Пункт 2

**7. Какие уровни оптимизации есть в GCC, и чем они характеризуются?**

3 пункт

-O0 (без оптимизации)

• Описание: Компилятор не выполняет никаких оптимизаций. Код генерируется в самом простом виде.

• Использование: Этот уровень оптимизации используется в основном для отладки программы, так как он создает код, который легко отследить и понять.

-O1 (базовая оптимизация)

• Описание: Включает базовые оптимизации, такие как константное распространение, устранение мертвого кода и простые преобразования циклов.

• Использование: Этот уровень оптимизации обеспечивает умеренное увеличение производительности с минимальным влиянием на время компиляции.

-O2 (более сложные оптимизации)

• Описание: Включает все оптимизации, доступные в -O1, а также более сложные преобразования, такие как инлайн-функции, более агрессивные оптимизации циклов и некоторые оптимизации доступа к кэшу.

• Использование: Этот уровень оптимизации предоставляет хороший баланс между производительностью и размером кода.

-O3 (максимальная оптимизация)

• Описание: Включает все оптимизации, доступные в -O2, а также более агрессивные преобразования, такие как векторизация, оптимизация для конкретной платформы и более сложные преобразования циклов.

• Использование: Этот уровень оптимизации используется для достижения максимальной производительности, но может привести к увеличению времени компиляции и размера кода.

-Os (оптимизация размера)

• Описание: Направлен на сокращение размера исполняемого файла за счет производительности.

• Использование: Этот уровень оптимизации используется для создания более компактных программ, например, для встраиваемых систем с ограниченными ресурсами.

-Ofast (максимальная оптимизация, игнорирование ограничений)

• Описание: Включает все доступные оптимизации, в том числе некоторые агрессивные преобразования, которые могут не соблюдать стандарты или вести к непредсказуемому поведению.

• Использование: Этот уровень оптимизации используется только в случаях, когда необходима максимальная производительность и правильность кода не является критичной.

Про Og:

включает отладочную информацию в создаваемый исполняемый файл. Эта информация позволяет отладчику (например, gdb) отображать:

• Стек вызовов: Отладочная информация показывает, какие функции вызывались друг за другом, позволяя вам проследить путь выполнения программы.

• Исходный текст: Отладочная информация связывает машинный код с исходным кодом программы. Таким образом, отладчик может отображать исходный код, соответствующий текущей строке выполнения.

• Точки останова: Флаг -g позволяет устанавливать точки останова на конкретных строках исходного кода, что позволяет приостановить программу в заданном месте и изучить ее состояние.

Важно:

• Выбор уровня оптимизации зависит от требований задачи и архитектуры целевой платформы.

• Не всегда более высокий уровень оптимизации приводит к лучшему результату. Иногда более простые оптимизации могут быть более эффективны.

• Рекомендуется проводить тестирование и сравнение результатов с разными уровнями оптимизации, чтобы выбрать оптимальный вариант.

**Режим работы компилятора (линковка и т.д):**

**1) препроцессинг - подставляет все библиотеки и инклуды с дефайнами, обработка макросов.**

**2)компиляция - перевод кода в низкоуравневый язык ассемблера**

**3)ассемблирование - перевод кода из ассемблера в машино-читаемый код, те бинарный**

**4) линкова собирает все разбросанные тобой файлы воедино, к примеру, хэдэры подставляются в один файл мэйн.**

**ассемблерный листинг**

**O1 – Os, Og работает примерно одинаково, понять их разницу**

**Доп вопросы:**

Почему раскрутка циклов ускоряет время работы (уже обсуждали количетсво проверок на выход из цикла)? Какую оптимизацию связанную с памятью мы не обсудили?

Как работает раскрутка циклов? Какая есть оптимизация связанная с goto?

Оптимизация цикла: в некоторых случаях, goto может использоваться для оптимизации цикла, например, для выхода из цикла в случае выполнения определенного условия.

