ЛАБОРАТОРНАЯ РАБОТА №1
«ОПРЕДЕЛЕНИЕ ВРЕМЕНИ РАБОТЫ ПРИКЛАДНЫХ ПРОГРАММ»

Цели работы

1. Изучение методики измерения времени работы подпрограммы.
2. Изучение приемов повышения точности измерения времени работы подпрограммы.
3. Изучение способов измерения времени работы подпрограммы.
4. Измерение времени работы подпрограммы в прикладной программе.

1. МЕТОДИКА ИЗМЕРЕНИЯ ВРЕМЕНИ ВЫПОЛНЕНИЯ ПРИКЛАДНОЙ ПРОГРАММЫ

Измерение времени выполнения прикладной программы или ее частей является одним из основных способов контроля характеристик аппаратного и программного обеспечения с точки зрения быстродействия. Такой контроль, с одной стороны, полезен для определения «узких мест» в алгоритме или программе, которые нуждаются в оптимизации. С другой стороны, позволяет судить о реальной производительности компьютера. Известно, что на время выполнения программы влияют разные факторы:

- характеристики самой программы,
- архитектура и конфигурация компьютера,
- операционная система,
- совместно работающие процессы,
- состояние компьютера на момент старта программы,
- влияние измерителя времени и т.п.

Отделить влияние интересующих характеристик от прочих далеко не всегда просто. Для этого существуют приемы, позволяющие снизить влияние

нежелательных факторов, а также интерпретировать полученные временные характеристики запусков программ.

Время выполнения программы или её частей (далее – программы), как правило, определяется разницей показаний таймера, которые снимаются перед началом исполнения программы и после её завершения.

Независимо от используемого таймера, измерение времени работы программы всегда выполняется с некоторой погрешностью (абсолютной или относительной). Погрешность измерения используемого таймера не должна выходить за рамки допустимой точности измерения. Величина допустимого значения погрешности определяет тип таймера, которым следует пользоваться при определении времени работы программы.

Абсолютная погрешность определяется разницей, например, в секундах между временем таймера и точным временем выполнения программы. Поскольку точное время выполнения программы никогда не известно, то абсолютная погрешность измерения оценивается точностью измерительного прибора. Например, если точность таймера 1 мс, то время было измерено с погрешностью не более 1 мс.

Относительная погрешность – это отношение абсолютной погрешности к величине временного интервала. Например, абсолютная погрешность в 1 мс даст относительную погрешность в 50%, если весь интервал был 2 мс и 0,1%, если интервал был 1 с. Таким образом, относительная погрешность показывает величину погрешности относительно всего интервала времени.

В современном компьютере имеется несколько таймеров с разной точностью. Для измерения времени работы программы можно использовать и внешние измерительные приборы, например, секундомер. Если точность выбранного таймера заведомо выше, чем требуется для заданного измерения, это ещё не гарантирует, что полученный результат измерения выполнен с требуемой точностью, т.к., возможно, измерение было искажено влиянием посторонних факторов. Для уменьшения этого возможного влияния

существует ряд приёмов. Ниже приведена процедура измерения времени выполнения программы.

Процедура измерения времени выполнения программы

1. Пусть задана величина допустимого значения погрешности и её тип (относительная и/или абсолютная точность). Если требуется удовлетворить только абсолютной точности измерения, то п.2. данной процедуры пропускается.

   2. Относительная точность преобразуется в абсолютную. Для этого необходимо любым способом оценить время выполнения программы. Далее выполняется п.3.
3. На компьютере, где будет выполняться измерение времени, оценивается степень загрузки процессора другими процессами. Если степень загрузки процессора высока, то выбирается таймер времени процесса, в противном случае выбирается произвольный таймер, например, из списка таймеров в п. «Способы получения показаний некоторых таймеров». Выбранный таймер должен обеспечивать требуемую точность измерения времени, иначе см. п.4.
4. Если доступные таймеры не могут обеспечить необходимую точность измерения из-за малой величины измеряемого интервала времени, увеличение точности измерения можно достигнуть следующим образом. Программа многократно запускается в цикле, и измеряется общее время выполнения этого цикла. Полученное значение времени выполнения цикла делится на число итераций в цикле. Абсолютная погрешность измерения уменьшится пропорционально числу итераций цикла.
5. Измеряется время выполнения программы с использованием приемов из пункта «Приемы уменьшения влияния посторонних факторов на время выполнения прикладной программы», затем проверяется качество измерения в п.6.

6. Если по условию измерения времени программы требовалось обеспечить абсолютную точность, то полученное время является результатом измерения, в противном случае полученное измерение времени считается новым оценочным временем выполнения программы, для которого ещё раз вычисляется требуемая абсолютная точность из заданной относительной. Если новая абсолютная точность выше, чем точность выбранного таймера, то выполняется возврат к п.3 с новым оценочным временем выполнения программы.

2. ПРИЕМЫ УМЕНЬШЕНИЯ ВЛИЯНИЯ ПОСТОРОННИХ ФАКТОРОВ НА ВРЕМЯ ВЫПОЛНЕНИЯ ПРИКЛАДНОЙ ПРОГРАММЫ

Многократное измерение. Время работы программы измеряется несколько раз. Измерения, как правило, будут отличаться. Это связано с тем, что сторонние факторы вносят различный вклад в измеряемый интервал при каждом запуске. Время же работы самой программы остаётся неизменным. Поэтому, из всех полученных измерений наиболее точным будет минимальное.

Исключение из измерения стадий инициализации и завершения. Если требуется измерить время работы некоторого фрагмента кода, например, некоторой процедуры, то целесообразно вынести весь код программы, предшествующий вызову процедуры, за первое измерение времени работы программы, а второе измерение осуществлять сразу после завершения её работы. Особенно это касается команд работы с устройствами ввода-вывода (чтение с клавиатуры, вывод на экран, работа с файлами) и, в меньшей степени, команд работы с оперативной памятью.

Уменьшение влияния кода измерения времени. Сам код снятия показаний того или иного таймера выполняется не мгновенно. Это означает, что в измеряемый интервал времени частично попадает и время его

исполнения. Это влияние следует, по возможности, уменьшать. Во-первых, не следует снимать показания времени часто, особенно в циклах. В идеале, код замера времени должен быть вызван лишь дважды – в начале и в конце работы измеряемого фрагмента кода. Во-вторых, необходимо следить за тем, чтобы измеряемый интервал был существенно больше, чем время работы функции замера времени.

Уменьшение влияния посторонних процессов. В случаях, когда процессор загружен другими процессами (например, системными процессами или задачами других пользователей), для получения более точного результата целесообразно использовать таймер времени выполнения процесса (см. примеры соответствующих таймеров в разделе 5.3). Такие таймеры, как правило, обладают более низкой точностью и плохо подходят для измерения небольших интервалов времени.

Сброс буфера отложенной записи на диск. В современных операционных системах, как правило, используется механизм кэширования при работе с внешней памятью, например, жесткими дисками. Этот механизм заключается в том, что при записи данных на диск данные сначала попадают в буфер отложенной записи, который располагается в оперативной памяти, а затем уже будут записаны позже. Этот механизм служит для ускорения доступа к файлам и уменьшению износа аппаратного обеспечения. В таких ОС возможна ситуация, когда во время работы вашей программы ОС решит сгрузить накопленные данные на диск, что наверняка повлияет на чистоту эксперимента. Поэтому перед проведением замеров времени следует освобождать буфер отложенной записи на диск. В ОС Linux/UNIX это делается с помощью утилиты `sync`. Эту команду можно набрать перед запуском своей программы из командной строки, либо вызвать её прямо из кода своей программы с помощью системного вызова `system` (см. system (3) man page).

Примечание: Тут и далее под man page подразумевается ссылка на стандартную линукс-документацию (Linux man pages). Чтобы посмотреть

документацию по той или иной команде можно в командной строке набрать следующую команду: `user@host:~ man 3 system`

Либо найти соответствующую страницу в Интернете.

3. ТАЙМЕРЫ

Вычислительная система имеет несколько программных и аппаратных таймеров, отражающих течение времени с различных точек зрения. Необходимо различать следующие таймеры:

- Таймер системного времени (system time, wall-clock time) – аппаратный счетчик, который отражает течение времени с точки зрения вычислительной системы и, как правило, соответствует реальному течению времени. Значение системного времени в каждый момент одинаково для всех программ, работающих на данном компьютере. Величина временного интервала, измеренного с помощью таймера системного времени, включает в себя время работы не только замеряющего процесса, но и других. Функции для получения системного времени:
  - Windows: GetSystemTime(), GetTickCount(), time(),
  - Linux: gettimeofday(), times(), time(), clock_gettime().

  Кроме того, в Windows используется функция clock(), которая позволяет получить величину системного времени, прошедшего с момента запуска данного процесса.
- Таймер времени процесса (process time, CPU time) – программный счетчик, который отражает использование процессорного времени только конкретным процессом. Шаг изменения этого счетчика относительно велик, поэтому его не следует использовать для измерения малых промежутков времени. Функции для получения времени процесса:
  - Windows: GetThreadTimes(),GetProcessTimes(),

- Linux: times(), clock().

- Счетчик тактов процессора (CPU time stamp counter) – аппаратный счетчик, значение которого увеличивается на каждом такте процессора. Такт процессора – самый малый интервал времени в вычислительной системе, который теоретически может быть замерен. Поэтому счетчик тактов позволяет с большой точностью измерять малые промежутки времени (вплоть до нескольких команд процессора). Счетчик тактов процессора имеет смысл использовать только для измерения интервалов времени меньших кванта времени, выделяемого процессу операционной системой. Для получения значения счетчика тактов используются специальные команды процессора, свои для каждой архитектуры:
  - x86/x86-64: rdtsc,
  - Alpha: rpcc,
  - Itanium: ar.itc,
  - PowerPC: mftb, mftbu.

4. СПОСОБЫ ПОЛУЧЕНИЯ ПОКАЗАНИЙ НЕКОТОРЫХ ТАЙМЕРОВ

4.1 Утилита `time`

Утилита `time` измеряет время работы приложения во многих конфигурациях ОС GNU Linux/UNIX (листинг 1).

Листинг 1

```
user@host:~$ time ./program.exe
real 0m0.005s
user 0m0.004s
sys  0m0.000s
```

Примечание. Тут и далее, строка ‘`user@host:~$`’ предваряет пользовательский ввод команд командной строки, она может отличаться на

разных системах. Вводить строку не нужно, она приводится в примерах для того, чтобы обозначить, что идёт ввод команды.

Утилита `time` выдаёт следующие временные характеристики работы программы:

real – общее время работы программы согласно системному таймеру,

user – время, которое работал пользовательский процесс (кроме времени работы других процессов) и

sys – время, затраченное на выполнение системных вызовов программы.

Дополнительная информация: time (1) man page.

Точность: определяется точностью системного таймера и точностью измерения времени работы процесса (см. описание соответствующих таймеров ниже).

Достоинство: готовая утилита, не требуется вносить изменения в программу.

Недостаток: измеряется только время работы всей программы, нет возможности измерить время работы отдельных её частей.

4.2 Библиотечная функция `clock_gettime`

Библиотечная функция `clock_gettime` получает значения системного таймера в ОС Linux/UNIX (листинг 2).

Листинг 2

```
01  #include <stdio.h>
02  #include <time.h> // for clock_gettime
03  int main(){
04    struct timespec start, end;
05    clock_gettime(CLOCK_MONOTONIC_RAW, &start);
06    // some work
07    clock_gettime(CLOCK_MONOTONIC_RAW, &end);
08    printf("Time taken: %lf sec.\n",
09      end.tv_sec-start.tv_sec
10      + 0.000000001*(end.tv_nsec-start.tv_nsec));
```

```
11      return 0;
12  }
```

Функция `clock_gettime` с параметром `CLOCK_MONOTONIC_RAW` сохраняет значение системного таймера в структуру `struct timespec`. Структура состоит из двух полей: `tv_sec` и `tv_nsec` (можно считать их тип `long int`), задающих количество секунд и наносекунд (10^{-9} сек.), прошедших с некоторого неспецифицированного момента времени в прошлом. В приведённом примере сохраняется значение таймера перед выполнением некоторого кода и после него. Разница показаний преобразуется в секунды и выводится на экран. Кроме системного таймера, функция позволяет получать значения и других таймеров, например, времени процесса или потока. Подробнее об этом можно прочитать в документации к этой функции. Реализация функции `clock_gettime` находится в библиотеке `rt`, поэтому при компиляции программы необходимо добавить ключ компиляции '`-lrt`'. Пример команды компиляции (в некоторых системах требуется ключ '`-lrt`' писать в конце команды):

```
user@host:~$ gcc prog.c –o prog -lrt
```

Дополнительная информация: clock_gettime (3) man page.

Точность: зависит от точности системного таймера. Обычно в ОС Windows: 55 мс ($55 \cdot 10^{-3}$ с), в ОС GNU Linux/UNIX: 1 нс ($1 \cdot 10^{-9}$ с).

Достоинство: переносимость – вне зависимости от аппаратного обеспечения функция доступна пользователю, т.к. реализуется ОС.

Недостатки: относительно низкая точность (обычно ниже, чем у счётчика тактов, но выше, чем у функции `times`), и измеренный интервал включает время работы других процессов, которые работали на процессоре в измеряемый период.

4.3 Библиотечная функция `times`

Функция измерения времени работы процесса `times` позволяет определить время работы данного процесса в многозадачной операционной системе, где каждый процессор (или ядро) выполняет несколько процессов (программ) в режиме разделения времени (рис.19). Этот показатель особенно важен, когда процессор сильно загружен другими процессами. В этом случае показания, например, системного таймера могут быть очень далеки от действительного времени работы программы. Процессор переключается между процессами с некоторой периодичностью. В обычных Linux/UNIX системах это около 10 мс (10^{-2} с). Функция `times` (листинг 3) позволяет определить, сколько таких квантов времени проработал наш процесс. Если перевести количество этих квантов во время, то можно определить, какое время работал процесс.

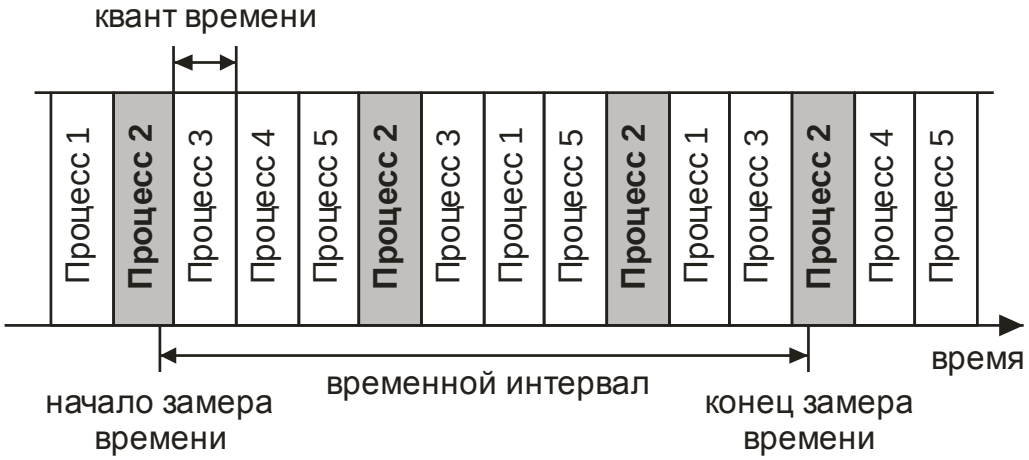

Рис. 19. Измерение времени в многозадачной операционной системе

Листинг 3

```
01  #include <stdio.h> // for printf
02  #include <sys/times.h> // for times
03  #include <unistd.h> // for sysconf
04  int main(){
05    struct tms start, end;
06    long clocks_per_sec = sysconf(_SC_CLK_TCK);
07    long clocks;
08
```

```
09    times(&start);
10    // some work
11    times(&end);
12
13    clocks = end.tms_utime - start.tms_utime;
14    printf("Time taken: %lf sec.\n",
15      (double)clocks / clocks_per_sec);
16    return 0;
17  }
```

В целом, её использование аналогично функции `clock_gettime`, отличие состоит в преобразовании единиц измерения из квантов времени в секунды. Количество квантов в секунду позволяет узнать системный вызов `sysconf`.

Дополнительная информация: times(2) man page, sysconf (3) man page.

Точность: зависит от кванта планировщика процессов, обычно 10 мс (10^{-2} с).

Достоинство: Из измеряемого времени исключается время, которое работали другие процессы, это обеспечивает более точное измерение времени работы программы на сильно загруженных процессорах по сравнению с другими способами.

Недостаток: относительно низкая точность, определяемая квантом времени переключения процессов.

4.4 Машинная команда `rdtsc`

Машинная команда `rdtsc` (Read Time Stamp Counter) снимает показания счётчика тактов в виде 64-разрядного беззнакового целого числа, равного количеству тактов, прошедших с момента запуска процессора (листинг 4). Время в секундах получается делением количества тактов на тактовую частоту процессора. В этом примере используется ассемблерная вставка команды процессора `rdtsc`, результат выполнения который записывается в объединение (union) `ticks` (старшая и младшая части). Разница показаний счётчика тактов преобразовывается в секунды в

зависимости от тактовой частоты. Узнать тактовую частоту процессора в ОС Linux/UNIX можно, например, распечатав содержимое системного файла `/proc/cpuinfo`.

Листинг 4

```
01  #include <stdio.h> // for printf
02  int main(){
03    union ticks{
04      unsigned long long t64;
05      struct s32 { long th, tl; } t32;
06    } start, end;
07    double cpu_Hz = 3000000000ULL; // for 3 GHz CPU
08
09    asm("rdtsc\n":"=a"(start.t32.th),"=d"(start.t32.tl));
10    // some work
11    asm("rdtsc\n":"=a"(end.t32.th),"=d"(end.t32.tl));
12
13    printf("Time taken: %lf sec.\n",
14      (end.t64-start.t64)/cpu_Hz);
15    return 0;
16  }
```

Точность: один такт (величина в секундах зависит от тактовой частоты процессора).

Достоинство: максимально возможная точность измерения времени.

Недостатки: привязка к архитектуре x86, в других архитектурах могут существовать аналогичные машинные команды. Затруднительно преобразование в секунды в процессорах с динамическим изменением частоты.

5. 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 15 секунд.

4. По приведенной методике определить время работы подпрограммы тестовой программы с относительной погрешностью не более 1%.
5. Составить отчет по лабораторной работе. Отчет должен содержать следующее:
    1. Титульный лист.
    2. Цель лабораторной работы.
    3. Вариант задания.
    4. Описание методики для определения времени работы программы.
    5. Результат измерения времени работы программы.
    6. Полный компилируемый листинг реализованной программы и команду для ее компиляции.
    7. Вывод по результатам лабораторной работы.

6. ВАРИАНТЫ ЗАДАНИЙ

1. Алгоритм вычисления числа Пи с помощью разложения в ряд (ряд Грегори-Лейбница) по формуле Лейбница N первых членов ряда:

$$\pi = 4\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1} = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \ldots + 4\frac{(-1)^n}{(2n+1)} + \ldots$$

2. Алгоритм вычисления числа Пи метом Монте-Карло. Алгоритм состоит в следующем. Сначала в квадрат с центром в начале координат и со стороной два вписывается круг с единичным радиусом. Затем в этом квадрате случайным образом с равномерным распределением генерируются N точек. Точка может попасть в окружность или нет (условие попадания $x^2 + y^2 \leq 1$). Далее определяется число M точек, попавших в круг. При достаточно большом числе бросков N, по значениям M и N вычисляется число Пи:

$$\pi \approx \frac{4M}{N}$$

3. Алгоритм вычисления определенного интеграла сложной функции методом трапеций:

$$\int_a^b f(x)dx = \frac{b-a}{N}\sum_{k=0}^{N-1}\frac{f(x_k)+f(x_{k+1})}{2}$$

где $f(x) = e^x \sin(x)$, a = 0, b = π, N - число интервалов.

4. Алгоритм вычисления функции sin x с помощью разложения в степенной ряд по первым N членам этого ряда:

$$\sin x = \sum_{n=1}^{\infty}\frac{(-1)^{n-1}x^{2n-1}}{(2n-1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + \frac{(-1)^{n-1}}{(2n-1)!}x^{2n-1} + \ldots$$

Область сходимости ряда: $-\infty \le x \le \infty$.

5. Алгоритм вычисления функции e^x с помощью разложения в ряд Маклорена по первым N членам этого ряда:

$$e^x = \sum_{n=0}^{\infty}\frac{x^n}{n!} = 1 + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots$$

Область сходимости ряда: $-\infty \le x \le \infty$.

6. Алгоритм вычисления функции ln(1+x) с помощью разложения в ряд по первым N членам этого ряда:

$$\ln(1+x) = \sum_{n=1}^{\infty}(-1)^{n+1}\frac{x^n}{n} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots + (-1)^{n+1}\frac{x^n}{n} + \ldots$$

Область сходимости ряда: $-1 < x \le 1$.

7. Алгоритм сортировки методом пузырька. Дан массив случайных чисел длины N. На первой итерации попарно упорядочиваются все соседние элементы; на второй – все элементы, кроме последнего элемента; на третьей – все элементы, кроме последнего элемента и предпоследнего элемента и т.п.

7. КОНТРОЛЬНЫЕ ВОПРОСЫ

- Назовите цели измерения времени.
- В чем состоит методика измерения времени работы программы?
- Назовите способы измерения времени работы программы. Перечислите их особенности.
- Каким способом лучше измерять большие промежутки времени (порядка нескольких часов)?
- Каким способом лучше измерять малые промежутки времени (порядка времени работы нескольких команд процессора)?

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?