1 слайд

Здравствуйте, уважаемая комиссия. Сегодня, я, хочу представить вам свою выпускную работу. Тема моей работы: Создать программу, автоматизирующую оптимизацию производительности участков поиска элементов данных в контейнерах std::list и std::vector с помощью индексированных контейнеров std::map в программах написанных на C++.

2 слайд

Цель выпускной работы: заключается в поиске оптимальной стратегии замен контейнеров «list» и «vector» контейнером «map» на участках непосредственного поиска элементов в контейнерах при условии, что требуемые для её реализации дополнительные ресурсы ОП не превысят верхней границы.

Задача работы: создание приложения, программно реализующего алгоритм оптимизации. Экспериментальная проверка эффективности.

Данный вид оптимизации даёт существенный прирост производительности на машинах с большим объёмом ОП.

В принципе, что такое оптимизация программы. Под оптимизацией программы подразумевают такие преобразования, в результате которых она становится более эффективной, т.е. становится более экономной по памяти или/и более быстрой по выполнению тех же функций, что и до оптимизации.

Оптимизация проводится по двум частным критериям: время выполнения программы и объём памяти, которую она использует. Но эти два критерия противоречат друг другу, т.к. чтобы уменьшить время работы, необходимо увеличить размер потребляемой памяти и наоборот.

Моя оптимизация основана на особенностях работы контейнеров vector, list и map, при поиске элементов.

3 слайд

Сформулируем оптимизационную задачу, минимизирующую суммарное время поиска в контейнерах std::vector и std::list с использованием вспомогательного индексированного контейнера std::map в виде модели

, где:

М –количество контейнер.

n*i* – количество элементов i-го контейнера данных;

v*i*– размер одного элемента i-го контейнера данных;

P*i* – среднее количество операций поиска в i-м контейнере данных;

V– верхняя граница дополнительного объема ОП, выделенной на оптимизацию;

z*i* – булева переменная равная единице, если для оптимизации;

4 слайд

b*i* – элемент массива булевых констант (входные данные). b*i* равна единице, если исходным контейнером хранения данных i-го массива являлся std::vector и нулю, в случае, когда исходным контейнером является std::list.

–вспомогательная функция, описывающая среднее время поиска элемента в i-м массиве с использованием вспомогательного контейнера std::map.

– скорость формирования контейнера std::map.

5 слайд

– средняя скорость сравнения двух произвольных элементов i-го массива.

–среднее время выборки и сравнения двух элементов i-го массива (равна );

–вспомогательная функция, описывающая среднее время поиска элемента в i-м массиве, реализованного в виде контейнера std::vector.

6 слайд

*–вспомогательная функция, описывающая среднее время поиска элемента в i-м массиве, реализованного в виде контейнера std::list.*

*–коэффициент пропорциональности > 1, описывающий характер превышения времени выборки данных в контейнере std::list по сравнению с std::vector . Более высокие временные затраты на перемещение между элементами списка объясняются тем что обращение к соседним элементам осуществляется посредством вспомогательных указателей.*

Следует отметить, что выигрыш в скорости поиска дается не только ценой дополнительной памяти, но также дополнительным процессорным временем, который тратится на создание индексированного массива std::map — соответствует второму слагаемому вспомогательной функции :

(2)

7 слайд

Пропорциональный характер времени формирования контейнера std::map относительно размера массива подтвержден экспериментально:

|  |  |
| --- | --- |
| Количество элементов типа double | Время формирования std::map  (миллисекунды) |
| 1000000 | 609 |
| 2000000 | 1250 |
| 3000000 | 1890 |
| 4000000 | 2563 |
| 5000000 | 3219 |
| 6000000 | 3890 |
| 7000000 | 4562 |
| 8000000 | 5234 |
| 9000000 | 5922 |
| 10000000 | 6610 |

Таблица 2.1. *Результаты тестовых замеров времени создания контейнера std::map*

Рис.2.1 Зависимость времени формирования контейнера std::map от размера массива.

8 слайд

Для того чтобы обеспечить возможность поиска решения задачи (1) при больших размерностях входных данных мной был выбран метод Монте-Карло, приведенный ниже (Алгоритм 2.1):

*Алгоритм 2.1.*

1. Ввод числа итераций N.
2. i=0.
3. Рекорду **R** присваиваем значение «бесконечность».
4. С помощью генератора случайных чисел генерируются булевы значения (0 либо 1) для каждого элемента **.**
5. Если условие не выполняется, то переход к шагу 9, иначе к следующему шагу.
6. Вычисляем значение целевой функции **,** с помощью выражения, указанного в модели (1).
7. Если F>=R, то переход к шагу 9, иначе к следующему шагу
8. R = F, сохраняем элементы массива {z} в массив {}.
9. i = i+1.
10. Если i<=N, то переход к шагу 4, в противном случае к следующему шагу.
11. Решение найдено. R содержит текущее значение целевой функции, а массив {} — значения элементов zi, соответствующие рекорду.

9-10 слайд

*Пусть имеется файл с исходным кодом ( на доске):*

Выполним поиск наилучшей стратегии оптимизации для данного кода пользователя:

На основании заданного текста программы выделим часть исходных данных(первый тык).

Примем в качестве верхней границы доступного для оптимизации объема ОП значение **7000** байт, а значения скоростей сравнения и создания контейнера 1.00E+13 байт/сек и 1.00E+10 байт/сек соответственно. Рассчитаем вспомогательные переменные ti по формуле .

11 слайд

Выполним расчет значений целевой функции для всех вариантов стратегий

Наилучшей стратегией является использование вспомогательных контейнеров std::map для 1-го и 3-го массивов, что обеспечит минимальное суммарное время поиска элементов в массивах пользовательского алгоритма, приведенного в примере.

12 слайд

1. Анализируем текст программы, выбранный пользователем;
2. Разбиваем по шаблону весь текст и находим необходимые блоки кода;
3. Определяем, возможно, неоптимальные участки кода, путём проверки: если в найденных блоках имеется поиск элементов в контейнерах, то, считаем, что это неоптимальный код, и мы будем его рассматривать.
4. Определяем тип рассматриваемых контейнеров;
5. Методом Монте-Карло определяем оптимальное решение, т.е. решение, где выигрыш будет максимальным, а затрачиваемые ресурсы удовлетворяют введённому ограничению;

13 слайд

Для реализации алгоритма, описанного выше, было разработано приложение на языке C#, платформы .NET 3.5 Windows Forms. Интерфейс программы представляет собой окно, содержащее элементы управления (кнопки) для загрузки пользователем файла с исходным кодом C++(рисунок на доске), а так же его анализом. Помимо них имеются иные элементы для вывода полученных данных анализа кода, а так же результат работы алгоритма.

Для загрузки файла с исходным кодом программы, написанной на C++, нажимаем на кнопку «Выберите файл». Откроется диалоговое окно, в котором нужно будет выбрать файл

14 слайд

Когда пользователь выберет файл, то его содержимое, будет выведено на экран

После выбора файла, пользователь должен нажать на кнопку «Начать анализ участка кода», т.е. программа должна проанализировать код и вывести соответствующие результаты на экран

15 слайд

Вывод результата анализа пользовательского текста программы

После анализа кода, пользователь должен ввести некоторые данные в специальном окне «Данные для подсчёта выигрыша», такие, как: (по умолчанию 10000000000), (по умолчанию 10000000000000), объём доступной ОП(V), число итераций для метода Монте-Карло (N = 10000 по умолчанию) и по желанию пользователя можно сделать активным флаг для вывода подробного решения. Так же в таблице необходимо заполнить столбец «Количество элементов» у каждого контейнера

Когда все данные введены, можно нажимать на кнопку «Автоматический расчёт».

16 слайд

В результате методом Монте-Карло будет подобрана оптимальная стратегия с учётом целевой функции и ограничения

Так же пользователь сам может выбирать элементы, вопреки тому, что вывела программа, но колонка «Рекомендации» остаётся неизменной, в случае если пользователь захочет использовать стратегию, предложенную программой.

17 слайд (Эксперименты)

Для демонстрации работоспособности разработанной программы, проведём эксперименты.

1 Эксперимент. Эффективность оптимизации

Эксперимент заключался в оценке степени прироста производительности, получаемом в результате замены контейнеров «vector» и «list» контейнером «map», на участках поиска элементов в контейнерах.

Описание: в ходе эксперимента, размер контейнеров «vector», «list» и «map» синхронно менялся от 100 000 элементов до 1 000 000 с шагом 100 000, а среднее число операций поиска в этих контейнерах (P) = 1000. Исходный код тестового примера:

КОД СТОПЭ

18 слайд (Эксперименты)

КОД ВТОРАЯ ЧАСТЬ

19 слайд (Эксперименты)

Эксперимент показал, что замена контейнеров «vector» и «list» контейнером «map», в случае осуществления поиска в контейнере даёт существенный прирост производительности.

20 слайд (Эксперименты)

2 Эксперимент. Зависимость времени анализа программы, написанной на C++, от размера кода

Эксперимент заключался в увеличение объёма текста программы и оценке производительности анализа кода.

ГРАФИК

Из рисунка видно, что при увеличении числа рассматриваемых блоков время анализа программы несущественно, но возрастает. Связано это с необходимостью перебора каждого блока для определения не оптимального кода.

21 слайд (Эксперименты)

3 Эксперимент. Зависимость времени нахождения решения от количества блоков неоптимизированного кода

При проведении эксперимента я брал программу, написанную на C++, где был обнаружен только один контейнер, возможно, подлежащий замене, и в дальнейшем увеличивал их количество от 1 до 10. Во всех программах , , число итераций N = 10000 и верхняя граница используемой оперативной памяти составляла V = 7000 байт.

РИСУНОК

Эксперимент показал, что при увеличении числа контейнеров, которые необходимо рассмотреть программе, растёт и время нахождения решения.

22 слайд

В результате исследования и выполнения работы мной были рассмотрены различные методы оптимизации. В том числе и оптимизация программы, написанной на языке C++, где анализировался код приложения и, предлагалась замена участков кода программы.

Для осуществления данной задачи было разработано приложение на языке C#. Также, были проведены исследования и на основании их построены графики, которые показывают эффективность данной работы и оптимизации в целом.

*Доверительный интервал* для среднего представляет интервал значений вокруг оценки, где с данным уровнем доверия (см. [*Элементарные понятия статистики*](http://statsoft.ru/home/textbook/esc.html)), находится "истинное" (неизвестное) среднее популяции. Например, если среднее выборки равно 23, а нижняя и верхняя границы доверительного интервала с уровнем *p*=.95 равны 19 и 27 соответственно, то можно заключить, что с вероятностью 95% интервал с границами 19 и 27 накрывает среднее популяции. Если вы установите больший уровень доверия, то интервал станет шире, поэтому возрастает вероятность, с которой он "накрывает" неизвестное среднее популяции, и наоборот. Хорошо известно, например, что чем "неопределенней" прогноз погоды (т.е. шире доверительный интервал), тем вероятнее он будет верным. Заметим, что ширина доверительного интервала зависит от объема или размера выборки, а также от разброса (изменчивости) данных. Увеличение размера выборки делает оценку среднего более надежной. Увеличение разброса наблюдаемых значений уменьшает надежность оценки (см. также [*Элементарные понятия статистики*](http://statsoft.ru/home/textbook/esc.html)). Вычисление доверительных интервалов основывается на предположении нормальности наблюдаемых величин. Если это предположение не выполнено, то оценка может оказаться плохой, особенно для малых выборок. При увеличении объема выборки, скажем, до 100 или более, качество оценки улучшается и без предположения нормальности выборки.