**Лекция 12. Стратегии и методы оптимизации программного кода**

**Оптимизация** - модификация системы для улучшения ее эффективности. Система может быть одиночной компьютерной программой, наборомкомпьютерив или даже целой сетью, такой как Интернет.

Хотя целью оптимизации является получение оптимальной системы, истинно оптимальная система в процессе оптимизации достигается далеко не всегда. Оптимизированная система обычно является оптимальной только для одной задачи или группы пользователей: где-то может быть важнее уменьшение времени, необходимого программе для выполнения работы, даже ценой потребления большего объема памяти; в приложениях, где важнее память, могут выбираться более медленные алгоритмы с меньшими запросами к памяти.

Более того, часто не существует универсального решения, которое работает хорошо во всех случаях, поэтому инженеры используют компромиссные (англ. Tradeoff) Решение для оптимизации только ключевых параметров. К тому же, усилия, необходимые для достижения полностью оптимальной программы, которую невозможно дальше улучшить, практически всегда превышают выгоду, которая может быть от этого получена, поэтому, как правило, процесс оптимизации завершается до того, как достигается полная оптимальность. К счастью, в большинстве случаев даже при этом достигаются заметные улучшения.

Оптимизация должна проводиться с осторожностью. Тони Хоар впервые произнес, а Дональд Кнут впоследствии часто повторял известное высказывание: "Преждевременная оптимизация - это корень всех бед". Очень важно иметь для начала озвученный алгоритм и работающий прототип.

**Основы оптимизации кода**

Некоторые задачи часто могут быть выполнены более эффективно. Например, программа на языке Си, которая суммирует все целые числа от 1 до N:

int i, sum = 0;

for (i = 1; i <= N; i + +)

sum + = i;

Имея в виду, что здесь нет переполнения, этот код может быть переписан в следующем виде с помощью соответствующей математической формулы:

int sum = (N \* (N + 1)) / 2;

Понятие «оптимизация» обычно подразумевает, что система сохраняет ту же функциональность. Однако, значительное улучшение производительности часто может быть достигнуто и с помощью удаления избыточной функциональности. Например, если допустить, что программе не нужно поддерживать более 100 элементов при вводе, то возможно использовать статическую выделения памяти вместо более медленного динамического.

**Компромиссы (tradeoff)**

Оптимизация в основном фокусируется на одиночном или повторном времени выполнения, использование памяти, дискового пространства, пропускной способности или некотором другом ресурсе. Это обычно требует компромиссов - один параметр оптимизируется за счет других. Например, увеличение размера программного кэша чего улучшает производительность времени выполнения, но также увеличивает потребление памяти. Другие распространенные компромиссы включают прозрачность кода и его выразительность, почти всегда цене деоптимизации. Сложные специализированные алгоритмы требуют больше усилий поналагодженни и увеличивают вероятность ошибок.

**различные области**

В исследовании операций, оптимизация - это проблема определения входных значений функции, при которых она имеет максимальное или минимальное значение. Иногда на эти значения накладываются ограничения, такая задача известна как **ограниченная оптимизация.**

В программировании, оптимизация обычно обозначает модификацию кода и его установок компиляции для данной архитектуры для производства более эффективного ПО.

Типичные проблемы имеют настолько большое количество возможностей, что программисты обычно могут позволить использовать только "достаточно хорошо" решения.

**узкие места**

Для оптимизации нужно найти узкое место (англ. Hotspot), иногда называемое бутылочным горлышком (англ. Bottleneck): Критическая часть кода, которая является основным потребителем необходимого ресурса. Улучшение примерно 20% кода иногда влечет за собой изменение 80% результатов). Для поиска узких мест используются специальные программы - профайлер. Утечка ресурсов (памяти, дескрипторов и т.д.) также может привести к падению скорости выполнения программы, для их нахождения также применяются специальные программы (например, BoundsChecker).

Архитектурный дизайн системы особенно сильно влияет на ее производительность. Выбор алгоритма влияет на эффективность больше, чем любой другой элемент дизайна. Более сложные алгоритмы и структуры данные могут хорошо оперировать с большим количеством элементов, в то время как простые алгоритмы подходят для небольших объемов данных - накладные расходы на инициализацию более сложного алгоритма могут перевесить выгоду от его использования.

Чем больше памяти использует программа, тем быстрее она обычно выполняется. Например, программа-фильтр обычно читает каждую строку, фильтрует и выводит эту строку непосредственно. Поэтому она использует память только для хранения одной строки, но ее производительность обычно очень плохая. Производительность может быть значительно улучшена чтением целого файла и записью потом отфильтрованного результата, однако этот метод использует больше памяти. Кэширования результата также эффективно, однако требует большего количества памяти для использования.

**Простейшие приемы оптимизации программ по затратам процессорного времени**

Оптимизация по затратам процессорного времени особенно важна для расчетных программ, в которых большой удельный вес имеют математические вычисления. Здесь приведены некоторые приемы оптимизации, которые может использовать программист при написании исходного текста программы.

**Инициализация объектов данных**

Во многих программах какую-то часть объектов данных необходимо инициализировать, то есть присвоить им начальные значения. Такое присваивание выполняется либо в самом начале программы, или, например, в конце цикла. Правильная инициализация объектов позволяет сэкономить драгоценное процессорное время. Так, например, если речь идет о инициализации массивов, использование цикла, скорее всего, будет менее эффективным, чем объявление этого массива прямым присвоением.

**Программирование арифметических операций**

В том случае, когда значительная часть времени работы программы отводится арифметическим вычислениям, немалые резервы повышения скорости работы программы таятся в правильном программировании арифметических (и логических) выражений. Важно, что различные арифметические операции значительно различаются по быстродействию. В большинстве архитектур, самыми быстрыми являются операции сложения и вычитания. Медленнее является умножение, затем идет разделение. Например, вычисление значения выражения, где a - константа, для аргументов с плавающей точкой производится быстрее в виде, где - константа, которая исчисляется на этапе компиляции программы (фактически медленная операция деления заменяется быстрой операцией умножения). Для целочисленного аргумента x вычисления выражения 2 x быстрее провести в виде x + x (Операция умножения заменяется операцией сложения) или с использованием операции сдвига влево (что обеспечивает выигрыш не на всех процессорах). Подобные оптимизации называются понижением силы операций. Умножение целочисленных аргументов в константу на процессорах семейства x86 может быть эффективно выполнена с использованием асемблернихкоманд LEA, SHL и ADD вместо использования команд MUL / IMUL:

; Выходной операнд в регистре EAX ADD EAX, EAX; Умножение на 2 LEA EAX, [EAX + 2 \* EAX]; Умножения на 3 SHL EAX, 2; Умножение на 4 LEA EAX, [4 \* EAX]; Другой вариант реализации умножения на 4 LEA EAX, [EAX + 4 \* EAX]; Умножение на 5 LEA EAX, [EAX + 2 \* EAX]; Умножение на 6 ADD EAX, EAX, и т.д.

Подобные оптимизации является микроархитектурными и обычно производятся оптимизирующим компилятором прозрачно для программиста.

Относительно много времени тратится на обращение к подпрограммам (передача параметров через стек, сохранение регистров и адреса возврата, вызов конструкторов копирования). Если подпрограмма содержит малое количество действий, она может быть реализована Подставляем (англ. Inline) - все ее операторы копируются в каждое новое место вызова (существует ряд ограничений на inline-подстановки: например, подпрограмма не должна быть рекурсивной). Это ликвидирует накладные расходы на обращение к подпрограмме, однако ведет к увеличению размера исполняемого файла. Само по себе увеличение размера исполняемого файла не является существенным, однако в некоторых случаях исполняемый код может выйти за пределы кэша команд, приведет к значительному падению скорости выполнения программы. Поэтому современные оптимизируют компиляторы обычно настройки оптимизации по размеру кода и по скорости выполнения.

Быстродействие также зависит и от типа операндов. Например, в языке Turbo Pascal, из-за особенностей реализации целочисленной арифметики, операция сложения оказывается наиболее медленной для операндов типа Byte и ShortInt: несмотря на то, что переменные занимают один байт, арифметические операции для них двухбайтовые и при выполнении операций над этими типами производится обнуление старшего б регистров и операнд копируется из памяти в младший байт регистра. Это и приводит к дополнительным затратам времени.

Программируя арифметические выражения, следует выбирать такую ​​форму их записи, чтобы количество «медленных» операций было сведено к минимуму. Рассмотрим такой пример. Пусть необходимо вычислить многочлен 4-й степени:

a x 4 + b x 3 + c x 2 + d x + e

При условии, что вычисления степени проводится перемножением основания определенное количество раз, нетрудно найти, что в этом выражении содержится 10 умножений ( "медленных" операций) и 4 сложения ( "быстрых" сделок). Это же выражение можно записать в виде:

(((A x + b) x + c) x + d) x + e

Такая форма записи называется схемой Горнера. В этом выражении 4 умножения и 4 сложения. Общее количество сделок сократилось почти в два раза, соответственно уменьшится и время вычисления многочлена. Подобные оптимизации является алгоритмическими и обычно не выполняется компилятором автоматически.

**циклы**

Различается и время выполнения циклов разного типа. Время выполнения цикла со счетчиком и цикла с постусловием при всех прочих равных условиях совпадает, цикл с предусловием выполняется несколько дольше (примерно на 20-30%).

При использовании вложенных циклов следует иметь в виду, что расходы процессорного времени на обработку такой конструкции могут зависеть от порядка следования вложенных циклов. Например, вложенный цикл со счетчиком языке Turbo Pascal:

for j = 1 to 100000 do

for k: = 1 to 1000 do

a = 1;

for j = 1 to 1000 do

for k: = 1 to 100000 do

a = 1;

Цикл в левой колонке выполняется примерно на 10% дольше, чем в правой.

На первый взгляд, и в первом, и во втором случае 10000000 раз выполняется оператор присваивания, и затраты времени на это должны быть одинаковы в обоих случаях. Но это не так. Объясняется наше наблюдение тем, что инициализации цикла, то есть обработка процессором его заголовка с целью определения начального и конечного значений счетчика, а также шага увеличения счетчика требует времени. В первом случае 1 раз инициализируется внешний цикл и 100000 раз - внутренний, то есть всего выполняется 100001 инициализация. Во втором случае, как нетрудно подсчитать, таких инициализаций оказывается всего лишь 1001.

Аналогично ведут себя вложенные циклы с предусловием и с постусловием. Можно сделать вывод, что при программировании вложенных циклов по возможности следует делать цикл с наибольшим числом повторений самым внутренним, а цикл с наименьшим числом повторений - самым внешним.

Но, лучших результатов можно добиться, объединив несколько циклов в один, когда такое допустимо. К примеру:

for (int y = 0;

y <height; y + +)

for (int x = 0;

x <width; x + +)

put\_puxel (x, y, 0x00FF00)

for (int i = 0, n = width \* height; i <n; i + +) VRAMptr\_dd [i] = 0x00FF00;

Улучшение за счет снижения количества цикловых команд, и чем уже width тем эффективнее.

Если в циклах содержатся обращения к памяти (обычно при обработке массивов), порядок обхода адресов памяти должен быть по возможности последовательным, что позволяет максимально эффективно использовать кэш и механизм аппаратной предвыборка данных из памяти (англ. Hardware Prefetch). Классическим примером подобной оптимизации является изменение порядка следования вложенных циклов при выполнении умножения матриц.

При исчислении сумм часто используются циклы, содержащие одинаковые операции, связанные с каждым слагаемого. Это может быть, например, общий множитель (речь Turbo Pascal):

sum: = 0; for i: = 1 to 1000 do sum = sum + a \* x [i];

sum: = 0; for i: = 1 to 1000 do sum = sum + x [i]; Sum = a \* sum;

Очевидно, что вторая форма записи цикла оказывается более экономной.

**Инвариантные фрагменты кода**

Оптимизация инвариантных фрагментов кода тесно связана с проблемой оптимального программирования циклов. Внутри цикла могут встречаться выражения, фрагменты которых никак не зависят от управляющей переменной цикла. Их называют инвариантными фрагментами кода. Современные компиляторы часто определяют наличие таких фрагментов и выполняют их автоматическую оптимизацию. Такое возможно не всегда, и иногда производительность программы зависит целиком от того, как запрограммирован цикл. В качестве примера рассмотрим следующий фрагмент программы (речь Turbo Pascal):

for i: = 1 to n do

begin

...

for k: = 1 to p do

for m = 1 to q do

begin

a [k, m] = Sqrt (x \* k \* m - i) + Abs (u \* i - x \* m + k)

B [k, m] = Sin (x \* k \* i) + Abs (u \* i \* m + k)

end;

...

am = 0;

Bm = 0;

for k: = 1 to p do

for m = 1 to q do

begin

am = am + a [k, m] / c [k];

Bm = bm + b [k, m] / c [k];

end;

end;

Здесь инвариантными фрагментами кода является слагаемое Sin (x \* k \* i) в первом цикле по переменной m и операция деления на элемент массива c [k] во втором цикле по m. Значение синуса и элемента массива не изменяются в цикле по переменной m, следовательно, в первом случае можно вычислить значение синуса и присвоить его вспомогательной переменной, которая будет использоваться в выражении, что находится внутри цикла. Во втором случае можно выполнить распределение после завершения цикла по m. Таким образом, можно существенно сократить количество трудоемких арифметических операций.

приоритеты оптимизации

• интерфейсный, т.е.желательно заранее ВСЕ согласовать с другими участниками проекта, включая кому сколько МАКСИМУМ% CPUuse на конкретном ПК.

• PS: ПК не должен быть современным и многопроцессорным ... Если такого нет - сознательно сильно ограничить суммарный CPU расход, можно утилитами типа slowcpu; на MP доп.CPU отключая в BIOS / фиксируя все потоки на один CPU (но, это не эквивалент SP, т.к.ОС будет висеть на др.CPU, только для процесса программирования, не окончательного тестирования) - исходя из того что релиз будет значительно тормозные ожидаемого, и чем сложнее тема, жестче сроки, и меньше опыта в теме - тем сильнее ... К тому же запас ВСЕГДА нужен еще и на сколько-то нормальную работу в режиме DEBUG, TRACE

• PS: согласовать и обсудить - не спеша, и каждую мелочь вплоть до того где использовать ООП, а где нет - для ускорения. PS: Подсказка - STL и УРЧ .- не использовать НИГДЕ ... С # - аналогично, Скрипты - ТОЛЬКО там где нельзя без них обойтись ..., тем больше они часто привносят много проблем с багами ...

• PS: причем не забыв каждую итерацию теста - искусственно сбрасывать L12 кэши и BTB ... Чтобы соединив все модули потом не удивляться ...

• алгоритмический, минус - принцип чем оптимизирован - тем менее понятно, тем более багов, но это самый эффективный способ, тем более наиболее оптимизируются алгоритмы уже давно оптимизированы и можно посмотреть на их реализацию.

Этот пункт можно можно разделить на:

• математически-алгоритмический

• логически-алгоритмический

• другое функционально-алгоритмический, например использование асинхронного выполнения всему можно

...

• кэш-оптимизации на уровне алгоритма, включая HW L1, L2, программные-для данных, программные ресурсов

• использование спецкоманда, например MMX, SSE, ... Правда суммарно минусов больше чем плюсов, так как это чисто пиаровские технологии, например трудозатраты (тем более на изучение их работы - 1) на разных архитектурах 2) у разных производителей CPU 3) в различных их моделях ...)

PS: но, должен всегда быть вариант кода - без их наличия, в т.ч.для возм.портирования в будущем [возможно другими людьми] и просто для Safe версии исполнимого файла, т.к.некоторие PC-совместимые ПК, тем больше лаптопы и УРЧ. - Не совсем совместимы и тем более безглючной ...

• замена кода на ассемблерный, весьма спорная, т.к.не считая непортированности, и хорошего уровня оптимизации (некоторых) современных компиляторов, чтобы в случае большого объема кода или применения нестандартных для программирования трюков приводит к очень плачевным результата - ошибок, источник которые чаще всего так и не оказывается ... Но оптимизация тоже имеет право на жизнь, особенно в inline или небольших ф-иях, типа rol и т.п. И если программист хоть сколько знаком с ассемблером и архитектурой ПК, так как ассемблер позволяет иногда ускорить (некоторые) блоки в десятки и сотни раз за счет визуального представления о функционировании и как следствие оптимизации так чтобы эффективно использовать L1, L2, промах которых на современных ПК исчисляется сотнями тактов.

Для каждой итерации оптимизации в идеале должен храниться неоптимизированный пример для понимания алгоритма другими людьми, или самому разработчику, для тестов на скорость и главное эквивалентности работы с и без оптимизации.