

Оптимизация (информатика)

Материал из Википедии — свободной энциклопедии

Оптимизация — модификация системы для улучшения её эффективности. Система может быть одиночной компьютерной программой, цифровым устройством, набором компьютеров 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, требуемого программе для выполнения работы, даже ценой потребления большего объёма памяти; в приложениях, где важнее память, могут выбираться более медленные алгоритмы с меньшими запросами к памяти.

Более того, зачастую не существует универсального решения (хорошо работающего во всех случаях), поэтому инженеры используют компромиссные (англ. *tradeoff*) решения для оптимизации только ключевых параметров. К тому же, усилия, требуемые 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 произнёс, а Дональд Кнут впоследствии часто повторял известное высказывание: «Преждевременная оптимизация — это корень всех бед». Очень важно иметь для начала озвученный алгоритм и работающий прототип.

Содержание

Основы

Компромиссы (tradeoff)

Различные области

Узкие места

Простейшие приёмы оптимизации программ по затратам процессорного времени

Инициализация объектов данных

Программирование арифметических операций

Циклы

Инвариантные фрагменты кода

См. также

Литература

Ссылки

Основы

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;
```

Подразумевая, 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 самую функциональность. Однако, значительное улучшение производительности часто может быть достигнуто и с помощью удаления избыточной функциональности. Например, если допустить, что программе не требуется поддерживать более, чем 100 элементов при вводе, то возможно использовать статическое выделение памяти вместо более медленного динамического.

Компромиссы (tradeoff)

Оптимизация в основном фокусируется на одиночном или повторном времени выполнения, использовании памяти, дискового пространства, пропускной способности или некотором другом ресурсе. Это обычно требует компромиссов — один параметр оптимизируется за счёт других. Например, увеличение размера программного кэша чего-либо улучшает производительность времени выполнения, но также увеличивает потребление памяти. Другие распространённые компромиссы включают прозрачность кода и его выразительность, почти всегда ценой деоптимизации. Сложные специализированные алгоритмы требуют больше усилий по отладке 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 настроек компиляции для данной архитектуры для производства более эффективного ПО.

Типичные проблемы имеют настолько большое количество возможностей, что программисты обычно могут позволить использовать только «достаточно хорошее» решение.

Узкие места

Для оптимизации требуется найти узкое место (англ. *bottleneck* - бутылочное горлышко): критическую часть кода, которая является основным потребителем необходимого ресурса. Улучшение примерно 20 % кода иногда влечёт за собой изменение 80 % результатов, согласно принципу Парето. Утечка ресурсов (памяти, дескрипторов и т. д.) также может привести к падению скорости выполнения программы. Для поиска таких утечек используются специальные отладочные инструменты, а для обнаружения узких мест применяются программы — профайлеры.

Архитектурный дизайн системы особенно сильно влияет на её производительность. Выбор алгоритма влияет на эффективность больше, чем любой другой элемент дизайна. Более сложные алгоритмы и структуры данных могут хорошо оперировать большим количеством элементов, в то время как простые алгоритмы подходят для небольших объёмов 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 её производительность обычно очень плохая. Производительность может быть значительно улучшена чтением целого файла и записью потом отфильтрованного результата, однако этот метод использует больше памяти. Кэширование результата также эффективно, однако требует большего количества памяти для использования.

Простейшие приёмы оптимизации программ по затратам процессорного времени

Оптимизация по затратам процессорного времени особенно важна для расчётных программ, в которых большой удельный вес имеют математические вычисления. Здесь перечислены некоторые приёмы 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, либо, например, в конце цикла. Правильная инициализация объектов позволяет сэкономить драгоценное процессорное время. Так, например, если речь идет об 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. Более медленным является умножение, затем идёт деление. Например, вычисление значения выражения $\frac{x}{a}$, где a —

константа, для аргументов с плавающей точкой производится быстрее в виде $x \cdot b$, где $b = \frac{1}{a}$ — константа, вычисляемая на этапе компиляции программы (фактически медленная операция деления заменяется быстрой операцией умножения). Для целочисленного аргумента x вычисление выражения $2x$ быстрее произвести 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 на константу на процессорах семейства x86 может быть эффективно выполнено с использованием ассемблерных команд LEA, SHL и ADD вместо использования команд MUL/IMUL:

```
; Исходный операнд в регистре EAX  
ADD  EAX, EAX           ; умножение на 2  
  
LEA   EAX, [EAX + 2*EAX] ; умножение 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 являются 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 число действий, она может быть реализована **подставляемой** (англ. *inline*) — все её 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, что повлечет значительное падение скорости исполнения программы. Поэтому современные оптимизирующие компиляторы обычно имеют настройки оптимизации по размеру кода и по скорости выполнения.

Быстродействие также зависит и от типа операндов. Например, в языке Turbo Pascal, ввиду 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 байта 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:

$$ax^4 + bx^3 + cx^2 + dx + e$$

При условии, что вычисление степени производится перемножением основания определенное число раз, нетрудно найти, что в этом выражении содержится 10 умножений («медленных» операций) и 4 сложения («быстрых» операций). Это же самое выражение можно записать в виде:

$$(((ax + b)x + c)x + d)x + e$$

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

выполняются компилятором автоматически.

Циклы

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

При использовании вложенных циклов следует иметь в виду, что затраты процессорного времени на обработку такой конструкции могут зависеть от порядка следования вложенных циклов. Например, вложенный цикл со счетчиком на 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 100 000 000 раз выполняется оператор присваивания и затраты времени на это должны быть одинаковы в обоих случаях. Но это не так. Объясняется это тем, что инициализации цикла (обработка процессором его заголовка с целью определения начального и конечного значений счётчика, а также шага приращения счётчика) требует времени. В первом случае 1 раз инициализируется внешний цикл и 100 000 раз — внутренний, то есть всего выполняется 100 001 инициализация. Во втором случае 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 наименьшим числом повторений самым внешним, а цикл с наибольшим числом повторений — самым внутренним.

Если в циклах содержатся обращения к памяти (обычно при обработке массивов), для максимально эффективного использования кэша и механизма аппаратной предвыборки данных из памяти (англ. *Hardware Prefetch*) порядок обхода адресов памяти должен быть по возможности последовательным. Классическим примером подобной оптимизации является смена порядка следования вложенных циклов при выполнении умножения матриц.

При вычислении сумм часто используются циклы, содержащие одинаковые операции, относящиеся к каждому слагаемому. Это может быть, например, общий множитель (язык 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;
```

Цикл в левой колонке более оптимален.

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 (язык 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 являются слагаемое $\text{Sin}(x * k * i)$ в первом цикле по переменной m и операция деления на элемент массива $c[k]$ во втором цикле по m . Значения 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, находящемся внутри цикла. Во втором случае можно выполнить деление после завершения цикла по m . Таким образом, можно существенно сократить количество трудоёмких арифметических операций.

См. также



- [Граф потока управления](#)
- [Отложенные вычисления](#)
- [Мемоизация](#)
- [Теория массового обслуживания](#)

Литература

- [Бентли, Джон Луис](#). Writing Efficient Programs. ISBN 0-13-970251-2
- [Дональд Кнут](#). Искусство программирования = The Art of Computer Programming. — 3-е изд. — М.: Вильямс, 2006. — Т. 1: Основные алгоритмы. — 720 с. — ISBN 0-201-89683-4.
- [Дональд Кнут](#). Искусство программирования = The Art of Computer Programming. — 3-е изд. — М.: Вильямс, 2007. — Т. 2: Получисленные алгоритмы. — 832 с. — ISBN 0-201-89684-2.
- [Дональд Кнут](#). Искусство программирования = The Art of Computer Programming. — 2-е изд. — М.: Вильямс, 2007. — Т. 3: Сортировка и поиск. — 824 с. — ISBN 0-201-89685-0.
- [С. А. Немнюгин](#). Практикум // Turbo Pascal. — 2-е изд. — СПб.: Питер, 2007. — 268 с. — ISBN 5-94723-702-4.

- *Крис Касперски*. Техника оптимизации программ. Эффективное использование памяти. — СПб.: БХВ-Петербург, 2003. — 464 с. — ISBN 5-94157-232-8.

Ссылки

- Intel 64 and IA-32 Architectures Optimization Reference Manual (<http://www.intel.com/Assets/PDF/manual/248966.pdf>) 
 - AMD Athlon Processor x86 Code Optimization Guide (https://web.archive.org/web/20100215205102/http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/22007.pdf) 
 - <http://www.agner.org/optimize/#manuals>
-

Источник — [https://ru.wikipedia.org/w/index.php?title=Оптимизация_\(информатика\)&oldid=113383459](https://ru.wikipedia.org/w/index.php?title=Оптимизация_(информатика)&oldid=113383459)

Эта страница в последний раз была отредактирована 3 апреля 2021 в 20:59.

Текст доступен по лицензии Creative Commons Attribution-ShareAlike; в отдельных случаях могут действовать дополнительные условия.

Wikipedia® — зарегистрированный товарный знак некоммерческой организации Wikimedia Foundation, Inc.