мощь и беспомощность автоматической оптимизации

крис касперски ака мыщъх, no-email

к концу 90х годов компиляторы по своей эффективности вплотную приблизились к ассемблеру, однако, все еще существует множество конструкций, неподдающихся автоматической оптимизации, но легко трансформируемых вручную. покажем как надо и как не надо оптимизировать программы на примере: Microsoft Visual C++, Intel C++, Borland Builder, GCC и Hewlett-Packard C++

введение

С точки зрения прикладного программиста, компилятор — это черный ящик, заглатывающий исходный текст и выплевывающий двоичный файл. Какие процессы протекают в его "пищеварительном тракте" — неизвестно. Разработчики компиляторов крайне поверхностно описывают механизмы оптимизации в прилагаемой документации, так и не давая ответа на вопрос: _что_ именно оптимизирует компилятор, а что нет.

Как следствие — одни разработчики пишут ужасно кривой код, надеясь, что все огрехи исправит компилятор (он же ведь "оптимизирующий!"). Другие же, наоборот, пытаются помощь компилятору, оптимизируя программу вручную и выполняя кучу глупых и ненужных действий, например, заменяя `a = b/4` на `a = b >> 2`, хотя _любой_ компилятор сделает это и сам, а вот поместить в регистр переменную, переданную по ссылке, он уже не решается (почему — см. "удаление лишних обращений к памяти"), то же самое относится и к выносу инвариантных функций из тела цикла.

Для достижения наивысшей эффективности, необходимо помочь компилятору, придерживаясь определенных привил программирования (кстати говоря, не описанных в штатной документации). Если вы недовольны быстродействием откомпилированной программы, не спешите переписать ее на ассемблер. Попробуйте сначала оптимизировать код путем реконструкции исходного текста. В большинстве случаев, вы получите тот же самый результат, но потратите меньше времени и сохраните переносимость.

что не надо оптимизировать

Начнем с того, что _не_ _надо_ оптимизировать, позволяя транслятору сделать это за нас (нехай делает). В частности, практически все оптимизирующие компиляторы умеют ***вычислять константы на стадии трансляции***. В различных русскоязычных источниках этот прием оптимизации называется как "сверткой", так и "размножением" констант, что соответствует английским терминам "constant folding/propagation". Еще один английский термин из той же кучи: "constant elimination" (буквально — "изгнание констант"). Все это синонимы и описывают один и тот же механизм вычисления константных выражения (как целочисленных, так и вещественных), в результате чего **`a = 2 * 2`** превращается в `a = 4`, а **`x = 4*y/2`** в `x = 2*y`.

Побочным эффектом оптимизации становится потеря переполнения (если таковое имело место быть). С точки зрения математика выражения **`foo = bar/4*4`** и `foo = bar` полностью эквиваленты, но если переменные `foo` и `bar` целые, то не оптимизированный вариант обнуляет два младших бита `bar`! Некоторые программисты умышленно используют этот примем, вместо того, чтобы воспользоваться "`foo = bar & (~3)`", а ругаются на "глючный" оптимизатор!

За исключением Intel C++ все рассматриваемые компиляторы поддерживают "***улучшенную свертку констант***" ("***advanced constant folding/propagation***"), заменяя все константные переменные их непосредственным значением, в результате чего выражение: **`a = 2; b = 2 * a; c = b - a;`** превращается в `c = 2`, а переменные `a` и `b` (если они нигде более не используются) уничтожаются.

константная подстановка в условиях

В операторах ветвления ("`if`", "`?`" и "`switch`") константные условия встречаются редко и обычно являются следствием чрезмерного увлечение `#define`, вот, например, как здесь:

```
#define MAX_SIZE       1024
#define REQ_SIZE       512
#define HDR_SIZE       16
…
int a = REQ_SIZE + HDR_SIZE;
if (a <= MAX_SIZE) foo(a); else return ERR_SIZE;
```

Листинг 1 не оптимизированный вариант

За исключением Intel C++ все рассматриваемые компиляторы выполняют константную подстановку, оптимизируя код, избавляясь от ветвления и ликвидируя "мертвый код", который никогда не выполняется:

```
foo(528);
```

Листинг 2 оптимизированный вариант

Выполнять эту оптимизацию вручную совершенно необязательно, поскольку оптимизированный листинг намного менее нагляден и совершенно негибок. По правилам этикета программирования, _все_ константы должны быть вынесены в `#define`, что значительно уменьшает число ошибок.

удаление копий переменных

Для повышения читабельности листинга программисты обычно загоняют каждую сущность в "свою" переменную, не обращая внимания на образующуюся избыточность: многие переменные либо полностью дублируются, либо связаны друг с другом несложным математическим соотношением и для экономии памяти их можно сократить алгебраическим путем.

В англоязычной литературе данный примем называется "***размножением копий***" ("***copy propagation***"), что на первый взгляд не совсем логично, но если задуматься, то все проясняется: да, мы сокращаем переменные, размножая копии, хранящихся в них значений, что наглядно продемонстрировано в следующем примере:

```
main(int n, char** v)
{
        int a, b;
        …
        a = n+1;
        b = 1-a;        // избавляется от переменной a: (1 - (n + 1));
        return a-b;     // избавляется от переменной b: ((n + 1) - (1 - (n + 1)));
}
```

Листинг 3 переменные a и b — лишнее

После оптимизации переменные `a` и `b` исчезают, а `return` возвращает значение выражения `(2*n+1)`:

```
main(int n, char** v)
{
        return 2*n+1;
}
```

Листинг 4 оптимизированный вариант, выбросивший лишние переменные

устранение хвостовой рекурсии

Хвостовой рекурсией (***tail recursion***) называется такой тип рекурсии, при котором вызов рекурсивной функции следует непосредственно за оператором `return`. Классическим примером тому является алгоритм вычисления факториала:

```
int fact(int n, int result)
{
        if(n == 0)
        {
                return result;
        }
                else
        {
                return fact(n - 1, result * n);
        }
}
```

Листинг 5 хвостовая рекурсия до оптимизации

Вызов функции — достаточно "дорогостоящая" (в плане процессорных тактов) операция и за исключением Intel C++ все рассматриваемые компиляторы трансформирует рекурсивный вызов в цикл:

```
for(i=0; i<n; i++) result *= n;
```

Листинг 6 хвостовая рекурсия после оптимизации

Естественно, оптимизированный код менее нагляден, поэтому выполнять такое преобразование вручную — совершенно необязательно.

что надо оптимизировать

Теперь поговорим о том, с чем оптимизирующие компиляторы не справляются и начинают буксовать, резко снижая эффективность целевого кода. Помочь им выбраться из болота — наша задача! Чип и Дейл уже спешат! Ну а мыщщъх вращает хвостом. Руководит, значит.

Начнем с функций. Из всех рассматриваемых компиляторов только Intel C++ поддерживает глобальную оптимизацию, а остальные — транслируют функции по отдельности, задействую "сквозную" оптимизацию только на встраиваемых (inline) функциях. Отсюда: чем выше степень дробления программы на функции (и чем меньше средний размер одной функции), тем ниже качество оптимизации, не говоря уже о накладных расходах на передачу аргументов, открытие кадра стека и т. д.

На мелких функциях, состоящих всего из нескольких строк, оптимизатору просто негде "развернуться", а задействовать агрессивный режим подстановки, "вживляющий" все мелкие функции в тело программы нежелательно, поскольку это приводит к чрезмерному "разбуханию" программного кода.

Оптимальная стратегия выглядит так: выключаем режим автоматического встраивания и стремиться программировать так, чтобы средний размер каждой функции составлял не менее 100-200 строк.

удаление неиспользуемых функций

Большинство компиляторов не удаляют неиспользуемые функции из исходного текста, поскольку используют технологию раздельной компиляции, транслируя исходные тексты в объектные модули, собираемые линкером в исполняемый файл, динамическую библиотеку или драйвер.

Функция, реализованная в одном объектном файле, может вызываться из любых других, но информацией о других модулях компилятор не обладает и потому удалять "не используемые" (с его точки зрения) функции не имеет права.

Теоретически, неиспользуемые функции должен удалять линкер, но популярные форматы объектных файлов к этому не располагают и в грубом приближении представляют собой набор секций (.text, .data и т. д.), каждая из которых с точки зрения линкера представляет монолитный блок, внутрь которого линкер не лезет, а просто объединят блоки тем или иным образом.

Вот потому-то и не рекомендуется держать весь проект в одном файле (особенно если это библиотека). Помещайте в файл только "родственные" функции, _всегда_ используемые в паре и по отдельности не имеющие никакого смысла. Посмотрите как устроена стандартная библиотека языка Си — большинство функций реализованы в "своем" собственном файле, компилируемом в obj, содержащим _только_ эту функцию и ничего сверх нее! Множество таких obj объединяются библиотекарем в один lib-файл, откуда линкер свободно достает любую

необходимую функцию, не таща ничего остального! Собственно говоря, именно для этого библиотеками и придумали. Программисты, помещающие реализации всех функций своей библиотеки в один-единственный файл, совершают большую ошибку!

Из всех рассматриваемых компиляторов, только Intel C++ умеет отслеживать неиспользуемые функции, предотвращая их включение в obj (для этого ему необходимо указать ключ–ipo, активирующий режим глобальной оптимизации).

внос инвариантных функций из циклов

Инвариантными называются функции, результат работы которых не зависит от параметров цикла и потому их достаточно вычислить всего один раз. Компиляторы, к сожалению, так не поступают, поскольку, транслируют все функции по отдельности и не могут знать какими побочными эффектами обладает та или иная функция (исключение составляют встраиваемые функции, непосредственно вживляемые в код программы).

Рассмотрим типичный пример:

```
for(a=0;a<strlen(s);a++) b+=s[a];
```

Листинг 7 не оптимизированный вариант с инвариантом в теле цикла

Если только компилятор не заинлайнит функцию `strlen`, она будет вычисляется на каждой итерации цикла, что приведет к значительному снижению производительности. Но если вынести инвариант за пределы цикла, все будет ОК:

```
t = strlen(s);
for(a=0;a<t;a++) b+=s[a];
```

Листинг 8 вынос инварианта за пределы цикла

нормализация циклов

Нормализованным называется цикл, начинающийся с нуля и в каждой итерации увеличивающий свое значение на единицу. В книгах по программированию можно встретить утверждение, что нормализованный цикл компилируется в более компактный и быстродействующий код, однако, это только теоретическая схема и многие процессорные архитектуры (включая x86) предпочитают иметь дело с циклом, стреляющимся к нулю.

Рассмотрим типичный цикл:

```
for (a = from; a < to; i+=(-step))
{
        // тело цикла
}
```

Листинг 9 ненормализованный цикл

Алгоритм нормализации выглядит так:

```
for (NCL = 0; i < (to - from + step)/step  - 1; 1)
{
        i = step*NLC + from;
        // тело цикла
}
i =  step * _max((to - from + step)/step, 0) + from;
```

Листинг 10 нормализованный цикл

Наибольшую отдачу нормализация дает на циклах с заранее известным количеством итераций, т. е. когда выражение `(to - from + step)/step` представляет собой константу, вычисляемую еще на стадии трансляции.

Формально, все рассматриваемые компиляторы поддерживают нормализацию циклов, но не всегда задействуют этот механизм оптимизации, поэтому в наиболее ответственных ситуациях циклы лучше всего нормализовать вручную.

разворот циклов

Процессоры с конвейерной архитектурой (к которым относится и x86) плохо справляются с ветвлениями (а циклы как раз и представляют одну из разновидностей

ветвлений), резко снижая свою производительность. Образно их можно сравнить с гоночной машиной, ползущей по петляющей дороге. И у машины, и у процессора максимальная скорость достигается только на участках свободных от ветвлений.

Компактные циклы вида `for(a=0;a<n;a++) *dst++= *src++;` исполняются крайне медленно и должны быть ***развернуты*** (**unrolled**). Под "разворотом" в общем случае понимается многократное дублирование цикла, которое в классическом случае реализуется так:

```
for(i=1; i<n;i+)
        k += (n % i);
```

Листинг 11 цикл до разворота

```
for(i=1; i<n;i+=4)
{
        k += (n % i) +\
        (n % i+1) + \
        (n % i+2) + \
        (n % i+3);
}

// выполняем оставшиеся итерации
for(i=4; i<n;i++) k += (n % i);
```

Листинг 12 цикл, развернутый на 4 итерации (меньшей размер, большая скорость)

За исключением Microsoft Visual C++, все остальные рассматриваемые компиляторы умеют разворачивать циклы и самостоятельно, но… делают это настолько неумело, что вместо ожидаемого увеличения производительности сплошь и рядом наблюдается ее падение, поэтому автоматический разворот лучше сразу запретить и оптимизировать программу вручную, подбирая подходящую степень разворота опытным путем (вместе с профилировщиком).

Тут ведь как — чем сильнее разворот, тем больше места занимает код и появляется риск, что в кэш первого уровня он может вообще не влезть, вызывая обвальное падение производительности! (Подробнее о влиянии степени разворота на быстродействие можно прочитать в моей "**технике оптимизации**", электронная копию которой как обычно лежит на моем мыщъхином ftp://nezumi.org.ru).

программная конвейеризация

Классический разворот цикла порождает зависимость по данным. Вернемся к листингу 14. Несмотря на то, что загрузка обрабатываемых ячеек происходит параллельно, следующая операция сложения начинается только после завершения предыдущей, а все остальное время процессор ждет.

Чтобы избавиться от зависимости по данным, необходимо развернуть не только цикл, но и "расщепить" переменную, используемую для суммирования. Такая техника оптимизации называется ***программной конвейеризацией*** (***software pipelining***) и из всех рассматриваемых компиляторов ее поддерживает только GCC, да и то лишь частично. В тоже самое время, она элементарно реализуется "руками":

```
// обрабатываем первые XXL - (XXL % 4) итераций
for(i=0; i<XXL;i+=4)
{

        sum_1 += a[i+0];
        sum_2 += a[i+2];
        sum_3 += a[i+3];
        sum_4 += a[i+4];
}

// обрабатываем оставшийся "хвост"
for(i-=XXL; i<XXL;i++)
        sum += a[i];

// складываем все воедино
sum += sum_1 + sum_2 + sum_3 + sum_4;
```

Листинг 13 оптимизированный вариант

авто-параллелизм

Многопроцессорные машины, двуядерные процессоры и процессоры с поддержкой Hyper-Threading эффективны лишь при обработке многопоточных приложений, поскольку всякий поток в каждый момент времени может исполняться только на одном процессоре. Программы, состоящие всего из одного потока на многопроцессорной машине исполняются с той же скоростью, что и на однопроцессорной (или даже чуть-чуть медленнее, за счет дополнительных накладных расходов).

Для оптимизации под многопроцессорные машины, следует разбивать циклы с большим количеством итераций на N циклов меньшего размера, помещая каждый из них в свой поток, где N — количество процессоров, обычно равное двум. Такая техника оптимизации называется ***авто-параллелизмом*** (***auto-parallelization***) и наглядно демонстрируется следующим примером:

```
for (i=0; i<XXL; i++)
        a[i] = a[i] + b[i] * c[i];
```

Листинг 14 не оптимизированный вариант

Поскольку зависимость по данным отсутствует, цикл можно разбить на два. Первый будет обрабатывать ячейки от 0 до XXL/2, а второй — от XXL/2 до XXL. Тогда на двухпроцессорной машине скорость выполнения цикла практически удвоится.

```
/* поток A */
for (i=0; i<XXL/2; i++)
        a[i] = a[i] + b[i] * c[i];

/* поток B */
for (i=XXL/2; i<XXL; i++)
        a[i] = a[i] + b[i] * c[i];
```

Листинг 15 оптимизированный вариант

Intel C++ – единственный из всех рассматриваемых компиляторов, поддерживающий технику авто-парализации, активируемую ключом **`-parallel`**, однако, качество оптимизации оставляет желать лучшего и эту работу лучше осуществлять вручную.

упорядочивание обращений к памяти

При обращении к одной-единственной ячейки памяти, в кэш первого уровня загружается целая строка, длина которой в зависимости от типа процессора варьируется от 32- до 128- или даже 256 байт, поэтому большие массивы выгоднее всего обрабатывать по строкам, а не по столбцам.

```
for(j=0;j<m;j++)
        for(i=0;i<n;i++)
                a[i][j] = b[i][j] + c[i][j];
```

Листинг 16 обработка массивов по столбцам (не оптимизированный вариант)

Здесь три массива обрабатываются по столбцам, что крайне непроизводительно и для достижения наивысшей эффективности циклы i и j следует поменять местами. Устоявшегося названия у данной методики оптимизации нет и в каждом источнике она называется по-разному: ***loop permutation/interchange/reversing, rearranging array dimensions*** и т. д. Как бы там ни было, оптимизированный вариант выглядит так:

```
for(i=0;i<n;i++)
        for(j=0;j<m;j++)
                a[i][j] = b[i][j] + c[i][j];
```

Листинг 17 обработка массивов по столбцам (оптимизированный вариант)

Все рассматриваемые компиляторы поддерживают данную стратегию оптимизации, однако их интеллектуальные способности очень ограничены и со следующим примером справляется только Hewlett-Packard C++:

```
for (i=0; i<n; i++)
        for (j=0; j<n; j++)
```

```
for (k=0; k<n; k++)
        a[j][i] = a[j][i] + b[k][i] * c[j][k];
```

Листинг 18 сложный случай обработки данных по столбцам

удаление лишних обращений к памяти

Компиляторы стремятся размещать переменные в регистрах, избегая "дорогостоящих" операций обращения к памяти, однако, компилятор никогда не может быть уверен, адресуют ли две переменных различные области памяти или обращаются к одной и той же ячейке памяти.

Вот, например:

```
f(int *a, int *b)
{
        int x;
        x = *a + *b;    // сложение содержимого двух ячеек
        *b = 0x69;      // изменение ячейки *b, адрес которой не известен компилятору
        x += *a;        // нет гарантии что запись в ячейку *b не изменила ячейку *a
}
```

Листинг 19 пример с лишними обращениями к памяти, от которых нельзя избавиться

Компилятор не имеет права на размещение содержимого ячейки `*a` в регистровой переменной, поскольку если ячейки `*a` и `*b` частично или полностью перекрываются, модификация ячейки `*b` приводит к неожиданному изменению ячейки `*a`! Бред, конечно, но ведь Стандарт этого не запрещает, а компилятор _обязан_ следовать Стандарту, иначе, его место — на свалке.

Тоже самое относится и к следующему примеру:

```
f(char *x, int *dst, int n)
{
        int i;
        for (i = 0; i < n; i++) *dst += x[i];
}
```

Листинг 20 пример с лишними обращениями к памяти, от которых можно избавиться вручную

Компилятор не имеет права выносить переменную `dst` за пределы цикла, в результате чего обращения к памяти происходят в каждой итерации. Чтобы повысить производительность, код должен быть переписан так:

```
f(char *x, int *dst, int n)
{
        int i,t =0;
        for (i=0;i<n;i++) t+=x[i];     // сохранение суммы во временной переменной
        *dst+=t;                       // запись конечного результата в память
}
```

Листинг 21 оптимизированный вариант

регистровые ре-ассоциации

На x86 платформе регистров общего назначения всего семь и их всегда не хватает, особенно в циклах. Чтобы втиснуть в регистры максимальное количество переменных (избежав тем самым обращения к медленной оперативной памяти) приходится прибегать во всяким ухищрениям. В частности, совмещать счетчик цикла с указателем на обрабатываемые данные.

Код вида "`for (i = 0; I < n; i++) n+=a[i];`" легко оптимизировать, если переписать его так: "`for (p= a; p < &a[n]; p++) n+=*p;`" Насколько известно мыщъх'у впервые эта техника использовалась в компиляторах фирмы Hewlett-Packard, где она фигурировала под термином *register reassociation*, а вот остальные рассматриваемые нам компиляторы этого делать, увы, не умеют.

Рассмотрим еще один пример, демонстрирующий оптимизацию цикла с тройной вложенностью:

```
int a[10][20][30];
void example (void)
```

```
{
        int i, j, k;
        for (k = 0; k < 10; k++)
                for (j = 0; j < 10;j++)
                        for (i = 0; i < 10; i++)
                                a[i][j][k] = 1;
}
```

Листинг 22 не оптимизированный кандидат на регистровую ре-ассоциацию

Для достижения наибольшей производительности код следует переписать так (разворот циклов опущен для наглядности):

```
int a[10][20][30];
void example (void)
{
        int i, j, k;
        register int (*p)[20][30];
        for (k = 0; k < 10; k++)
                for (j = 0; j < 10; j++)
                        for (p = (int (*)[20][30]) &a[0][j][k], i = 0; i < 10; i++)
                                *(p++[0][0]) = 1;
}
```

Листинг 23 оптимизированный вариант — счетчик цикла совмещен с указателем на массив

заключение

Собирать свою коллекцию как надо и как не надо оптимизировать программы мыщъх начал еще давно (здесь приведена лишь крошечная ее часть). Время шло, компиляторы совершенствовались и все больше примеров перемещалось из первой категории во вторую. А затем… разработчики компиляторов поутихли и со временен Microsoft Visual C++ 6.0 новых рывков что-то не наблюдается, поэтому у статьи есть все шансы сохранить свою актуальность в течении нескольких лет. А, возможно, и нет.