техника оптимизации под LINUX
часть II — ветвления

крис касперски ака мыщъх, no-email

сегодня мы продолжим сравнение Linux-компиляторов, начатое в прошлом номере журнала и рассмотрим оптимизацию условных переходов и операторов типа switch. механизмы их трансформации достаточно многочисленны и разнообразны. за последнее время было предложено множество новых идей, воплощенных в компиляторах GCC 3.3.4 и Intel C++ 8.0 (сокращенно icl), однако, древний Microsoft Visual C++ 6.0 (сокращенного msvc) не сдает своих позиций, так что не спешите списывать его в утиль и сдавать на свалку.

оптимизация ветвлений/branch

Ветвления (по-английски ***branch***, они же ***условные/безусловные переходы***) относятся к фундаментальным основам любого языка, без которых не обходится ни одна программа. Даже "hello, world!"! Ведь выход из функции main — это тоже ветвление, пускай и неявное (но ведь процессор синтаксисом языка не проведешь!). А сколько ветвлений содержит сама функция printf? А Библиотека времени исполнения?

Суперскалярные микропроцессоры, построенные по конвейерной архитектуре (а все современные микропроцессоры именно так и устроены), быстрее всего выполняют линейный код и ненавидят ветвления. В лучшем случае они дезориентируют процессор, слегка приостанавливая выполнение программы, в худшем же — полностью очищают конвейер. А на последних Pentium'ах он очень длинный (и с каждой последующей моделью становится все длиннее и длиннее). Быстро его не заполнишь… на это может уйти не одна сотня тактов, что вызовет обвальное падение производительности.

Оптимизация переходов дает значительный выигрыш, особенно если они расположены внутри циклов (кстати говоря, циклы это те же самые переходы), поэтому качество компилятора не в последнюю очередь определяется его умением полностью или частично избавляться от ветвлений.

выравнивание переходов

Процессоры, основанные на ядрах Intel P6 и AMD K6, не требуют выравнивания переходов, исключая тот случай, когда целевая инструкция или сама команда перехода расщепляется границей кэш-линейки напополам, в результате чего наблюдается значительное падение производительности. Наибольший ущерб причиняют переходы, находящиеся в циклах с компактным телом и большим уровнем вложения.

Чтобы этого не происходило, некоторые компиляторы прибегают к выравниванию, располагая инструкции перехода по кратным адресам и заполняют образующиеся дыры незначащими инструкциями, такими как XCHG EAX,EAX (обмен содержимого регистров EAX местами) или MOV EAX, EAX (пересылка содержимого EAX в EAX). Это увеличивает размер кода и несколько снижает его производительность, поэтому бездумное выравнивание (оно же "агрессивное") только вредит.

Легко показать, что машинная команда требует выравнивания в тех, и только тех случаях, когда условие (`(addr % cache_len + sizeof(ops)) > cache_len`) становится истинным. Здесь: addr – линейный адрес инструкции, cache_len размер кэш-линейки (в зависимости от типа процессора равный 32-, 64- или 128 байтам), ops – сама машинная инструкция. Количество выравнивающих байт рассчитывается по формуле: (`cache_len - (addr % cache_len)`).

Именно так поступают все ассеблерщики, но только не компиляторы! MSVC и icl вообще не выравнивают переходов, а gcc выравнивает целевую инструкцию на фиксированную величину, кратную степени двойки (т. е. 2, 4, 8…), что является крайне неэффективной стратегией выравнивания, однако, даже плохая стратегия все же лучше, чем совсем никакой. Кстати говоря, именно поэтому, компиляторы msvc и icl генерируют неустойчивый код, точнее код с "плавающей" производительностью, быстродействие которого главным образом зависит от того, расщепляются ли глубоко вложенные переходы или нет. А это в свою очередь зависит от множества трудно прогнозируемых обстоятельств, включая фазу луны и количество осадков.

Учитывая, что средняя длина x86-инструкций составляет 3.5 байта, целесообразнее всего выравнивать переходы по границе четырех байт (ключ `-falign-jumps=4` компилятора gcc). Ключ `-fno-align-jumps` (или эквивалентный ему `-falign-jumps=1`) отключает выравнивание. Ключ `-falign-jumps=0` задействует выравнивание по умолчанию, автоматически выбираемое компилятором в зависимости от типа процессора.

```
if ((addr % cache_len + sizeof(ops)) > cache_len)
                        align = cache_len - (addr % cache_len);
```

Листинг 1 формула для расчета оптимальной кратности выравнивания для процессоров от Intel Pentium II и выше

частичное вычисление условий

"Летят два крокодила — один квадратный, другой тоже на север" — вот хороший пример техники быстрого булевского вычисления (оно же "частичное вычисление условий", "Partial evaluation of test conditions" или "short-circuiting"). Собственно, ничего "быстрого" в нем нет. Если первое из нескольких условий, связанных оператором AND, ложно (где видели квадратных крокодилов?), остальные уже не вычисляются. Соответственно, если первое из нескольких условий, связанных оператором OR, истинно, вычислять остальные нет нужды.

Кстати говоря, это отнюдь не свойство оптимизатора (как утверждают некоторые рекламные букеты), а требование языка, без которого ветвления вида: `if (x && (y/x))` были бы невозможны. Все три рассматриваемых компилятора, поддерживают быстрое булево вычисление.

```
if ((a == b) || (c == d))…
```

Листинг 2 если выражение (a == b) истинно, выражение (c == d) уже не проверяется

удаление избыточных проверок

Небрежное кодирование часто приводит к появлению избыточных или даже заведомо ложных проверок, полностью или частично дублирующих друг друга, например: "`if (a > 9) … if (a > 6)…`". Очевидно, что вторая проверка лишняя и icl благополучно ее удаляет. Остальные рассматриваемые компиляторы такой способностью не обладают, послушно генерируя тупой код.

```
if (n > 10) a++; else return 0;
if (n > 5) a++; else return 0;        // избыточная проверка
if (n < 2) a++; else return 0;        // заведомо ложна проверка
```

Листинг 3 не оптимизированный вариант

```
if (n > 10) a+=2; else return 0;
```

Листинг 4 оптимизированный вариант

удаление проверок нулевых указателей

Программисты до сих пор не могут определиться: кто должен осуществлять проверку аргументов — вызывающая или вызываемая функция. Многие стандарты кодирования предписывают выполнять такую проверку обоим:

```
f1(int *p)
{
        if (p) return *p+1; else return -1;
}

f2(int *p)
{
        if (p) *p = 0x69; else return -1;
        return f1(p);
}
```

Листинг 5 не оптимизированный вариант

Лишние проверки увеличивают размер кода и замедляют его выполнение (особенно если находятся глубоко в цикле), поэтому компилятор gcc поддерживает специальный ключ, `-fdelete-null-pointer-checks`, вычищающий их из программы. Вся соль в том, что x86-процессоры (как и большинство других) на аппаратном уровне отслеживают обращения к нулевым указателям, автоматически выбрасывая исключение, если такое обращение действительно произошло. Поэтому, если проверке на "легитимность" указателя предшествует операция чтения/записи по этому указателю, эту проверку можно не выполнять. Допустим, наш указатель равен нулю, тогда исключение выпрыгнет при первом же обращении к нему (в нашем случае — при выполнении *p = 0x69) и до проверки дело просто не дойдет. Если же указатель не равен нулю — зачем его проверять?

Компиляторы msvc и icl такой техники оптимизации не поддерживают. Правда, в режиме глобальной оптимизации, icl может удалять несколько подряд идущих проверок (при встраивании функций это происходит достаточно часто), поскольку это является частным случаем техники удаления избыточных проверок, однако, ситуацию: "проверка/модификация-указателя/обращение-к-указателю/проверка", icl уже не осилит. А вот gcc справляется с ней без труда!

совмещение проверок

Совмещение проверок очень похоже на повторное использование подвыражений: если она и та же проверка присутствует в двух или более местах и отсутствуют паразитные зависимости по данным, все проверки можно объединить в одну:

```
if (CPU_TYPE == AMD)          // проверка
        x = AMD_f1(y);
else
        x = INTEL_f1(y);
…
if (CPU_TYPE == AMD)          // еще одна проверка
        a = AMD_f2(b);
else
        a = INTEL_f2(b);
```

Листинг 6 не оптимизированный вариант, 2 проверки

```
if (CPU_TYPE == AMD)          // только одна проверка
{
        x = AMD_f1(y);
        a = AMD_f2(b);
}
        else
{
        x = INTEL_f1(y);
        a = INTEL_f2(b);
}
```

Листинг 7 оптимизированный вариант, только одна проверка

Из всех трех компиляторов, совмещать проверки умет только icl, да и то не всегда.

сокращение длины маршрута

Если один условный или безусловный переход указывает на другой безусловный переход, то все три рассматриваемых компилятора автоматически перенаправляют первый целевой адрес на последний, что и демонстрирует следующий пример:

```
        goto lab_1     ; // переход к метке lab_1 на безусловный переход к метке lab_2
….
lab_1:  goto lab_2     ; // переход к метке lab_2
….
lab_2:
```

Листинг 8 не оптимизированный вариант

```
        goto lab_2     ; // сразу переходим к метке lab_2, минуя lab_1
….
lab_1:  goto lab_2     ; // переход к метке lab_2
….
```

```
lab_2:
```

Листинг 9 оптимизированный вариант

Разумеется, оператор goto необязательно должен присутствовать в программном коде в явном виде и он вполне можно быть "растворен" в цикле:

```
while(a)                        // lab_1:   if (a) goto lab_4
{
        while(b)                // lab_2:   if (b) goto lab_3      /* переход на безусловный переход */
        {
                /* код цикла */
        }                       //          goto lab_2
}                               // lab_3:   goto lab_1
                                // lab_4:
```

Листинг 10 не оптимизированный вариант

После оптимизации этот код будет выглядеть так:

```
while(a)                        // lab_1:   if (a) goto lab_4
{
        while(b)                // lab_2:   if (b) goto lab_1      /* оптимизировано */
        {
                /* код цикла */
        }                       //          goto lab_2
}                               // lab_3:   goto lab_1
                                // lab_4:
```

Листинг 11 оптимизированный вариант

Аналогичным способом осуществляется и оптимизация условных/безусловных переходов, указывающих на другой условный переход. Вот, посмотрите:

```
        jmp lab_1               ; // переход на условный переход
        …
lab_1: jnz lab_2
```

Листинг 12 не оптимизированный вариант

```
        jnz lab_2               ; // оптимизировано
        …
lab_1: jnz lab_2
```

Листинг 13 оптимизированный вариант

Заметим, что в силу ограниченной "дальнобойности" условных переходов, в некоторых случаях длину цепочки приходится не только уменьшать, но и увеличивать, прибегая к следующему трюку:

```
jz far_far_away                         jnz next_lab
               ——трансформация———>      jmp far_far_away
                                        next_lab:
```

Листинг 14 трансформация условных переходов, до которых процессор не может "дотянуться"

Условный или безусловный переход, указывающий на выход из функции, всеми тремя компиляторами заменяется на непосредственный выход из функции, при условии, что код эпилог достаточно мал и накладные расходы на его дублирование невелики:

```
f(int a, int b)
{
        while(a--)
        {
                if (a == b) break; // условный переход на return a;
        }
        return a;
}
```

Листинг 15 не оптимизированный вариант

```
f(int a, int b)
{
        while(a--)
        {
                if (a == b) return a; //непосредственный return a;
        }
        return a;
}
```

Листинг 16 оптимизированный вариант

уменьшение кол-ва ветвлений

Все три компилятора просматривают код в поисках условных переходов, перепрыгивающих через безусловные (conditional branches over unconditional branches) и оптимизируют их: инвертируют условный переход, перенацеливая его на адрес безусловного перехода, а сам безусловный переход удаляют, уменьшая тем самым количество ветвлений на единицу:

```
        if (x) a=a*2; else goto lab_1;// двойное ветвление if – else
        b=a+1
lab_1:  c=a*10
```

Листинг 17 не оптимизированный вариант

```
if (!x) goto lab_1;                  // одинарное ветвление
        a=a*2;
        b=a+1;
lab_1: c=a*10;
```

Листинг 18 оптимизированный вариант

Оператор goto необязательно должен присутствовать в тексте явно, он вполне может быть частью do/while/break/continue. Вот, например:

```
while(1)
{
        if (a==0x66) break;    // условный переход
        a=a+rand();
};                             // скрытый безусловный переход на начало цикла
```

Листинг 19 не оптимизированный вариант, 2 ветвления

```
if (a!=0x66)           // "сдирание" одной итерации цикла
do{
        a=a+rand();
}while(a!=0x66);       // инвертируем переход, только одно ветвление
```

Листинг 20 оптимизированный вариант, 1 ветвление (ветвление перед началом цикла не считается, т.к. исполняется всего лишь раз)

сокращение количества сравнений

Процессоры семейства x86 (как и многие другие) обладают одну очень интересной концепцией, которой нет ни в одном языке высокого уровня. Операции вида if (a>b) выполняются в два этапа. Сначала из числа a вычитается число b и состояние вычислительного устройства сохраняется в регистре флагов. Различные комбинации флагов соответствуют различным отношениям чисел и за каждый из них отвечает "свой" условный переход. Например:

```
        cmp eax,ebx    // сравниваем eax с ebx, запоминая результат во флагах
        jl lab_1       // переход, если eax < ebx
        jg lab_2       // переход, если eax > ebx
lab_3:                 // раз мы здесь, eax == ebx
```

Листинг 21 сравнение двух чисел на языке ассемблера

На языках высокого уровня все не так, и операцию сравнения приходится повторять несколько раз подряд, что не способствует ни компактности, ни производительности. Но, может быть, ситуацию исправить оптимизатор? Рассмотрим следующий код:

```
if (a < 0x69) printf("b");
if (a > 0x69) printf("g");
if (a == 0x69) printf("e");
```

Листинг 22 сравнение двух чисел на языке высокого уровня

Дизассемблирование показывает, что компилятору msvc потребовалось целых два сравнения, а вот icl было достаточно и одного. Компилятор gcc не заметил подвоха и честно выполнил все три сравнения.

избавление от ветвлений

Теория утверждает, что любой вычислительный алгоритм можно развернуть в линейную конструкцию, заменив все ветвления математическими операциями (как правило, запутанными и громоздкими).

Рассмотрим функцию поиска максимума среди двух целых чисел: "`((a>b)?a:b)`", для наглядности записанную так: "`if (a<b) a=b;`". Как избавиться от ветвления? В этом нам поможет машинная команда **SBB**, реализующая вычитание с заемом. На языке ассемблера это могло бы выглядеть, например, так:

```
SUB b, a
; отнять от содержимого 'b' значение 'a', записав результат в 'b'
; если a > b, то процессор установит флаг заема в единицу

SBB c, c
; отнять от содержимого 'c' значение 'c' с учетом флага заема,
; записав результат обратно в 'c' ('c' - временная переменная)
; Если a <= b, то флаг заема сброшен, и 'c' будет равно 0,
; Если a > b, то флаг заема установлен и 'c' будет равно -1.

AND c, b
; выполнить битовую операцию (c & b), записав результат в 'c'
; Если a <= b, то флаг заема равен нулю, 'c' равно 0,
; значит, c =(c & b) == 0, в противном случае: c == b - a
;
ADD a, c
; выполнить сложение содержимого 'a' со значением 'c', записав
; результат в 'a'.
; если a <= b, то c = 0 и a = a
; если a > b, то c = b - a, и a = a + (b-a) == b
```

Листинг 23 поиск максимума среди двух целых чисел без использования ветвлений

Компилятор msvc поддерживает замену ветвлений математическими операциями, однако, использует несколько другую технику, отдавая предпочтение инструкции **SETcc**, устанавливающая переменную в единицу, если условие `cc` истинно. Как показывает практика, msvc оптимизирует только ветвления константного типа, т. е. "`if (n > m) a = 66; else a = 99;`" еще оптимизируется, а "`if (n > m) a = x; else a = y;`" уже нет.

Компилятор gcc, использующий инструкцию условного присвоения **CMOVcc**, оптимизирует все конструкции типа min, max, set flags, abs и т. д., что существенно увеличивает производительность, однако, требует как минимум Pentium Pro (инструкция SETcc работает и на Intel 80386). За это отвечают ключи `-fif-conversion` и `-fif-conversion2`, которые на платформе Intel эквивалентны друг другу. (Вообще говоря, gcc поддерживает множество ключей, отвечающих за ликвидацию ветвлений, однако, на платформе Intel они лишены смысла, поскольку в лексиконе x86 процессоров просто нет соответствующих команд, это вам не PDP-11, элегантность которой остается непревзойденной до сих пор и которая имена специальную команду "пропустить следующую машинную инструкцию, если условие cc истинно/ложно", — вот где был простор для избавления от ветвлений!).

Компилятор icl – единственный из всех трех, кто не заменяет ветвления математическими операциями (что в свете активной агитации за команды SBB/CMOVcc, развернутой компаний Intel, выглядит довольно странно). Во всяком случае компилятор не

делает этого явно и в качестве компенсации предлагает использовать инстриксы и функции мультимедийной библиотеки SIMD. В частности, цикл вида:

```
short a[4], b[4], c[4];
for (i=0; i<4; i++)
        c[i] = a[i] > b[i] ? a[i] : b[i];
```

Листинг 24 не оптимизированный вариант с ветвлениями

может быть переписан так:

```
Is16vec4 a, b, c
c = select_gt(a, b, a, b);     // функция векторного поиска максимума без ветвлений
```

Листинг 25 устранение ветвлений путем использования функции select_gt библиотеки классов Intel SIMD

Естественно, такой код непереносим и на других компиляторах он работать не будет, поэтому, прежде чем подсаживаться на Intel следует все взвесить и тщательно обдумать. Тем более, что альтернативные мультимедийные библиотеки имеются и на других компиляторах.

оптимизация switch

Оператор множественного выбора ***switch*** очень популярен среди программистов (особенно, разработчиков Windows-приложений). В некоторых (хотя и редких) случаях, операторы множественного выбора содержат сотни (а то и тысячи) наборов значений, и если решать задачу сравнения "в лоб", время выполнения оператора switch окажется слишком большим, что не лучшим образом скажется на общей производительности программы, поэтому пренебрегать его оптимизацией ни в коем случае нельзя.

балансировка логического древа

Если отвлечься от устоявшейся идиомы "*оператор SWITCH дает специальный способ выбора одного из многих вариантов, который заключается в проверке совпадения значения данного выражения с одной из заданных констант в соответствующем ветвлении*", легко показать, что switch представляет собой завуалированный оператор поиска соответствующего case-значения.

Последовательный перебор всех вариантов, соответствующий тривиальному линейному поиску — занятие порочное и крайне неэффективное. Допустим, наш оператор switch выглядит так:

```
switch (a)
{
        case 98 : /* код обработчика */ break;
        case 4  : /* код обработчика */ break;
        case 3  : /* код обработчика */ break;
        case 9  : /* код обработчика */ break;
        case 22 : /* код обработчика */ break;
        case 0  : /* код обработчика */ break;
        case 11 : /* код обработчика */ break;
        case 666: /* код обработчика */ break;
        case 96 : /* код обработчика */ break;
        case 777: /* код обработчика */ break;
        case 7  : /* код обработчика */ break;
}
```

Листинг 26 не оптимизированный вариант оператора множественно выбора

Тогда соответствующее ему не оптимизированное логическое дерево будет достигать в высоту одиннадцати гнезд (см. рис. 1 слева). Причем, на левой ветке корневого гнезда окажется аж десять других гнезд, а на правой – вообще ни одного. Чтобы исправить "перекос", разрежем одну ветку на две и прицепим образовавшиеся половинки к новому гнезду, содержащему условие, определяющее: в какой из веток следует искать сравниваемую переменную. Например, левая ветка может содержать гнезда с четными значениями, а правая – с нечетными. Но это плохой критерий: четных и нечетных значений редко бывает поровну и вновь образуется перекос. Гораздо надежнее поступить так: берем наименьшее из всех значений и бросаем его в кучу `A`, затем берем наибольшее из всех значений и бросаем его в кучу `B`. Так повторяем до тех пор, пока не рассортируем все, имеющиеся значения. (см. рис. 1 справа)

Поскольку, оператор switch требует уникальности каждого значения, т. е. каждое число может встречаться лишь однажды, легко показать, что: а) в обеих кучах будет содержаться равное количество чисел (в худшем случае – в одной куче окажется на число больше); б) все числа кучи `A` меньше наименьшего из чисел кучи `B`. Следовательно, достаточно выполнить только одно сравнение, чтобы определить: в какой из двух куч следует искать сравниваемое значения.

Рисунок 1 несбалансированное (слева) и сбалансированное (справа) switch/case-дерево

Высота вновь образованного дерева будет равна , где N – количество гнезд старого дерева. Действительно, мы же делим ветвь дерева надвое и добавляем новое гнездо – отсюда и берется и +1, а (N+1) необходимо для округления результата деления в большую сторону. Т. е. если высота не оптимизированного дерева достигала 100 гнезд, то теперь она уменьшилась до 51. Что? Говорите, 51 все равно много? Но кто нам мешает разбить каждую из двух ветвей еще на две? Это уменьшит высоту дерева до 27 гнезд! Аналогично, последующее уплотнение даст 16 → 12 → 11 → 9 → 8… и все! Более плотная упаковка дерева уже невозможна. Но, согласитесь, восемь гнезд – это не сто! Оптимизированный вариант оператора switch в худшем случае потребует лишь пяти сравнений, но и это еще не предел!

Учитывая, что x86 процессоры все три операции сравнения <, =, > совмещают в одной машинной команде, двоичное логическое дерево можно преобразовать в троичное, тогда новых гнезд для его балансировки добавлять не нужно. Простейший алгоритм, называемый ***методом отрезков***, работает так: сортируем все числа по возрастанию и делим получившийся отрезок напополам. Число находящееся посередине (в нашем случае это 11), объявляем вершиной дерева, а числа расположенные слева от него — его левыми ветвями и подветвями (в нашем случае это 0, 3, 4 и 7). Остальные числа (22, 96, 98, 666, 777) идут направо. Повторяем эту операцию рекурсивно до тех пор, пока длина подветвей не сократится до единицы. В конечном счете, вырастет следующее дерево (см. рис. 2):

Рисунок 2 троичное дерево, частично сбалансированное методом отрезков

Очевидно, что это не самое лучше дерево из всех. Максимальное количество сравнений (т. е. количество сравнений в худшем случае) сократилось с пяти до четырех, а количество ветвлений возросло вдове, в результате чего, время выполнения оператора switch только возросло. К тому же, структура построения дерева явна не оптимальна. Гнезда (a<=3), (a>=7), (a<=96), (a>=666) имеют свободные ветви, что увеличивает высоту дерева на единицу. Но, может быть, компилятор сумеет это оптимизировать?

Дизассемблирование показывает, что компилятор msvc генерирует троичное дерево, сбалансированное по улучшенному алгоритму отрезков, содержащее всего лишь 7 операций сравнения, 9 ветвлений и таблицу переходов на 10 элементов (см. "создание таблицы переходов"). В худшем случае, выполнение оператора switch требует 3 сравнений и 3 ветвлений. Троичное дерево построенное компилятором gcc, сбалансировано по классическому алгоритму отрезков и состоит из 11 сравнений и 24 ветвлений. В худшем случае, выполнение оператора switch растягивается на 4 сравнения и 6 ветвлений. Компилятор icl, работающий по принципу простого линейного поиска, строит двоичное дерево из 11 сравнений и 11 ветвлений. В худшем случае, все узлы дерева "пережевываются" целиком. Вот так "оптимизация"!

создание таблицы переходов

Если значения ветвей выбора представляют собой арифметическую прогрессию (см. листинг 27), компилятор может сформировать ***таблицу переходов*** – массив, проиндексированный case-значениями и содержащий указатели на соответствующие им case-обработчики. В этом случае, сколько бы оператор switch ни содержал ветвей, – один или миллион – он выполняется за одну итерацию. Красота!

```
switch (a)
{
        case 1  : /* код обработчика */ break;
        case 2  : /* код обработчика */ break;
        case 3  : /* код обработчика */ break;
```

```
        case 4  : /* код обработчика */ break;
        case 5  : /* код обработчика */ break;
        case 6  : /* код обработчика */ break;
        case 7  : /* код обработчика */ break;
        case 8  : /* код обработчика */ break;
        case 9  : /* код обработчика */ break;
        case 10 : /* код обработчика */ break;
        case 11 : /* код обработчика */ break;
}
```

Листинг 27 не оптимизированный switch, организованный по принципу упорядоченной арифметической прогрессии

```
        cmp     eax, 0Bh                        ; switch 12 cases
        ; сравниваем a с 11

        ja      short loc_80483F5 ; default
        ; если a > 11 выходим из оператора switch
        ;
        jmp     ds:off_804857C[eax*4] ; switch jump
        ; передаем управление соответствующему case-обработчику
        ; таким образом, мы имеем всего лишь одно сравнение и два ветвления

; // таблица смещений case-обработчиков
off_804857C     dd offset loc_80483F5   ; DATA XREF: main+11↑r
        dd offset loc_80483E8   ; jump table for switch statement
        dd offset loc_80483F9
        dd offset loc_8048402
        dd offset loc_804840B
        dd offset loc_8048414
        dd offset loc_804841D
        dd offset loc_8048426
        dd offset loc_804842F
        dd offset loc_8048438
        dd offset loc_8048441
        dd offset loc_804844F
```

Листинг 28 дизассемблерный листинг оптимизированного варианта оператора switch

Создавать таблицы переходов умеют все три рассматриваемых компилятора, даже если элементы прогрессии некоторым образом перемешаны:

```
switch (a)
{
        case 11 : /* код обработчика */ break;
        case 2  : /* код обработчика */ break;
        case 13 : /* код обработчика */ break;
        case 4  : /* код обработчика */ break;
        case 15 : /* код обработчика */ break;
        case 6  : /* код обработчика */ break;
        case 17 : /* код обработчика */ break;
        case 8  : /* код обработчика */ break;
        case 19 : /* код обработчика */ break;
        case 10 : /* код обработчика */ break;
        case 21 : /* код обработчика */ break;
}
```

Листинг 29 не оптимизированный switch, организованный по принципу упорядоченной арифметической прогрессии

Если один или один или несколько элементов прогрессии отсутствуют, соответствующие им значения дополняются фиктивными переходниками к default-обработчику. Таблица переходов от этого, конечно, распухает, однако, на скорости выполнения оператора switch это практически никак не отображается.

Однако, при достижении некоторой пороговой величины "разрежения", таблица переходов внезапно трансформируется в двоичное/троичное дерево. Компилятор msvc единственный из всех трех, способный комбинировать таблицы переходов с логическими деревьями. Разряженные участки, с далеко отстоящими друг от друга значениями, монтируется в виде дерева, а густо населенные области упаковываются в таблицы переходов.

Вернемся к листингу 26. Значения 9, 11 22, 74, 666, 777 упорядочиваются в виде дерева, а 0, 3, 4, 7, 9 ложатся в таблицу переходов, благодаря чему достигается предельно высокая скорость выполнения, далеко опережающая конкурентов.

свободная таблица

компилятор / действие	**Microsoft Visual C++ 6**	**Intel C++ 8.0**	**GCC 3.3.4**
выравнивание переходов	не выравнивает	не выравнивает	выравнивает по границе степени двойки
быстрое булево вычисление	поддерживает	поддерживает	поддерживает
удаление избыточных проверок	не удаляет	удаляет	не удаляет
удаление проверок нулевых указателей	не удаляет	не удаляет	удаляет
совмещение проверок	не совмещает	совмещает	не совмещает
сокращение длины маршрута	сокращает	сокращает	сокращает
уменьшение кол-ва ветвлений	уменьшает	уменьшает	уменьшает
сокращение количества сравнений	сокращает	частично сокращает	не сокращает
избавление от ветвлений	избавляется от ветвлений константного типа	никогда не избавляется	избавляется всегда, когда это возможно
балансировка логического древа	троичное дерево, сбалансированное улучшенным методом отрезков	двоичное, несбалансированное дерево	троичное дерево, сбалансированное методом отрезков
создание таблицы переходов	создает	создает	создает
поддержка разряженной таблицы переходов	поддерживает	поддерживает	поддерживает
совмещение таблицы переходов с деревом	совмещает	не совмещает	не совмещает

заключение

Справедливости ради, оливка ветвь пальмы первенства на этот раз не достанется никому. Все три компилятора показывают впечатляющий результат, но прокалываются в мелочах. Позиция icl выглядит достаточно сильной, однако, на безусловное господство никак не тянет. Как минимум ему предстоит научиться выравнивать переходы, заменять ветвления математическими операциями и переваривать оператор switch.

Компилятор gcc, с учетом его бесплатности, по-прежнему остается наилучшим выбором. Он реализует многие новомодные способы оптимизации ветвлений (в частности, использует инструкцию CMOVcc), однако, по ряду позиций проигрывает насквозь коммерческому msvc, лишний раз подтверждая основной лозунг Microsoft: "Bill always win". При переходе от ветвлений к циклам (а, циклы, как известно, "съедают" до 90% производительности программы), этот разрыв лишь усиливается. Но о циклах в другой раз. Это слишком объемная, хотя и увлекательная тема. Современные компиляторы не только выбрасывают из цикла все ненужное (например, заменяют цикл с предусловием на цикл с пост-условием который на одно ветвление короче) но и трансформируют сам алгоритм, подгоняя порядок обработки данных под особенности архитектуры конкретного микропроцессора.

>>> мини-врезка советы

* избегайте использования глобальных и статических переменных — локальные переменные намного проще оптимизируются;
* не используйте переменные, там где можно использовать константы;
* везде используйте беззнаковые переменные где это только возможно, они намного легче оптимизируются, особенно в тех случаях, когда компилятор пытается избавиться от ветвлений;
* заменяйте int a; if ((a >= 0) && (a < MAX)) на if ((unsigned int)a < MAX), — последняя конструкция на одно ветвление короче;
* ветвление с проверкой на нуль оптимизируется намного проще, чем на любое другое значение;
* конструкции типа x = (flag?sin:cos)(y) не избавляют от ветвлений, но сокращают объем кодирования;
* не пренебрегайте оператором goto – зачастую он позволяет проектировать более компактный и элегантный код;

>>> трансляция коротких условных переходов

Одна из неприятных особенностей x86 процессоров – ограниченная "дальнобойность" команд условного перехода. Разработчики микропроцессора в стремлении добиться высокой компактности кода, отвели на целевой адрес всего один байт, ограничив тем самым длину прыжка интервалом в 255 байт. Это, так называемый, *короткий* (*short*) переход, адресуемый относительным знаковым смещением, отсчитываемым от начала следующий за инструкцией перехода командой (см. рис. 3). Такая схема адресации ограничивает длину прыжка "вперед" (т. е. "вниз") всего 128 байтами, а "назад" (т. е. "вверх") и того меньше – 127! (Прыжок вперед короче потому, что ему требуется "пересечь" и саму команду перехода). Этих ограничений лишен *ближний* (*near*) безусловный переход, адресуемый двумя байтами и действующий в пределах всего сегмента.

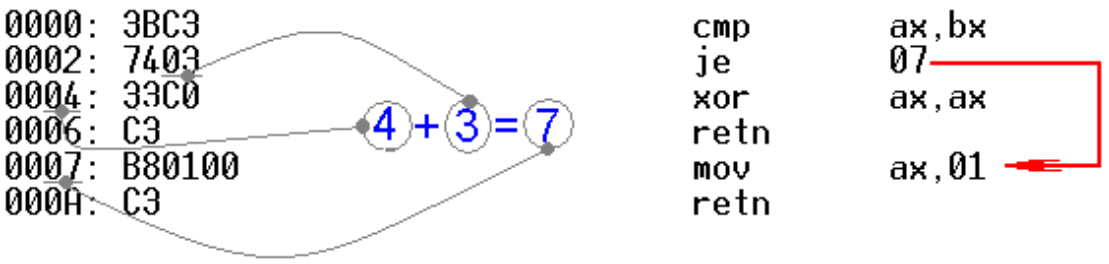

Рисунок 3 внутреннее представление короткого (short) перехода

Короткие переходы усложняют трансляцию ветвлений – ведь не всякий целевой адрес находится в пределах 128 байт! Существует множество путей обойти это ограничение. Наиболее популярен следующий примем: если транслятор видит, что целевой адрес выходит за пределы досягаемости условного перехода, он инвертирует условие срабатывания и совершает короткий (short) переход на метку continue, а на do_it передает управление ближним (near) переходом, действующим в пределах одного сегмента (см. рис. 4)

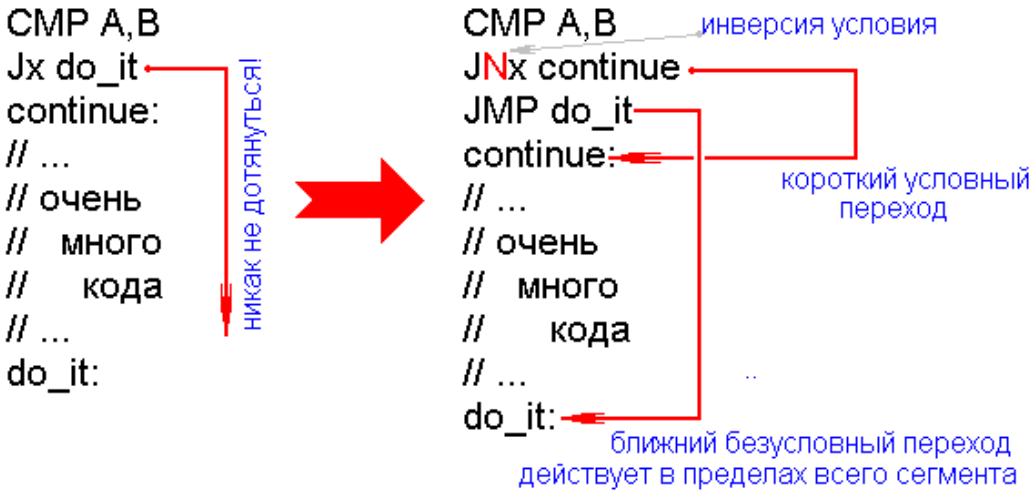

Рисунок 4 трансляция коротких переходов

Начиная с Intel 80386 в лексиконе процессора появилась пятибайтные команды условных переходов, "бьющие" в пределах всего четырех гигабайтного адресного пространства, однако, в силу своей относительно невысокой производительности, так и оставшиеся невостребованными.