техника оптимизации под линуха

крис касперски aka мыщъх, no-email

...вычислительная машина – новый могущественный инструмент. Однако немногие имеют представление об источнике этого могущества.

Дж Вейценбаум Дж "Возможности вычислительных машин и человеческий разум. От суждений к вычислениям"

сегодня мы займемся исследованием двух наиболее популярных Linux-компиляторов: GCC 3.3.4 и Intel C++ 8.0, а конкретно — сравнением мощности их оптимизаторов. для полноты картины в этот список включен Microsoft Visual C++ 6.0 — один из лучших Windows-компиляторов.

(альтернативный вариант вступления на усмотрение редакции)
что находится под черной крышкой оптимизатора? чем один компилятор отличается от другого? правда ли, что Intel C++ намного круче GCC, а сам GCC бьет любой Windows-компилятор? хотите узнать как помочь оптимизатору сгенерировать намного более эффективный код?

общие соображения по оптимизации

Качество оптимизирующих компиляторов обычно оценивают по результатом комплексных тестов (мультимедийных, "общесистемных" или математических). Что именно оптимизируется и как — остается неясным. Основной "интеллект" оптимизаторов сосредоточен в высокоуровневом препроцессоре — своеобразном "ликвидаторе" наиболее очевидных программистских ошибок. Чем качественнее исходный код, тем хуже он поддается оптимизации. Только ведь… над качественным кодом _работать_ надо! Много знать и ожесточенно думать, ломая карандаши или вгрызаясь в клавиатуру. Кому-то это в радость, а кто-то предпочитает писать кое-как. Все равно, мол, компилятор, соптимизирует!

Желание перебросить часть работы на транслятор — вполне естественно и нормально (для творчества больше времени останется), но нужно заранее знать, что именно он оптимизирует, а что только пытается. Но как это можно узнать? На фоне полнейшей терминологической неразберихи, когда одни и те же приемы оптимизации в каждом случае называются по-разному, прячась за ничего не говорящими штаммами типа "copy propagation" (размножение копий) или "redundancy elimination" (устранение избыточности), требуется очень качественная документация на компилятор, но она — увы — обычно ограничивается тупым перечислением оптимизирующих ключей с краткой пометкой за что каждый из них отвечает. Какие копии размножает компилятор и с какой целью? Какую избыточность он устраняет и зачем? Не является ли размножение внесением избыточности, которую самому же оптимизатору и приходится удалять?!

Взять хотя бы документацию на компилятор Intel C++ 7.0/8.0. Будь у меня бумажная версия, я бы ее "употребил". Все равно, ни на что другое она не пригодна. Это просто перечень ключей командной строки, разбавленный словесным мусором, в котором нет никакой конкретики. Скачайте для сравнения документацию на компилятор фирмы Hewlett-Packard (кстати, моя любимая фирма): http://docs.hp.com/en/B6056-96002/B6056-96002.pdf. Доходчивое описание архитектуры процессора, советы по кодированию, тактика и стратегия оптимизирующей трансляции на конкретных примерах. Настоящая библия программиста!

Остальные компиляторы оптимизируют примерно таким же образом, поэтому, эта библия вполне приемлема и для них. "Эффективность оптимизации" из абстрактных цифр превращается серию простых тестов, каждый из которых можно прогнать через транслятор и потрогать руками. Дизассемблирование откомпилированных файлов позволяет однозначно установить — справился ли оптимизатор со своей задачей или нет.

Здесь сравниваются два наиболее популярных Linux-компилятора: GCC 3.3.4 (стабильная версия, проверенная временем, входящая в большинство современных дистрибьютивов), и Intel C++ 8.0 (далее по тексту icl), пропагандируемый как самый

эффективный компилятор всех временен и народов, 30-дневная ознакомительная версия которого лежит на ftp-сервере фирмы. Для полноты картины в этот список включен древний, но все еще используемый Windows-компилятор Microsoft Visual C++ 6.0, для краткости обозначаемый как vc.

Если не оговорено обратное, приведенные примеры должны компилироваться со следующими ключами: -O3 –march=pentium3 (gcc), -O3 –mcpu=pentium4 (icl) и /Ox (vc).

* константы

свертка констант

Вычисление констант на стадии компиляции (оно же, "размножение" или "свертка" констант, constant elimination/folding/propagation или сокращенно CP) — популярный прием оптимизации, избавляющий программиста от необходимости постоянно иметь калькулятор под рукой. Все константные выражения (как целочисленные, так и вещественные) обрабатываются транслятором самостоятельно и в откомпилированный код не попадают.

Рассмотрим следующий пример: `a = 2 * 2; b = 4 * c / 2;` Компиляторы, поддерживающие такую стратегию оптимизации превратят его в: `a = 4; b = 2 * c;`

Улучшенная свертка констант, в англоязычной литературе именуемая "advanced constant folding/propagation" заменяет все константные переменные их непосредственным значением, например: `a = 2; b = 2 * a; c = b - a;` превращается в `c = 2;`

Свертку констант в полной мере поддерживают все три рассматриваемых компилятора: vc, icl и gcc.

объединение констант

Несколько строковых констант с идентичным содержимым для экономии памяти могут быть объедены ("merge") в одну. Тоже самое относится и к вещественным значениям. Целочисленные 32-битные константы объединять невыгодно, поскольку ссылка на константу занимает больше места, чем машинная команда с копией константы внутри.

Покажем технику объединения на следующем примере:

```
printf("hello,world!\n");            // одна строковая константа
printf("hello,world!\n");            // другая константа идентичная первой
printf("i say hello, world!\n");     // константа с идентичной подстрокой
```

Листинг 1 не оптимизированный кандидат в объединение констант

Компилятор vc "честно" генерирует все три строковые константы, а gcc и icl только две из них: `"hello,world!\n"` и `"i say hello, world!\n"`. На то, что первая строка совпадает с концом второй, ни один из компиляторов не обратил внимание. Это тем более печально, что язык Си не позволяет управлять размещением переменных в памяти и классический ассемблерный трюк с передачей указателя на середину строки здесь не проходит. Можно, конечно, поднатужиться и написать:

```
s[]="i say hello, world!\n";         // размещаем строку в памяти
printf(&s[6]);                       // выводим подстроку
printf(&s[6]);                       // выводим подстроку
printf(s);                           // выводим всю строку целиком
```

Листинг 2 частично оптимизированный вариант

Но ведь позицию подстроки в строке придется определять вручную, тупым подсчетом букв! Или использовать макросы, определяя длину строки с помощью sizeof, только… это же сколько кодить надо!

```
#define s1 "i say "                  // левая половинка
#define s2 "hello, world!\n"         // правая половинка
char s[]=s1 s2;                      // конструируем всю строку целиком
printf(&s[sizeof(s1)-1]);            // автоматически вычисляем позицию
printf(&s[sizeof(s1)-1]);            // автоматически вычисляем позицию
printf(s);                           // выводим всю строку
```

Листинг 3 полностью оптимизированный вариант

Причем, в отличии от ситуации с объедением двух полностью тождественных строк (которой можно только похвастаться перед остальными компиляторами), задача объединения подстроки со строкой встречается довольно часто.

константная подстановка в условиях

Переменные, использующиеся для "принятия решения" о ветвлении, в каждой из веток имеют вполне предсказуемые значения, зачастую являющееся константами. Код вида: `if (a == 4) b = 2 * a;` может быть преобразован в: `if (a == 4) b= 8`, что компиляторы vc/gcc и осуществляют, избавляясь от лишней операции умножения, а вот icl этого сделать не догадывается.

константная подстановка в функциях

Если все аргументы функции — константы, и она не имеет никаких побочных эффектов типа модификации глобальных/статических переменных, возвращаемое значение так же будет константой. Однако, компиляторы в подавляющем большинстве случаев об этом не догадываются. Поле зрения оптимизатора ограничено телом функции. "Сквозная" подстановка аргументов ("свертка функций") осуществима лишь в случае встраиваемых (inline) функций или глобального режима оптимизации.

Компилятор icl имеет специальный набор ключей –ip/-ipo, форсирующий глобальную оптимизацию в текущем файле и всех исходных текстах соответственно, что позволяет ему выполнять константную подстановку в следующем коде:

```
f1(int a, int b)
{
        return a+b;
}

f2 ()
{
        return f1(0x69, 0x96) + 0x666;
}
```

Листинг 4 кандидат на константную подстановку

Компилятор gcc достигает аналогичного результата лишь за счет режима агрессивного "инлайнинга", а vc "честно" выполняет вызов функции, до тех пор пока программист не вставит ключевое слово "inline", но никаких гарантий, что vc не проигнорирует это предписание, у нас нет.

** код и переменные*

удаление мертвого кода

"Мертвым кодом" (dead code) называется код, никогда не получающий управления и впустую транжирящий дисковое пространство и оперативную память. В простейшем случае он представляет собой условие, ложность которых очевидна еще на стадии трансляции, или код, расположенный после безусловного возврата из функции.

Вот, например: `if (0) n++; else n--;` Это несложная задача и с нею успешно справляются все три рассматриваемых компилятора.

А вот в следующем случае бесполезность выражения "n++" уже не столь очевидна: `for (a = 0; a < 6; a++) if (a < 0) n++;` Условие (a < 0) всегда ложно, поскольку цикл начинается с 0 и продолжается до 6. Из всех трех рассматриваемых компиляторов это по "зубам" только icl.

удаление неиспользуемых функций

Объявленные, но ни разу не вызванные функции, компилятору не так-то просто удалить. Технология раздельной компиляции (одни файл исходного текста – один объектный модуль) предполагает, что функция, не использующаяся в текущем модуле, вполне может вызываться из остальных. Реальный расклад выявляется лишь на стадии компоновки. Выходит,

неиспользуемые функции должен удалить линкер? Но "выцарапать" мертвую функцию из объектного файла еще сложнее! Для этого линкеру необходимо иметь мощный дизассемблер и нехилый искусственный интеллект в придачу.

Компилятор icl в режиме глобальной оптимизации (ключ –ipo) отслеживает неиспользуемые функции еще на этапе трансляции и в объектный модуль они не попадают. Остальные компиляторы ничем таким похвастаться не могут. Поэтому, категорически не рекомендуется держать весь проект в одном файле (особенно если вы пишите библиотеку). Помещайте в файл только "родственные" функции, которые всегда используются в паре и по отдельности не имеют никакого смысла. Одна функция на объектный файл – это вполне нормально. Две-три еще терпимо, а вот больше — уже перебор. Присмотритесь как устроены стандартные библиотеки языка Си и ваши программы сразу похудеют.

удаление неиспользуемых переменных

Объявленные, но неиспользуемые переменные, удаляются всеми современными компиляторами. Древние оптимизаторы удаляли лишь переменные, к которым не происходило ни одного обращения, сейчас же оптимизатор строит своеобразное "абстрактное дерево" и ветви, ведущие в никуда полностью обрубаются.

В приведенном ниже примере, vc, icl и gcc удаляют все три переменных — a, b и c:

```
main(int n, char **v)
{
        int a,b,c;
        a =n;
        b = a + 1;
        c = 6*b; // переменная c не используется, а значит переменные a и b лишнее
        return n;
}
```

Листинг 5 пример программы с неиспользуемыми переменными

удаление неиспользуемых выражений

Неиспользуемые выражения удаляются всеми тремя рассматриваемыми компиляторами.

Вот например:

```
main(int n, char** v)
{
        int a,b;
        a = n+0x666;            // не используется, перекрывается (2*n)
        b = n-0x999;            // теряется при выходе их функции
        a = 2*n;                // единственное используемое выражение
        return a;

}
```

Листинг 6 пример программы с неиспользуемыми выражениями

Выражение (n+0x666) не используется, перекрываясь следующей операцией присвоения (2*n). Выражение (n-0x999) теряется при выходе из функции. Следовательно, наш код эквивалентен: `return (n - 0x999)`.

Не всегда такая оптимизация проходит безболезненно. Компиляторы "забывают" о том, что некоторые вычисления имеют побочные эффект в виде выброса исключения. Код вида: `a = b/c; a = d`, можно оптимизировать в том, и только в том случае, если переменная с заведомо не равна нулю. Но ни один из трех рассматриваемых компиляторов такой проверки не выполняет! Забавно, но в сокращении арифметических выражений (о которых речь еще впереди), оптимизаторы ведут себя намного более осторожно — никто из них не рискует заменять сокращать выражение `(a/a)` 1 до единицы, даже если переменная a заведомо не равна нулю! Бардак в общем.

удаление лишних обращений к памяти

Компиляторы стремятся размещать переменные в регистрах, избегая "дорогостоящих" операций обращения к памяти. Взять хотя бы такой код: "`i =*p+c; b = *p - d;`". Очевидно, что второе обращение к *p лишнее и компиляторы поступают так: `t =*p; i = t+c; b = t-d`, при этом неявно полагается, что содержимое ячейки *p не изменяется никаким внешним кодом, в противном случае оптимизация будет носить диверсионно-разрушительный характер. Что если переменная используется для обмена данными/синхронизации нескольких потоков? Что если какой-то драйвер возвращает через нее результат своей работы? Наконец, что если мы хотим получить исключение по обращению к странице памяти? Для усмирения оптимизатора во всех этих случаях необходимо объявлять переменную как `volatile` (буквально: "изменчивый", "неуловимый"), тогда при каждом обращении она будет перечитываться из памяти.

Указатели – настоящий бич оптимизации. 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 7 пример с лишними обращениями к памяти, от которых нельзя избавиться

Компилятор не может разместить содержимое *a во временной переменной, поскольку если ячейки *a и *b частично или полностью перекрываются, модификация ячейки *b приводит к неожиданному изменению ячейки *a!

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

Компилятор не может (не имеет права) выносить переменную dst за пределы цикла и обращения к памяти будут происходить на каждой итерации. Чтобы этого избежать, программист должен переписать код 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 9 оптимизированный вариант

удаление копий переменных

Для экономии памяти компиляторы обычно сокращают количество используемых переменных, выполняя алгебраический разворот выражений и удаляя лишние копии. В англоязычной литературе за данной техникой оптимизации закреплен термин "copy propagation", суть которого поясняется 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 10 переменные a и b — лишнее

Очевидно, что его можно переписать как (`2*n+1`), избавившись сразу от двух переменных. Все три рассматриваемых компилятора именно так и поступают. (С технической точки зрения данный примем оптимизации является частным случаем более общего механизма алгебраического упрощения выражений и распределения регистров).

размножение переменных

На процессорах с конвейерной архитектурой удаление "лишних" копий порождает ложную зависимость по данным, приводящую к падению производительности и переменные приходится не только "сворачивать", но и размножать!

Вот например:

```
a = x + y;
b = a + 1;      // b зависит от a
a = i - j;
c = a - 1;      // c зависит от a… точнее от ее второй "копии"
```

Листинг 11 ложная зависимость по данным

Операции (x + y) и (i – j) могут быть выполнены одновременно, но чтобы сохранить результат вычислений, часть процессорных модулей вынуждена простаивать в ожидании пока не освободиться переменна a.

Чтобы устранить эту зависимость код необходимо переписать так:

```
a1 = x + y;
b = a1 + 1;     // b зависит от a1
a2 = i - j;
c = a2 - 1;     // c зависит от a2, но не зависит от a1
```

Листинг 12 размножение переменной a устраняет зависимость по данным

Простейшие зависимости по данным процессоры от Pentium Pro и выше устраняют самостоятельно. Ручное размножение переменных здесь только вредит — количество регистров общего назначения ограничено и компиляторам их катастрофически не хватает. Но это только снаружи. Внутри процессора содержится здоровый регистровый файл, автоматически "расщепляющий" регистры по мере необходимости.

Сложные зависимости по данным на микро-уровне уже неразрешимы и чтобы справится с ними, необходимо иметь доступ к исходному тексту программы. Компилятор icl устраняет большинство зависимостей, остальные же оставляют все как есть.

распределение переменных по регистрам

Регистров общего назначения всего семь, а чаще и того меньше. Регистр EBP используется для организации фреймов (так же называемых стековыми кадрами), регистр EAX по общепринятому соглашению используется для возвращения значения функции. Некоторые команды (строковые операции, умножение/деление) работают с фиксированным набором регистров, который на протяжении всей функции приходился держать "под сукном" или постоянно гонять данные от одного регистра к другому, что так же не добавляет производительности.

Стратегия оптимального распределения переменных по регистрам (global registers allocation) — сложная задача, которую еще предстоит решить. Пусть слово "global" не вводит вас в заблуждение. Эта глобальность сугубо локального масштаба, ограниченная одной-единственной функцией, а то и ее частью.

Компиляторы стремятся помещать в регистры наиболее интенсивно используемые переменные, однако, под "интенсивностью" здесь понимается отнюдь не частота использования, а количество "упоминаний". Но ведь не все "упоминания" равнозначны! Вот, например, `if (++a % 16) b++; else c++;` обращение к переменной c происходит в 16 раз чаще! Статистка обращений не всегда может быть получена путем прямого анализа исходного кода программы, так что ждать помощи со стороны машины — наивно.

Языки Си/Си++ поддерживают специальное ключевое слово "register", управляющее размещением переменных, однако, оно носит характер рекомендации, а не императива и все три

рассматриваемых компилятора его игнорируют, предпочитая интеллекту программиста свой собственный машинный интеллект.

Представляет интерес сравнить распределение переменных по регистрам в глубоко вложенных циклах, поскольку его вклад в общую производительность весьма значителен. Рассмотрим следующий пример:

```
int *a, *b;
main(int n, char **v)
{
        int i,j; int sum=0;
        for (i = 0; i < n; i++)
                for (j = 0; j < n; j++)
                        sum += sum*a[n*i + j] + sum/b[j] + x++;
        return sum+x;
}
```

Листинг 13 глубоко вложенный цикл чувствителен к качеству распределения переменных по регистрам

Компилятору vc регистров общего назначения уже не хватило и три переменных, обрабатываемых внешним циклом, "вылетели" в стек. Компилятору icl "уложился" в 14 (!) стековых переменных, 5 (!) из которых обрабатываются во внутреннем цикле! О какой производительности после этого можно говорить?! Второе место занял gcc – из 10 стековых переменных, 5 расположены во внутреннем цикле. А вы еще Microsoft ругаете…

регистровые ре-ассоциации

Для преодоления катастрофической нехватки регистров, некоторые компиляторы стремятся совмещать счетчик цикла с указателем на обрабатываемые данные. Код вида "`for (i = 0; I < n; i++) n+=a[i];`" превращается ими в "`for (p= a; p < &a[n]; p++) n+=*p;`" Экономия налицо! Впервые (насколько мне известно) эта техника использовалась в компиляторах фирмы Hewlett-Packard, где она фигурировала под термином *register reassociation*. А что же конкуренты?! Возьмем следующий код (кстати, выдранный из документации на HP компилятор):

```
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 14 неоптимизированный кандидат на регистровую ре-ассоциацию

Грамотный оптимизатор должен переписать его так:

```
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 15 оптимизированный вариант — счетчик цикла совмещен с указателем на массив

Эксперимент показывает, что ни vc, ни gcc не выполняют регистровых реассоциаций ни в сложных, ни даже в простейших случаях. С приведенным примером справился один лишь icl, впрочем, это его все равно не спасает и vc распределяет регистры намного лучше.

упрощение выражений

Выполнять алгебраические упрощения оптимизаторы научились лишь недавно, но эффект, как говориться превзошел все ожидания. Редкий программистский код не содержит выражений, которые было бы нельзя сократить. Откройте документацию по MFC на разделе "Changing the Styles of a Window Created by MFC" и поучитесь как нужно писать программы.

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
        // Create a window without min/max buttons or sizable border
        cs.style = WS_OVERLAPPED | WS_SYSMENU | WS_BORDER;

        // Size the window to 1/3 screen size and center it
        cs.cy = ::GetSystemMetrics(SM_CYSCREEN) / 3;
        cs.cx = ::GetSystemMetrics(SM_CXSCREEN) / 3;
        cs.y = ((cs.cy * 3) - cs.cy) / 2;
        cs.x = ((cs.cx * 3) - cs.cx) / 2;

        // Call the base-class version
        return CFrameWnd::PreCreateWindow(cs);
}
```

Листинг 16 это так Microsoft нас учит писать программы

Неудивительно, что Windows так тормозит! Чтобы понять очевидное, парням из Microsoft потребовалось две операции умножения, две — деления и две — сложения. Итого: шесть операций. Проверим, сможет ли оптимизатор избавится от мусорных операций, предварительно переписав код так:

```
struct CS{int x;int y;};
main(int n, char *v)
{
        int x,y; struct CS cs;
        cs.y = n; cs.x = n;
        y = ((cs.y * 3) - cs.y) / 2;
        x = ((cs.x * 3) - cs.x) / 2;
        return y - x;
}
```

Листинг 17 неоптимизированный кандидат на алгебраическое упрощение

Компилятор vc выбрасывает лишь часть операций, но чем он руководствуется при этом — непонятно. Оптимизатор легко раскрывает скобки ((cs.y*3) – cs.y), но дальше этого он не идет, послушно выполняя бессмысленную операцию (cs.y*2 / 2). И тут же, словно одумавшись, принудительно обнуляет регистр EAX, возвращая ноль. Судя по всему, результат выражения вычисляется компилятором еще на стадии трансляции, но он словно не решается им воспользоваться:

```
        mov eax, [esp+arg_0]
        ; загрузка n

        add     eax, eax
        ; n *= 2; ( без учета знака)

        cdq
        ; преобразовать двойное знаковое слово

        sub     eax, edx
        ; учесть знак

        sar     eax, 1
        ; n /= 2;
        xor     eax, eax
        ; n = 0;
```

Листинг 18 загадочный код, сгенерированный компилятором vc

Компилятор icl выбрасывает мусорный код полностью, генерируя честный XOR EAX,EAX, а вот gcc вообще не выполняет никаких упрощений! Однако, могущество icl очень переменчиво. Возьмем такой пример:

```
main(int n, char *v)
{
        int x,y;
        x = n-n; y = n+n;
        return x+y-2*n+(n/n);
}
```

Листинг 19 еще один кандидат на алгебраическое упрощение

Казалось бы, чего в нем сложного? Компиляторы vc и gcc выкидывают все кроме (n/n), оставляя его на тот случай, если переменная n окажется равной нулю. Поразительно, но icl выполняет все вычисления целиком, не производя никаких упрощений.

Таким образом, выполнение алгебраических упрощений — весьма капризная и непредсказуемая операция. Не надейтесь, что компилятор выполнит ее за вас!

упрощение алгоритма

Наибольший прирост производительности дает именно алгоритмическая оптимизация (например, замена пузырьковой сортировки на сортировку вставками). Никакой компилятор с этим справится не в состоянии. Во всяком случае пока. Но первый шаг уже сделан. Современные компиляторы распознают (или во всяком случае пытаются распознать) смысловую нагрузку транслируемого кода и при необходимости заменяют исходный алгоритм другим, намного более эффективным.

Вот, например:

```
main(int n, char **v)
{
        int a = 0;  int b = 0;
        for(i=0; i<n; i++) a++;       // многократное сложение - это умножение
        for(j=0; j<n; j++) b++;
        return a*b;
}
```

Листинг 20 кандидат на упрощение алгоритма

Для человека очевидно, что этот код можно записать так: (n*n). Мой любимый vc именно так и поступает, а вот icl с gcc накручивают циклы на кардан.

использование подвыражений

Хорошие оптимизаторы никогда не вычисляют значение одного и того же выражения дважды. Рассмотрим следующий пример:

```
        a = x*y + n;
        b = x*y - n;   // выражение (x*y) уже встречалось! и x,y с тех пор не менялись!
```

Листинг 21 подвыражение (x*y) – общее

Чтобы избавиться от лишнего умножения этот код необходимо переписать так:

```
        t = x*y;       // вычислить выражение (x*y) и запомнить результат
        a = t + n;     // подстановка уже вычисленного значения
        b = t - n;     // подстановка уже вычисленного значения
```

Листинг 22 оптимизированный вариант

Американцы называют это "удалением избыточности" (redundancy elimination) или "совместным использованием общих выражений" (Share Common Subexpressions). Полное удаление избыточности (оно же Full Redundancy Elimination или сокращенно FRE) предполагает, что совместное использование выражение происходит только в основных путях (path) выполнения программы. Ветвления при этом игнорируются. Частичное удаление избыточности (оно же Partial redundancy elimination или сокращенно PRE), охватывает весь программный код — как внутри ветвлений, так и снаружи. То есть, частичное удаление

избыточности удаляет избыточность намного лучше, чем полное, хотя при полном программа компилируется чуть-чуть быстрее. Вот такая вот терминологическая путаница. Вся заковырка в том, что выражение "Partial redundancy elimination" переводится на русский языке отнюдь не как "частичное удаление избыточности" (хоть это и общепринятый вариант), а "удаление частичной избыточности", а "full redundancy elimination" – "удаление полной избыточности", что совсем не одно и тоже!

Все три рассматриваемых компилятора поддерживают совместное использование выражений. С приведенным примером они справляются легко. Но давайте усложним задачу, предложив им код подсчета суммы соседних элементов массива:

```
/* Sum neighbors of i,j */
up      = a[(i-1)*n + j  ];
down    = a[(i+1)*n + j  ];
left    = a[i*n     + j-1];
right   = a[i*n     + j+1];
sum     = up + down + left + right;
```

Листинг 23 пример с неочевидной разбивкой на подвыражения

Даже для человека не совсем очевидно, что его можно переписать так, сократив количество операций умножения с четырех до одного:

```
inj = i*n + j;                // однократное вычисление подвыражения
up =    val[inj - n];         // избавление от лишнего сложения
down =  val[inj + n];         // избавления от одного сложения и умножения
left =  val[inj - 1];         // избавление от одного сложения и умножения
right = val[inj + 1];         // избавление от одного сложения и умножения
sum = up + down + left + right;
```

Листинг 24 полностью оптимизированный вариант (ручная оптимизация)

Компилятор vc успешно удалил лишнее выражение (i*n), избавившись от одного умножения и сгенерировал довольно туманный и тормознутый код, не оправдывающий возлагаемых на него надежд. Аналогичным образом поступил и gcc. Его основной конкурент — icl хоть и сократил количество умножений наполовину, сгенерировал очень громоздкий код, сводящий на нет весь выигрыш от оптимизации. Короче говоря, с предложенным примером в полной мере не справился никто и для достижения наивысшей производительности программист должен выполнять все преобразования самостоятельно. По крайней мере необходимо добиться, чтобы все совместно используемые выражения в исходном тексте присутствовали в явном виде.

А как обстоят дела с удалением частичной избыточности? Добавим в нашу программу ветвление и перекомпилируем пример:

```
if (n) a = x*y + n; else a = x*y - n;
```

Листинг 25 случай удаления частичной избыточности

Компилятор vc уже не справляется и генерирует две операции умножения, вместо одной. А вот компиляторы icl и gcc поступают правильно, вычисляя выражение (x*y) всего один раз.

сводная таблица качества оптимизации

компилятор / действие	**Microsoft Visual C++ 6**	**Intel C++ 8.0**	**GCC 3.3.4**
свертка констант	выполняет улучшенную свертку	выполняет улучшенную свертку	выполняет улучшенную свертку
объединение констант	никогда не объединяет	объединяет идентичные строковые и вещественные константы	объединяет идентичные строковые и вещественные константы
константная подстановка в условиях	подставляет	не подставляет	подставляет
свертка функций	сворачивает только встраиваемые	с ключом -ipo сворачивает все	сворачивает только встраиваемые
удаление мертвого кода	удаляет только в основной ветке	удаляет во всех ветках	удаляет только в основной ветке
удаление неиспользуемых функций	никогда не удаляет	удаляет с ключом –ipo	никогда не удаляет
удаление неиспользуемых переменных	удаляет с все неявно	удаляет с все неявно	удаляет с все неявно

	неиспользуемые отслеживанием генетических связей	неиспользуемые отслеживанием генетических связей	неиспользуемые отслеживанием генетических связей
удаление неиспользуемых выражений	удаляет	удаляет	удаляет
удаление лишних обращений к памяти	частично	частично	частично
удаление копий переменных	удаляет	удаляет	удаляет
размножение переменных	не размножает	размножает	не размножает
распределение переменных по регистрам	распределяет отлично	распределяет плохо	распределяет средне
реассоциацирует регистры	не реассоцирует	реассоцирует	не реассоцирует
алгебраическое упрощение выражений	в большинстве случаев выполняет упрощение	упрощает простые и некоторые сложные выражения	упрощает простые выражения
упрощение алгоритма	упрощает некоторые операции	никогда не выполняет	никогда не выполняет
использование подвыражений	распознает явные подвыражения только в основной ветке	распознает все явные и частично неявные подвыражения во всех ветках	распознает явные подвыражения во всех ветках

Таблица 1 механизмы оптимизации, поддерживаемые различными компиляторами

заключение

Современные методики оптимизации носят довольно противоречивый характер. С одной стороны, они улучшают код, с другой — страдают непредсказуемыми побочными эффектами. Опытные программисты подобных "вольностей" не одобряют и режимом агрессивной оптимизации пользуются с большой осторожностью. Однако, полностью отказываться от машинной оптимизации даже самые закоренелые консерваторы уже не решаются. Ручное "вылизывание" кода обходится слишком дорого, правда последствия иной оптимизации выходят еще дороже. Наращивая мощь оптимизаторов, разработчики компиляторов допускают все больше ошибок и в ответственных случаях программистам приходится идти на компромисс, поручая оптимизатору только ту часть работы, в результате которой можно быть полностью уверенным (свертка констант, константная подстановка и т. д.)

Собственно говоря, наше исследование компиляторов еще не закончено и перечисленные приемы оптимизации это даже не верхушка айсберга, а небольшой его кусочек. В следующей статье этого цикла мы рассмотрим трансформацию циклов и прочие виды ветвлений. Уверяю вас, это очень интересная тема и здесь есть чему поучиться! (ну дожили… трансляторы учат программистов… [бурчание удаляющегося мыщъх'а]).