ТЕХНИКА ОПТИМИЗАЦИИ ПОД LINUX

…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»

Что находится под черной крышкой оптимизирующего компилятора? Чем один из них отличается от другого? Правда ли, что Intel C++ намного круче GCC, а сам GCC бьет любой Windows-компилятор? Хотите узнать, как помочь оптимизатору сгенерировать более эффективный код? 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.

КРИС КАСПЕРСКИ

Общие соображения по оптимизации

Качество оптимизирующих компиляторов обычно оценивают по результатам 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 естественное и нормальное (для творчества больше времени останется), но при этом 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 штампами 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. Это просто перечень ключей командной строки, разбавленный словесным мусором, в котором нет никакой конкретики. Скачайте для сравнения документацию на компилятор фирмы Hewlett-Packard: 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 в серию простых тестов, каждый из которых можно прогнать через транслятор и потрогать руками. Дизассемблирование откомпилированных файлов позволяет однозначно установить – справился оптимизатор со своей задачей или нет.

В данной статье сравниваются два наиболее популярных Linux-компилятора: GCC 3.3.4 (стабильная версия, проверенная временем, входящая в большинство современных дистрибутивов) и Intel C++ 8.0 (далее по тексту icl), позиционируемый как самый эффективный компилятор всех времен и народов, 30-дневная ознакомительная, а также бесплатная для некоммерческого применения полнофункциональная Linux-версия которого лежит на ftp-сервере Intel: ftp://download.intel.com/software/products/compilers/downloads/l_cc_p_8.0.055.tar.gz. Некоммерческую лицензию можно оформить прямо на сайте компании (без лицензии компилятор работать не будет). Для полноты картины в этот список включен древний, но все еще используемый Windows-компилятор Microsoft Visual C++ 6.0, для краткости обозначаемый, как msvc. Если не оговорено обратное, приведенные примеры должны компилироваться со следующими ключами: -O3 -march= pentium3 (gcc), -O3 -mcpu=pentium4 (icl) и /Ox (msvc). Разница между архитектурами объясняется тем, что GCC 3.3.4 еще не поддерживает режима оптимизации для Pentium 4, а Intel C++8.0 не имеет специального ключа для Pentium III, в результате чего между ними возникает некоторая «нестыковка». Однако на результаты наших экспериментов она никак не влияет, поскольку никакие специфичные для Pentium 4 возможности здесь не используются.

Константы

Свертка констант

Вычисление констант на стадии компиляции (оно же, «размножение» или «свертка» констант, constant elimination/folding/propagation, или сокращенно CP) – популярный при-

ем 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; При этом возникает целый букет побочных эффектов: (4*a/2) не эквивалентно (2*a), потому что при (4*a) может наступить переполнение, а при (a/2) уже может и не наступить! Скажите, что это наигранный пример? А вот и нет! При работе с битовыми масками – это норма. Выражение (k*a/n) часто используется для сброса старших битов переменной с их последующем сдвигом вправо на заданное количество позиций. Но после того как над программой поработает оптимизатор, этот эффект теряется! С вещественной арифметикой еще хуже. Значение double (4*a/2) вполне может и не совпасть с double (2*a) даже безо всякого переполнения! В double начинает играть значение конечная точность. Если (int)4/(int)2 это точно 2, то double(4)/double(2) – это 2.0 плюс-минус что-то очень маленькое. В научном мире это правило часто формулируют так: «результат вычисления с double не может равняться 0» (поэтому все конструкции типа if(double_function(x)==0) надо заменять на if(fabs(double_function(x))<EPS), иначе не работает – результат вычислений будет зависеть от версии компилятора и ключей оптимизации). Если оптимизатор превращает (4*a/2) просто в (2*a) (выкидывает одну операцию), он теряет точность не два раза, а один. В нормально написанной программе (не оперирующей с очень малыми или очень большими числами) это несущественно, т.к. программист изначально проектирует код так, чтобы от этих погрешностей не зависеть. Видимо, разработчики компиляторов на эту категорию людей и ориентируются, а тот, кто умножает $10^{(-308)}$ на $10^{(+307)}$, желая получить именно 0.1, и, таким образом, завязывается на погрешность, – сам себе и виноват!

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: msvc, icl и gcc.

Объединение констант

Несколько строковых констант с идентичным содержимым для экономии памяти могут быть объедены («merge») в одну. То же самое относится и к вещественным значениям. Целочисленные 32-битные константы объединять невыгодно, поскольку ссылка на константу занимает больше места[1], чем машинная команда с копией константы внутри, да и выборка константы из памяти быстродействию, в общем-то, не способствует. Покажем технику объединения на следующем примере:

```
Листинг 1. Не оптимизированный кандидат на объединение
констант

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
```

Компилятор msvc «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 внимания. Приходится напрягаться и писать:

```
Листинг 2. Частично оптимизированный вариант

char 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
```

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!

```
Листинг 3. Полностью оптимизированный вариант

char *s1 = "I say hello,world!\n";
char *s2 = s1+sizeof("I say ")-1; //"hello, world!\n"
printf(s2);
printf(s2);
printf(s1);
```

В отличие от ситуации с объедением двух полностью тождественных строк, практическая ценность которой сомнительна, задача объединения подстроки со строкой встречается довольно часто. Код на листинге 3 предпочтительнее, чем «частично оптимизированный вариант» (листинг 2), поскольку в нем отсутствует двукратное вычисление адресов (&s[6]), которое компилятор может и не догадаться свернуть, а учет смещения на sizeof(«I say »)-1 == 6 байт производится еще на этапе трансляции.

Константная подстановка в условиях

Переменные, использующиеся для «принятия решения» о ветвлении, в каждой из веток имеют вполне предсказуемые значения, зачастую являющиеся константами. Код вида: if (a == 4) b = 2 * a; может быть преобразован в: if (a == 4) b = 8, что компиляторы msvc/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 (и не зависит от их значения), результат выполнения функции также будет константой. Компиляторы в подавляющем большинстве случаев об этом не

[1] В 32-разрядном режиме ссылка занимает 32 бита плюс от одного до шести байтов на поля адресации (один байт «съедает» ModR/M, задающее способ адресации и выбирающее регистры, другой байт приходится на факультативное поле SIB – Scale-Index-Base, используемое для масштабирования, и еще четыре байта отводятся под непосредственное смещение константы в памяти), поэтому ссылки становятся выгодными только начиная с 64/80-битных констант.

догадываются. Поле зрения оптимизатора ограничено телом функции. «Сквозная» подстановка аргументов («свертка функций») осуществляется 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:

Листинг 4. Кандидат на константную подстановку

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

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

Компилятор gcc достигает аналогичного результата лишь за счет того, что на уровне оптимизации -O3 он автоматически встраивает все мелкие функции в тело программы, в то время как msvc встраивает лишь некоторые из них. И даже если функция объявлена как «inline», оптимизатор оставляет за собой право решать: осуществлять ли встраивание в данном случае или 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 (один 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 компоновщику необходимо иметь мощный дизассемблер и нехилый искусственный интеллект впридачу. 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 подобным похвастать не могут. Поэтому категорически не рекомендуется держать весь проект в одном файле (особенно если вы пишите библиотеку), иначе после компоновки ваш код будет переполнен ненужными приложению функциями. 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 msvc, icl и gcc удаляют все три переменные – a, b и c:

Листинг 5. Пример программы с неиспользуемыми переменными

```
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 лишние
    return n;
}
```

Удаление неиспользуемых выражений

Неиспользуемые выражения удаляются всеми тремя рассматриваемыми компиляторами. Например:

Листинг 6. Пример программы с неиспользуемыми выражениями

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

}
```

Выражение (n+0x666) не используется, поскольку перекрывается 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 эффекты 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 сокращать выражение (a/a) 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 переменные различные области памяти или обращаются к одной и той же ячейке памяти. Вот, например:

```
Листинг 7. Пример с лишними обращениями к памяти,
от которых нельзя избавиться

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
}
```

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!

То же самое относится и к следующему примеру:

```
Листинг 8. Пример с лишними обращениями к памяти,
от которых можно избавиться вручную

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

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:

```
Листинг 9. Оптимизированный вариант

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
}
```

Удаление копий переменных

Для экономии памяти компиляторы обычно сокращают количество используемых переменных, выполняя алгебраическое развертывание выражений и удаляя лишние копии. В англоязычной литературе за данной техникой оптимизации закреплен термин «copy propagation», суть которого поясняется в следующем примере:

```
Листинг 10. Переменные a и b – лишние

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)));
}
```

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 прием оптимизации является частным случаем более общего механизма алгебраического упрощения выражений и распределения регистров, который будет рассмотрен ниже.)

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!

Рассмотрим пример:

```
Листинг 11. Ложная зависимость по данным

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»
```

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

Чтобы устранить эту зависимость, код необходимо переписать так:

```
Листинг 12. Размножение переменной a устраняет зависимость
по данным

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
```

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 микроуровне 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 приходится держать «под сукном» или постоянно гонять данные от одного регистра к другому, что также 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 с 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:

Листинг 13. Глубоко вложенный цикл чувствителен к качеству распределения переменных по регистрам

```
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;
}
```

Компилятору msvc регистров общего назначения уже не хватило и три переменных, обрабатываемых внешним циклом, «вылетели» в стек. Компилятор 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…

За счет чего достигается такой выигрыш? Обычно для распределения регистров используются графы, которые в зависимости от интенсивности использования регистров раскрашиваются в различные цвета – от «холодного» до «горячего» (поэтому эта техника часто называется color-map). Однако это в теории. На практике же для достижения приемлемого качества распределения приходится прибегать к некоторому набору эвристических шаблонов, делающих «интуитивные» предположения о реальной «загруженности» той или иной переменной. Судя по всему, компилятор msvc использует большое количество эвристических шаблонов, поэтому с большим отрывом и побеждает всех остальных.

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 была применена в компиляторах фирмы Hewlett-Packard, где она фигурировала под термином register reassociation. А что же конкуренты?! Рассмотрим следующий код (кстати, взятый из документации на компилятор HP):

Листинг 14. Неоптимизированный кандидат на регистровую ре-ассоциацию

```
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;
}
```

Грамотный оптимизатор должен переписать его так:

Листинг 15. Оптимизированный вариант – счетчик цикла совмещен с указателем на массив

```
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;
}
```

Эксперимент показывает, что ни msvc, 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 msvc оказывается впереди за счет более оптимального распределения регистров.

Выражения

Упрощение выражений

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

Листинг 16. Это так Microsoft нас учит 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);
}
```

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 избавиться от мусорных операций, предварительно переписав код так:

Листинг 17. Неоптимизированный кандидат на алгебраическое упрощение

```
struct CS{int x;int y;};
 f(int n,)
{
    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;
}
```

Компилятор msvc 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 константный ноль. Создается устойчивое впечатление, что результат выражения вычисляется компилятором еще на стадии трансляции, но он никак не решается им воспользоваться:

Листинг 18. Загадочный код, сгенерированный компилятором msvc

```
    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;
```

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:

Листинг 19. Еще один кандидат на алгебраическое упрощение

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

Казалось бы, чего в нем сложного? Компиляторы msvc 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 справиться 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:

Листинг 20. Кандидат на упрощение алгоритма

```
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;
}
```

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

Использование подвыражений

Хорошие оптимизаторы никогда не вычисляют значение одного и того же выражения дважды.

Рассмотрим следующий пример:

Листинг 21. Подвыражение (x*y) – общее

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

Чтобы избавиться от лишнего умножения, этот код необходимо переписать так:

Листинг 22. Оптимизированный вариант

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

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 выражений 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 терминологическая путаница.

Вся заковырка в том, что выражение «Partial redundancy elimination» переводится на русский язык отнюдь не как «частичное удаление избыточности» (хоть это и общепринятый вариант), а «удаление частичной избыточности», и «full redundancy elimination» – «удаление полной избыточности», что совсем не одно и то же!

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:

Листинг 23. Пример с неочевидной разбивкой на подвыражения

```
/* 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;
```

Даже человеку не всегда очевидно, что его можно переписать следующим образом, сократив количество операций умножения с четырех до одной:

Листинг 24. Полностью оптимизированный вариант (ручная оптимизация)

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

Компилятор msvc успешно удалил лишнее выражение (i*n), избавившись от одного умножения, и сгенерировал довольно туманный и медленный 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?

Вот, например:

Листинг 25. Случай удаления частичной избыточности

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

Компилятор msvc 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

Таблица 1. Механизмы оптимизации, поддерживаемые различными компиляторами

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 все неявно неиспользуемые отслеживанием генетических связей	Удаляет все неявно неиспользуемые отслеживанием генетических связей	Удаляет все неявно 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
Реассоциирует регистры	Не реассоциирует	Реассоциирует	Не реассоциирует
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

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!