техника оптимизации под LINUX III
оптимизация циклов

крис касперски

большое тестовое сравнение Linux-компиляторов продолжается! тема сегодняшнего исследования — циклы и их оптимизация. в основном мы будем говорить о трех наиболее популярных компиляторах —GCC 3.3.4, Intel C++ 8.0 и Microsoft Visual C++ 6.0, к которым теперь присоединился и GCC 4.0.0 со своим новым оптимизатором циклов

введение

По статистике, до 90% времени исполнения приходится на глубоко вложенные циклы, от качества оптимизации которых зависит быстродействие всей программы в целом. Современные компиляторы поддерживают множество прогрессивных технологий оптимизации и способны буквально на чудеса!

Стратегия оптимизации циклов тесно связана с архитектурой процесса, кэш-контроллера и контроллера оперативной памяти. Это слишком объемная, можно даже сказать, монументальная тема, и в рамках настоящей статьи она не обсуждается. Читайте документацию, распространяемую фирмами Intel и AMD (причем не только по процессорам, но и по чипсету) или "Технику оптимизации программ — эффективная работа с оперативной памяти" Криса Касперски, электронная версия которой свободно распространяется через файлообменные сети. Там эти вопросы описаны достаточно подробно.

выравнивание циклов

Выравнивание циклов (***loop alignment***) имеет много общего с выравниванием ветвлений, о котором мы уже говорили в предыдущей статье. Кратно опишем ситуацию, для тех кто не в курсе. Компиляторы msvc и icl вообще не выравнивают циклы, располагая их в памяти как бог на душу положит. В худшем случае это приводит к трех-четырех кратному падению производительности и порядка 30% быстродействия теряется в среднем.

Компилятор gcc позволяет выравнивать циклы на величину, кратную степени двойки. За это отвечает ключ **`-falign-loops=n`** (по умолчанию n равен двум, что как уже отмечалось, далеко не самая лучшая стратегия выравнивая и предпочтительнее использовать n? равный четырем). Ключ **`-fno-align-loops`** (аналогичный ключу **`-falign-loops=1`**) отключает выравнивание. На уровнях оптимизации –O2 и -O3 выравнивание циклов по умолчанию включено и отключать его было бы неразумно.

* msvc: не выравнивает
* icl: не выравнивает
* gcc: выравнивает по границе степени двойки

разворот циклов

Микропроцессоры с конвейерной архитектурой (а к таковым относятся все современные x86-процессоры), плохо приспособлены для работы с циклами. Для достижения наивысшей производительности процессору необходим достаточно протяженный участок "трассы выполнения", свободный от ветвлений. Компактные циклы вида `for(a=0;a<n;a++) *dst++= *src++;` исполняются очень медленно и для увеличения быстродействия приходится прибегать к их ***развороту*** (***unrolling***).

Под "разворотом" в общем случае понимается многократное дублирование цикла, которое реализуется приблизительно так:

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

Листинг 1 цикл до разворота

```
// разворот цикла на 4 итерации
// выполняем первые n - (n %4) итераций
for(i=1; i<n;i+=4)
{
        k += (n % i) +\
        (n % i+1) + \
        (n % i+2) + \
        (n % i+3);
}

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

Листинг 2 цикл после разворота (меньшей размер, большая скорость)

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

for(i=4; i<n;i++) k += (n % i);
```

Листинг 3 цикл после разворота (меньший размер, меньшая скорость — машинное представление i++ намного более компактно, чем i+const однако, если выражения …(n % i+const_1) + (n % i+const_2) … могут выполняться одновременно, то …(n % i++) + (n % i++)… вычисляются последовательно, поскольку содержат зависимость по данным)

Компилятор msvc вообще не разворачивает циклы и эту работу приходится выполнять вручную.

Компилятор icl разворачивает только циклы for во сценарию больший размер — большая скорость. Причем, циклы с заранее известным количеством итераций — `for(a=0;a<const;a++)` — где const <= 32 трансформируются в линейный код и цикл как таковой при этом исчезает (см "шелушение циклов"). Как следствие — размер программы существенно возрастает, а вместе с ним возрастает и риск "вылететь" за пределы кэша первого уровня, после чего производительность упадет так, что не поднимешь и домкратом. Если const >32, цикл разворачивается на кол-во итераций, кратное const, но не более 10h, при этом компилятор учитывает количество инструкций в теле цикла — более компактные циклы разворачиваются на больше число итераций. Циклы, количество итераций которых компилятору неизвестно — `for(a=0;a<n;a++)` – всегда разворачиваются на 5x, а оставшиеся (n % 5) итераций выполняются в отдельном неразвернутом цикле. Циклы с ветвлениями разворачиваются только при трансформации в линейный код (при этом ветвления, естественно исчезают): `for (a =0; a<3;a++) if (a%2) x[a]++; else x[a]--;` преобразуется в: `x[0]--; x[1]++; x[2]--;`

Компилятор gcc по умолчанию не разворачивает циклы даже на уровне оптимизации -O3, и делает это только с ключом `-funroll-all-loops`, поддерживающим все виды циклов, а не только цикл for. Циклы с известным количеством итераций, где const <= 32 разворачиваются полностью, при const > 32 — на ~4x (точное значение зависит от количества инструкций в теле цикла). Если внутри цикла присутствует одно или несколько ветвлений, при развороте они неизбежно дублируются, в результате чего производительность оказывается даже ниже, чем была до оптимизации! Циклы с неизвестным количеством итераций не разворачиваются вообще. Для тонкой настройки существуют ключи `max-unrolled-insns` (максимальное кол-во инструкций, при котором цикл еще может быть развернут, и если он будет развернут это значение определяет величину разворота), `max-average-unrolled-insns` (максимальное оценочное количество инструкций, которое цикл может иметь после разворота) и `max-unroll-times` (максимальная степень разворота). Разворот по всех случаях выполняется по сценарию больший размер — большая скорость.

* msvc: не разворачивает никакие циклы;
* icl: циклы с переменным кол-вом итераций разворачивает на 5 итераций,
  циклы с cost <= 32 разворачивает полностью:
  циклы с const > 32 разращивает на величину кратную const, но не больше 10h,

учитывает количество инструкций в цикле;
циклы разворачиваются по сценарию: больший размер — большая скорость
* gcc: на уровне O3 по учаолчанию не разворачивает,
циклы с переменным кол-вом итераций никогда не разворачиваются;
циклы с постоянным кол-вом интераций развариваются на ~4x (но тут все зависит от числа инструкций),

шелушение циклов

Идея ***шелушения*** циклов (по-английски "***peeling***") заключается в "сдирании" с цикла одной или нескольких итераций с последующей трансформацией в линейный код. Фактически, шелушение циклов является частным случаем разворота, однако, область его применения одним лишь разворотом не ограничивается.

Рассмотрим следующий код:

```
for(i=0; i<n; i++)
        a[i] = b[i] + 1;

for(j=0; j<n+1; j++)
        c[j] = d[j] - 1;
```

Листинг 4 кандидат на оптимизацию

Чтобы объединить оба цикла в один (см. "объединение циклов"), необходимо "содрать" с цикла j одну "лишнюю" итерацию:

```
for(i = 0; i < n; i++)
{
        a[i] = b[i] + 1;
        c[i] = d[i] - 1;
}
        c[i] = d[i] - 1;
```

Листинг 5 оптимизированный вариант

Компилятор msvc шелушить циклы не умеет, icl и gcc — умеют, но особой радости от этого никто не испытывает, поскольку они никогда не комбинируют "шелушение" с другими приемами оптимизации, что его обеспечивает (а вот компиляторы от SUN или Hewlett-Packard — комбинируют!).

Компилятор gcc содержит специальный ключ **`-fpeel-loops`**, полностью разворачивающий циклы с небольшим количеством итераций. Пороговое значение назначается ключами: `max-peel-times` (сколько итераций можно сдирать с одного цикла), `max-completely-peel-times` (максимальное количество итераций цикла, который еще может быть развернут), `max-completely-peeled-insns` (максимальное количество инструкций, при которых цикл еще может быть развернут) и `max-peeled-insns` (максимальное количество инструкций развернутого цикла) .

* msvc: не шелушит
* gcc: шелушит
* icl: шелушит

фальцевание циклов

Фальцевание циклов (***fold loops***) внешне очень похоже на разворот, но преследует совсем другие цели, а именно: увеличение количества потоков данных на одну итерацию. Чем выше плотность (strength) потоков, тем выше параллелизм обработки данных, а, значит, и скорость выполнения цикла.

Рассмотрим следующий пример:

```
for(i=0; i<XXL; i++)
        sum += a[i];
```

Листинг 6 не оптимизированный вариант – один поток данных на итерацию

Чтобы сократить время выполнения цикла, необходимо увеличить количество потоков, переписав наш код так:

```
for(i=0; i<XXL, I += 4)
{
	sum += a[i];
	sum += a[i+1];
	sum += a[i+2];
	sum += a[i+3];
}

for (i -= 4; i<XXL; i++)
	sum += a[i];
```

Листинг 7 оптимизированный вариант — четыре потока данных на итерацию

Все три рассматриваемых компилятора поддерживают данную стратегию оптимизацию.

* msvc: фальцует
* gcc: фальцует
* icl: фальцует

векторизация

Начиная с Pentium MMX, в x86-процессорах появилась поддержка векторных команд (они же "мультимедийные"). К сожалению, в ANSI Си векторные операции отсутствуют, а нестандартные языковые расширения создают проблему переносимости, поэтому вся забота по векторизации циклов ложится на компилятор. Он должен проанализировать исходный код и определить какие операции могут выполняться параллельно, а какие нет.

Допустим, исходный цикл выглядел так:

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

Листинг 8 цикл до векторизации

Используя общепринятую векторную нотацию, его можно переписать так:

```
a[0:N] = a[0:N] + b[0:N];
```

Листинг 9 тот же цикл, записанный в векторной нотации

Старшие представители процессоров Pentium могут обрабатывать до 8 порций данных параллельно, и если N превышает это число, приходится поступать так:

```
// обрабатываем первые (N-N%VF) ячеек векторным способом
// VF -кол-во порций данных, которые процессор будет обрабатывать за один раз
for (i=0; i<XXL; i+=VF)
	a[i:i+VF] = a[i:i+VF] + b[i:i+VF];

// обрабатываем оставшийся "хвост" обычным способом
for (XXL -= VF ; i < XXL; i++)
	a[i] = a[i] + b[i];
```

Листинг 10 цикл после векторизации

Однако, это методика срабатывает далеко не всегда. Вот пример цикла, который нельзя векторизовать:

```
for (i=1; i<XXL; i++)
	a[i] = a[i-1] + b[i];
```

Листинг 11 цикл который нельзя векторизовать

Поскольку, последующая ячейка (a[i]) вычисляется на основе предыдущей (a[i-1]) данные не могут обрабатываться параллельно.

Компилятор msvc не поддерживает векторизацию, а icl поддерживает, но задействует ее только в том случае, если указан ключ –ax. Компилятор gcc поддерживает векторизацию только начиная с версии 3.4.3, да и то если присутствует флаг -ftree-vectorize.

* msvc: не векторизует
* icl: векторизует
*gcc: векторизует начиная с версии 3.4.3

авто-параллелизм

Многопроцессорные машины на рабочем столе это не утопия, а объективная данность и с каждым годом их количество будет только расти. В операционных системах семейства LINUX многозначность заканчивается на уровне потоков (в старых ядрах — процессах). Всякий поток в каждый момент времени может выполняться только на одном процессоре. Программы, состоящие целиком из одного потока, на многопроцессорной машине исполняются с той же скоростью, что и на однопроцессорной.

Некоторые компиляторы автоматически разбивают большие (с их точки зрения) циклы на несколько циклов меньшего размера, помещая каждый из них в свой поток. Такая техника оптимизации называется авто-параллелизмом (auto-parallelization). Продемонстрируем ее на следующем примере:

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

Листинг 12 нерптимизированный вариант

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

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

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

Листинг 13 оптимизированный вариант

Компилятор icl – единственный из всех трех, способный на "параллелизацию" циклов. Чтобы ее задействовать используйте ключ **`-parallel`**, только помните, что при выполнении кода на однопроцессорных машинах, оптимизация дает обратный эффект (накладные расходы на организацию потоков дают о себе знать).

* msvc: авто-параллелизм не поддерживается
* icl: авто-параллелизм поддерживается
* gcc: авто-параллелизм не поддерживается

программная конвейеризация

Разворот цикла традиционными методами (см. "разворот циклов") порождает зависимость по данным. Вернемся к листингу 3. Смотрите, хотя загрузка обрабатываемых ячеек происходит параллельно (ну… практически параллельно, они будут ползти по конвейеру находясь на различных стадиях готовности), следующая операция сложения не может быть начата до тех пор, пока не будет завершена предыдущая.

Для усиления параллелизма, необходимо суммировать все ячейки в своих переменных, как показано ниже:

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

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

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

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

Листинг 14 оптимизированный вариант

Такая техника оптимизации называется ***программной конвейеризацией*** (***software pipelining***) и из трех рассматриваемых компиляторов ею не обладает ни один. Только gcc робко пытается ослабить зависимость по sum, используя два регистра при развороте на четыре итерации. Остальные же компиляторы грузят все ячейки в один и тот же регистр, очевидно, руководствуясь принципом экономии. Регистров общего назначения всего семь и их постоянно не хватает. К сожалению, компилятор не отличает ситуацию действительно дефицита регистров от их избытка, вынуждая нас прибегать к ручной оптимизации.

* msvc: программная конвейеризация не поддерживается
* icl: программная конвейеризация не поддерживается
* gcc: программная конвейеризация частично поддерживается

предвычисление индуктивных циклов

Цикл называется индуктивным, если его тело целиком состоит из выражения, последующее значение которого вычисляется на основе предыдущего. Легко доказать, что значение индуктивного цикла зависит только от количества итераций и начального значения аргументов выражения, благодаря чему оно может быть вычислено еще на стадии компиляции.

Рассмотрим следующий пример:

```
for (i=0; i<XXL; i++)
        sum++;
```

Листинг 15 индуктивный цикл до оптимизации

Очевидно, что конечное значение sum равно:

```
sum += XXL;
```

Листинг 16 индуктивный цикл после оптимизации

Компилятор msvc всегда пытаются предычислить значение индуктивного цикла, однако, далеко не всегда это ему удается, особенно если индуктивное выражение представляет собой сложную математическую формулу. Два остальных компилятора оставляют индуктивные циклы такими какие они есть, даже не пытаясь их оптимизировать!

* msvc: предвычисляет некоторые индуктивные циклы
* icl: не предвычисляет индуктивные циклы
* gcc: не предвычисляет индуктивные циклы

разбивка длинных цепочек зависимостей

Если индуктивный цикл предычислить не удалось, необходимо по крайней мере ослабить зависимость между итерациями, чтобы они могли исполнятся параллельно. Рассмотрим следующий код:

```
for(i=0;i<XXL;i++)
{
        x += 2;
```

```
        a[i] = x;
}
```

Листинг 17 индуктивный цикл до оптимизации

Выражение (a[i]=x) не может быть выполнено до тех пор, пока не будет вычислено (x+=2), а оно в свою очередь должно дожидаться завершения предыдущей итерации. Индукция, однако! От нее можно избавится, вычисляя значение n'ой итерации на "лету":

```
for(i=0;i<XXL;i++)
{
        a[i] = i*2 + x;
}
```

Листинг 18 индуктивный цикл после оптимизации

Расплатой за отказ от индукции становится появление "лишней" инструкции умножения, однако, накладные расходы на ее выполнение с лихвой окупаются конвейеризацией (а при желании и векторизацией!) цикла. Такая техника оптимизации называется "***разбивка длинных цепочек зависимостей***" (***breaks long dependency chains***) и реализована только в последних версиях gcc (за это отвечает ключ **`-fsplit-ivs-in-unroller`**), а все остальные рассматриваемые компиляторы на это, увы, не способны.

* msvc: не разбивает длинные цепочки зависимостей
* icl: не разбивает длинные цепочки зависимостей
* gcc: разбивает длинные цепочки зависимостей

устранение хвостовой рекурсии

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

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

Листинг 19 хвостовая рекурсия до оптимизации

Вызов функции — достаточно "дорогостоящая" операция и от него лучше избавится. Это легко! Хвостовая рекурсия легко трансформируется в цикл. Смотрите:

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

Листинг 20 хвостовая рекурсия после оптимизации

Компиляторы msvc и gcc всегда разворачивают хвостовую рекурсию в цикл, а вот icl этого делать не умеет.

msvc: устраняет хвостовую рекурсию
icl: не устраняет хвостовую рекурсию
gcc: устраняет хвостовую рекурсию

объединение циклов

Несколько циклов с одинаковым заголовком могут быть объедены в один, сокращая накладны расходы на его организацию. Эта методика имеет множество названий — ***loops fusion***, ***merge loops***, ***jam loops***, создающих большую путаницу и вводящих программистов в заблуждение. В действительности же это не три различных стратегии оптимизации, а всего одна, но какая! Продемонстрирует ее на следующем примере:

```
for(i=0; i<XXL; i++)
        a[i] = b[i] + 1;

for(j=0; j<XXL; j++)
        d[j] = y[j] -1;
```

Листинг 21 циклы до объединения (не оптимизированный вариант)

Поскольку, заголовки обоих циклов абсолютно идентичны, нам достаточно лишь "коллективизировать" их содержимое:

```
for(i = 0; i < XXL; i++)
{
        a[i] = b[i] + 1;
        d[j] = y[j] -1;
}
```

Листинг 22 циклы после объединения (оптимизированный вариант)

А вот более сложный пример:

```
for(i=0; i<XXL; i++)
        a[i] = b[i] + 1;

for(j=0; j<XXL-1; j++)
        d[j] = y[j] -1;
```

Листинг 23 кандидат в оптимизацию путем объединения

Непосредственно объединить циклы невозможно, поскольку цикл j на одну итерацию короче. Чтобы уравнять оба заголовка в правах, предварительно необходимо "содрать" (см. "loop peeling") с цикла i одну итерацию:

```
for(i = 0; i < XXL; i++)
{
        a[i] = b[i] + 1;
        d[i] = y[i] -1;

}       a[i] = b[i] + 1;
```

Листинг 24 оптимизированный вариант

Документация на компилятор icl утверждает, объединение циклов как будто бы поддерживается (см. главу "loop transformation" фирменного руководство), однако, дизассемблирование показывает что это не так. Остальные компиляторы объединение циклов так же не выполняют и на это способен только компилятор от Hewlett-Packard и другие "серьезные" компиляторы для больших машин.

* msvc: не объединяет циклы
* icl: не объединяет циклы
* gcc: не объединяет циклы

трепание циклов

Трепание циклов (***loop spreading***) представляет собой разновидность шелушения, при котором "содранные" итерации упаковываются в самостоятельный цикл, что бывает полезным, например, при объединении двух циклов с "разнополыми" заголовкам.

Рассмотрим следующий код:

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

```
for(j=0; j<m; j++;)
        s[j] = s[j] + s[j+1];
```

Листинг 25 два цикла с различным количеством итераций (не оптимизированный вариант)

Если n не равно m, полное объединение циклов i и j уже невозможно, но min(n,m) итераций мы все же можем объединить:

```
for(i=0; i<min(n,m); i++;)
{
        a[i] = a[i] + c;
        s[i] = s[i] + s[i+1];
}

for(i = min(n,m); i<max(n,m); i++;)
        s[i] = s[i] + s[i+1];
```

Листинг 26 частичное объединение циклов путем "трепки" (оптимизированный вариант)

Ни msvc, ни icl, ни gcc способностью к "трепке" циклов не обладают, однако, это умеют делать, например, компиляторы от Hewlett-Packard.

* msvc: не треплет циклы
* icl: не треплет циклы
* gcc: не треплет циклы

расщепление циклов

Расщепление циклов (***loop distribution, loop fission, loop splitting…***) прямо противоположно их объединению. К такому трюку прибегают в тех случаях, когда оптимизируемый цикл содержит слишком много данных. Ассоциативность кэш-памяти первого уровня у большинства x86-процессоров равна четырем, реже — восьми, а это значит, что каждая обрабатывая ячейка данных может претендовать лишь на одну из четырех (восьми) возможных кэш-линеек и если они к этому времени уже заняты ячейками остальных потоков, происходит их неизбежное вытеснение из кэша, многократно снижающее производительность.

Рассмотрим следующий пример:

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

Листинг 27 не оптимизированный вариант

Чтобы сократить количество потоков данных, следует вынести выражение (c[j] = 0) в отдельный цикл, переписав код так:

```
for(j = 0; j < n; j++)
        c[j] = 0;

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

Листинг 28 оптимизированный вариант

Компилятор icl, кстати говоря, именно так и поступает, снимая это бремя с плеч программиста. Остальные рассматриваемые компиляторы этой способностью, увы, не обладают.

* msvc: не расщепляет циклы
* icl: расщепляет циклы
* gcc: не расщепляет циклы

нормализация циклов

Нормализованным называется цикл, начинающийся с нуля, и в каждой итерации увеличивающий свое значение на единицу, а приведение произвольного цикла к указанной форме называется его ***нормализацией*** (***loop normalization***). Принято считать, что на большинстве архитектур нормализованный цикл компилируется в более компактный и быстродействующий код, однако, в отношении x86-процессоров это совсем не так и более компактным оказывается цикл, стремящийся к нулю (см. "стремление циклов к нулю"). Тем не менее, нормализация бывает полезной, например, при объединении двух циклов с различными заголовками. Если эти циклы предварительно нормализовать, тогда они будут отличаться друг от друга всего лишь числом итераций, а как объединять циклы с несовпадающим количеством итераций мы уже знаем (см. "трепание циклов").

Возьмем произвольный цикл:

```
for (i = lower; i < upper; i+=(-incre))
{
        // тело цикла
}
```

Листинг 29 ненормализованный цикл

В общем случае, стратегия нормализации выглядит так:

```
for (NCL = 0; i < (upper - lower + incre)/incre - 1; 1)
{
        i = incre*NLC + lower;
        // тело цикла
}
i =  incre * _max((upper - lower + incre)/incre, 0) + lower;
```

Листинг 30 нормализованный цикл

Легко показать, что нормализация дает выигрыш только на циклах с заранее известным количеством итераций, позволяющих вычислить значение выражения (upper - lower + incre)/incre еще на стадии компиляции.

Все три рассматриваемых компилятора поддерживают нормализацию циклов (см. раздел loop normalization в документации на icl и описание ключа `-fivcanon` компилятора gcc), но не всегда ею пользуются.

Рассмотрим следующий пример:

```
int i, x[0x10];
for(i=1; i<0x10; i++)
        x[i]=i-1;
```

Листинг 31 ненормализованный цикл

Очевидно, что его можно нормализовать, одним махом избавившись от нескольких "лишних" инструкций (кстати говоря, XOR reg,reg — обнуление регистра reg — намного более компактно, чем MOV reg, const — инициализация регистра константой):

```
int i, x[10]; int *p; p = &x[1];
for(i=0; i<9; i++)
        p[i]=i;
```

Листинг 32 нормализованный цикл

Поразительно, но ни один из трех рассматриваемых компиляторов этого не делает, поэтому все они получают незачет.

* msvc: нормализует некоторые циклы
* icl: нормализует некоторые циклы
* gcc: нормализует некоторые циклы

масштабирование циклов

Масштабированием (***scaling***) в общем случае называется умножение индекса массива на некоторое, как правило, целочисленное значение, например, x = a[4*i]. Идея

масштабирования циклов заключается в выносе множителя в индуктивный инкремент счетчика цикла.

Допустим, исходный цикл выглядел так:

```
for(i=0; i < XXL; i++)
       a[4*i]= b[i];
```

Листинг 33 исходный цикл (неоптимизированный вариант)

После масштабирования он приобретает следующий вид:

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

Листинг 34 цикл после масштабирования (оптимизированный вариант)

В времена XT/AT такая оптимизация еще имела смысл, но начиная с 80386, в процессорах появилась аппаратная поддержка масштабирования на 2x, 4x, 8x и с некоторыми ограничениями на 3x, 5x и 9x, поэтому масштабировать такие циклы уже не нужно.

Масштабирование поддерживают все три рассматриваемых компилятора, но пользуются им довольно неумело, оставляя цикл несмаштабированным даже тогда, когда это ему явно не помешало бы.

* msvc: масштабирует некоторые циклы
* icl: масштабирует некоторые циклы
* icl: масштабирует некоторые циклы

замена циклов с предусловием на циклы с пост-условием

Циклы с предусловием (for, while) содержат по меньшей мере на одно ветвление больше, чем аналогичные им циклы с постусловием. Как нетрудно сообразить — в конце цикла с предусловием находится безусловный переход, возвращающий управление в начало, а в цикле с постусловием передача управления по "совместительству" еще выполняет и проверку условия.

Все три рассматриваемых компилятора всегда заменяют циклы с предусловиям на циклы с постусловием, когда это выгодно.

* msvc: заменяет циклы с предусловием на циклы с постусловием
* icl: заменяет циклы с предусловием на циклы с постусловием
* gcc: заменяет циклы с предусловием на циклы с постусловием

стремление циклов к нулю

На большинстве процессорных архитектур, инструкция декремента (уменьшения регистра на единицу) автоматически взводит специальный флаг при достижении нуля, поэтому, цикл, стремящийся к нулю (incrementing by zero, хотя правильнее было назвать его decrementing by zero) намного выгоден как с точки зрения компактности, так и с точки зрения быстродействия.

Рассмотрим следующий код:

```
for(i=0; i<n; i++) printf("hello!\n");
```

Листинг 35 не оптимизированный вариант

Поскольку, никаких обращений к счетчику цикла здесь нет, его можно развернуть в обратном направлении. Это легко. А вот пример посложнее:

```
for(i=0;i<n; i++) sum+=a[i];
```

Листинг 36 не оптимизированный вариант

В этом случае, за трансформацию цикла приходится расплачиваться усложнением его тела, однако, выигрыш по-прежнему остается на нашей стороне:

```
i=n; do
{
        sum+=*a;
        a++;
} while(--i);
```

Листинг 37 оптимизированный вариант

Компилятор msvc всегда стремится генерировать циклы, стремящиеся к нулю, icl этого не делает вообще, а gcc прибегает к трансформации циклов только в тех в некоторых, наиболее очевидных случаях. В частности, он избегает трогать циклы, содержащие ссылки на память.

* msvc: всегда стремит циклы к нулю
* icl: никогда не стремит циклы к нулю
* gcc: стремит некоторые циклы к нулю

отказ от branch-count-reg

Многие микропроцессоры имеют специальную команду: уменьши-регистр-на-единицу-и-ветвись-если-нуль (branch-count-reg). На x86-процессорах этим занимается LOOP, часто встречающаяся в ассемблерных вставках начинающих программистов, но практически никогда в коде, сгенерированном компилятором. И вовсе не потому, что компилятор "тупой", а потому, что эта инструкция медленная, хотя и компактная.

Компиляторы msvc и icl никогда ее не используют, а gcc предоставляет специальный ключ `-fbranch-count-reg`, предписывающий выбирать LOOP вместо DEC reg/JNZ begin_loop, правда, до машинного когда она все равно не доживает и уничтожается оптимизатором.

* msvc: не использует branch-count-reg
* icl: не использует branch-count-reg
* gcc: не использует branch-count-reg

вынос инвариантных ветвлений

Ветвления, инвариантные по отношению цикла, могут быть вынесены за его пределы, путем расщепления одного цикла на два или более. Размеры кода при этом возрастают, но и производительность увеличивается (конечно, при условии, что цикл влезает в кэш).

Покажем это на следующем примере:

```
for (i = 0; i < n; i++)
{
        for (j = 0; j < n; j++)
        {
                if (flag < 0x669) a[i][j]=i+j; else a[i][j]=0;
        }
}
```

Листинг 38 цикл с инвариантным ветвлением (не оптимизированный вариант)

Поскольку, ветвление if (n < 0x669) инвариантно по отношению к циклу j, мы от него избавляемся:

```
if (flag < 0x669)
        for (i = 0; i < n; i++)
                for (j = 0; j < n; j++)
                        a[i][j]=i+j;
else
        for (i = 0; i < n; i++)
                for (j = 0; j < n; j++)
                        a[i][j]=0;
```

Листинг 39 оптимизированный вариант

На суперкомпьютерных компиляторах такая техника оптимизации используется уже давно и называется ***loop promotion***, ***loop interchange*** или ***loop unswitching***, но на PC она впервые появилась только в последней версии компилятора gcc. Этим заведует ключ **`-funswitch-`**

loops (задействовать вынос инвариантных ветвлений), `max-unswitch-insns` (максимальное количество инструкций, которое может иметь "расщепляемый" цикл) и `max-unswitch-level` (максимальное количество инвариантный ветвлений, которые может иметь "расщепляемый цикл").

Приверженцы остальных компиляторах вынуждены выносить инварианты самостоятельно.

* msvc: не выносит инвариантные ветвления
* icl: не выносит инвариантные ветвления
* gcc: выносит инвариантные ветвления, начиная с версии 3.4.3

ротация ветвлений

Бесконечные циклы с выходом по break могут быть преобразованы в конечные циклы с постусловием. При этом, тело цикла как бы прокручивается, чтобы оператор break переместился на место while(1), а сам while(1) сомкнулся с оператором do и "коллапсировал".

Продемонстрируем это на следующем примере:

```
do
{
        printf("1й оператор цикла\n");
        if (--a<0) break;
        printf("2й оператор цикла\n");
} while(1);
```

Листинг 40 неоптимизированный вариант

После ротации ветвлений наш код будет выглядеть так:

```
// дублируем операторы цикла, расположенные до break
printf("1й оператор цикла\n");
a--;

// а должен ли вообще выполнятся остаток цикла?
if (a>=0)
{
        a++;
        do
        {
                // после трансформации второй оператор стал первым…
                printf("2й оператор\n");

                // …а первый оператор — вторым
                printf("1й оператор цикла\n");
        } while(--a<0); // сюда попало условие из if - break
}
```

Листинг 41 оптимизированный вариант

Из всех трех рассматриваемых компиляторов удалять лишние ветвления умеет лишь один лишь msvc.

* msvc: выполняет ротацию ветвлений
* icl: не выполняет ротацию ветвлений
* gcc: не выполняет ротацию ветвлений

упорядочивание обращений к памяти

При обращении к одной-единственной ячейки памяти, в кэш первого уровня загружается целая строка, длина которой в зависимости от типа процессора варьируется от 32 до 128 или даже 256 байт, поэтому большие массивы выгоднее всего обрабатывать по строкам, а не по столбцам. К сожалению, многие программисты об этом забывают и отдуваться приходится компилятору:

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

```
                a[i][j] = b[i][j] + c[i][j];
```

Листинг 42 обработка массивов по столбцам (не оптимизированный вариант)

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

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

Листинг 43 обработка массивов по столбцам (оптимизированный вариант)

Все три рассматриваемых компилятора поддерживают данную стратегию оптимизации, однако их интеллектуальные способности очень ограничены и со следующим примером не справился ни один.

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

Листинг 44 сложный случай обработки данных по столбцам

А вот компиляторы от Hewlett-Packard оптимизируют этот цикл так (хвост цикла для упрощения понимания не показан):

```
for(i=0; i<n; i+=4)
{
        for (j=0; j<n; j++)
        {
                for (k=0; k<n; k++)
                {
                        a[j][i] = a[j][i] + b[k][i] * c[j][k];
                        a[j][i+1] = a[j][i+1] + b[k][i+1] * c[j][k];
                        a[j][i+2] = a[j][i+2] + b[k][i+2] * c[j][k];
                        a[j][i+3] = a[j][i+3] + b[k][i+3] * c[j][k];
                }
        }
}
```

Листинг 45 оптимизированный вариант с разворотом на 4 итерации (обратите внимание, какой цикл был развернут!)

Оптимизатор скомбинировал сразу три методики: разворот, объединение и переупорядочивание циклов, благодаря чему скорость выполнения значительно возросла. К сожалению, еще долгое время PC-программистам придется оптимизировать свои программы самостоятельно, хотя стремительное совершенствование gcc дает робкую надежду на изменение ситуации.

* msvc: частично упорядочивает обращения к памяти
* icl: частично упорядочивает обращения к памяти
* gcc: частично упорядочивает обращения к памяти

свободная таблица качества оптимизации

компилятор / действие	**Microsoft Visual C++ 6**	**Intel C++ 8.0**	**GCC 3.3.4**
выравнивание циклов	не выравнивает	не выравнивает	выравнивает по границе степени двойки
разворот циклов	не разворачивает	разворачивает циклы без ветвлений с переменным и постоянным кол-вом итераций	разворачивает циклы с постоянным кол-вом итераций
шелушение циклов	не шелушит	шелушит	шелушит

векторизация циклов	не векторизует	векторизует	векторизует начиная с версии 3.4.3
авто-параллелизм	не поддерживает	поддерживает	не поддерживает
программная конвейеризация	не поддерживает	не поддерживает	частично поддерживает
предвычисление индуктивных циклов	предвычисляет простые циклы	не предвычисляет	не предвычисляет
разбивка длинных цепочек зависимостей	не разбивает	не разбивает	разбивает, начиная с версии 4.0.0
устранение хвостовой рекурсии	устраняет	не устраняет	устраняет
объединение циклов	не объединяет	не объединяет	не объединяет
трепание циклов	не поддерживает	не поддерживает	не поддерживает
расщепление циклов	не расщепляет	расщепляет	не расщепляет
нормализация циклов	нормализует некоторые циклы	нормализует некоторые циклы	нормализует некоторые циклы
масштабирование циклов	масштабирует некоторые циклы	масштабирует некоторые циклы	масштабирует некоторые циклы
замена циклов с предусловием на циклы с постусловием	заменяет	заменяет	заменяет
стремление циклов к нулю	всегда стремит циклы к нулю	никогда не стремит циклы к нулю	стремит некоторые циклы к нулю
branch-count-reg	не использует	не использует	не использует
вынос инвариантных ветвлений	не выносит	не выносит	выносит, начиная с версии 3.4.3
ротация ветвлений	выполняет	не выполняет	не выполняет
упорядочивание обращение к памяти	частично упорядочивает обращения к памяти	частично упорядочивает обращения к памяти	частично упорядочивает обращения к памяти

заключение

Прогресс не стоит на месте и со времени появления msvc компиляторы сделали большой шаг вперед. Особенно порадовал gcc 4.0.0 с его новым оптимизатором циклов, который намного превосходит icl и msvc всех вместе взятых. Тем не менее, до совершенства ему еще далеко. Некоторые техники, присутствующие еще в msvc, в нем не реализованы (например, ротация ветвлений), не говоря уже про "серьезные" компиляторы от Hewlett-Packard. Так что на компилятор надейся, а сам не плошай!