война миров: ассемблер против си

крис касперски ака мыщхъ, no-email

действительно ли эффективны (неэффективны) си-компиляторы и насколько большой выигрыш можно получить, переписав программу на языке ассемблера. какую цену за это придется заплатить? об этих и многих других вопросах, волнующих хакерскую общественность уже несколько лет, мыщъх постарался дать предельно объективный и непредвзятый ответ.

введение

Любовь хакеров к ассемблеру вполне понятна и объяснима. Разве не заманчиво знать язык, которым владеют немногие? Ассемблер окружен мистическим ареалом — это символ причастности к хакерским кругам, своеобразный пропуск в клан системных программистов, вирусописателей и взломщиков. Ассемблер теснее всех других языков приближен к железу и ниже его находятся только машинные коды, уже вышедшие из употребления, а жаль!

Программисты каменного века с презрением относились к ассемблеру, поскольку для тех времен он был слишком высокоуровневым языком, абстрагирующимся от целого ряда архитектурных особенностей. Программируя на ассемблере, можно не знать о порядке следования байт в слове, о системе кодирования машинных инструкций, ассемблер скрывает тот факт, что команда "`ADD AL, 6h`" может быть закодирована и как 04h 06h и как 80h C0h 06h, хуже того! Ассемблер не предоставляет никаких средств _выбора_ между этими вариантами! Хорошие трансляторы автоматически выбирают наиболее короткий вариант, но никаких гарантий, что они это сделают, у нас нет, а в самомодифицирующимся коде это весьма актуально! Да что там самомодифицирующийся код (или код, использующий коды мнемоник как константы) — на ассемблере невозможна эффективная реализация выравнивания команд! Тупая директива `align`, вставляющая `NOP`'ы не в счет. В частности, "`ADD AL, 6h`", закодированная как "80h C0h 06h"? намного эффективнее, чем 04h 06h + 90h (`NOP`).

Кто-то наверняка скажет: ассемблер позволяет закодировать любую команду через директиву `DB`, следовательно, на нем можно делать _все_ Весьма спорное утверждение. Язык си так же позволяет объявлять массивы вида `unsigned char buf[] = "\x04\x06\x90"` и умеет преобразовывать указатели на данные в указатели на функции. Рассуждая по аналогии, можно сказать: на языке си легко сделать тоже самое, что и на ассемблере, даже не используя ассемблерных вставок (которые на самом деле не часть языка, а самостоятельное расширение). Но вряд ли программу, полностью состоящую из "`\x04\x06\x90`", можно назвать программой на языке си. Точно так и с ассемблером. Это вовсе не язык неограниченных возможностей, каким его иногда представляют. Ассемблер — всего лишь средство выражения программисткой мысли, рабочий инструмент, а выбор инструмента всегда должен быть адекватен. Не стоит рыть траншею лопатой, если под рукой есть экскаватор, но и строить собачью конуру с помощью крана и бетонных боков — не верх инженерной культуры, а признак ее отсутствия.

Считается, что программа, написанная на ассемблере, по определению компактнее и производительнее аналогичной программы, написанной на языке высокого уровня. Действительно, человек _всегда_ в состоянии обогнать даже самый совершенный компилятор, потому что компилятор действует по строго заданному шаблону (ну, хорошо, нескольким шаблонам), а человек способен на _принципиально_ новые решения. Однако, ассемблерные программы, написанные начинающими программистами, как правило, _значительно_ хуже кода, сгенерированного компилятором. Распределение переменных по регистрам, устранение зависимостей по данным, переупорядочивание инструкций с целью предотвращения простоев конвейера — это слишком нудная работа, отнимающая кучу сил и времени, и хотя человек _потенциально_ способен добиться _намного_ лучшего распределения, чем компилятор, этот разрыв не настолько велик и с коммерческой точки зрения ничем не окупается.

Ассемблерная программа, оптимизированная вручную, становится совершенно немобильной. Если потребуется внести в код даже незначительные изменения или выйдет процессор с новыми правилами оптимизации — всю работу придется начинать заново. А на языке высокого уровня — просто перекомпилировал и все! Большинство программных комплексов, написанных на Си/Си++, никогда бы не увидели свет, если бы в качестве основного языка разработки был выбран ассемблер, требующий неимоверной концентрации труда. Механизация на то и придумана, чтобы облегчать человеку жизнь и воплощать грандиозные

замыслы. Никто же не спорит, что на дачном участке, ручной уход за растениями дает намного больший урожай, чем тракторист на колхозном поле, но обработать колхозное поле вручную практически невозможно!

Несколько десятилетий тому назад, когда счет памяти шел на каждый килобайт и приходилось экономить каждый такт, ручная оптимизация еще имела смысл, поскольку откомпилированные программы на массовом железе тормозили со страшной скоростью, но сейчас все изменилось. Системные требования уже давно перестали быть основным потребительским фактором. Теперь в ассемблере нуждаются лишь критичные к быстродействию модули, связанные с обработкой огромного количества данных в реальном времени. Во всех остальных случаях, лучше воспользоваться качественным оптимизирующим компилятором, например, Microsoft Visual C++, GCC 2.95 или другим.

Более новые версии компиляторов в основном пекутся о качестве поддержки очередной редакции Си++ стандарта, оставляя оптимизирующий движок без изменений, поскольку новых идей ни у кого нет. Единственное исключение составляет Intel Fortran/C++, реализующий режим глобальной оптимизации (остальные компиляторы оптимизируют код только в пределах одной функции) и проложивший мост между профилировщиком и компилятором. Используя данные профилировки, компилятор, в частности, может размещать в регистрах наиболее интенсивно используемые переменные, а остальные — гнать в память. К сожалению, эта методика далека от совершенства и хотя Intel C++ "официально" обгоняет GCC, с этим согласны далеко не все и поклонники GCC демонстрируют множество программ, на которых Intel C++ значительно отстает от GCC, но это уже тема совсем дискуссии…

эффективность кодогенерации си компиляторов

Желая продемонстрировать превосходство ассемблера над си, обычно берут программы типа "hello, world!" и сравнивают размеры _откомпилированных_ файлов, причем, ассемблер использует прямые вызовы API-функций `GetStdHandle()/WriteFile()`, а программу на си заставляют обращаться к `printf()`, которая тащит за собой библиотеку времени исполнения (она же Run Time Library или, сокращенно, RTL). Ну и где же здесь честность?! Как будто на си нельзя программировать без RTL! Можно — для этого достаточно переименовать функцию `main()` во что-то другое, указав линкеру точку входа в файл вручную. Размер откомпилированного файла сразу же сократится, вплотную приближаясь (или даже совпадая) с ассемблированным файлом, но 99% пространства будет занимать служебная информация PE-формата и место, оставленное линкером для выравнивания секций. На этом фоне различия между компилятором и ассемблером становятся совершенно незаметными!

Мы же поступим иначе — напишем небольшую программку, например, вычисляющую CRC8 (большая — просто бы не уместилась в статью), а затем откомпилируем ее и, прогнав полученный файл через дизассемблер, посмотрим — насколько эффективно компилятор справился со своей задачей и какой простор он оставил нам для ручной оптимизации.

Исходный текст подопытной функции выглядит так:

```
CRC(unsigned char *p, int n)
{
        int a; unsigned char crc=0;
        for (a=0;a<n;a++) crc += p[a];
        return 0 - crc;

}
```

Листинг 1 ключевой фрагмент демонстрационной программы CRC.c

Компилируем ее Microsoft Visual C++ в режиме максимальной оптимизации (ключ Ox — "`cl.exe /Ox crc.c`") и загружаем полученный obj в дизассемблер:

```
.text:00000000          _CRC    proc near
.text:00000000
.text:00000000          var_1   = dword ptr -1
.text:00000000          arg_0   = dword ptr  7
.text:00000000          arg_4   = dword ptr  0Bh
.text:00000000
.text:00000000 51               push    ecx
.text:00000001 8B 54 24 0C      mov     edx, [esp+1+arg_4]
.text:00000005 32 C9            xor     cl, cl
.text:00000007 33 C0            xor     eax, eax
.text:00000009 88 4C 24 00      mov     byte ptr [esp+1+var_1], cl
```

```
.text:0000000D 85 D2          test    edx, edx
.text:0000000F 7E 16          jle     short loc_27
.text:00000011 53             push    ebx
.text:00000012 56             push    esi
.text:00000013 8B 74 24 10    mov     esi, [esp+9+arg_0]
.text:00000017
.text:00000017 loc_17:
.text:00000017 8A 1C 30       mov     bl, [eax+esi]
.text:0000001A 02 CB          add     cl, bl
.text:0000001C 40             inc     eax
.text:0000001D 3B C2          cmp     eax, edx
.text:0000001F 7C F6          jl      short loc_17
.text:00000021 5E             pop     esi
.text:00000022 88 4C 24 04    mov     byte ptr [esp+5+var_1], cl
.text:00000026 5B             pop     ebx
.text:00000027
.text:00000027 loc_27:
.text:00000027 8B 44 24 00    mov     eax, [esp+1+var_1]
.text:0000002B 25 FF 00 00+   and     eax, 0FFh
.text:00000030 F7 D8          neg     eax
.text:00000032 59             pop     ecx
.text:00000033 C3             retn
.text:00000033 _CRC    endp
```

Листинг 2 результат трансляции crc() компилятором MS VC++ 6.0

Сразу бросается в глаза, что компилятору не хватило регистров (!) и счетчик контрольной суммы зачем-то задвинулся в локальную переменную. Впрочем, это не сильно сказалось на производительности, поскольку задвижение произошло _после_ выхода из цикла, но сам факт! А ведь, чтобы исправить ситуацию, всего-то и требовалось заменить "`MOV byte ptr [ESP+5+var_1],CL/MOV EAX,[ESP+1+var_1]/AND EAX, 0FFh`" на "`MOVZX EAX,CL`", что _гораздо_ короче и _намного_ быстрее, особенно, если n невелико и функция вызывается большое количество раз.

Следующее замечание: компилятору потребовалось целых 5 (!) регистров, в то время как для решения данной задачи вполне достаточно 3х: один — на сумматор CRC, один — на указатель и еще один — на адрес конца.

Ассемблер пример, с ручной оптимизацией приведен ниже:

```
00000000: 51                push    ecx
00000001: 8B4C240C          mov     ecx,[esp+arg_p]
00000005: 8B542408          mov     edx,[esp+arg_n]
00000009: 03CA              add     ecx,edx
0000000B: 33C0              xor     eax,eax
0000000D: EB03              jmps    000000012
0000000F: 0201              add     al,[ecx]
00000011: 41                inc     ecx
00000012: 3BCA              cmp     ecx,edx
00000014: 72F9              jb      00000000F
00000016: 59                pop     ecx
00000017: C3                retn
```

Листинг 3 ручная ассемблерная реализация crc()

18h ассемблерных байт (и 12 команд) против 34h откомпилированных байт (и 23 команд) — для классического цикла это хороший результат. Что же тогда говорить о нетривиальных вещах: сложных структурах данных, циклах в высоким уровнем вложенности, многомерных массивах и т. д.? С другой стороны, не все так плохо! Компилятор успешно распознал конструкцию "`return 0 - crc`" и вместо тупого вычитания из нуля, подобрал адекватную машинную команду `NEG`, означающую "дополнение до нуля".

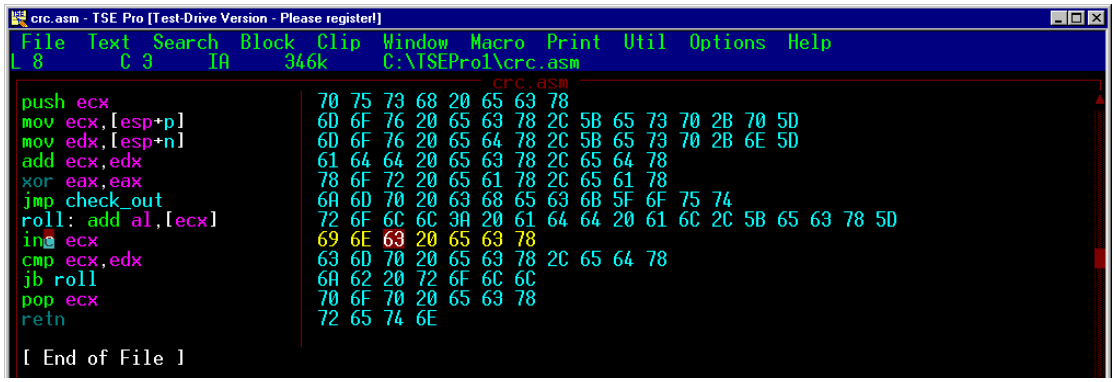

Рисунок 1 набор ассемблерной программы в редакторе TSE Pro

А что на счет GCC? Для начала возьмем древнюю, но все еще широко используемую версию 2.95, отличающуюся стабильностью и высокой скоростью трансляции. На самом высоком уровне оптимизации (ключ -O3: "`gcc crc.c -O3 -o crc`"), компилятор генерирует следующий код:

```
.text:080483E0 CRC             proc near
.text:080483E0
.text:080483E0         arg_0   = dword ptr  8
.text:080483E0         arg_4   = dword ptr  0Ch
.text:080483E0
.text:080483E0 55              push    ebp
.text:080483E1 31 D2           xor     edx, edx
.text:080483E3 89 E5           mov     ebp, esp
.text:080483E5 53              push    ebx
.text:080483E6 8B 4D 0C        mov     ecx, [ebp+arg_4]
.text:080483E9 31 C0           xor     eax, eax
.text:080483EB 8B 5D 08        mov     ebx, [ebp+arg_0]
.text:080483EE 39 CA           cmp     edx, ecx
.text:080483F0 7D 16           jge     short loc_8048408
.text:080483F2 8D B4 26 00+    lea     esi, [esi+0]
.text:080483F9 8D BC 27 00+    lea     edi, [edi+0]
.text:08048400
.text:08048400 loc_8048400:
.text:08048400 02 04 1A        add     al, [edx+ebx]
.text:08048403 42              inc     edx
.text:08048404 39 CA           cmp     edx, ecx
.text:08048406 7C F8           jl      short loc_8048400
.text:08048408
.text:08048408 loc_8048408:
.text:08048408 5B              pop     ebx
.text:08048409 0F B6 C0        movzx   eax, al
.text:0804840C F7 D8           neg     eax
.text:0804840E 5D              pop     ebp
.text:0804840F C3              retn
.text:0804840F         CRC     endp
```

Листинг 4 результат трансляции crc() компилятором GCC 2.95

В отличии от MS VC, компилятору GCC хватило всего 4х регистров без обращения к локальным переменным, а сам код уложился в 30h байт, что на 4 байта короче, чем у конкурента, но до ручной ассемблерной оптимизации еще все равно далеко. Однако, если присмотреться к телу цикла повнимательнее, можно обнаружить, что GCC, в отличие от MS VC, совместил счетчик цикла с инкрементом указателя, то есть, откомпилированный цикл исполняется практически с той же самой степенью эффективности, что и ручной. "Практически" — потому, что в отличии от нас, компилятор использовал сложную адресацию "`ADD AL, [EDX+EBX]`", напрягающую процессор и требующую нескольких экстра тактов на декодирование (впрочем, к последним версиям P-4 и Athlon это уже не относится).

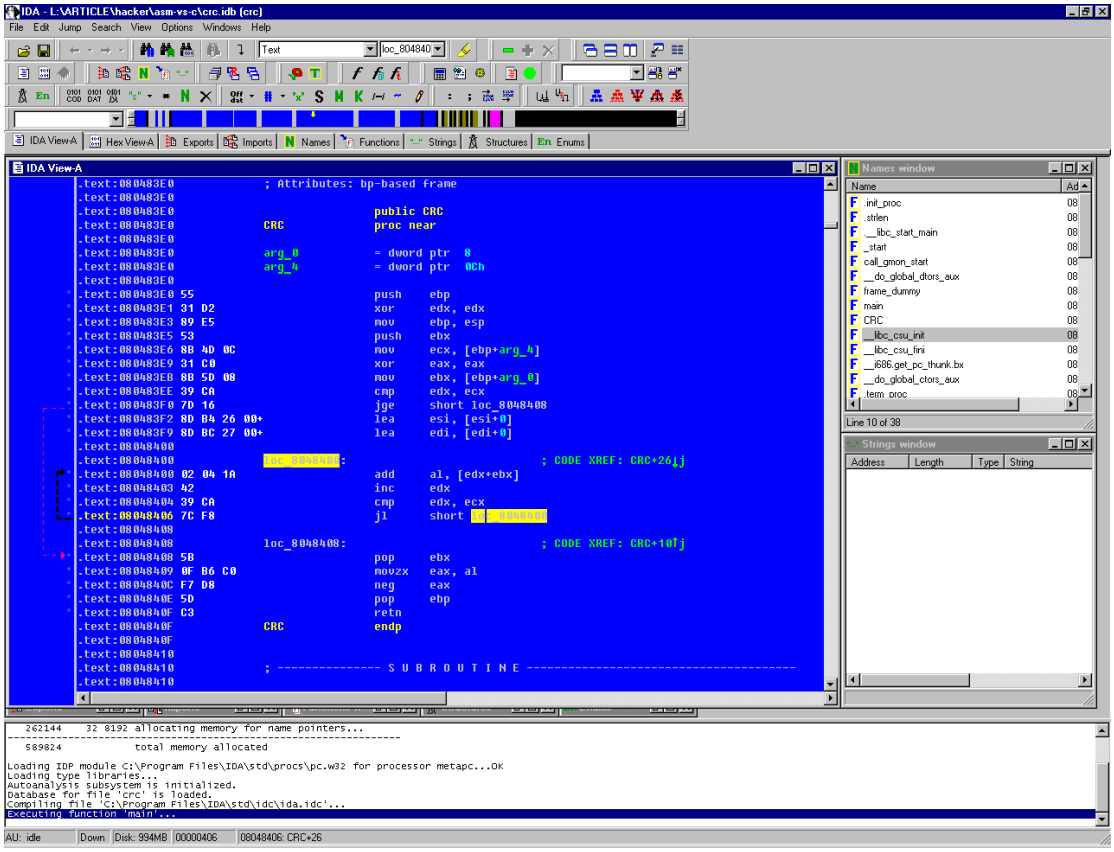

Рисунок 2 IDA Pro за работой

Причем, цикл выровнен по адресам, кратным 10h (команды "`LEA ESI,[ESI+0]`" и "`LEA EDI,[EDI+0]`" используются для выравнивания), что с одной стороны хорошо, а с другой — не очень. Начиная с процессоров поколения P6 (к которым, в частности, принадлежит Pentium Pro, Penium-II) и AMD K5, выравнивание циклов требуется только тогда, когда целевой переход попадает на команду, расщепленную двумя линейками кэш-памяти первого уровня, длина которых в зависимости от процессора составляет 32, 64 или даже 256 байт. В противном случае наблюдается существенное снижение быстродействия.

Компилятор MS VC вообще не выравнивает циклы, поэтому их быстродействие зависит от воли случая — попадет ли первая инструкция цикла на "расщепляющий" адрес или… не попадет. Компилятор GCC выравнивает циклы, но по слишком ретивой стратегии, всегда подтягивая их к адресам, кратным 10h. Это хорошо работает на первопнях, но вот на более современных процессорах команды, расходующиеся на выравнивание, занимают лишнее место и впустую съедают производительность (особенно на вложенных циклах, где они выполняются многократно).

Зато, в отличии от своего конкурента, GCC "догадался" использовать команду "`MOVZX EAX, AL`", только вот _зачем_ она ему понадобилась — не понятно. Команда "`MOVZX`" пересылает значение из источника в приемник, дополняя его нулями до 16- или 32-бит (в зависимости от разрядности). Но ведь в нашем случае старшие биты регистра EAX _уже_ равны нулю, поскольку компилятор _сам_ обнулил их инструкцией "`XOR EAX,EAX`". Следовательно, команда "`MOVZX EAX,AL`" совершенно не нужна и со всей своей очевидностью избыточна.

Интересно, изменилось ли что-нибудь в новых версиях? Компилируем программу с помощью GCC 3.4.2 и смотрим полученный результат:

```
.text:080484C0          CRC     proc near
.text:080484C0
.text:080484C0          arg_0   = dword ptr  8
.text:080484C0          arg_4   = dword ptr  0Ch
.text:080484C0
.text:080484C0 55               push    ebp
.text:080484C1 89 E5            mov     ebp, esp
.text:080484C3 8B 4D 0C         mov     ecx, [ebp + arg_4]
```

```
.text:080484C6 53             push    ebx
.text:080484C7 31 C0          xor     eax, eax
.text:080484C9 8B 5D 08       mov     ebx, [ebp+arg_0]
.text:080484CC 31 D2          xor     edx, edx
.text:080484CE EB 04          jmp     loc_80484D4
.text:080484D0
.text:080484D0 loc_80484D0:
.text:080484D0 02 04 13       add     al, [ebx+edx]
.text:080484D3 42             inc     edx
.text:080484D4
.text:080484D4 loc_80484D4:
.text:080484D4 39 CA          cmp     edx, ecx
.text:080484D6 7C F8          jl      short loc_80484D0
.text:080484D8 0F B6 C0       movzx   eax, al
.text:080484DB F7 D8          neg     eax
.text:080484DD 5B             pop     ebx
.text:080484DE C9             leave
.text:080484DF C3             retn
```

Листинг 5 результат трансляции crc() компилятором GCC 3.4.2

Ага! Программа сократилась до 20h байт, что всего на 8 байт длиннее ассемблерной программы, но цикл практически не изменился. По прежнему используется сложная адресация и никому не нужная команда `MOVZX`. Изменилась только точка входа в цикл. Вместо того, чтобы проверять значение аргумента n _до_ входа в цикл, компилятор сформировал условный переход на проверку, выполняемую перед выходом из цикла, то есть использовал тот же самый трюк, что и мы в нашей ассемблерной программе, однако, в отличи от нас, компилятор использовал 4е регистра, а не 3 и к тому же сгенерировал стандартный пролог/эпилог, который, впрочем, при желании можно подавить ключами командной строки, но это все равно не поможет, поскольку…

Цикл остается неразвернутым! (под "разворотом" в общем случае понимается многократное дублирование цикла) На таком крохотном "пяточке" процессору просто негде развернуться, поэтому все будет очень жутко тормозить – как при ручной оптимизации, так и при машинной. Компилятор MS VC вообще не умеет разворачивать циклы. По жизни. GCC умеет, но по умолчанию не делает этого даже на уровне оптимизации -O3, разве что его специального попросить, указав ключ `-funroll-all-loops` в командной строке. Циклы с известным количеством итераций, где const <= 32 разворачиваются полностью, при const > 32 — на ~4x (точное значение зависит от количества инструкций в теле цикла), но циклы с неизвестным количеством итераций (то есть такие циклы, параметр которых — переменная, а не константа) не разворачиваются вообще! И в этом есть свой резон.

При малых количествах итераций цикл лучше не разворачивать, поскольку выигрыш в скорости не окупится увеличением размера программы и усложнением логики, особенно, если цикл расположен внутри редко вызываемой функции. А вот разворот часто вызываемого цикла с большим количеством итераций дает по меньшей мере двукратный прирост производительности. Главное — не переборщить и не развернуть цикл сильнее, чем требуется. На большинстве процессоров рост скорости прекращается при развороте на 8 итераций, и дальнейшее дублирование тела цикла лишь увеличивает его размер (см. рис. 3).

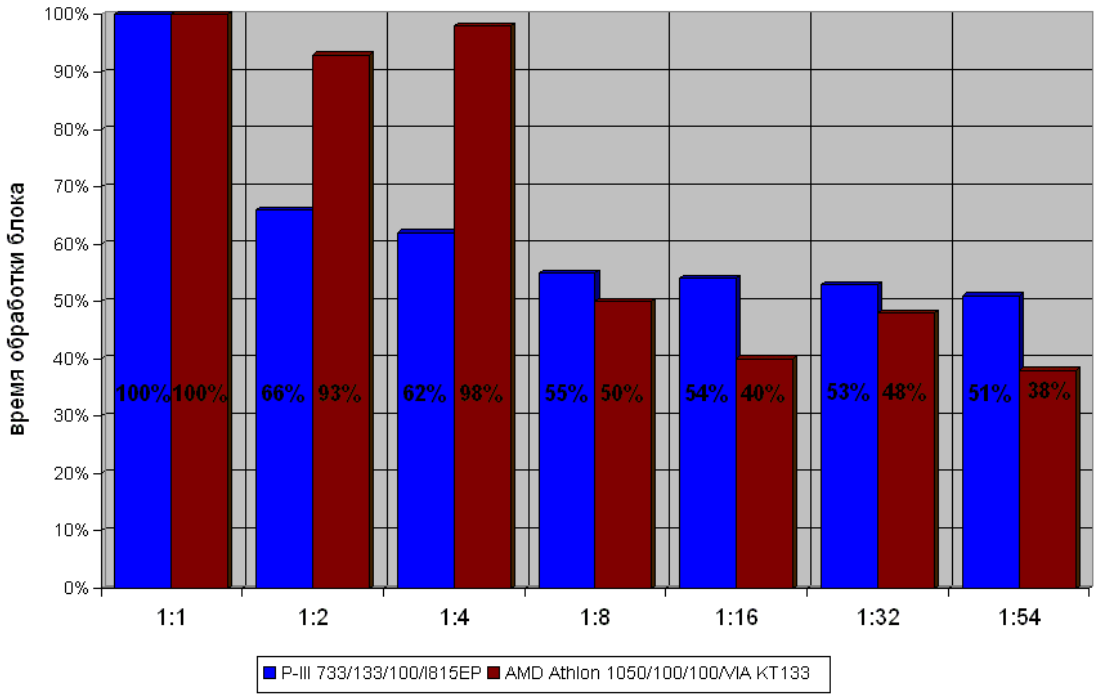

Рисунок 3 влияние кратности разворота цикла на производительность на различных типах процессоров

Выполнить разворот цикла можно как на ассемблере (на MASM'е за счет поддержки развитой системы макрокоманд он реализуется особенно легко), так и на… любом языке высокого уровня, действуя в обход компилятора.

После "ручной" оптимизации исходный текст нашей программы будет выглядеть так:

```
if ((a=n)>3)
        // обрабатываем первые n - (n % 4) итераций
        for (a = 0; a < n - 3; a += 4)
        {
                crc_1 += p[a+0];
                crc_2 += p[a+2];
                crc_3 += p[a+3];
                crc_4 += p[a+4];
        }

// обрабатываем оставшийся "хвост"
for (a = n - x % 4; a < x; a++) crc += p[a];

// складываем все воедино
crc += crc_1 + crc_2 + crc_3 + crc_4;
```

Листинг 6 оптимизированный вариант crc() с развернутым циклом

При сравнении со своим ассемблерным аналогом (так же развернутым) она покажет практически идентичный вариант по скорости, и будет вполне сопоставима по размеру, поскольку львиная доля кода придется на развернутое тело цикла, на фоне которого меркнут мелкие накладные расходы. Но! Это справедливо _только_ для циклов с большим количеством итераций, берущих данные из медленной оперативной памяти, а то и считывающих их с диска. Тут на отдельных машинных командах можно не экономить, главное — выбрать правильную стратегию разворота, а она для каждого из процессоров разная.

Часто вызываемые циклы с небольшим количеством итераций, по-прежнему эффективнее писать на ассемблере, поскольку даже одна лишняя машинная команда приводит к существенному оверхиду, а он нам нужен? Если, конечно, мы вообще, заботимся о производительности…

Кстати, обратите внимание, что в развернутом цикле ячейки памяти складываются в различные переменные, а не суммируются в одну! Если развернуть цикл так, как показано в листинге 7, то выигрыш в производительности окажется намного меньшим.

```
// выполняем первые n - (n %4) итераций
```

```
for(a = 0; a < n - 3; a += 4)
{
        crc += p[a+0] + p[a+1] + p[a+2] + p[a+3];
}
```

Листинг 7 пример неправильного разворота цикла

Почему?! Да потому что образуется паразитная зависимость по данным. Процессор не может выполнять "`+ p[a+1]`" пока не завершится сложение `crc` с `p[a+0]` и вычислительный конвейер вынужден простаивать в ожидании!

В моей книге "техника оптимизации программ" (фрагменты которой можно скачать с ftp://nezumi.org.ru) описано множество трюков высокоуровневой оптимизации, основанных на архитектурных особенностях железа (DRAM-банков, контроллеров памяти, процессора), учет которых дает значительный выигрыш даже без использования ассемблера!

заключение

И все же есть области, в которых ассемблер необходим, можно даже сказать, неизбежен. В первую очередь это высокопроизводительные математические и графические библиотеки, использующие векторные инструкции типа MMX или SSE. На эффективную векторизацию данных компиляторы, увы, не способны.