экстремальный разгон процессора

крис касперски, aka мыщъх, aka nezumi, aka souriz, aka elraton, no-email

непрерывный мониторинг внутреннего состояния процессора позволяет значительно повысить его разгонный потенциал, автоматически подстраиваясь под характер запросов конкретных приложений, основываясь на показании счетчиков производительности, которые легко считать крохотной ассемблерной программой

введение

Процессор представляет собой сложное устройство, состоящее из множества "разнокалиберных" узлов, гонимая способность каждого из которых сильно неодинакова, но все они "запитываются" от общего генератора тактовой частоты и потому менее гонимые блоки тормозят все остальные, особенно когда оказываются интенсивно задействованы каким-нибудь "тяжеловесным" приложением.

Материнские платы и процессоры последних поколений поддерживают динамический разгон, основанный на показаниях термодатчика. Как только температура кристалла достигает первой критической отметки, материнская плата увеличивает обороты вентилятора, пытаясь снизить нагрев. Если же вентилятор не справляется и температура по прежнему продолжает расти, при достижении второй критической отметки процессор начинает либо вставлять холостые циклы, либо снижает тактовую частоту всех своих компонентов, что приводит к неоправданному падению производительности.

Большинство систем динамического разгона (как программных, так и аппаратных) основано именно на температурных показаниях и не полностью реализуют потенциал процессора, поскольку, кристалл обладает большой температурной инерционностью, кроме того, абсолютное показание температуры еще ни о чем не говорят! Вот и приходится оставлять солидный запас "прочности" по частоте, чтобы обеспечить стабильную работу системы. А что еще можно ожидать от таких грубых методов?! Мыщъх провел широкомасштабное исследование, длившееся несколько лет, и в конечном счете, совершившее настоящий прорыв в область высоких скоростей и недостижимых ранее тактовых частот.

разбор полетов и крушений

Последствия чрезмерного разгона всем хорошо известны — это критические ошибки приложений и "голубые экраны смерти". С первого взгляда ничего удивительно тут нет. Какой-то из модулей процессора не выдержал издевательств и поехал крышей, возвратив некорректный результат. Какое-то время мыщъх не уделял этому вопросу особого внимания, но потом заинтересовался и решил исследовать: какой же из блоков сбоит чаще всего?

Исследования на "голом" железе без операционной системы, показали, что АЛУ (арифимитическо-логическое устройство) сохраняет работоспособность и всегда возвращает правильный результат даже на запредельных тактовых частотах, при которых стабильно завешивается MS-DOS, ну а Windows даже и не пытается загружаться! Почему?!

Снижаем тактовую частоту до такого уровня, при котором Windows успевает выдать сообщение о критической ошибке, сохранив дамп памяти (если исключение произошло в ядре) или сгенерировав отчет Доктора Ватсона (если исключение произошло на прикладном уровне). Анализ полученных данных долгое временя не давал никакой осмысленной информации. Ошибки происходили по разным адресам, охватывая практически весь набор инструкций: от целочисленных до MMX/SSE, и, казалось, что эксперименты (загубившие немало процессоров) пора прекращать, поскольку, никакого полезного выхлопа они все равно не принесут.

К тому же, некоторые дампы с точки зрения здравого смысла выглядели абсолютно бессмысленными и даже мистическими. Как-то раз Доктор Ватсон заявил, что машинная команда XOR ECX, ECX возбудила исключение типа Access Violation по адресу C23BD2BAh, тогда как сам ECX равнялся 87h. Но ведь этого не может быть!!! Это же полная и абсолютная ерунда!!! Инструкция XOR ECX, ECX _вообще_ не обращается к памяти!!! Но… протокол Доктора Ватсона есть протокол (читай — документ) и одним движением хвоста в корзину его не выбросишь…

Озарение, как обычно, пришло после хорошей травы, тьфу, то есть во сне, точнее не совсем во сне, а на границе сумеречной зоны, отделяющий один мир от другого, когда после

30 часов непрерывного траханья с Доктором Ватсоном, ты спишь наяву, уткнувшись в очередной фрагмент кода, вызвавший сбой:

```
00000000: 33C9        XOR    ECX, ECX       ; (начальное значение ECX == 87h)
00000002: 33D2        XOR    EDX, EDX
00000004: 3BC2        CMP    EAX, EDX
```

Листинг 1 инструкция XOR ECX, ECX (для наглядности выделенная полужирным шрифтом) вызвала нарушение доступа, обратившись к памяти по адресу C23BD2BAh

Ничего не напоминает?! Постойте-постойте! Но ведь… адрес, вызывавший исключение, содержит в себе байты инструкции CMP EAX, EDX и частично XOR EDX, EDX, а если записать опкоды этих команд и сложить их со значением регистра ECX, получится: C23BD233h + 87h == **C23BD2BAh**, то есть тот самый непонятно откуда взявшийся адрес исключения (ну это раньше он был непонятным, теперь же все стало ясно). Записав инструкцию XOR ECX,ECX в двоичном виде (00011 0011 1100 100) и изменив всего один бит, превращающий C9h в 89h, мы получим… мы получим вот что!

```
00000000: 338933D23BC2 XOR    ECX,[ECX][0C23BD233]
```

Листинг 2 предыдущий фрагмент кода, в котором искажен всего один бит в инструкции XOR (C9h → 89h)

Оторвать мне хвост!!! Вот как оказывается в _действительности_ выглядела машинная команда, возбудившая исключение и вызвавшая сбой. Сразу видно, что АЛУ тут совершенно не причем. Процессор функционировал в общем-то исправно. Весь вопрос в том, почему Доктор Ватсон показал не "XOR ECX, [ECX][0C23BD233]", а "XOR ECX, ECX"?! Да потому, что искажение бита произошло в кэш-памяти первого уровня, а при составлении отчета Доктор Ватсон возвратил неискаженное содержимое кэш-памяти второго уровня!!! Откуда у меня такая уверенность, что все именно так и происходило? Так ведь процессор использует раздельную кэш память первого уровня для кода и данных, поэтому, прочитать истинное содержимое инструкции, вызывавшей сбой, Доктор Ватсон просто физически не в состоянии и это можно установить только косвенным путем.

Так, значит, главный виновник — это кэш? Дальнейшие эксперименты показали, что все обстоит именно так. Причем, сбои происходят в кэш памяти обоих уровней и вероятность их возникновения напрямую связана с интенсивностью кэш-промахов (т. е когда приложение обновляет большое количества кода/данных). С другой стороны, длительное хранение кода/данных без их модификации, создает другую угрозу — угрозу "загнивания" байт, особенно часто случающуюся при некачественном питании.

Изменить тактовую частоту кэш-модуля невозможно, но… если пораскинуть хвостом, можно найти довольно простое и элегантное решение.

руководящая идея

Процессоры семейства Pentium поддерживают счетчики производительности (performance-monitoring events), позволяющие подсчитывать различные события, в том числе и количество кэш-промахов — как раз то, что нам нужно! Пишем несложную программу, работающую в фоновом режиме и несколько раз в секунду считывающие значение счетчика кэш-промахов. Зафиксировав стремительный рост кэш-промахов, слегка тормозим процессор, чтобы кэш в промежутках между загрузкой новой порцией данных успевал приостыть. Так же, обнаружив, что данные в кэш памяти давно не менялись, обновляем их, предотвращая возможное "загнивание".

Параметры "торможения" и частоту обновления данных в кэш-памяти необходимо подбирать экспериментально, лавируя между производительностью и надежностью, причем и производительность, и надежность будут намного выше чем при обычных методах разгона. Мыщъх в последнее время обнаглел до того, что перестал заботится об охлаждении и перешел на обычную термопасту и дешевые алюминиевые радиаторы с медленно вращающимся (а, значит, бесшумными) пропеллерами.

Вот какие преимущества дает программный разгон! Причем ключевой исходный код легко укладывается в несколько сотен строк и пишется (с отладкой!) за один вечер, плавно перетекающий в ночь, проведенный за игрой в 3D-стрелянку или перекодировку DVD в DivX – это уж кто чем больше заниматься любит.

как мы будем действовать

Счетчики производительности по разному реализованы в процессорах семейства P6 (к которым принадлежат Pentium Pro/Pentium-II/Pentium-3) и Pentium-4. Никаких принципиальных различий нет, но коды счетчиков производительности и номера MSR-регистров слегка другие и код, предназначенный, для P6, попав на Pentium-4, вызывает исключение, как правило, заканчивающиеся голубым экраном смерти под Windows NT.

Главным образом мы будем говорить про семейство процессоров P6 и в этом есть свой резон, во-первых, они в наибольшей степени нуждаются в разгоне (Pentium-4 и без того производительны), и, во-вторых, в отличии от Pentium-4 они не поддерживают автоматическое снижение тактовой частоты при перегреве, уменьшая свой разгонный потенциал. Но, как бы там ни было, перенести код с P6 на Pentium-4 сможет любой программист, даже начинающий, так что не будет отвлекаться на несущественные различия между этими платформами, а сразу перейдем к делу.

Процессоры семейства P6 несут на своем борту два счетчика производительности, физически представляющие собой внутренние 40-битные MSR-регистры — PerfCtr0 и PerfCtr1, каждый из которых может подсчитывать события определенного вида, коды которых задаются другими MSR-регистрами — PerfEvtSel0 и PerfEvtSel1 соответственно. Они же отвечают за запуск/останов счетчиков производительности.

Коды событий, которые процессор может подсчитывать, перечислены в приложении "A" руководства по системному программированию "Intel Architecture Software Developer's Manual Volume 3: System Programming Guide". В частности, событие "промах кэш памяти данных" проходит под номером 48h, а "промах кэш памяти кода" — 81h.

Unit	Event Num.	Mnemonic Event Name	Unit Mask	Description	Comments
Data Cache Unit (DCU)	47H	DCU_M_LINES_ OUT	00H	Number of M state lines evicted from the DCU. This includes evictions via snoop HITM, intervention or replacement.	
	48H	DCU_MISS_ OUTSTANDING	00H	Weighted number of cycles while a DCU miss is outstanding.	An access that also misses the L2 is short-changed by 2 cycles (i.e., if counts N cycles, should be N+2 cycles). Subsequent loads to the same cache line will not result in any additional counts. Count value not precise, but still useful.
Instruction Fetch Unit (IFU)	80H	IFU_IFETCH	00H	Number of instruction fetches, both cacheable and noncacheable.	
	81H	IFU_IFETCH_ MISS	00H	Number of instruction fetch misses.	

Рисунок 1 номера различных событий, за которыми можно вести мониторинг с помощью счетчиков производительности

Чтение/запись MSR регистров осуществляется командами RDMSR/WRMSR, доступными _только_ из нулевого кольца и действующими следующим образом: в регистр ECX помещается номер выбранного MSR-регистра, а в регистровой паре EDX:EAX – возвращаемое/записываемое значение. Номера MSR-регистров так же можно узнать из руководства по системному программированию. Так например, номер регистр PerfEvtSel0 имеет номер 186h, а структура его управляющих полей приведена на рис. 1.

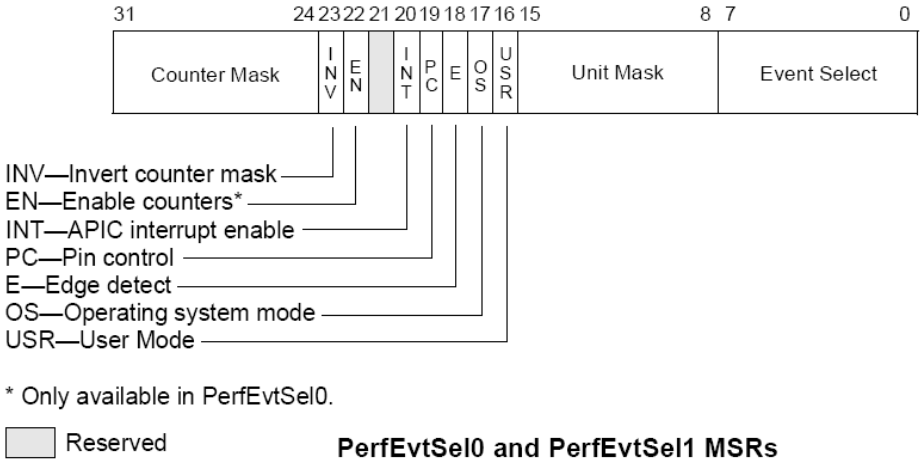

Рисунок 2 структура MSR-регистров PrefEvtSel0/ PrefEvtSel1

Собственно говоря, все, что нам нужно это занести код события в регистр PerfEvtSel0/PerfEvtSel1 (биты 0-7), маску события, в данном случае равную нулю (биты 8-15) и взвести флажок Enable Counter (бит 22), чтобы начать подсчет событий. Описание остальных битов можно найти в документации, нам они совершенно не интересны за исключением, пожалуй, поля USR (бит 16), открывающего к счетчику доступ с пользовательского уровня, позволяя реализовать основной код в программе прикладного режима, которую намного проще отлаживать чем драйвер.

Но все-таки совсем без драйвера обойтись не получится, поскольку инструкция RDMSR на прикладном уровне возбуждает неизменное исключение. Как же быть?! Intel предоставила крошечную лазейку в виде команды RDPMC читающей текущий счетчик производительности в регистровую EDX:EAX. Текущий — это тот, который до этого был установлен командой WRMSR, запустивший MSR-регистр PerfEvtSel0 или PerfEvtSel1. Однако, по умолчанию, RDMSR с прикладного уровня недоступна и прежде, чем ей удастся воспользоваться необходимо взвести PCE флажок в регистре CR4 (бит 8), модифицировать который можно только из нулевого кольца, зато потом наступает благодать!!!

Подробнее о счетчиках производительности и всем, что с ними связано можно прочитать в разделе "Performance-Monitoring Events and Counters" руководства "Intel Architecture Optimization Reference Manual" или уже упомянутой "библии" системного программиста "Intel Architecture Software Developer's Manual Volume 3: System Programming Guide"

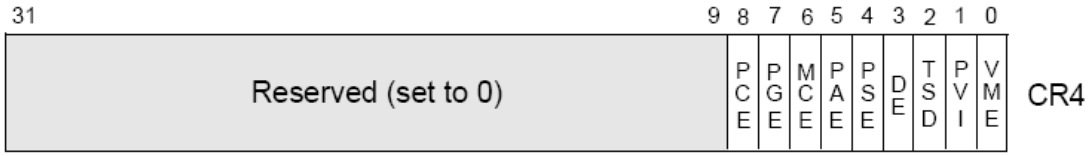

Рисунок 3 бит PCE регистра CR4 управляет доступом к команде RDPMC с прикладного уровня

Таким образом, мыщъх'иная программа состоит из двух частей: крохотного псевдодрайвера и прикладной части. Драйвер обеспечивает загрузку необходимого кода события в соответствующий MSR-регистр (PerfEvtSel0 или PerfEvtSel1) и запускает счетчик, предварительно "разблокировав" команду RDPMC.

Поскольку, RDPMC способна читать только один счетчик (а нам необходимо отслеживать по меньшей мере два события — промахи кэш памяти кода и данных), драйвер должен обеспечивать IOCTL-интерфейс с прикладным приложением, позволяя ему переключаться с одного счетчика на другой.

Чтобы не переводить понапрасну бумагу, ниже будут приведены только ключевые фрагменты кода, а все остальное читатель без труда допишет и сам. В частности, процедура инициализации драйвера среди прочего должна содержать:

```
DriverInitialize:             ; // процедура инициализации драйвера
        …
        MOV     EAX, CR4
        OR      EAX, 100h ; // разрешаем доступ к RDPMC с прикладного уровня
        MOV     CR4, EAX
        …
```

Листинг 3 фрагмент процедуру инициализации драйвера

Следующий код обеспечивает взаимодействие драйвера с прикладной программой через API-функцию **`DeviceIOControl`**, передающий в IOCTL-коде номер события, за которым необходимо вести мониторинг. По соображениям наглядности, здесь используется всего лишь один счетчик производительности, управляемый MSR-регистром PerfEvtSel0.

```
IRP_MJ_DEVICE_CONTROL:                ; // процедура обработки IOCTL-запросов
        ; // настраиваем регистр perfevtsel0 для мониторинга нужных событий
        XOR     EDX, EDX
        MOV     EAX, pisl->Parameters.DeviceIoControl.IoControlCode ; //номер события
        TEST    EAX, EAX              ; // если код события равен нулю
        JZ      wrt                   ; // то вырубаем счетчик

        OR      EAX, 10000h           ; // делаем счетчик доступным
                                      ; // с прикладного уровня
        OR      EAX, 400000h          ; // пускаем счетчик
wrt:
        MOV ECX,0x186                 ; // выбираем MSR-регистр PERFEVTSEL0
        WRMSR
```

Листинг 4 фрагмент драйвера, отвечающий за выбор нужного события

При деинициализации драйвера крайне желательно "отобрать" доступ к команде RDPMC с прикладного уровня и остановить все ранее запущенные счетчики производительности, сбросив флажок Enable Counter в MSR-регистрах PerfEvtSel0/PerfEvtSel1 (код, приведенный ниже останавливает только PerfEvtSel0):

```
DriverUnload:                 ; // процедура деинициализации драйвера
        …
        ; // сбрасываем бит pce регистра cr4 для запрета чтения
        ; // счетчика производительности с пользовательского уровня
        MOV     EAX, CR4
        MOV     ECX, 100h
        NOT     ECX           ; // запрещаем доступ к RDPMC с прикладного уровня
        AND     EAX, ECX
        MOV     CR4, EAX

        ; // останавливаем счетчик производительности
        XOR     EDX, EDX
        XOR     EAX, EAX
        MOV     ECX, 186h
        WRMSR
        …
```

Листинг 5 фрагмент процедуры деинициализации драйвера

Прикладная программа первым делом должна загрузить драйвер (пусть для определенности он будет называться 996.SYS), открыв его с помощью функции CreateFile. При этом управление получит процедура инициализации, открывающая доступ к машинной команде RDPMC, но сами счетчики производительности еще не заданы, так что читать, собственно говоря, нечего и незачем.

Нет никакой необходимости писать загрузку драйвера на ассемблере и лучше всего воспользоваться для этой цели языком Си:

```
// определения необходимых констант
#define PrefCtrl0             0x0000
#define DCU_MISS_OUTSTANDING  0x0048

// дескриптор драйвера 996
static HANDLE _996_handle = INVALID_HANDLE_VALUE;

int _996_init()
{
```

```
        if (_996_handle == INVALID_HANDLE_VALUE)
        {
                _996_handle = CreateFile("\\\\.\\996",GENERIC_READ,
                                        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
                                        OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

                if (_996_handle == INVALID_HANDLE_VALUE) return 0;
        } return 1;
}
```

Листинг 6 прикладная функция, загружающая драйвер в память

Тоже самое относится и к функции, вызывающей DeviceIoControl и передающей ей код интересующего нас события. На языке Си она выглядит гораздо нагляднее:

```
int _996_select(int xCode, int REG)
{
        DWORD x;
        if (REG != PrefCtrl0) return 0;

        // если программист забыл загрузить драйвер,
        // данная функция делает это самостоятельно
        if (_996_handle == INVALID_HANDLE_VALUE) _996_init();

        // если загрузка драйвера провалилась сваливаем отсюда
        if (_996_handle == INVALID_HANDLE_VALUE) return 0;
        return DeviceIoControl(_996_handle, xCode, &x,0, &x, 0, &x, 0);
}
```

Листинг 7 прикладная функция, позволяющая выбирать интересующее нас событие для его мониторинга

Процедура закрытия драйвера должна просто вызывать CloseHandle, а все остальное за нас сделает сам драйвер. Впрочем, драйвер можно и не закрывать. При выходе из приложения операционная система сделает это автоматически.

```
int _996_exit()
{

        if (_996_handle != INVALID_HANDLE_VALUE)
        {
                CloseHandle(_996_handle);
        }
        return 1;
}
```

Листинг 8 прикладная функция выгружающая драйвер из памяти

А вот при снятии показаний со счетчиков производительности без ассемблера уже не обойтись! Для упрощения программирования можно использовать ассемблерные вставки, хоть это и является признаком дурного тона, затрудняющих перенос программы на другие платформы и препятствующей ее компиляции другим компилятором. Правильным решением было бы создание отдельного ассемблерного модуля, но это слишком хлопотно, тем более, что мы пишем не коммерческую программу, а всего лишь демонстрационный макет.

```
        // ИНИЦИЛИЗАЦИЯ ДРАЙВЕРА 996
        if (_996_init()==0) return printf("-ERR: 996 driver not loaded!\n");

        // ВЫБОР СОБЫТИЯ ДЛЯ МОНИТОРИНГА И ЗАПУСК СЧЕТЧИКА
        _996_select(DCU_MISS_OUTSTANDING, PrefCtrl0);

        for(;;)
        {
                __asm
                {
                        mov     ecx, PrefCtrl0 ; // читаем регистр PrefCtrl0...
                        RDPMC                  ; // ...и помещаем результат в EDX:EAX

                        mov _edx, edx          ; // сохраняем EDX:EAX в...
                        mov _eax, eax          ; //           ...одноименных переменных
                }

                // анализ кол-ва кэш-промахов
```

```
        // ==========================

        …
        Sleep(0);                    ; // отдаем остаток кванта и спим
    }
```

Листинг 9 ключевой фрагмент функции, осуществляющей контроль за кэш-активностью

При снятии показания со счетчиков производительности следует учитывать, что они возвращают количество кэш-промахов с момента запуска счетчика, а не между двумя соседними замерами, так что дельту придется считать самостоятельно. И если эта дельта вдруг превысит некоторое пороговое значение (задаваемое настройками нашей программы), необходимо "притормозить" процессор, чтобы кэш чуть-чуть приостыл. А как это можно сделать? Ведь даже если материнская плата поддерживает изменение тактовой частоты процессора на лету, каждая из них делает это по-разному и у нас получается громоздкая и не универсальная программа.

На самом деле, нет ничего проще! Достаточно просто прекратить отдавать кванты, загрузив процессор "тупой" работой, не требующей обращения к памяти. Например, складывать два регистра в цикле. При условии, что в системе имеются два активных потока, один из которых принадлежит приложению, гоняющему кэш и в хвост и в гриву, а другой поток — гонят цикл в нашей программе, на однопроцессорных материях операционная система будет выделять приложению только 50% машинного времени, следовательно, нагрузка на кэш упадает. А если мы запустим три потока, мотающие такие циклы, кэш-приложение получит только 25% машинного времени! Количество протоков и продолжительность выполнения цикла подбираются экспериментально и для каждого приложения они индивидуальны (а это значит, что для достижения наивысшей производительности придется отслеживать какие приложения запущены и выбирать соответствующий им профиль. муторно конечно, но разгон того стоит):

```
        MOV ECX,-1
cool:
        ADD EAX,ECX
        DEC ECX
        LOOP cool
```

Листинг 10 цикл, отбирающий процессорные такты у приложения, напрягающего кэш и дающее ему время на остыв

Остается разобраться с "загниванием" байтов в "застоявшейся" кэш-памяти. Ну тут все просто! Хоть мы не можем непосредственно обновить ее содержимое, достаточно просто с некоторой периодичностью (определяемой опять-таки чисто экспериментально) загружать в кэш посторонние данные (ну там мусор какой-нибудь), заставляя приложение заново перечитывать оригинальное содержимое из оперативной памяти. Учитывая, что пропускная способность современных DRAM-контроллеров измеряется гигабайтами в секунду, особого падения производительности это не вызовет, зато позволит разогнать процессор до сумасшедших тактовых частот!

заключение

Разгон — дело рискованное, можно не только потерять данные на жестком диске, но и вывести процессор из строя, а то и всю материнскую плату. Тем более, что таковая частота в большинстве случаев не является самым "узким" местом и разгон носит скорее спортивный интерес.