Beginnings of a document on random P4 optimizations and insights

Those considering assembly language optimizations on the P4 should first read the Intel manuals at http://developer.intel.com/design/Pentium4/manuals/ especially the Processor Optimization Reference manual at http://developer.intel.com/design/pentium4/manuals/248966.htm.

Each section tries to analyze one small aspect of the P4 architecture. It is followed by a conclusion or speculation regarding the observed behavior. Often, these test cases raise more questions than they answer.

Each code snipet is timed in a loop of at least 100 iterations. Memory is aligned on a 4KB page boundary. The code is then run 200 times and the best timing is reported. Note my machine is a 1.4 GHz P4 with PC600 memory (theoretical bandwidth of 2.4 GB/sec). Note: I've recently been told that the peak bandwidth for PC600 memory is 2.0 GB/sec (that is, PC600 is in reality PC512 memory!). I'm trying to track down a reference for this, so far unsuccessfully.

Note, the Intel manuals sometimes refer to cache line size as 128 bytes and sometimes as 64 bytes. In the examples below I'll refer to a cache line as 64 bytes. The L1 cache line size is 64 bytes. The L2 cache is kind of a hybrid 128/64 byte cache. A cache line fill reads 128 bytes, but my tests show that the L2 cache maintains separate dirty bits for each 64-byte chunk.

Reading contiguous data

This code reads a contiguous block of memory using 4 bytes at a time. Timings are done on 3 memory size. 4KB will read from the L1 cache only, 64KB will read from the L2 cache only, and 1MB will test reading from main memory.

```
        mem = 4096              ; Read 4KB
        cnt = mem/64            ; Read 64 bytes per iteration
        mov     ecx, cnt
loop1:  mov     eax, [esi]                ; Read one cache line
        mov     eax, [esi+4]              ; 4 bytes at a time
        mov     eax, [esi+8]
        mov     eax, [esi+12]
        mov     eax, [esi+16]
        mov     eax, [esi+20]
        mov     eax, [esi+24]
        mov     eax, [esi+28]
        mov     eax, [esi+32]
        mov     eax, [esi+36]
        mov     eax, [esi+40]
        mov     eax, [esi+44]
        mov     eax, [esi+48]
        mov     eax, [esi+52]
        mov     eax, [esi+56]
        mov     eax, [esi+60]
        lea     esi, [esi+64]           ; Next cache line
        sub     ecx, 1
        jnz     loop1
        lea     esi, [esi-mem]          ; Restore esi
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Mem	Clocks	Clocks per loop iteration	Bandwidth	Comments
4096 (4KB)	1037	16.2	5.53GB/sec	The expected result
65536 (64KB)	23625	23.07	3.88GB/sec	Any good theories here?
1048576 (1MB)	1180000	72.02	1.24GB/sec	50% of theoretical throughput. Why?

Now using MMX instructions:

```
        mem = 4096                      ; Read 4KB
        cnt = mem/64                    ; 64 bytes per iteration
        mov     ecx, cnt
loop1:  movq    mm1, [esi]              ; Read one cache line
        movq    mm1, [esi+8]            ; 8 bytes at a time
        movq    mm1, [esi+16]
        movq    mm1, [esi+24]
        movq    mm1, [esi+32]
        movq    mm1, [esi+40]
        movq    mm1, [esi+48]
        movq    mm1, [esi+56]
        lea     esi, [esi+64]           ; Next cache line
        sub     ecx, 1
        jnz     loop1
        lea     esi, [esi-mem]
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Mem	Clocks	Clocks per loop iteration	Bandwidth	Comments
4096 (4KB)	523	8.17	10.96GB/sec	As expected.
65536 (64KB)	15395	15.03	5.96GB/sec	
1048576 (1MB)	1036299	63.25	1.42GB/sec	Still not theoretical throughput. Why?

Now using XMM instructions:

```
        mem = 4096                      ; Read 4KB
        cnt = mem/64                    ; 64 bytes per iteration
        mov     ecx, cnt
```

```
loop1:  movapd  xmm1, [esi]             ; Read one cache line
        movapd  xmm1, [esi+16]          ; 16 bytes at a time
        movapd  xmm1, [esi+32]
        movapd  xmm1, [esi+48]
        lea     esi, [esi+64]           ; Next cache line
        sub     ecx, 1
        jnz     loop1
        lea     esi, [esi-mem]
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Mem	Clocks	Clocks per loop iteration	Bandwidth	Comments
4096 (4KB)	268	4.19	21.40GB/sec	
65536 (64KB)	7766	7.58	11.81GB/sec	
1048576 (1MB)	817776	49.91	1.80GB/sec	Still not theoretical throughput. Why?

Conclusions: The P4 is unable to hide all the latencies when reading from the L2 cache. This is somewhat surprising, as it should be getting near perfect branch prediction. ***Also, how does Intel justify claims of 44.8GB/sec for L2 cache bandwidth?***

One would hope that the P4 would get closer to its 2.4GB/sec peak bandwidth. ***Why isn't it?***

Now we will see if using the SSE2 prefetch instructions can get us closer to the 2.4GB/sec theoretical bandwidth. We'll also see how much as a performance penalty prefetch incurs when reading from the L2 cache.

```
        mem = 16*16*4096                ; Read 1MB
        pages = mem/4096                ; Number of 4KB pages
        mov     ecx, pages
        sub     ebx, ebx
loop0:  mov     eax, [esi+4096]         ; Preload next page TLB
loop1:  movapd  xmm1, [esi]             ; Read 2 cache lines
        movapd  xmm1, [esi+16]          ; 16 bytes at a time
        movapd  xmm1, [esi+32]
        movapd  xmm1, [esi+48]
        movapd  xmm1, [esi+64]
        movapd  xmm1, [esi+80]
        movapd  xmm1, [esi+96]
        movapd  xmm1, [esi+112]
        prefetcht1 [esi+4096]           ; Preload 2 lines in next page
        lea     esi, [esi+128]          ; Next 2 cache lines
        add     bl, 256/32              ; 32 iterations per page
        jnc     loop1
        sub     ecx, 1
        jnz     loop0
        lea     esi, [esi-mem]
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Mem	Clocks	Clocks per 64 byte cache line	Bandwidth	Comments
65536 (64KB)	8602	8.40	10.67GB/sec	Unnecessary prefetch causes some overhead
1048576 (1MB)	737952	45.04	1.99GB/sec	Still not theoretical throughput. Why?

Conclusions: The prefetch instruction caused a modest 11% overhead (8602 vs. 7766 clocks) when not reading from memory. This overhead may well disappear in real-world code where the loop is actually doing real work.

Reading non-contiguous data

This example reads an entire block of memory, but in a non-contiguous fashion.

```
        mem = 4096                      ; Read 4KB
        cnt = mem/(4*128)               ; 128 bytes per iteration
        dist = 64
        sub     ebx, ebx
        mov     ecx, cnt
loop1:  movapd  xmm1, [esi+0*dist]      ; Read 8 cache lines
        movapd  xmm1, [esi+1*dist]
        movapd  xmm1, [esi+2*dist]
        movapd  xmm1, [esi+3*dist]
        movapd  xmm1, [esi+4*dist]
        movapd  xmm1, [esi+5*dist]
        movapd  xmm1, [esi+6*dist]
        movapd  xmm1, [esi+7*dist]
        lea     esi, [esi+16]           ; Same 8 cache lines
        add     bl, 256/4               ; 4 inner loop iterations
        jnc     loop1
        lea     esi, [esi-4*16+8*dist]  ; Next set of 8 cache lines
        sub     ecx, 1
        jnz     loop1
        lea     esi, [esi-mem]
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Mem	Clocks	Clocks per 64 byte cache line	Bandwidth	Comments
4096 (4KB)	268	4.19	21.40GB/sec	Same as contiguous
65536 (64KB)	5900	5.76	15.55GB/sec	Much faster!
1048576 (1MB)	818193	49.94	1.80GB/sec	Same as contiguous

Conclusions: This is very interesting. If your application’s data fits in the L2 cache, grouping your L1 cache misses together increases your throughput. You may want to try preloading L2 cache lines into the L1 cache in an earlier iteration. For example, adding the line

```
        mov     eax, [esi+512]          ; Preload line into L1 cache
```

in the contiguous read code speeds up the 64KB timing. Also, if your application uses prefetcht1 to preload L2 cache lines and your application is not limited by main memory bandwidth, then preloading lines into the L1 cache could speed up your application.

Writing contiguous data

```
        mem = 4096                      ; Write 4KB
        cnt = mem/64                    ; 64 bytes per iteration
        mov     ecx, cnt
loop1:  movapd  [esi], xmm1             ; Write one cache line
        movapd  [esi+16], xmm1          ; 16 bytes at a time
        movapd  [esi+32], xmm1
        movapd  [esi+48], xmm1
        lea     esi, [esi+64]           ; Next cache line
        sub     ecx, 1
        jnz     loop1
        lea     esi, [esi-mem]
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Mem	Clocks	Clocks per 64 byte cache line	Bandwidth	Comments
4096 (4KB)	902	14.09	6.36GB/sec	Slower than reading
65536 (64KB)	14372	14.04	6.38GB/sec	Slower than reading.
1048576 (1MB)	1768230	107.92	0.83GB/sec	

Conclusions: The write-through L1 cache is a killer here. It takes 14 clocks to write the L1 cache line back to the L2 cache. This places a limit on how fast you can code an application that reads and writes only from the L1 and L2 cache. To keep the CPU fully working you must find 14 clocks of work to do for each 64 bytes your program writes. Clearly, the writes to the L2 cache are not pipelined.

The main memory benchmark is also interesting. Obviously, the processor needs to read the cache line from memory before doing the write operation. The theoretical maximum bandwidth is thus 1.2GB/sec. This example achieves only 66% of theoretical, while the contiguous read example achieves 75% of theoretical. ***Why?***

Now let's try it using the movntpd instruction. This should avoid the reading of the cache line prior to writing the cache line.

```
        mem = 4096                      ; Write 4KB
        cnt = mem/64                    ; 64 bytes per iteration
        mov     ecx, cnt
loop1:  movntpd  [esi], xmm1            ; Write one cache line
        movntpd  [esi+16], xmm1         ; 16 bytes at a time
        movntpd  [esi+32], xmm1
        movntpd  [esi+48], xmm1
        lea     esi, [esi+64]           ; Next cache line
        sub     ecx, 1
        jnz     loop1
        lea     esi, [esi-mem]
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Mem	Clocks	Clocks per 64 byte cache line	Bandwidth	Comments
65536 (64KB)	45052	44.00	2.03GB/sec	Slower, but see conclusions.
1048576 (1MB)	722035	44.07	2.03GB/sec	

Conclusions: At first glance, the L2 cache test run looks slower than the movapd case. This is misleading. Using the movapd instruction left the L2 full of dirty cache lines that must one day be written to main memory. What's worse is the cache lines written to the L2 cache may have ousted other cache lines that might have been used in the future. These dirty cache lines will slow down future cache line reads (the cache line will need to be written before the new cache line can be read in). So, use movntpd for those cases where you truly will not be using the data in the future.

The main memory benchmark is also interesting. I can find no reason why this should be faster than the read contiguous data case, but it is. ***Why?***

Writing non-contiguous data

```
        mem = 4096                      ; Write 4KB
        cnt = mem/(4*128)               ; 128 bytes per iteration
        dist = 64
        sub     ebx, ebx
        mov     ecx, cnt
loop1:  movapd  [esi+0*dist], xmm1      ; Write 8 cache lines
        movapd  [esi+1*dist], xmm1
        movapd  [esi+2*dist], xmm1
        movapd  [esi+3*dist], xmm1
        movapd  [esi+4*dist], xmm1
        movapd  [esi+5*dist], xmm1
        movapd  [esi+6*dist], xmm1
        movapd  [esi+7*dist], xmm1
        lea     esi, [esi+16]           ; Same cache lines
        add     bl, 256/4               ; 4 inner loop iterations
        jnc     loop1
        lea     esi, [esi-4*16+8*dist]  ; Next set of 8 cache lines
        sub     ecx, 1
        jnz     loop1
        lea     esi, [esi-mem]
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Mem	Clocks	Clocks per	Bandwidth	Comments

		iteration		
4096 (4KB)	2845	88.25	2.02GB/sec	Much slower than contiguous writes
65536 (64KB)	45516	88.25	2.02GB/sec	Also slower than contiguous case.
1048576 (1MB)	1957866	239.00	0.75GB/sec	Slower than reading. Why?

Conclusions: Group your writes together!!! Failure to do so reduces throughput by 66%. Here each cache-line write takes 11 clocks instead of 14 in the contiguous case. This test below will tell us if write-combining is helping to some degree.

Now we'll do the same as above, but only writing to 16 bytes in each cache line.

```
        mem = 4096                      ; Write 4KB
        cnt = mem/(8*64)                ; 8 cache lines each iteration
        dist = 64
        mov     ecx, cnt
loop1:  movapd  [esi+0*dist], xmm1      ; Write 8 cache lines
        movapd  [esi+1*dist], xmm1
        movapd  [esi+2*dist], xmm1
        movapd  [esi+3*dist], xmm1
        movapd  [esi+4*dist], xmm1
        movapd  [esi+5*dist], xmm1
        movapd  [esi+6*dist], xmm1
        movapd  [esi+7*dist], xmm1
        lea     esi, [esi+8*dist]       ; Next set of 8 cache lines
        sub     ecx, 1
        jnz     loop1
        lea     esi, [esi-mem]
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Mem	Clocks	Clocks per iteration	Comments
4096 (4KB)	706	88.25	Same as timing above

Conclusions: This indicates that each write takes 11 clocks. Write-combining was of no value in the previous benchmark. I speculate the 14 clocks in the contiguous write case is broken down as follows, 11 for the write to L2 cache and 1 clock for each of the 3 movapd write-combinings that took place.

Cost of a TLB miss

This code reads one cache line from each 4KB page. The L1 and L2 cache line collisions are minimized so that we can analyze the cost of a TLB miss.

```
        tlbs = 32
        cnt = tlbs/8
        mov     ecx, cnt
        sub     eax, eax
```

```
loop1:  movapd  xmm1, [esi]               ; Read from a 4KB page
        lea     esi, [esi+4096+64]        ; Different line, next page
        add     al, 256/8                 ; 8 inner loops - for
        jnc     loop1                     ; perfect branch prediction
        sub     ecx, 1
        jnz     loop1
        lea     esi, [esi-tlbs*(4096+64)]
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Tlbs	Clocks	Clocks per TLB	Comments
32	82	2.56	
64	147	2.3	Timings vary widely
72	357	4.95	TLB miss penalty starting to kick in
128	2608	20.38	Should be no TLB hits

Conclusion: A TLB miss where the page table entry is in the L2 cache costs 18 clocks.

Real world programs – reading and writing contiguous data

The above examples are nice, but most applications read data, operate on it, and write it back.

```
        mem = 4096                        ; Work on 4KB
        cnt = mem/64                      ; 64 bytes per iteration
        mov     ecx, cnt
loop1:  movapd  xmm0, [esi]               ; Read one cache line
        movapd  xmm1, [esi+16]
        movapd  xmm2, [esi+32]
        movapd  xmm3, [esi+48]
        subpd   xmm0, xmm0                ; Operate on the data
        pxor    xmm1, xmm1
        subpd   xmm2, xmm2
        pxor    xmm3, xmm3
        movapd  [esi], xmm0               ; Write the cache line
        movapd  [esi+16], xmm1
        movapd  [esi+32], xmm2
        movapd  [esi+48], xmm3
        lea     esi, [esi+64]             ; Next cache line
        sub     ecx, 1
        jnz     loop1
        lea     esi, [esi-mem]
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Mem	Clocks	Clocks per 64 byte cache line	Bandwidth	Comments
4096 (4KB)	901	14.07	6.36GB/sec	Limited by write-through to L2 speed
65536 (64KB)	24669	24.09	3.72GB/sec	

1048576 (1MB)	1893992	115.60	0.78GB/sec	

Conclusions: We need a good explanation for the 24 clock number in the L2 cache timing. ***Ideas?***

If you want to write a program that is CPU bound instead of memory bandwidth bound, you must perform at least **116 clocks of ALU/FPU instructions** for every 64 bytes read! Obviously, it is very difficult to do this for applications whose data does not fit in the L2 cache.

Now let's try replacing the last 4 movapd instructions with movntpd:

Mem	Clocks	Clocks per 64 byte cache line	Bandwidth	Comments
1048576 (1MB)	1822550	111.24	0.81GB/sec	

Conclusion: Not much difference. This could be within our margin of error in timing or it could indicate that there is a slight penalty for reading into a dirty cache line as opposed to writing dirty cache lines immediately. Warning: Even though this benchmark suggests using movntpd, using movapd is probably advantageous. In programs that are not limited by memory bandwidth, you may well be able to completely hide the write time by using prefetch instructions. Just be aware in scheduling your prefetch instructions that the cache line you are reading may force a write prior to the read.

In coding one application, I thought it would be advantageous to use the clflush instruction to write out dirty cache lines during a period where the program was CPU bound (a kind of prefetcht1 instruction in reverse!) This turned out to be a losing move. Adding a "clflush [esi]" instruction after the 4 write instructions **balloons the timing to 2625967 clocks. Avoid clflush at all costs.**

Now what happens when we read from one area of memory and write to another:

```
        mem = 1048576                   ; Work on 1MB
        cnt = mem/64                    ; 64 bytes per iteration
        mov     ecx, cnt
loop1:  movapd  xmm0, [esi]             ; Read one cache line
        movapd  xmm1, [esi+16]
        movapd  xmm2, [esi+32]
        movapd  xmm3, [esi+48]
        subpd   xmm0, xmm0              ; Operate on the data
        pxor    xmm1, xmm1
        subpd   xmm2, xmm2
        pxor    xmm3, xmm3
        movntpd  [esi+mem], xmm0        ; Write the cache line
        movntpd  [esi+mem+16], xmm1
        movntpd  [esi+mem+32], xmm2
        movntpd  [esi+mem+48], xmm3
        lea     esi, [esi+64]           ; Next cache line
        sub     ecx, 1
        jnz     loop1
        lea     esi, [esi-mem]
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Mem	Clocks	Clocks per 64 byte cache line	Bandwidth	Comments
1048576 (1MB)	1680771	102.59	0.87GB/sec	

Conclusions: ***Why is this 10% faster than writing to the cache line just read in?***

A real world program – fast Fourier transform on a huge data set

I run a distributed computing search for large prime numbers at http://www.mersenne.org/prime.htm. The program performs fast Fourier transforms on large data sets (up to 4 million doubles – 32MB of data). The present program is all assembly language and needs to be upgraded with p4 optimized code. Using SSE2 instructions and being mindful of P4 cache considerations should yield a threefold performance boost. Hopefully, what I learn in constructing the basic building blocks of this FFT will yield some insights useful to other P4 assembly language programmers.

All you need to know about fast Fourier transforms is they perform a ton of floating point operations in a regular, but non-contiguous data access pattern.

We will analyze using the variations of the macro below. What it does is unimportant. It reads 8 XMM registers from 8 different cache lines, does stuff, and writes the new values back. Since it does 20 addpd and subpd instructions, a best case timing is 40 clocks.

```
test_macro MACRO R1,R2,R3,R4,R5,R6,R7,R8
        movapd  xmm0, R4                ;; R4
        movapd  xmm1, R8                ;; R8
        subpd   xmm0, xmm1              ;; new R8 = R4 - R8
        movapd  xmm2, R2                ;; R2
        movapd  xmm3, R6                ;; R6
        subpd   xmm2, xmm3              ;; new R6 = R2 - R6
        addpd   xmm1, R4                ;; new R4 = R4 + R8
         movapd xmm7, XMM_SQRTHALF      ;; square root of 1/2
         mulpd  xmm0, xmm7              ;; R8 = R8 * square root
        addpd   xmm3, R2                ;; new R2 = R2 + R6
         mulpd  xmm2, xmm7              ;; R6 = R6 * square root
        movapd  xmm4, R1                ;; R1
        movapd  xmm5, R5                ;; R5
        addpd   xmm5, xmm4              ;; new R1 = R1 + R5
         subpd  xmm3, xmm1              ;; R2 = R2 - R4 (final R4)
         multwo xmm1                    ;; R4 = R4 * 2
        movapd  xmm6, R3                ;; R3
        movapd  xmm7, R7                ;; R7
        addpd   xmm7, xmm6              ;; new R3 = R3 + R7
         subpd  xmm2, xmm0              ;; R6 = R6 - R8 (Real part)
         multwo xmm0                       ;; R8 = R8 * 2
         subpd  xmm5, xmm7                 ;; R1 = R1 - R3 (final R3)
         multwo xmm7                       ;; R3 = R3 * 2
        subpd   xmm4, R5                   ;; new R5 = R1 - R5
         addpd  xmm1, xmm3                 ;; R4 = R2 + R4 (new R2)
        addpd   xmm0, xmm2                 ;; R8 = R6 + R8 (Imag. part)
        subpd   xmm4, xmm2                 ;; R5 = R5 - R6 (final R7)
        multwo  xmm2                       ;; R6 = R6 * 2
        subpd   xmm6, R7                   ;; new R7 = R3 - R7
         addpd  xmm7, xmm5                 ;; R3 = R1 + R3 (new R1)
        subpd   xmm6, xmm0                 ;; R7 = R7 - R8 (final R8)
        multwo  xmm0                       ;; R8 = R8 * 2
        subpd   xmm7, xmm1                 ;; R1 = R1 - R2 (final R2)
        multwo  xmm1                       ;; R2 = R2 * 2
        movapd  R4, xmm3
        movapd  R3, xmm5
        addpd   xmm2, xmm4                 ;; R6 = R5 + R6 (final R5)
        addpd   xmm0, xmm6                 ;; R8 = R7 + R8 (final R6)
        addpd   xmm1, xmm7                 ;; R2 = R1 + R2 (final R1)
        movapd  R7, xmm4
        movapd  R8, xmm6
        movapd  R2, xmm7
        movapd  R5, xmm2
        movapd  R6, xmm0
        movapd  R1, xmm1
        ENDM

test_disp MACRO mac, s, d1, d2, d4
        mac [s],[s+d1],[s+d2],[s+d2+d1],[s+d4],[s+d4+d1],[s+d4+d2],[s+d4+d2+d1]
        ENDM

multwo  MACRO   r
        mulpd   r, XMM_TWO
        ENDM
```

I time the above macro 3 ways, using 128 bytes, 2KB, or 64KB. The 128 bytes and 2KB timings should give best-case scenarios – operating out of the fast L1 cache. The 2KB example will imitate the loop constructs typical in an FFT program. The 64KB example is more like how I'll be using the macro – operating out of the L2 cache.

```
use_128_bytes:
        mov     edx, macro_count
loop2:  test_disp test_macro, esi, 64, 128, 256
        sub     edx, 1                    ; Check loop counter
        jnz     loop2                     ; Loop if necessary
```

```
use_2KB:
        mov     edx, macro_count/16
        sub     eax, eax
loop3:  test_disp test_macro, esi, 64, 128, 256
        lea     esi, [esi+16]
        add     al, 256/4               ; 4 XMMs per cache line
        jnc     loop3
        lea     esi, [esi-4*16+512]
        add     ah, 256/4               ; 4 512-byte blocks in 2KB
        jnc     loop3
        lea     esi, [esi-4*512]
        sub     edx, 1                  ; Check loop counter
        jnz     loop3                   ; Loop if necessary

use_64KB:
        mov     edx, macro_count/512
        sub     eax, eax
loop4:  test_disp test_macro, esi, 64, 128, 256
        lea     esi, [esi+16]
        add     al, 256/4               ; 4 XMMs per cache line
        jnc     loop4
        lea     esi, [esi-4*16+512]
        add     ah, 256/128             ; 128 512-byte blocks in 64KB
        jnc     loop4
        lea     esi, [esi-128*512]
        sub     edx, 1                  ; Check loop counter
        jnz     loop4                   ; Loop if necessary
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Bytes processed	Average clocks per test_macro	Comments
128	106.2	Limited by write-through to L2 speed
2KB	88.8	
64KB	104.0	

Conclusions: No where near the 40 clocks we hoped for. The 2KB case clearly seems limited by the 11 clocks per L1 write-through operation. The 128 byte example is probably incurring some store-forwarding penalties. The 64KB example is unable to hide the slower L2 cache latencies. This might be due to using bandwidth for L1 write-throughs?

In any event, it seems my FFT cannot be an in-place FFT. Fortunately, it is not hard to modify the code to read from 8 cache lines and write contiguously to a small memory block. Then later read from the small memory block and write contiguously back to my FFT data area.

The new macro is as such:

```
test_macro MACRO dstreg,R1,R2,R3,R4,R5,R6,R7,R8
```

```
movapd  xmm0, R4                ;; R4
movapd  xmm1, R8                ;; R8
subpd   xmm0, xmm1              ;; new R8 = R4 - R8
movapd  xmm2, R2                ;; R2
movapd  xmm3, R6                ;; R6
subpd   xmm2, xmm3              ;; new R6 = R2 - R6
addpd   xmm1, R4                ;; new R4 = R4 + R8
 movapd xmm7, XMM_SQRTHALF      ;; square root of 1/2
 mulpd  xmm0, xmm7              ;; R8 = R8 * square root
addpd   xmm3, R2                ;; new R2 = R2 + R6
 mulpd  xmm2, xmm7              ;; R6 = R6 * square root
movapd  xmm4, R1                ;; R1
movapd  xmm5, R5                ;; R5
addpd   xmm5, xmm4              ;; new R1 = R1 + R5
 subpd  xmm3, xmm1              ;; R2 = R2 - R4 (final R4)
 multwo xmm1                    ;; R4 = R4 * 2
movapd  xmm6, R3                ;; R3
movapd  xmm7, R7                ;; R7
addpd   xmm7, xmm6              ;; new R3 = R3 + R7
 subpd  xmm2, xmm0              ;; R6 = R6 - R8 (Real part)
 multwo xmm0                    ;; R8 = R8 * 2
 subpd  xmm5, xmm7              ;; R1 = R1 - R3 (final R3)
 multwo xmm7                    ;; R3 = R3 * 2
subpd   xmm4, R5                ;; new R5 = R1 - R5
 addpd  xmm1, xmm3              ;; R4 = R2 + R4 (new R2)
addpd   xmm0, xmm2              ;; R8 = R6 + R8 (Imag. part)
subpd   xmm4, xmm2              ;; R5 = R5 - R6 (final R7)
multwo  xmm2                    ;; R6 = R6 * 2
subpd   xmm6, R7                ;; new R7 = R3 - R7
 addpd  xmm7, xmm5              ;; R3 = R1 + R3 (new R1)
subpd   xmm6, xmm0              ;; R7 = R7 - R8 (final R8)
multwo  xmm0                    ;; R8 = R8 * 2
subpd   xmm7, xmm1              ;; R1 = R1 - R2 (final R2)
multwo  xmm1                    ;; R2 = R2 * 2
movapd  [dstreg+3*16], xmm3
movapd  [dstreg+2*16], xmm5
addpd   xmm2, xmm4              ;; R6 = R5 + R6 (final R5)
addpd   xmm0, xmm6              ;; R8 = R7 + R8 (final R6)
addpd   xmm1, xmm7              ;; R2 = R1 + R2 (final R1)
movapd  [dstreg+6*16], xmm4
movapd  [dstreg+7*16], xmm6
movapd  [dstreg+1*16], xmm7
movapd  [dstreg+4*16], xmm2
movapd  [dstreg+5*16], xmm0
movapd  [dstreg+0*16], xmm1
ENDM
```

```
test_disp MACRO mac, s, dst, d1, d2, d4
        mac dst,[s],[s+d1],[s+d2],[s+d2+d1],[s+d4],[s+d4+d1],[s+d4+d2],[s+d4+d2+d1]
        ENDM

use_128_bytes:
        mov     edx, macro_count
loop2:  test_disp test_macro, esi, ebp, 64, 128, 256
        sub     edx, 1                  ; Check loop counter
        jnz     loop2                   ; Loop if necessary

use_2KB:
        mov     edx, macro_count/16
        sub     eax, eax
loop3:  test_disp test_macro, esi, ebp, 64, 128, 256
        lea     esi, [esi+16]
        lea     ebp, [ebp+128]
        add     al, 256/4               ; 4 XMMs per cache line
        jnc     loop3
        lea     esi, [esi-4*16+512]
        lea     ebp, [ebp-4*128]
        add     ah, 256/4               ; 4 512-byte blocks in 2KB
        jnc     loop3
        lea     esi, [esi-4*512]
        sub     edx, 1                  ; Check loop counter
        jnz     loop3                   ; Loop if necessary

use_64KB:
        mov     edx, macro_count/512
        sub     eax, eax
loop4:  test_disp test_macro, esi, ebp, 64, 128, 256
        lea     esi, [esi+16]
        lea     ebp, [ebp+128]
        add     al, 256/4               ; 4 XMMs per cache line
        jnc     loop4
        lea     esi, [esi-4*16+512]
        lea     ebp, [ebp-4*128]
        add     ah, 256/128             ; 128 512-byte blocks in 64KB
        jnc     loop4
        lea     esi, [esi-128*512]
        sub     edx, 1                  ; Check loop counter
        jnz     loop4                   ; Loop if necessary
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Bytes processed	Average clocks per test_macro	Comments
128	40.0	Perfect!
2KB	40.0	My funny looping constructs cost nothing!

64KB	45.9	Still some work to be done.

Conclusions: Excellent progress – an impressive display by the P4 instruction scheduler. We still need to work on ways to reduce the penalty of operating out of the L2 cache.

The following examples detail my attempts at reducing the 5.9 clock penalty. Our earlier examples suggest preloading the cache lines into the L1 cache will be helpful.

```
use_64KB:
        mov     edx, macro_count/512
        sub     eax, eax
loop4:  test_disp test_macro, esi, ebp, 64, 128, 256
        lea     esi, [esi+16]
        lea     ebp, [ebp+128]
        add     al, 256/4               ; 4 XMMs per cache line
        jnc     loop4
        mov     ebx, [esi-4*16+512+512] ; Preload cache lines into L1
        mov     ebx, [esi-4*16+512+512+128]
        mov     ebx, [esi-4*16+512+512+256]
        mov     ebx, [esi-4*16+512+512+384]
        mov     ebx, [esi-4*16+512+512+64]
        mov     ebx, [esi-4*16+512+512+128+64]
        mov     ebx, [esi-4*16+512+512+256+64]
        mov     ebx, [esi-4*16+512+512+384+64]
        lea     esi, [esi-4*16+512]
        lea     ebp, [ebp-4*128]
        add     ah, 256/128             ; 128 512-byte blocks in 64KB
        jnc     loop4
        lea     esi, [esi-128*512]
        sub     edx, 1                  ; Check loop counter
        jnz     loop4                   ; Loop if necessary
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Bytes processed	Average clocks per test_macro	Comments
64KB	45.9	No change

I next tried moving the

```
        lea     esi, [esi+16]
```

as early as possible. That is, after the

```
        subpd   xmm6, R7                ;; new R7 = R3 - R7
```

instruction. This should make little or no difference since the P4 should have no difficulty scheduling the lea instruction. However, I wanted to give the processor every opportunity to schedule the preloads as soon as possible.

Timing for the above on a 1.4GHz P4 with PC600 memory:

Bytes processed	Average clocks	Comments

	per test_macro	
64KB	42.7	Wow! A theoretically useless change saves 3 clocks

Next I tried to distribute the L1 cache preloads more evenly at the cost of using another register.

```
use_64KB:
        mov     edx, macro_count/512
        sub     eax, eax
        lea     ecx, [esi+640]          ; Preload 10 cache lines ahead
loop4:  test_disp test_macro, esi, ebp, 64, 128, 256
        mov     ebx, [ecx]              ; Preload cache lines into L1
        mov     ebx, [ecx+64]
        lea     ecx, [ecx+128]
        lea     esi, [esi+16]
        lea     ebp, [ebp+128]
        add     al, 256/4               ; 4 XMMs per cache line
        jnc     loop4
        lea     esi, [esi-4*16+512]
        lea     ebp, [ebp-4*128]
        add     ah, 256/128             ; 128 512-byte blocks in 64KB
        jnc     loop4
        lea     esi, [esi-128*512]
        lea     ecx, [ecx-128*512]
        sub     edx, 1                  ; Check loop counter
        jnz     loop4                   ; Loop if necessary
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Bytes processed	Average clocks per test_macro	Comments
64KB	41.1	As coded above
64KB	40.9	With “lea esi, [esi+16]” moved as described earlier

Conclusions: While I’ve not figured out a way to get the perfect 40.0 clocks, I have gotten close. While I don’t have an explanation for the observed behavior, we can conclude that programs should endeavor to compute addresses as early as possible and preload the L1 cache if possible.

A real world program – fast Fourier transform on a huge data set (continued)

We’ve shown that the most basic of FFT building locks can come very close to perfect scheduling running out of the L2 cache. Now we’ll organize put together many executions of the test_macro building block that emulates how my FFT operates on 64KB of data. We’ll test this running on the same 64KB over and over again as well as operating on contiguous 64KB blocks of main memory. We hope to use prefetch to eliminate all waiting for main memory.

Note: I’ve changed the test_macro below to accept an argument specifying how much the source register should be incremented.

```
test_macro MACRO srcreg,inc,dstreg,R1,R2,R3,R4,R5,R6,R7,R8
        movapd  xmm0, R4                ;; R4
        movapd  xmm1, R8                ;; R8
        subpd   xmm0, xmm1              ;; new R8 = R4 - R8
        movapd  xmm2, R2                ;; R2
        movapd  xmm3, R6                ;; R6
        subpd   xmm2, xmm3              ;; new R6 = R2 - R6
        addpd   xmm1, R4                ;; new R4 = R4 + R8
         movapd xmm7, XMM_SQRTHALF      ;; square root of 1/2
         mulpd  xmm0, xmm7              ;; R8 = R8 * square root
        addpd   xmm3, R2                ;; new R2 = R2 + R6
         mulpd  xmm2, xmm7              ;; R6 = R6 * square root
        movapd  xmm4, R1                ;; R1
        movapd  xmm5, R5                ;; R5
        addpd   xmm5, xmm4              ;; new R1 = R1 + R5
         subpd  xmm3, xmm1              ;; R2 = R2 - R4 (final R4)
         multwo xmm1                    ;; R4 = R4 * 2
        movapd  xmm6, R3                ;; R3
        movapd  xmm7, R7                ;; R7
        addpd   xmm7, xmm6              ;; new R3 = R3 + R7
         subpd  xmm2, xmm0              ;; R6 = R6 - R8 (Real part)
         multwo xmm0                    ;; R8 = R8 * 2
         subpd  xmm5, xmm7              ;; R1 = R1 - R3 (final R3)
         multwo xmm7                    ;; R3 = R3 * 2
        subpd   xmm4, R5                ;; new R5 = R1 - R5
         addpd  xmm1, xmm3              ;; R4 = R2 + R4 (new R2)
        addpd   xmm0, xmm2              ;; R8 = R6 + R8 (Imag. part)
        subpd   xmm4, xmm2              ;; R5 = R5 - R6 (final R7)
        multwo  xmm2                    ;; R6 = R6 * 2
        subpd   xmm6, R7                ;; new R7 = R3 - R7
        IF inc NE 0
        add     srcreg, inc
        ENDIF
         addpd  xmm7, xmm5              ;; R3 = R1 + R3 (new R1)
        subpd   xmm6, xmm0              ;; R7 = R7 - R8 (final R8)
        multwo  xmm0                    ;; R8 = R8 * 2
        subpd   xmm7, xmm1              ;; R1 = R1 - R2 (final R2)
        multwo  xmm1                    ;; R2 = R2 * 2
        addpd   xmm2, xmm4              ;; R6 = R5 + R6 (final R5)
        addpd   xmm0, xmm6              ;; R8 = R7 + R8 (final R6)
        addpd   xmm1, xmm7              ;; R2 = R1 + R2 (final R1)
        movapd  [dstreg+0*16], xmm1
        movapd  [dstreg+1*16], xmm7
        movapd  [dstreg+2*16], xmm5
        movapd  [dstreg+3*16], xmm3
        movapd  [dstreg+4*16], xmm2
        movapd  [dstreg+5*16], xmm0
        movapd  [dstreg+6*16], xmm4
        movapd  [dstreg+7*16], xmm6
        ENDM

test_disp MACRO mac, s, inc, dst, d1, d2, d4
    mac s,inc,dst,[s],[s+d1],[s+d2],[s+d2+d1],[s+d4],[s+d4+d1],[s+d4+d2],[s+d4+d2+d1]
        ENDM
```

Here’s the new macro that executes the above macro 4608 times. Don’t worry about the details too much. All you really need to know is it executes the macro 512 times in place, then 4 times it executes the macro 1024 times – copying data to a small scratch area and back to the source area.

```
xpass2 MACRO
        LOCAL   b1b, b2a, b2b, b3b, b4a, b4b, b5b
        LOCAL   b6a, b6b, b7b, b8a, b8b, b9b, bab

        sub    ecx, ecx

; FFT levels 1-2 in place

        mov     eax, 512                ;; 512 iterations
b1b:    test_disp test_macro, esi, 0, esi, 16, 32, 64
        lea     esi, [esi+128]
        sub     eax, 1                  ;; Test inner loop counter
        jnz     b1b                     ;; Iterate if necessary

; FFT levels 3-4 to scratch area

        lea     esi, [esi-512*128]      ;; Next source pointer
b2a:    mov     ebp, OFFSET XMM_SCRATCH_AREA
b2b:    test_disp test_macro, esi, 32, ebp, 16, 256, 512
        lea     ebp, [ebp+128]
        add     al, 256/4               ;; Test inner loop counter
        jnc     b2b                     ;; Iterate if necessary
        lea     esi, [esi-4*32+4*256]   ;; Next source pointer
        add     ah, 256/4               ;; Test inner loop counter
        jnc     b2b                     ;; Iterate if necessary

; FFT levels 5-6 from scratch area

        lea     esi, [esi-16*256]       ;; Next source pointer
        mov     ebp, OFFSET XMM_SCRATCH_AREA
b3b:    test_disp test_macro, ebp, 128, esi, 16, 32, 64
        lea     esi, [esi+256]
        add     al, 256/16
        jnc     b3b
        add     cl, 256/16              ;; 16 iterations
        jnc     b2a                     ;; Iterate if necessary
        lea     esi, [esi-256*256+128]  ;; Next source pointer
        add     ch, 256/2               ;; 2 iterations
        jnc     b2a                     ;; Iterate if necessary

; FFT levels 7-8 to scratch area

        lea     esi, [esi-2*128]        ;; Next source pointer
b4a:    mov     ebp, OFFSET XMM_SCRATCH_AREA
b4b:    test_disp test_macro, esi, 32, ebp, 16, 256, 512
        lea     ebp, [ebp+128]
        add     al, 256/4               ;; Test inner loop counter
        jnc     b4b                     ;; Iterate if necessary
        lea     esi, [esi-4*32+4*256]   ;; Next source pointer
        add     ah, 256/4               ;; Test inner loop counter
        jnc     b4b                     ;; Iterate if necessary

; FFT levels 9-10 from scratch area

        lea     esi, [esi-16*256]       ;; Next source pointer
        mov     ebp, OFFSET XMM_SCRATCH_AREA
b5b:    test_disp test_macro, ebp, 128, esi, 16, 32, 64
        lea     esi, [esi+256]
        add     al, 256/16
        jnc     b5b

        add     cl, 256/16              ;; 16 iterations
        jnc     b4a                     ;; Iterate if necessary
        lea     esi, [esi-256*256+128]  ;; Next source pointer
        add     ch, 256/2               ;; 2 iterations
        jnc     b4a                     ;; Iterate if necessary

; FFT levels 7-8 to scratch area

        lea     esi, [esi-2*128]        ;; Next source pointer
b6a:    mov     ebp, OFFSET XMM_SCRATCH_AREA
b6b:    test_disp test_macro, esi, 32, ebp, 16, 256, 512
        lea     ebp, [ebp+128]
        add     al, 256/4               ;; Test inner loop counter
        jnc     b6b                     ;; Iterate if necessary
        lea     esi, [esi-4*32+4*256]   ;; Next source pointer
        add     ah, 256/4               ;; Test inner loop counter
        jnc     b6b                     ;; Iterate if necessary

; FFT levels 5-6 from scratch area
```

```
        lea     esi, [esi-16*256]       ;; Next source pointer
        mov     ebp, OFFSET XMM_SCRATCH_AREA
b7b:    test_disp test_macro, ebp, 128, esi, 16, 32, 64
        lea     esi, [esi+256]
        add     al, 256/16
        jnc     b7b

        add     cl, 256/16              ;; 16 iterations
        jnc     b6a                     ;; Iterate if necessary
        lea     esi, [esi-256*256+128]  ;; Next source pointer
        add     ch, 256/2               ;; 2 iterations
        jnc     b6a                     ;; Iterate if necessary

; FFT levels 3-4 to scratch area

        lea     esi, [esi-2*128]        ;; Next source pointer
b8a:    mov     ebp, OFFSET XMM_SCRATCH_AREA
b8b:    test_disp test_macro, esi, 32, ebp, 16, 256, 512
        lea     ebp, [ebp+128]
        add     al, 256/4               ;; Test inner loop counter
        jnc     b8b                     ;; Iterate if necessary
        lea     esi, [esi-4*32+4*256]   ;; Next source pointer
        add     ah, 256/4               ;; Test inner loop counter
        jnc     b8b                     ;; Iterate if necessary

; FFT levels 1-2 from scratch area

        lea     esi, [esi-16*256]       ;; Next source pointer
        mov     ebp, OFFSET XMM_SCRATCH_AREA
b9b:    test_disp test_macro, ebp, 128, esi, 16, 32, 64
        lea     esi, [esi+256]
        add     al, 256/16
        jnc     b9b

        add     cl, 256/16              ;; 16 iterations
        jnc     b8a                     ;; Iterate if necessary
        lea     esi, [esi-256*256+128]  ;; Next source pointer
        add     ch, 256/2               ;; 2 iterations
        jnc     b8a                     ;; Iterate if necessary

        lea     esi, [esi-2*128]        ;; Restore source pointer
        ENDM
```

The macro is timed two ways:

```
in_place:
        mov     edx, 100
loop1:  xpass2
        sub     edx, 1
        jnz     loop1

main_memory:
        mov     edx, 100
loop2:  xpass2
        lea     esi, [esi+65536]
        sub     edx, 1
        jnz     loop2
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Memory access pattern	Clocks for each xpass2	Clocks per 64 byte cache line processed	Clocks per test_macro call	Comments
64KB in place	212780	207.8	46.2	Close to expected
64KB main mem.	303935	296.8	66.0	

Conclusion: The 46.2 clocks per test_macro closely matches the timings above where the L1 cache was not preloaded. Note that one-third of the test_macro calls (those operating from the small scratch area) should be operating on data in the L1 cache. However, preloading the L1 cache for the other two-thirds of the test_macro calls may well get us close to the previous section's 40.8 clocks.

Also, note the 89 clocks spent reading and writing main memory. This makes sense as it translates to (64 bytes written + 64 bytes read) / 89 clocks * 1.4GHz = 2GB/sec. Using prefetch, it should be possible to hide these 89 clocks among the 207.8 clocks of CPU activity.

Now we modify the in place section slightly allowing moving the computation of the new esi value earlier in the loop:

```
; FFT levels 1-2 in place

        mov     eax, 512                ;; 512 iterations
        mov     ebp, esi
b1b:    test_disp test_macro, esi, 128, ebp, 16, 32, 64
        lea     ebp, [ebp+128]
        sub     eax, 1                  ;; Test inner loop counter
        jnz     b1b                     ;; Iterate if necessary
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Memory access pattern	Clocks for each xpass2	Clocks per 64 byte cache line processed	Clocks per test_macro call	Comments
64KB in place	207614	202.7	45.1	Wow.
64KB main mem.	303349	296.2	65.8	

Conclusion: This is amazing. The difference 5166 clock difference must come from the first 512 test_macro calls. That means, there is a 10 clock benefit to using two addressing registers and computing the new source register value as soon as possible.

Something is fishy with the above result. When we analyzed moving the lea instruction earlier we never saw a 10 clock benefit. I tried several ideas before this rather startling discovery. Add a nop here:

```
; FFT levels 7-8 to scratch area

        nop
        lea     esi, [esi-2*128]        ;; Next source pointer
b4a:    mov     ebp, OFFSET XMM_SCRATCH_AREA
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Memory access pattern	Clocks for each xpass2	Clocks per 64 byte cache line processed	Clocks per test_macro call	Comments
64KB in place	203266	198.5	44.1	Wow.
64KB main mem.	300511	293.5	65.2	

Conclusions: Apparently there is an undocumented penalty dealing with code alignment!

Further investigation shows that if two jump instructions are 15 bytes apart and the first jump instruction starts on an odd address, then the BTB fails in predicting the second jump instruction. When I replace the four occurrences of:

```
        jnc     b8b                     ;; Iterate if necessary
        lea     esi, [esi-4*32+4*256]   ;; Next source pointer
        add     ah, 256/4               ;; Test inner loop counter
        jnc     b8b                     ;; Iterate if necessary
```

with:

```
        jnc     b8b                     ;; Iterate if necessary
        nop
        lea     esi, [esi-4*32+4*256]   ;; Next source pointer
        add     ah, 256/4               ;; Test inner loop counter
        jnc     b8b                     ;; Iterate if necessary
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Memory access pattern	Clocks for each xpass2	Clocks per 64 byte cache line processed	Clocks per test_macro call	Comments
64KB in place	199998	195.3	43.4	
64KB main mem.	297440	290.5	64.5	

Conclusions: Despite the P4 optimization manual saying that the P4 is less affected by code alignment than earlier processors, there are still circumstances where code alignment can have a major impact on performance.

However, my findings above are pretty weird. I can't imagine a design decision Intel could have made that would result in the 15-byte even/odd anomaly. Obviously, more research is needed on the implications of the P4's branch prediction algorithm.

Now, we'll try to use prefetch to reduce the main memory timing. Here is my first attempt:

```
xpass2 MACRO
        LOCAL   b1b, b2a, b2b, b3b, b4a, b4b, b5b
        LOCAL   b6a, b6b, b7b, b8a, b8b, b9b, bab

        sub     ecx, ecx

;; Touch the TLBs we are about to FFT so they are marked most
;; recently used. Then load the TLBs for the next 64KB FFT chunk
;; so that prefetcht1 works.

bab:    mov     eax, [esi]
        lea     esi, [esi+4096]         ;; Next page
        add     cl, 256/32              ;; 32 pages
        jnc     bab                     ;; Iterate if necessary
        lea     esi, [esi-32*4096]      ;; Next source pointer

; FFT levels 1-2 in place

        mov     eax, 512                ;; 512 iterations
        mov     ebp, esi
b1b:    test_disp test_macro, esi, 128, ebp, 16, 32, 64
        lea     ebp, [ebp+128]
        sub     eax, 1                  ;; Test inner loop counter
        jnz     b1b                     ;; Iterate if necessary

; FFT levels 3-4 to scratch area

        lea     esi, [esi-512*128]      ;; Next source pointer
b2a:    mov     ebp, OFFSET XMM_SCRATCH_AREA
b2b:    test_disp test_macro, esi, 32, ebp, 16, 256, 512
        lea     ebp, [ebp+128]
        add     al, 256/4               ;; Test inner loop counter
        jnc     b2b                     ;; Iterate if necessary
        prefetcht1 [esi-4*32+65536]
        lea     esi, [esi-4*32+4*256]   ;; Next source pointer
        add     cl, 256/4               ;; Test inner loop counter
        jnc     b2b                     ;; Iterate if necessary
```

```
; FFT levels 5-6 from scratch area

        lea     esi, [esi-16*256]       ;; Next source pointer
        mov     ebp, OFFSET XMM_SCRATCH_AREA
b3b:    test_disp test_macro, ebp, 128, esi, 16, 32, 64
        lea     esi, [esi+256]
        add     al, 256/16
        jnc     b3b

        add     ah, 256/16              ;; 16 iterations
        jnc     b2a                     ;; Iterate if necessary
        lea     esi, [esi-256*256+128]  ;; Next source pointer
        add     ch, 256/2               ;; 2 iterations
        jnc     b2a                     ;; Iterate if necessary

; FFT levels 7-8 to scratch area

        lea     esi, [esi-2*128]        ;; Next source pointer
b4a:    mov     ebp, OFFSET XMM_SCRATCH_AREA
b4b:    test_disp test_macro, esi, 32, ebp, 16, 256, 512
        lea     ebp, [ebp+128]
        add     al, 256/4               ;; Test inner loop counter
        jnc     b4b                     ;; Iterate if necessary
        prefetcht1 [esi-4*32+65536+256]
        lea     esi, [esi-4*32+4*256]   ;; Next source pointer
        add     cl, 256/4               ;; Test inner loop counter
        jnc     b4b                     ;; Iterate if necessary

; FFT levels 9-10 from scratch area

        lea     esi, [esi-16*256]       ;; Next source pointer
        mov     ebp, OFFSET XMM_SCRATCH_AREA
b5b:    test_disp test_macro, ebp, 128, esi, 16, 32, 64
        lea     esi, [esi+256]
        add     al, 256/16
        jnc     b5b

        add     ah, 256/16              ;; 16 iterations
        jnc     b4a                     ;; Iterate if necessary
        lea     esi, [esi-256*256+128]  ;; Next source pointer
        add     ch, 256/2               ;; 2 iterations
        jnc     b4a                     ;; Iterate if necessary

; FFT levels 7-8 to scratch area

        lea     esi, [esi-2*128]        ;; Next source pointer
```

```
b6a:    mov     ebp, OFFSET XMM_SCRATCH_AREA
b6b:    test_disp test_macro, esi, 32, ebp, 16, 256, 512
        lea     ebp, [ebp+128]
        add     al, 256/4                ;; Test inner loop counter
        jnc     b6b                      ;; Iterate if necessary
        prefetcht1 [esi-4*32+65536+512]
        lea     esi, [esi-4*32+4*256]    ;; Next source pointer
        add     cl, 256/4                ;; Test inner loop counter
        jnc     b6b                      ;; Iterate if necessary

; FFT levels 5-6 from scratch area

        lea     esi, [esi-16*256]        ;; Next source pointer
        mov     ebp, OFFSET XMM_SCRATCH_AREA
b7b:    test_disp test_macro, ebp, 128, esi, 16, 32, 64
        lea     esi, [esi+256]
        add     al, 256/16
        jnc     b7b

        add     ah, 256/16               ;; 16 iterations
        jnc     b6a                      ;; Iterate if necessary
        lea     esi, [esi-256*256+128]   ;; Next source pointer
        add     ch, 256/2                ;; 2 iterations
        jnc     b6a                      ;; Iterate if necessary

; FFT levels 3-4 to scratch area

        lea     esi, [esi-2*128]         ;; Next source pointer
b8a:    mov     ebp, OFFSET XMM_SCRATCH_AREA
b8b:    test_disp test_macro, esi, 32, ebp, 16, 256, 512
        lea     ebp, [ebp+128]
        add     al, 256/4                ;; Test inner loop counter
        jnc     b8b                      ;; Iterate if necessary
        prefetcht1 [esi-4*32+65536+768]
        lea     esi, [esi-4*32+4*256]    ;; Next source pointer
        add     cl, 256/4                ;; Test inner loop counter
        jnc     b8b                      ;; Iterate if necessary

; FFT levels 1-2 from scratch area

        lea     esi, [esi-16*256]        ;; Next source pointer
        mov     ebp, OFFSET XMM_SCRATCH_AREA
b9b:    test_disp test_macro, ebp, 128, esi, 16, 32, 64
        lea     esi, [esi+256]
        add     al, 256/16
        jnc     b9b
        add     ah, 256/16              ;; 16 iterations
        jnc     b8a                     ;; Iterate if necessary
        lea     esi, [esi-256*256+128]  ;; Next source pointer
        add     ch, 256/2               ;; 2 iterations
        jnc     b8a                     ;; Iterate if necessary

        lea     esi, [esi-2*128]        ;; Restore source pointer
        ENDM
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Memory access pattern	Clocks for each xpass2	Clocks per 64 byte cache line processed	Clocks per test_macro call	Comments
64KB in place	199117	194.5	43.2	
64KB main mem.	222513	217.3	48.3	

Since 128KB should easily fit in the cache and the prefetch instructions are spaced a good ways apart (at least 160 clocks), it is a little surprising that the prefetch does not completely eliminate the penalties of main memory access.

Next, I changed the first loop to prefetch lines that weren't prefetched as hoped in processing the previous 64KB:

```
b1b:    test_disp test_macro, esi, 128, ebp, 16, 32, 64
```

is replaced with

```
b1b:    prefetcht1 [esi+8*128]
        test_disp test_macro, esi, 128, ebp, 16, 32, 64
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Memory access pattern	Clocks for each xpass2	Clocks per 64 byte cache line processed	Clocks per test_macro call	Comments
64KB in place	199386	194.7	43.3	
64KB main mem.	214751	209.7	46.6	

Trace cache and branch prediction

I created a very simple loop and timed both the best case and average case of 200 calls to this code. The results are quite puzzling:

```
        mov     edx, 16384
loopy:  subsd   xmm0, xmm0
        subsd   xmm0, xmm0
        sub     edx, 1
        jnz     loopy
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Best case	Average case	Comments
131280	~190000	The average varies quite a bit.

Conclusions: Why would we not get 131,280 every time this code is executed. On the second through 200th calls the entire loop should be in the trace cache.

Next, I tried flushing the trace cache by using self-modifying code prior to entering the loop:

```
trace_cache_flush MACRO
        LOCAL   rrr
rrr:    mov     ds:BYTE PTR rrr, 0C6h
        ENDM

        mov     edx, 16384
        trace_cache_flush
loopy:  subsd   xmm0, xmm0
        subsd   xmm0, xmm0
        sub     edx, 1
        jnz     loopy
```

Timing for the above on a 1.4GHz P4 with PC600 memory:

Best case	Average case	Comments
131780	~132400	Much more consistent results

Conclusion: There is some kind of flaw in the trace cache design. I've tried adding nops to the loop (when not flushing the trace cache) to see if more could be learned. Depending on how many nops are added, you will get better averages, but in no case is the result as good as flushing the trace cache.

If you have loops that are not giving you consistent timings, this might be the cause.

Tricks with SSE floating point exponents

In my FFT macros, I often need to multiply an XMM register by two. At first I thought of two options:

```
     ADDPD     xmm, xmm            ;; Add register to itself
```

Or

```
     XMM_TWO   DQ   2.0, 2.0
     MULPD     xmm, XMM_TWO        ;; Multiply by a global variable
```

I later thought of a third option:

```
     XMM_INC_EXP DB 1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0
     PADDB     xmm, XMM_INC_EXP    ;; Add 1 to each exponent!
```

In my case the ADDPD option was unattractive because I was already doing a lot of ADDPDs and they can only be issued every other clock cycle. The PADDB instruction is an attractive alternative to the MULPD instruction in that it has a shorter latency and makes it easier for the CPU to schedule other MULPD instructions. Note: I haven't tested the above yet!

There are probably other opportunities for such optimizations.