Speeding up Memory Copy

Jan Wassenberg

2006-03-26

Contents

1 Introduction **1**

2 Theory **2**

3 Implementation **3**

4 Performance Review **5**

4.1 Benchmark 6

4.2 Results 6

5 Conclusion **10**

A Adding to Visual C++ Projects **10**

B Source Code **10**

1 Introduction

This technical report shows how to significantly speed up memory transfers of all sizes by means of hand-tuned assembly language code. Copying is time-critical in certain applications, e.g. file caches (in cases where zero-copy IO isn't possible) or chess transposition tables. Unfortunately, the ANSI C89 `memcpy` implementation shipping with current compilers (e.g. Visual C++ 2003 and others) isn't nearly efficient enough – it can only make use of about one third of DDR PC2100 memory's peak bandwidth. One reason for this is a lack of updates for modern CPUs. In the case of VC7.1's code, certain obsolete Pentium U/V pipe optimizations are still visible. Also, the C runtime library cannot assume newer CPU features (e.g. SSE) to be present. Checking for them can slow down the implementation on older CPUs – ironically those most in need of optimization.

Our solution is to write a replacement for this function, using specialized transfer instructions where available and taking the cache into account. The rest of this report is organized as follows: A brief introduction to memory copying techniques is given and our algorithm is described. Performance is then compared with that of the current `memcpy`.

2 Theory

Due to the complex IA-32 / "x86" instruction set, there are several ways to copy a variable number of bytes. We briefly introduce them:

- series of `MOVS` instructions. Each one copies 1, 2 or 4 bytes; the `ESI` register gives the source address and `EDI` the destination. The transfer size is influenced by jumping midway into the series of instructions; since each one is a one-byte opcode, the address is EndOfSeries − NumberOfBytesToCopy. This technique is obviously only workable for small transfer sizes.
- `REP MOVS`. Repeats `MOVS` according to the count in `ECX`. This is typically what `memcpy` intrinsics map to.
- unrolled loop of `MOV`. Copies 4 bytes per instruction and is more efficient than `MOVS`-based approaches due to decode considerations.
- MMX `MOVQ`. Copies 8 bytes at a time and therefore makes full use of the 64-bit bus.
- FPU `FILD/FISTP`. Copies 8 bytes at a time but is inexplicably slow on Pentium IIIs.
- SSE `MOVNTQ`, together with software or block prefetching. This instruction avoids 'polluting' the cache with data that will never be used again. Prefetching also pulls data into the cache before it's needed, thereby reducing memory latency.

There is no single best approach for copying – we must choose among these techniques at runtime depending on transfer size or possibly other factors (alignment).

AMD have provided an implementation of the above; refer to their white paper (AMD1) and presentation (AMD2) for details. Unfortunately that code has two flaws: Performance for small transfer sizes is seriously lacking. This is due to unnecessary `SFENCE/EMMS` instructions being invoked as well as several conditional jumps in the implementation. A further disadvantage is that it requires SSE, i.e. only Pentium III, Athlon XP or above processors are supported.

Our contribution will be to introduce solutions to these problems that result in a more generally useful implementation. The above copy techniques can be packaged in interchangeable 'building blocks'. Organizing them as such allows us to easily rearrange them to reduce conditional branches (a major performance issue for all superscalar CPUs). We optimize for small transfer sizes and strive to make them run in a straight line without a single jump. Rationale is that the cost of jumps will be amortized over large transfer sizes anyway.

The real trick is to combine CPU feature detection with choosing the copy technique depending on size. The SSE-enabled codepaths are only reached if size exceeds a threshold. What we do is take the bitwise-AND of size and a CPU-feature mask; on CPUs lacking e.g. SSE, the mask is set to 0, thus preventing that codepath from ever being reached. Note that this technique could be expanded to support several mask values in case there are other features to be checked. Since the conditional jumps are necessary anyway, this approach only ends up adding minimal overhead (one instruction).

With all the pieces in place, we will now consider the implementation.

3 Implementation

The basic design goal is a drop-in replacement for `memcpy` that is as fast as possible over all input sizes/alignments. We are willing to require MMX – this has been available for about ten years now and is universally supported.

Given the above ideas, there are two things left to establish: which 'building blocks' to use for which transfer sizes, and how to order them.

Unfortunately, even with knowledge of the microarchitecture, performance of the individual copy techniques is difficult to predict; instead we implement them all and measure the results. A benchmark program observes elapsed time (accurate to a single CPU clock) over various transfer sizes and buffer misalignments. Building upon this measure, each technique has been painstakingly micro-optimized. Our conclusion is that for transfers of less than 64 bytes, an unrolled `MOV` loop together with a jump table for the last 3 bytes is fastest. The techniques used in AMD's implementation are good choices in all other cases. That means all transfer sizes are handled well by the combination of: unrolled `MOV` loop, MMX, SSE and SSE+block prefetching. The borders between them were chosen as follows:

64 bytes: lower limit for one MMX transfer cycle using all registers; larger sizes make unrolled code onerous.

64 KiB: L1 data cache size; beyond this, techniques that avoid polluting the cache are a big win.

256 KiB: L2 data cache size; from here on, splitting transfers into chunks that fit in-cache delivers huge gains.

It remains to determine the ordering of these four building blocks. There are two constraints: codepaths not using weakly-ordered writes (`MOVNTQ`) or MMX should not issue `SFENCE`/`EMMS` instructions, respectively; also, small copies should not require jumps. That induces the following layout: prolog, jump to large transfer code if necessary, small block copy, epilog. The 'large transfer' code makes use of the mask trick explained above; it then jumps to the SSE / SSE+block prefetch code or falls through to the MMX path. After each of these blocks, the minimum possible cleanup is performed and we jump to the 'leftovers' implementation (some of the techniques can only process e.g. 64 bytes, so any remainders must be handled separately). Analysing the number of branches taken in choosing a codepath, we see that tiny transfers get away with 0, while large transfers require 5/3 conditional and 4/3 unconditional on average (assuming equal distribution of transfer sizes).

Finally, we discuss the considerable hand-tuning that was undertaken. Contrary to conventional wisdom, micro-optimization can easily result in gains of 15 %; this is in addition to taking advantage of e.g. the SSE instruction set. The following guidelines have been incorporated into our implementation:

- prevent jumps by inlining code (e.g. `EPILOG`),
- minimize conditional branches,
- reduce memory R/W turnarounds,
- be frugal with registers (Win32 call convention says that everything

except EAX,ECX,EDX is callee-save). We only use those and ESI,EDI, so prolog/epilog register save code is kept to a minimum.

- avoid address generation stalls,
- align critical branch targets to 16 bytes,
- choose good prefetch stride,
- use complex addressing modes to reduce loop counter increments.

The results speak for themselves:

4 Performance Review

We now compare performance of our implementation vs. various others on Athlon XP and Pentium III test systems. Unless otherwise mentioned, the C++ compiler and runtime library are that of VC7.1. The contenders are:

1. `std::copy`. This is implemented as a call to `memcpy` (which is identical to `memmove`). We can therefore remove it from consideration; it is indeed always at least 1 % worse.
2. intrinsic `memcpy`. This boils down to `REP MOVSD` and `REP MOVSB` for the remainder. It never beats `memcpy` on the Athlon system; see discussion of `MOVS` above.
3. `memcpy`. This does buffer overlap and alignment checks, handles smaller transfers via unrolled `MOV` loop, and otherwise `REP MOVSD`. It does quite well up to 64 bytes, but cannot compete from there on due to lack of MMX/SSE codepaths.
4. `memcpy_amd` [AMD1]. This code also chooses several copy techniques depending on size. Unfortunately performance on smaller transfers is severely hobbled due to the aforementioned `EMMS`/conditional branch issues, but this becomes negligible for large transfers.
5. `ia32_memcpy`, our new optimized code. It performs well for both large and small copy sizes.

4.1 Benchmark

Before presenting the results, we further describe the benchmark program. All implementations are called from thunk functions; this factors in the bonus of inlining that the `memcpy` intrinsic enjoys. We measure elapsed time over single calls to the target code via CPU clock counter (`RDTSC`). External interference is avoided by setting scheduler priority to the maximum, repeating the measurement 100 times, median-filtering, and averaging 10 repetitions of the whole process. We are therefore confident of the measured times, irrespective of transfer sizes.

An attempt is made to choose realistic misalignments that are typical to real-world programs. The cached is pre-loaded, which biases performances figures. However, this is not unrealistic (applications will tend to use what they copy), all implementations profit from this equally, and it's difficult to avoid. Source code is included below.

4.2 Results

Performance is indicated by plots of CPU cycles vs. transfer size.

Figure 4.2 shows performance of small transfers; `ia32_memcpy` significantly outperforms all other implementations. `memcpy` comes closest; although using the same basic idea, we are 5..6 % faster up to 32 bytes, increasing that to 15..20 % between that and 64 bytes. This is due to less overhead and more efficient unrolled `MOV` code.

Figure 4.2 shows elapsed time for mid-sized transfers, where a technique using MMX is worthwhile. Our code is fastest (by margins of up to 20 %) up to 1 KiB; from there on, prefetching code present in `memcpy_amd` yields enough improvement to offset our smaller function overhead. Unfortunately we cannot use prefetch here for reasons of CPU compatibility, so our performance lags behind by about 5 %.

Figure 4.2 gives the same data for large transfers, where SSE codepaths become active. The L2 cache size can clearly be seen – as transfer sizes reach 192 KiB (L2 contains the contents of L1, hence 256 KiB-64 KiB), `memcpy_amd` and `ia32_memcpy` fare much better than the others due to non-temporal copies and block-prefetching. It is here that speedups of 358 % are seen. The relative performance of these two implementations lies between -1 % and +10 %. Reduced overhead no longer matters at these sizes, but

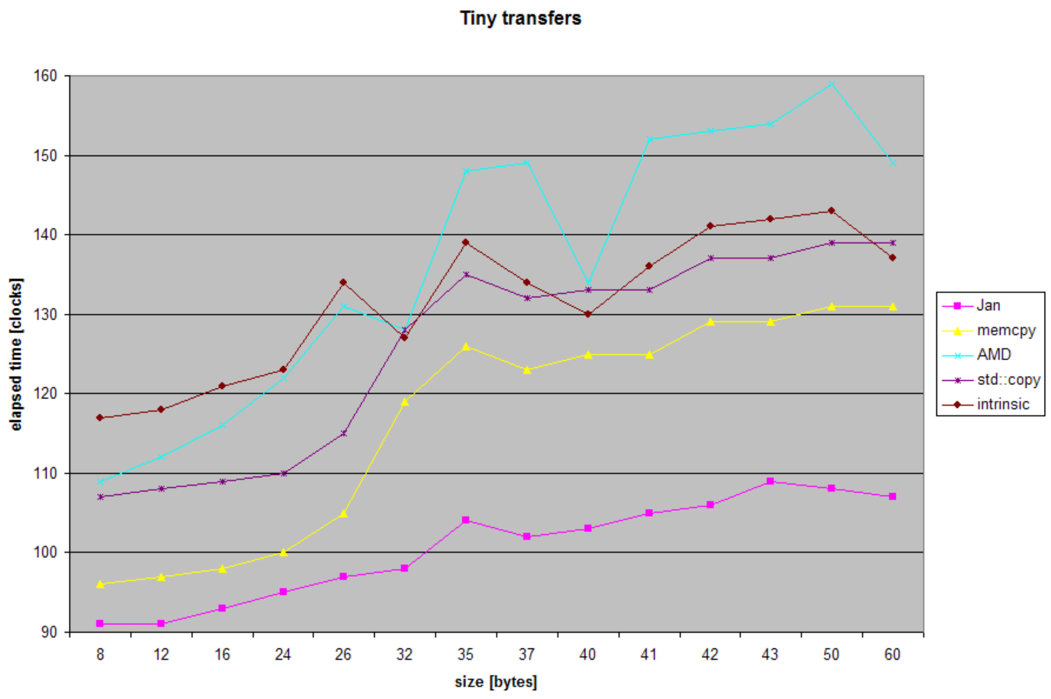

Figure 1: Elapsed time [clocks] for transfers of less than 64 bytes.

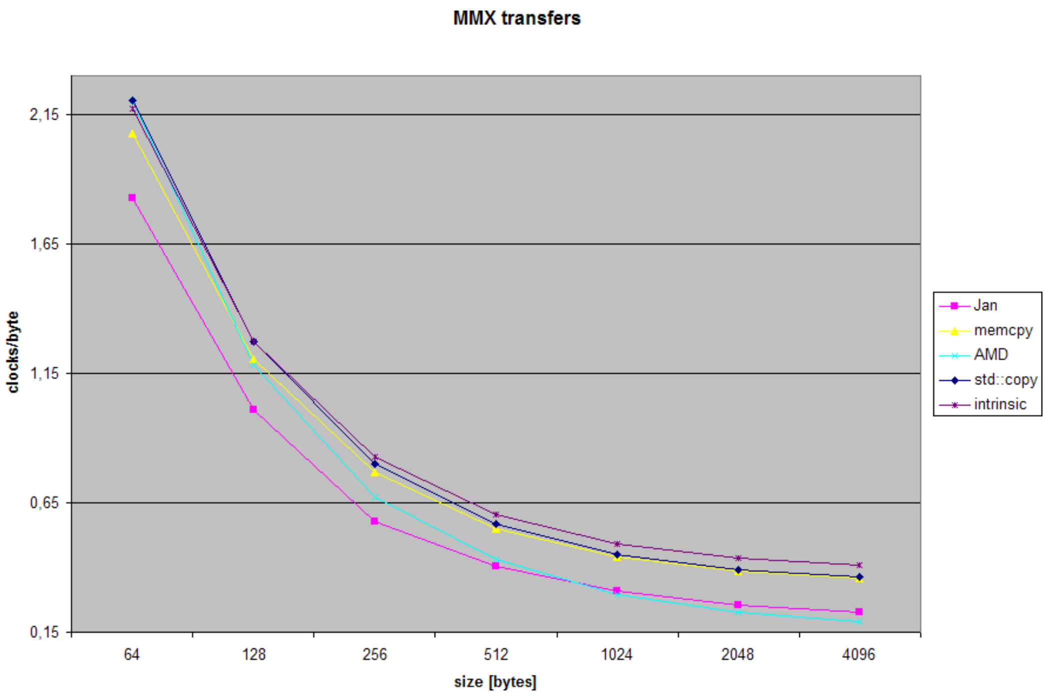

Figure 2: Elapsed time [clocks] for medium-sized transfers.

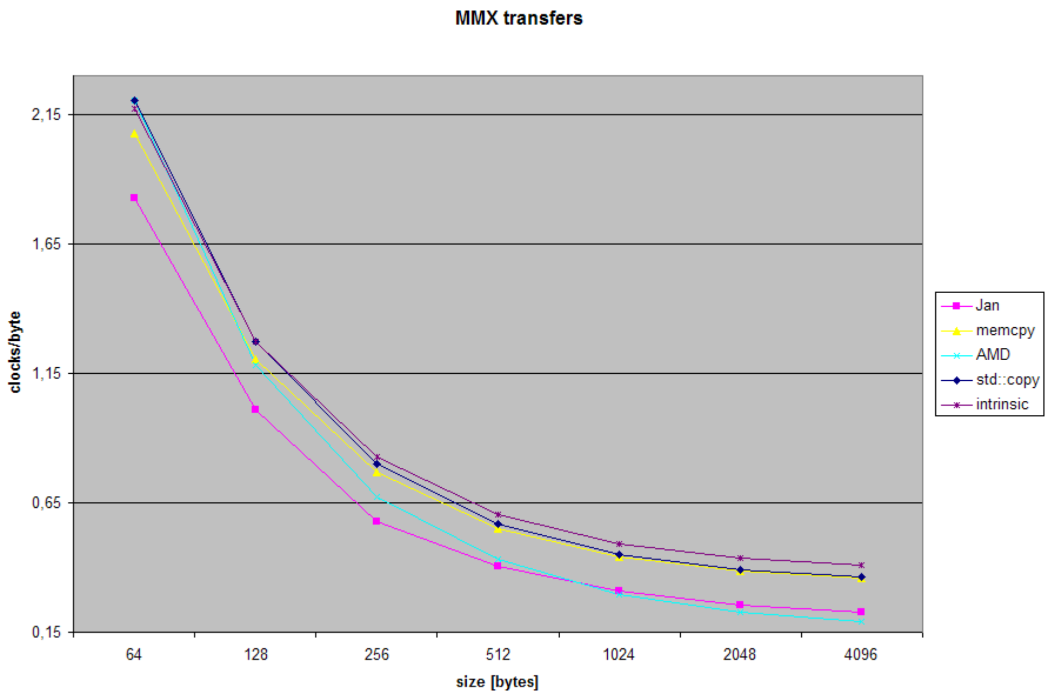

Figure 3: Elapsed time [clocks] for large transfers.

our code still generally comes out ahead due to more efficient branching and loops.

5 Conclusion

A very fast implementation of `memcpy` has been presented that significantly outperforms current runtime library offerings and even AMD's optimized code. It does not require special CPU features and can easily be dropped into your project. Everyone can therefore profit from this work.

Questions, comments, suggestions, bug reports? Feedback is welcome; send to Jan1.WassenbergATstud.uni-karlsruhe.de (remove digit and replace AT with @).

Bibliography ============

AMD1: Wall, Mike. "Optimizing Memory Bandwidth". Available online at `http://cdrom.amd.com/devconn/events/AMD_block_prefetch_paper.pdf` AMD2: Wall, Mike. "White Paper: Using Block Prefetch for Optimized Memory Performance". Available online at `http://cdrom.amd.com/devconn/events/gdc_2002_amd.pdf`

A Adding to Visual C++ Projects

Simply add the asm and C++ files to your project; for .asm files, set the following in their Custom Build Step properties:
Command Line: `nasm -f win32 -o "$(IntDir)\$(InputName).asm.obj" "$(InputPath)"`
Outputs: `$(IntDir)\$(InputName).asm.obj` (note: output files are named `*.asm.obj`; this avoids conflicts with C++ files of the same base name.)

The free NASM assembler is required and can be found at `http://nasm.sourceforge.net`.

B Source Code

Note: Be sure to call `ia32_memcpy_init` before using `ia32_memcpy`!

(licensed under the GPL; contact me to discuss the possibility of alternate licensing for your project)

```
; set section attributes
section .data data align=32 use32
section .bss  bss  align=16 use32
section .text code align=64 use32
; activate .text (needs to be separate because __SECT__ will otherwise
; complain that the above definition is redeclaring attributes)
section .text

; Usage:
; use sym(ia32_cap) instead of _ia32_cap - on relevant platforms, sym() will add
; the underlines automagically, on others it won't
%ifdef DONT_USE_UNDERLINE
%define sym(a) a
%else
%define sym(a) _ %+ a
%endif

;-----------------------------------------------------------------------------
; fast general memcpy
;-----------------------------------------------------------------------------

; drop-in replacement for libc memcpy(). only requires CPU support for
; MMX (by now universal). highly optimized for Athlon and Pentium III
; microarchitectures; significantly outperforms VC7.1 memcpy and memcpy_amd.
; for details, see accompanying article.

; if transfer size is at least this much,
; .. it's too big for L1. use non-temporal instructions.
UC_THRESHOLD equ 64*1024
; .. it also blows L2. pull chunks into L1 ("block prefetch").
BP_THRESHOLD equ 256*1024

; maximum that can be copied by IC_TINY.
IC_TINY_MAX equ 63

; size of one block prefetch chunk.
BP_SIZE equ 8*1024
```

```
;-------------------------------------------------------------------------------

; [p3] replicating this instead of jumping to it from tailN
; saves 1 clock and costs (7-2)*2 bytes code.
%macro EPILOG 0
pop esi
pop edi
mov eax, [esp+4] ; return dst
ret
%endm

align 64
tail1:
mov al, [esi+ecx*4]
mov [edi+ecx*4], al
align 4
tail0:
EPILOG

align 8
tail3:
; [p3] 2 reads followed by 2 writes is better than
; R/W interleaved and RRR/WWW
mov al, [esi+ecx*4+2]
mov [edi+ecx*4+2], al
; already aligned to 8 due to above code
tail2:
mov al, [esi+ecx*4]
mov dl, [esi+ecx*4+1]
mov [edi+ecx*4], al
mov [edi+ecx*4+1], dl
EPILOG

[section .data]
align 16
tail_table dd tail0, tail1, tail2, tail3
__SECT__

; 15x unrolled copy loop - transfers DWORDs backwards.
; indexed via table of 8-bit offsets.
; rationale:
; - [p3] backwards vs. forwards makes no difference.
; - MOV is faster than MOVSD.
; - index table is needed because calculating end-6*i is slower than
;   a LUT and we wouldn't want to expand entries to 8 bytes
;   (that'd increase code footprint by 30 bytes)
; - a byte index accessed via MOVZX is better due to less dcache usage.
; - only unrolling 8x and 'reentering' the loop is possible but
;   slower due to fiddling with esi/ecx.
align 64
unrolled_copy_code_start:
%assign i 15
%rep 14 ; 15 entries, 1 base case handled below
uc_ %+ i:
    mov     eax, [esi+i*4-4]
    mov     [edi+i*4-4], eax
%assign i i-1
%endrep
; base case: no displacement needed; skip it so that code will
; be aligned to 8 bytes after this.
uc_1:
    mov     eax, [esi]
    mov     [edi], eax
uc_0:
jmp [tail_table+edx*4]

[section .data]
align 32
unrolled_copy_index_table:
%assign i 0
%rep 16
db (uc_ %+ i) - unrolled_copy_code_start
%assign i i+1
%endrep
__SECT__


;------------------------------------------------------------------------------
; tiny copy - handles all cases smaller than IC_MOVQ's 64 byte lower limit.
; > edx = number of bytes (< IC_TINY_MAX)
; < does not return.
; x eax, ecx, edx
%macro IC_TINY 0
mov ecx, edx
shr ecx, 2
; calculating this address isn't possible due to skipping displacement on uc1;
; even so, it'd require calculating -6*ecx, which is slower than LUT.
movzx eax, byte [unrolled_copy_index_table+ecx]
and edx, byte 3
add eax, unrolled_copy_code_start
jmp eax
; never reached! the unrolled loop jumps into tailN, which
; then returns from the memcpy function.
%endm


;-------------------------------------------------------------------------------
; align destination address to multiple of 8. important for large transfers,
; but doesn't affect the tiny technique.
; > esi, edi -> buffers (updated)
; > ecx, edx = transfer size (updated)
; x eax
%macro IC_ALIGN 0
mov eax, edi
and eax, byte 7 ; eax = # misaligned bytes
jz already_aligned ; early out
lea eax, [align_table_start+eax*2]
jmp eax

; [p3] this is no slower than a table of mov and much smaller/simpler
align 8
align_table_start:
%rep 8
dec ecx
movsb
%endrep
mov edx, ecx
already_aligned:
%endm


;-------------------------------------------------------------------------------
```

```
; MMX MOVQ technique. used for in-cache transfers of 64B..64*KiB.
; must run on all CPUs, i.e. cannot use the SSE prefetchnta instruction.
; > ecx = -number_of_bytes (multiple of 64)
; > esi, esi point to end of the buffer, i.e. &last_qword+8.
; < ecx = 0
; x
%macro IC_MOVQ 0

align 16
%%loop:

; notes:
; - we can't use prefetch here - this codepath must support all CPUs.
;   [p3] that makes us 5..15% slower on 1KiB..4KiB transfers.
; - [p3] simple addressing without +ecx is 3.5% faster.
; - difference between RR/WW/RR/WW and R..R/W..W:
;   [p3] none (if simple addressing)
;   [axp] interleaved is better (with +ecx addressing)
; - enough time elapses between first and third pair of reads that we
;   could reuse MM0. there is no performance gain either way and
;   differing displacements make code compression futile anyway, so
;   we'll just use MM4..7 for clarity.
movq mm0, [esi+ecx]
movq mm1, [esi+ecx+8]
movq [edi+ecx], mm0
movq [edi+ecx+8], mm1
movq mm2, [esi+ecx+16]
movq mm3, [esi+ecx+24]
movq [edi+ecx+16], mm2
movq [edi+ecx+24], mm3
movq mm4, [esi+ecx+32]
movq mm5, [esi+ecx+40]
movq [edi+ecx+32], mm4
movq [edi+ecx+40], mm5
movq mm6, [esi+ecx+48]
movq mm7, [esi+ecx+56]
movq [edi+ecx+48], mm6
movq [edi+ecx+56], mm7
add ecx, byte 64
jnz %%loop
%endm
```

```
;-------------------------------------------------------------------------------
; SSE MOVNTQ technique. used for transfers that do not fit in L1,
; i.e. 64KiB..192KiB. requires Pentium III or Athlon; caller checks for this.
; > ecx = -number_of_bytes (multiple of 64)
; > esi, esi point to end of the buffer, i.e. &last_qword+8.
; < ecx = 0
; x
%macro UC_MOVNTQ 0

align 16
%%loop:
; notes:
; - the AMD optimization manual recommends prefetch distances according to
;   (200*BytesPerIter/ClocksPerIter+192), which comes out to ~560 here.
;   [p3] rounding down to 512 bytes makes for significant gains.
; - [p3] complex addressing with ecx is 1% faster than adding to esi/edi.
prefetchnta [esi+ecx+512]
movq mm0, [esi+ecx]
movq mm1, [esi+ecx+8]
movq mm2, [esi+ecx+16]
movq mm3, [esi+ecx+24]
movq mm4, [esi+ecx+32]
movq mm5, [esi+ecx+40]
movq mm6, [esi+ecx+48]
movq mm7, [esi+ecx+56]
movntq [edi+ecx], mm0
movntq [edi+ecx+8], mm1
movntq [edi+ecx+16], mm2
movntq [edi+ecx+24], mm3
movntq [edi+ecx+32], mm4
movntq [edi+ecx+40], mm5
movntq [edi+ecx+48], mm6
movntq [edi+ecx+56], mm7
add ecx, byte 64
jnz %%loop
%endm


;-------------------------------------------------------------------------------
```

```
; block prefetch technique. used for transfers that do not fit in L2,
; i.e. > 192KiB. requires Pentium III or Athlon; caller checks for this.
; for theory behind this, see article.
; > ecx = -number_of_bytes (multiple of 64, <= -BP_SIZE)
; > esi, esi point to end of the buffer, i.e. &last_qword+8.
; < ecx = -remaining_bytes (multiple of 64, > -BP_SIZE)
; < eax = 0
%macro UC_BP_MOVNTQ 0
push edx

align 4
%%prefetch_and_copy_chunk:
; pull chunk into cache by touching each cache line
; (in reverse order to prevent HW prefetches)
mov eax, BP_SIZE/128 ; # iterations
add esi, BP_SIZE
align 16
%%prefetch_loop:
mov edx, [esi+ecx-64]
mov edx, [esi+ecx-128]
add esi, byte -128
dec eax
jnz %%prefetch_loop

; copy chunk in 64 byte pieces
mov eax, BP_SIZE/64 ; # iterations (> signed 8 bit)
align 16
%%copy_loop:
movq mm0, [esi+ecx]
movq mm1, [esi+ecx+8]
movq mm2, [esi+ecx+16]
movq mm3, [esi+ecx+24]
movq mm4, [esi+ecx+32]
movq mm5, [esi+ecx+40]
movq mm6, [esi+ecx+48]
movq mm7, [esi+ecx+56]
movntq [edi+ecx], mm0
movntq [edi+ecx+8], mm1
movntq [edi+ecx+16], mm2
movntq [edi+ecx+24], mm3
movntq [edi+ecx+32], mm4
movntq [edi+ecx+40], mm5
movntq [edi+ecx+48], mm6
movntq [edi+ecx+56], mm7

add ecx, byte 64
dec eax
jnz %%copy_loop

; if enough data left, process next chunk
cmp ecx, -BP_SIZE
jle %%prefetch_and_copy_chunk

pop edx
%endm


;-------------------------------------------------------------------------------

; void* __declspec(naked) ia32_memcpy(void* dst, const void* src, size_t nbytes)
; drop-in replacement for libc memcpy() (returns dst)
global sym(ia32_memcpy)
align 64
sym(ia32_memcpy):
push edi
push esi

mov ecx, [esp+8+4+8] ; nbytes
mov edi, [esp+8+4+0] ; dst
mov esi, [esp+8+4+4] ; src

mov edx, ecx
cmp ecx, byte IC_TINY_MAX
ja choose_larger_method

ic_tiny:
IC_TINY
; never reached - IC_TINY contains memcpy function epilog code

choose_larger_method:
IC_ALIGN
```

```
; setup:
; eax = number of 64 byte chunks, or 0 if CPU doesn't support SSE.
;       used to choose copy technique.
; ecx = -number_of_bytes, multiple of 64. we jump to ic_tiny if
;       there's not enough left for a single 64 byte chunk, which can
;       happen on unaligned 64..71 byte transfers due to IC_ALIGN.
; edx = number of remainder bytes after qwords have been copied;
;       will be handled by IC_TINY.
; esi and edi point to end of the respective buffers (more precisely,
;       to buffer_start-ecx). this together with the ecx convention means
;       we only need one loop counter (instead of having to advance
;       that and esi/edi).

; this mask is applied to the transfer size. the 2 specialized copy techniques
; that use SSE are jumped to if size is greater than a threshold.
; we simply set the requested transfer size to 0 if the CPU doesn't
; support SSE so that those are never reached (done by masking with this).
extern sym(ia32_memcpy_size_mask)
mov eax, [sym(ia32_memcpy_size_mask)]
and ecx, byte ~IC_TINY_MAX
jz ic_tiny ; < 64 bytes left (due to IC_ALIGN)
add esi, ecx
add edi, ecx
and edx, byte IC_TINY_MAX
and eax, ecx
neg ecx

cmp eax, BP_THRESHOLD
jae near uc_bp_movntq
cmp eax, UC_THRESHOLD
jae uc_movntq

ic_movq:
IC_MOVQ
emms
jmp ic_tiny

uc_movntq:
UC_MOVNTQ
sfence
emms
```

```
jmp ic_tiny

uc_bp_movntq:
UC_BP_MOVNTQ
sfence
cmp ecx, byte -(IC_TINY_MAX+1)
jle ic_movq
emms
jmp ic_tiny


;-------------------------------------------------------------------------------
; CPUID support
;-------------------------------------------------------------------------------

[section .data]

; these are actually max_func+1, i.e. the first invalid value.
; the idea here is to avoid a separate cpuid_available flag;
; using signed values doesn't work because ext_funcs are >= 0x80000000.
max_func dd 0
max_ext_func dd 0

__SECT__


; extern "C" bool __cdecl ia32_cpuid(u32 func, u32* regs)
global sym(ia32_cpuid)
sym(ia32_cpuid):
push ebx
push edi

mov ecx, [esp+8+4+0] ; func
mov edi, [esp+8+4+4] ; -> regs

; compare against max supported func and fail if above
xor eax, eax ; return value on failure
test ecx, ecx
mov edx, [max_ext_func]
js .is_ext_func
mov edx, [max_func]
```

```
.is_ext_func:
cmp ecx, edx
jae .ret ; (see max_func decl)

; issue CPUID and store result registers in array
mov eax, ecx
cpuid
stosd
xchg eax, ebx
stosd
xchg eax, ecx
stosd
xchg eax, edx
stosd

; success
xor eax, eax
inc eax
.ret:
pop edi
pop ebx
ret


;--------------------------------------------------------------------------------
; init
;--------------------------------------------------------------------------------

; extern "C" bool __cdecl ia32_asm_init()
global sym(ia32_asm_init)
sym(ia32_asm_init):
push ebx

; check if CPUID is supported
pushfd
or byte [esp+2], 32
popfd
pushfd
pop eax
xor edx, edx
shr eax, 22 ; bit 21 toggled?
```

```
jnc .no_cpuid

; determine max supported CPUID function
xor eax, eax
cpuid
inc eax ; (see max_func decl)
mov [max_func], eax
mov eax, 0x80000000
cpuid
inc eax ; (see max_func decl)
mov [max_ext_func], eax
.no_cpuid:

pop ebx
ret

================================================================================
C++ header
================================================================================

typedef unsigned int u32;
typedef unsigned int uint;

// package code into a single statement.
// notes:
// - for(;;) { break; } and {} don't work because invocations of macros
//   implemented with STMT often end with ";", thus breaking if() expressions.
// - we'd really like to eliminate "conditional expression is constant"
//   warnings. replacing 0 literals with extern volatile variables fools
//   VC7 but isn't guaranteed to be free of overhead. we will just
//   squelch the warning (unfortunately non-portable).
#define STMT(STMT_code__) do { STMT_code__; } while(false)

// execute the code passed as a parameter only the first time this is
// reached.
// may be called at any time (in particular before main), but is not
// thread-safe. if that's important, use pthread_once() instead.
#define ONCE(ONCE_code__)\
STMT(\
static bool ONCE_done__ = false;\
if(!ONCE_done__)\
{\
ONCE_done__ = true;\
ONCE_code__;\
}\
)

#define debug_warn(str) assert(0 && str)

#define BIT(n) (1ul << (n))

#define restrict

extern "C" void ia32_asm_init();

// CPU caps (128 bits)
// do not change the order!
enum CpuCap
{
// standard (ecx) - currently only defined by Intel
SSE3 = 0+0,// Streaming SIMD Extensions 3
EST  = 0+7,// Enhanced Speedstep Technology

// standard (edx)
TSC  = 32+4,// TimeStamp Counter
CMOV = 32+15,// Conditional MOVe
MMX  = 32+23,// MultiMedia eXtensions
SSE  = 32+25,// Streaming SIMD Extensions
SSE2 = 32+26,// Streaming SIMD Extensions 2
HT   = 32+28,// HyperThreading

// extended (ecx)

// extended (edx) - currently only defined by AMD
AMD_MP        = 96+19,// MultiProcessing capable; reserved on AMD64
AMD_MMX_EXT   = 96+22,
AMD_3DNOW_PRO = 96+30,
AMD_3DNOW     = 96+31
};

extern "C" bool ia32_cap(CpuCap cap);
```

```
// order in which registers are stored in regs array
// (do not change! brand string relies on this ordering)
enum IA32Regs
{
EAX,
EBX,
ECX,
EDX
};

// try to call the specified CPUID sub-function. returns true on success or
// false on failure (i.e. CPUID or the specific function not supported).
// returns eax, ebx, ecx, edx registers in above order.
extern "C" bool ia32_cpuid(u32 func, u32* regs);

extern "C" void* ia32_memcpy(void* dst, const void* src, size_t nbytes);

==============================================================================
C++ init code
==============================================================================

// set by ia32_init, referenced by ia32_memcpy (asm)
extern "C" u32 ia32_memcpy_size_mask = 0;

bool ia32_cap(CpuCap cap)
{
// treated as 128 bit field; order: std ecx, std edx, ext ecx, ext edx
// keep in sync with enum CpuCap!
static u32 caps[4];
ONCE(\
u32 regs[4];
if(ia32_cpuid(1, regs))\
{\
caps[0] = regs[ECX];\
caps[1] = regs[EDX];\
}\
if(ia32_cpuid(0x80000001, regs))\
{\
caps[2] = regs[ECX];\
caps[3] = regs[EDX];\
}\
);

const uint tbl_idx = cap >> 5;
const uint bit_idx = cap & 0x1f;
if(tbl_idx > 3)
{
debug_warn("cap invalid");
return false;
}
return (caps[tbl_idx] & BIT(bit_idx)) != 0;
}

void ia32_init()
{
ia32_asm_init();

// memcpy init: set the mask that is applied to transfer size before
// choosing copy technique. this is the mechanism for disabling
// codepaths that aren't supported on all CPUs; see article for details.
// .. check for PREFETCHNTA and MOVNTQ support. these are part of the SSE
// instruction set, but also supported on older Athlons as part of
// the extended AMD MMX set.
if(ia32_cap(SSE) || ia32_cap(AMD_MMX_EXT))
ia32_memcpy_size_mask = ~0u;
}


=====================================================================
benchmark
=====================================================================

// call through a function pointer for convenience and to foil optimizations.
typedef void*(*PFmemcpy)(void* restrict, const void* restrict, size_t);

// allocated memory is aligned to this; benchmark adds misalign.
static const size_t ALIGN = 64;
// max copy size; we allocate 1 byte more for sentinel
static const size_t MAX_SIZE = 1*MiB;
static const size_t MAX_MISALIGN = 64;

// note: various debug_asserts checking if dt > 0 have an important purpose:
```

```
// they catch values of -1.#IND0, which indicates MMX code lacking a
// trailing EMMS instruction.

static i64 measure_impl(PFmemcpy impl, void* dst, const void* src, size_t size)
{
debug_assert(size <= MAX_SIZE);

// (we check below if values were correctly copied)
const u8 SENTINEL = 0xDF;
const u32 prev_value = *(u32*)((u8*)dst-4);
memset(dst, (int)SENTINEL, MAX_SIZE+1);

// repeat measurement several times and filter the results to
// reduce external interference (e.g. task switch). this is a
// problem when measuring large transfers despite REALTIME thread pri.
const int REPS = 8;
double sum_of_avgs = 0.0;
for(int reps = 0; reps < REPS; reps++)
{
const int NUM_SAMPLES = 80;
i64 dts[NUM_SAMPLES];
for(int i = 0; i < NUM_SAMPLES; i++)
{
const i64 t0 = rdtsc();
impl(dst, src, size);
const i64 t1 = rdtsc();

const i64 dt = t1-t0;
debug_assert(dt > 0); // shouldn't be negative or 0
dts[i] = dt;

// make sure the implementation copied correctly.
// this is only done once to avoid randomized effects (e.g. the
// first run takes care of half, the second does the rest),
// and to reduce benchmark execution time.
if(i == 0)
{
// copied everything successfully
debug_assert(memcmp(src, dst, size) == 0);
// didn't overflow i.e. sentinel byte still intact
debug_assert( ((u8*)dst)[size] == SENTINEL );
// didn't underflow
debug_assert(prev_value == *(u32*)((u8*)dst-4));
}
}
// .. median filter (discard upper and lower fringes; average the rest)
// this is good to 0.3% spread on 64KB transfers.
std::sort(dts, dts+NUM_SAMPLES);
i64 total_dt = 0;
const int LO = NUM_SAMPLES/10, HI = 1*NUM_SAMPLES/5;
for(int i = LO; i < HI; i++)
total_dt += dts[i];
const double avg_dt = (double)total_dt / (double)(HI-LO);
sum_of_avgs += avg_dt;
debug_assert(sum_of_avgs > 0.0);
}

const i64 result = ia32_i64_from_double(sum_of_avgs/REPS);
debug_assert(result > 0);
return result;
}


// perform the benchmark for <impl>, copying from src_mem to dst_mem.
// the execution time of transfers of all sizes and misalignments up to
// MAX_* are added together and returned [in CPU clocks].
static i64 measure_foreach_misalign(PFmemcpy impl, void* dst_mem,
const void* src_mem, size_t size)
{
i64 total_dt = 0;

static const size_t misalignments[] = { 0,8,4,0,8,1,0,8,4,0,8,1,0,8,4 };
// static const size_t misalignments[] = { 0,0,1,2,3,4 };
// static const size_t misalignments[] = { 1 };
for(int misalign_idx = 0; misalign_idx < ARRAY_SIZE(misalignments); misalign_idx
{
const size_t misalign = misalignments[misalign_idx];
debug_assert(misalign <= MAX_MISALIGN);
void* dst = (char*)dst_mem + misalign;
const void* src = (const char*)src_mem + misalign;

const i64 dt = measure_impl(impl, dst, src, size);
```

```
debug_assert(dt > 0);
total_dt += dt;
}
debug_assert(total_dt > 0);

const double avg_dt = (double)total_dt / ARRAY_SIZE(misalignments);
const i64 result = ia32_i64_from_double(avg_dt);
debug_assert(result > 0);
return result;
}



static void* thunk_std_copy(void* dst, const void* src, size_t size)
{
std::copy<const u8*, u8*>((const u8*)src, (const u8*)src+size, (u8*)dst);
return dst;
}

#pragma intrinsic(memcpy)
static void* thunk_intrinsic(void* dst, const void* src, size_t size)
{
return memcpy(dst, src, size);
}
#pragma function(memcpy)

static void* thunk_amd(void* dst, const void* src, size_t size)
{
return memcpy_amd(dst, src, size);
}

static void* thunk_jan(void* dst, const void* src, size_t size)
{
return memcpy2(dst, src, size);
}

static void* thunk_vc71(void* dst, const void* src, size_t size)
{
return memcpy(dst, src, size);
}
```

```
struct Contender
{
const char* name;
PFmemcpy impl;
i64 elapsed_clocks;
};
static struct Contender contenders[] =
{
{"Jan      ", thunk_jan},
{"VC7.1    ", thunk_vc71},
{"AMD      ", thunk_amd},
{"std::copy", thunk_std_copy},
{"intrinsic", thunk_intrinsic}
};


static void do_benchmark_for_sizes(const size_t sizes[],
void* aligned_dst, const void* aligned_src)
{
for(int size_idx = 0; ; size_idx++)
{
const size_t size = sizes[size_idx];
if(!size)
break;

debug_printf("\n%d:\n", size);
for(int i = 0; i < ARRAY_SIZE(contenders); i++)
contenders[i].elapsed_clocks = measure_foreach_misalign(
contenders[i].impl, aligned_dst, aligned_src, size);

// find minimum
i64 min_clocks = LLONG_MAX;
for(int i = 0; i < ARRAY_SIZE(contenders); i++)
min_clocks = MIN(min_clocks, contenders[i].elapsed_clocks);

// display results with delta to best value
for(int i = 0; i < ARRAY_SIZE(contenders); i++)
{
const double difference = (contenders[i].elapsed_clocks - min_clocks) / (double)
debug_printf("  %s: %I64d (+%.1f%%)\n", contenders[i].name, contenders[i].elapse
}
}
}


// globals passed between init, shutdown and memcpy_benchmark
static void* dst_mem;
static const void* src_mem;
static void* aligned_dst;
static const void* aligned_src;
static int old_policy; static sched_param old_param;

static void init()
{
const size_t total_size = MAX_SIZE+1 + ALIGN + MAX_MISALIGN;
dst_mem = malloc(total_size);
src_mem = (const void*)malloc(total_size);
if(!dst_mem || !src_mem)
{
debug_warn("memcpy_benchmark failed; couldn't allocate buffers");
throw std::bad_alloc();
}

// align pointers [just so that misalignment actually goes from 0..63,
// instead of e.g. 21..20 (mod 64)]
aligned_dst = (void*)round_up((uintptr_t)dst_mem, ALIGN);
aligned_src = (const void*)round_up((uintptr_t)src_mem, ALIGN);

// fill src with an easily discernable test pattern - this is to ensure
// "memcmp = 0 <==> copy is correctly implemented".
for(size_t i = 0; i < MAX_SIZE; i++)
((u8*)aligned_src)[i] = (u8)(i % 256);

// set max priority, to reduce interference while measuring.
pthread_getschedparam(pthread_self(), &old_policy, &old_param);
sched_param max_param = {0};
max_param.sched_priority = sched_get_priority_max(SCHED_FIFO);
pthread_setschedparam(pthread_self(), SCHED_FIFO, &max_param);
}

static void shutdown()
```

```
{
// restore previous policy and priority.
pthread_setschedparam(pthread_self(), old_policy, &old_param);

free(dst_mem);
free((void*)src_mem);
}

void memcpy_benchmark()
{
ia32_init(); // necessary - we may be called via self-test before normal init

init();

debug_printf("TINY TRANSFERS\n");
debug_printf("---------------\n");
// static size_t small_sizes[64];for(int i = 0; i < 32; i++) small_sizes[i] = i+
static const size_t small_sizes[] = { 8, 12, 16, 24, 26, 32, 35, 37, 40,41,42,43
do_benchmark_for_sizes(small_sizes, aligned_dst, aligned_src);

debug_printf("MMX TRANSFERS\n");
debug_printf("---------------\n");
static const size_t med_sizes[] = { 64, 128, 256, 512, 1024, 2048, 4096, /*16384
do_benchmark_for_sizes(med_sizes, aligned_dst, aligned_src);

debug_printf("LARGE TRANSFERS\n");
debug_printf("---------------\n");
static const size_t large_sizes[] = { 64*KiB, 96*KiB, 128*KiB, 192*KiB, 256*KiB,
do_benchmark_for_sizes(large_sizes, aligned_dst, aligned_src);

shutdown();
}
```