Testing Conditions

- **CPU**
- Results are from `run_rand_bench`. Each trials (**100,000**) runs **4096** iterations the `al_src`, `al_dst`, and `sz` parameters. These values are taken from the SPEC2017 distribution. For purpose of simplifying these results `sz` is only of values in range `[0, 32]`.
- The memcpy implementation I tested only supported `sz` in `[0, 32]` in an attempt to isolate what I believe to be the problem.
- The results are from **throughput** tests. **NOT latency**.
- All of the benchmark code is on its own page and only uses address range % 4096 in `[0, 2047]`
- The memcpy code is on its own page and only uses address range % 4096 in `[2048, 4095]`.

Notes on Testing Conditions

- **The testing conditions described above are the only ones I have run the entire benchmark for** (roughly 16 hours). I have tinkered with some other conditions that I though may have had an effect or be skewing the results entirely and **Did NOT see any conditions which appeared to alter the trends described below**.
- The benchmark loop calling memcpy is aligned to 64 bytes. I have not run the full benchmark on other loop alignments but have somewhat tested that other alignments / addition `nop` padding values do not change the trend in the results (although can have some effect on the exact values by what seems to be a constant factor).
- Splitting the address space on different boundaries, for example benchmark code in `[0, 511]` and memcpy in `[512, 4095]` (or `[0, 1023]` and `[1024, 4095]`) also yielded the same trend with minimal (if any) effect on the results.
- Measuring latency as opposed to throughput significantly shrunk the variance between the "fast" and "slow" mode, although the same trend in overall performance and performance counter values was still visible.

Notes on Tested Parameters / Implementation

The **parameters** tested are as follows: - `ALIGN_ENTRY` in `[16, 32, 48, 64]`. - The formula for computing the entry address % 4096 of `memcpy` is `2048 + (ALIGN_ENTRY % 64)`. - `PADDING` in `[0...1025]` s.t `padding % 16` in `[0, 1]`

The **implementation** with the above parameters is as follows:

```
 // + ALIGN_ENTRY % 64

memcpy:
 800:   48 83 fa 20             cmp    $0x20,%rdx
 804:   72 2a                   jb     830 <memcpy_dev_v32_movsb+0x30>
 806:   62 e1 fe 28 6f 06       vmovdqu64 (%rsi),%ymm16
 80c:   62 61 fe 28 6f 7c 16    vmovdqu64 -0x20(%rsi,%rdx,1),%ymm31
 813:   ff
 814:   62 e1 fe 28 7f 07       vmovdqu64 %ymm16,(%rdi)
 81a:   62 61 fe 28 7f 7c 17    vmovdqu64 %ymm31,-0x20(%rdi,%rdx,1)
 821:   ff
 822:   c3                      retq
 823:   66 66 2e 0f 1f 84 00    data16 nopw %cs:0x0(%rax,%rax,1)
 82a:   00 00 00 00
 82e:   66 90                   xchg   %ax,%ax

 // + PADDING

L(less_vec):
 830:   83 fa 10                cmp    $0x10,%edx
 833:   73 2b                   jae    860 <memcpy_dev_v32_movsb+0x60>
 835:   83 fa 08                cmp    $0x8,%edx
 838:   73 46                   jae    880 <memcpy_dev_v32_movsb+0x80>
 83a:   83 fa 04                cmp    $0x4,%edx
 83d:   73 61                   jae    8a0 <memcpy_dev_v32_movsb+0xa0>
 83f:   83 fa 01                cmp    $0x1,%edx
 842:   72 11                   jb     855 <memcpy_dev_v32_movsb+0x55>
 844:   0f b6 4c 16 ff          movzbl -0x1(%rsi,%rdx,1),%ecx
 849:   74 06                   je     851 <memcpy_dev_v32_movsb+0x51>
 84b:   0f b7 36                movzwl (%rsi),%esi
 84e:   66 89 37                mov    %si,(%rdi)
 851:   88 4c 17 ff             mov    %cl,-0x1(%rdi,%rdx,1)
 855:   c3                      retq
 856:   66 2e 0f 1f 84 00 00    nopw   %cs:0x0(%rax,%rax,1)
 85d:   00 00 00
 860:   c5 fa 6f 06             vmovdqu (%rsi),%xmm0
 864:   c5 fa 6f 4c 16 f0       vmovdqu -0x10(%rsi,%rdx,1),%xmm1
 86a:   c5 fa 7f 07             vmovdqu %xmm0,(%rdi)
 86e:   c5 fa 7f 4c 17 f0       vmovdqu %xmm1,-0x10(%rdi,%rdx,1)
 874:   c3                      retq
 875:   66 66 2e 0f 1f 84 00    data16 nopw %cs:0x0(%rax,%rax,1)
 87c:   00 00 00 00
 880:   48 8b 0e                mov    (%rsi),%rcx
 883:   48 8b 74 16 f8          mov    -0x8(%rsi,%rdx,1),%rsi
 888:   48 89 0f                mov    %rcx,(%rdi)
 88b:   48 89 74 17 f8          mov    %rsi,-0x8(%rdi,%rdx,1)
 890:   c3                      retq
 891:   66 66 2e 0f 1f 84 00    data16 nopw %cs:0x0(%rax,%rax,1)
 898:   00 00 00 00
 89c:   0f 1f 40 00             nopl   0x0(%rax)
 8a0:   8b 0e                   mov    (%rsi),%ecx
```

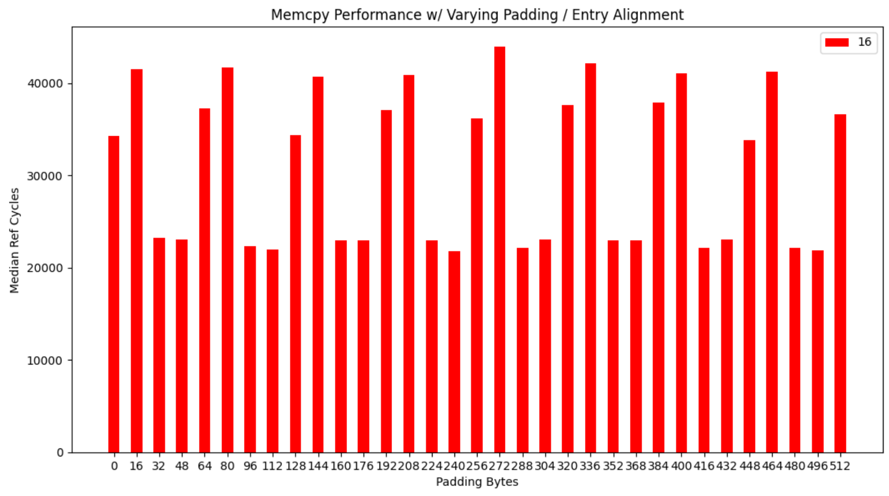
```
8a2:   8b 74 16 fc             mov    -0x4(%rsi,%rdx,1),%esi
8a6:   89 0f                   mov    %ecx,(%rdi)
8a8:   89 74 17 fc             mov    %esi,-0x4(%rdi,%rdx,1)
8ac:   c3                      retq
```

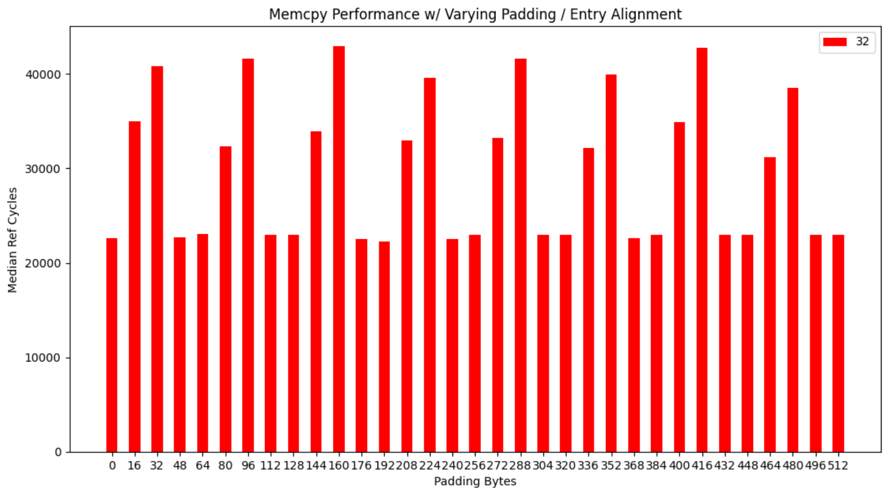
Please Note

- **The exact address, and address % 64 of `L(less_vec)` changes with `ALIGN_ENTRY`.**
- **`PADDING` essentially represents `address_of(memcpy) - address_of(L(less_vec)) + 48`**

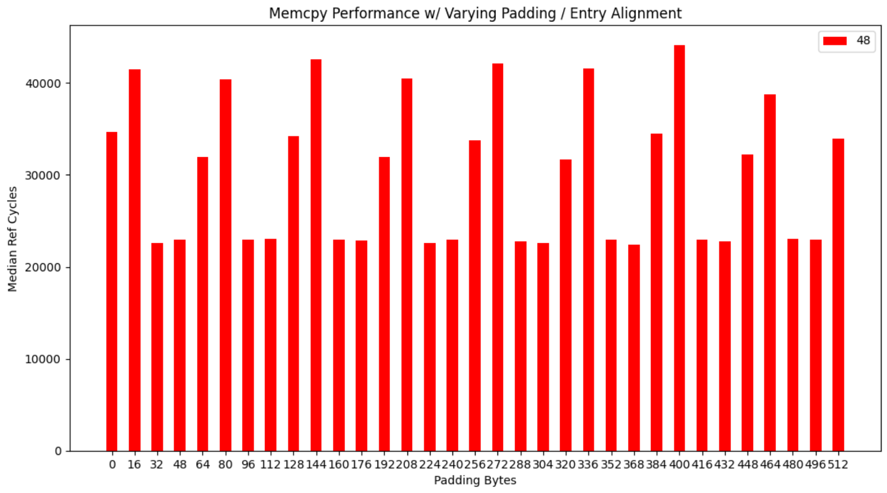
Results

The results for `Ref Cycles` (from `rdtsc`) vs `PADDING` for `ALIGN_ENTRY` in `[16, 64]` are:

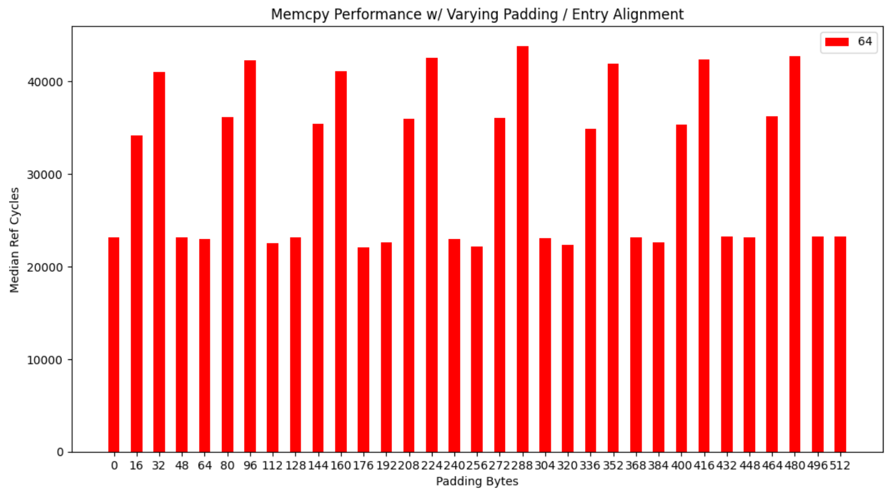




Memcpy Performance w/ Varying Padding / Entry Alignment
32
Median Ref Cycles
Padding Bytes
0
10000
20000
30000
40000
0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240 256 272 288 304 320 336 352 368 384 400 416 432 448 464 480 496 512



Memcpy Performance w/ Varying Padding / Entry Alignment
48
Median Ref Cycles
Padding Bytes
0
10000
20000
30000
40000
0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240 256 272 288 304 320 336 352 368 384 400 416 432 448 464 480 496 512



The results seems to depend on `ALIGN_ENTRY` and `PADDING % 64`. The following table sumarizes the basic results:

ALIGN_ENTRY	0	16	32	48
16	SLOW	SLOW	FAST	FAST
32	FAST	SLOW	SLOW	FAST
48	SLOW	SLOW	FAST	FAST
64	FAST	SLOW	SLOW	FAST