Beware Of Your Cacheline

— Processor Specific Optimization Techniques —

Hagen Paul Pfeifer
hagen@jauu.net

http://protocol-laboratories.net

Jan 18 2007

Introduction

- Why?
  - Memory bandwidth is high (more or less) – but today memory latency comes into play (compared to cpu speed)
- BTW: Excuse my Intel/AMD affinity
- Core 2 Extreme QX6700: L1D: 32 KByte 8-way; L1I: 32 KByte; 2x4 MByte 16-way
- Core 2 : L1D: 32 KByte 8-way; L1I: 32 KByte; 4/2 MByte 16-way
- Athlon 64 X2 / FX-62: L1D: 64 KByte 2-way; L1I: 64 KByte; 2x 512 KByte / 2x 1 MByte (FX-62) 16-way
- Pentium 4: L1D: 16 KByte 8-way; L1I: 12 Kuops (equivalent 8 till 16 KByte); 1 MByte 8-way

Prefetch

- Use prefetch
- Data at contiguous memory locations and in ascending order
- Use for loops that process large amounts of data
- If data in loop is less than a cacheline: unroll loop (by hand to take full control - avoid `-funroll-loops`)
- AMD `PREFETCH` and `PREFETCHW`
  - K6-2:
    - Small performance improvement – share FSB, low priority
    - Don't add overhead through useless prefetching
  - K6-3:
    - Separate backside bus
  - Up to 20% improvements

- 64 bytes per prefetch
- AMD tip: prefetch 192 (3*64) bytes ahead or three cachelines (Prefetch Distance)
- BTW: FSB and AMD is a separate topic (keyword HyperTransport, Memory controller, Direct Connect Architecture)

- Usage (AMD):

```
double a[A_REALLY_LARGE_NUMBER];
double b[A_REALLY_LARGE_NUMBER];
double c[A_REALLY_LARGE_NUMBER];
for (i=0; i<A_REALLY_LARGE_NUMBER/4; i++) {
    prefetchw (a[i*4+64]); // will be modifying a
    prefetch (b[i*4+64]);
    prefetch (c[i*4+64]);
    a[i*4]   = b[i*4]   * c[i*4];
    a[i*4+1] = b[i*4+1] * c[i*4+1];
    a[i*4+2] = b[i*4+2] * c[i*4+2];
```

```
  a[i*4+3] = b[i*4+3] * c[i*4+3];
}
```

memcpy

- `movdqa/movdqu` (aligned/unaligned)
  - SIMD extension
  - Moves a double quadword from/to mmx register to/from mmx register/memory location
  - `movdqa`: 16 byte aligned or general-protection exception
  - Advantages: faster ordinary memcpy for bigger sizes
- p4: `prefetchnta, prefetcht0, prefetcht1, prefetcht2`
  - p3: 32 byte
  - p4: 128 byte
- Like `madivce(2)` it's a hint and not a command
- `prefetchnta`
  - Prefetches data into L2 cache without polluting L1 caches (non temporal)

- SMP: reduce the CPU – cache traffic

- BTW: icc replace `memcpy()` in `string.h` with a processor specific version (handles alignment, tlb priming and prefetching)

MMX, SSE, SSE2, SSE4

- SIMD: single instruction, multiple data
- More then one instruction processed concurrently
- MMX:
  - Concurrent processing with FP unit -> restore register
  - `mm[0-7]` are aliases for `r[0-7]` (FPU Register)
  - No over- or underflow exceptions (no carry, overflow or adjust flags) like every SIMD instructions
  - Wrap-Around or saturation mode!
- SSE2:
  - Mainly graphics and codecs precessing
  - Added 122 instructions
- SSE3:

- Introduction: Pentium 4
- 13 additional SIMD instructions
- Thread synchronisation (`MONITOR`, `MWAIT`)

- SSSE3:
  - Xeon 5100 and Core 2 Duo
  - 32 new instructions (16 x 64-bit MMX, 16 x 128-bit XMM registers)
- Intel's Core-Architecture extension: SSE4 (Nehalem New Instructions)

  - 50 new OpCodes
  - Major improvements:
    - String and text processing
    - Vectorizing
  - Hardware CRC32 calculations (`crc32`)

- New advanced string instructions
- Release: 2008

GCC Attributes

- `packed`
  - Specifies a minimum alignment
  - Unaligned exception
  - `int x __attribute__ ((aligned (16))) = 0;` (align on a 16byte boundary)
  - `short array[3] __attribute__ ((aligned));` (unspecific alignment - largest alignment)
- `packed`
  - Smallest possible alignment
  - ```
    struct foo {
      char a;
      int x[2] __attribute__ ((packed));
    };
    ```
    ```

```
struct __attribute__ ((packed)) bar { ... };
```

- Beware of you architecture: e.g. performance killer on powerpc

- ```
  #define likely(x)   __builtin_expect (!!(x), 1)
  #define unlikely(x) __builtin_expect (!!(x), 0)
  ```
- Incidentally: this code isn't portable anymore! ;-)
- `#pragma`
  ```

Tools

- `gcc -S`
  - Play with gcc flags and take a look at the generated results!
- `rdtscll`
- `cachegrind` – a cache profiler
  - `valgrind --tool=cachegrind command`
  - `cg_annotate`
- `pahole` (Arnaldo Carvalho de Melo)
  - Utilize DWARF2 information

```
[acme@newtoy net-2.6]$ pahole kernel/sched.o task_struct
/* include2/asm/system.h:11 */
struct task_struct {
  volatile long int       state;          /*     0     4 */
  struct thread_info *    thread_info;    /*     4     4 */
```

```
    atomic_t                usage;          /*     8     4 */
    long unsigned int       flags;          /*    12     4 */
    <SNIP>
    short unsigned int          ioprio;     /*    52     2 */
    /* XXX 2 bytes hole, try to pack */
    long unsigned int           sleep_avg;  /*    56     4 */
    unsigned char               fpu_counter; /*   388     1 */
    /* XXX 3 bytes hole, try to pack */
    int                         oomkilladj; /*   392     4 */
    <SNIP>
};  /* size: 1312, sum members: 1287, holes: 3, sum holes: 13, padding: 12 */
```

- `pfunct` print function details
- `git://git.kernel.org/pub/scm/linux/kernel/git/acme/pahole.git`
- `vtune` (Intel, Single Developer: $699)

Additional Information

- Some more tips:
  - Avoid branching – use branchless instructions
  - `sched_setaffinity(2)`
  - Avoid SMP trashing
  - `CPUID` opcode (CPU IDentification)
  - `gcc`
    - X86 Built-in Functions
      (`v2df __builtin_ia32_addsubpd (v2df, v2df)`)
    - `march=`
    - `-msse` or `-msse2`
    - `-mfpmath=` (387, sse)
  - Use `strlen()` – avoid `while(*p++) ++i;`

- PowerPC 4xx support `dlmzb` instruction
- `dlmzb`: determine left-most zero byte
- `strcpy()` is another candidate – `dlmzb` with support of `lswx` and `stswx`

- Intel Smart Memory Access
- Intel Advanced Smart Cache

Fin

- Questions?

Additional Information

- Links:
  - A solaris memcpy implementation for pentium cpu
  - SSE4 Introduction
  - Extending the World's Most Popular Processor Architecture
  - x86 Instruction Reference
  - 3DNow! Instruction Porting Guide
  - How Cachegrind works
  - X86 Built-in Functions
  - GCC Intel 386 and AMD x86-64 Options
  - Data Cache Optimizations for Java Programs

Contact

- Hagen Paul Pfeifer
- EMail: hagen@jauu.net
  - Key-ID: `0x98350C22`
  - Fingerprint: `490F 557B 6C48 6D7E 5706 2EA2 4A22 8D45 9835 0C22`