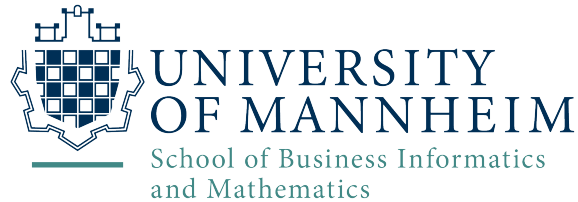
Database Systems II – Exercise #2

Sheet #2: Cache & Prefetching, SIMD

Daniel Flachs

Chair of Practical Computer Science III:
Database Management Systems

06/03/2019



Created March 11, 2019

Contents

Contents

Task 1a

Download the zip archive for this exercise sheet from the website and use the provided make file to compile `experiments1.cc`.

First, run `./experiment1`. Then, run `./experiment1 -r`. You will get different outputs. Analyse the code and interpret the observed outputs.

Answer: The code measures the performance of the hardware prefetcher for sequential access and random access. Random access is enabled by the command line parameter `-r`. The number of array elements is controlled by `-n <number>`. As expected, random access is slower than sequential access.

Task 1b

Install the profiling tool *valgrind* and rerun the above experiments. This time, track the number of cache misses using valgrind:

```
valgrind --tool=cachegrind ./experiment1 -n 1000000
valgrind --tool=cachegrind ./experiment1 -n 1000000 -r
```

Note

To provide cachegrind with as much information as possible, compile your program with the "debug info" option -g, or even -g3.

Cachegrind Results

- When executing cachegrind, it prints some results summary to the commandline: cache accesses for instructions (I), cache accesses for data (D), combined figures for I and D in the last-level cache (LL)
- More detailed data is written to a cachegrind output file (cachegrind.out.<pid>) and can be extracted using the cachegrind tool cg_annotate.
- Printing function-by-function statistics:
  cg_annotate <cg output file>
- Printing line-by-line statistics:
  cg_annotate <cg output file> <source code file>
  Note: <source code file> seems to have to be specified using its absolute path.

Results Summary

Sequential Access

```
I    refs:      107,985,782
I1   misses:          1,924
LLi  misses:          1,843
I1   miss rate:        0.00%
LLi  miss rate:        0.00%

D    refs:       49,992,471  (39,728,520 rd   + 10,263,951 wr)
D1   misses:        271,122  (   143,493 rd   +    127,629 wr)
LLd  misses:        136,395  (     9,707 rd   +    126,688 wr)
D1   miss rate:         0.5% (       0.4%     +        1.2%   )
LLd  miss rate:         0.3% (       0.0%     +        1.2%   )

LL refs:            273,046  (   145,417 rd   +    127,629 wr)
LL misses:          138,238  (    11,550 rd   +    126,688 wr)
LL miss rate:           0.1% (       0.0%     +        1.2%   )
```

Random Access

```
I    refs:      254,924,108
I1   misses:          1,935
LLi  misses:          1,857
I1   miss rate:        0.00%
LLi  miss rate:        0.00%

D    refs:      135,992,560  (94,728,678 rd   + 41,263,882 wr)
D1   misses:      2,217,662  ( 2,090,031 rd   +    127,631 wr)
LLd  misses:        136,803  (    10,114 rd   +    126,689 wr)
D1   miss rate:         1.6% (       2.2%     +        0.3%   )
LLd  miss rate:         0.1% (       0.0%     +        0.3%   )

LL refs:          2,219,597  ( 2,091,966 rd   +    127,631 wr)
LL misses:          138,660  (    11,971 rd   +    126,689 wr)
LL miss rate:           0.0% (       0.0%     +        0.3%   )
```

Per-Function Results

Sequential Access

```
--------------------------------------------------------------------------------
          Dr     D1mr    DLmr          Dw    D1mw    DLmw  file:function
--------------------------------------------------------------------------------
 8,000,000        5       2  4,000,000       0       0  ???:random_r
 9,000,000        1       1  2,000,000       0       0  ???:random

 5,000,004        0       0  2,000,004  62,500  62,500  experiment1.cc:fillData(int*, int)
10,000,008  125,003   1,447  1,000,011       0       0  experiment1.cc:sum(int const*, \
                                                            int const*, unsigned long)
 6,000,004        0       0  1,000,004  62,501  62,500  experiment1.cc:
                                                            fillIndex(int*, int)

 1,000,000        0       0  1,000,000       0       0  ???:rand
   211,981    3,036   2,198     70,177      20       8  ???:_dl_lookup_symbol_x
 1,000,183       24       8         51       1       1  ???:???
   271,820    5,647     954    121,026      19      12  ???:do_lookup_x
   103,223    6,463   4,158     36,526   1,831     990  ???:_dl_relocate_object
    35,516      404     134          0       0       0  ???:strcmp
    52,361      748     390     17,516      11       4  ???:check_match
```

Per-Function Results

Random Access

```
--------------------------------------------------------------------------------
         Dr         D1mr    DLmr          Dw    D1mw    DLmw  file:function
--------------------------------------------------------------------------------
15,999,992           10       2  7,999,996       0       0  ???:random_r
17,999,991            4       1  3,999,998       0       0  ???:random
11,999,772    1,016,044       0  9,999,810       1       1  /usr/include/c++/8.2.1/bits/move.h
 9,999,961            0       0  2,999,983       1       1  /usr/include/c++/8.2.1/bits/stl_al

 5,000,004            0       0  2,000,004  62,500  62,500  experiment1.cc:fillData(int*, int)
10,000,008    1,055,426     955  1,000,011       0       0  experiment1.cc:sum(int const*,
                                                              int const*, unsigned long)

 8,999,829            0       0  5,999,886       0       0  /usr/include/c++/8.2.1/bits/move.h
 3,999,924            0       0  3,999,924       0       0  /usr/include/c++/8.2.1/bits/stl_al

 6,000,004            0       0  1,000,004  62,501  62,500  experiment1.cc:
                                                              fillIndex(int*, int)

 1,999,999            0       0  1,999,999       0       0  ???:rand
 2,000,184           24       8         51       1       1  ???:???
   211,981        3,036   2,198     70,177      20       8  ???:_dl_lookup_symbol_x
   271,820        5,647     954    121,026      19      12  ???:do_lookup_x
   103,223        6,463   4,158     36,526   1,831     990  ???:_dl_relocate_object
```

Some Calculations

- Array size: 1 000 000 integers, 4 B each $\Rightarrow$ 4 000 000 B
- One cache line holds 64 B (i.e. 16 integers).
  $\Rightarrow$ Array occupies $\frac{4\,000\,000\,\text{B}}{64\,\text{B}} = 62\,500$ cache lines.
- L1 cache size: 32 KiB = 32 768 B
  $\Rightarrow \frac{32\,768\,\text{B}}{4\,000\,000\,\text{B}} \approx 0.82\,\%$ fit in the cache, $\approx 99.2\,\%$ do not.
- Sequential access produces 62 500 cache misses because each consecutive block of 16 integers resides in one cache line. Only the first yields a cache miss, the following 15 are already present when they are requested.
- Random access requests integers from different cache lines that are not necessarily present in the cache. 992 924 L1 cache misses were measured.

Simple Model for the Expexted Number of Cache Misses

- Cache size $s = 32\,\text{KiB} = 32\,768\,\text{B}$,
  data size $d = 4\,000\,000\,\text{B}$,
  number of accesses $n = 10^6$.
- Assumptions:
  1. independent, uniformly distributed random accesses
  2. "warm" cache
- Probability for one cache hit: $p_{\text{hit}} = \frac{s}{d} = \frac{32\,768\,\text{B}}{4\,000\,000\,\text{B}} \approx 0.82\,\%$
- Number of hits for n independent uniform random accesses: $n \cdot p_{\text{hit}}$
- Number of cache misses: $n - n \cdot p_{\text{hit}} = n \cdot (1 - p_{\text{hit}})$
- In our case: $n = 10^6 \Rightarrow 10^6 \cdot (1 - \frac{32\,768\,\text{B}}{4\,000\,000\,\text{B}}) \approx 991\,808$ cache misses.
- Why does this number differ from the actual measurement (992 924)?
- How could you change the implementation so it matches the assumptions better?

Task 2

In this exercise, we implement some functions using SIMD instructions to speedup computation. First, use non-SIMD instructions to implement the functions in a naïve, unoptimized way. Then, reimplement your functions using 128- or 256-bit SIMD instructions. What performance gain do you expect? Measure the execution time of your functions using different compiler optimization settings.

a) Implement a function that takes two integer arrays by pointer and adds them component-wise into a third array. To allow for dynamic allocation at runtime, the arrays should be stored on the heap.

b) Implement a function that takes one integer array representing an n-dimensional vector and computes this vector's Euclidian norm.

Notes

- Compile example
  clang++
  -std=c++17 -Wall -Wextra
  -march=native -O0 -fno-tree-vectorize
  -o simd_main main.cc simd.cc
- Check compiler support for SSE/AVX
  g++ -Q --help=target | grep -P "\-m(sse|avx).*\[enabled\]"
- Intel Intrinsics Guide
  https://software.intel.com/sites/landingpage/IntrinsicsGuide/#

Measurements

Vector Add

Array size: 10 000 000

Optimization O0

```
'Add' runtime without SIMD in ms:    33.287
'Add' runtime with SSE in ms:        11.171
'Add' runtime with AVX in ms:         7.548
```

Optimization O3

```
'Add' runtime without SIMD in ms:    15.898
'Add' runtime with SSE in ms:         6.365
'Add' runtime with AVX in ms:         6.152
```

Intel(R) Core(TM) i5-6500 CPU @ 3.20 GHz (Linux numenor 4.20.7-arch1-1-ARCH)

Measurements

Vector Norm

Optimization O0

Array size: 10 000 000

```
'Norm' result value:                          181219.826
'Norm' runtime without SIMD in ms:  26.036  (Val: 180468.578)
'Norm' runtime with SSE in ms:      10.496  (Val: 181257.812)
'Norm' runtime with SSEx2 in ms:     9.878  (Val: 181377.188)
'Norm' runtime with AVX in ms:       6.248  (Val: 181399.828)
```

Optimization O3

```
'Norm' result value:                          181219.826
'Norm' runtime without SIMD in ms:  11.439  (Val: 180468.578)
'Norm' runtime with SSE in ms:       2.865  (Val: 181257.812)
'Norm' runtime with SSEx2 in ms:     1.644  (Val: 181377.188)
'Norm' runtime with AVX in ms:       1.785  (Val: 181399.828)
```

Intel(R) Core(TM) i5-6500 CPU @ 3.20 GHz (Linux numenor 4.20.7-arch1-1-ARCH)