Group Information:

Liu Feifei
Wang Yuxi
Li Shulin

CPU:

Intel(R) Core(TM)2 Duo CPU E8400 @ 3.00GHz

Code 1:

```
/* asmstride.c

   This program accesses memory in specific patterns and
   measure the average memory access time.

   Bug reprot to Gu Lin
*/

#include <inttypes.h>
#include <stdio.h>

#define CACHE_MAX (512*1024*1024)       /* largest cache size in byte */
#define TOTAL_ACCESS 5*100000000ULL /* You may change this constant */

char memory[CACHE_MAX];

/* Global variables
  */
int64_t ticks0, ticks1, ticks2; /* to record time ticks */
int64_t low, high, stride;          /* to control memory accesses */
int64_t total8, mock_total8;

/* Read the processor's ticks counter as an approximate
     time measurement.
*/
static __inline__ int64_t rdtsc_ticks(void)
{
   unsigned a, d;

   /* The processor increments its ticks counter every clock cycle
        and resets it to 0 whenever the processor is reset.
        "rdtsc" is an assembly instruction to load the processor's
        current ticks counter into the EDX:EAX registers.
   */
   __asm__ __volatile__("rdtsc" : "=a" (a), "=d" (d));
```

```
    return ((uint64_t)a) | (((uint64_t)d) << 32);
}

/* This function reads 64-bit data items from memory[low] to memory[high].

     Memory is accessed in such a sequence:

        memory[low+896], memory[low+768], ..., memory[low],
        memory[low+stride+896], memory[low+stride+768], ..., memory[low+stride],
        memory[low+2*stride+896], memory[low+2*stride+768], ..., memory[low+2*stride],
        ...
        ...
        ...
        memory[low+k*stride+896], memory[low+k*stride+768], ..., memory[low+k*stride],

        until low+k*stride>= high.

     This function is written in inline assembly in order to
     avoid uncertainty introduced by the compilers. You can
     read asmstride.lst to see what the assembly statements
     are translated to. Load asmstride.list in your favorate editor
     and search for this function name 'measure1'.

     You can find more information on the inline assembly for gcc at:
        http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html
  */
int64_t measure1() {
    int64_t i64a = 0;
    int64_t p = (int64_t)memory;
    int64_t t8;

    t8 = total8;

    // Inline assembly codes
    __asm__ __volatile__(
                                "access1K: \n"

                                "cmpq %6, %5 \n"            // compare low+k*stride and high
                                "jge end_access1K\n"    // if (low+k*stride >= high) then goto
end_access1K

                                "addq 896(%4), %1 \n" // access memory[low+k*stride+896]
using addq
```

```
                        "addq 768(%4), %1 \n" // access memory[low+k*stride+768]
using addq
                        "addq 640(%4), %1 \n" // access memory[low+k*stride+640]
using addq
                        "addq 512(%4), %1 \n" // access memory[low+k*stride+512]
using addq
                        "addq 384(%4), %1 \n" // access memory[low+k*stride+384]
using addq
                        "addq 256(%4), %1 \n" // access memory[low+k*stride+256]
using addq
                        "addq 128(%4), %1 \n" // access memory[low+k*stride+128]
using addq
                        "addq (%4), %1 \n"      // access memory[low+k*stride] using
addq

                        "addq $1, %2 \n"        // increment a counter (t8)
                        "addq %7, %5 \n"        // add stride to the register holding
low+k*stride
                        "addq %7, %4 \n"        // p += stride
                        "jmp access1K \n"
                        "end_access1K: \n"

                /* set up output registers, so that
                        %0 denotes a register used for p,
                        %1 denotes a register used for i64a
                        %2 denotes a register used for t8
                        %3 denotes a register used for low */
                                : "=r"(p), "=d"(i64a), "=r"(t8), "=r"(low)

                /* set up input registers, so that
                        %4 denotes a register used for p (same as %0),
                        %5 denotes a register used for low (same as %3)
                        %6 denotes a register used for high
                        %7 denotes a register used for stride
                        %8 denotes a register used for t8 (same as %2)
                        %9 denotes a register used for i64a (same as %1) */
                                : "0"(p), "3"(low), "r"(high), "r"(stride), "2"(t8), "1"(i64a)
                                : "%rax"
                                );

   total8 = t8;
   return i64a;
}
```

```
/* Mock the iteration overhead
  */
int64_t overhead_measure1() {
   int64_t i64a;
   int64_t p = (int64_t)memory;
   int64_t t8;

   t8 = mock_total8;

   // Embedded assembly codes
   __asm__ __volatile__(
                                "mock_access1K: \n"

                                "cmpq %6, %5 \n"
                                "jge mock_end_access1K\n"

                                "addq $1, %2 \n"
                                "addq %7, %5 \n"
                                "addq %7, %4 \n"
                                "jmp mock_access1K \n"
                                "mock_end_access1K: \n"

                                : "=r"(p), "=d"(i64a), "=r"(t8), "=r"(low)
                                : "0"(p), "3"(low), "r"(high), "r"(stride), "2"(t8)
                                : "%rax"
                                );

   mock_total8 = t8;
   return i64a;
}

/* Measure the access time
  */
double measure_access_time(int32_t low_index,
                                     int32_t high_index,
                                     int32_t stride_value,
                                     int64_t total_accesses) {

   int32_t strides_per_loop = (high_index - low_index) / stride_value;
   int32_t total_loops = total_accesses / 8.0 / strides_per_loop;

   printf("%ld @ %ld x 8 x %ld = %ld ",
            high_index, stride_value, total_loops, total_accesses);
```

```
    // Variable initialization
    total8 = mock_total8 = 0;
    high = high_index;
    stride = stride_value;

    // Record the starting time
    ticks0 = rdtsc_ticks();

    // Perform memory accesses
    register int32_t count = 0;
    while (count++ < total_loops) {
        low = low_index;
        measure1();
    }
    ticks1 = rdtsc_ticks(); // Record the end time 1

    // Mock the iteration overhead
    count = 0;
    while (count++ < total_loops) {
        low = low_index;
        overhead_measure1();
    }
    ticks2 = rdtsc_ticks(); // Record the end time 2

    int64_t d10 = ticks1 - ticks0; // The total running time
    int64_t d21 = ticks2 - ticks1; // The time of the iteration overhead
    int64_t d_mem_ticks = d10 - d21; // The time of memory accesses only (approximately)

    // Calculate the average time of each memory access
    double ticks_per_read = (double)d_mem_ticks/total8/8;
    printf("    ... avg: %e\n", ticks_per_read);
    return ticks_per_read;
}

int main(int argc, char **argv) {
    int64_t tick;
    tick = rdtsc_ticks();

    // Warm up
    int32_t i;
    for (i = 0; i < CACHE_MAX; i += 128) {
        memory[i] = (i >> 7);
    }
```

```
    printf("Starting at tick %lld to probe memory performance\n\n", tick);
    printf("probe_size @ stride x 8 x iterations = number of memory reads    ... avg: ticks per read\n");
    double ticks_per_read;

    // Here are some examples of how to use the function "measure_access_time"

    // Read from memory[0] to memory[8192] with stride 4096 for TOTAL_ACCESS memory accesses
    //ticks_per_read = measure_access_time(0, 8192,                    4096, TOTAL_ACCESS);
    // Read from memory[0] to memory[16384] with stride 1024 for TOTAL_ACCESS memory accesses
    //ticks_per_read = measure_access_time(0, 16ULL*1024,          1024, TOTAL_ACCESS);

    // START adding code here.
    // You may add your more calls to measure_access_time() with different parameters below
    // In addition to adding code in this section, you may change the constant
    // TOTAL_ACCESS at the beginning.

    int64_t size;
    for (size=1024; size<=512ULL*1024*1024; size*=2)
ticks_per_read = measure_access_time(0, size, 1024, TOTAL_ACCESS);

    return;
}
```

Code 2:

```
/* asmstride.c

    This program accesses memory in specific patterns and
    measure the average memory access time.

    Bug reprot to Gu Lin
*/

#include <inttypes.h>
#include <stdio.h>

#define CACHE_MAX (512*1024*1024)        /* largest cache size in byte */
#define TOTAL_ACCESS 5*100000000ULL /* You may change this constant */
```

```
char memory[CACHE_MAX];

/* Global variables
  */
int64_t ticks0, ticks1, ticks2; /* to record time ticks */
int64_t low, high, stride;           /* to control memory accesses */
int64_t total8, mock_total8;

/* Read the processor's ticks counter as an approximate
     time measurement.
*/
static __inline__ int64_t rdtsc_ticks(void)
{
   unsigned a, d;

   /* The processor increments its ticks counter every clock cycle
        and resets it to 0 whenever the processor is reset.
        "rdtsc" is an assembly instruction to load the processor's
        current ticks counter into the EDX:EAX registers.
   */
   __asm__ __volatile__("rdtsc" : "=a" (a), "=d" (d));
   return ((uint64_t)a) | (((uint64_t)d) << 32);
}

/* This function reads 64-bit data items from memory[low] to memory[high].

     Memory is accessed in such a sequence:

        memory[low+896], memory[low+768], ..., memory[low],
        memory[low+stride+896], memory[low+stride+768], ..., memory[low+stride],
        memory[low+2*stride+896], memory[low+2*stride+768], ..., memory[low+2*stride],
        ...
        ...
        ...
        memory[low+k*stride+896], memory[low+k*stride+768], ..., memory[low+k*stride],

        until low+k*stride>= high.

     This function is written in inline assembly in order to
     avoid uncertainty introduced by the compilers. You can
     read asmstride.lst to see what the assembly statements
     are translated to. Load asmstride.list in your favorate editor
     and search for this function name 'measure1'.
```

```
    You can find more information on the inline assembly for gcc at:
        http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html
 */
int64_t measure1() {
   int64_t i64a = 0;
   int64_t p = (int64_t)memory;
   int64_t t8;

   t8 = total8;

   // Inline assembly codes
   __asm__ __volatile__(
                              "access1K: \n"

                              "cmpq %6, %5 \n"           // compare low+k*stride and high
                              "jge end_access1K\n"    // if (low+k*stride >= high) then goto
end_access1K

                              "addq 896(%4), %1 \n" // access memory[low+k*stride+896]
using addq
                              "addq 768(%4), %1 \n" // access memory[low+k*stride+768]
using addq
                              "addq 640(%4), %1 \n" // access memory[low+k*stride+640]
using addq
                              "addq 512(%4), %1 \n" // access memory[low+k*stride+512]
using addq
                              "addq 384(%4), %1 \n" // access memory[low+k*stride+384]
using addq
                              "addq 256(%4), %1 \n" // access memory[low+k*stride+256]
using addq
                              "addq 128(%4), %1 \n" // access memory[low+k*stride+128]
using addq
                              "addq (%4), %1 \n"       // access memory[low+k*stride] using
addq

                              "addq $1, %2 \n"          // increment a counter (t8)
                              "addq %7, %5 \n"          // add stride to the register holding
low+k*stride
                              "addq %7, %4 \n"          // p += stride
                              "jmp access1K \n"
                              "end_access1K: \n"

                     /* set up output registers, so that
                           %0 denotes a register used for p,
```

```
                        %1 denotes a register used for i64a
                        %2 denotes a register used for t8
                        %3 denotes a register used for low */
                                : "=r"(p), "=d"(i64a), "=r"(t8), "=r"(low)

                    /* set up input registers, so that
                        %4 denotes a register used for p (same as %0),
                        %5 denotes a register used for low (same as %3)
                        %6 denotes a register used for high
                        %7 denotes a register used for stride
                        %8 denotes a register used for t8 (same as %2)
                        %9 denotes a register used for i64a (same as %1) */
                                : "0"(p), "3"(low), "r"(high), "r"(stride), "2"(t8), "1"(i64a)
                                : "%rax"
                                );

    total8 = t8;
    return i64a;
}

/* Mock the iteration overhead
  */
int64_t overhead_measure1() {
    int64_t i64a;
    int64_t p = (int64_t)memory;
    int64_t t8;

    t8 = mock_total8;

    // Embedded assembly codes
    __asm__ __volatile__(
                                "mock_access1K: \n"

                                "cmpq %6, %5 \n"
                                "jge mock_end_access1K\n"

                                "addq $1, %2 \n"
                                "addq %7, %5 \n"
                                "addq %7, %4 \n"
                                "jmp mock_access1K \n"
                                "mock_end_access1K: \n"

                                : "=r"(p), "=d"(i64a), "=r"(t8), "=r"(low)
                                : "0"(p), "3"(low), "r"(high), "r"(stride), "2"(t8)
```

```
                                : "%rax"
                                );

    mock_total8 = t8;
    return i64a;
}

/* Measure the access time
  */
double measure_access_time(int32_t low_index,
                                        int32_t high_index,
                                        int32_t stride_value,
                                        int64_t total_accesses) {

    int32_t strides_per_loop = (high_index - low_index) / stride_value;
    int32_t total_loops = total_accesses / 8.0 / strides_per_loop;

    printf("%ld @ %ld x 8 x %ld = %ld ",
                high_index, stride_value, total_loops, total_accesses);

    // Variable initialization
    total8 = mock_total8 = 0;
    high = high_index;
    stride = stride_value;

    // Record the starting time
    ticks0 = rdtsc_ticks();

    // Perform memory accesses
    register int32_t count = 0;
    while (count++ < total_loops) {
        low = low_index;
        measure1();
    }
    ticks1 = rdtsc_ticks(); // Record the end time 1

    // Mock the iteration overhead
    count = 0;
    while (count++ < total_loops) {
        low = low_index;
        overhead_measure1();
    }
    ticks2 = rdtsc_ticks(); // Record the end time 2
```

```
    int64_t d10 = ticks1 - ticks0; // The total running time
    int64_t d21 = ticks2 - ticks1; // The time of the iteration overhead
    int64_t d_mem_ticks = d10 - d21; // The time of memory accesses only (approximately)

    // Calculate the average time of each memory access
    double ticks_per_read = (double)d_mem_ticks/total8/8;
    printf("    ... avg: %e\n", ticks_per_read);
    return ticks_per_read;
}

int main(int argc, char **argv) {
    int64_t tick;
    tick = rdtsc_ticks();

    // Warm up
    int32_t i;
    for (i = 0; i < CACHE_MAX; i += 128) {
        memory[i] = (i >> 7);
    }

    printf("Starting at tick %lld to probe memory performance\n\n", tick);
    printf("probe_size @ stride x 8 x iterations = number of memory reads    ... avg: ticks per
read\n");
    double ticks_per_read;

    // Here are some examples of how to use the function "measure_access_time"

    // Read from memory[0] to memory[8192] with stride 4096 for TOTAL_ACCESS memory
accesses
    //ticks_per_read = measure_access_time(0, 8192,                    4096, TOTAL_ACCESS);
    // Read from memory[0] to memory[16384] with stride 1024 for TOTAL_ACCESS
memory accesses
    //ticks_per_read = measure_access_time(0, 16ULL*1024,          1024,
TOTAL_ACCESS);

    // START adding code here.
    // You may add your more calls to measure_access_time() with different parameters below
    // In addition to adding code in this section, you may change the constant
    // TOTAL_ACCESS at the beginning.

int64_t size;
    for (size=1024*1024; size<=8*1024*1024; size+=0.5*1024*1024)
      ticks_per_read = measure_access_time(0, size,1024, TOTAL_ACCESS);
```

```
    return;
}
```

Analysis

In principle, the cache has impact on the memory access time. If we read a small amount of memory, it's likely that all memory accesses hit the cache. Then the memory access time is small. When the amount of memory is very large, there will be very possibly many cache misses, the memory access time is larger. Thus, we can conclude that the point at which ticks per read start to increase sharply with the memory size is when the memory size accessed starts to exceed the cache size. Moreover, the smooth curve before such a point shows very much the size of the cache since the memory size accessed approximates the cache size at the time.

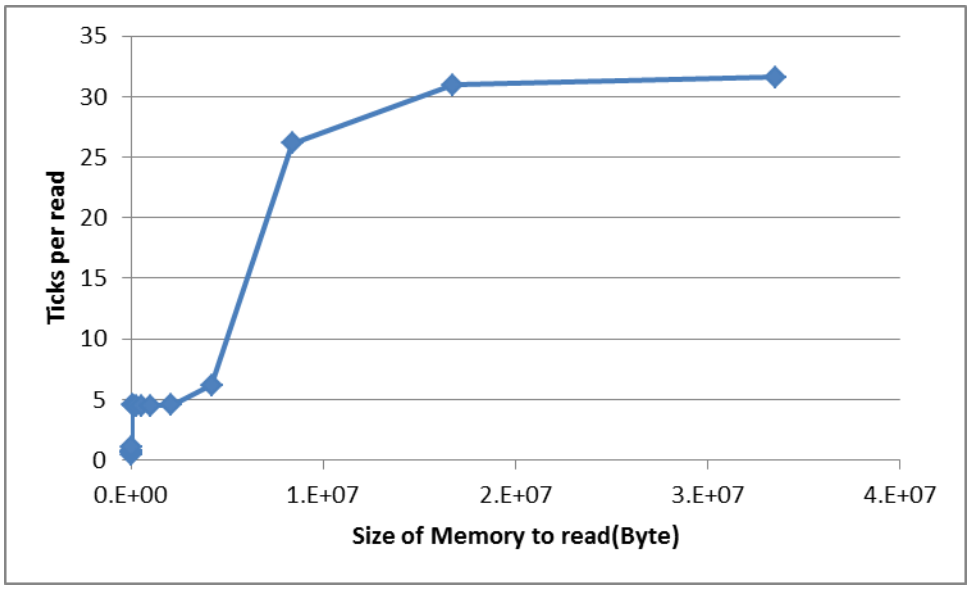
Firstly, we start to run with code1 to access different sizes of memory in order to obtain an overall view of cache size. We started to access the memory from 1KB. After each memory access finished, we double the size accessed. It means that we start to read from memory[0] to memory[1024] in first run, and we will read from memory[0] to memory[2048] in second run, and read from memory[0] to memory[1024*4] in third run. We repeated the steps until the memory size equals 512MB.

After it, we get the following data and plot the corresponding graph.

Data 1:

```
probe_size @ stride x 8 x iterations = number of memory reads  ... avg: ticks pe
r read
1024 @ 1024 x 8 x 62500000 = 500000000   ... avg: 6.097441e-01
2048 @ 1024 x 8 x 31250000 = 500000000   ... avg: 4.087535e-01
4096 @ 1024 x 8 x 15625000 = 500000000   ... avg: 6.384004e-01
8192 @ 1024 x 8 x 7812500 = 500000000   ... avg: 6.857606e-01
16384 @ 1024 x 8 x 3906250 = 500000000   ... avg: 7.504603e-01
32768 @ 1024 x 8 x 1953125 = 500000000   ... avg: 1.020353e+00
65536 @ 1024 x 8 x 976562 = 500000000   ... avg: 4.518849e+00
131072 @ 1024 x 8 x 488281 = 500000000   ... avg: 4.512062e+00
262144 @ 1024 x 8 x 244140 = 500000000   ... avg: 4.496054e+00
524288 @ 1024 x 8 x 122070 = 500000000   ... avg: 4.491946e+00
1048576 @ 1024 x 8 x 61035 = 500000000   ... avg: 4.492732e+00
2097152 @ 1024 x 8 x 30517 = 500000000   ... avg: 4.531671e+00
4194304 @ 1024 x 8 x 15258 = 500000000   ... avg: 6.197533e+00
8388608 @ 1024 x 8 x 7629 = 500000000   ... avg: 2.612898e+01
16777216 @ 1024 x 8 x 3814 = 500000000   ... avg: 3.095047e+01
33554432 @ 1024 x 8 x 1907 = 500000000   ... avg: 3.159193e+01
67108864 @ 1024 x 8 x 953 = 500000000   ... avg: 3.133279e+01
134217728 @ 1024 x 8 x 476 = 500000000   ... avg: 3.094079e+01
268435456 @ 1024 x 8 x 238 = 500000000   ... avg: 3.073525e+01
536870912 @ 1024 x 8 x 119 = 500000000   ... avg: 3.061844e+01
```

<u>Figure 1</u>:only plot the first 16 points)



From Data1 and Figure1 shown above, we can estimate that the cache size is between 1 MB and 8 MB, we change the source code of asmstride.c to code2 in order to get the more accurate estimation of the cache size.

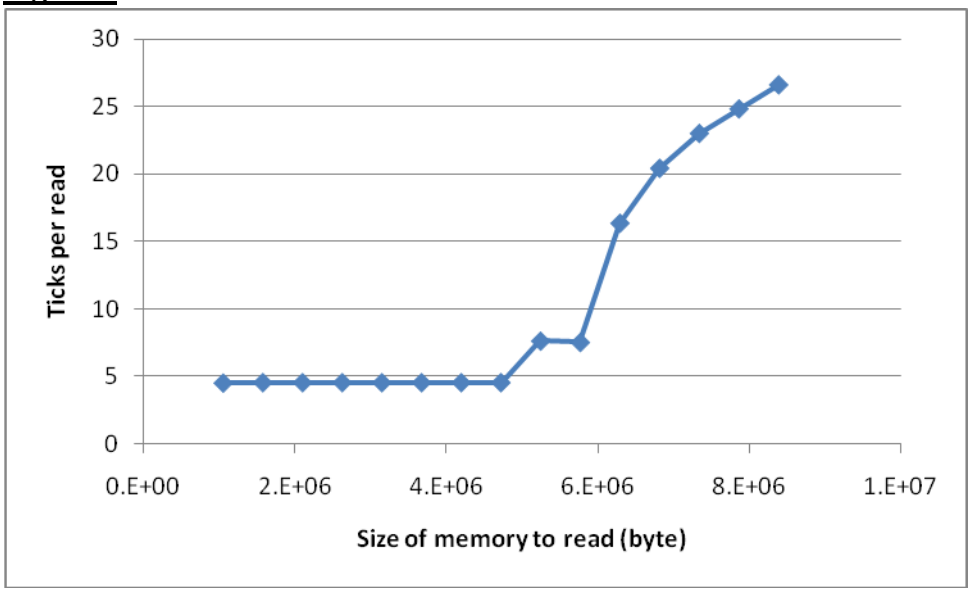
In the second part of the test, we started to access the memory from 1MB. After each memory access finished, we added another 0.5 MB to the current accessed memory size, and then accessed the memory again. It means that we start to read from memory[0] to memory[1024*1024] in first run, and we will read from memory[0] to memory[1024*1024*1.5] in second run, and read from memory[0] to

memory[1024*1024*2] in third run. We repeated the steps until the memory size equals 8MB, got our test data and plot a figure to show the variation trend from 1MB to 8MB. Since there will be oscillation for the test data, we did several times of test and gathered the data and the figures which have the similar variation trend for estimation.

Data 2:

```
1048576 @ 1024 x 8 x 61035 = 500000000   ... avg: 4.496180e+00
1572864 @ 1024 x 8 x 40690 = 500000000   ... avg: 4.531017e+00
2097152 @ 1024 x 8 x 30517 = 500000000   ... avg: 4.528931e+00
2621440 @ 1024 x 8 x 24414 = 500000000   ... avg: 4.531032e+00
3145728 @ 1024 x 8 x 20345 = 500000000   ... avg: 4.527252e+00
3670016 @ 1024 x 8 x 17438 = 500000000   ... avg: 4.529102e+00
4194304 @ 1024 x 8 x 15258 = 500000000   ... avg: 4.530161e+00
4718592 @ 1024 x 8 x 13563 = 500000000   ... avg: 4.530083e+00
5242880 @ 1024 x 8 x 12207 = 500000000   ... avg: 7.613528e+00
5767168 @ 1024 x 8 x 11097 = 500000000   ... avg: 7.510891e+00
6291456 @ 1024 x 8 x 10172 = 500000000   ... avg: 1.635723e+01
6815744 @ 1024 x 8 x 9390 = 500000000   ... avg: 2.043204e+01
7340032 @ 1024 x 8 x 8719 = 500000000   ... avg: 2.302093e+01
7864320 @ 1024 x 8 x 8138 = 500000000   ... avg: 2.484236e+01
8388608 @ 1024 x 8 x 7629 = 500000000   ... avg: 2.662301e+01
```

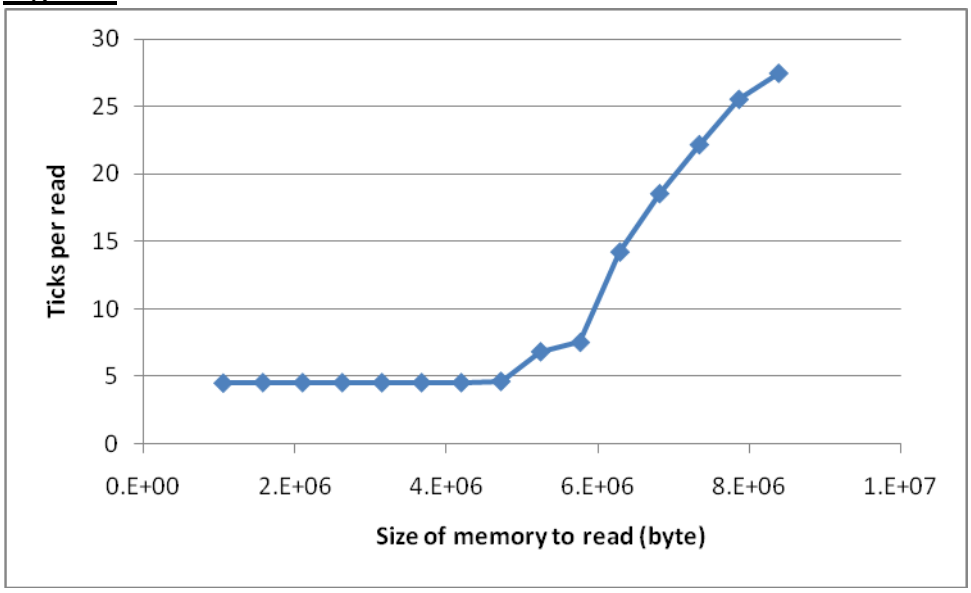
Figure 2:



Data 3:

```
1048576 @ 1024 x 8 x 61035 = 500000000   ... avg: 4.490792e+00
1572864 @ 1024 x 8 x 40690 = 500000000   ... avg: 4.532014e+00
2097152 @ 1024 x 8 x 30517 = 500000000   ... avg: 4.527899e+00
2621440 @ 1024 x 8 x 24414 = 500000000   ... avg: 4.527609e+00
3145728 @ 1024 x 8 x 20345 = 500000000   ... avg: 4.527967e+00
3670016 @ 1024 x 8 x 17438 = 500000000   ... avg: 4.527070e+00
4194304 @ 1024 x 8 x 15258 = 500000000   ... avg: 4.528499e+00
4718592 @ 1024 x 8 x 13563 = 500000000   ... avg: 4.618977e+00
5242880 @ 1024 x 8 x 12207 = 500000000   ... avg: 6.821572e+00
5767168 @ 1024 x 8 x 11097 = 500000000   ... avg: 7.514287e+00
6291456 @ 1024 x 8 x 10172 = 500000000   ... avg: 1.421787e+01
6815744 @ 1024 x 8 x 9390 = 500000000   ... avg: 1.854687e+01
7340032 @ 1024 x 8 x 8719 = 500000000   ... avg: 2.217843e+01
7864320 @ 1024 x 8 x 8138 = 500000000   ... avg: 2.554824e+01
8388608 @ 1024 x 8 x 7629 = 500000000   ... avg: 2.747655e+01
```

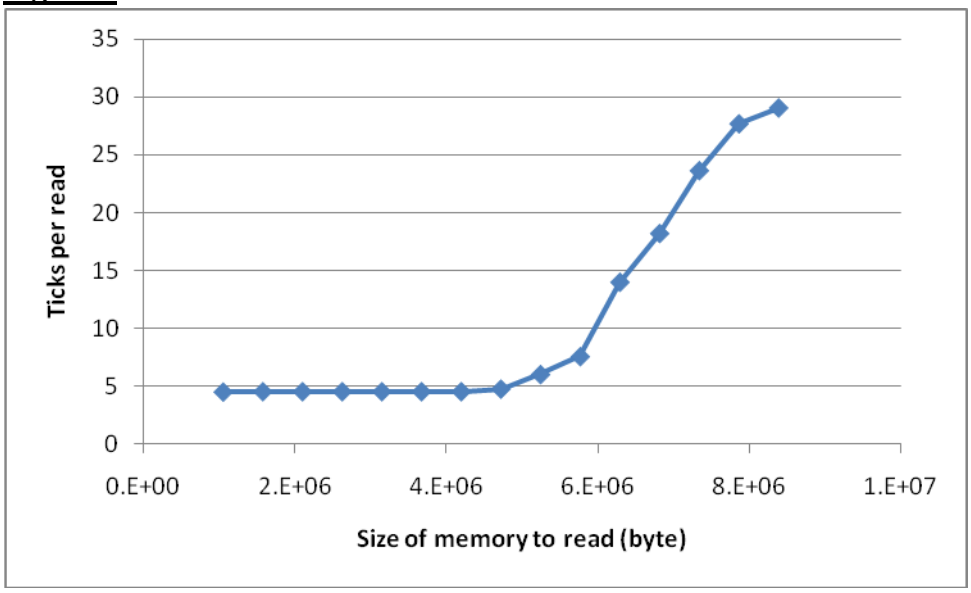
Figure 3:



Data 4:

```
1048576 @ 1024 x 8 x 61035 = 500000000   ... avg: 4.496469e+00
1572864 @ 1024 x 8 x 40690 = 500000000   ... avg: 4.531788e+00
2097152 @ 1024 x 8 x 30517 = 500000000   ... avg: 4.529482e+00
2621440 @ 1024 x 8 x 24414 = 500000000   ... avg: 4.530857e+00
3145728 @ 1024 x 8 x 20345 = 500000000   ... avg: 4.529424e+00
3670016 @ 1024 x 8 x 17438 = 500000000   ... avg: 4.528974e+00
4194304 @ 1024 x 8 x 15258 = 500000000   ... avg: 4.528736e+00
4718592 @ 1024 x 8 x 13563 = 500000000   ... avg: 4.750312e+00
5242880 @ 1024 x 8 x 12207 = 500000000   ... avg: 6.009410e+00
5767168 @ 1024 x 8 x 11097 = 500000000   ... avg: 7.552542e+00
6291456 @ 1024 x 8 x 10172 = 500000000   ... avg: 1.397609e+01
6815744 @ 1024 x 8 x 9390 = 500000000   ... avg: 1.818385e+01
7340032 @ 1024 x 8 x 8719 = 500000000   ... avg: 2.360591e+01
7864320 @ 1024 x 8 x 8138 = 500000000   ... avg: 2.764926e+01
8388608 @ 1024 x 8 x 7629 = 500000000   ... avg: 2.901880e+01
```

Figure 4:



Since there will be little cache misses when we read a small amount of memory while there will be many cache misses when the amount of memory becomes large, we think the corresponding "Size of memory to read" of the point where the "Ticks per read" start increasing in our figure can present the cache size indirectly. From the figures, we can find that the "Ticks per read" start increasing at a point whose "Size of memory to read" is around between 5.5MB and 6.0MB. Since there may be someone else using the same computer at the time we execute our testing program. Therefore, the real cache size may be slightly larger than the one shown in the graph. Thus, we think 6MB is an appropriate and meaningful estimation of the cache size.

Therefore, we estimate that the cache size for the CPU is 6MB.