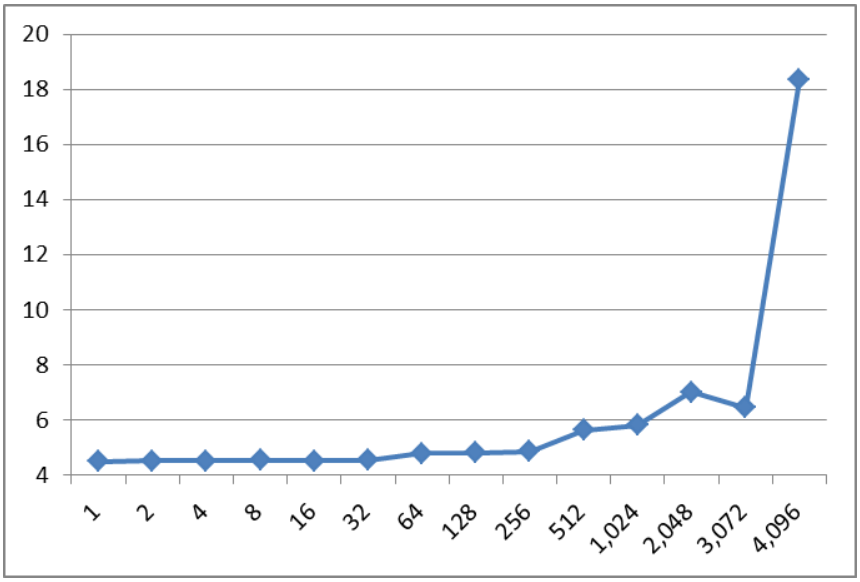
Yung-Chih Chen

1. Testing the performance of my computer using C++

   I used arrays of different sizes and read them, using timing to differentiate the sizes of L1, L2, and L3. To avoid prefetch, the program read and wrote the values of the array every 64 bytes. The result is as follows. [3.1 C++ source code]

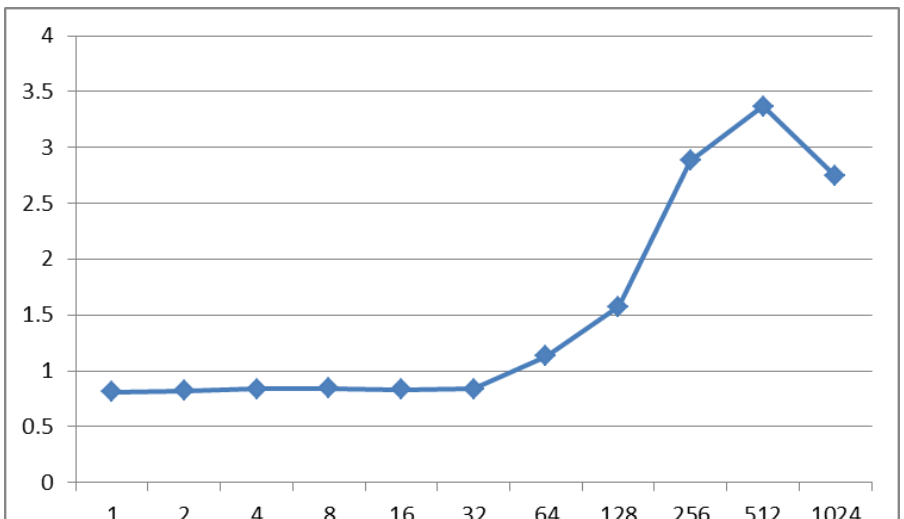
Size (KB)	Time
1	4.490150
2	4.523056
4	4.520750
8	4.543643
16	4.512092
32	4.539567
64	4.782894
128	4.811265
256	4.862778
512	5.627449
1024	5.806925
2048	7.001900
3072	6.451247
4096	18.35128



From the above graph, I could know that the size of data cache L1 is 32KB and the size of L2 is 256KB. When the size of array was more 3072KB, the read and write time would increase significantly. Thus the size of cache L3 is 3072KB.

To obtain cache line size, this program measured the time in which it read and wrote the value of the array, when the index increased certain amount. Before it accessed the values, the cache should be flushed. The result is as follows.

Byte	Time
1	0.812432
2	0.820624
4	0.836255
8	0.843048
16	0.831703
32	0.840117



64	1.131680
128	1.574819
256	2.884853
512	3.369160
1024	2.743147

From the above graph, I could guess that the block size is between 32 bytes and 128 bytes.

The single-core sequential throughput and the single-core sequential throughput of main memory could be obtained by below C++ source code (3.3). First, the cache was flushed to cause miss. Then, I measured the time in which the data was accessed sequentially and the time in which it was randomly. The result is as follows.
Sequentially reading and writing throughput: 512MB / 8.98s
Randomly reading and writing throughput: 512MB / 8.99s

If I change the page size other than 4kB, TLB miss or hit rate would be changed. Therefore, Timing would respond to the change. In this regard, I test three cases, #define PAGE_SHIFT 12, #define PAGE_SHIFT 14, #define PAGE_SHIFT 16.

From the tools, x865info and cupid, TLB and the associativity of the cache were shown above.

2. The Computer in practice
   I chose an x86-64bit processor and run Linux as a guest VM. The detail information I got from the tools, x86info and cpuid, is as follows.
   CPU Clock Speed: 2.1GHz Main Memory: 1GB
   Instruction Cache L1: 32KB Data Cache L1:32KB
   Cache L2: 256KB Cache L3: 3072KB
   Page Size Extensions Hardware Prefetch
   7 ways associative cache
   TLB information:
   Instruction TLB: 4K pages, 4-way associative, 64 entries.
   Data TLB: 4KB or 4MB pages, fully associative, 32 entries.
   Data TLB: 4KB pages, 4-way associative, 64 entries
   Data TLB: 4K pages, 4-way associative, 512 entries.
   Data TLB: 4KB or 4MB pages, fully associative, 32 entries.

3. Code

3.1

```
#include <stdint.h>
#include <cstdio>
#include <sys/mman.h>
#include <time.h>

const int k = 1024;
const int data_size = 12*k*k;

uint64_t read_tsc(void) {
    uint64_t hi, lo;
    /*
     * Embed the assembly instruction 'rdtsc', which should not be relocated (`volatile').
     *    The instruction will modify r_a_x and r_d_x, which the compiler should map to
     * lo and hi, respectively.
     *
     * The format for GCC-style inline assembly generally is:
     * __asm__ ( ASSEMBLY CODE : OUTPUTS : INPUTS : OTHER THINGS CHANGED )
     *
     * Note that if you do not (correctly) specify the side effects of an assembly operation, the
     * compiler may assume that other registers and memory are not affected. This can easily
     * lead to cases where your program will produce difficult-to-debug wrong answers when
     * optimizations are enabled.
     */
    __asm__ volatile ( "rdtsc" : "=a"(lo), "=d"(hi));
    return lo | (hi << 32);
}

int main() {
    void *map = mmap(NULL, 12*1024*1024, PROT_READ | PROT_WRITE,
```

```
MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);
	int *data = (int*)map;
	for (int i = 0;i< 12*1024*1024/sizeof(int);i++) {
		data[i]++;
	}

	int steps[] = { 1*1024, 2*1024, 4*1024, 8*1024, 16*1024, 32*1024, 64*1024,
128*1024,
				256*1024, 512*1024, 1024*1024, 2*1024*1024,
3*1024*1024, 4*1024*1024, 5*1024*1024, 6*1024*1024 };
	for (int i = 0; i <= 15; i++) {
		double t_total, t_h, t_l;
		t_l = read_tsc();
		for (int m = 0; m < 8; m++) {
			int size = steps[i] / sizeof(int) - 1;
			for (int j = 0; j < 64*k*k; j++) {
				++data[ (j * 64) & size ];
			}
		}
		t_h = read_tsc();
		t_total = (t_h-t_l)/2000000000.0;
		printf("%d time: %lf\n", steps[i]/k, t_total);
	}
	munmap(map, (size_t)data_size);
	return 0;
}
```

3.2

```
#include <stdint.h>
#include <cstdio>
#include <time.h>
#include <sys/mman.h>

const int k = 1024;
const int data_size = 64 *k*k;

int cacheflush(){
	void *map = mmap(NULL, 64*1024*1024, PROT_READ | PROT_WRITE,
```

```
MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);
	int *dummy = (int*)map;
	for (int i = 0;i< 64*1024*1024/sizeof(int);i++) {
		dummy[i]++;
	}
    // munmap(map, (size_t)data_size);
}


uint64_t read_tsc(void) {
	uint64_t hi, lo;
	/*
	 * Embed the assembly instruction 'rdtsc', which should not be relocated (`volatile').
	 *    The instruction will modify r_a_x and r_d_x, which the compiler should map to
	 * lo and hi, respectively.
	 *
	 * The format for GCC-style inline assembly generally is:
	 * __asm__ ( ASSEMBLY CODE : OUTPUTS : INPUTS : OTHER THINGS CHANGED )
	 *
	 * Note that if you do not (correctly) specify the side effects of an assembly operation, the
	 * compiler may assume that other registers and memory are not affected. This can easily
	 * lead to cases where your program will produce difficult-to-debug wrong answers when
	 * optimizations are enabled.
	 */
	__asm__ volatile ( "rdtsc" : "=a"(lo), "=d"(hi));
	return lo | (hi << 32);
}

int main() {

	int* data = new int[64*1024*1024];
	for (int i = 0;i< 64*1024*1024/sizeof(int);i++) {
```

```
            data[i]*=87;
        }
        int steps[] = { 1, 2,4, 8, 16, 32, 64, 128, 256, 512, 1024};
        for (int i = 0; i <= 10; i++) {
            double t_total, t_h, t_l;

            t_l = read_tsc();
                int size = sizeof(64*1024*1024)/sizeof(int) - 1;
                cacheflush();
                for (int j = 0; j < 64*k*k; j++) {
                    ++data[ (steps[i]*j) & size];
            }
            t_h = read_tsc();
            t_total = (t_h-t_l);
            printf("%d time: %lf\n", steps[i], t_total/1000000000);
        }
        return 0;
}
```

3.3

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <cstdio>
#include <time.h>
#include <sys/mman.h>
#include <unistd.h>

const int k = 1024;
const int data_size = 64 *k*k;

int cacheflush(){
        void *map = mmap(NULL, 64*1024*1024, PROT_READ | PROT_WRITE,
MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);
        int *dummy = (int*)map;
        for (int i = 0;i< 64*1024*1024/sizeof(int);i++) {
            dummy[i]++;
```

```
        }
    // munmap(map, (size_t)data_size);
}

uint64_t read_tsc(void) {
        uint64_t hi, lo;
        /*
         * Embed the assembly instruction 'rdtsc', which should not be relocated (`volatile').
         *    The instruction will modify r_a_x and r_d_x, which the compiler should map to
         * lo and hi, respectively.
         *
         * The format for GCC-style inline assembly generally is:
         * __asm__ ( ASSEMBLY CODE : OUTPUTS : INPUTS : OTHER THINGS CHANGED )
         *
         * Note that if you do not (correctly) specify the side effects of an assembly operation, the
         * compiler may assume that other registers and memory are not affected. This can easily
         * lead to cases where your program will produce difficult-to-debug wrong answers when
         * optimizations are enabled.
         */
        __asm__ volatile ( "rdtsc" : "=a"(lo), "=d"(hi));
        return lo | (hi << 32);
}

int main() {

        int* data = new int[64*1024*1024];
        for (int i = 0;i< 64*1024*1024/sizeof(int);i++) {
                data[i]++;
        }
        int random;
                double t_total=0, t_h, t_l;
```

```
        for (int m = 0; m < 8; m++) {
            int size = sizeof(64*1024*1024)/sizeof(int) - 1;
            cacheflush();
            for (int j = 0; j < 64*k*k; j++) {
                random =(rand() % (64*1024*1024-64+1)) +64;
                t_l = read_tsc();
                +data[ (j*random*64) & size];
                t_h = read_tsc();
                t_total = (t_h-t_l)+t_total;
            }

        }
        printf("time: %lf\n", t_total/2000000000);

    return 0;
}
```