3-Level Cache Simulation Using SimpleScalar

CIS 5642 Course Project Final report

Honghao Gan
tuj89290@temple.edu

Zhijia Chen
tuh17884@temple.edu

***Abstract*—This is the final report of CIS 5642 course project. In this project, we show how we modify SimpleScalar toolsets to support 3-level cache with inclusion property, and to support multicore. We also added Least Frequently Used (LFU) replacement policy to the toolsets. We used the modified toolsets to simulate a 3-level cache with single core and a 3-level cache with double cores.

Keywords—cache design; SimpleScalar

I. Project Introduction

The goal of this class projects is to get experience with the design and implementation of a 3-level cache. In this project, we modified SimpleScalar to support 3-level cache and multicore. We then simulated implemented 3-level cache (L1, L2, L3) for both single core and double core using SimpleScalar. L1 cache consists of separate I and D caches, both L2 and L3 are unified caches. They demonstrate inclusion property, i.e, all data in the L1 D-cache are present in L2, and all data in L2 are present in L3 cache.

Our tasks include:

1) Get familiar with Simplescalar. The simulator and its related documents can be downloaded at http://www.simplescalar.com/.

2) Modify the SimpleScalar to support inclusion property and 3-level cache.

3) Modify the SimpleScalar to support multicore.

4) Perform experiments to evaluate the cache design. We developed 2 test programs to collect the cache hit/miss information for the following configurations:

Configuration 1: single core processor

L1 D-cache: 16KB, L1 I- Cache: 16KB, 2-way, block size: 64B

L2 cache: 512KB, 4-way, block size: 64B

L3 cache: 8MB, 8-way, block size: 64B

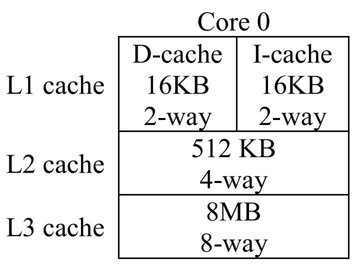

Figure 1. Cache structure for configuration 1

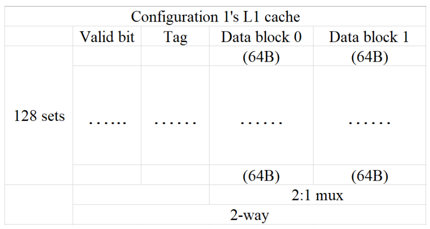

Figure 2. L1 cache design for configuration 1

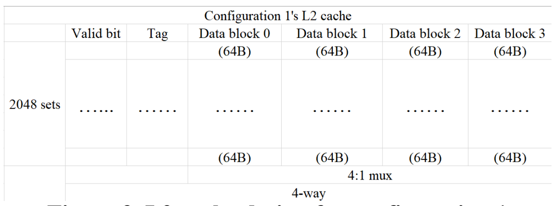

Figure 3. L2 cache design for configuration 1

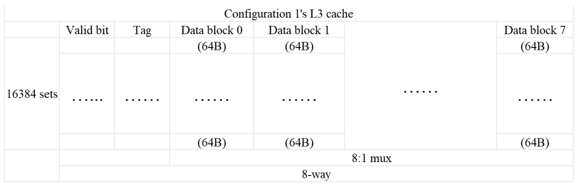

Figure 4. L3 cache design for configuration 1

Configuration 2: Dual core processor, each core has its own L1 and L2 cache and two cores share the L3 cache.

L1 D-cache: 8KB, L1 I- Cache: 8KB, direct mapped, block size: 64B

L2 cache: 128KB, 2-way, block size: 64B,

L3 cache: 16MB, 4-way, block size: 256B

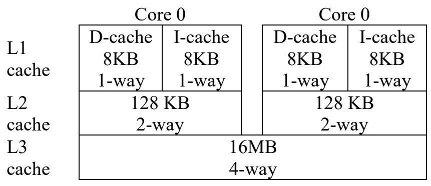

Figure 5. Cache structure for configuration 2

Configuration 2's L1 cache			
	Valid bit	Tag	Data block 0
128 sets			(64B)
	……	……	……
			(64B)
		1:1 mux	
	Direct mapped		

Figure 6. L1 cache design for configuration 2

Configuration 2's L2 cache				
	Valid bit	Tag	Data block 0	Data block 1
1024 sets			(64B)	(64B)
	……	……	……	……
			(64B)	(64B)
			2:1 mux	
	2-way			

Figure 7. L2 cache design for configuration 2

Configuration 2's L3 cache						
	Valid bit	Tag	Data block 0	Data block 1	Data block 2	Data block 3
65536 sets			(64B)	(64B)	(64B)	(64B)
	……	……	……	……	……	……
			(64B)	(64B)	(64B)	(64B)
			4:1 mux			
	4-way					

Figure 8. L3 cache design for configuration 2

II. SimpleScalar Background

SimpleScalar is a toolset that models a virtual computer system with CPU, cache and memory hierarchy. It allows users to build modeling applications that simulate real programs running on a range of modern processors and systems. The toolset includes simulators ranging from a fast-functional simulator to a detailed, dynamically scheduled processor model that supports non-blocking caches, speculative execution, and branch prediction. Figure 9 gives an overview of the SimpleScalar toolset.

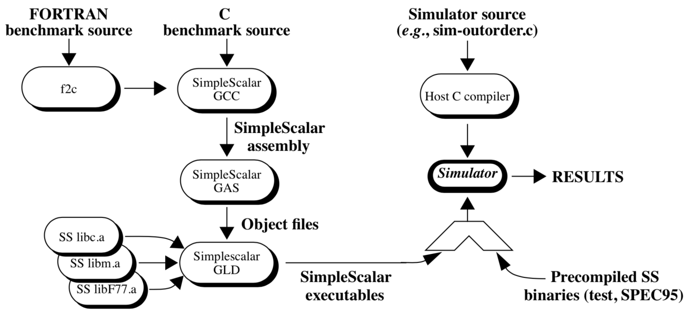

Figure 9. SimpleScalar toolset overview

SimpleScalar can simulate Alpha and PISA (Portable ISA). The PISA instruction set is a simple MIPS-like instruction set maintained primarily for instructional use. The tool set takes binaries compiled for the SimpleScalar architecture and simulates their execution on one of several provided processor simulators. The machine running SimpleScalar is called the Host machine or Host while the ISA that one is targeting such as Alpha or PISA is called Target. Gcc cross-compiler for PISA is available on the internet. We will use gcc cross-compiler in this project.

The toolset provides a collection of microarchitecture simulators that emulate the microprocessor at different levels of details, as listed following.

- **sim-fast**: fast instruction interpreter, optimized for speed. This simulator does not account for the behavior of pipelines, caches, or any other part of the microarchitecture. It performs only functional simulation using in-order execution of the instructions.
- **sim-safe**: slightly slower instruction interpreter, as it checks for memory alignment and memory access permission on all memory operations.
- **sim-profile**: instruction interpreter and profiler. This simulator keeps track of and reports dynamic instruction counts, instruction class counts, usage of address modes, and profiles of the text and data segments.
- **sim-cache**: memory system simulator. This simulator can emulate a system with multiple levels of instruction and data caches, each of which can be configured for different sizes and organizations. Since we are implementing 3-level cache, we are going to use this simulator as it provides sufficient simulation details for studying the cache but still has relatively fast performance.
- **sim-bpred**: branch predictor simulator. This tool can simulate difference branch prediction schemes and reports results such as prediction hit and miss rates. Like sim-cache, this does not simulate accurately the effect of branch prediction on execution time.
- **sim-outorder**: detailed microarchitectural simulator. This tool models in detail and out-of-order microprocessor with all of the bells and whistles, including branch prediction, caches, and external memory. This simulator is highly parameterized and can emulate machines of varying numbers of execution units. As our goal is to simulate a 3-level cache, the sim-cache simulator is used in this project, and our modifications are mainly focus on this simulator’s source codes.

III. Team Member & Contribution

Honghao Gan:

- Modify the SimpleScalar to add LFU set replacement policy.
- Write the testing programs.
- Run benchmarks and collect data.

Zhijia Chen:

- Modify the SimpleScalar to support inclusion property and 3-level cache.
- Modify the SimpleScalar to support multicore.

IV. Design Description

A. 3-Level Cache Support

The official SimpleScalar only support 2 level of caches. We use the following steps to add support for the level 3 cache:

1. Declare L3 cache:

```
static struct cache_t *cache_il1 = NULL;
static struct cache_t *cache_il2 = NULL;
static struct cache_t *cache_il3 = NULL;
static struct cache_t *cache_dl1 = NULL;
static struct cache_t *cache_dl2 = NULL;
static struct cache_t *cache_dl3 = NULL;
```

2. Link L3 I/D-cache to L2 I/D-cache:

```
static unsigned int
il2_access_fn(…)
{
    if (cache_il3)
        return cache_access(cache_il3, …);
    else
        return 1;
}
```

   In the above code, we make L2 cache to access L3 cache if L3 cache pointer is not NULL which means it is configured.

3. L3 cache configuration check

   We extend the cache configuration format to allow user to configure the L3 cache just like what he/she would do with the L1 and L2 cache, and after passing sanity check, we create L3 cache:

```
…
if (!mystricmp(cache_dl3_opt, "none"))
    cache_dl3 = NULL;
else
{
    if (sscanf(cache_dl3_opt, "%[^:]:%d:%d:%d:%c",
name, &nsets, &bsize, &assoc, &c) != 5)
    fatal("bad l3 D-cache parms: "
        "<name>:<nsets>:<bsize>:<assoc>:<repl>");
    cache_dl3 = cache_create(…);
}
…
```

B. Inclusion Property

The inclusive property is that all the block present in the upper level cache should present in the lower level cache as well. The basic idea for the implementation is to add a present counter in the cache block data structure. We then increase the present counter of a block by one whenever the block is read and decrease the counter by one whenever an upper level block that read from this block is replaced. And we will only replace those blocks whose present counters equal to zero. The following codes shows the modified cache block type structure (the useCnt member is used for LFU replacement policy).

```
struct cache_blk_t
{
    …
    unsigned int presentCnt;
    unsigned int useCnt;
    …
}
```

We increase/decrease the present counter in the cache_access function. When there is a cache miss, this function will be called to read a block from lower level cache and replace a block if there is no enough space. So, we can track all the cache read and replacement in this function and make sure the present counter is correct. The following codes shows how we modify the function.

```
unsigned int cache_access(
        struct cache_t *cp,
        enum mem_cmd cmd,
        md_addr_t addr
        …)  /* some argument omitted to save space
*/
{
    …
/* write back replaced block data */
    if (repl->status & CACHE_BLK_VALID)
     {
         …
/* decrease presence counter of the replaced block
by 1 */
        cp->blk_present_fn(-1, CACHE_MK_BADDR(cp,
repl->tag, set));
     }
     …
/* increase presence counter of the accessed block
by 1 */
     cp->blk_present_fn(1, CACHE_BADDR(cp, addr));
     …
}
```

The function blk_present_fn is the function to increase/decrease the present counter of a block, please refer to our source codes for the implementation of this function.

C. Cache Block Replacement Policy(LFU & Random)

The cache.h file defines LRU, LFU, Random and FIFO cache replacement policy:

```
enum cache_policy
{
    LRU, /* replace least recently used block
(perfect LRU) */
    Random, /* replace a random block */
    FIFO, /* replace the oldest block in the set */
    LFU /* replace least frequently used block, LFU
added */
};
```

Those policies are specified in the cache.c file:

```
switch (cp->policy)
{
    case LRU:…
    case FIFO:…
    case Random:
    {
        int bindex = myrand() & (cp->assoc - 1);
        repl = CACHE_BINDEX(cp, cp->sets[set].blks,
bindex);
        while (repl->presentCnt != 0)
        {
            int bindex = myrand() & (cp->assoc - 1);
            repl = CACHE_BINDEX(cp,
cp->sets[set].blks, bindex);
        }
    }
    case LFU: // LFU added
    {         /* Find the least frequently used
block*/
        int min;

        for (blk = cp->sets[set].way_head; blk; blk
= blk->way_next)
        {
            if (blk->presentCnt == 0)
            {
                min = blk->useCnt;
                break;
            }
        }

        for (blk = cp->sets[set].way_head; blk; blk
= blk->way_next)
        {
            if (blk->useCnt < min && blk->presentCnt
== 0)
            {
                min = blk->useCnt;
            }
        }

        for (blk = cp->sets[set].way_head; blk; blk
= blk->way_next)
        {
            if (blk->useCnt == min &&
blk->presentCnt == 0)
            {
                repl = blk;
                break;
            }
        }
    }
    break;
    …
}
```

The random policy's idea is pretty simple. If there is a cache miss, use the random number generator to specify a "victim" cache block, then replace this block. The function CACHE_BINDEX will specify the cache block based on its parameters, and variable repl will be used later in other cache block replacement functions.

However, since we must use inclusive property, we can only replace the block that only exist in this cache. Thus we use the variable persentCnt to determine whether this block could be replaced or not. If this block's presentCnt is not equal to 0, then we must choose another victim block to replace using random number generator.

As for LFU, the Least Frequently Used replacement policy, we added the variable useCnt to record each block was hit how many times. If there is a cache hit, then this block's useCnt would increase by 1. If there is a cache miss, then replace the block with the minimum number of useCnt among all the blocks in the same cache set, and reset the value of useCnt to 0. Once again, we must pay attention to inclusive property of cache. We can only replace the block with both minimum useCnt and presentCnt equals 0.

D. Multi-Clore Structure Design

To support multicore, we need to meet following requirements:

- The simulator should be able to run multiple test programs in parallel.

  In this project, our goal is to add the multicore support with minimum codes modification, so we make each core a process and run them in parallel, and we can achieve this by using system call fork (). Then the parent process and the child process will have their own copy of all resources such as registers and won't have to worry about interfering with each other. The following function is called to duplicate process. The syncCnt variable is used for synchronization.

```
void sim_dup_core(void)
{
    syncCnt = (size_t*)shmat(shmget(IPC_PRIVATE,
2*sizeof(size_t), 0666 | IPC_CREAT), NULL, 0);
    syncCnt[0] = 0;
    syncCnt[1] = 0;
    pid = fork();
}
```

- Each core should access its own cache data if the cache is private and access a common cache data if the cache is shared.

  Since each core is a process, they have independent memory space, so the private cache is created as usual by allocating dynamic memory. As for shared cache, we create share memory so both the parent process and the child process have access. We also need to create share mutex (which can be shared between parent process and child process) to avoid race condition for the shared cache. We modified the cache data structure to add necessary data members for shared mutex, as shown in the following codes:

```
struct cache_t
{
    …
    int shared; // cache shared flag, 1 for shared,
otherwise private
    pthread_mutex_t mutex;
       pthread_mutexattr_t mutexattr;
       …
}
```

And we modified cache creating function to create cache as private or shared as requested by the caller:

```
struct cache_t *
cache_create(…, int shared, …)
{
    …
    int shmid;
    if (shared != 1) // private cache
    {
        cp = (struct cache_t *)
            calloc(1, cacheSize);
        …
    }
    else // shared cache
    {
        shmid = shmget(IPC_PRIVATE, cacheSize, 0666
| IPC_CREAT);
        cp = (struct cache_t *)
            shmat(shmid, NULL, 0);
        …
        memset(cp, 0, cacheSize);
        cp->shmid = shmid;
        pthread_mutexattr_init(&cp->mutexattr);
        pthread_mutexattr_setpshared(&cp->mutexattr,
PTHREAD_PROCESS_SHARED);
        pthread_mutex_init(&cp->mutex,
&cp->mutexattr);
    cp->shared = shared;
    …
}
```

- All processes should be synchronized, i.e., they should execute instructions in the same pace.

For each core we create a synchronization counter which counts how many instructions have been executed, and we make child process to follow the pace of the parent process, i.e., the child process will only begin to execute an instruction after its parent started execution, and the parent will always wait for the child to finish current execution before it move to the next.

```
void sim_main(void)
{
    …
    if(pid != 0)// parent process, denote it as core 
1 process
    {
        while(syncCnt[0] > syncCnt[1]) ;//wait for 
core 2 process to follow up
        ++syncCnt[0];
    }
        else// child process, denote it as core 2 
porcess
    {
        while(syncCnt[0] <= syncCnt[1]) ;//wait for 
core 1 process to proceed
        ++syncCnt[1];
    }
    …
}
```

We sum up our multicore implementation with a list of critical steps:

1. Initialize cache, create private cache in dynamic memory and share cache in shared memory.
2. Initialize synchronization variables in shared memory.
3. Duplicate process.
4. Load testing program for simulation.
5. Synchronize instructions executions.
6. End synchronization when simulation is finished.
7. Exit.

Note that in our implementation, the parent process will load the first testing program, and the child will load the second testing program (if provided by user).

V. Methodology

A. Test Enviornment

We installed the SimpleScalar on a Ubuntu 14.04.5 LTS virtual machine.

Architecture:	i686
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	4
On-line CPU(s) list:	0-3
Thread(s) per core:	1
Core(s) per socket:	2
Socket(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
Model:	158
Stepping:	9
CPU MHz:	2808.002
BogoMIPS:	5616.00
Hypervisor vendor:	VMware
Virtualization type:	full
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	6144K
Memory:	4G
Disk:	10G

B. Benchmarks

We have two benchmarks writen in C language. One is based on matrix multiplication and another is based on RC4 encryption. The idea is to test two cache configurations with sequential access to memory and random access to memory. The matrix multiplication is sequentially access to memory. The randomly scramble the array has a process of using random number generator to access the array, and will scramble the array, so it can be seen as randomly access to memory.

In order to fully test our cache design, for each configuration and replacement policy, we will run each benchmark at least 3 times.

Benchmark 1: Matrix multiplication

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 300

int a[N][N];
int b[N][N];
int c[N][N];

int MM1(int a[][N], int b[][N], int c[][N])
{

    int i = 0, j = 0, k = 0;

    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                c[i][j] = c[i][j] + a[i][k] * 
b[k][j];

    return 0;
}

int init(int a[][N], int b[][N], int c[][N])
{
    int i, j;
    srand((unsigned)time(NULL));
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
        {
            a[i][j] = (rand() % 100);
            b[i][j] = (rand() % 100);
            c[i][j] = 0;
        }
    return 0;
}

void display(int a[][N])
{
    int i, j;
    printf("Matrix:\n");
    for (i = 0; i < N; i++)
    {
        for (j = 0; j < N; j++)
        {
            printf("%-5d ", a[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}

int main()
{
    init(a, b, c);

    MM1(a, b, c);
    display(a);
    display(b);
    display(c);

    getchar();
    return 0;
}
```

Benchmark 2: Randomly scramble the array

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>

int a[100000];

int main()
{
    int i, j, tmp;
    srand(unsigned(time(NULL)));
    for (i = 0, j = 0; i < 100000; i++, j++)
    {
        i = rand() % 100000;
        j = rand() % 100000;
        tmp = a[i];
        a[i] = a[j];
        a[j] = tmp;
    }
    return 0;
}
```

VI. Evaluation

The sim-cache simulator outputs many statistics of which we are only interested in cache access, cache miss, cache hit and cache writeback.

In SimpleScalar, both the cache miss and cache writeback rely on the cache access function to read the missed block from the lower level cache or write the evicted block to the lower level cache, and each call to the cache access function will increase the cache access counter by one, thus we have the following formula:

$$\mathrm{L}_{\mathrm{i+1}}\ \text{access} = \mathrm{L}_{\mathrm{i}}\ \text{miss} + \mathrm{L}_{\mathrm{i}}\ \text{writeback}$$

Where L_{i} is the level i cache.

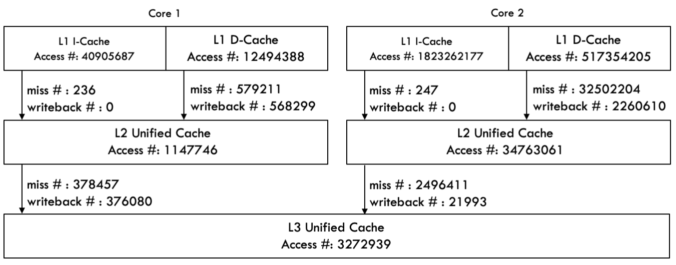

Figure 10. Configuration 2 with LFU policy test results

Figure 10 shows the cache access, cache miss and cache write back numbers of the simulation for the configuration 2 with LFU replacement policy (runs both benchmarks with double cores). We see that the miss number and the write back number of the L1 I-Cache and L2 D-cache add up to the access number of the L2 unified cache for both core 1 and core 2.

We use the configuration 1 to run simulation with the combination of the two benchmarks and the two replacement polices. The results are presented in Figure. For Benchmark 1, all the caches have pretty low miss rate, which give us intuitive impression on the importance of a good cache design. When we look into the results of Benchmark 2, we can see the instruction L1 cache's miss rate is extremely close to 0. The reason for this is that Benchmark 2 only has a few codes, thus has less instructions need to be executed. The instruction cache can hold almost all the instructions. However, the L2 cache's miss rate is much higher than the result of Benchmark 1. This is because Benchmark 2 has the idea of randomly access to memory, which means the distances between needed memory addressed could be relatively far, so sometimes the cache was not hold the needed data, causing lots of cache misses. In general, the results between LFU and random replacement policy are not much different, both of them works well.

The results of configuration 2 (multi-core) is show in Figure 12, and they are pretty similar to the results of configuration 1.

Configuration 1 Test Results				
	Benchmark 1 LFU	Benchmark 2 LFU	Benchmark 1 Random	Benchmark 2 Random
I cache L1 access	2197786020	15985041	2197777980	12949888
I cache L1 miss	9437310	230	18929186	257
I cache L1 miss rate	0.004	0.00001	0.0086	0.00001
I cache L1 write backs	0	0	0	0
D cache L1 access	616970914	4779100	616970080	3852169
D cache L1 miss	30545147	128694	32688616	102226
D cache L1 miss rate	0.0495	0.0269	0.053	0.0265
D cache L1 write backs	1147702	127547	2318995	100235
U cache L2 access	41130159	256471	53936797	202718
U cache L2 miss	1732253	47678	1732829	36013
U cache L2 miss rate	0.0421	0.186	0.0321	0.178
U cache L2 write backs	22493	43450	22709	31915
U cache L3 access	1755096	91336	1755888	68136
U cache L3 miss	17488	6676	18663	6684
U cache L3 miss rate	0.01	0.0731	0.0106	0.0981
U cache L3 write backs	0	0	1059	63

Figure 11. Configuration 1 test results

Configuration 2 Test Results				
	Benchmark 1 LFU	Benchmark 2 LFU	Benchmark 1 Random	Benchmark 2 Random
Private I cache L1 access	1823262177	40905687	1823262177	40905687
Private I cache L1 miss	247	236	247	236
Private I cache L1 miss rate	0.000001	0.000006	0.000001	0.000006
Private I cache L1 write backs	0	0	0	0
Private D cache L1 access	517354205	12494388	517354205	12494388
Private D cache L1 miss	32502204	579211	32502204	579211
Private D cache L1 miss rate	0.0628	0.0464	0.0628	0.0464
Private D cache L1 write backs	2260610	568299	2260610	568299
Private U cache L2 access	34763061	1147746	34763061	1147746
Private U cache L2 miss	2496411	378457	3041685	467258
Private U cache L2 miss rate	0.0718	0.3297	0.0875	0.4071
Private U cache L2 write backs	21993	376080	184377	464904
Shared U cache L3 access	3272939		4158222	
Shared U cache L3 miss	6039		6079	
Shared U cache L3 miss rate	0.0018		0.0015	
Shared U cache L3 write backs	0		112	

Figure 12. Configuration 2 test results

References

[1] Burger, Doug, and Todd M. Austin. "The SimpleScalar tool set, version 2.0." *ACM SIGARCH computer architecture news* 25.3 (1997): 13-25.

[2] Manjikian, Naraig. "Multiprocessor enhancements of the SimpleScalar tool set." *ACM SIGARCH Computer Architecture News* 29.1 (2001): 8-15.

[3] https://github.com/jsantangelo/sim-cache-mod