Implementing write-through data L1 cache in SimpleScalar

Tejeswini Jayaramareddy (tjayaramareddy@uh.edu)
Department of Electrical and Computer Engineering
University of Houston

1. Abstract

SimpleScalar is an open source software infrastructure developed by Todd Austin and Doug Burger used to model virtual computer system processor, cache and memory hierarchy. It was developed when Todd Austin was a PhD student at the University of Wisconsin Madison. Later on, it is maintained by SuimpleScalar LLC founded by Austin. Using the SimpleScalar, user can build programs and simulate them running on a range of modern processors and systems. As of today, the tool can simulate write-back cache behavior and doesn't support write-through cache policy. Through this project, my goal has been to implement write through cache support in open source simplescalar tool and evaluate performance parameters for the existing write-back and implemented write-through policies. For a generalized analysis, simulations are performed for different applications in SPEC 2000 benchmark.

2. Introduction

Modern processors need to have faster memory. Ideas behind faster memory are: First, if an item is referenced, there is high probability that the same will be referenced soon, called as temporal locality and the items whose addresses are close by tend to be referenced soon, known as spatial locality. Second, Smaller memories are faster as it takes lesser time to sort through. Finally, on chip memories are faster as they reduce communication time drastically. So memory is built in hierarchy in which each level has greater capacity than the previous one and holds a subset of data from the previous one. This temporary memory is embedded onto the processor called cache. CPU accesses cache faster than RAM or disk. Modern day cache is available in various forms up to 3 levels. Cache with more data will take more time to access but a very small cache will result in higher miss rate. It is only when processor fails to find data in any level cache, that it will move to main system memory. Unified cache is one having both data and instruction saved in the same cache. To, avoid structural hazard caused by simultaneous load and store instructions, data and instructions are separated into data cache and instruction cache. Modern processors also consider different write policies to cache- write-through, write-back and an array of other options. In write-through, data written to cache block will also be written to lower level memory. Whereas, in write-back, data is written to cache block and a dirty bit is set. Only when the cache block with dirty bit has to be evicted out of cache, lower level memory will be updated. Making choice of available cache designs needs simulation and analysis of caching techniques for a targeted application. There are issues of performance, reliability and complexity in making this choice. We also analyze system performance by selecting and varying micro-architecture parameters including Fetch queue width, Decode queue width, Issue width, RUU width, Load / Store queue size and a combination of different parameters.

The task of simulation is to analyze and ascertain the number of hits, misses, Missrate, write-back rate and various performance metrics including IPC (Instructions Per Cycle) for a program execution. Simulation is fastest and cost effective solution. SimpleScalar is wellknown cache simulator that leverage faster, more flexible S/W development cycle and permits more design space exploration. It facilitates validation before processor hardware is available. It also models, detailed micro architecture of the hardware and software. Using Simplescalar tools, users can build modeling applications that simulate real programs running on a range of modern processors and systems. But, as of today, simple scalar doesn't support write-through cache simulation.

This project is aimed to implement write-through support in SimplScalar.

The rest of this document contains information about downloading, installing and running Simpescalar, SimpleeScalar architecture, Write-back and write through cache differences, write-through cache implementation description, Verification and evaluation of memory and performance. In Section 3 we present a detailed procedure of downloading, installing and running SimpleScalar. In Section 4, we describe the SimpleScalar architecture. In Section 5, we discuss the main differences between write-back and write-through cache. In Section 6, we study the existing write-back cache implementation in SimpleScalar. In section 7, we discuss the implementation of write-through support. In Section 8, we verify the implanted design with a PISA compiled test program. In section 9, we simulate cache behavior upon executing SPEC2000 benchmark applications and provide detailed Analysis. In section 10, we provide a detailed discussion and evaluation of micro-architecture parameters and section 11 concludes with scope for future development.

3.Download, Compile and Run SimpleScalar

SimpleScalar tools can be downloaded from SimpleScalarLLC website (www.simplescalar.com) founded by Todd M Austin. Downloads section has symbolic links to three files: 1. simplesim-3v0e.tgz, simulator suit 2. simpletools-2v0.tgz, Debug and verification infrastructure, performance visualization tools and 3. simpleutils-2v0.tgz, GNU binary utilities. GNU binary utilities are required to run your own test programs that are not provided by the SimpleScalar LLC. Mentioned files are downloaded and extracted using (gun-zip) and then “tar –xvf file.tar” commands. SimpleScalar can be configured for PISA (Portable ISA) or ALPHA instruction set architectures. PISA is a simple MIPS like instruction set. Sim-fast, Sim-safe, Sim-profile, Sim-cache, Sim-bpred, Sim-outorder are the simulators available in SimpleScalar toolset. Sim-outoreder is a detailed micro-architectural simulator that models out-of –order microprocessor including branch prediction, caches and external memory. Sim-outorder is the simulator opted for this project and is compiled using make command “make sim-outorder”. Sim-outorder then is runusing the command below.

./sim-outorder -config default.cfg –fastfwd 100000000 -max:inst 100000000 is the command used to execute the benchmarks.

4. Simple Scalar: An Architectural Simulator

A tool that produces the behavior of a computing device is called an architectural simulator. SimpleScalar is an architectural simulator taking system inputs and producing system outputs as well as System metrics as shown below in figure1.

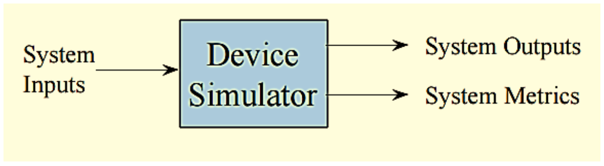


Figure 1. An architectural Device Simulartor.

Out-order simulator stages are shown in figure 2.

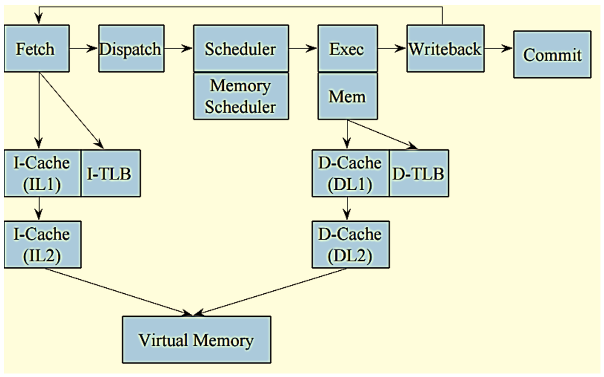


Figure2. Simulation stages of SimpleScalar for an out-order execution

Fetch-Stage fetches instructions from Instruction cache one cache line at a time and places in Fetch queue abbreviated as IFQ further in this document. It probes branch predictor to find next cache line to be fetched from I-cache (Instruction cache).

Dispatch-Stage takes instructions from fetch queue, decodes, executes and updates Register Update Unit (RUU) and Load-Store Queue (LSQ). An early detection of branch mis-prediction is taken at decoder stage. It also updates Register Rename Table and Machine state.

Scheduler locates instructions whose input registers are ready and **Memory Scheduler** locates loads with memory inputs ready.

Exec-Mem stage models functional unit and D-cache access. It takes instructions from scheduler and updates functional unit and D-cache state.

Write-back Stage detects mis-predictions, wakes up ready instructions and models write-back bandwidth.

Commit: Retires instructions to Data cache.

The loop shown in figure 3 is iterated for each machine cycle and performs each stage activities shown in figure 2.

```
ruu_init()
for(;;) {
    ruu_commit();
    ruu_writeback();
    lsq_refresh();
    ruu_issue();
    ruu_dispatch();
    ruu_fetch();
}
```

Figure3: Loop iterated for each machine cycle

5. write-back vs Write through cache

Write-through cache mechanism is shown in figure 4. When processor finds read hit, processor will be updated with data value. In case of a read miss, a block to be replaced is found based on replacement mechanism, and data is read from lower memory into cache block replaced as shown in left branch of figure 4. Right branch models the behavior of a write request. In case of write hit both cache and lower memory are updated. While, a write miss writes data to lower memory.

Figure 5 shows Write-back cache mechanism. Left branch models read request behavior. In case of read hit, data is returned to the processor. Read miss checks the dirty bit of cache block. If the dirty bit is found set, existing data in that block is written to lower memory and reads required data

from memory into that block and resets dirty bit. If dirty bit is found to unset, data copy is in match with main memory. So, it is over written in cache with new required data fetched from memory. Similarly, right branch models write behavior of write-back cache.

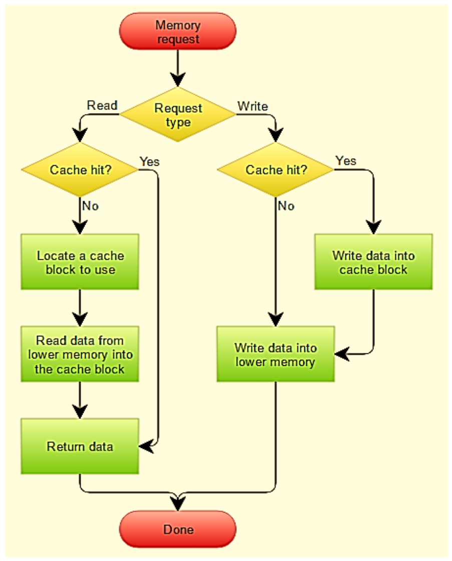

Memory request
Request type
Read
Write
Cache hit?
Yes
No
Cache hit?
Yes
No
Locate a cache block to use
Write data into cache block
Read data from lower memory into the cache block
Write data into lower memory
Return data
Done

Figure 4: Write-through cache Read and Write Mechanism

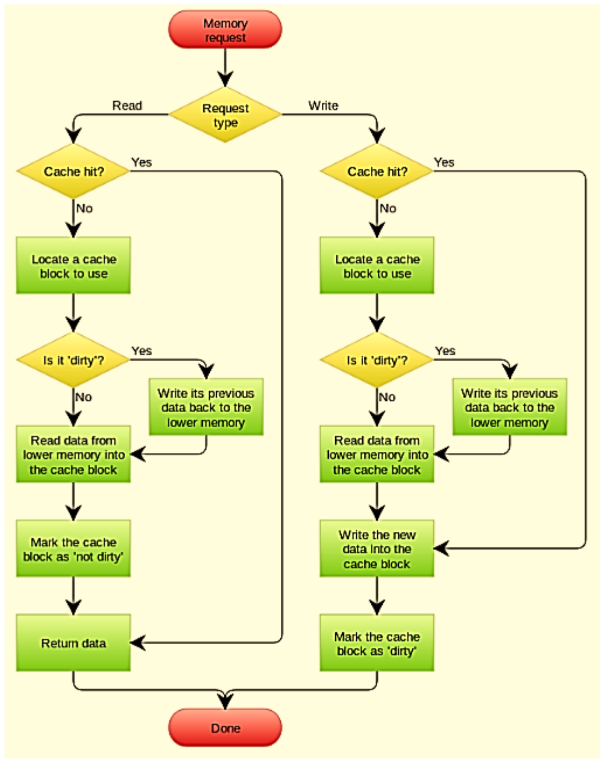

Memory request
Request type
Read
Write
Cache hit?
Yes
No
Cache hit?
Yes
No
Locate a cache block to use
Locate a cache block to use
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Is it 'dirty'?
Yes
No
Write its previous data back to the lower memory
Read data from lower memory into the cache block
Read data from lower memory into the cache block
Mark the cache block as 'not dirty'
Write the new data into the cache block
Return data
Mark the cache block as 'dirty'
Done

Figure 5: Write-back cache Read and Write Mechanism

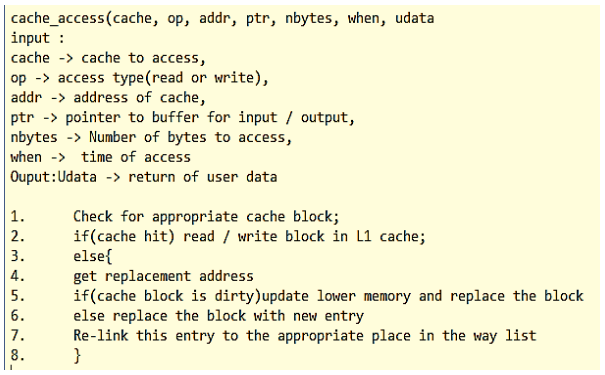
Write-through cache is beneficial for applications that write data and then read frequently. Whereas, write-back cache is suitable for mixed work-load.

Write through cache is simpler and consistent with main memory. Write-back cache is harder to implement and inconsistent with main memory.

Writes to write through cache are longer and write traffic is huge. Writes to a write-back cache occur at cache speed and multiple writes to same block will be written to main memory at once.

6. Write-back design details in simplescalar

Data cache is accessed at two stages in sim-out order simulator design. For Data read in ruu_issue() and writing to data cache in ruu_commit(). Data L1 cache is accessed with an interface to cache module: cache_access ().

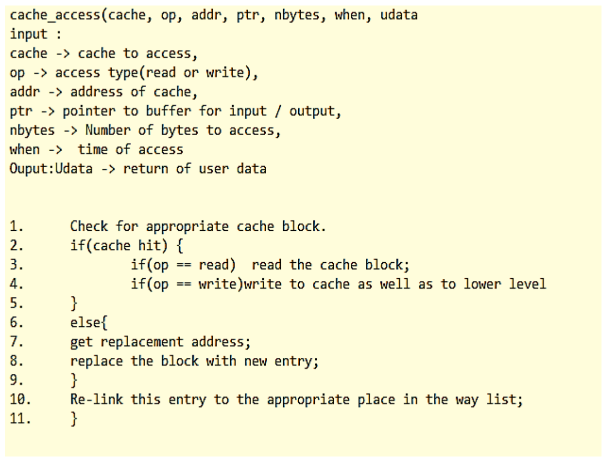


```
cache_access(cache, op, addr, ptr, nbytes, when, udata
input :
cache -> cache to access,
op -> access type(read or write),
addr -> address of cache,
ptr -> pointer to buffer for input / output,
nbytes -> Number of bytes to access,
when ->  time of access
Ouput:Udata -> return of user data

        Check for appropriate cache block;
        if(cache hit) read / write block in L1 cache;
        else{
        get replacement address
        if(cache block is dirty)update lower memory and replace the block
        else replace the block with new entry
        Re-link this entry to the appropriate place in the way list
        }
```

Figure 6: Algorithm for write-back cache policy in SimpleScalar

7. Write-through feature implementation in cache

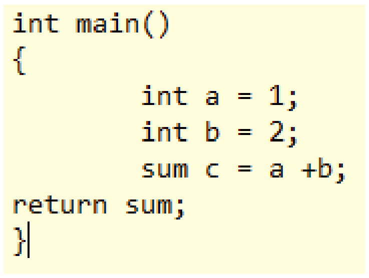


```
cache_access(cache, op, addr, ptr, nbytes, when, udata
input :
cache -> cache to access,
op -> access type(read or write),
addr -> address of cache,
ptr -> pointer to buffer for input / output,
nbytes -> Number of bytes to access,
when ->  time of access
Ouput:Udata -> return of user data


        Check for appropriate cache block.
        if(cache hit) {
                if(op == read)  read the cache block;
                if(op == write)write to cache as well as to lower level
        }
        else{
        get replacement address;
        replace the block with new entry;
        }
        Re-link this entry to the appropriate place in the way list;
        }
```

Figure 7: Algorithm for write-through cache policy in SimpleScalar

8.Verification of write-through implementation with a sample program:



```
int main()
{
        int a = 1;
        int b = 2;
        sum c = a +b;
return sum;
}
```

Figure 8: test program run on PISA configured Sim-outorder

Above test program is run on pisa simulator and observed huge write-traffic to lower memory for write-through cache compared to write-back cache as shown in figure 9 and is verified that write to lower memory takes place for every write request from processor using DLit debugger provided with SompleScalar tools and GDB.

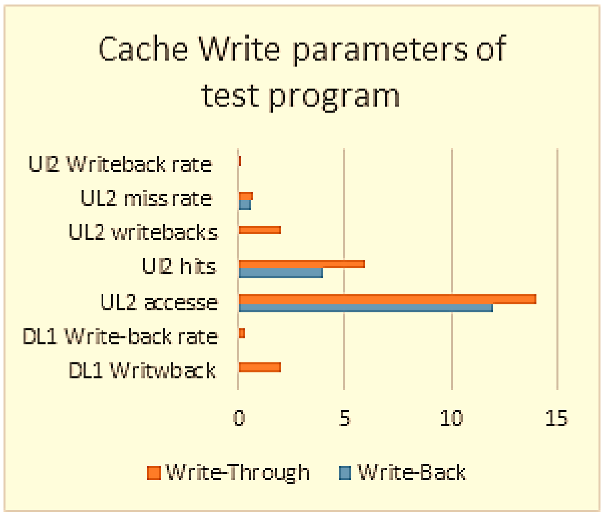
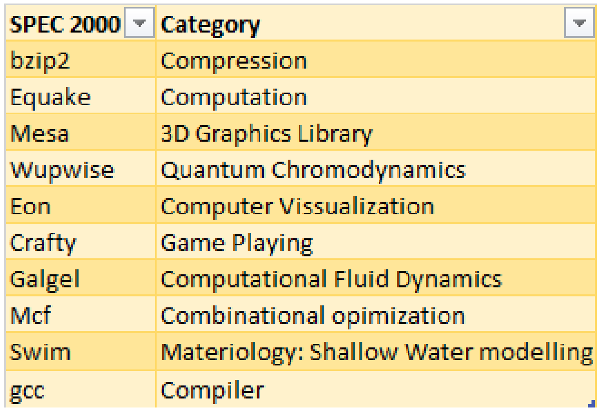


Figure 9. Write-back and Write through parameters compared

9. Analysis of write-back vs write through design.

Impact of cache parameters like L1 data cache size, L1 cache block size, L1 cache associativity, L2 Cache size, L2 block size for write-back and write-through cache designs are evaluated in terms of performance parameters, write traffic, missrate, Accesses to lower level memory and IPC. Micro-architecture parameters Instruction Fetch Queue width, Decoder Bandwidth, RUU issue bandwidth, Capacity of load Store queue size and their joint impact on IPC is analyzed. Memory access pattern in each program is different. So, for generic analysis, Simulations are done on SPEC 2000 benchmarks tabulated in figure 10.



SPEC 2000	Category
bzip2	Compression
Equake	Computation
Mesa	3D Graphics Library
Wupwise	Quantum Chromodynamics
Eon	Computer Vissualization
Crafty	Game Playing
Galgel	Computational Fluid Dynamics
Mcf	Combinational opimization
Swim	Materiology: Shallow Water modelling
gcc	Compiler

Figure 10. SPEC 2000 Benchmarks

Write-traffic from L1:

One of the distinguishing factor between write-through and write-back design is write-traffic. Data is written back from Level 1 for every processor write in write-through cache. Whereas, in write-back, data is written back to lower level memory only when the block needs eviction. This is observed in figure 11 showing higher write traffic for write-through cache. As the size of block is increased, miss rate is reduced, number of evictions are fewer and number of write-backs are reduced. This is observed by dropping curve for write-back with increasing L1 D-cache block size.

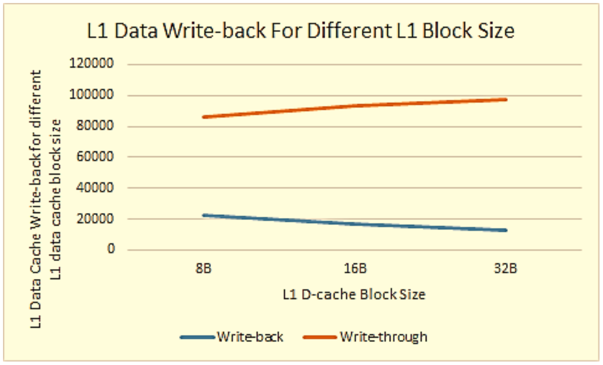


Figure.11 L1 Write-backs compared for write-through and write-back with different cache size

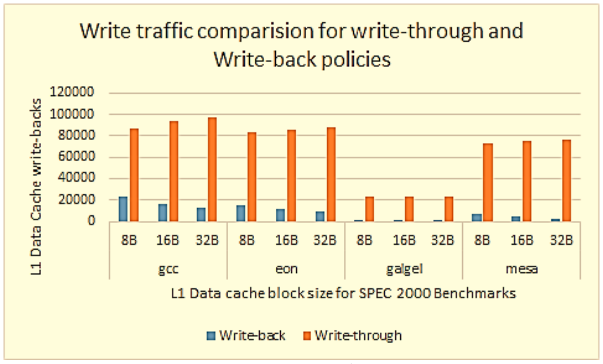
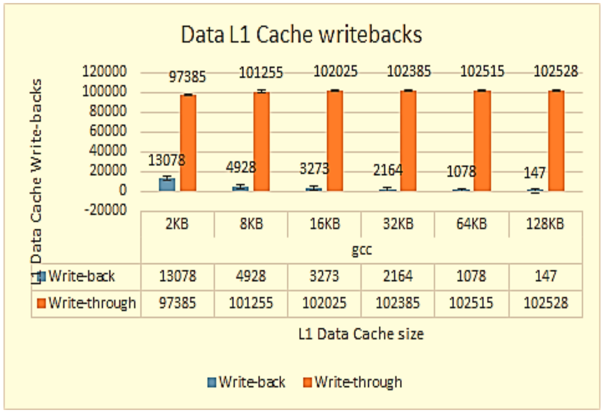


Figure 12. L1 Data cache write-backs for write-through and write-back with different cache size on SPEC 2000 benchmarks

For a generic evaluation, the impact of write-through and write-back policies on L1 data cache write-backs is considered for six of the SPEC 2000 bench marks in figure 12 with varying cache block size. A huge increase in write-back is observed for write-through desin. Larger blocks take the advantage of special locality. But, larger blocks increased conflict misses for, they hold fewer blocks and also increase miss penalty.

Similar trend is found with increase in L1 Data cache size and Associativity as shown in figure 13 and figure 14.



	2KB	8KB	16KB	32KB	64KB	128KB
	gcc					
Write-back	13078	4928	3273	2164	1078	147
Write-through	97385	101255	102025	102385	102515	102528

Figure.13 L1 Write-backs compared for write-through and write-back with different cache size

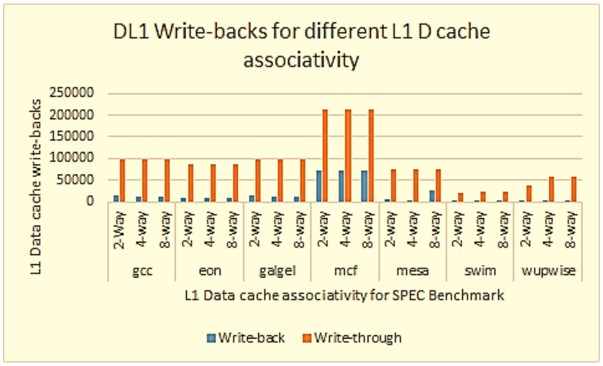


Figure14. L1 Write-backs compared for write-through and write-back with different L1 cache associativity.

Data L1 Missrate:

L1 data cache missrate adversely affects CPU performance. From simulation, it is observed in figure 15 that L1 D-cache missrate is not affected by write-through and write-back cache designs. We can take advantages of write-through design without affecting L1 missrate. Further, with increasing cache size, missrate is reduced as bigger cache takes the advantage of special locality.

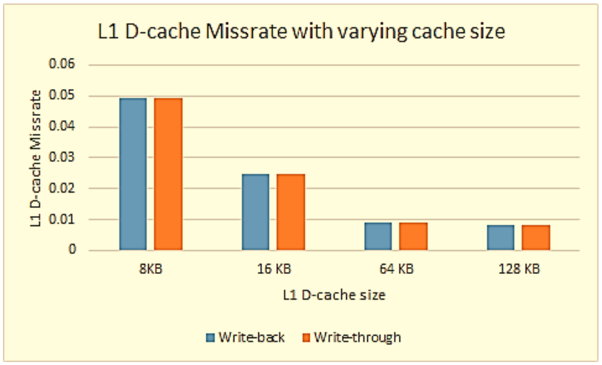


Figure 15. Impact of L1 D cache size on Missrate for GCC bench mark.

For gcc bench mark, 83.13% missrate is reduced increasing L1 cache size from 8KB to 128KB. Overall average reduction in missrate for four of the SPEC 2000 benchmarks is approximately 21.25% and observation is shown in in the figure 16.

Similar trend is observed for Data L1 missrate by varying cache block size. Missrate reduces with increase in cache block size due to special locality as shown in figure 17. But, increasing block size unreasonably, will result in more potential conflict misses as fewer blocks would be available.

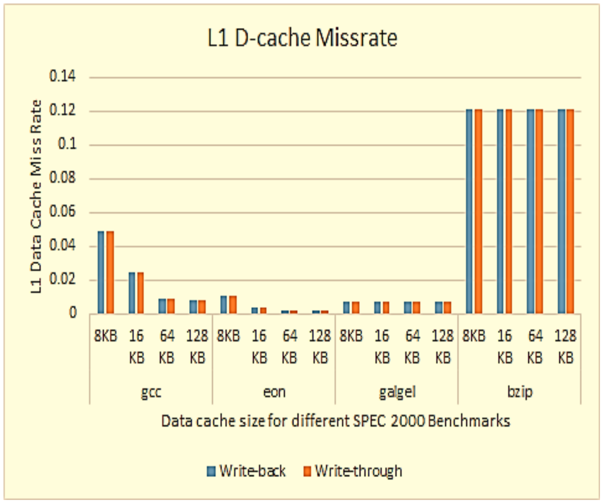
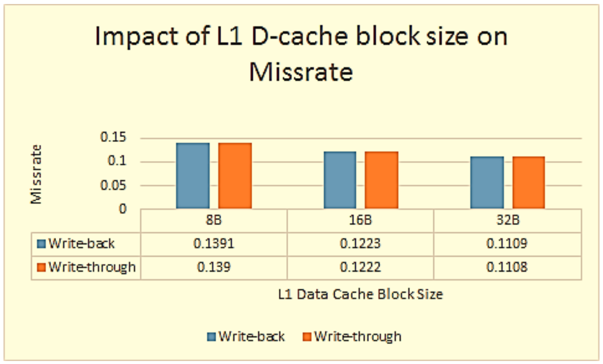


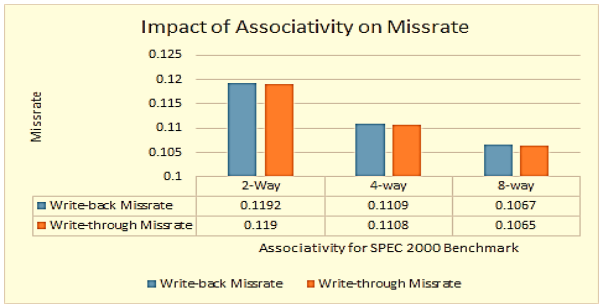
Figure 16. Impact of L1 D cache size on Misstate for SPEC 2000 bench marks.



	8B	16B	32B
Write-back	0.1391	0.1223	0.1109
Write-through	0.139	0.1222	0.1108

Figure 17. Impact of L1 D cache block size on Misstate for SPEC 2000 bench marks.

With Higher associativity for L1 cache, each set has more blocks. Therefore, there is less chance of conflict between two addresses. Thus a higher associative cache has lower missrate and is observed as shown in the figure 18. This will reduce the memory stalls as well as Average Memory Access Time.



	2-Way	4-way	8-way
Write-back Missrate	0.1192	0.1109	0.1067
Write-through Missrate	0.119	0.1108	0.1065

Figure 18. Impact of L1 Data cache associativity on Missrate

Average Memory Access Time(AMAT):

Read Average memory access time for both write-back cache is given by,

Read AMAT = Read Hit Time + Missrate * Read Lower Level Time (1)

Read Hit time, L1 missrate and Read from Lower level are unaffected for write through and write-back designs. So, Average memory access time to read is same for both the designs.

Write Average Memory Access Time for write back design with no write-allocation is given by,

Write AMAT = Write hit time + Missrate * Write to Lower Level Time. (2)

Whereas, Write AMAT for Write-through cache design with no write-allocation is given by,

Write AMAT = Write hit time + Write to Lower Level Time. (3)

From equations 2 and 3, Write AMAT is lower for write-back cache design compared to write through cache design unless every reference to cache is a miss.

Unified L2 Accesses:

In write-back cache, L2 cache is accessed only when there is a read / write miss in L1 cache. In write-through cache, L2 is accessed when there is a read miss, write miss and write hit. So, More L2 accesses are observed for write-through compared to write back shown in figure 19.

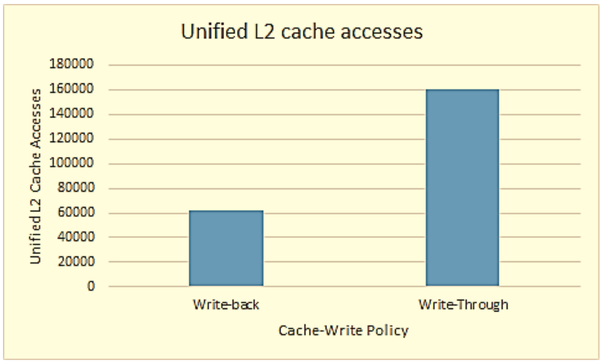


Figure 19. Unified L2 Caches compared for write-through and Write-back cache.

Unified l2 Missrate:

Memory access for L2 cache is simulated and observed the behavior shown in figure 20. This figure shows unified L2 cache access, hits, misses and miss rate with increased L2 cache size. Increasing cache size decreased misses in both write-through and write-back designs. Missrate variating with L2 cache size is shown in figure 21.

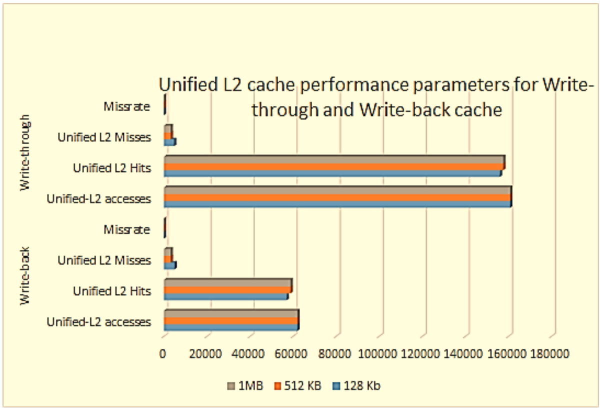


Figure 20. Unified L2 cache access parameters for write-through and write-back cache

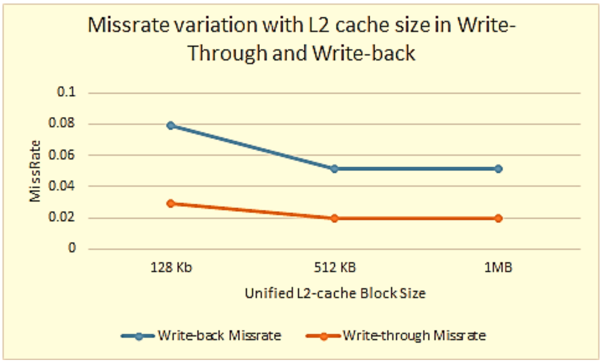


Figure 21. Missrate variation with L2 cache size for write-through and write-back cache.

Write-through cache is consistent with main memory so, the number of misses at level 2 cache are lower in write-through than in write-back.

Unified L2 Write-backs:

Write traffic from L2 cache to main memory shows huge difference for Write-through and Write-back caches designs. As L2 Cache size increase, the potential L2 misses are reduced. Therefore, replacements and write to main memory are considerably lower for write-back cache as shown in the figure 22. As every write request from the processor is written back to main memory in write-through, the traffic is found very high.

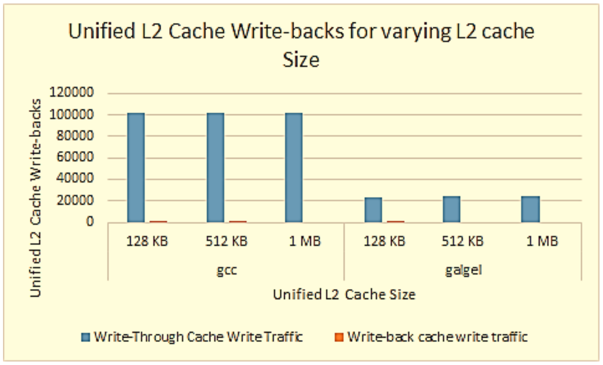


Figure 22. Unified L2 cache write-backs for varying L2 cache size compared between write-through and write-backs designs

10. Micro-Architecture Parameters Modelong in SimpleScalar

SimpleScalar also models Micro-architecture of a real processor. Micro-architecture parameters considered in this report are Instruction Fetch Queue Size, Decoder Bandwidth, Issue bandwidth, RUU occupancy, Capacity of load Store queue size and combination of these parameters.

Instruction Fetch Queue width, Decode Queue width and Issue Queue Width

Instruction Fetch Queue width, decode with and issue width specifies the number instructions the processor can fetch decode and issue in one machine cycle. The performance metric considered to evaluate the impact is CPI. Figure 23 shows the IPC improvement with increasing mentioned queue width from 2:2:2 to 8:8:8. At 8:8:8, the processor can fetch, decode and issue 8 instructions in one cycle.

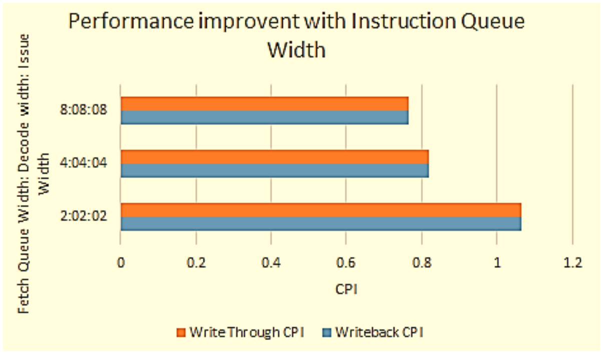


Figure 23. Impact of instruction queue width, decode bandwidth and Issue width on CPI

A reduction of 28.27% CPI is observed for fetch queue, decode and issue queue width increase from 2:2:2 to 8:8:8. Considering all SPEC 2000 bench marks, an average improvement is around 18%. But in reality, increasing queue size doesn't necessarily speed up the processor always. Because, as the queue size is more, bus interface takes time to fill the queue and if there is a branch instruction, the entire queue has to be flushed and need to re-fill. Also, increasing queue size required additional hardware cost.

Impact of RUU size on IPC:

Dispatch stage sends instructions to RUU. RUU is a hardware data structure that keeps track of the data dependencies and also commit the completed instructions. An entry in RUU is as shown in figure 24.

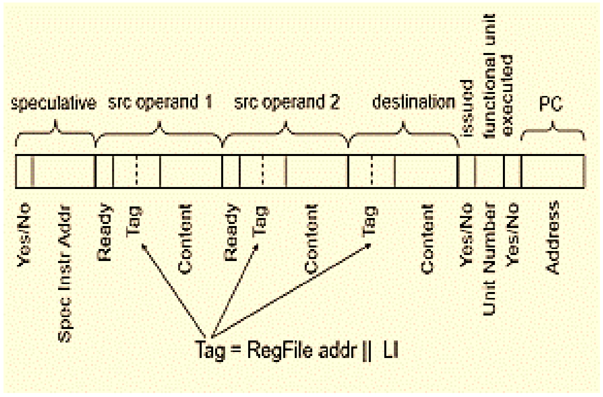


Figure 24. Model of Typical RUU entry

Here, we evaluated the impact of RUU size on IPC and observation is shown in Figure 25. It is clearly seen that IPC improvement is observed in both write-through and write-back cache design with increasing RUU width. Percentage of IPC improvement for GCC bench mark for every power two increase in RUU size is 32.8%. And, effectively saturates after 16. The aim of RUU is to have just enough instructions to be given to functional units. For more functional units available, ideally RUU should be higher. But, the number of instructions that can be fetched is greater than the number of instructions that can be issued. So, RUU must hold the instructions to be given to functional units and doesn't need to have unreasonably higher size.

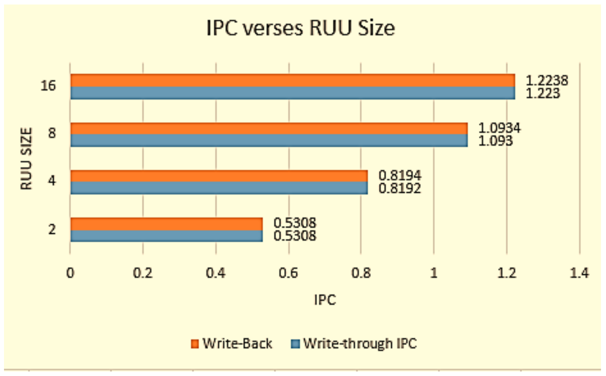


Figure 25. Impact of RUU occupancy on IPC

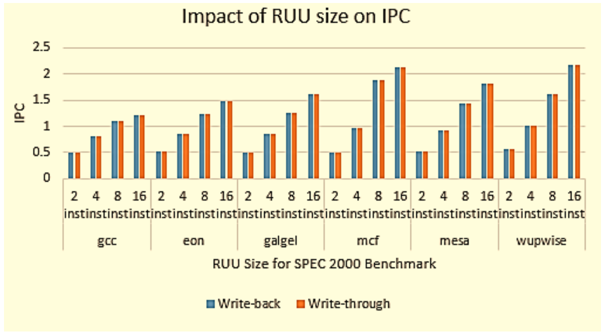


Figure 26. Impact of RUU occupancy on IPC for SPEC 2000 Bench marks

Impact of Normalized RUU size on IPC considered for SPEC 2000 benchmarks is found to be around 38% for both write-back and write-through designs.

Joint Impact of RUU and Issue width.

Increasing RUU size to 16 along with issue width to 8, together gives better results and observed an average of 55.7% improvement in IPC. And is shown in figure 27.

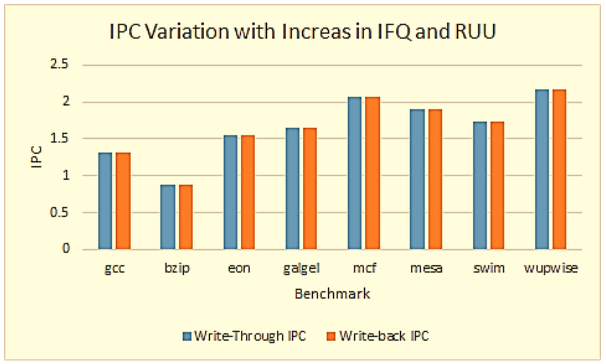


Figure 27. Impact of increase in IFQ of 8 instructions and RUU occupancy on IPC for SPEC 2000 Bench marks

Impact of IPC with load/store queue size:

Dispatch stage sends instructions to LSQ. LSQ is a hardware data structure that keeps track of the memory dependencies. For Load / Store Instructions two operations are required: effective address calculation and memory operation. Load Store Queue (LSQ) keeps track of memory operations that are issued and not committed. Effective address calculation operation is given an entry in RUU and memory operation in LSQ. Impact of LSQ size on IPC is similar for both cache designs and it is observed that IPC saturates for increase in LSQ at 16. Because, through LSQ size is increased, committing an instruction has to wait for RUU result to be available.

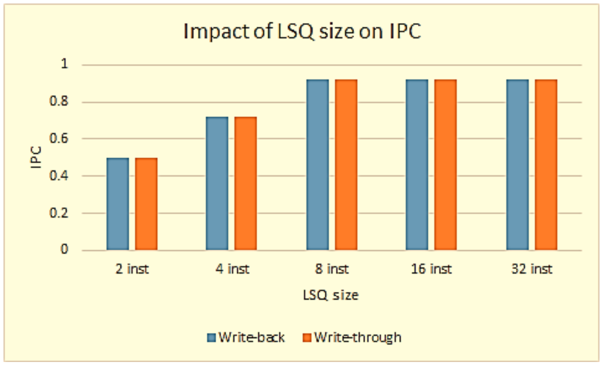


Figure 28. Impact of increase in LSQ to 8 instructions on IPC for bzip Bench mark

With increase in LSQ size, there is an improvement of 84% in IPC for bzip banch mark shown in figure 28. On an average, there is 60.48% improvement in IPC. As shown in figure 29.

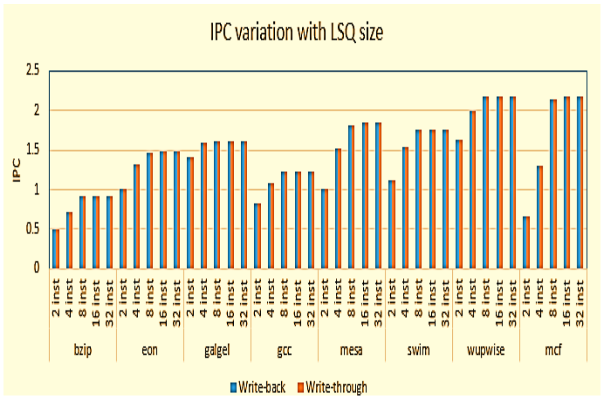


Figure 29. Impact of increase in LSQ to 8 instructions on IPC for SPEC 2000 Bench marks

Joint Impact of RUU and LSQ on IPC

From the section LSQ impact on IPC, it is intuitive that joint impact of higher LSQ and RUU would give better performance. And the same is observed in figure 30. However, the impact is same for both write-through and write-back cache designs.

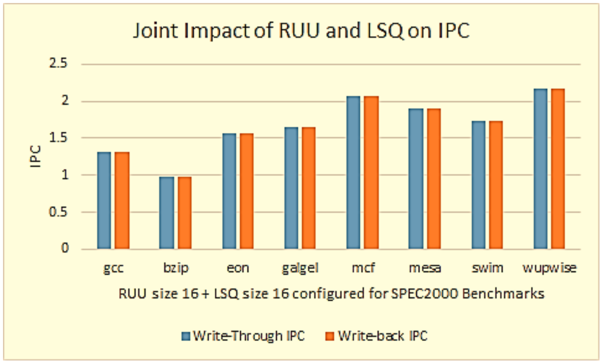


Figure 30. Joint impact of increase in LSQ to 32 instructions and RUU occupancy to 32 on IPC for SPEC 2000 Bench marks

Impact of In-order and Out-Order Issue

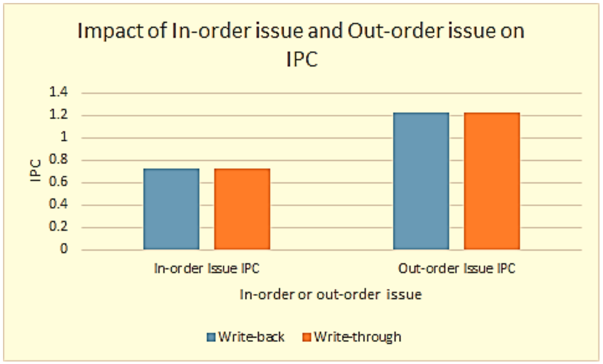


Figure 31. Impact of In-order and Out-order issue on IPC

In in-order issue, the instructions are issued in the order they occur and they may have to stall if necessary. In out-order issue, decode pipeline is decoupled from execute pipeline. So, the instructions can be fetched and decoded until the decode pipeline is full. Later, when functional unit is available, instructions are executed. So, Performance metric in terms of IPC is better for out-order issue and this observation is common across write-through and write-back cache designs.

In Summary, Write-through cache is slower, cleaner and consistent with main memory. So, good for fault tolerance and recovery. Implementation is simpler. Varying cache size, block size and associativity have similar effect as write-back. Optimally selecting micro-architecture parameters improve the performance in terms of IPC independent of cache write policy.

11. Conclusion

For high performance design, it is important to understand the effect of the different parameters and interactions between parameters. This project enabled SimpleScalar tool to model write-through cache behavior. Implemented write-through design doesn't show any performance improvement but it has the advantages that it is simpler, consistent with main memory, suitable for applications that write once and read repeatedly. Implementing write-buffer can be a target to extend this project. Supporting Non-blocking cache through write-through by implementing MSHR that keeps track of outstanding misses and pending loads and stores that refer to same block can be another target for extension of this project. Aiming at high performance design, implementing victim cache, pipelined cache access, multi-banked cache, pre-fetching support, pipelining cache writes are the advanced optimization techniques that can be potential targets for extension of this project.

References

[1] D.Burger and T.M. Austin . "The SimpleScalar Tool set. Technical Report TR-97-1342", University of Wisconsin, 1997.

[2] http://www.simplescalar.com/docs/hack_guide_v2.pdf.

[3] www.simplescalar.com.