ZSim with Migratory Sharing Optimization Final Report

Zhuoyu Ji
zhuoyuj@andrew.cmu.edu

Haoyang Wang
haoyang4@andrew.cmu.edu

Friday 16th December, 2022

1 Summary

We modified the ZSim multicore simulator source code to add a migratory sharing optimization feature. This feature can detect migratory objects and improve performance when accessing them. We designed six micro-benchmark programs. We run them with both the default and our optimized ZSim on the same Ubuntu 16.04 virtual machine. The experimental results show that our optimization is effective.

2 Background

2.1 MESI protocol

MESI is an invalidation-based cache coherence protocol. We assume the reader has a thorough understanding of the MESI protocol.

2.2 ZSim

ZSim is an open-source x86-64 multicore simulator. The cache system of ZSim uses a fully directory-based architecture. It uses a tree structure where the root node is the Last Level Cache (LLC) and the leaf nodes are the L1 data (L1d) and instruction (L1i) caches of each processor. Each node in the tree is itself a cache object. Each cache object has a coherence controller to maintain cache coherence.

For each non-leaf node, a coherence controller consists of a top coherence controller (`TopCC`) and a bottom coherence controller (`BottomCC`). `TopCC` keeps all the directory information for each cache line in this node such as which child nodes are sharing a cache line. `BottomCC` keeps the coherence states (MESI) of each cache line in this node. Besides, `BottomCC` is in charge of sending the processor requests recursively to the parent if the current node has no privilege to handle the request. For example, if there is a read-exclusive request for a cache line, but the current L1d cache line state is S, it needs to send this request recursively to parents until reaching a node in M or E state. (In case the LLC doesn't have the cache line with an appropriate state, the line is fetched from the main memory.) When `BottomCC` finds a node to handle the request, its task is done. Then, `TopCC` is in charge of updating the directory information, changing the coherence state of the requesting child node, and sending invalidations to the impacted child nodes if necessary. Following the recursive path created by `BottomCC` in the reverse order, `TopCC` of all nodes in this path will work so that directory and coherence state updates can be passed to the leaf node that gets the original request from its processor. Leaf nodes have only `BottomCC` as they have no child nodes.

Next, we will introduce the ZSim terminologies used in this report. First, for requests from processors, there are `GETS` for read and `GETX` for read-exclusive. Second, for invalidations, true invalidation (`INV`) means an invalidation that changes the state to I; downgrade invalidation (`INVX`) means an invalidation that changes an exclusive state (M or E) to S. Third, an upgrade miss means a read-exclusive request on a cache line in S state. As mentioned earlier, a cache line in S state has no privilege to handle a read-exclusive request and extra communication is required to properly handle the request.

2.3 Migratory Sharing and Migratory objects

In Per Stenström, Mats Brorsson, and Lars Sandberg's paper [1], they mention a migratory sharing access pattern where multiple processors read and modify shared resources one at a time. "One at a time" means only one processor is using the resources at a time. In implementations, the most straightforward example is multiple processors using a shared data structure inside a critical section. In this case, the shared data structure is called a migratory object.

Under MESI protocol, accessing a migratory object results in an access pattern like Figure 1. In Figure 1, there are two downgrade invalidations, two true invalidations, and two upgrade misses. If migratory objects can be detected, the access pattern in Figure 1 can be optimized to Figure 2. In Figure 2, there are only two true invalidations; two downgrade invalidations and two upgrade misses are eliminated. In ZSim, downgrade invalidations and upgrade misses all need communication between child and parent nodes, and the communication latencies will be calculated. In real invalidation-based caches, downgrade invalidations and upgrade misses surely have performance penalties as the cache traffic introduced consumes cache communication bandwidth and increases latencies of other cache communications as well. Therefore, if migratory objects can be detected and treated specially as in Figure 2, there should be performance improvement when accessing them.

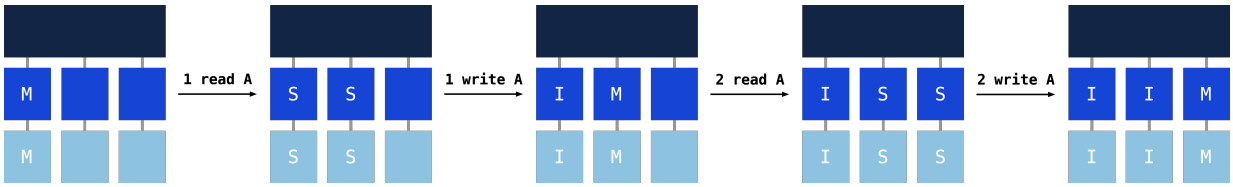

Figure 1: Standard Pattern when Accessing Migratory Object A (3-core: 0, 1, 2)

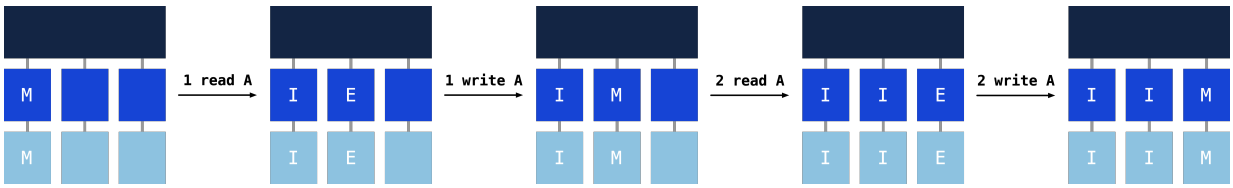

Figure 2: Optimized Pattern when Accessing Migratory Object A (3-core: 0, 1, 2)

3 Approach

In this project, we want to implement in ZSim a migratory object detection algorithm similar to the one proposed by Per Stenström, Mats Brorsson, and Lars Sandberg. The algorithm in their paper is a modification of Stanford DASH. Below is our altered version implemented in ZSim.

To begin with, the cache coherence controller needs to know whether a cache line is considered migratory. Detection of migratory objects is triggered whenever there is a read-exclusive request on a cache line that is not marked migratory at that moment. Then, if this cache line has exactly two sharers at that moment and the previous read-exclusive requester is not the current requester, this cache line is marked as a migratory object. For a cache line marked migratory, any read request is treated as a read-exclusive request. The migratory mark will only be gone when the number of sharers for the cache line exceeds 2 at any moment or the cache line is evicted.

In other words, consider a non-migratory object which has most recently been modified by processor 0. Processor 1 first reads it (causing the number of sharers to be 2) before attempting to modify it. When the read-exclusive request for modification is handled, this object satisfies the conditions to be marked migratory. Figure 3 shows how a migratory object is detected and how the access pattern changes in our implementation.

Now, we explain our implementation in ZSim. Since this algorithm requires extra states and knowledge of sharers for each cache line, it perfectly fits the directory-based cache system of ZSim. Remember that `TopCC` in ZSim keeps directory information for each cache line including the number of sharers. To be more specific, in ZSim, `TopCC` has an array of `Entry`s where each `Entry` belongs to a cache line and stores its directory information. An `Entry` stores the number of sharers, a bit vector to indicate who are the sharers, and a flag to indicate whether this cache line is in an exclusive state. Given that, `Entry` is the ideal place to add extra states for each cache line. We add a boolean `isMigratory` to store whether this cache line is migratory and a `uint32_t lastGETX` to store the index of the last read-exclusive requester. Their default values are `false` and `0xFFFFFFFF`. Those are added to file `coherence_ctrls.h`.

The next step is implementation of the detection logic. We add those in the `processAccess()` function of `TopCC` in file `coherence_ctrls.cpp`. Remember when there is a request originating from one leaf node, it is recursively processed by `BottomCC` and after `BottomCC` finishes its tasks, `TopCC` comes to handle the request. Specifically, `processAccess()` is called to handle those requests.

Overall, our code is clean and self-explanatory. Here, the only point to mention is that for a migratory object, the special treatment is: if there is a `GETS` request on a migratory object, it is treated as `GETX`. We use this straightforward

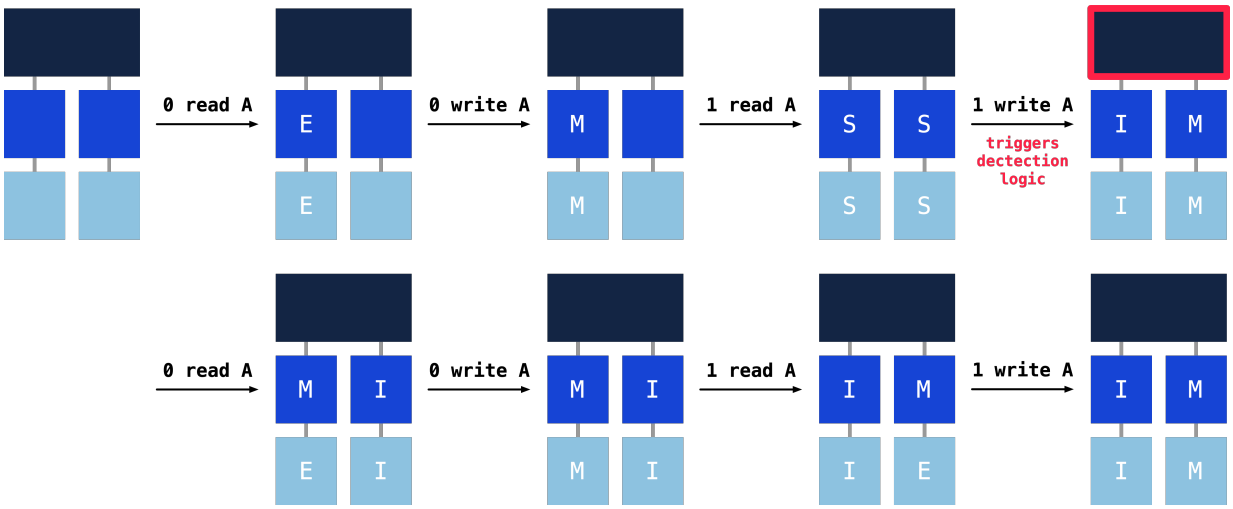

Figure 3: How Migratory Object A is Detected and Access Optimized (2-core: 0, 1)

special treatment to minimize changes in ZSim's source code. Besides, this treatment is also inspired by the idea of exclusive-mode prefetching.

Treating any `GETS` as `GETX` means M state will be granted directly. However, this special treatment only applies to the first `TopCC` handling the request. Take the cache in Figure 3 as an example. An important observation is that L3 is the only cache with multiple child nodes. So, for L1 and L2, the migratory detection condition of two sharers can never be satisfied, which means L1 and L2 never consider any cache line as migratory even if the same cache line is marked migratory by L3. Thus, the special treatment only applies to L3. As a result, after a migratory object is detected, when a processor starts its turn by sending a read request, the `GETS` is eventually sent to L3 and treated as `GETX`. Then, L3's `TopCC` changes state of the requesting L2 to M. Next, requesting L2's `TopCC` works but still treats the request as `GETS`. At this moment, since L3's `TopCC` has already changed the directory information, requesting L2's `TopCC` handling `GETS` could grant requesting L1 the E state. That's why in Figure 3, after a migratory object is detected, after a read request, we see L2 state as M but L1 state as E. This looks confusing. But it does not affect correctness since whether a cache line is dirty depends on its state in the leaf node (L1d).

To end this section, our algorithm seems to be effective but what if some processor decides to no longer modify a migratory object after the object is marked migratory? Such edge cases are considered and tested. We will discuss them later in detail.

4 Results

4.1 Introduction

We are aware of many existing standard benchmarks such as SPEC. However, in this project, we decide to write our own micro-benchmark programs for two reasons. First, developers of ZSim suggest formal experiments be done in a native x86-64 Linux environment to best utilize the underlying hardware. However, our two experiment environment candidates are VM on one member's x86-64 PC, which is not very powerful, and VM on AWS, which is very basic given how much we can afford. Therefore, we don't believe those environments can efficiently run the standard benchmarks with hundreds of billions of instructions. As a note, CMU and PSC machines are out of consideration since ZSim requires a rather old Linux version with root access to modify system configurations. The second reason is that our optimization only targets the migratory sharing access pattern and it should have no effects on non-migratory objects. For those standard benchmarks, we don't know how many of the instructions are following the migratory sharing access pattern; even if we know, it's very likely to be a very small portion, which means the overall improvement could be close to zero as we can only optimize the very small portion. So, micro-benchmarking is ideal to make improvement visible.

In total, we write six micro-benchmark programs. The first two have about 40 million instructions per core simulated in ZSim. After getting the expected results from those two initial tests, we increase the workload of the other four tests to about 100 million instructions per core simulated in ZSim.

Before presenting the test results, we should first mention the common setup for all six tests. All six tests use the same ZSim simulation configurations as shown in Figure 4. To be more specific, we simulate 2 OOO cores, a 3-level cache where L1 data cache of each core is 64KB, L1 instruction cache of each core is 32KB, L2 cache of each core

is 256KB, and shared L3 cache is 2-banked with a total size of 4MB. The remaining configurations follow the ZSim defaults. Note that the OOO core means "Out-Of-Order" core. With OOO core, ALU computation and memory access have different queues in simulation, which makes the simulation memory-bounded if there are many memory accesses and the ALU computation is not heavy. Using 2 cores is enough to demonstrate all access patterns including two interesting edge cases. Using 2 cores also simplifies data collection.

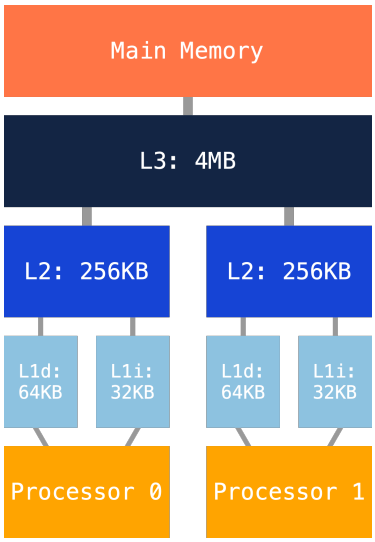

Figure 4: ZSim Simulation Configurations of Cache Hierarchy

Besides the same ZSim testing environment, we are also following the same ideology when writing the tests, which is to value simplicity. First, given that our tests need to compile and run in an outdated Linux version with old compilers and packages, we would like to avoid any too advanced language features or packages which can lead to extra dependencies so that the tests can run smoothly. Secondly, given that we want those micro-benchmark programs to generate primarily memory access patterns that we are interested in, we would like to avoid unnecessary computations of other kinds.

Moreover, we use the same data collection and analytics strategies. For each test, we collect data from four runs, two with default ZSim and two with our optimized ZSim. ZSim simulation is not deterministic, which means the same program can have different output statistics in different runs due to different numbers of instructions getting simulated. Thus, we will pick one run using default and one run using optimized ZSim with close numbers of instructions simulated for data analytics.

Our experiment results have three dimensions, that is, each data point can be identified by three parameters. The first parameter is the ZSim version, as we have default and optimized ZSim. The second parameter is the core ID or processor ID, as all tests are configured to run with a 2-core setup and the statistics provided by ZSim output also differentiate between the two simulated cores. The third parameter is the data category. We are interested in those raw statistics: the number of instructions simulated on a core, the number of cycles taken to simulate those many instructions on this core, the upgrade miss count on L1 data / L2 cache, the `INV` count on L1 data / L2 cache, and the `INVX` count on L1 data / L2 cache. Besides raw statistics, we calculate instructions per cycle (IPC) for each core with each version of ZSim. Then, using the IPCs, we calculate the average IPC improvement percentage as the metric to evaluate performance improvement.

We don't care about L3 statistics since it is the shared LLC. Moreover, since both cores' L1 and L2 caches have almost the same upgrade miss, `INV`, and `INVX` statistics in all tests, when plotting the test results we show the sum of L1 and L2 statistics. Raw statistics can be found in the Appendix section.

Finally, all outputs collected are from tests run on Haoyang Wang's Ubuntu 16.04 VM on his x86-64 PC. Zhuoyu Ji also successfully set up a similar Ubuntu 16.04 VM on AWS. However, after testing runs on each VM, it turns out that the basic AWS machine is not more powerful or efficient while executing simulations. We finally decide to collect all test outputs using Haoyang's machine.

Below, we go over the six tests.

4.2 Test 0

Test 0 is to verify that our implementation is effective in improving performance when cores are accessing migratory objects. The migratory object in this test is an integer array of length 32. In the micro-benchmark program, two

threads are repetitively attempting to enter a critical section and modify all values in the array. Using our optimized ZSim, the access pattern on the migratory object would be the same as Figure 3. Table 1 and Figure 5 are the results of this test.

Table 1: Instruction and Cycle Count of Test 0

Implementation	Core 0 ins	Core 0 cycle	Core 0 IPC	Core 1 ins	Core 1 cycle	Core 1 IPC	Average IPC improvement (%)
Base	40959334	51574552	0.7972943342	40775651	51142532	0.7972943342	0.1513786423
Optimized	40960190	51497937	0.7985053393	40776932	51066574	0.7985053393	

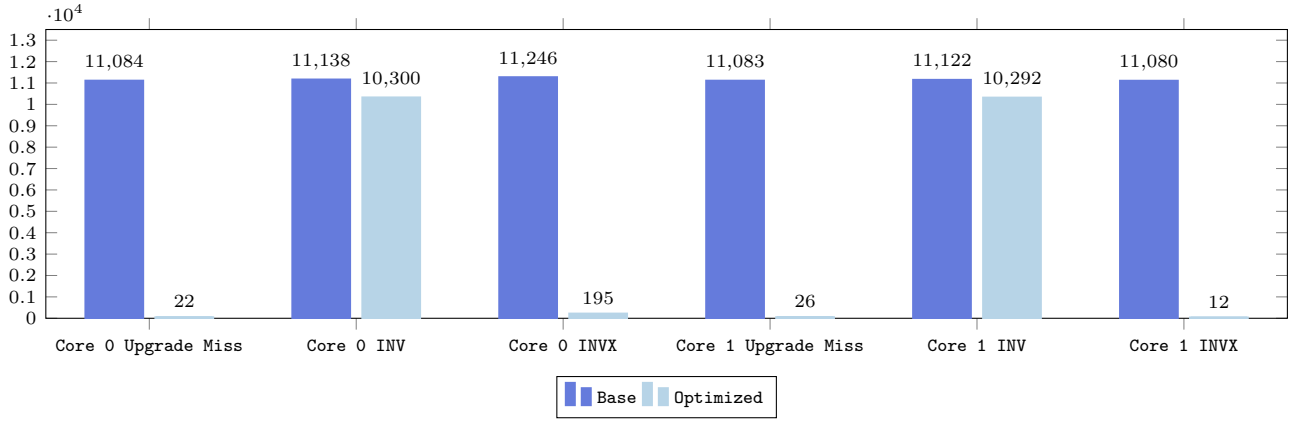

Figure 5: Stats for Test 0 (L1 & L2 counts combined)

As expected, our optimized ZSim greatly reduces the number of upgrade misses and downgrade invalidations in both cores. However, the average IPC improvement is only about 0.15%. This is also expected.

Our optimization is not like an optimization that implements a parallel version of a sequential program, which can result in a huge improvement. Instead, it's an optimization of an architecture for simulation, which is already highly optimized. So, in our case, even 1% improvement is significant. The major reason why our improvement is as small as 0.15% is that ZSim does NOT simulate traffic contention in cache coherence communications. For example, no matter how much traffic is there between L2 and L3 at one moment, if an invalidation message is sent from L3 to L2, its latency in ZSim simulation will always be a fixed number, which is not very large when we use the default latency configurations. And there is no way to configure those latencies to be dynamic, reflecting traffic contention. Therefore, even though we reduce a lot of cache coherence messages that could be generated by downgrade invalidations and upgrade misses, the improvement is not significant. If our optimization can be implemented in a real cache, we expect to observe a bigger improvement due to the amount of traffic contention our optimization can reduce.

4.3 Test 1

Test 1 is to verify that our implementation won't do anything special on non-migratory objects. The non-migratory object in this test is still the integer array with size 32. However, while one thread is repetitively modifying the array, the other is only repetitively reading the array. There is no critical section used for read operations as we don't care what is read. (The other reason is that if we use `pthread mutex` to protect the read operations, those read operations will be treated as read-exclusive according to output statistics. We didn't investigate why. An educated guess is that the compiler or Pin tool turns read requests inside `pthread mutex` protection into read-exclusive requests.)

This simple access pattern is a very naive consumer-producer pattern, which is mentioned by Per Stenström, Mats Brorsson, and Lars Sandberg as a pattern that should not be detected as migratory sharing. So, in both default and our optimized ZSim, the access pattern would be like Figure 6. Table 2 and Figure 7 are the results of this test.

Table 2: Instruction and Cycle Count of Test 1

Implementation	Core 0 ins	Core 0 cycle	Core 0 IPC	Core 1 ins	Core 1 cycle	Core 1 IPC	Average IPC improvement (%)
Base	40848255	51297351	0.7963033998	40478789	50444215	0.8024466036	-0.001147717365
Optimized	40848255	51297164	0.7963063026	40478789	50445551	0.8024253516	

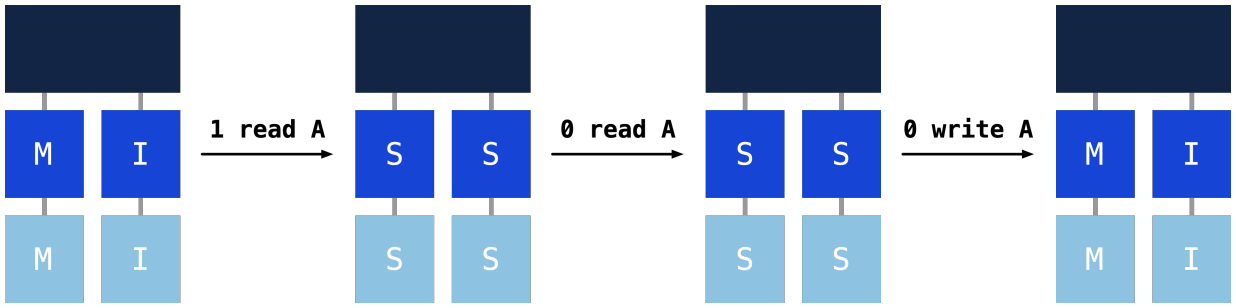

Figure 6: Standard Naive Producer-Consumer Access Pattern (2-core: 0, 1)

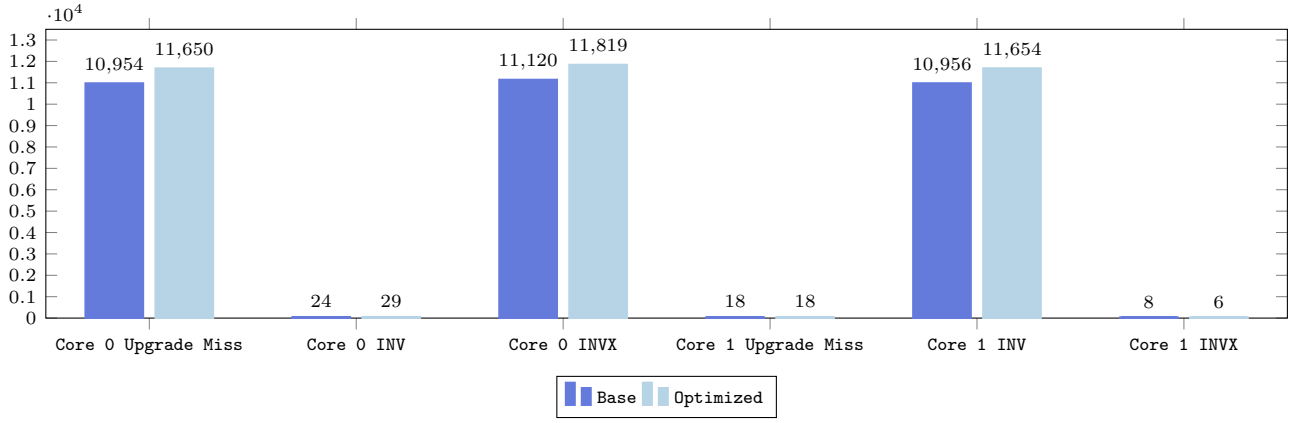

Figure 7: Stats for Test 1 (L1 & L2 counts combined)

As expected, both default and optimized ZSim have very similar output statistics, and the performance in terms of IPC is almost the same. While core 0 has many upgrade misses and downgrade invalidations with few true invalidations, core 1 shows the opposite statistics. So, we can conclude that in the simulation, core 0 is running the "producer" thread which is modifying the array and core 1 is running the "consumer" thread which is only reading the array.

4.4 Test 2 and Test 3

Test 2 and test 3 are extensions of test 0 with different migratory objects and increased workload. They are also to verify that our implementation is effective in improving performance when accessing migratory objects. The migratory object in test 2 is a custom linked list created on the heap with length 16, where each node is 16 bytes in size. The migratory object in test 3 is a custom struct array with size 32, where each array element is 16 bytes in size. The micro-benchmark programs are similar: two threads repetitively attempt to enter a critical section and modify all values in the migratory object. Using our optimized ZSim, the access pattern on the migratory object would be the same as Figure 3. Table 3, Table 4, Figure 8, and Figure 9 are the results of test 2 and test 3.

Table 3: Instruction and Cycle Count of Test 2

Implementation	Core 0 ins	Core 0 cycle	Core 0 IPC	Core 1 ins	Core 1 cycle	Core 1 IPC	Average IPC improvement (%)
Base	101414640	128525342	0.789063374	101227078	128102355	0.790204661	0.1543741893
Optimized	101414944	128328283	0.7902774169	101227243	127904454	0.7914286003	

As expected, our optimized ZSim greatly reduces the number of upgrade misses and downgrade invalidations in both cores. The average IPC improvement metrics, which are about 0.15% for test 2 and 0.13% for test 3, are similar to test 0.

4.5 Test 4 and Test 5

Here start the interesting edge cases. Recall the problem: what if after the migratory object detection some processor no longer modifies the migratory object? Simply put, our implementation cannot detect this alteration of access patterns. The reason why we didn't implement this alteration check will be discussed later.

Test 4 is the first edge case in our 2-core scenario: both threads start to work as they do in test 3 to try modifying the

Table 4: Instruction and Cycle Count of Test 3

Implementation	Core 0 ins	Core 0 cycle	Core 0 IPC	Core 1 ins	Core 1 cycle	Core 1 IPC	Average IPC improvement (%)
Base	105487777	131609804	0.8015191406	105303021	131192108	0.8026627715	0.1302873499
Optimized	105492247	131444466	0.8025613418	105310317	131030143	0.8037106164	

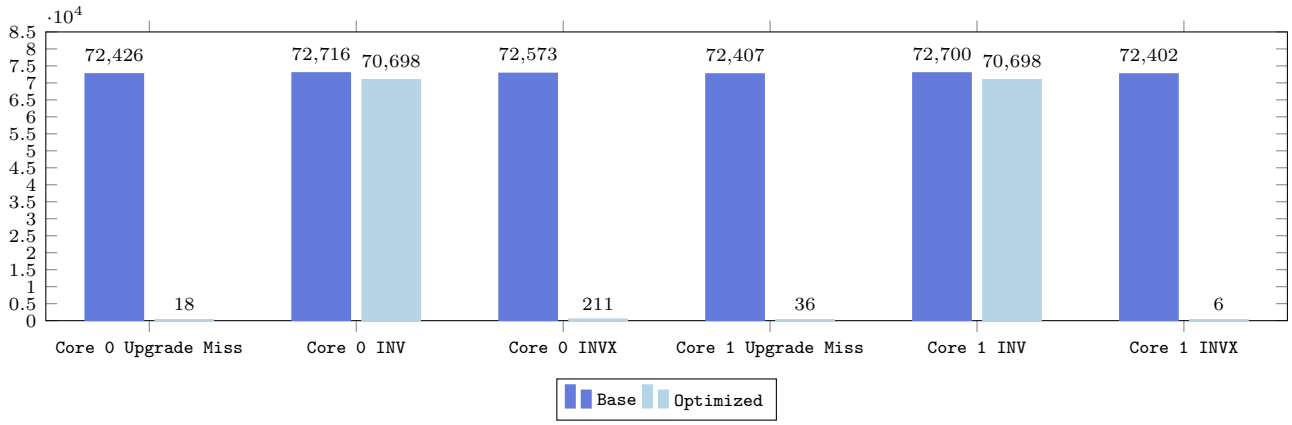

Figure 8: Stats for Test 2 (L1 & L2 counts combined)

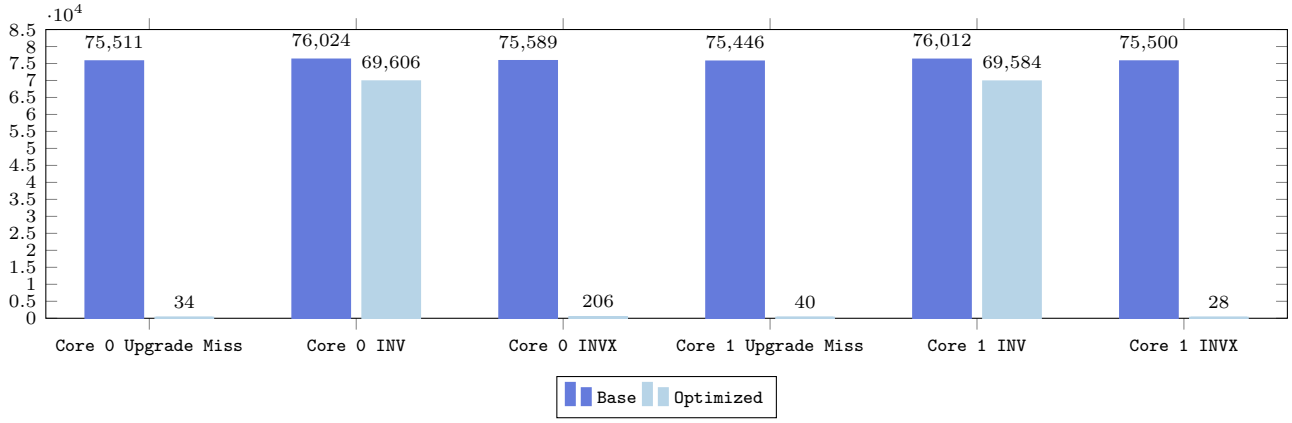

Figure 9: Stats for Test 3 (L1 & L2 counts combined)

struct array. However, after 50 iterations (which is enough to trigger migratory object detection), one thread changes its behavior to only read the struct array. So, in this 2-core edge case, a migratory sharing access pattern changes to a naive producer-consumer access pattern, and our optimization cannot detect this alteration.

After the alteration, in default ZSim, the access pattern would be like Figure 6. By contrast, in our optimized ZSim, the access pattern would be like Figure 10. As we observe from comparing two access patterns, the downgrade invalidation in Figure 6 turns into the first true invalidation in Figure 10; the true invalidation in Figure 6 turns into the second true invalidation in Figure 10. Besides, the one upgrade miss in Figure 6 is eliminated in Figure 10. Given that the total number of invalidations is still the same in those two patterns, we expect the performance to be close, which means our optimized ZSim will not degrade in this 2-core edge case. Table 5 and Figure 11 are the results of this test.

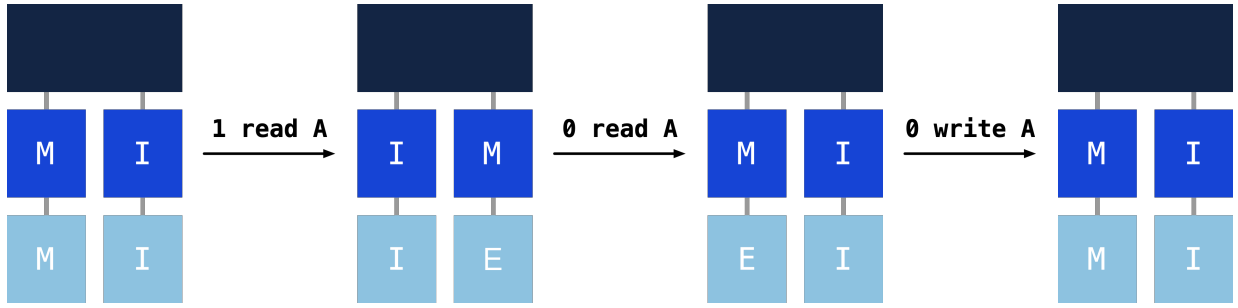

Figure 10: Naive Producer-Consumer Access Pattern Treated as Migratory Sharing (2-core: 0, 1)

Table 5: Instruction and Cycle Count of Test 4

Implementation	Core 0 ins	Core 0 cycle	Core 0 IPC	Core 1 ins	Core 1 cycle	Core 1 IPC	Average IPC improvement (%)
Base	105473191	131292665	0.8033441244	104077090	128027300	0.8129288831	0.002278383362
Optimized	105473152	131290458	0.8033573316	104077044	128023524	0.8129525008	

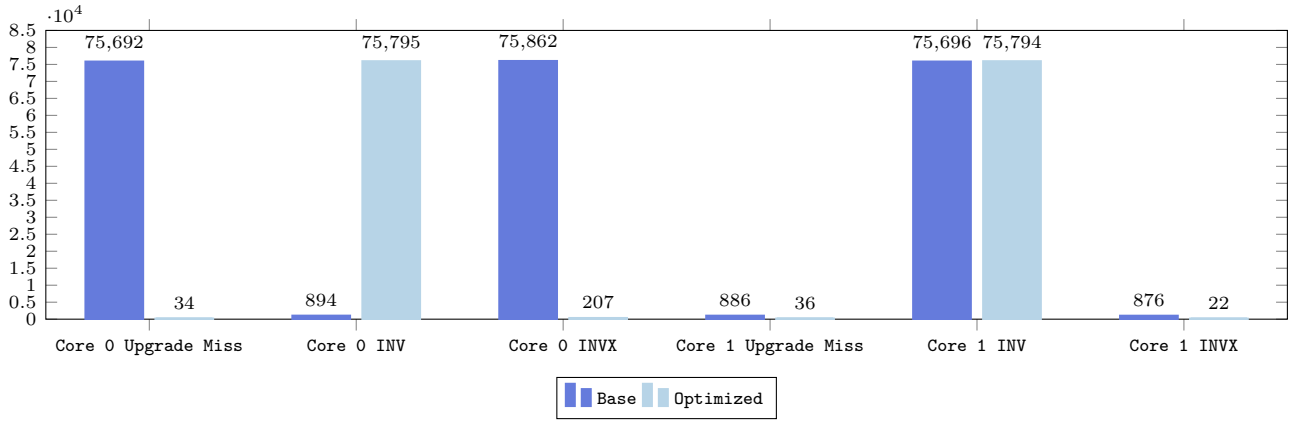

Figure 11: Stats for Test 4 (L1 & L2 counts combined)

As expected, while core 1 under both default and optimized ZSim has similar upgrade miss, `INVX`, and `INV` statistics, core 0 shows the opposite statistics between default and optimized ZSim. Yet, despite the differences, the average IPC improvement is close to zero, which means equivalent performance. Therefore, our limited implementation, which can not detect access pattern alteration, will not degrade in a 2-core scenario when one of the two processors decides to no longer modify the migratory object.

But, what if both processors eventually decide to no longer modify the migratory object? Test 5 investigates this edge case.

Test 5 is the second edge case in the 2-core scenario: both threads start to work as they do in test 3 to try modifying the struct array. However, after 50 iterations, both threads change the behavior to only read the struct array. So, in this 2-core edge case, a migratory sharing access pattern changes to a read-only access pattern.

Here, we can directly predict that our optimized ZSim would degrade the performance. The reason is simple: after alteration to read-only access pattern, in default ZSim, the relevant cache lines will always stay in S state shared by two processors. However, in our optimized ZSim, as we cannot detect this alteration, the relevant cache lines will

always be considered migratory until getting evicted; as a result, a read request from one processor will always be treated as read-exclusive and trigger a true invalidation if necessary. The processors would waste cycles invalidating each other. Table 6 and Figure 12 are the results of this test.

Table 6: Instruction and Cycle Count of Test 5

Implementation	Core 0 ins	Core 0 cycle	Core 0 IPC	Core 1 ins	Core 1 cycle	Core 1 IPC	Average IPC improvement (%)
Base	104260413	127778163	0.8159485984	104077044	127345461	0.8172811436	-0.5407563013
Optimized	104260600	128481455	0.8114836495	104077217	128029753	0.8129142997	

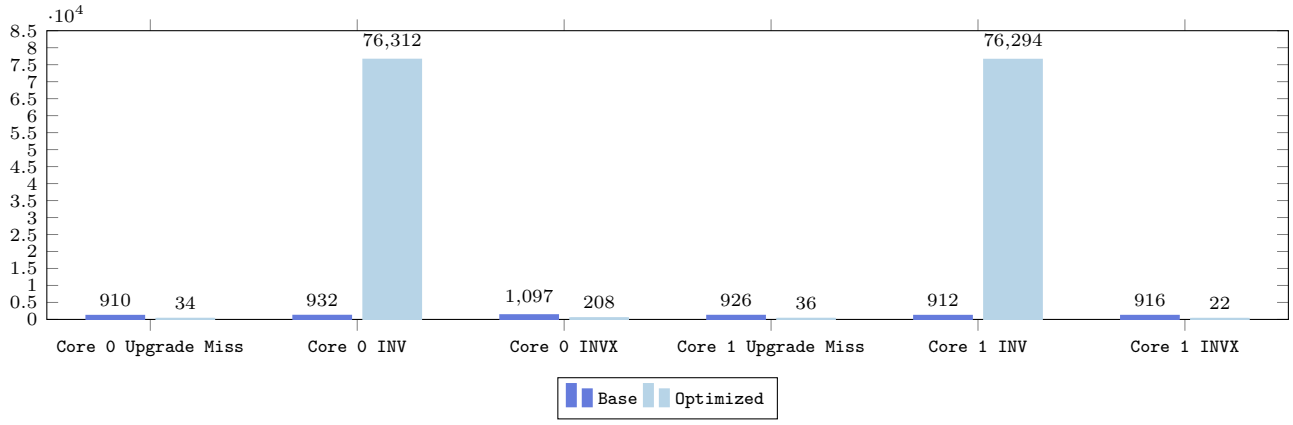

Figure 12: Stats for Test 5 (L1 & L2 counts combined)

As expected, `INV` statistics stand out to be very high in our optimized ZSim and the average IPC improvement metric is -0.54%, which is a rather large degradation compared to the improvement we can achieve in earlier tests.

4.6 The Solution to Alteration

We will discuss how to fix this problem caused by access pattern alteration. In their paper, Per Stenström, Mats Brorsson, and Lars Sandberg propose the following solution: when it's processor 0's turn to use the migratory object, the first read request gets an exclusive copy in "migrating" state. Then, two events X and Y are defined. Event X happens when processor 0 modifies the migratory object. Event Y happens when processor 1 sends a read request on the migratory object. If Event X happens first, the modification triggers a silent state change creating no cache coherence communication, and the object stays migratory; if Event Y happens first, the migratory object is no longer migratory and the object is then shared in S state in both processors.

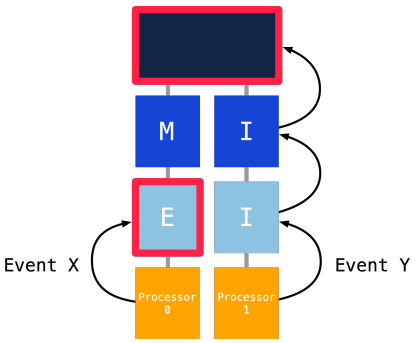

Figure 13: Our Limited Implementation (Separation of Event X and Y)

This solution works and the key is to know whether Event X or Event Y happens first; however, in current ZSim, it's impossible to know which comes first. In Figure 13, taking our 2-core cache system as an example, after processor 0 gets the exclusive copy of a migratory object, only its L1 data cache knows if processor 0 makes a subsequent `GETX` request to modify the object since this `GETX` would be a silent hit, i.e. E state silently upgrades to M state. On the other hand, if processor 1 sends a read request on the migratory object, eventually L3 gets the request and has the privilege to handle it, but L3 does not know if processor 0 has modified the migratory object and L3 also won't inform processor 0's L1 that processor 1 wants to read the migratory object. As a result, L3 would assume processor 0 has

finished its turn and give processor 1 an exclusive copy of the migratory object and send invalidation messages to invalidate the copy in processor 0. Simply put, in ZSim, the detection of Event X and Event Y happens in two separate places and there is no communication workflow to synchronize the information. Therefore, the solution introduced by Per Stenström, Mats Brorsson, and Lars Sandberg cannot be easily implemented in ZSim.

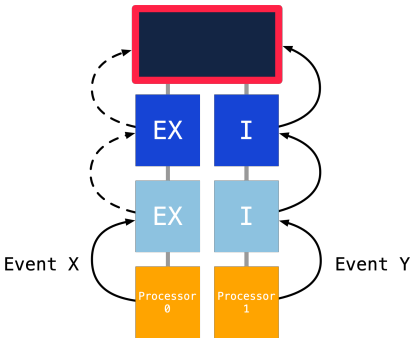

Figure 14: ZSim Alteration Check Solution (L3 Listens to Event X and Y)

Yet, we can modify the ZSim design to fit into this idea as shown in Figure 14. First, we introduce a new state EX: when it's processor 0's turn to use a migratory object, the first read request gets an exclusive copy but in EX state. If processor 0 sends a subsequent `GETX` request on the migratory object to modify it, EX transitions to M state and a special message is sent to parent nodes until reaching the cache shared by all processors (L3 in our case). In this way, L3 should be able to know whether Event X or Event Y happens first; and, based on this, L3 can decide whether the migratory object is no longer migratory. But, as this implementation requires adding a new state to the state machine and adding a new kind of communication workflow, we decide not to implement it in this project given the complexity and risk of breaking the ZSim code base.

4.7 Conclusion

To conclude, we verify that our optimization is effective to improve performance on consistent migratory sharing access patterns. We also understand why such optimization is rarely implemented in real-world cache architectures by analyzing the tradeoffs. For benefits, it is only optimizing a special access pattern, which takes a very small portion of all accesses. Besides, in our experiments where the micro-benchmark programs primarily follow the migratory sharing access pattern, the observed performance improvement is still very small. On the other hand, for costs, in ZSim, our limited implementation with no alteration check needs one extra boolean and one extra index number per cache line; in a real directory-based cache system, the boolean flag is one bit and the index number can be encoded in several bits. Those extra bits per cache line are a huge cost. Even if the hardware has the capacity to hold those bits, an architecture designer would surely use them to optimize other general features instead of spending them purely to improve performance when accessing migratory objects. Therefore, given the tradeoffs, a migratory sharing optimization may not be worth it in real directory-based caches.

5 Reflection

5.1 Shift of Direction

The original plan of the team was to implement MOESI and MESIF protocols in ZSim. However, after studying the ZSim source code closely, the team realized that it's ineffective to implement MESIF in ZSim as its cache system is fully directory-based and also found difficulty implementing MOESI due to many unknowns in data movement simulation.

Fortunately, the team was introduced to Per Stenström, Mats Brorsson, and Lars Sandberg's paper. After reading the paper, as the team was already very familiar with the cache coherence parts of ZSim, the team determined that implementing the migratory sharing optimization would be effective and less risky in ZSim because the implementation would not require huge modifications in the source code. As a result, the team shifted direction in this project.

5.2 Comments on ZSim

At the very beginning, the team was interested in optimization based on ZSim because the team wants to expand an existing reliable multicore cache simulator instead of building a naive one from scratch. And as one member has a not very good experience using the Gem5 simulator, the team decided to focus on ZSim.

However, the overall ZSim experience has both positives and negatives. For positives, first, ZSim presents huge flexibility in simulation controls. There are so many configuration options, many of which we are still not familiar

with. Second, the ZSim source code is created with wisdom. After getting to understand most of the cache coherence related source code, we found the recursive request processing, which utilizes the tree structure of the cache hierarchy and has latency calculation embedded, such an elegant design.

On the other hand, the negatives are also noticeable. First, there is almost no documentation as the ZSim developers propose a "the documentation is the source code" idea. In addition, there are few tutorial resources on ZSim. Those make learning the ZSim source code very difficult, especially at the beginning. Fortunately, thanks to Ziqi Wang's article [2], we were able to eventually understand most of the source code related to cache coherence.

The second negative is that ZSim is currently outdated in a sense. The official source code [3] has not been updated for years and is reported to be only runnable on Ubuntu 12.04. The team tried to configure a proper Ubuntu 12.04 environment but encountered many problems. Again, thanks to Ziqi Wang for his open-source ZSim-base [4], which fixes many issues and makes ZSim runnable on Ubuntu 16.04. This helps the team a lot.

Overall, the team learns that expanding an existing open-source project, especially one like ZSim which is a very complicated multicore simulator, is not an easy job. We are able to finish this project smoothly because the migratory sharing optimization fits very well with how ZSim internals are designed.

5.3 Potential Improvement

There is still room for improvement regarding our migratory sharing optimization implemented in ZSim. First, if we have more time, we could implement the complicated solution mentioned earlier for alteration check. Second, if we have the ideal testing machine, we could run some standard benchmark programs. Third, if we get to know more about ZSim, especially those simulation configuration details, we could make the testing environment more realistic (for example, simulating an existing architecture). Finally, we could try to add traffic contention simulation to ZSim so our improvement metrics could be more accurate.

6 Contributions

Tasks are distributed evenly between team members. ZSim source code modification, micro-benchmark programming, and report writing are done by both members. Haoyang Wang is independently responsible for performing the tests on his machine and managing the website. Zhuoyu Ji is independently responsible for creating all graphs and visuals for demonstration and analytics.

7 Appendix

Project Website

Our Optimized ZSim

All Source Code with a Compilation and Test Guide (Autolab Submission)

Raw Test Outputs

Data for Analysis

References

[1] Per Stenström, Mats Brorsson, and Lars Sandberg. "An adaptive cache coherence protocol optimized for migratory sharing". In: *ACM SIGARCH Computer Architecture News* 21.2 (1993), pp. 109–118. DOI: 10.1145/173682.165147.

[2] Dec. 2019. URL: https://wangziqi2013.github.io/article/2019/12/25/understand-zsim-cc-sim.html.

[3] s5z. *S5Z/zsim: A fast and scalable x86-64 multicore simulator*. URL: https://github.com/s5z/zsim.

[4] wangziqi2013. *WANGZIQI2013/zsim-base: Base repo of a workable zsim on newer version of ubuntu, with PIN-2.14 binary (the original zSim no longer works)*. URL: https://github.com/wangziqi2013/zsim-base.