Optimizing Cache Configuration for Warehouse Robots via Simple Software Simulation

Daiwei Shen
daiweishen2024@u.northwestern.edu
Northwestern University
Evanston, Illinois, USA

Abstract

asfasdfdsafsadfdsafsadasdfasdf

1 Introduction

2 Cache Access Simulator

2.1 Implementation

In order to study the impact of the cache configuration on overall computing speeds on the tasks relevant to warehouse robots, it is necessary that data on design and performance metrics of various cache configurations are collected. Due to the costs and infeasibility of performing such experiments with physical devices, the best alternative is to simulate the behavior in software. To this end, the author has implemented a Python program[1] which can generically simulate the core functionality of access by the processor to the cache as well as the main memory.

The simulator does not attempt to represent any specific design or architecture associated with any known processor. Rather, it provides a generic model which describes caching and memory access at a high abstraction level, and can be used to roughly approximate relative performance between different cache designs in a range of processors and systems. While such simulators are best implemented in lower-level languages like C or C++ for the benefit of simulation speed, Python is chosen because of its syntactical simplicity, and a lack of necessity for quicker simulation. Specifically, the implementation does not simulate any other component of the CPU beyond a single layer of generic L1 data cache which directly interacts with the main memory and an unspecified processor core. In the interest of simplicity, and because a uniprocessor model can adequately capture the high-level essence of a cache, the implementation in question does not attempt to simulate the cache and all of its complexities and nuances in a multiprocessor environment (e.g. cache coherence and other data sharing mechanisms are ignored).

Beyond the cache, it is not necessary to simulate other components of the processor because the study in question depends primarily on capturing latency, which can be measured in cycle count without simulating the execution of non-memory operations. In cases where such an operation exists, a "cycles" variable can simply be incremented by the amount of cycles the execution would require. In addition, the simulation of the cache is at an abstracted level which does not involve any layer of abstraction below a logical representation of the relevant mechanisms. That is, the cache is simply represented as a collection of data arrays and functions of logical operations in the program, and the interactions of the cache are captured at purely a logical level. To this end, the cache has a number of features: an address decoder, which is used to translate memory addresses into cache tag, set index (if available), and block offset bits; a stats logger, which tracks the number of hits and misses, the miss rate, and the number of cycles of execution; a block replacement unit, which supports both Least Recently Used (LRU) and Random Replacement (RR) policies; a write line, which supports any combination of write-through/write-back and write-allocate/write-no-allocate, writing of data to both the cache and the main memory, and updating of valid bits, dirty bits, and LRU state; and a read line, which supports reading of data from the cache or from main memory, and updating of valid bits and the LRU state. In contrast to the complexity of the cache implementation, the main memory module is much simpler: it is just a data array with read and write lines.

2.2 Parameterization

In order to study design trade offs of cache in a system, it is necessary that multiple different configurations are analyzed. Thus, the simulator is implemented with a number of modifiable parameters. These parameters include number of sets, associativity, block size, write policy, write allocation, and replacement policy. These parameters can have a profound impact on caching performance, as later illustrated in the Results section. Still, there are some limitations to the possible parameters. As discussed above, only two options are implemented for each of write policy, write allocation, and block replacement policy. Furthermore, the block size and set index must both be a power of 2 because of incorrect behavior otherwise.

2.3 Powers of 2

Attempts to implement a cache that uses a number of sets and/or a block size which are not powers of 2 can lead to multiple issues. For instance, if a non-power-of-2 number of sets is allowed, say 6, its representation would require 3 bits. To address the problem of indexing out of range (e.g. 111 = 7 would index the 8th set, which doesn't exist), a simple fix

[1] insert github

is to perform the modulo operation, such as set_idx % 6, so that the index wraps around. However, this fix introduces a new problem: because indices 6 and 7 wrap around to indices 0 and 1, respectively, it is now possible for multiple blocks to have the same cache tag and be present in the same set, which violates one of the invariants of set-associative caches. Specifically, two different addresses reflecting different memory locations can now map to identical bits. Of course, it is always an option to represent 6 sets with only 2 bits, but then 2 of the sets won't be accessed and the cache will behave identically to one that has only 4 sets. Using a block size that is not a power of 2 has similar consequences. Indexing bytes which are out of range can be addressed in a similar fashion as that for number of sets. However, once again, it is possible for entirely different memory addresses to end up as identical addresses. To avoid these issues and to keep the simulator simple, support for parameters which are not powers of 2 is not implemented.

2.4 Reads

The simulator handles read requests with a dedicated read module and in a sequential manner, meaning no interleaving of read operations is possible. This behavior keeps the simulation simple and intuitive. On a read request, the address is first decoded using the address decoder module. Then, a search is done into the set to which the address maps for the tag. This procedure holds even if the cache configuration is direct mapped or fully associative. If the cache is DM, each block is treated as its own set; if the cache is FA, all the blocks are treated as though they belong to a single set. If there is a tag match, data of some specified input size is loaded from the corresponding block, and the stats logger unit registers a cache hit and a cycle count, which has been estimated as 1 cycle for read hits[2]. On the other hand, if a tag match is not found in the corresponding set, a request is sent to the block replacement unit to check if every block in the set has its valid bit set. If this condition is true, then the set is full, and one of the blocks needs to be evicted according to the specified replacement policy. On the other hand, if the set is not full, the block replacement unit will return the block index of the first empty block in the set from ascending order. The block index in question is returned to the main read module, which can now overwrite the block with new data. Before this is done, a block must be loaded from memory, which requires that the address be aligned to the block size. The alignment is performed by taking the address of a word which resulted in a miss on a particular block and subtracting the address % block size. A block starting at this new address is subsequently loaded from the main memory module and written into the cache block specified by the block replacement unit, and the valid bit is set. The stats logger will register a read miss as well as its associated cycle count, estimated to be 200 cycles based on InstLatx64[3]. In cases of both cache hits and misses, the requested data of a specified input size is returned by the read module.

2.5 Writes

Writes are handled in much the same way as reads. There is a dedicated write module which handles write requests in a sequential manner. When a write request is sent, the address decoder module is called to decompose the address into the relevant bits. The write module checks for a tag match in the set in a similar fashion as done for reads. On a tag match, a write hit is registered and the input data of some specified size is written into the corresponding block in cache. If the write policy is write-through, the block will be subsequently written to the memory module, and a write cycle count of 300 clock cycles is registered according to figures from InstLatx64. If the write policy is write-back, the dirty bit for the block is set, and no operation is made on the main memory. If instead a tag match is not found, then there are two possibilities. First, if write-allocate is specified, then a request is sent to the block replacement unit, which checks for possible unoccupied cache lines or a potential block eviction according to the replacement policy. If there is a conflict and if the write policy is write-back, the dirty bit of the conflicting block is checked. If the bit is set, then the block is written to the memory module. In either case, a block index is returned to the write module, which subsequently loads the missed block from the memory module into the block specified by the block index. On the other hand, if write-no-allocate is specified, then no block replacement is necessary and on a write miss, the data will simply be written to the memory, but not to the cache. In all cases of a write miss, a cycle count of 300 is registered. For write hits a cycle count of 1 is registered, except when write-through is used where a cycle count of 300 is registered.

2.6 Least Recently Used

The LRU replacement policy is implemented in the following way. A LRU queue is used to keep track of block accesses. Each time a block is requested, whether through a write or a read, it is added to the queue. If the block is already in the queue, then it is removed (possibly from the middle) and readded to the back of the queue. During block replacement, if all blocks in the set are occupied, a dequeue operation is performed on the LRU queue, and the resulting block is evicted.

2.7 Random Replacement

Random replacement is much simpler than LRU. It involves generating a random integer in a uniform distribution on

[2]In real systems, cache hits on L1 can take longer.

[3]Memory access cycles on a modern processor like the Intel Core i9 10900k is estimated to be between 243-288 clock cycles, according to a range of benchmarks: http://instlatx64.atw.hu/

the interval [0, associativity]. The resulting number is then used to specify the block that is chosen for eviction and subsequent replacement.

3 Test Code

A small piece of simple C code is written to approximate potential code sections in warehouse robot programs:

```
bool obj_check(int obj, int* obj_list, int len) {
    int curr_obj;
    for (int i = 0; i < len; i++) {
        curr_obj = obj_list[i];
        if (obj == curr_obj) {
            return true;
        }
    }
    return false;
}

//item_sighted/item: signals from computer vision
//other signals are from other code sections
//not shown here

void perform_task(int item_sighted, int item,
    int direc, int* obj_list,
    int len, int speed, int arm, int grab) {
    if (item_sighted) {
        speed = 0;
        arm = direc;
        int item_match = obj_check(item,
        obj_list, len);
        if (item_match) {
            grab = 1;
        }
    //theoretical wait (i.e. spin) here.
        speed = 5;
        arm = ~direc;
        grab = 0;
    }
}
```

This C code is by no means comprehensive, and is instead intended to be a small sample of a much larger program with many more dependencies. In addition, while the written code is treated as a one-pass section in simulation, it is likely to be executed repeatedly over time with different inputs in a real environment. The reasons that the above code section is used in this study are the following. First, the larger programs that the robots run in a real world setting are not available for public access; second, they require special hardware to run correctly (e.g. camera feed from the robot, joints and arms, motors, sensors, etc), and such devices are not accessible to the author; third, the actual program can vary from specific robots to specific use cases, and thus confining the study to any specific program would not align with the study's goal of analyzing cache configuration on a general, abstract level. Therefore, the simple C code section is a fitting alternative which represents a sample of the much larger programs, and can roughly capture some general operations and memory access patterns of these programs. In particular, a typical task a warehouse robot performs may involve checking a particular set of items in the warehouse and either updating or requesting their associated information. As another example, a robot may search for a particular set of items at a point in time in order to displace said items to a different location, or operate on the items in some other way. Regardless, these tasks can be captured by iterative behavior such as for loops or while loops in code, as done above. Notably, they involve repeated memory accesses which potentially exhibit a high degree of temporal and/or spatial locality. These properties can be exploited by certain cache configurations to dramatically increase computational efficiency, which is the subject of this study.

In order to represent the sample code in the cache simulator, it is compiled into RISC-V assembly, which can be parsed by the simulator as individual instructions:

```
//obj_check(int, int*, int):
addi    sp, sp, -32
sw      ra, 28(sp)
sw      s0, 24(sp)
addi    s0, sp, 32
sw      a0, -16(s0)
sw      a1, -20(s0)
sw      a2, -24(s0)
li      a0, 0
sw      a0, -32(s0)
j       .LBB0_1
.LBB0_1:
lw      a0, -32(s0)
lw      a1, -24(s0)
bge     a0, a1, .LBB0_6
j       .LBB0_2
.LBB0_2:
lw      a0, -20(s0)
lw      a1, -32(s0)
slli    a1, a1, 2
add     a0, a0, a1
lw      a0, 0(a0)
sw      a0, -28(s0)
lw      a0, -16(s0)
lw      a1, -28(s0)
bne     a0, a1, .LBB0_4
j       .LBB0_3
.LBB0_3:
li      a0, 1
sb      a0, -9(s0)
j       .LBB0_7
.LBB0_4:
```

```
j       .LBB0_5
.LBB0_5:
lw      a0, -32(s0)
addi    a0, a0, 1
sw      a0, -32(s0)
j       .LBB0_1
.LBB0_6:
li      a0, 0
sb      a0, -9(s0)
j       .LBB0_7
.LBB0_7:
lbu     a0, -9(s0)
lw      ra, 28(sp)
lw      s0, 24(sp)
addi    sp, sp, 32
ret
//perform_task(int, int, int, int*,
//int, int, int, int):
addi    sp, sp, -48
sw      ra, 44(sp)
sw      s0, 40(sp)
addi    s0, sp, 48
sw      a0, -12(s0)
sw      a1, -16(s0)
sw      a2, -20(s0)
sw      a3, -24(s0)
sw      a4, -28(s0)
sw      a5, -32(s0)
sw      a6, -36(s0)
sw      a7, -40(s0)
lw      a0, -12(s0)
beqz    a0, .LBB1_4
j       .LBB1_1
.LBB1_1:
li      a0, 0
sw      a0, -32(s0)
lw      a0, -20(s0)
sw      a0, -36(s0)
lw      a0, -16(s0)
lw      a1, -24(s0)
lw      a2, -28(s0)
call    obj_check(int, int*, int)
sw      a0, -44(s0)
lw      a0, -44(s0)
beqz    a0, .LBB1_3
j       .LBB1_2
.LBB1_2:
li      a0, 1
sw      a0, -40(s0)
j       .LBB1_3
.LBB1_3:
li      a0, 5
sw      a0, -32(s0)
lw      a0, -20(s0)
not     a0, a0
sw      a0, -36(s0)
li      a0, 0
sw      a0, -40(s0)
j       .LBB1_4
.LBB1_4:
lw      ra, 44(sp)
lw      s0, 40(sp)
addi    sp, sp, 48
ret
```

The instructions are classified into 3 categories: read, write, and non-memory-access. This instruction stream is passed into the simulator, and a simulation can be run by calling the cache module with either read or write requests for each of the relevant instruction classifications.

4 Memory Access Fraction

Based on the scenarios described in section 4, a logical conclusion might be that the fraction of instructions which are memory accesses executed by the onboard CPU of the robot will increase as the size of the problem (i.e. number of items to update) increases.

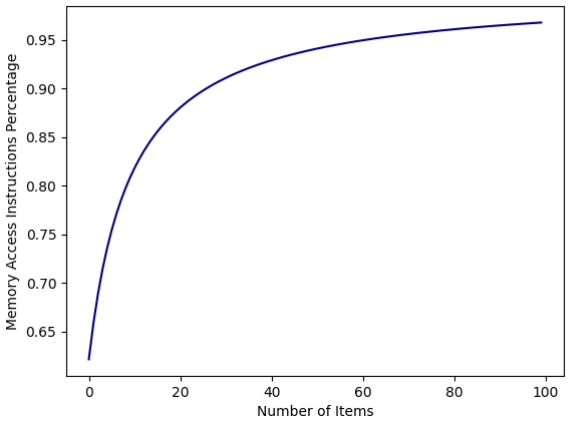

Figure 1

Indeed, an analysis of the relationship shows that such is the case, especially when the problem size is between 0 and 100, as shown in Figure 1. Figure 2 shows a much less dramatic increase in memory access fraction as the problem size increases; nonetheless, the memory access fraction approaches 1. These results confirm that under scenarios where the problem size is large, a decrease of AMAT[4] can lead to a significant speedup overall according to Amdahl's law [1]. Specifically, with N_{t1} = new execution time of a fraction-based speedup, N_{t2} = new execution time of an

[4]Average memory access time.

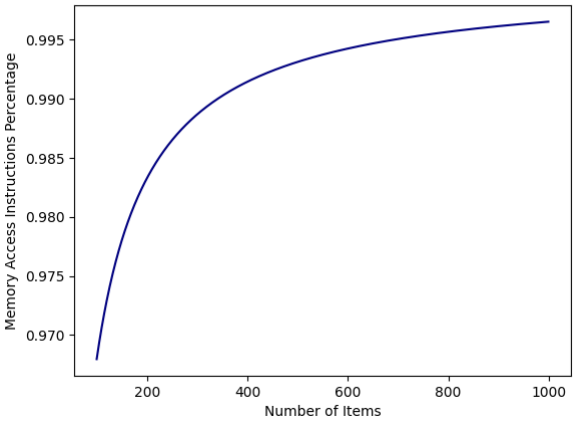

Figure 2

overall speedup, O_t = old execution time, S = speedup, F = fraction affected by speedup, we have the following:

$$N_{t1} = \frac{O_t}{S} \times F + (1 - F) \times O_t \tag{1}$$

$$N_{t2} = \frac{O_t}{S} \tag{2}$$

For a speedup which affects a large fraction $F1 \approx 1$,

$$N_{t1} = \frac{O_t}{S} \times F_1 + (1 - F_1) \times O_t \tag{3}$$

$$(1 - F_1) \times O_t \approx 0 \tag{4}$$

$$\frac{O_t}{S} \times F_1 \approx \frac{O_t}{S} \tag{5}$$

$$N_{t1} \approx N_{t2} = \frac{O_t}{S} \tag{6}$$

Therefore, if a fraction affected by speedup is large, the resulting overall speedup is similar to that of an application of the performance enhancement to the overall system. Because memory accesses account for a large fraction of the instructions when the problem size is large, a speedup to these instructions can be approximated as a speedup to all instructions. Any performance enhancement of this area can thus have a significant impact on overall performance.

5 CO_2 Emissions Savings

An efficient cache configuration can lead to a speedup of execution time of a program. Such a speedup will allow a warehouse robot to perform the same computational work in significantly less time, thus requiring substantially less cumulative energy over time. Because carbon footprint is required to generate this energy, computational speedup of the robot CPU through efficient cache design will ultimately lead to savings in carbon emissions and greenhouse gases (GHGs), increasing the sustainability of the robots. A simple equation has been devised to estimate the carbon footprint of a particular cache configuration,

$$C_E = \frac{10^{-3} C_P O_H F_C}{S} \tag{7}$$

The CPU power usage C_P is estimated based on the UltraSPARC T1 processor, which consumes 63W [2], a figure similar to power usage metrics of more modern processors[5]. O_H is the number of hours of operation of the robot over its lifespan, and is conservatively estimated to be 10,000 hours. F_C is a conversion factor between power usage in kilowatt-hours and carbon-equivalent footprint in kilograms. The factor is estimated according to the Greenhouse Gas Equivalencies calculator created by the EPA[6]. For the calculations of this study, the factor is estimated as $F_C = .41690375$. S is the performance speedup of a particular cache configuration over a naive design. The expression is scaled by the reciprocal of the speedup S to capture the proportionally lower operation time, and by extension, energy usage and carbon footprint, for a CPU with a faster cache to complete the same amount of computational tasks.

6 Simulation Setup

Two separate experiments are conducted to observe the performance of different cache configurations on different code sections. In the first experiment, the instruction stream tested involves only a simple for loop and thus the memory access pattern exhibits only spatial locality (assume the iteration variable is stored in a register). In the second experiment the program specified in section 4 is used, which has a mix of memory accesses that exhibit both spatial and temporal locality. In both cases, the main memory is initialized with random data. The specific data used is irrelevant for the purposes of these experiments because the simulation does not involve specific computation or operations on said data. Rather, the focus is on hits and misses, which deal with the addresses and tags.

6.1 Loop Simulation

The purpose of this experiment is to observe caching behavior on a simple yet common memory access pattern. Loop accesses tend to exhibit spatial locality, specifically when the address accessed is incremented with each loop iteration. For instance, in a loop that indexes each element of an array one by one, the address is incremented by 1 on each iteration. As a result, following any access to a particular memory address, the neighboring addresses will be accessed in the near future. In a single loop no address is repeated, and as such, no temporal locality exists. In this experiment, the simulation iterates through a for loop which has a write

[5]https://www.intel.com/content/dam/doc/white-paper/resources-xeon-measuring-processor-power-paper.pdf
[6]https://www.epa.gov/energy/greenhouse-gas-equivalencies-calculator

instruction followed by a read instruction. Both instructions operate on a 4-byte size and on the same address in memory, which is randomized on a uniform integer distribution from 0 to 1999. In a single trial, the randomized address is fixed and the loop iterates 100 times representing robot computation on 100 items. For a particular cache configuration, 100 trials are performed, each trial with a different seed and thus a different randomized address. The average performance of the 100 trials is recorded for each cache configuration tested.

6.2 Program Simulation

This experiment is more sophisticated in that it is conducted using the code section discussed in section 4. It therefore captures a mix of memory accesses exhibiting both spatial and temporal locality. While the loop simulation tests a limited number of aspects of the cache design, this experiment tests the effectiveness of all 6 of the configurable parameters. Each of the variables declared in the code in section 4 is assigned a different address. Like in the experiment above, these addresses are taken from non-overlapping uniform integer distributions of 2000 in length, starting from 0 and ending at 20000. Accesses are simulated on these addresses both inside loops, as well as outside, depending on the variable. The first loop has 2 reads followed by 1 write and 2 more reads. After the loop, there are 3 reads followed by a single read and a write. Then, the first loop is simulated again with the same accesses as before, followed in order by 3 writes, 1 read, and 2 writes. Unless specified otherwise, the simulations on the program use a loop iteration of 100, corresponding to a robot operating on 100 items. A single trial is defined as running the simulation on a particular cache configuration and a fixed set of addresses, which involves simulating the entire code section using 100 iterations (items) for the loops. On each trial, a new set of random addresses are generated. In the experiment, the average performance of 100 trials is recorded for each cache design.

6.3 Why Randomize Addresses?

We find that the address at which variables are located has a profound impact on caching performance. For instance, the miss rate can differ by as much as 70% between simulation runs on the same set of instructions with different addresses. The reason for this discrepancy is because of the way sets are assigned to addresses based on the set bits. If multiple addresses map to the same set, the number of conflict misses can become large as blocks in the cache which will be reaccessed in the near future will likely be evicted to make space for incoming blocks. The lower the associativity, the more problematic this phenomenon becomes. In a direct mapped cache, if all accesses are to addresses that map to the same "set", then every incoming block will cause the previously accessed block to be evicted, leading to a pathological thrashing situation. Of course, address-induced conflict misses are not present in a fully associative cache. However, for testing non-FA cache configurations it is important to account for variations in cache performance when using different variable addresses. Randomizing the variable addresses and taking the average performance over a large number of simulation runs is an effective way to measure the performance of a particular cache configuration without the biases of any particular set of addresses which happen to either benefit or undermine the specific configuration.

7 Results

7.1 Loop Simulation Results

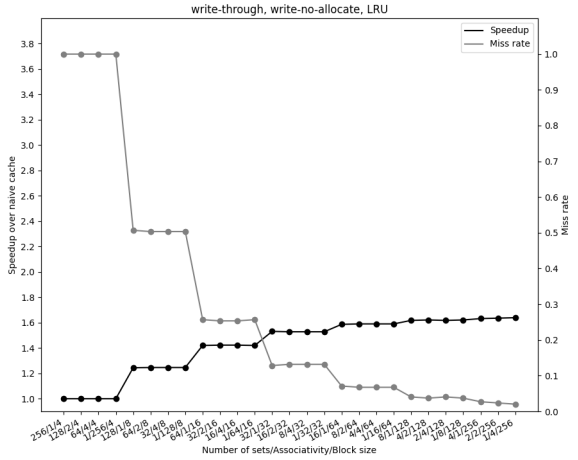

Figure 3. Loop sim with write-through, write-no-allocate, LRU.

The loop simulation has been run with a number of cache configurations. The focus here is on the associativity and block size, not on the total capacity of the cache. Therefore, all tested configurations have the same arbitrary capacity of 1024 bytes (1 KiB). For each block size, 4 configurations are tested: direct mapped, 2-way set associative, 4-way set associative, and fully associative. The speedup of each configuration is calculated by taking the ratio of the number of cycles of execution of a naive configuration which misses 100% of the time, and that of the particular configuration in question. Note that the configuration of the naive cache and specific speedups over its performance are not of particular interest; rather, the focus is on the relative performance between the different non-naive configurations. The results demonstrate that increasing the block size is the best way to extract more performance for this experiment (shown in figures 3, 4, 5, 6). This observation is in line with our intuition about spatial locality and its scaling with block size.

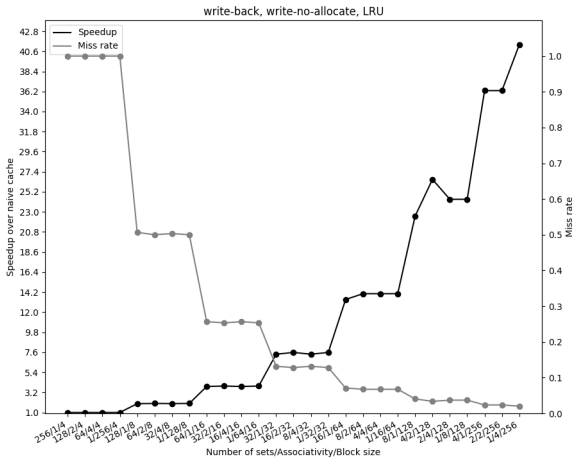

Figure 4. Loop sim with write-back, write-no-allocate, LRU.

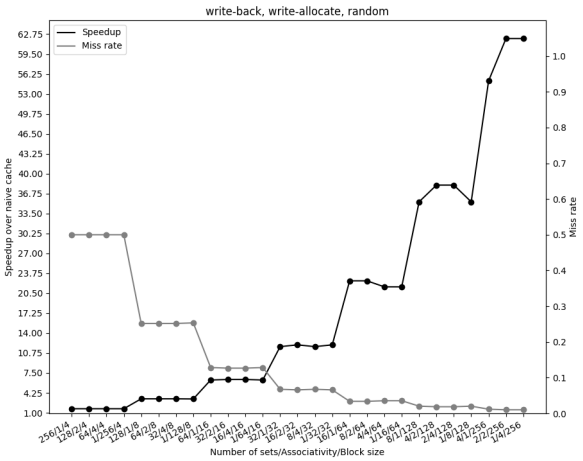

Figure 6. write-back, write-allocate, RR.

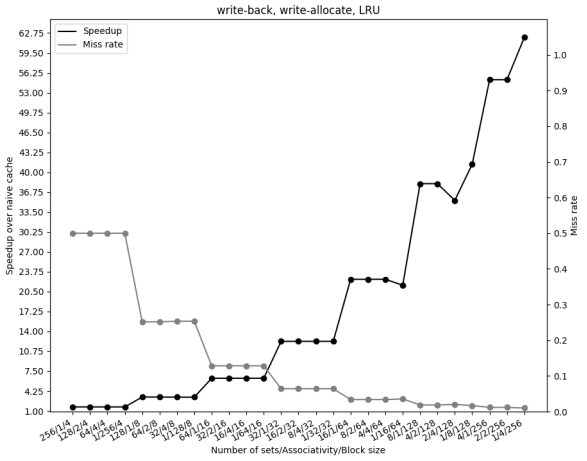

Figure 5. Loop sim with write-back, write-allocate, LRU.

Because the loop simulation exhibits no temporal locality, associativity has no effect on caching performance. Any variations in speedup between different associativities is within margin of error (due to factors like randomized addresses or random replacement). Because there is a store on every loop iteration, the write-through policy performs poorly (figure 3) with at best only a modest 1.6x speedup over the naive cache configuration, even when the miss rate is very low. Under write-through, the miss rate becomes decoupled from write access time because even on hits, a write is made to main memory. Write-back (figure 4) demonstrates a significant improvement over write-through, achieving over a 40x speedup over the naive cache in the best configuration. However, because the write access is first in the loop, every block_size/access_size iterations there is an additional read miss caused by a lack of block allocation on the write miss. For this reason, write-back with write-allocate results in higher performance (figure 5) with the best configuration achieving roughly 60x speedup over the naive design. Data is also recorded for random replacement (figure 6), and as we would expect there is no difference between RR and LRU in this case as all the misses are either compulsory or capacity related, not conflict induced.

7.2 Program Simulation Results

The program simulation involves simulating the code sample shown in section 4. The code exhibits both spatial and temporal locality because some variables are accessed inside loops, while others are accessed multiple times outside of loops. As above, all cache configurations have the same total capacity of 1024 bytes. Speedup is measured against a naive design that misses 100% of the time. Figures 7, 8, 9, 10, and 11 illustrate that the highest performance is achieved by the fully associative design with 64 byte blocks. The 4 way set associative designs also show strong performance. In fact, the optimal design is likely the 4-way set associative configuration with 64 byte blocks because FA caches are costly in hardware resource/complexity, and increased block lookup time (not accounted for here) can be detrimental to performance. Interestingly, the direct mapped designs universally demonstrate poor performance. Because the cache size is much smaller than the memory address space, many

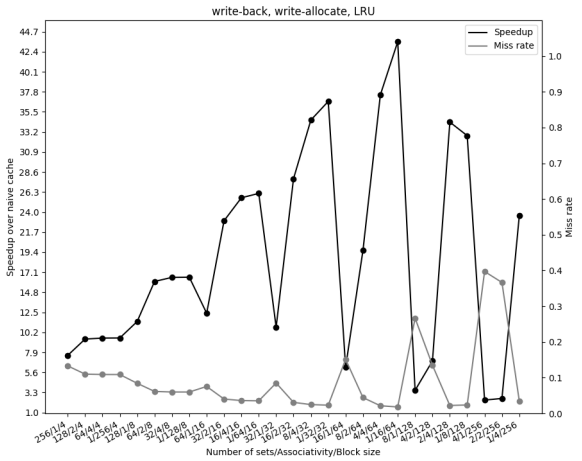

Figure 7. Program sim with write-back, write-allocate, LRU.

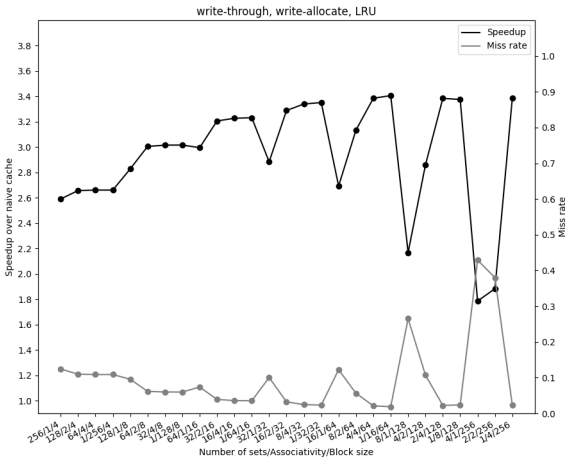

Figure 9. Program sim write-through, write-allocate, LRU.

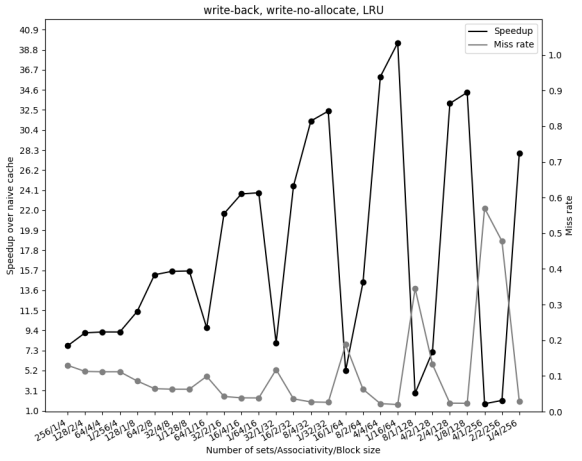

Figure 8. Program sim with write-back, write-no-allocate, LRU.

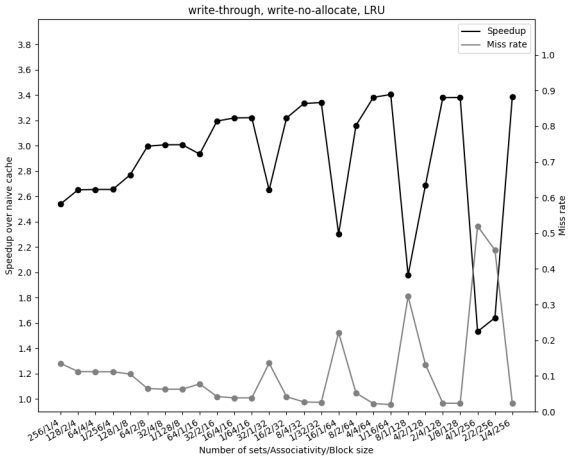

Figure 10. Program sim write-through, write-no-allocate, LRU.

addresses will map to the same block in the DM caches resulting in conflict misses. These misses are reduced dramatically by increasing the associativity to 2-way. Unlike in the above experiment, there is little difference between write-allocate and write-no-allocate (figures 7 and 8) because in most instances of writes, missed blocks are loaded into the cache by a nearby read. On the other hand, write-through leads to a severe degredation of performance as illustrated in figures 9 and 10. Because a large portion of the memory accesses in the program are writes, writing to the memory repeatedly on write hits becomes extremely costly. Figure 11 shows the performance of the same configuration as shown in figure 7 except the replacement policy is RR instead of LRU. While RR results in slightly higher speedup in the optimal configuration, there is overall little performance difference between the two. Figure 12 shows the speedup over a different naive cache that has a miss rate of 95% with respect to the problem size. The result indicates a roughly proportional increase in overall performance as the problem size increases from 0 to 100, confirming earlier observations about the performance

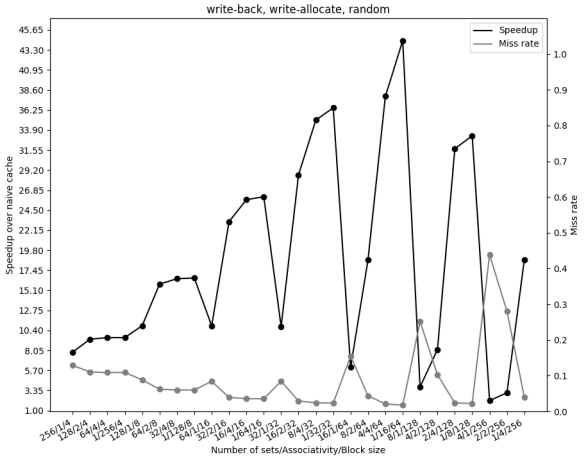

Figure 11. Program sim with write-back, write-allocate, RR.

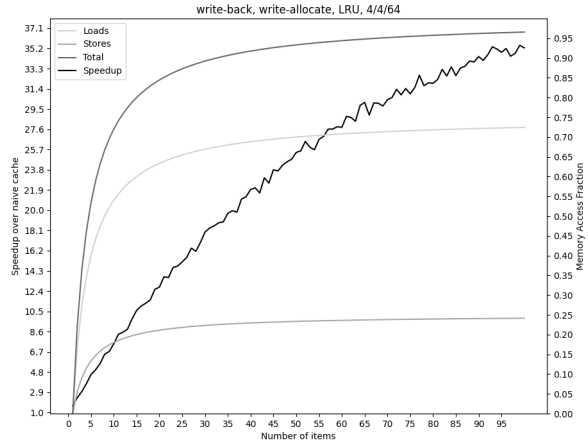

Figure 12. Program sim: overall speedup with respect to problem size.

benefits, based on Amdahl's law, of reducing AMAT for large problem sizes.

7.3 Carbon Savings

A reduction in

References

[1] Gene M. Amdahl. 1967. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference* (Atlantic City, New Jersey) *(AFIPS '67 (Spring))*. Association for Computing Machinery, New York, NY, USA, 483–485. https://doi.org/10.1145/1465482.1465560

[2] Ana Sonia Leon, Brian Langley, and Jinuk Luke Shin. 2006. The UltraSPARC T1 Processor: CMT Reliability. In *IEEE Custom Integrated Circuits Conference 2006*. 555–562. https://doi.org/10.1109/CICC.2006.320989