ECE366 Lab 4: Cache Simulation

For this lab assignment, you will write a configurable cache simulator (in C, Java, or whatever programming language you prefer). Your cache simulator will read an address trace (a chronological list of memory addresses referenced), simulate the cache, generate cache hit and miss data, and calculate the execution time for the executing program. The address trace has been generated by a simulator executing some real programs. Your cache simulator is not the end product of this lab, but a tool you will use to complete it. In this lab, you will experiment with various cache configurations and make conclusions about the optimal cache organization for this set of programs.

The simulator:

The simulator takes the trace file as its input and generates a report as the output. Each trace contains the memory accesses of millions of instructions. Your simulations should be able to process all of them. The report should include:

A. Number of accesses, number of misses and miss rates for all accesses (load and stores)
B. Number of accesses, number of misses and miss rates for loads only
C. Instruction count and execution time for the program, in cycles and seconds
D. Total memory access time and average memory access time (cycles per access).
E. Average CPI

To calculate execution time, assume that:

1- Each instruction takes one cycle to execute.
2- A load takes one cycle plus the memory access time.

Your program should support, at least, the following 5 tunable parameters:

1- Cache size
2- Cache associativity
3- Cache-line size
4- Miss-penalty
5- Cycle time
6- Write policy : either write-allocate policy or write-around policy

If you are doing the extra credit parts, you will also need the following parameters:

1. Replacement policy: either LRU or Randomize

The cache specifications:

The baseline cache configuration will be 16-byte line size, direct-mapped, 16 KB cache size, write-through and write-allocate. Assume a default clock rate of 1 GHz. Memory access time for a load hit is 0 cycles (the same cycle as load being executed) and for a load miss is 40 cycles (unless specified otherwise). The default replacement policy (for set associative caches) is LRU.

The address trace:

An address trace is simply a list of addresses produced by a program running on a processor. These are the addresses resulting from load and store instructions in the code as it is executed. Some address traces would include both instruction fetch addresses and data (load and store) addresses, but you will be simulating only a data cache, so these traces only have data addresses. These traces were generated by a simulator of a RISC processor running five programs, gcc, gzip, mcf, swim, and twolf from the SPEC 2000 benchmarks.
All lines of the address trace are of the format: **l/s ADDRESS IC** where l for a load and s for a store, ADDRESS is an 8-character hexadecimal number, and IC is the number of instructions executed between the previous memory access and this one (not including the load or store instruction itself). There is a single space between each field. The instruction count information will be used to calculate execution time (or at least cycle count). A sample address trace starts out like this:

l 7fffed80 6
l 10010000 0
l 10010003 2
s 10010030 3
l 1001000E 5
l 10010064 3
s 10010034 3

You should assume no accesses address multiple cache lines (e.g., assume all accesses are for 32 bits or less). In the trace shown, the first 24 instructions should take 144 cycles, assuming 3 cache misses and 2 cache hits for the 5 accesses, and a 40-cycle miss penalty (assuming baseline configuration).

Cache configuration experiment:

You will re-evaluate the default parameters one at a time, in the following order. In each configuration, choose a best value for each parameter, then use that for all subsequent analyses. Because the workload is five programs (trace files), you will run five simulations for each configuration you evaluate, and then combine the results, using average execution time as your evaluation metric.

A. Look at cache line (block) sizes of 16, 32, and 64 bytes. Assume that it takes *two extra cycles* to load 32 bytes into the cache, and *6 extra cycles* to load 64 bytes. (i.e., raise the miss penalty accordingly). Choose the best cache line size (in terms of miss penalty) and proceed.
B. Look at 16 KB, 32 KB, and 128 KB cache sizes. Larger caches take longer to access, so assume that the 32 KB cache requires a *5% longer cycle time*, and the 128 KB cache *15% longer*. Choose the best configuration and proceed to the next step.
C. Look at cache associativity of direct-mapped, 2-way set-associative, and 8-way set-associative. Assume that 2-way associative adds *5% to the cycle time*, and 8-way *adds 10%.* Choose the best configuration and proceed.
D. Compare the write allocate policy with a write-around policy (see notes 1,2 and 3).
E. **(extra credit)**. Compare the LRU replacement policy with randomize policy.

Lab 4A demo:

In the first week of the lab, you will demo the first two sets of configurations and find the optimal cache line size and cache size. In both cases, the cache is a direct-mapped cache.

Lab 4B demo:

In the second week of this lab, you will demo the effect of cache associativity on execution time.

Lab 4C demo:

In the last week of this lab, you will demo the effect of write policy on execution time. If you also implement the replacement policy, you will show your work in this demo.

Submit a lab report with the following:

(a) Show your results and design decisions for each of the four configuration parameters above. Provide data / results in meaningful charts or figures.

(b) Is cache miss rate a good indicator of performance? In what cases did the option with the lowest miss rate not have the lowest execution time? Why?

(c) Were results uniform across the five programs? In what cases did different programs give different conclusions? Speculate as to why that may have been true.

(d) What was the speedup of your final design over the default?

(e) Summarize the data structure and algorithm you used in the simulator.

(f) Turn in your source code file with sample output.

Note & Hints:

1. Details about write-allocate cache: A write-allocate cache, on a write miss, will make a spot for the line in the cache, load it from memory, and it becomes available to read in "miss penalty" cycles later. If the processor accesses it (with a load) before that, it stalls until the data is ready. So for a write-allocate cache, you can get a store miss at time T, followed by a load hit at time *T+7*. However, the load still must stall until the data becomes available at time *T+ miss_penalty* (for bookkeeping, it is still a load hit, but for timing, you must record the stall), so the processor will see a miss penalty on this cache *hit* of (miss_penalty – 7) cycles. Thus, you never stall for a store, but you may stall on a later load to the same cache line (that is, we're assuming a large store buffer). You need not stall (more than the original miss penalty) on a miss that evicts a line in transit (still waiting for the store to complete) – we'll assume that the first transfer can complete, and the original store complete, while the new data is being brought from memory.
2. A store instruction can hit or miss in the cache (and thus change the contents of the cache for a write-allocate cache), but does not incur a miss penalty, since no data is returned to the CPU.
3. In write-around policy, the processor still does not stall for stores, and a store miss does not change the contents of the cache.
4. Think about how to intelligently debug and test your program. Running immediately on the entire input gives you little insight on whether it is working (unless it is way off).
5. Speed matters. These simulations should take a couple minutes (actually, less) on an unloaded machine. If it is taking much more than that, do yourself a favor and think about what you are doing inefficiently.

Cache Simulation

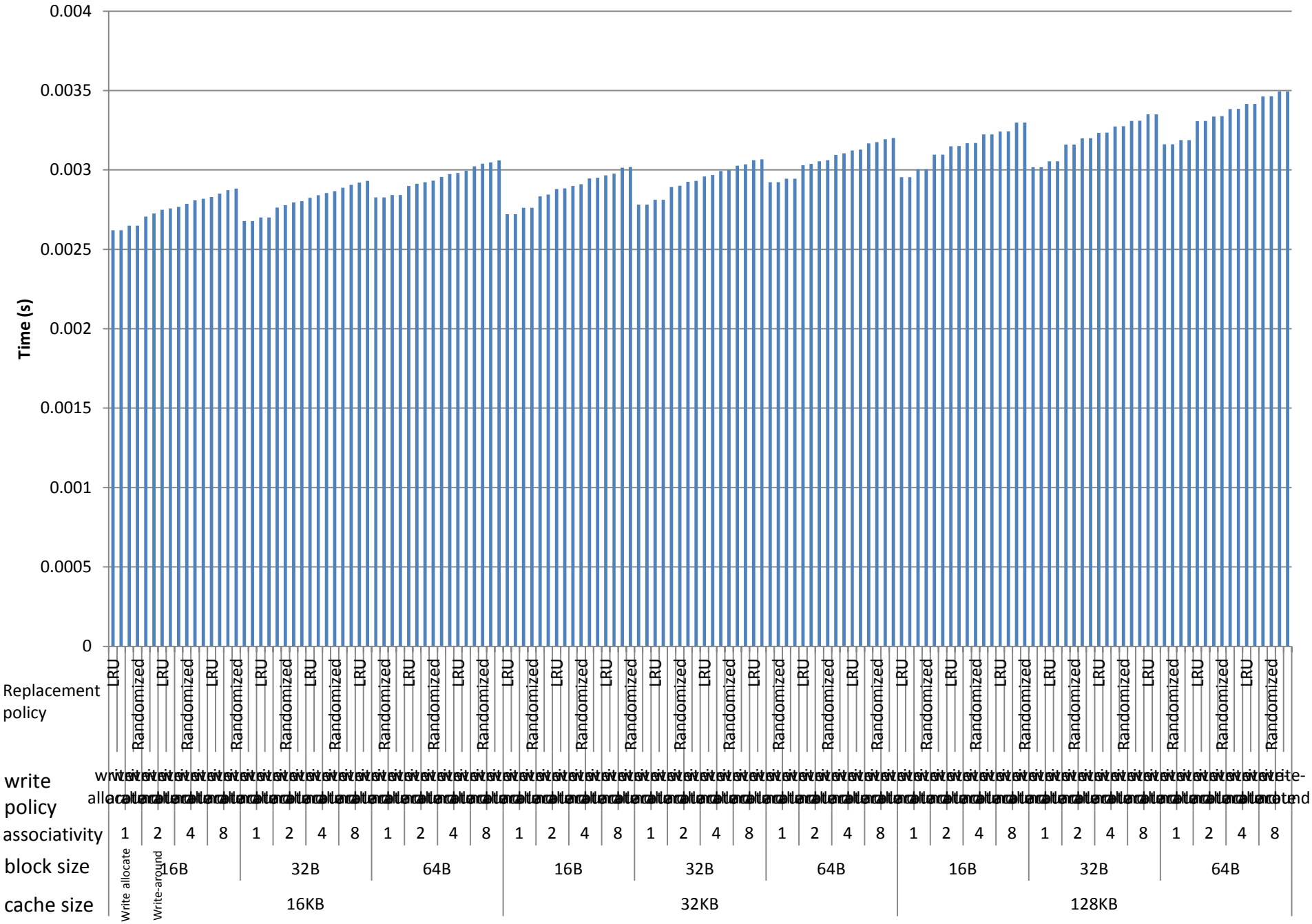
Kai Zhao

ECE 366

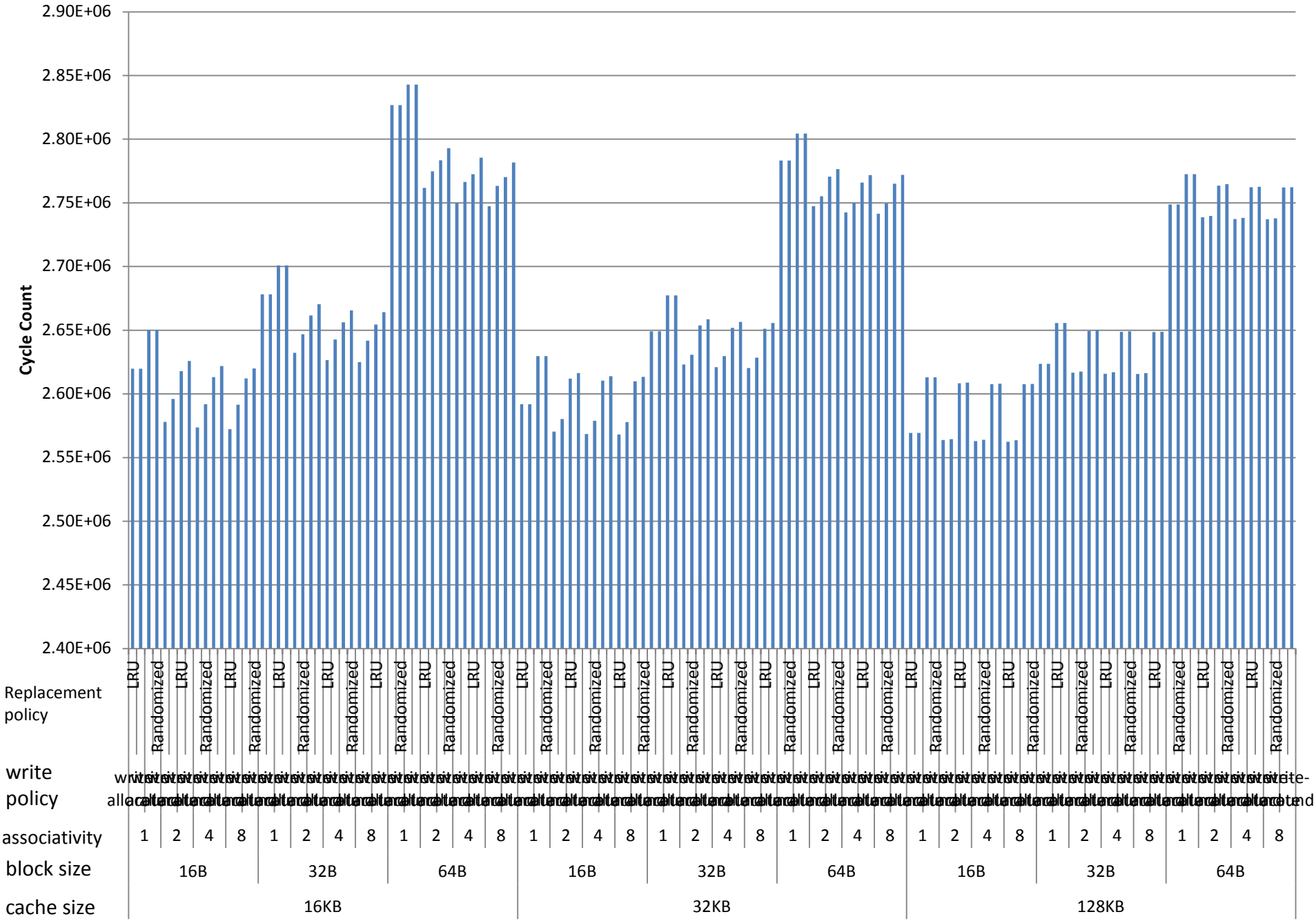
Lab 04

2013 Dec 1

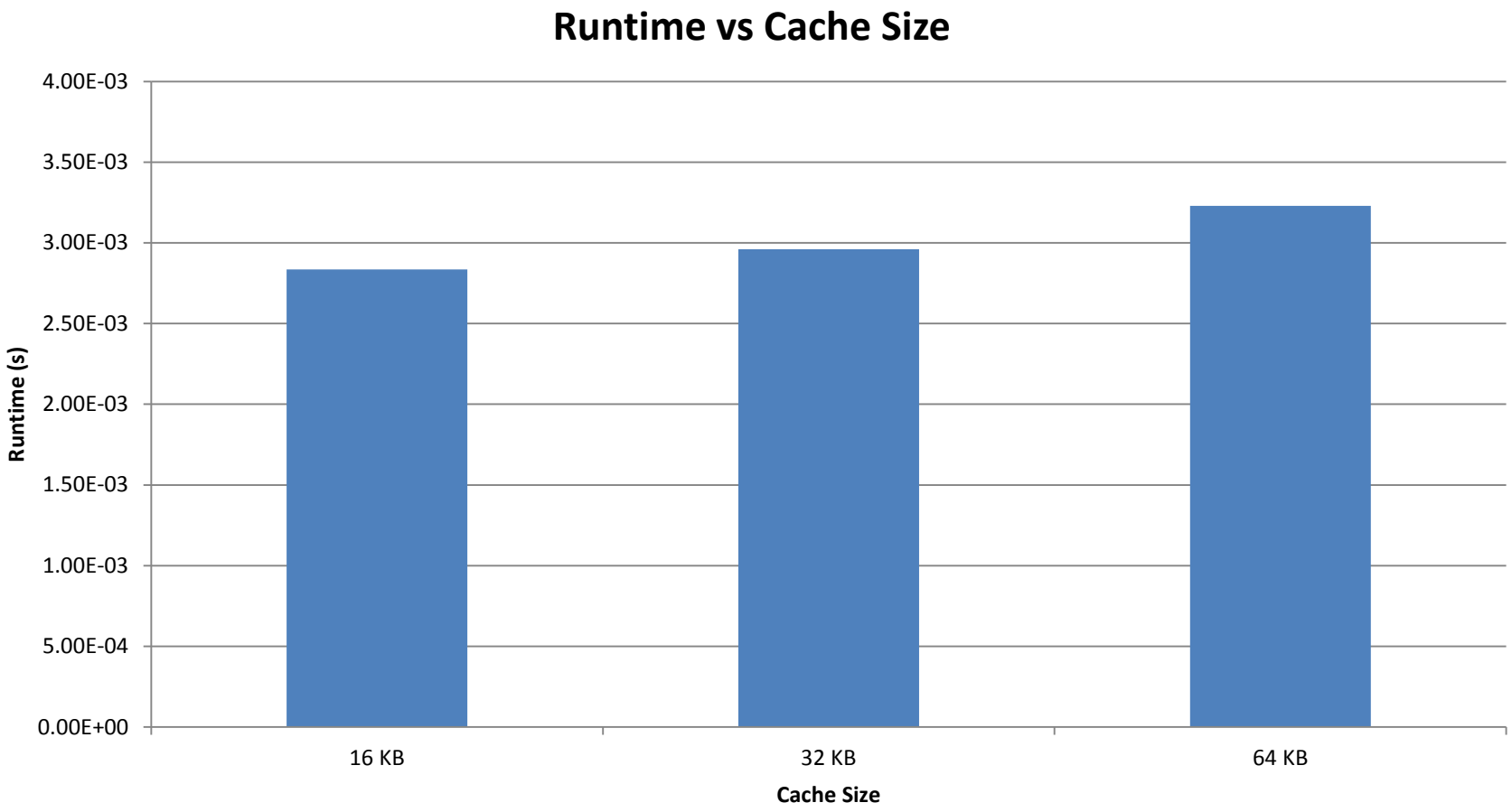
Runtime Average of All 5 Trace Files



Cycle Count Average of All 5 Trace Files

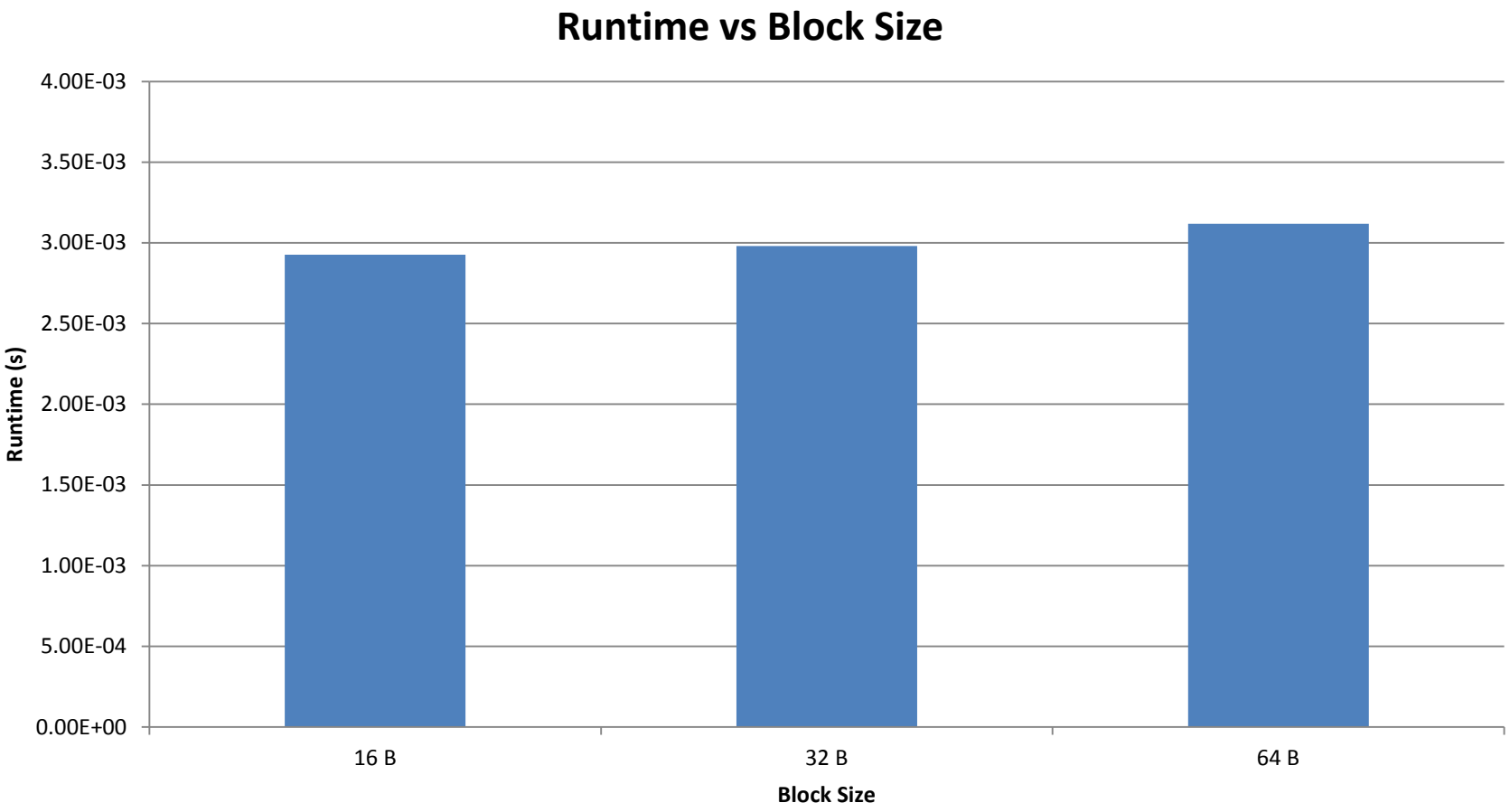


16 KB is Fastest Cache Size



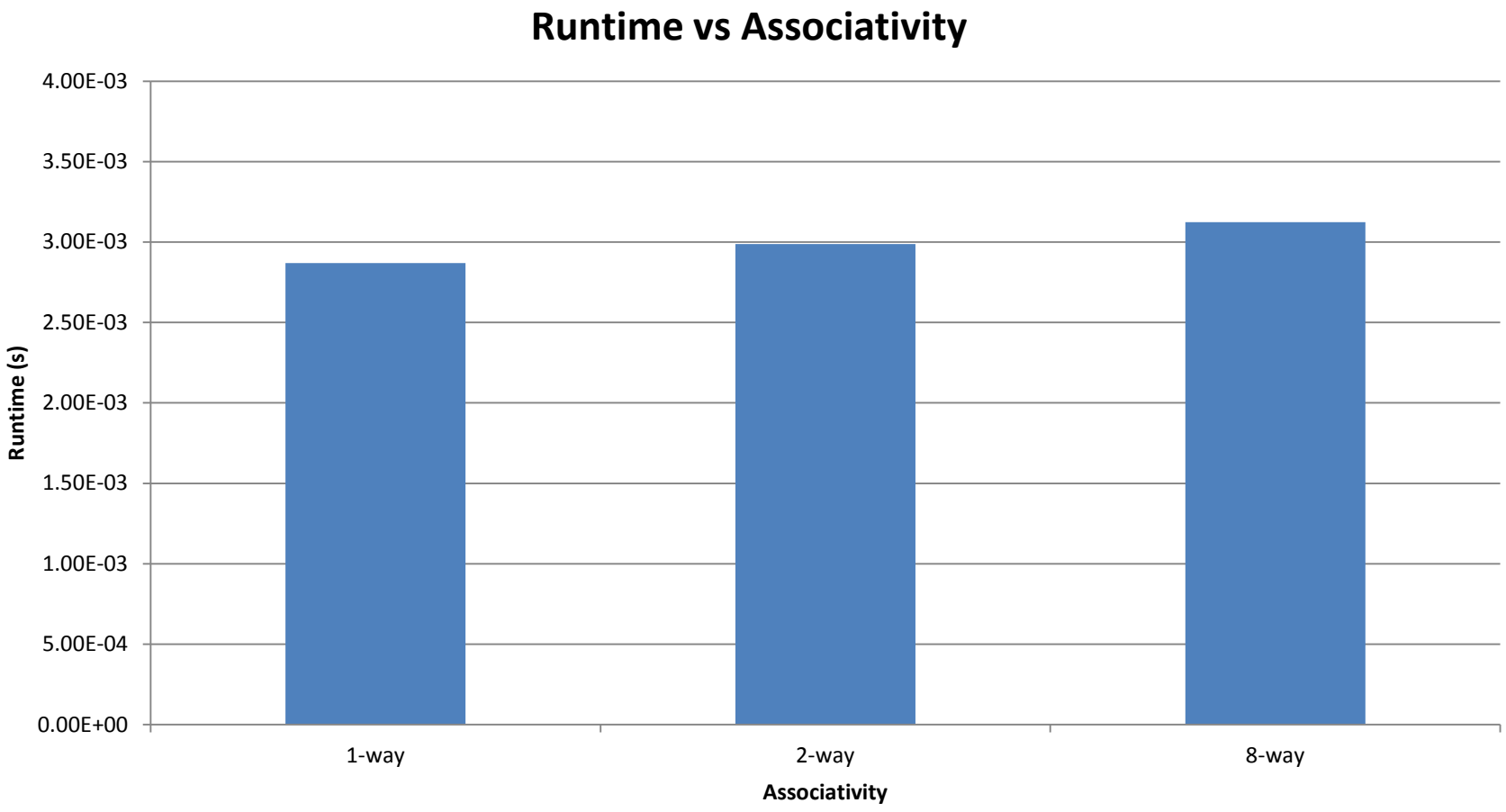
Results are the average of all configurations with the cache size

16 Byte is Fastest Block Size



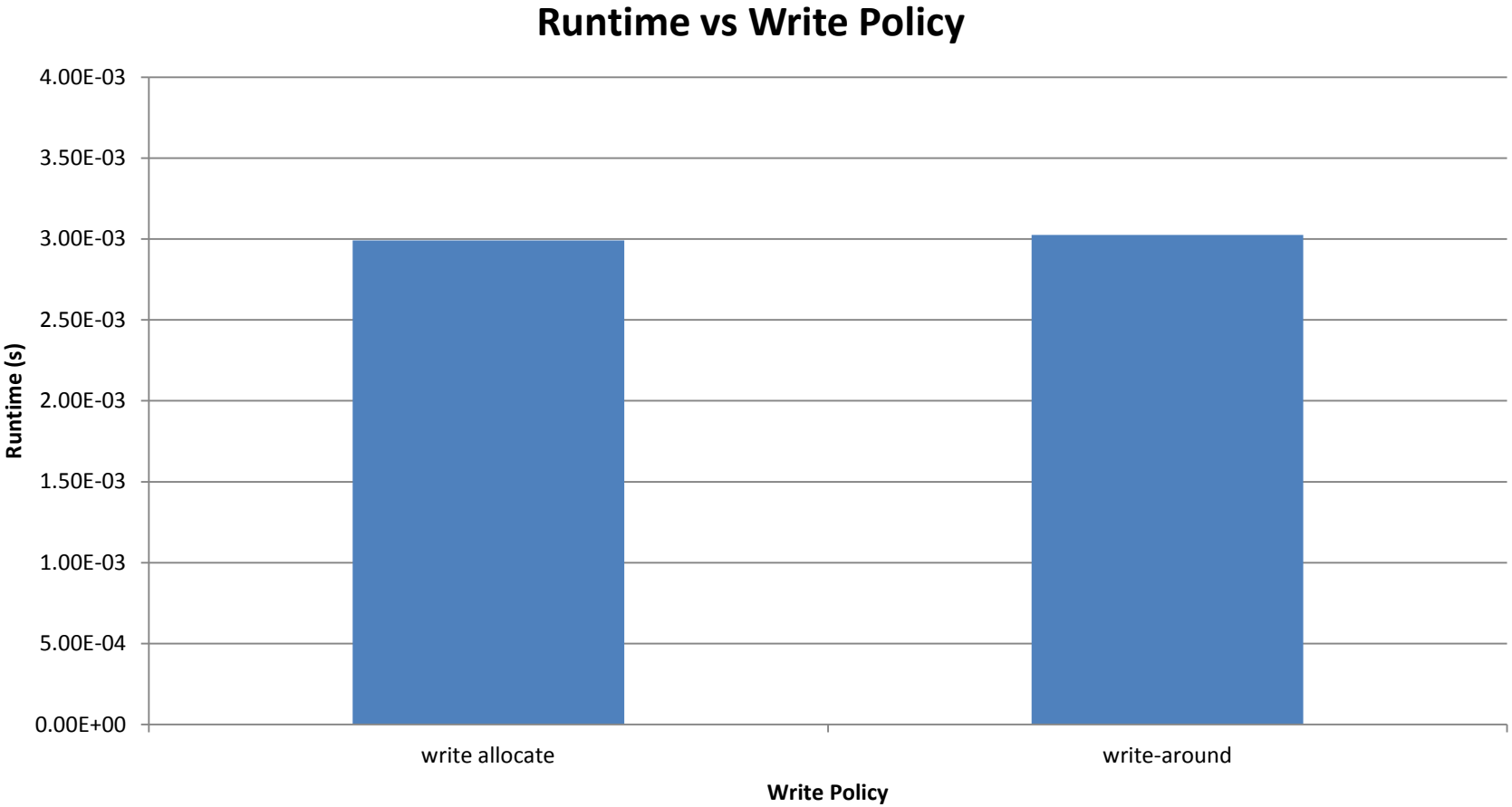
Results are the average of all configurations with the block size

Direct-mapped (1-way) cache is fastest



Results are the average of all configurations with the associativity

Write Allocate is Slightly Faster

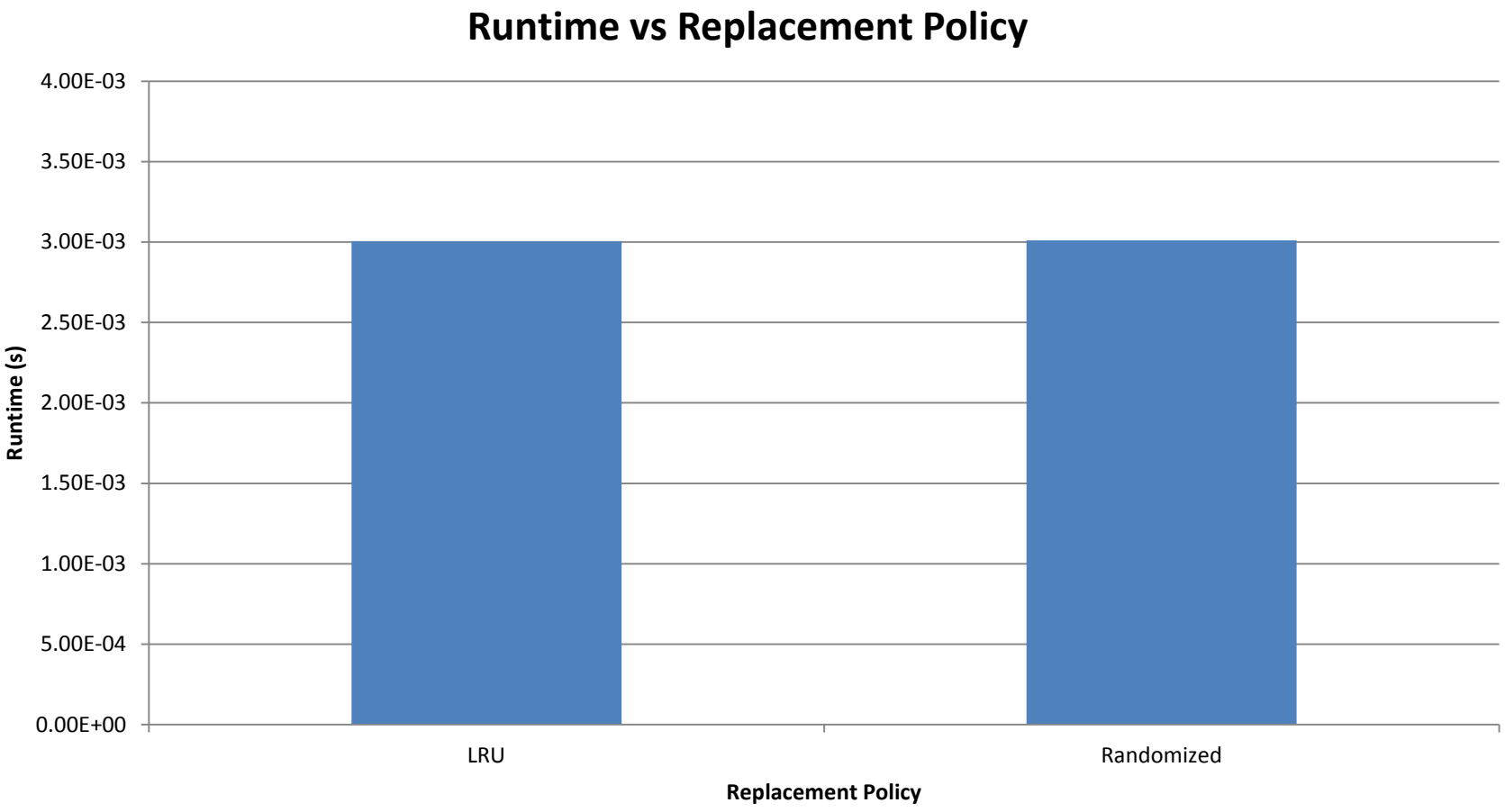


Results are the average of all configurations with the write policy

Assumption: data can be read from the write buffer. For write-around with loads after a store, instead of two memory access penalties (one to write to memory, another to read from memory), there will only be one memory access penalty to load the remaining of the block.

Assumption summary: no penalty and no FIFO for writing to memory

LRU is Infinitesimal Faster



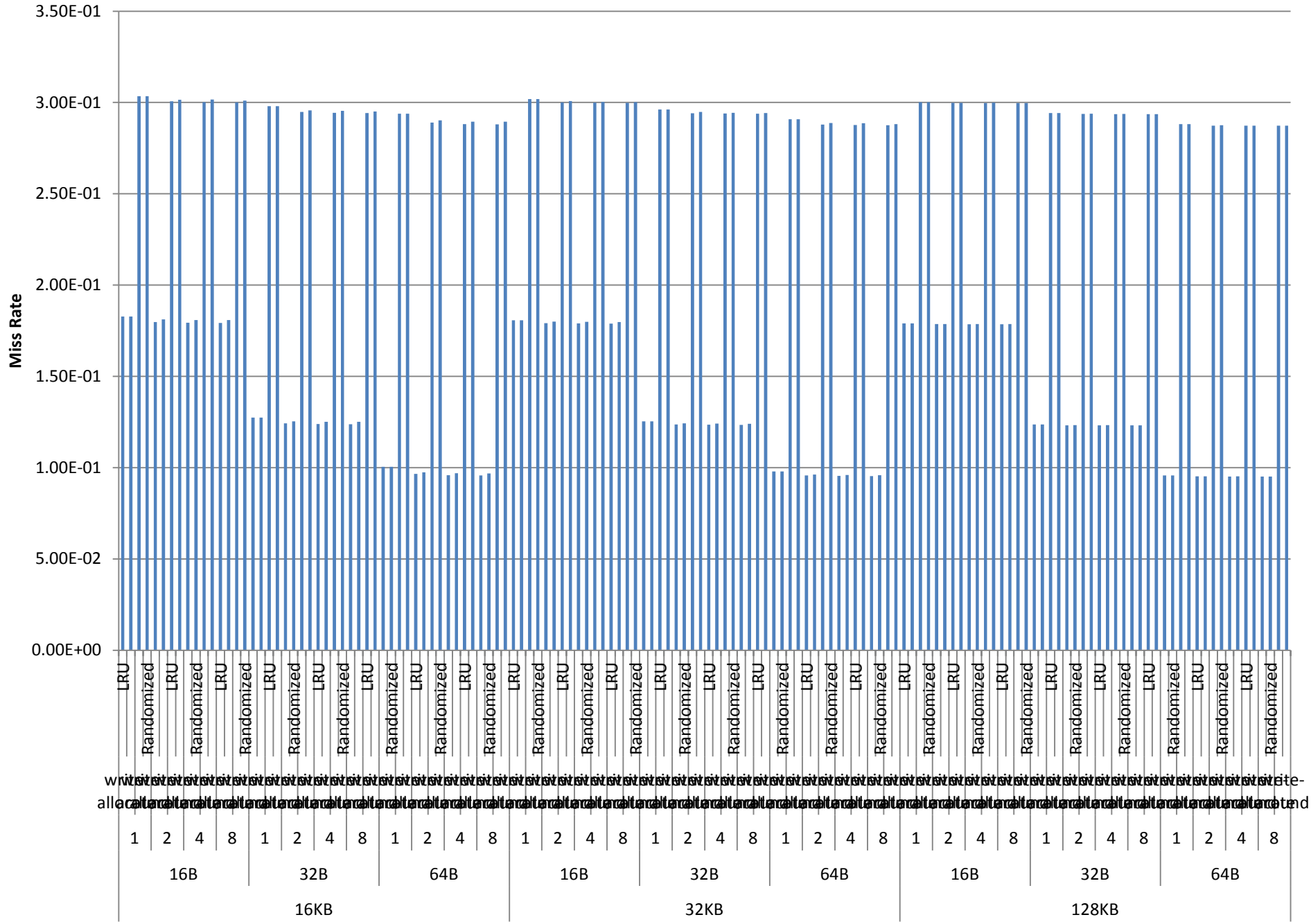
Results are the average of all configurations with the replacement policy

Part A

Show your results and design decisions for each of the four configuration parameters above.

- In order of importance
  - 16 KB is the fastest cache size
  - Direct-mapped cache is the fastest associativity
  - 16 Byte is the fastest block size
  - Write allocate is faster than write-around
  - LRU is faster than randomized replacement policy

Miss Rate Average of All 5 Trace Files

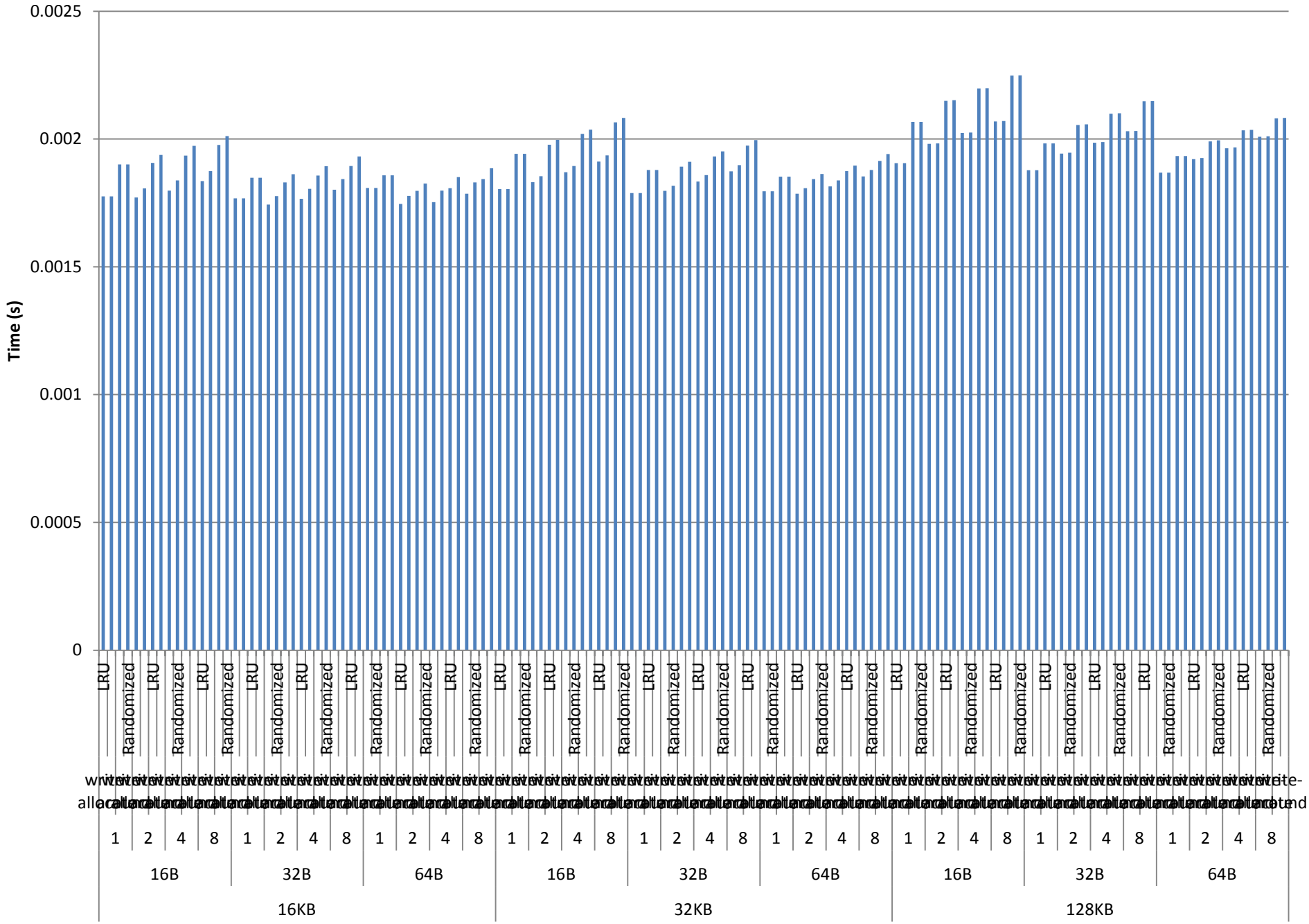


Part B

Is cache miss rate a good indicator of performance? In what cases did the option with the lowest miss rate not have the lowest execution time? Why?

- Cache miss rate is not a good indicator of performance
  - Although 64 Byte block sizes have the lowest miss rate, they do have the longest execution time.
  - The extra miss penalty cycles is too expensive relatively to the lower miss rates

Runtime of gcc.trace



Part C

Were results uniform across the five programs? In what cases did different programs give different conclusions?

- Most programs agree that the default configuration is fastest
  - 16 KB cache size, 16 B block size, 1-way associativity, write allocation, and LRU
- For gcc.trace, there is another configuration that is 1.65% faster than the default configuration
  - Due to much lower miss rates compared to the default

Part D

What was the speedup of your final design over the default?

- 0% because my final design is the default

Part E Data Structures

Summarize the data structure and algorithm you used in the simulator.

- Configurations: traceFile, cacheSize, blockSize, associativity, writePolicy, and replacementPolicy
- 2D array of integers: the first dimension is number if indices, the second dimension is number of ways
  - int[][] cache; // used to store the tag of every block
  - int[][] validBit; // initialized as 0's to keep determine if the data in cache is valid
  - int[][] timeWhenReady; // used for write allocate to determine when the cache will be ready when there is a load hit after a store miss
  - int[][] lastUsedTime; // updated when reference to implement LRU policy
- Counters for cache simulation output: clockCycleNumber, instuctionCount, numberOfLoadHits, numberOfLoadMisses, numberOfStoreHits, numberOfStoreMisses

Part E Algorithms

Summarize the data structure and algorithm you used in the simulator.

- Search for cache hit:
  - Find the index and transverse each way checking if the validBit > 0 and tag matches
- Stall if load after store miss:
  - if clockCycleNumber + numberOfInstructionsInbetween < timeWhenReady
    clockCycleNumber = timeWhenReady
    else
    clockCycleNumber += numberOfInstructionsInbetween + 1
- Write allocate:
  - Update cache and time timeWhenReady on store misses
- LRU:
  - Loop through each way to check if any validBit == 0. If not, then find the cache way with the lowest lastUsedTime to replace