L1, L2, DRAM Cache Simulation

Adapted from F21 Computer Architecture: ***Final Project Report***

Gregory Giovannini

Technical Paper Advisor: Prof. Bo Yuan

***Abstract*— The L1-D2-DRAM memory hierarchy system remains widely used among modern CPUs. This investigation seeks to develop a L1-L2 cache simulator in the C programming language to study the effects of cache size, associativity, and prefetching on the performance of the individual L1 and L2 caches, as well as the overall cache miss rate. It was found that miss rate generally decreases with increasing cache size and with increasing associativity. Prefetching generally decreases the miss rate of the cache, though the effect is more prominent in benchmarks that exhibit a high amount of spatial locality.**

Keywords*— *cache, L1, L2, memory hierarchy, miss rate, associativity, prefetching, C

I. Introduction

A L1-L2-DRAM memory hierarchy system remains a frequently-used model for modern CPUs. In this investigation, a simple L1-L2 cache simulator is developed in the C programming language. C was selected for its low-level access to memory and fast runtime.

The goal of this project is not to simulate memory itself, but rather, to simulate the behavior of the cache hierarchy to demonstrate how various configurations of cache size, associativity, and block size, along with prefetching adjacent memory addresses, impact the performance of a multilevel cache among different benchmarks.

On a high level, the simulator should be able to take as input a memory access trace, which must include a memory address and whether the access was a read or write. From there, the simulator will interpret each instruction and calculate the number of cache misses and hits for an L1-L2 cache hierarchy.

Thus, the general plan of development includes creating two adjacent caches (L1 and L2), each customizable in terms of size, associativity, block size, and replacement policy (for the sake of scale, however, only a least-recently-used policy will be implemented), along with the optional ability to prefetch an adjacent address block. The caches need to be properly indexable using the bits of the address, and each address must be correctly parsed from every line of a given memory trace. Moreover, memory traces must be generated to use as benchmarks, so another aspect of development will be using a memory analysis tool along with some scripting to automate trace generation. Finally, the cache must then be tested for both correctness and investigation of the effects of cache size, associativity, and prefetching on L1 and L2 caches.

II. Technical Background

As stated previously, the cache simulator will model an L1-L2 cache. This means that, when a memory access is attempted, the L1 cache will first be checked to see if the requested address is in it. If it misses, the L2 cache will then be checked; if it misses as well, then the address will be pulled from memory (see Fig. 1). Generally, the L1 cache is very small and focuses on fast access time for relatively lower hit rate; the L2 cache, because of its larger size, allows for a higher hit rate at the cost of slower access time [1].

Additionally, the cache replacement policy to be implemented for both caches is least-recently-used (LRU). In an LRU policy, when an address is being pulled into a full cache, the entry that gets evicted is that which has been referenced the least recently [2]. In practice, this can be implemented in a number of ways; the simplest in terms of simulation is for each cache entry to keep track of its usage as a counter that increments every time it is not used. At eviction time, the entry with the highest usage counter will then be evicted. It should be noted that this method is not the most efficient, nor does it account for possible overflow in very large caches.

Prefetching is a technique that can exploit the spatial locality of a cache by bringing data in before it is requested via an access. With prefetching, on a cache miss, both the accessed block and another adjacent block of memory can be brought into the cache; of course, this results in an extra memory read, but the idea is that it (hopefully) saves more memory reads later on [3]. As an example, given memory address 0x30 that experiences a cache miss and a block size of 8 bytes, a prefetch would bring both the 0x30 and 0x30 + 8 blocks into the cache. Note that prefetching only occurs on cache misses, and if the prefetched block is already in the cache, no extra memory read occurs. Additionally, for the sake of simplicity, it is assumed in this investigation that prefetching only applies to the level of the cache closest to memory (i.e., the L2 cache).

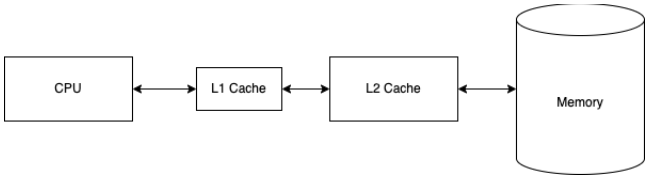

Fig. 1: L1-L2-DRAM Memory Hierarchy.

III. System Setup & Implementation

A. System Setup

For this investigation, development and testing has taken place on Ubuntu 20.04.2 LTS, running on the Rutgers University iLab machines. The cache simulator is written in the C programming language, compiled using the GNU Compiler Collection (gcc). For generation of memory access traces, Python 3.7+ and Valgrind are required; Valgrind analyzes a program's memory usage, and Python is used for scraping the output from Valgrind and converting it to a memory trace for the simulator.

For information on how to build and run the simulator, please see Section V below.

B. Implementation

The cache simulator consists of two main source files, "cache-sim.c" and "cache-sim.h". The former contains the majority of the logic and source code for the simulation; the latter outlines some key data structures for the caches, including a Line (block), a Set, which contains a hash table of Lines, and a Cache, which contains a hash table of Sets.

In "cache-sim.c", the L1 and L2 caches are defined separately, each with their own size, associativity, etc. Upon invoking the cache simulator, both the L1 and L2 caches can be customized with a size, associativity, replacement policy, and block size. However, for this investigation, only an LRU eviction policy is implemented.

The simulator also accepts a path to a memory access trace file (.txt), with each line formatted as follows:

for a write: "0xnnnnnnn: W 0xmmmmmmm";

for a read: "0xnnnnnnn: R 0xmmmmmmm";

where the first item (preceding the colon) is the PC address at which the given memory access occurred; the second letter (W or R) indicates a write or read, respectively; and the third item is the memory address that was accessed. Here, the addresses are hexadecimal strings. For the sake of this investigation, it is assumed that addresses are 48 bits; any addresses that are shorter will be zero-extended automatically. The final line of these files should be "#eof".

For this simulation, the first item in the memory access traces (PC address) is ignored, as the order of accesses is taken to be the order of the trace itself, making the PC address unnecessary for this experiment.

As the simulator reads in each access from the memory trace, it simultaneously updates a pair of L1 and L2 caches without prefetching and another with prefetching. Prefetching occurs according to the conditions described in Section II.

Bit masking is used to separate the set and tag bits for each address; these bits are then used to hash into the respective Set and Line of the cache during a fetch.

The following logic is applied to each memory request from the trace file, omitting details:

- If the access is a read (R):
    - Check the L1 cache; on miss, check the L2 cache; on miss, increment memory reads, and if prefetching, prefetch into the L2 cache.
- If the access is a write (W):
    - Check the L1 cache; on miss, check the L2 cache; on miss, increment memory reads, and if prefetching, prefetch into the L2 cache. Regardless of outcome, increment memory writes.

Four test memory access traces ("test1.txt" – "test4.txt") were adapted from previous work on cache simulation and are used in the evaluation of this cache. In order to be able to generate useful benchmarks for the cache simulator, a Python script called "mem_trace.py" has been written to create and format memory access traces using Valgrind to obtain real addresses for load and store instructions. These trace files can then be used with the cache simulator similarly to the existing traces in the "tests" directory. For the purposes of evaluation, this script was run on a few simple programs to generate memory traces, including programs for matrix multiplication, calculating digits of pi, and looping repeatedly.

The cache simulator outputs various analytics about the performance of the cache, including total number of instructions, number of cache hits and misses for L1 and L2, miss rate for L1 and L2, and a similar breakdown for the caches in prefetch mode. Miss rate is defined as the ratio of cache misses to the total number of instructions.

Please note that, for the sake of brevity, some other implementation details are being omitted from this paper, as they are either trivial, implied based on common cache knowledge, or too verbose to warrant elaboration.

IV. Evaluation Results

A. Correctness

The behavior of the cache simulator was verified for correctness using small, readable subsets of the given stack traces, along with dedicated debugging functions such as "printCache" to visualize the caches at a given stage of the program. Additionally, various error reporting mechanisms were created to assist with usage and explicitly denote the cause of any input-based errors. Correctness was achieved for all given tests.

B. Evaluation of L1 Cache

The first set of experiments to investigate with the cache simulator seeks to establish a relationship between the L1 cache size and the miss rate of the L1 cache. Therefore, the L1 cache size is varied between 32 to 512 bytes, in multiples of 2. Results of these experiments are reported in Table 1 and graphed in Fig. 2. The commands used to generate these outputs can be found in #3 and #4 of Section VII. For this and all other experiments, it is assumed that the L1 cache is direct-mapped and the L2 cache is associative. The L1 cache block size is fixed at 4 bytes; the L2 cache is fixed at 1 KB, with 8-byte blocks.

As seen in Table 1, the "test2.txt" benchmark does not provide enough instructions to obtain a significant correlation between cache size and miss rate. The experiment is repeated using "test3.txt", which shows a much stronger correlation. As the L1 cache size increases from 32 bytes to 512 bytes, the L1 miss rate decreases from 0.849 to 0.456. This is expected, as a larger cache should experience fewer cache misses.

C. Evaluation of L2 Cache

The second set of experiments seeks to establish a relationship between the L2 cache size and the miss rate of the L2 cache, along with a separate relationship between the associativity and miss rate. Therefore, the L2 cache size is varied between 1 to 8 kilobytes, in multiples of 2, and the associativity of the L2 cache is varied between 1-way to 8-way in multiples of 2. Results of these experiments are reported in Tables 2 and 3; Table 3 is shown graphically in Fig. 3. The commands used to generate these outputs can be found in #5 and #6 of Section VII. The L1 cache is fixed at 32 bytes, with 4-byte blocks; the L2 cache block size is fixed at 8 bytes.

As Table 2 shows, varying the associativity and cache size of the L2 cache has almost no effect on the miss rate of the L2 cache for the "test2.txt" benchmark, likely because of this memory trace's brevity. "test3.txt" provides a more useful picture in Table 3: miss rate decreases with increasing associativity and with increasing cache size. For the same reason as the L1 cache, the size result is expected. The associativity result makes sense considering that higher associativity increases the number of blocks that can be mapped to in a set, potentially increasing the efficiency of the cache.

D. Evaluation of Overall Cache Behavior

The final experiment with the cache simulator tests the effect of prefetching versus no prefetching on L1, L2, and overall cache miss rate, across all benchmarks. Results are reported in Table 4 and graphed in Fig. 4, and the commands used to generate these outputs can be found in #7 of Section VII. For this experiment, L1 is fixed at 32 bytes, with 4-byte blocks; L2 is fixed at 4 KB and 4-way associativity, with 8-byte blocks.

Table 4 seems to display a lower miss rate for the L2 and overall cache when prefetching versus when not using prefetching. For instance, "test4.txt" yields a L2 miss rate of 0.329 and overall miss rate of 0.281 with no prefetching, compared to a L2 miss rate of 0.260 and overall miss rate of 0.222 with prefetching. For this implementation, prefetching is only implemented for the L2 cache, so it makes sense that prefetching has no result on the L1 miss rate.

L1 Cache Size (Bytes)	L1 Miss Rate ("test2.txt")	L1 Miss Rate ("test3.txt")
32	0.350	0.849
64	0.339	0.743
128	0.333	0.640
256	0.330	0.529
512	0.329	0.456

Table 1: Evaluation of L1 Cache.

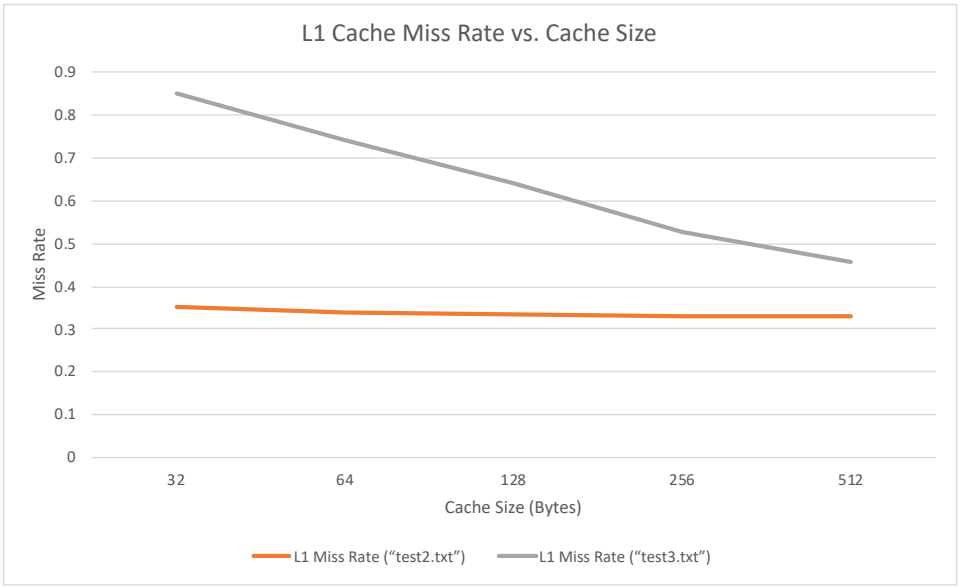

Fig. 2: L1 Cache Miss Rate vs. Cache Size.

L2 Cache Size	1-way	2-way	4-way	8-way
1 KB	0.369	0.334	0.329	0.328
2 KB	0.340	0.314	0.312	0.309
4 KB	0.319	0.304	0.302	0.301
8 KB	0.301	0.295	0.295	0.294

Table 3: Evaluation (Miss Rate) of L2 Cache for "test3.txt" Benchmark.

L2 Cache Size	1-way	2-way	4-way	8-way
1 KB	0.471	0.469	0.469	0.469
2 KB	0.469	0.469	0.469	0.469
4 KB	0.469	0.469	0.469	0.469
8 KB	0.469	0.469	0.469	0.469

Table 2: Evaluation (Miss Rate) of L2 Cache for "test2.txt" Benchmark.

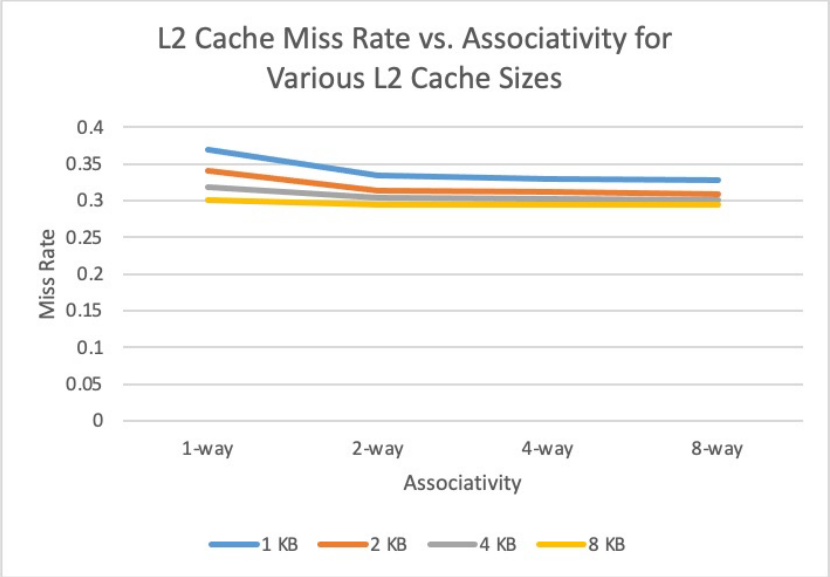

Fig. 3: L2 Cache Miss Rate vs. Associativity for Various L2 Cache Sizes.

For a benchmark like "looping_test.txt", which is just incrementing a counter sequentially, there exists a significant amount of spatial locality. Therefore, it is expected for the miss rate to be much smaller when prefetching versus when not. The results confirm this with a 0.262 L2 miss rate when prefetching versus 0.427 when not.

Also, it is worth noting that, with a few exceptions among the benchmarks, the L2 cache experiences a lower miss rate than the L1 cache for both prefetching and no prefetching. This is reasonable considering how much larger the L2 cache is than the L1. The exceptions are likely due to the structure of the programs that produced the nonconforming benchmarks, with more blocks mapping to the L2 cache.

V. Source Code & Usage

All source code and benchmarks for this project can be found on GitHub at:

https://github.com/greg300/ECE563_Final_Project

To build the cache simulator, please first ensure all requirements listed in Section III Part A are met, then see #1 in Section VII.

The usage for running the cache simulator is enumerated below, along with all options / requirements for each parameter:

Benchmark	No Prefetching L1 Miss Rate	Prefetching L1 Miss Rate	No Prefetching L2 Miss Rate	Prefetching L2 Miss Rate	No Prefetching Overall Miss Rate	Prefetching Overall Miss Rate
test1.txt	0.336	0.336	0.500	0.500	0.168	0.168
test2.txt	0.350	0.350	0.469	0.468	0.164	0.164
test3.txt	0.849	0.849	0.302	0.234	0.256	0.198
test4.txt	0.853	0.853	0.329	0.260	0.281	0.222
matrix_mult_test.txt	0.496	0.496	0.423	0.260	0.210	0.129
pi_test.txt	0.027	0.027	0.154	0.090	0.004	0.002
looping_test.txt	0.416	0.416	0.427	0.262	0.178	0.109

Table 4: Evaluation of Overall Cache Performance With and Without Prefetching.

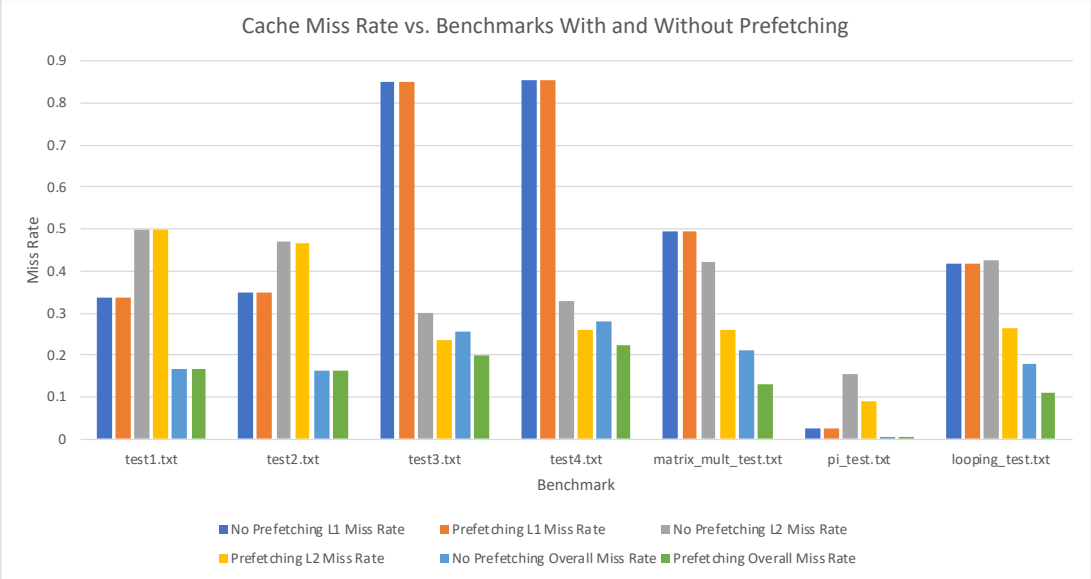

Fig. 4: Cache Miss Rate vs. Benchmarks With and Without Prefetching.

./bin/cache-sim l1_cache_size l1_assoc l1_replace_policy l1_block_size l2_cache_size l2_assoc l2_replace_policy l2_block_size trace_file

l1_cache_size: int - size of L1 cache in bytes; must be a power of 2

l1_assoc: str - associativity of L1 cache; can be one of:

direct - direct mapped cache

assoc - fully associative cache

assoc:n - n-way associative cache, where n is a power of 2

l1_replace_policy: str - L1 cache replacement policy (lru only is supported)

l1_block_size: int - size of L1 cache block in bytes; must be a power of 2

l2_cache_size: int - size of L2 cache in bytes; must be a power of 2

l2_assoc: str - associativity of L2 cache; can be one of:

direct - direct mapped cache

assoc - fully associative cache

assoc:n - n-way associative cache, where n is a power of 2

l2_replace_policy: str - L2 cache replacement policy (lru only is supported

l2_block_size: int - size of L2 cache block in bytes; must be a power of 2

trace_file: str - path to trace file used as input to the simulator

To create a memory access trace file using the created Python script and Valgrind, see #2 in Section VII.

All commands for reproducing the evaluation results in Section IV can be found in #3-7 of Section VII below.

VI. Conclusion & Future Work

By creating a L1-L2 cache simulator and evaluating the performance on memory traces obtained from various benchmarks, correlations and relationships were shown based on cache size, associativity, and the use of prefetching on cache miss rate. Increasing the L1 cache size decreases the miss rate. Likewise, increasing the associativity and size of the L2 cache decreases the miss rate. Prefetching also can reduce the miss rate of the overall cache, with a more noticeable effect on benchmarks with higher spatial locality.

Future work to be done on this project includes adding support for more cache eviction policies, such as FIFO (first in, first out), random, or adaptive (the latter of which has been studied and proven to outperform LRU [4]), as well as to explore more potential evaluations with varying block size or L1 cache replacement policy.

VII. Addendum: Commands

The following is a list of most commands used in the production of this report, consolidated here to declutter the remainder of the paper.

1. Build the cache simulator:

   make

2. Create a memory access trace file from a compiled binary (requires Valgrind):

   python mem_trace.py <prog_name>

   (Note: may need to use "python3" if "python" references Python 2, e.g. on some macOS installations.)

3. Run L1 cache evaluation on benchmark "test2.txt":

   ./bin/cache-sim 32 direct lru 4 1024 assoc lru 8 tests/test2.txt

   ./bin/cache-sim 64 direct lru 4 1024 assoc lru 8 tests/test2.txt

   ./bin/cache-sim 128 direct lru 4 1024 assoc lru 8 tests/test2.txt

   ./bin/cache-sim 256 direct lru 4 1024 assoc lru 8 tests/test2.txt

   ./bin/cache-sim 512 direct lru 4 1024 assoc lru 8 tests/test2.txt

4. Run L1 cache evaluation on benchmark "test3.txt":

   ./bin/cache-sim 32 direct lru 4 1024 assoc lru 8 tests/test3.txt

   ./bin/cache-sim 64 direct lru 4 1024 assoc lru 8 tests/test3.txt

   ./bin/cache-sim 128 direct lru 4 1024 assoc lru 8 tests/test3.txt

   ./bin/cache-sim 256 direct lru 4 1024 assoc lru 8 tests/test3.txt

   ./bin/cache-sim 512 direct lru 4 1024 assoc lru 8 tests/test3.txt

5. Run L2 cache evaluation on benchmark "test2.txt":

   ./bin/cache-sim 32 direct lru 4 1024 assoc:1 lru 8 tests/test2.txt

   ./bin/cache-sim 32 direct lru 4 1024 assoc:2 lru 8 tests/test2.txt

   ./bin/cache-sim 32 direct lru 4 1024 assoc:4 lru 8 tests/test2.txt

   ./bin/cache-sim 32 direct lru 4 1024 assoc:8 lru 8 tests/test2.txt

   (For the remaining 12 commands, vary the size [1024] between [2048], [4096], and [8192].)

6. Run L2 cache evaluation on benchmark "test3.txt":

   ./bin/cache-sim 32 direct lru 4 1024 assoc:1 lru 8 tests/test3.txt

   ./bin/cache-sim 32 direct lru 4 1024 assoc:2 lru 8 tests/test3.txt

   ./bin/cache-sim 32 direct lru 4 1024 assoc:4 lru 8 tests/test3.txt

   ./bin/cache-sim 32 direct lru 4 1024 assoc:8 lru 8 tests/test3.txt

   (For the remaining 12 commands, vary the size [1024] between [2048], [4096], and [8192].)

7. Run overall cache evaluation on all benchmarks:

   ./bin/cache-sim 32 direct lru 4 4096 assoc:4 lru 8 tests/test1.txt

   ./bin/cache-sim 32 direct lru 4 4096 assoc:4 lru 8 tests/test2.txt

   ./bin/cache-sim 32 direct lru 4 4096 assoc:4 lru 8 tests/test3.txt

   ./bin/cache-sim 32 direct lru 4 4096 assoc:4 lru 8 tests/test4.txt

   ./bin/cache-sim 32 direct lru 4 4096 assoc:4 lru 8 tests/matrix_mult_test.txt

   ./bin/cache-sim 32 direct lru 4 4096 assoc:4 lru 8 tests/pi_test.txt

   ./bin/cache-sim 32 direct lru 4 4096 assoc:4 lru 8 tests/looping_test.txt

References

[1] Ying Zheng, B. T. Davis, and M. Jordan, "Performance evaluation of exclusive cache hierarchies," *IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software, 2004*, Sep. 2004.

[2] J. Alghazo, A. Akaaboune, and N. Botros, "SF-LRU Cache Replacement Algorithm," *Records of the 2004 International Workshop on Memory Technology, Design and Testing, 2004.*, 2004.

[3] S. P. Vanderwiel and D. J. Lilja, "Data Prefetch Mechanisms," *ACM Computing Surveys*, vol. 32, no. 2, pp. 174–199, 2000.

[4] N. Megiddo and D. S. Modha, "Outperforming LRU with an adaptive replacement cache algorithm," *Computer*, vol. 37, no. 4, pp. 58–65, 2004.