Cache Simulator

Lucas Herrera and Doug Slater

***Abstract*—This paper outlines the design and implementation of a cache simulator that demonstrates the performance of a cache given cache size, block size, associativity, and write policy. With the impact of a cache design on the execution of a piece of code, it is beneficial to rapidly test the code using a cache simulator to understand the affects of different cache designs. After evaluating the simulation results, a developer can make the necessary adjustments to the cache design or the instruction order.**

Keywords*—*cache simulator, write-back, write-through, associativity

I. INTRODUCTION

INTEGRATED cache memory allows a processor to speed up memory accesses, and mitigate the bottlenecks associated with reading and writing from main memory. Three types of cache memory exist on desktop and server systems, an instruction cache, data cache, and a translation lookaside buffer. The functional unit that comprises these different cache types are known as cache entries.

II. BACKGROUND AND MOTIVATION

A. Cache Entries

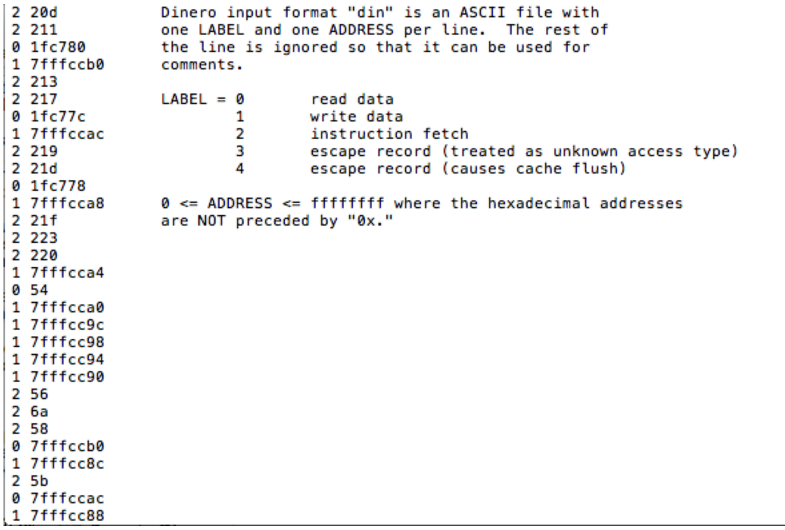
Cache entries are characterized by the data in the entry, the address (tag) from which the data was copied, and flag bits that indicate the validity of the data. The data is transferred between memory and cache in blocks called cache lines. The tag contains a semiunique portion of the memory address. When a read or write is executed, the cache is checked first. If the data is in the cache, a cache hit occurs, and the processor immediately uses the cache line. Otherwise, a cache miss occurs, and the cache allocates a new entry, the data is copied, and then the processor uses the data in the cache.

B. Replacement Policies

In order to make room for a new entry on a cache miss, the cache may have to evict an existing entry. The method that determines the replacement is known as the replacement policy. The best policy will remove the entry that is least likely to be used in the future. Obtaining this goal wholly depends on the instruction set, and there is no perfect way to predict this. One popular replacement policy removes the least recently used cache entry.

C. Write Policies

If data is written to cache, it must also be written to main memory. The method of writing to main memory is known as the write policy. There are basically two policies to execute the main memory writes. Write-through policy means that every write to the cache causes a write to the main memory. Write-back policy means that the cache tracks which locations have been written over, and writes occur upon eviction of that data.



```
2 20d           Dinero input format "din" is an ASCII file with
2 211           one LABEL and one ADDRESS per line.  The rest of
0 1fc780        the line is ignored so that it can be used for
1 7fffccb0      comments.
2 213
2 217           LABEL = 0       read data
0 1fc77c                1       write data
1 7fffccac              2       instruction fetch
2 219                   3       escape record (treated as unknown access type)
2 21d                   4       escape record (causes cache flush)
0 1fc778
1 7fffcca8      0 <= ADDRESS <= ffffffff where the hexadecimal addresses
2 21f           are NOT preceded by "0x."
2 223
2 220
1 7fffcca4
0 54
1 7fffcca0
1 7fffcc9c
1 7fffcc98
1 7fffcc94
1 7fffcc90
2 56
2 6a
2 58
0 7fffccb0
1 7fffcc8c
2 5b
0 7fffccac
1 7fffcc88
```

Fig. 1: Sample.din, a trace file that the simulator uses to track the cache performance

D. Associativity

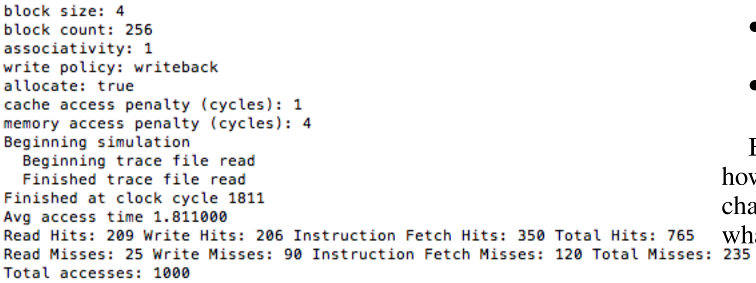
Since the cache is smaller than the main memory, a relationship is needed to dictate how the cache is organized. In a fully associative cache there is no predictable relationship, and the write policy is free to evict and replace any entry that it wishes. A direct mapped cache dictates that every address in main memory has a specific place in the cache that it must be placed. Most elements implement an N-way set associative policy, which combines the benefits of these policies by incorporating the flexibility of fully associative, with the predictive capabilities of

E. Motivation

A cache simulator allows a developer to run code on different memory environments to determine which parameters and design will provide the best performance.

III. SIMULATOR DESIGN

Although less sophisticated, the simulator has been designed with the *Dinero* cache simulator in mind. There are no external code dependencies, and the simulator accepts `.DIN` files as input. The simulator has been developed in `Python` to allow



```
block size: 4
block count: 256
associativity: 1
write policy: writeback
allocate: true
cache access penalty (cycles): 1
memory access penalty (cycles): 4
Beginning simulation
  Beginning trace file read
  Finished trace file read
Finished at clock cycle 1811
Avg access time 1.811000
Read Hits: 209 Write Hits: 206 Instruction Fetch Hits: 350 Total Hits: 765
Read Misses: 25 Write Misses: 90 Instruction Fetch Misses: 120 Total Misses: 235
Total accesses: 1000
```

Fig. 2: Figure showing the simulator output tracking the performance of the cache design

cross-platform execution. The simulator is written in four class files.

1) *Clock.py* contains clock cycle counter shared between memory and clock
2) *Memory.py* contains RAM simulation
3) *Cache.py* contains the CPU cache
4) *CacheSim.py* contains the simulation reads parameters and reads simulation file.

These four classes comprise the entire cache simulation and facilitate the flexibility to vary all of the design parameters.

A. *Features*

The cache simulator is flexible by design and offers several parameters that can be varied to find the most effective cache design. The replacement policy used in the cache simulator is a least recently used policy.The associativity can be directly mapped or set associative. The set associativity can be organized by 2, 4, or 8 block sets. The simulator offers both write-through and write-back policies that account for the penalties associated with either policy. The memory system is organized in a non-layered hierarchy where the one cache interacts with the main memory directly. The block size and cache size are variable by powers of two. This allows the developer to more accurately represent the cache system they desire to simulate. Furthermore, the instruction cache and data cache are separate. The simulator does not actually move data as that is not necessary given the variable miss penalty parameter.

Running simulations with different cache policies, the developer will be able to see which cache design will allow the code to execute in the least amount of cycles. As processor frequencies, which vary across different processors, will dictate the runtime of the code fragment, clock cycles are the best measure of performance for a cache design.

IV. SIMULATOR OUTPUT

Given the input `.DIN` file, the simulator will output several metrics that indicate the performance of the cache design. The output includes:

- parameters: the design characteristics of the cache,
- cycles per fetch: the average cycles to fetch each instruction,
- hits: the number of times a fetch executed without interacting with the main memory, and
- misses: the number of times a fetch had to interact with main memory.

By comparing the output information, it is easy to see how a design performs for each input file. Altering the cache characteristics and policies will allow a developer to decide what cache system is best suited for the needs of the code.