HW3 Cache Optimization

郭宗信, 111550105

***Abstract*—This study focuses on optimizing the L1 data cache (D$) of the Aquila System-on-Chip (SoC) to improve the efficiency of π computation using Machin's formula. The baseline design, a 4-way set associative cache, was thoroughly analyzed to assess its hit/miss rates, latency, and resource utilization. The optimization process involved adjustments to cache associativity and replacement policies, with a particular emphasis on implementing a random and LRU replacement policy. This approach outperformed traditional FIFO strategy in terms of overall performance. However, reducing associativity was found to have a limited impact on the computation time for π. Future work will explore the integration of prefetching mechanisms and dynamic cache designs to further enhance performance and adaptability for a wider range of workloads.**

Keywords—Data Cache Optimization; Cache Associativity;Replacement Policies; FPGA-based SoC;Random Replacement Policy;

I. INTRODUCTION

The performance of FPGA-based System-on-Chip (SoC) architectures is closely tied to the efficiency of their memory hierarchies.

This work focuses on optimizing the L1 data cache for π computation. The baseline 4-way set associative cache was analyzed for hit/miss rates, latency, and resource utilization to identify optimization opportunities. Key contributions include the implementation and evaluation of replacement policies, specifically LRU and Random, with the latter demonstrating superior adaptability and performance compared to traditional FIFO strategy. Experiments on varying associativity levels (2-way, 4-way, and 8-way) revealed that changes in associativity had limited impact on computation time for π, while replacement policies significantly influenced cache performance. The LRU replacement policy optimized hit rates and reduced latency more effectively than other strategies. These findings highlight the potential of replacement policy modifications in enhancing cache performance, establishing a basis for future exploration into advanced optimization techniques such as prefetching and dynamic replacement policies to improve SoC adaptability and efficiency.

II. SYSTEM DESCRIPTION

A. Description

The implementation leverages a modular Verilog cache design (`dcache.v`) and a C program (`pi.c`) to generate the workload.

This study utilizes the Aquila System-on-Chip (SoC) as the experimental platform, featuring a configurable L1 data cache (D$). The cache design serves as a critical component for evaluating the performance of memory-intensive computations, such as π calculation using Machin's formula.

B. Cache Configuration

The baseline cache design includes the following key characteristics:

- **Cache Size**: 4 KB
- **Associativity**: 4-way set associative
- **Replacement Policy**: FIFO
- **Data Block Size**: 32 bytes

C. Workload

The workload for this study involves calculating π to 5000 decimal places using Machin's formula. The computation introduces a high volume of repeated memory accesses (read:36,282,622 write: 26,147,435).

III. IMPLEMENTATION

This part focused on three main aspects of implementation: data analysis, replacement policy modification, and associativity adjustment. Each aspect contributed to evaluating and optimizing the L1 data cache for better performance under specific workloads.

A. Data Analysis

To facilitate performance analysis, key cache-related variables were monitored using `(*mark_debug = "true" *)`, enabling observation in the Integrated Logic Analyzer (ILA). Metrics such as hit/miss counts, hit/miss rates, and latency were computed for both read and write operations. The analysis process is summarized in the flowchart (Fig. 1 and 2). By focusing on these metrics, the implementation provided a detailed view of cache performance and identified potential bottlenecks for further optimization.

Fig. 1. cache_statistics_flowchart*)*

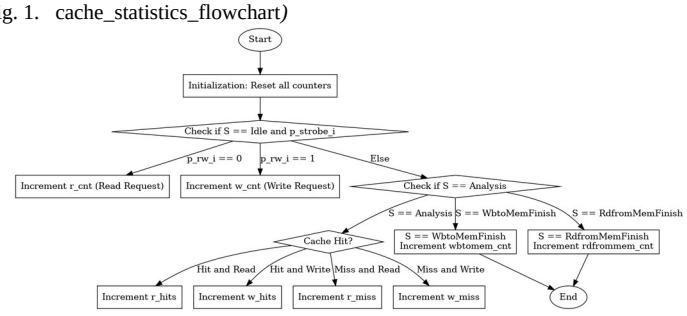
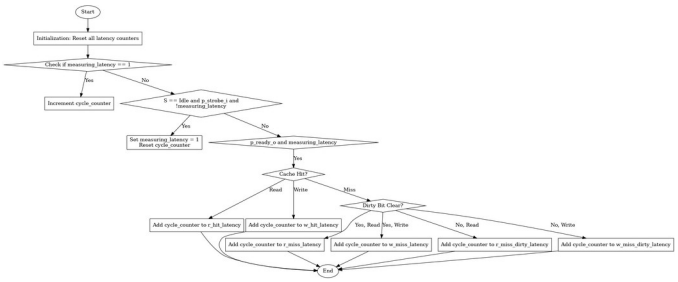


Fig. 2. *latency_tracking_flowchart*



B. Replacement Policy Implementation

Two replacement policies, **Least Recently Used (LRU)** and **Random**, were implemented and evaluated against the baseline FIFO policy.

LRU Replacement Policy: The LRU implementation extended the logic of the baseline FIFO. A two-dimensional array `LRU` of size `N_LINES * N_WAYS` was introduced to track the usage order of cache lines, where `[0]` indicated the least recently used way and `[N_WAYS-1]` represented the most recently used. During cache hits, the hit way was moved to the most recent position, while other ways were shifted forward. For cache misses, the least recently used way, `LRU[line_index][0]`, was selected as the victim. The LRU array was initialized during reset and updated dynamically during cache operations.

Random Replacement Policy: The random policy was designed to avoid patterns that might degrade cache performance. Two seeds, `random_seed_lfsr` and `random_seed_time`, were used to generate pseudo-random numbers. The XOR of these seeds, modulo `N_WAYS`, determined the victim way during cache misses. The `random_seed_lfsr` was implemented using a Linear Feedback Shift Register (LFSR) for pseudo-random generation, while `random_seed_time` acted as a simple counter. This approach ensured a diverse selection of victim ways, minimizing bias in replacement decisions.

C. Associativity Adjustment

To explore the impact of associativity on performance, the cache was modified to support 2-way and 8-way set-associative configurations. Adjustments required changes to key logic components:

Way Matching: For way_hit signals, only way_hit[0] and way_hit[1] were active in 2-way, while all eight way_hit[0] to way_hit[7] signals were used in 8-way.

Cache Hit Detection: The cache_hit logic was updated to (way_hit[0] || way_hit[1]) for 2-way and (way_hit[0] || way_hit[1] || ... || way_hit[7]) for 8-way configurations.

Hit Index Calculation: The hit_index logic was adjusted to handle 2 or 8 cases, ensuring proper identification of the hit way.

IV. Results and Discussion

One of the most critical factors influencing cache performance is the **hit rate**, as higher hit rates reduce the frequency of expensive memory accesses to lower levels of the memory hierarchy, such as main memory or disk. These accesses significantly impact overall execution time. To evaluate performance, I recorded both **read** and **write** hit/miss events during the π computation process and computed the respective hit rates, as shown in **Table I**.

The data reveals that **LRU** consistently achieves the highest read and write hit rates, followed by the **Random** replacement policy, while **FIFO** performs the least effectively in this regard. Interestingly, despite these differences in hit rates, the **2-way and 4-way associativity configurations** exhibit comparable performance in terms of execution time, suggesting that higher associativity (e.g., 8-way) does not provide significant benefits for the given workload. The minor differences in hit rates and performance between the 2-way and 4-way configurations indicate that the trade-off between complexity and resource utilization can favor lower associativity without significant performance degradation.

The cache controller's **Finite State Machine (FSM)** provides insights into the latency impact of cache misses. Under ideal conditions, the FSM transitions directly from `Idle` to `Analyze`, resulting in a hit. However, for misses, the required operations depend on whether the cache line is **dirty**:

Dirty Misses: Data must be written back to memory before loading new data. This write-back operation significantly increases latency. Observations using the Integrated Logic Analyzer (ILA) showed that dirty misses introduce a latency of approximately 51 cycles between p_strobe_i and p_ready_o.

Clean Misses: These involve only reading data from memory into the cache and incur a latency of approximately 31 cycles.

A. Hit and Miss Analysis

The relationship between hit/miss metrics and cache performance is crucial. A higher hit rate minimizes the need for expensive memory accesses, significantly reducing execution time. Table 1 summarizes the hit and miss rates for each.

TABLE I. READ HIT RATE AND WRITE HIT RATE OF DIFFERENT POLICY

Policy	***Read Hit Rate***	***Write Hit Rate***
4-way FIFO	77.57%	83.30%
4-way LRU	80.44%	84.66%
4-way Random	78.61%	84.62%
8-way FIFO	77.57%	83.30%
2-way FIFO	77.57%	83.30%

This figure show that LRU consistently outperformed other policies by achieving the highest hit rates for both read and write operations. Also demonstrate that adjusting associativity (2-way and 8-way) did not significantly affect hit rates.

*B. **Latency and Dirty Miss Analysis***

Latency analysis revealed critical insights into the impact of dirty and clean misses. Misses with dirty bits incurred additional latency due to write-back operations, as shown in Table II:

TABLE II. LATENCY OF DIFFERENT SITUATION

Policy	*Dirty Miss Latency (Cycles)*	*Clean Miss Latency (Cycles)*
4-way FIFO	51	31
4-way LRU	50	31
4-way Random	50	30

We can find that :

1. The latency overhead of dirty misses significantly impacts cache performance, with a single dirty miss averaging 51 cycles.

2. Both LRU and Random policies slightly reduced average latency compared to FIFO by better managing replacements and minimizing unnecessary write-backs. But this may due to the analysis residual.

C. Execution Time Comparison

Execution time was measured for π computations to 5,000 decimal places across different configurations. Table III highlights the execution times:

TABLE III. EXECUTION TIME OF DIFFERENT POLICY AND ASSOCIATIVITY

Policy	Execution Time (ms)
4-way FIFO	30,524
4-way LRU	28,983
4-way Random	29,702
8-way FIFO	30,517
2-way FIFO	30,516

LRU achieved the best execution time, improving by approximately 5% over FIFO due to fewer dirty misses and improved replacement decisions.

The Random policy followed closely, demonstrating its ability to adapt to workloads with unpredictable access patterns.

Execution times for 4-way and 8-way configurations were comparable, suggesting diminishing returns for increased associativity under the tested workload or just the acceptable tolerance.

V. FUTURE WORK

Future work can focus on exploring alternative strategies for improving the efficiency of π computation and optimizing the D-cache design. For instance, modifying the program logic to reduce memory access intensity or reorganizing data flow in the π computation process could minimize cache dependency. Using alternative methods for calculating π, such as the Gauss-Legendre algorithm or Monte Carlo simulations, may reveal how different computational patterns interact with cache policies and associativity levels.

Also, from a hardware perspective, implementing a write-back buffer could mitigate the latency caused by dirty block writes. By temporarily storing write-back data, the processor could continue executing subsequent instructions without waiting for memory operations to complete. This approach could significantly improve throughput, particularly for write-heavy workloads.

Futher more, adopting advanced replacement policies like LFU (Least Frequently Used) or dynamic replacement strategies that adjust based on workload patterns could further enhance cache efficiency. For example, a dynamic policy that transitions between random and LRU depending on access behavior may optimize hit rates while reducing resource overhead.

Last, we can focus on dynamic associativity, where the cache adjusts the number of active ways based on the workload, could also be explored. This method would allow the cache to balance between resource usage and performance, activating more ways for complex workloads and reducing them for simpler tasks.

In summary, these potential advancements in both software and hardware approaches could pave the way for more efficient π computation and optimized cache performance, addressing current limitations while setting a foundation for future exploration.