Literature Review

Cache Write Policies
Cache Replacement Strategies
Cache Initialization

Justin Clark

Introduction

In computer architecture, efficient memory access is crucial to overall system performance. Further, designing efficient cache write policies, replacement strategies, and initialization all play an important role. Cache write policies are chosen to decide the best time to update main memory with the contents in cache memory. Cache replacement strategies are are implemented for the system to effectively choose which contents of cache memory should remain and which should be overwritten. The problem of cache initialization deals with how cache memory should be initialized in the first place (after a system reboot, for example).

These three issues in cache design are all focused on optimizing one thing: the *hit ratio* (though there are nuances, like the *effective* hit ratio which accounts for other architectural inefficiencies, e.g. queuing delays, scheduling, etc.). The hit ratio is the proportion of memory accesses where the fetched data or instruction is in cache memory. When a computer system has a high hit ratio, cache memory is being used effectively. The following discussion of these three important cache memory topics will cover the foundational concepts as well as review recent literature on each topic.

Cache Write Policies

As discussed in [1], there are two major cache write policies. The first is called *write-through cache* where the the system updates main memory with the current contents of the corresponding cache memory location any time that cache memory location is written to. This means that main memory could be written to many times, thus creating a bottleneck. The more frequently a given cache memory location is altered, the more ineffective the write-through policy is. Although easy to implement, the biggest issue with write-through cache is that it undermines the entire purpose of cache - to minimize the use of main memory.

On the other end of the spectrum, there is a policy called *write-back cache.* This cache write policy is designed to only update the corresponding main memory location when the cache memory location is removed from cache. This policy is more complex to implement as it requires a separate bit and corresponding logic to determine whether or not to update main memory, but the performance improvement is worth the cost in many modern computer systems.

There are some nuances to the implementation of write-back cache policies (or any other cache mechanism for that matter). For one, write-backs do not always happen instantaneously and are instead stored in a write-back buffer [2]. Typically, these write-backs are using the primary bus between cache memory and main memory. As discussed in [2], this causes a bottleneck and can degrade overall system performance. The authors propose a new architecture with a secondary bus for cache write-backs as shown in Figure 1. The results of a simulations run in [2] show that there are dramatic decreases in queuing delays when using a secondary bus that connects the write-back buffer directly to many memory. Results and benchmarks used are in Table 1 [2].

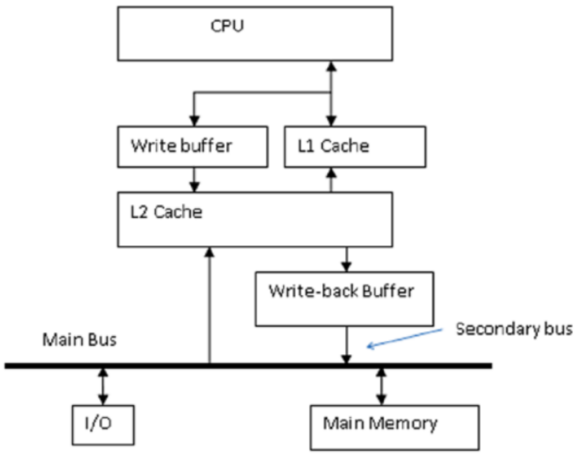

Figure 1: Secondary bus architecture [2]

Benchmark	Cycles queued without secondary	Cycles queued with secondary	% decrease
matscalar	58,109,468	883,571	98.48
mattrans	37,005,612	4,208,484	88.63
minigeo	62,802,508	9,534,934	84.82

Table 1: Decrease in queued cycles on primary bus

Of course, write-through and write-back cache policies aren't the only types of cache write policies there are. There are many policies that are domain specific. For example, in [3], the authors discuss a cache write policy for parallel image processing. Their method is called *cache write generate*. This cache write policy updates the cache directly when cache writes are missed. A cache write miss is the converse of a hit, *i.e.* when there is a failed attempt to write to the cache. On a miss, the system is required to access main memory which is undesirable. Some results of the cache write generate policy simulations are shown in Table 2 [3].

SPEEDUP FOR THE *os* PROGRAM 64K CACHES

No. Proc.	1 p						4 p					
Memory	f		m		s		f		m		s	
Cache	x	y	x	y	x	y	x	y	x	y	x	y
N/G	0.97	0.99	0.95	0.99	1.08	1.19	1.41	1.48	1.55	1.46	1.56	1.52
A/G	1.00	1.00	1.02	1.01	1.21	1.30	1.05	1.14	1.30	1.27	1.48	1.47

Table 2: N is a normal cache write-back policy, A is another method called cache write allocate, and G represents cache write generate. The speedup is defined as the ratio of processing times in the first column (N/G and A/G). There are results from fast, medium, and slow simulated memory devices as well as on a single and four simulated processor(s). The x and y in the Cache row represent two different simulated implementations of a cache where x is a simpler implementation and y is more complex.

If we focus on the speedup over the typical write-back cache (N/G), we notice that for single processors, cache write generate shows no improvement over write-back cache. However, for the four processor system, there is a substantial improvement over the normal write-back cache policy. The authors discuss that cache write generate is an effective strategy for an increasing number of processes on a shared bus [3].

Cache Replacement Strategies

As described in [1], there are many techniques for determining which cache locations to keep and which to overwrite. A couple of the most popular strategies are (LRU) and *first-in, first-out* (FIFO). LRU has been the industry standard for decades due to it's simplicity and good approximation of an ideal cache system [1,4]. LRU, as the name implies, replaces the cache

location that was used the longest time ago. Both FIFO and LRU are ways of replacing the cache location with the lowest temporal locality. Of course, there have been many proposed "better" solutions to replace LRU. One major area of inquiry is in producing cache replacement strategies that hold more predictive power than temporal locality as discussed in [4,5].

In [4], the authors propose a method that creates predictors of the *reuse information* in a cache. The reuse information of a cache are the bits that determine the locality of a cache entry. The authors of [4] create two predictive algorithms for the reuse information. One is a static predictor that uses reuse information of profiling runs to determine associated memory calls. The second predictor is a dynamic predictor that adjusts the reuse information at runtime. The authors also extend their predictors to include the option of an LRU proxy for when the predictors are likely to perform poorly. When simulated, this predictor outperformed LRU in level-2 (L2) cache by 12.32% and 19.95% on SPEC benchmarks for the static and dynamic predictors respectively [4].

Along similar lines, the authors of [5] describe a probabilistic strategy for replacing cache locations. Instead of a deterministic algorithm or prediction, the *Probabilistic Replacement Policy* (PRP) replaces the cache location with the minimum hit probability. This cache replacement policy is specifically designed for *last-level caches* (LLC) Figure 2 and its description of a PRP architecture is taken from [5]:

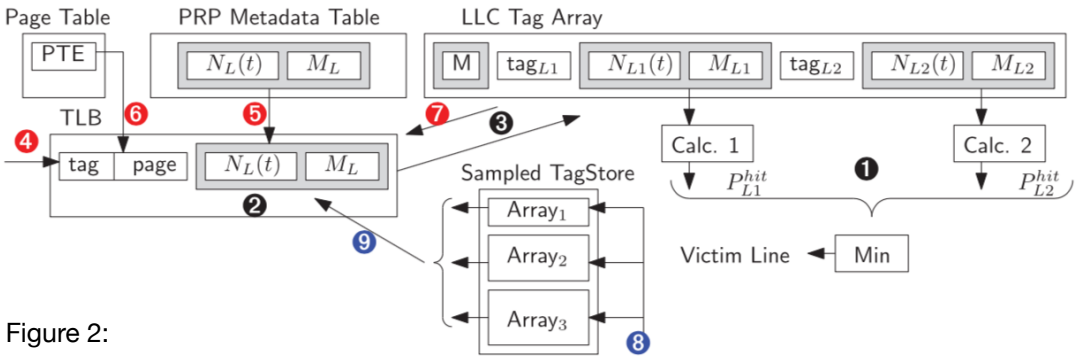

Figure 2:

Overall hardware for PRP, $N_L(t)$ is the frequency of the reuse distance t, and M_L is the last access timestamp for line L. Reuse distance distribution, $N_L(t)$, of lines are stored in the cache as metadata, and are used to make replacement decisions by evicting the line with minimum hit probability, P_L^{hit} (Section 4) [❶, ❷, ❸]. $N_L(t)$ is stored in a separate DRAM area, and fetched with TLB misses (Section 4.4) [❹, ❺, ❻, ❼]. Sampled Tagstore is used to eliminate fetching timestamps (Section 4.4) [❽, ❾].

The performance of PRP was assessed using the SPEC-CPU2006 benchmarks and outperformed the previous state-of-the-art policy (called SHiP), by 4.0% and reduced LLC misses by 6.6%. Though state-of-the-art, one major thing to note about an architecture like this is that much more overhead is required beyond a simple LRU. For example, this architecture needs to save the distribution of the reuse distance and a last access timestamp as metadata for each cache location. As with all policies, there is a trade-off between maximizing the hit ratio and hardware overhead. The design to be used by a given system depends heavily on the purpose of the system.

To recap, LRU and its temporal distance variants remain the de-facto standards for basic cache replacement strategies because of their simplicity and good approximation to an ideal cache replacement policy. However, there have been many state-of-the-art replacement policies designed [4,5,6,7] that move beyond temporal locality policies in hopes of providing computer designers with faster cache designs at all levels. In [6], the authors propose a method called *Locality-Aware Cost Sensitive Cache Replacement*. Their focus is on the LLC. In [7], the authors attempt to solve the problem of effectively caching location-dependent data in mobile environments.

Cache Initialization

Cache initialization is how cache locations are initialized when cache is starting up. As discussed in [1], this cannot be overlooked because if there are “garbage” instructions in cache locations the system could run into serious issues (since the processor is executing incorrect instructions). The most popular and simple method is to include a *valid bit* at each cache location where 1 means it is a valid location and 0 means it is not valid. If a cache location is accessed and it has a 0 valid bit, the operation constitutes a miss and the fetch will reach to main memory and overwrite that cache location with a valid bit equal to 1. An important thing

to note here is that a system does not have to restart for cache to become *un*-initialized. It is mentioned in [1] that the operating system could flush cache for security purposes.

While there is a simple existing strategy for initializing cache (the “valid bit strategy”), it may be undesirable for a system to wait for cache to refill with valid entries after a cache reset. In [8], the authors develop a preemptive cache loading for real-time (highly time-sensitive) scheduling systems. As stated in the abstract, the paper introduces the concept of persistent cache blocks which are used to capture the reuse information (defined in the previous section) of cache blocks between executions of a task. This means that cache blocks will never be evicted once loaded into cache. This new method improves performance by 10% in schedulability over the baseline (UCB-union multi-set approaches discussed in [8]) using the Malardalen Benchmark. Similar preemptive approaches are covered in [9] and [10], but are omitted for brevity.

Conclusions

Each of the preceding three sections discussed major components to an optimally functioning cache memory system. There have been long-lived, simple, and generally effective developments for cache write policies (write-back cache), cache replacement strategies (LRU), and cache initialization (the use of a “valid” bit). However, these simple approaches are not necessarily optimal for all systems. For example, there are cases where cache write policies that handle highly parallel image processing might be necessary [3]. As discussed in the “Cache Initialization” section, the authors of [8,9,10] all propose cache initialization strategies for time-sensitive systems. One of the key takeaways is that form should follow function if computer architecture is to reach optimal performance under given constraints.

References

[1] Dumas, Joseph D. II. Computer Architecture: Fundamentals and Principles of Computer Design, Second Edition, CRC/Taylor & Francis, 2017. ISBN 978-1-4987-7271-6.

[2] O'farrell, J., & Baskiyar, S. (2015). Enhanced real-time performance using a secondary bus for cache write-backs. International Journal of Computers and Applications, 37(1). doi: 10.1080/1206212X.2015.1061157

[3] Wittenbrink, C., Somani, A., & Chung-Ho Chen. (1996). Cache write generate for parallel image processing on shared memory architectures. Image Processing, IEEE Transactions on, 5(7). doi:10.1109/83.502410

[4] Feng, M., Tian, C., Lin, C., & Gupta, R. (2011). Dynamic access distance driven cache replacement. ACM Transactions on Architecture and Code Optimization (TACO), 8(3). doi: 10.1145/2019608.2019613

[5] Das, S., Aamodt, T., & Dally, W. (2016). Reuse Distance-Based Probabilistic Cache Replacement. ACM Transactions on Architecture and Code Optimization (TACO), 12(4). doi: 10.1145/2818374

[6] Kharbutli, M., & Sheikh, R. (2014). LACS: A Locality-Aware Cost-Sensitive Cache Replacement Algorithm. Computers, IEEE Transactions on, 63(8). doi:10.1109/TC.2013.61

[7] Gupta, A., & Shanker, U. (2018). SPMC-CRP:A Cache Replacement Policy for Location Dependent Data in Mobile Environment. Procedia Computer Science, 125. doi:10.1016/j.procs. 2017.12.081

[8] Rashid, S., Nelissen, G., Hardy, D., Akesson, B., Puaut, I., & Tovar, E. (2016). Cache-Persistence-Aware Response-Time Analysis for Fixed-Priority Preemptive Systems. Real-Time Systems (ECRTS), 2016 28th Euromicro Conference on. IEEE. doi:10.1109/ECRTS.2016.25

[9] Conte, T., Sathaye, S., & Banerjia, S. (1996). A persistent rescheduled-page cache for low overhead object code compatibility in VLIW architectures. Microarchitecture, 1996. MICRO-29.Proceedings of the 29th Annual IEEE/ACM International Symposium on. IEEE Publishing. doi:10.1109/MICRO.1996.566445

[10] G. C. Buttazzo, M. Bertogna, G. Yao, "Limited preemptive scheduling for real-time systems. a survey", *Industrial Informatics IEEE Transactions on*, vol. 9, no. 1, pp. 3-15, 2013.