

Collaborative Caching for Unknown Cache Sizes

Xiaoming Gu*

xiaoming@cs.rochester.edu

I. INTRODUCTION

A number of hardware systems have been built or proposed to provide an interface for software to influence cache management. Examples include cache hints on Intel Itanium [1], bypassing access on IBM Power series [4], and evict-me bit [5]. Wang et al. called a combined software-hardware solution *collaborative caching* [5]. Our work showed that in theory collaborative caching may enable a program to manage variations of LRU cache optimally [3].

A common limitation, however, is that the optimal management is cache size specific. For the same execution trace, a different collaboration scheme may be needed if the cache size changes. The size dependence hurts portability and makes it unattractive to consider optimal collaborative caching by a general-purpose compiler.

Our recent work showed a generalized model called *LRU-MRU cache* [2]. It supports two types of accesses: the normal LRU access and the special MRU access. The datum loaded by an MRU access is managed by MRU replacement. At an eviction, it selects the most recently used data as the victim. MRU access is equivalent to cache bypassing in IBM processors [4] and tagging the loaded data with an evict-me flag [5]. With LRU-MRU cache, a program controls the cache management by selecting which data to be accessed by which type. But like previous schemes, the optimal selection changes with the cache size.

In this work we first present a prioritized LRU model. For each memory access, a program specifies a priority. The loaded datum can be inserted in the middle of the cache stack between the LRU and MRU positions. It may be bypassed if the associated priority is too low. Prioritized LRU naturally organizes program accesses for all cache sizes.

Alternatively, we describe a dynamic cache control scheme. Like prioritized LRU, each access is associated with a priority. The dynamic cache control compares the priority with the cache size and then chooses either the LRU or the MRU position for placing the data. As a result, a program is optimized for all instead of single cache sizes.

In the discussion, we assume fully associative cache. The same idea can be applied to each set of set associative cache. Another problem is the instruction overhead now that each load and store is associated with a number. One way to reduce the cost is to use a few registers to hold the priority numbers and then include a few bits in a load or store instruction to indicate the priority register.

*Xiaoming Gu is being advised by Prof. Chen Ding (cding@cs.rochester.edu) in Department of Computer Science, University of Rochester, Rochester, NY 14627.

II. PRIORITIZED LRU

In prioritized LRU, an access specified with a priority p makes the accessed datum d go to the p th position in a cache stack. There are 6 cases determined by the relations of i —the current position, p —the priority indicating the target position, and m —the cache size: ① hit move-up ($1 \leq p < i \leq m$)— d is moved up from i th position to p th position; ② hit no-move ($1 \leq i = p \leq m$)— d keeps staying at the same position; ③ hit move-down ($1 \leq i < p \leq m$)— d is moved down from i th position to p th position but still in cache; ④ hit bypass ($1 \leq i \leq m < p$)— d is moved out of cache; ⑤ miss move-in ($i = \infty$ and $1 \leq p \leq m$)— d is moved into cache; ⑥ miss bypass ($i = \infty$ and $p > m$)— d keeps staying out of cache.

In LRU, the accessed datum always goes to the top of a cache stack, i.e. has value 1 for the priority. In LRU-MRU, the accessed datum has two choices—the top or the bottom. Correspondingly, the priority value may be 1 for LRU accesses or the size of the cache for MRU ones. However, the target position indicated by the priority is still limited to the two ends of a cache stack. In prioritized LRU, the accessed datum may go in any possible positions including the middle and the outside of a cache stack.

We have proved that prioritized LRU holds inclusion property: *An access trace is executed on two prioritized LRU caches— C_1 and C_2 ($|C_1| < |C_2|$). At every access, the content of cache C_1 is always a subset of the content of cache C_2 .*

III. DYNAMIC LRU-MRU DETERMINATION

An OPT cache simulation is required to minimize misses on an LRU-MRU cache with a specific size [3]. When an eviction happens in the simulation, the most recent access to the evicted datum is changed from the default LRU to MRU.

The optimality can be achieved without OPT cache simulations if we use the forward OPT stack distance as the priority. The hardware compares the priority with the cache size. If the priority is greater, the access is realized as an MRU access. Otherwise, the access behaves as a normal LRU access. In this way, only a preprocess of OPT stack distance analysis is needed.

REFERENCES

- [1] K. Beyls and E. D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 2005.
- [2] X. Gu and C. Ding. On the theory and potential of LRU-MRU collaborative cache management. In *Proceedings of the 10th International Symposium on Memory Management*, 2011.
- [3] X. Gu, T. Bai, Y. Gao, C. Zhang, R. Archambault, and C. Ding. P-OPT: Program-directed optimal cache management. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 2008.
- [4] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM J. Res. Dev.*, 2005.
- [5] Z. Wang, K. S. McKinley, A. L. Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2002.