An Intra-Tile Cache Set Balancing Scheme

Mohammad Hammoud, Sangyeun Cho, and Rami G. Melhem
Department of Computer Science, University of Pittsburgh
Pittsburgh, PA, USA
mhh@cs.pitt.edu, cho@cs.pitt.edu, melhem@cs.pitt.edu

ABSTRACT

This poster describes an intra-tile cache set balancing strategy that exploits the demand imbalance across sets within the same L2 cache bank. This strategy retains some fraction of the working set at underutilized sets so as to satisfy far-flung reuses. It adapts to phase changes in programs and promotes a very flexible sharing among cache sets referred to as *many-from-many* sharing. Simulation results using a full system simulator demonstrate the effectiveness of the proposed scheme and show that it compares favorably with related cache designs on a 16-way tiled CMP platform.

Categories and Subject Descriptors: B.3.2 [Memory Structures]: Design Styles —*cache memories*

General Terms: Management.

Keywords: Set Balancing, Many-From-Many Sharing.

1. INTRODUCTION AND MOTIVATION

Computer programs exhibit a non-uniform distribution of memory accesses across different cache sets. In this work we observe that some cache sets at different physically distributed, logically shared L2 banks on a tiled CMP suffer from large local miss ratios while some others remain underutilized. Furthermore, we observe that a very large fraction of cache lines placed at L2 remain unused between placement and eviction. As such, these lines don't contribute to good utilization of the silicon estate devoted to the caches. One reason for this phenomenon is that cache lines might be re-referenced at distances greater than the cache associativity. The problem is magnified on CMPs that share caches as on-chip lifetimes of cache lines can become shorter due to the increasing interferences between co-scheduled threads/processes. Cache performance can be improved by retaining some fraction of the working set long enough to provide cache hits on future reuses. Our work extends the lifetime of cache lines by exploiting the large asymmetry in cache sets' usages and flexibly retaining cache lines evicted from highly pressured sets at underutilized ones.

2. THE PROPOSED MECHANISM

We propose an intra-tile cache set balancing strategy that we refer to as Flexible Set Balancing (FSB). FSB allows a highly pressured set (or source) to retain its lines at many underutilized sets (or destinations) within the same L2 bank (or tile). We refer to this sharing as *many-from-one* sharing because the capacity of many sets can be shared by a single set. Furthermore, FSB allows many source sets to retain their lines at a single destination set. This sharing is referred to as *one-from-many* sharing because the capacity of a single set can be shared by multiple sets. Consequently, FSB offers a very flexible (*many-from-many*) capacity sharing among cache sets. FSB adapts to phase changes in programs and doesn't associate any set with any other set (i.e., make a set a sole owner of another set). As long as there is a space available at any set, any source set can immediately leverage that space.

FSB is oriented towards last level caches (in our case L2). FSB requires three main capabilities: (1) deciding upon source and destination sets, (2) retaining working sets of source sets at destination sets in a many-from-many sharing fashion, and (3) locating retained blocks on destination sets upon future reuses. We next describe each capability.

2.1 Retention Limits

The pressure at a cache set can be measured in terms of cache misses or hits. In this work we adopt cache misses as a pressure function. The pressure information can be recorded at an array embedded within the L2 controller of a cache bank. Each cache set corresponds to an entry in the pressure array and the indexes of the cache sets are used to index the array. Each time a miss occurs at a certain set, the array can be updated accordingly (by incrementing the corresponding array slot). In order to allow the array to accurately represent pressures at sets, after every time interval, we keep only part of the pressure values (e.g., 0.25 of values by shifting each value 2 bits to the right).

We define two limits, the low pressure limit (LPL) and the high pressure limit (HPL), to allow a *range* of source sets to retain their blocks at a range of destination sets. A range can encompass one or many sets. When the pressure of a set is below LPL, the set is deemed to be within the limit of the destination sets and can receive lines from *any* source set. In contrast, when the pressure of a set is above HPL, the set is considered to be within the limit of source sets and is permitted, accordingly, to retain its lines at *multiple* destinations sets. Clearly, this allows many-from-many sharing among cache sets. LPL and HPL are defined in equations (1) and (2). The range of source and destination sets can be expanded or contracted by altering α. The max and min parameters are the maximum and minimum pressures on the pressure array.

$$LowPressureLimit(LPL) = min + (\alpha \times (max - min)) \quad (1)$$

$$HighPressureLimit(HPL) = max - (\alpha \times (max - min)) \quad (2)$$

PACT'10, September 11–15, 2010, Vienna, Austria.
ACM 978-1-4503-0178-7/10/09.

2.2 Retention Policy

FSB maintains a small retention table (RT) per each L2 bank. Each cache set has a corresponding RT entry. As such, the number of entries in RT equals the number of cache sets in the L2 bank. RT stores in the i-th entry, RT(i), up to P pointers, each either marked as invalid or pointing to a destination set with a different index. These pointers can be used by FSB to locate retained blocks upon future reuses (more on this shortly).

When an LRU line, L, is evicted from a set i, our retention policy proceeds as follows:

1. The i's corresponding pressure value in the pressure array is looked up, minimum (MIN) and maximum (MAX) values are generated, and HPL and LPL are calculated.
2. If i's pressure is greater than HPL, i becomes a source set and L is deemed eligible for retention. Otherwise, L is discarded.
3. In parallel, RT(i) entry is looked up. If L is eligible for retention and all the P pointers in RT(i) are invalid, MIN is checked. If MIN is less than LPL, L is retained at the cache set corresponding to MIN and an invalid pointer in RT(i) is marked to point to that set. Otherwise, L is discarded.
4. If RT(i) entry, on the other hand, has valid pointers (or at least one valid pointer), these pointers are used to index the pressure array, and the minimum value out of the indexed values is generated and compared against LPL. If satisfied, L is retained at the corresponding cache set and no new pointer is marked to point to that set. Otherwise, the policy checks if an invalid pointer exists.
5. If an invalid pointer is found in RT(i) and MIN is less than LPL, L is retained at the cache set corresponding to MIN and one of the invalid pointers is marked to point to that set. Otherwise, L is discarded.

Note that upon retention, L is inserted as the most recently used (MRU) line in the selected destination set. The LRU line evicted at the destination set, to make room for L, is discarded simply because the destination set doesn't satisfy HPL. As such, FSB avoids ripple effects.

The LRU evicted line, L, at the source set can be either *native* or *retained*. If L is native, FSB simply proceeds with the retention process. Otherwise, we check if L is *active*. We define L to be active if at least one core on the CMP platform had cached a copy of L (in its L1). This can be easily determined from L's associated directory bit vector. We assume that an active L is currently in use by the caching core(s) and, accordingly, attempt to retain it again. If L is retained and not active, we assume that it has been kept long enough in the cache without providing a cache hit, and, accordingly, avoid retaining it over again (although eligible for retention).

We update the pressure array not only at a miss but further when retaining a line at a destination set. Such an update is critical so as to reflect the progressive increasing pressure on a destination set each time it receives a retained line. This makes FSB very flexible and attentive as it allows selecting a different destination set once the pressure of the current destination set surpasses LPL.

Finally, retaining cache lines at destination sets requires extending lines' tags with index fields. This is due to the fact that a cache line must have a one-to-one correspondence with a unique address. Upon discarding a retained line, R, from a destination set, D, we match R's augmented index j with the augmented indexes of D's resident lines. A "no match" outcome means that R is the last retained line at D from the source set j. Consequently, we index RT(j) entry and invalidate the pointer that points to D. Finally, we note that the retention process is activated in parallel with the resolution of a definitive miss. As such, the latency required to retain a cache block becomes completely hidden as resolving an L2 miss usually takes hundreds of cycles.

2.3 Lookup Policy

Upon a request to a cache line, L, the cache starts by always looking up the set i that L's index designates. RT(i) entry is also looked up concurrently. If a hit occurs at set i, the request is satisfied. If, on the other hand, a miss occurs at set i, the cache sets identified by the pointers in RT(i) (if any) are *serially* looked up until either a secondary hit is acquired or a definitive miss is proclaimed. Sets' lookups are serialized in order to keep FSB simple, avoid port contention, and reduce power dissipation. We found in our experimental evaluation that such a serial policy doesn't hurt performance because the gain from hits on retained lines exceedingly offsets the loss from sequential lookups. If no secondary hit is obtained and a definitive miss is asserted, the pressure array is updated at slot i and the retention policy is triggered. In parallel, the requested cache line is fetched from the main memory and inserted at set i.

3. PERFORMANCE RESULTS

For the evaluations, we use the simics simulator and our own CMP cache modules developed in-house. A 16-tile CMP platform is utilized running with Solaris 10 OS. Each tile encompasses 32KB I/D L1 caches and a 512KB L2 cache bank. We assume a low pressure limit (LPL) and a high pressure limit (HPL) each with $\alpha = 0.2$. Lastly, we use several multithreading and multiprogramming benchmarks from SPEC2006, PARSEC, and SPLASH-2 suites

We ran FSB with 1, 2, 4, and 8 pointers in RT(i) and found that, on average, they achieve miss rate reductions of 14.6%, 23.9%, 36.6%, and 48.7% over the nominal shared CMP scheme, respectively. This effectively translates to 4.3%, 8.8%, 13%, and 18.6% improvement in system performance, respectively. Furthermore, we conducted a sensitivity study on different α values (0.1 and 0.3, in addition to 0.2) and observed a low sensitivity of FSB to the examined values. Lastly, we compared FSB against some recent proposals including dynamic set balancing (DSBC) [2] and variable-way set associative (V-WAY) [1]. Results demonstrated that FSB with 4 pointers outperforms V-WAY and DSBC by averages of 7.8% and 8.8%, respectively.

4. REFERENCES

[1] M. K. Qureshi, D. Thompson, and Y. N. Patt. "The V-WAY Cache: Demand-Based Associativity via Global Replacement," *Proc. Int'l Symp. Computer Architecture*, June 2005.

[2] D. Rolán, B. B. Fraguela, and R. Doallo "Adaptive line placement with the set balancing cache," *Proc. Int'l Symp. Microarchitecture*, Dec. 2009.