Scavenger: A New Last Level Cache Architecture with Global Block Priority

Arkaprava Basu[§*] Nevin Kırman[†] Meyrem Kırman[†] Mainak Chaudhuri[§] José F. Martínez[†]

[§]Department of CSE
Indian Institute of Technology
Kanpur 208016 INDIA
{arka,mainakc}@cse.iitk.ac.in

[†]Computer Systems Laboratory
Cornell University
Ithaca, NY 14853 USA
http://m3.csl.cornell.edu/

Abstract

Addresses suffering from cache misses typically exhibit repetitive patterns due to the temporal locality inherent in the access stream. However, we observe that the number of intervening misses at the last-level cache between the eviction of a particular block and its reuse can be very large, preventing traditional victim caching mechanisms from exploiting this repeating behavior. In this paper, we present Scavenger, a new architecture for last-level caches. Scavenger divides the total storage budget into a conventional cache and a novel victim file architecture, which employs a skewed Bloom filter in conjunction with a pipelined priority heap to identify and retain the blocks that most frequently missed in the conventional part of the cache in the recent past. When compared against a baseline configuration with a 1MB 8-way L2 cache, a Scavenger configuration with a 512kB 8-way conventional cache and a 512kB victim file achieves an IPC improvement of up to 63% and on average (geometric mean) 14.2% for nine memory-bound SPEC 2000 applications. On a larger set of sixteen SPEC 2000 applications, Scavenger achieves an average speedup of 8%.

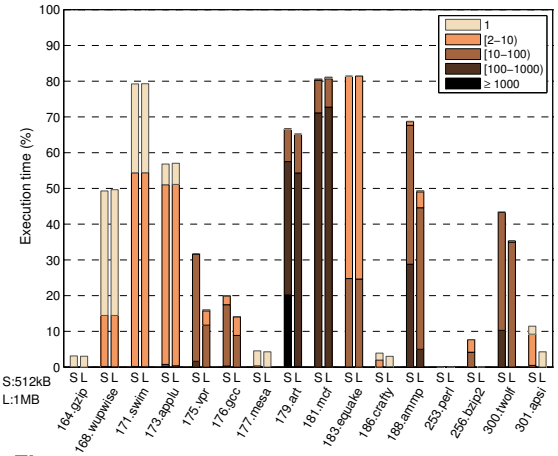

Figure 1. Fraction of the total execution time that retirement is blocked due to a load missing in the L2 cache and where the processor cannot dispatch any instruction. Stall time is broken down based on the number of occurrences of such L2 cache misses for each block address. Results for two L2 cache configurations (S:512kB, L:1MB) are given.

1. Introduction

Over the last decade, DRAM latency emerged as the biggest bottleneck to the evolution of high-end computers, and severely hampered the performance growth in the desktop as well as the server arena. To mitigate this high off-chip data access latency, the microprocessor industry incorporated techniques such as sophisticated hardware prefetchers, large on-chip caches, deep on-chip cache hierarchies, or highly associative last-level caches. But even with these techniques, a significant number of blocks still miss repeatedly in the last level of a cache hierarchy.

Figure 1 quantifies the performance impact of L2 cache misses across sixteen SPEC 2000 applications during the execution of 200 million representative dynamic instructions using 512kB and 1MB 8-way set-associative L2 caches at the last level, and an aggressive hardware stride prefetcher in both cases.[1] The plot shows the fraction of total execution time in which retirement from reorder buffer (ROB) is blocked due to an L2 cache miss at its head, and the processor cannot dispatch new instructions due to lack of free resources. In the experiment, L2 caches are initially empty. We categorize the stall time based on the total number of times the offending block address appears in the L2 cache miss address stream.

The results confirm that the stall time due to long-latency loads is very significant, 30% or higher (up to 80%) for nine out of sixteen applications in the case of the 512kB L2 cache configuration. For many applications, increasing the L2 cache from 512kB to 1MB helps little in reducing the misses, implying that the additional 512kB storage is not utilized effectively.

The results further demonstrate that, for a significant number of these applications, an important part of their reported stall time is due to block addresses that appear in the L2 cache miss address stream repeatedly, suggesting significant potential for improvement by learning such repeated misses and storing these critical blocks.

Often times, a small fully associative victim cache is used

* Now with Sybase Inc.

[1]The setup for this motivating experiment is identical to that in our evaluation (Section 3), except that the evaluation setup uses larger samples of one billion dynamic instructions.

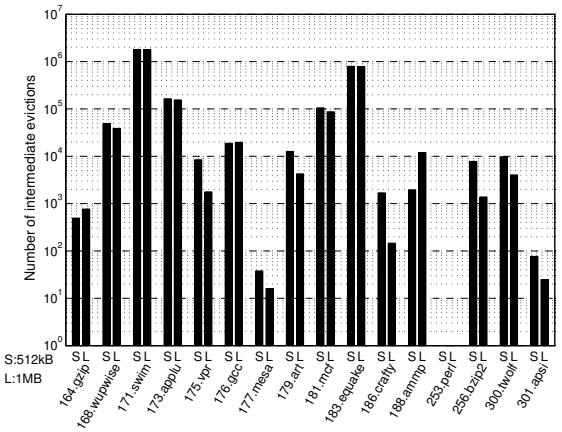

Figure 2. Number of intermediate evictions from the L2 cache since the time a particular block is evicted from the L2 cache until it is requested again. For each missing block address, the median across all eviction-use intervals is first calculated, and then the median across all distinct block addresses is taken as the final result. Note that the y axis is in log scale.

to store evictions from the L2 cache, providing fast restore for the data it holds. However, a traditional victim cache may not be effective in capturing these L2 cache misses. In Figure 2, for each application, we calculate first the median of the number of intermediate evictions across all eviction-use intervals for each distinct block address, and then the median across all block addresses. Many applications exhibit on average thousands of intervening evictions in between eviction-use pairs, which far exceeds the capacity of typical victim caches. These results suggest that different organizations and policies are needed to exploit the repeating miss behavior in the last-level cache.

In this paper, we introduce Scavenger, a new cache organization designed specifically for the last level of a cache hierarchy. Our solution globally prioritizes the block addresses missing in the L2 cache based on the number of times the addresses have been observed in the L2 cache miss address stream in the past and allocates only the high priority blocks in a reasonably sized victim file when they are evicted from the L2 cache. This priority scheme is based on the hypothesis that a cache block, which has caused a large number of misses in the L2 cache in the past, has a higher probability of inflicting more misses in the L2 cache in future. The objective of the proposed architecture is to scavenge the top k most frequently missing blocks in a victim file where k is the size of the victim file.

The Scavenger architecture needs to have three primary capabilities. First, given a block evicted from the L2 cache, it should be able to offer an estimate of the frequency of misses seen to this block address. Second, by comparing this frequency to the minimum frequency among all the blocks currently held in the victim file, it should be able to decide whether to accept this evicted block into the victim file. Third, the victim file organization should be such that it can replace the block having the minimum frequency with a block evicted from the L2 cache having a higher or equal frequency, irrespective of the addresses of these two blocks. Further, finding a particular block in the victim file should be fast enough to be of practical use.

Scavenger makes two major contributions:

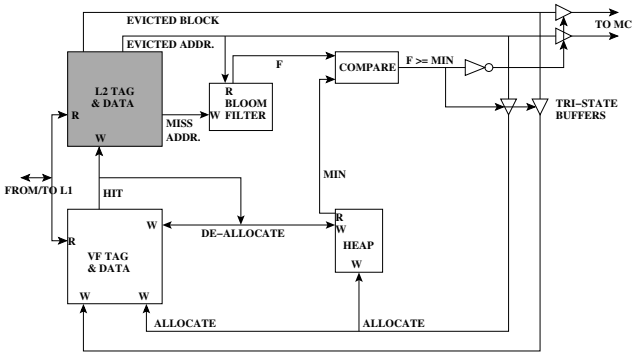

Figure 3. High-level L2 cache organization of Scavenger. The relevant read and write operations on various structures are shown with R and W, respectively. The traditional L2 cache is shown shaded.

- A novel application of skewed Bloom filters (Section 2.1) to accurately estimate the frequency of the blocks appearing in the miss address stream seen so far from the L2 cache.
- A selective low-latency victim caching scheme for a pointer-linked victim buffer enabled by a pipelined priority queue to help retain the blocks with high miss frequency (Sections 2.2 and 2.3).

Execution-driven simulation results (Sections 3 and 4) show that a 512kB+512kB Scavenger configuration achieves an IPC improvement of up to 63%, and average 14.2%, on nine memory-bound SPEC 2000 applications (stall time due to L2 misses greater than 30% (15%) for the 512kB (1MB) configuration in Figure 1), compared to a baseline that uses a conventional 1MB 8-way set-associative L2 cache, and with an aggressive multi-stream stride prefetcher enabled in both architectures. The average IPC improvement achieved on a bigger set of sixteen SPEC 2000 applications is 8%. We also present a thorough analysis of the dynamic and static energy overheads of our proposal.

2. Scavenger Architecture

The Scavenger architecture (Figure 3) divides the total storage budget allocated to the L2 cache into a traditional cache and the proposed victim file (VF) which are kept mutually exclusive. An incoming L1 miss request checks the tag arrays of both structures. On a hit in the conventional cache, the requested sector is returned. If the request hits in the VF, the requested sector is returned and the entire block is moved to the conventional L2 cache. If the request misses in the conventional L2 cache, a Bloom filter, used to keep track of the approximate frequency of L2 cache misses to block addresses, is updated to account for one more miss for this block address, irrespective of whether the VF has the block or not. If the request misses in both the L2 cache and the VF, it is sent to the memory controller (MC). When a cache block is evicted from the conventional L2 cache, the Bloom filter is looked up with the block address to get an estimate of the number of times (priority value) the block has missed in the conventional L2 cache in the past. If the priority value of the evicted cache block is less than the minimum priority value among all the

blocks currently in the VF, the evicted block is not allocated in the VF; otherwise the cache block in the VF with the lowest priority value is victimized to make room for the new L2 cache eviction. A priority queue, organized as a min-heap, is used to maintain the priority values of the blocks currently residing in the VF. The priority queue is updated after a hit in the VF (leading to a de-allocation) or after a new allocation in the VF to reflect the new partial order among the priority values of the cache blocks in the VF. In the following we discuss the major components of Scavenger in detail.

2.1. Frequency Estimator

Scavenger uses a Bloom filter to count the L2 cache misses to block addresses in a cost-effective way. This count or frequency is used to assign a priority value to each missing block. This is a novel application of counting Bloom filters, previously employed for implementing hierarchical store queues [1], coarse-grain coherence [11], snoop filters [12, 19], prefetching and data speculation [13], low-energy synonym lookup [20], etc. A low power counting Bloom filter was proposed in [17]. Our design is influenced by the Spectral Bloom filter proposed in [4].

In our proposal, on arrival of a block address which has missed in the conventional part of the L2 cache, the address is partitioned into p parts and each segment is used to index and increment an 8-bit saturating counter in one of the p banks. In the case of an eviction from the conventional part of the L2 cache, the Bloom filter offers a priority for the evicted block as follows. The evicted block's address is partitioned into p segments and each segment is used to read out one counter from each of the p banks. The miss count or priority of the cache block is computed as the minimum of these p count values. This estimate can be either exact or an over-estimate, but can never be an under-estimate. Notice that if there is at least one of the p counters not aliased with any other block address, the Bloom filter will return the exact miss count. It can be shown that the probability of getting over-estimates (or equivalently false positives) grows exponentially with the number of distinct elements put into the Bloom filter. So, the accuracy of a Bloom filter degrades quickly as more distinct block addresses are hashed into it.

We observed that within a working set the lower order bits of the block address distinguish most of the blocks while the higher order bits do not change much. Based on this observation we use an unevenly banked Bloom filter. We devote major portion of the Bloom filter storage to the lower order banks while keeping only a few counters in the higher order banks. We call this architecture a skewed Bloom filter. Figure 4 shows one such design. In our simulation environment an L2 cache block address is 26 bits (we have a 32-bit address space and 64-byte cache blocks). We partition this address into three segments to index into three main banks. To further reduce the false-positive rate (i.e. the over-estimates), we over-provision the Bloom filter by adding two more small and skewed "overlap" banks. These extra banks often help disambiguate some of the collisions that may have happened in both of the two adjacent normal banks. This effect was briefly mentioned in [12]. The index bits for each bank in our design are shown in Figure 4. The total storage required by our frequency estimator is just over 33kB with one byte counters. Each of the five banks is equipped with one double-ended read/write port.

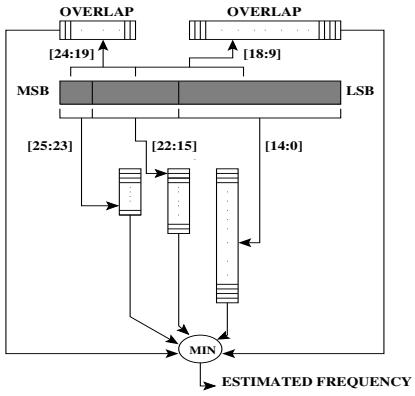

Figure 4. A skewed Bloom filter with five banks for estimating the miss frequency of cache block addresses. The block address bits used to index the banks are also shown.

After operating for a long time, all the counters in the Bloom filter may get saturated to the highest value (255 in our case) and never change again leading to a meaningless estimate of the miss frequency. To solve this problem, for every 8,192 queries, if more than 6,000 of the estimates are saturated, the Bloom filter counters are gang-cleared by pulling down all the bit cell nodes using a wide NMOS transistor. At this time all priority values stored in the priority queue are also gang-cleared so that the stale priority values can be discarded. At this point, all blocks (valid and invalid) in the victim file have zero priority.

2.2. Priority Queue

When a cache block is evicted from the conventional part of the L2 cache, the Bloom filter provides an estimate of the number of times this block has suffered from a miss in the conventional part in the past. This serves as the priority of the cache block and is used to decide whether it should be put in the victim file (VF). An allocation takes place only if the block's priority is higher than or equal to the minimum priority among all the blocks in the VF. The priority values of all the blocks currently residing in the VF are maintained in a priority queue[2] organized as a min-heap.[3] The min-heap is embedded in a k-entry SRAM array, which has two logical fields in each row, namely, the priority value (same size as the Bloom filter counters i.e. one byte) and a pointer ($\lceil \log(k) \rceil$ bits) to the corresponding VF tag entry.

Figure 5 shows the organization of the priority queue with an example. The VPTR field corresponds to the pointer. All the priority values are initialized to zero at the boot time while the VPTR field of entry i is initialized to i.

The priority queue is updated on two occasions. First, on a VF hit, the block is de-allocated from the VF. This requires changing the priority of this VF entry to zero signifying an empty slot. Second, on a VF allocation, normally the new higher- or equal-priority block replaces the minimum-priority

[2] If the running minimum is maintained in a single register, it cannot be updated properly at the time of a de-allocation from the VF.

[3] A min-heap is a balanced binary tree such that the minimum priority value among all the nodes of a subtree is held in the root node of the subtree. This property will be referred to as the "heap property."

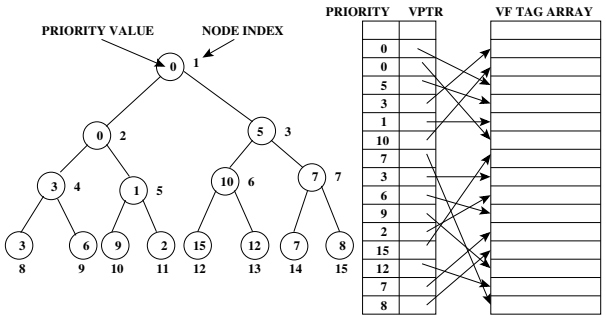

Figure 5. A logical min-heap and its SRAM array implementation. Notice that leaving the first row empty allows us to obtain the children node indices from the parent node index i by simple shift and OR operations: $(i << 1)$ and $((i << 1)|1)$. To accomodate blocks with index zero, one possible solution is to make a small change in the address decoder of the VF tag RAM so that an address that would normally map to index zero of VF is now forced to map to the last entry of the VF. The tag of the last VF entry is extended by one bit to distinguish between these two indices mapping to the same entry.

block pointed to by the VPTR of the root node. In a random mode of operation, which will be explained in Section 2.3.3, a new block may replace any VF block regardless of its priority. This requires replacing a priority value at an arbitrary position in the heap. There are a few more cases that require priority insertion at non-root position (Section 2.3).

Insertion of a new value at the root of a heap is well-understood and can be done in $O(\log(k))$ time. However, replacement of an arbitrary element in the heap is unique to our design. In the following we discuss how an arbitrary priority value in the heap can be replaced by any new value while maintaining the heap property in $O(\log(k))$ time. More importantly, we show how this generalized replacement algorithm can be pipelined to mitigate this high latency.

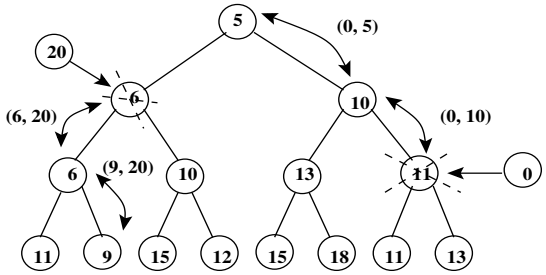

Figure 6. Right subtree: A VF de-allocation requiring insertion of a zero priority value. Left subtree: A VF allocation at a non-root node. Swapped priority values are shown within parentheses.

Figure 6 shows two different scenarios of heap replacement at non-root positions. The first case (right subtree of Figure 6) involves insertion of a zero priority at a non-root node due to a VF de-allocation. The second case (left subtree of Figure 6) requires replacement of a non-root node during a VF allocation. Notice that in both cases, the new value is first written to the node to be replaced and then depending on its value relative to its neighbors it either moves up or moves down the heap. As a generalization, let us assume that the new value is y and the replaced value is x. If x is equal to y, the heapify operation does not need to do anything. If x is less than y, the heapify operation needs to worry about only the subtree rooted at the replacement node because y can only flow down the min-heap during the heapify operation. Similarly, if x is greater than y, the value y can only move up the min-heap toward the root following the path connecting the root to the replacement node. Therefore, in all the cases, the number of element swaps is at most the depth of the entire heap. While moving down the heap, every step requires reading out the two children of an element, comparing them with the element, and swapping the element with the child holding the smaller value (multiplexing driven by a comparison outcome). Organizing the two children nodes $2i$ and $2i+1$ of parent i in a single row allows us to read out both of them with just one read port. Swapping two elements requires two write ports. We assume the existence of three byte-comparators so that three pairs of priority values can be compared concurrently (pairwise among the parent and the two children). While moving up the heap, every step requires reading out the parent of an element, comparing the parent with the element, and performing a swap, if needed. Therefore, the critical path of each step in all the cases is $r + c + w$ where the read latency is r, the write latency is w, and the comparison and multiplexing latency taken together is c. Thus, the total latency of the algorithm is at most $(r + c + w)\log(k)$ time units plus the latency of one extra comparison (between x and y) at the beginning.

Due to high latency of the priority queue operations (proportional to depth), efficient handling of bursty cache miss requests induced by phase transitions requires pipelining the heap control. Our design is influenced by the pipelined heap of [9], where processing at each level of the heap is a separate pipeline stage, and data flow in the pipeline is from root level to the leaf level of the heap. Nodes at each level are stored in a separate RAM bank. Each stage is itself a micro-pipeline of read, compare, and write operations involving the incoming data from the previous stage and the nodes at that level. Note that, only one heapify operation is in progress in a stage at any point in time. However, a node in a stage can be overwritten with a value determined by the comparison outcome of the heapify operation in the next level. In case of data dependencies, these are resolved with short bypass busses between consecutive levels. As a result, one read port and two write ports are sufficient for each RAM bank. The pipeline depth is roughly $3\log(k)$ with stage latency equal to $\max(r, c, w)$. The overall cycle time of the pipelined priority queue is determined by $\max(r, c, w)$ of the last level, since it will have the largest RAM bank.

In our generalized replacement algorithm, the information may move up or down depending on the situation. Even worse, the operations can originate from arbitrary nodes of the heap. This is remarkably different from the pipelined operations supported in [9]. A replacement at the root does not require any special handling compared to what has already been implemented in [9]. Our implementation of the pipeline to handle the cases in Figure 6 are shown in Figures 7(a) and 7(b). In each case the operation starts at the root with inputs being the new value and the replacement node index. The new value is compared against the root and the bigger one moves down the heap participating in the conventional pipelined read, compare, write operations. At each level the controller computes the subtree (from the replacement node index) along which the information should be passed on so that the replacement node can be reached eventually. Once the replacement node is reached, the value to be replaced is overwritten by the pri-

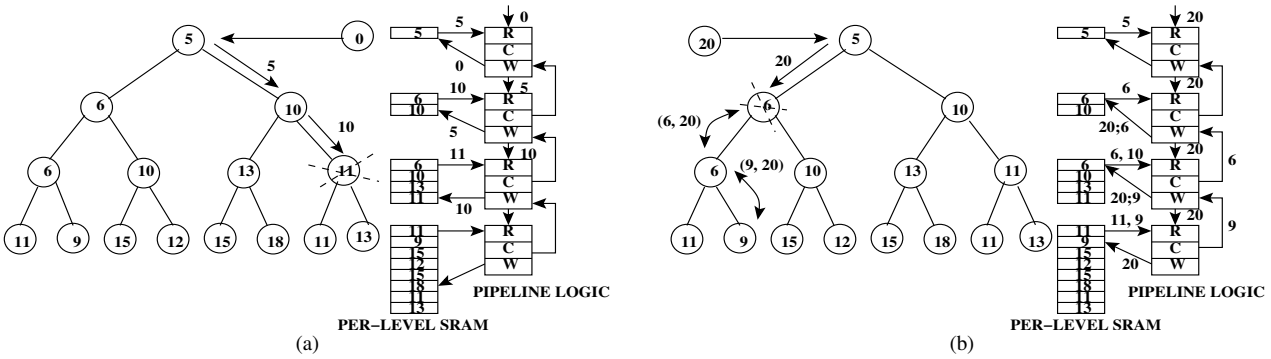

Figure 7. (a) Pipelined insertion of a zero value at an arbitrary position. (b) Pipelined replacement of a value at an arbitrary position. In both cases the 12-stage pipe featuring four blocks of local read-compare-write pipe is shown on the right. Also, the per-level RAMs communicating with the pipeline through read/write port connections are shown. The values read/written are shown on the respective port connections. How the comparison logic of a level controls the write logic of the previous level is also shown. On a write port connection "a;b" means that the value a is written first and later overwritten by the value b communicated by the comparison logic of the next level.

ority value that trickled down the pipe up to that point. Note that in Figure 7(b), the value that trickles down to the replacement node is always the inserted value. Also, in this case the write logic never gets invoked up to this point. Once the replacement node is reached, the operation shown in Figure 7(a) completes (subsequent pipe stages have nothing to do), while the one in Figure 7(b) continues exactly as in the heapify operation supported in [9]. The end-result is that we can initiate a heap replacement every $\max(r, c, w)$ time units.

2.3. Victim File Array

The victim file (VF) is responsible for holding the high priority blocks evicted from the conventional part of the L2 cache. It needs to support three operations. First, it must respond quickly to an L1 cache miss request. Second, in case of a hit in the VF, the block must be de-allocated and sent to the conventional L2 cache. Note that this is different from the conventional victim buffers where usually a swap is performed between the main cache and the victim buffer. However, in our case an allocation in the VF must get approval from the priority queue logic. This leads us to the third operation of the VF: It must be able to allocate a cache block by replacing the lowest-priority block currently in the VF, irrespective of the addresses of these two blocks. This operation requires the VF to be equivalent to a fully associative cache so that an arbitrary block of choice can be replaced. Clearly, a fully associative cache of large size that we are targeting would be too slow and power-hungry to be of any practical use.

We propose to organize the VF as a direct-mapped hash table, while chaining the tags mapping to the same index in the form of a variable-length doubly-linked list. An index is computed just as it would be done in a direct-mapped cache of the same size. The head of a tag list is always the entry corresponding to that index. A head bit (H) per tag, together with the valid bit, indicates whether a valid list exists for the corresponding index. A tail bit (T) per tag is used to mark the end of a tag list. For each VF entry, a pointer to its corresponding priority queue entry is maintained. This pointer RAM will be referred to as the HPTR RAM. The HPTR entry i is initialized to i much like the VPTR entries in the priority queue. We show the high-level organization of the VF in Figure 8. In the following we detail the operations mentioned above.

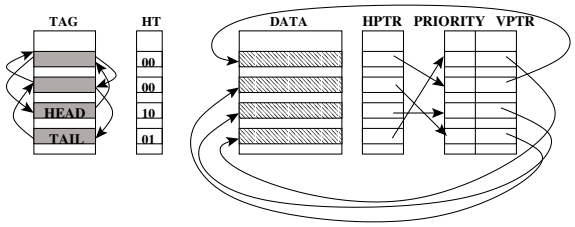

Figure 8. Victim file architecture. A tag list is also shown.

2.3.1 VF Hit/Miss Check and De-allocation

On an incoming L1 miss request, if a valid list head exists at the indexed location, a list walk is initiated by reading one tag entry at a time and following the downstream pointer until either a hit is encountered or the tail is reached signifying a miss; otherwise a VF miss is flagged immediately. On a VF hit, the tag is de-linked from its list. If the tag is a head tag, the second tag (if any) in the list is first migrated to the head location, and then the second tag is de-linked. This tag migration requires tag and data movement in the VF tag and data RAMs, a swap of the HPTR pointers of the two entries, and the corresponding updates in the VPTR entries. The frequency of this case involving tag/data migration is presented in Section 4.1. At the end, a zero insertion is initiated in the priority queue at the min-heap location obtained from the HPTR entry of the de-allocated block. During the heapify operation, the VPTR and HPTR entries are updated concurrently with every swap/update in the priority queue. We offer more details on the HPTR update in Section 2.3.4.

The performance-critical metric of the VF is the number of tag accesses needed before the hit/miss signal can be made available. In Section 4.1 we show that this number is quite small even when no limit is imposed on the tag list length.

2.3.2 Insertion into VF

On a VF allocation, typically a higher priority new block replaces the minimum-priority block in the VF, whose VF index is determined from the VPTR field of the root in the min-heap. However, if the location indexed by the new block is already empty (no tag list exists), the new tag and data are filled in and there is no need to de-allocate the victim block. In parallel, the heapify operation is initiated to replace the priority of the min-heap node identified from the HPTR entry. Note that this case may require a priority insertion at a non-root heap node.

In the other cases, the victim block is de-allocated to be used in the allocation of the new block. For this, the tag to be replaced is de-linked from its list (if any). This may require block migration as described in VF de-allocation on a hit if it is a head tag. Next, the new tag must be inserted into its list. If the list is present, the reclaimed tag is linked into the second position of the list and filled with the new tag. The corresponding data RAM entry is filled with the data block. Concurrently, the insertion of the new priority value is executed in the min-heap. The HPTR and VPTR entries are also updated during this heapify operation. Otherwise, if the list is not present but there is a valid tag that belongs to another list at the indexed location (i.e. H bit is reset), this location is first freed by migrating the block to the reclaimed entry and the corresponding tag is re-linked with its list. Finally, the new tag and data are filled into the freed location. Concurrently, heapify operation is performed in the priority queue.

In our design, we impose a limit on the maximum list length to offer a latency guarantee for snoop/intervention responses to the external interface in shared memory multiprocessor systems if Scavenger is used as the last-level cache. We make this limit equal to eight. For each tag, there is a pointer to the head of the list it belongs to. These pointers are maintained in a HEADPTR RAM and are updated only at the time of allocating new tags. Corresponding to each head tag, there is a count (maintained in a COUNT RAM) which records the length of the list headed by that tag. If an allocation into the VF finds that the list length has already reached eight, instead of replacing the global lowest priority block as dictated by the heap, it replaces the second element in its tag list, thereby keeping the list length within eight. Compared to unlimited list length, we found that this algorithm suffers from a maximum performance loss of 0.1% across the aforementioned sixteen applications. This is expected, given that the list length seldom grows beyond eight (Section 4.1).

2.3.3 Random Mode of Operation

The L2 cache miss stream may occasionally stop exhibiting repeated block addresses. As a result, none of the evicted blocks may get inserted into the VF because their repetition frequency (typically one for non-repeated evictions) may fall below the current minimum. One such scenario is when an application moves on to a new phase of computation, even the maximum miss frequency in that phase may fall below the minimum miss frequency in the earlier phase. To solve this problem, we continuously monitor the health of the VF. Over a window of 8,192 misses in the conventional L2 cache, if both the number of hits in the VF and the number of blocks accepted by the min-heap for insertion into the VF fall below a threshold of 64, our algorithm switches to a *random* mode of operation. During this mode, every block evicted from the conventional L2 cache is put into the VF irrespective of its priority. The block to be replaced from the VF is decided by a global random replacement policy (as opposed to the root of the min-heap). As soon as the number of blocks evicted from the conventional part of the L2 cache with priority more than the maximum priority seen during the random mode[4] crosses a threshold of 4,096 (signifying a good amount of repetitions in the miss address stream), the hardware comes out of the random mode. Note that at any point in time, the priority queue continues to maintain the priorities of the blocks residing in the VF, as usual. Out of the 16 SPEC 2000 applications that we consider in this paper, 171.swim spends a significant portion of its execution time in the random mode.

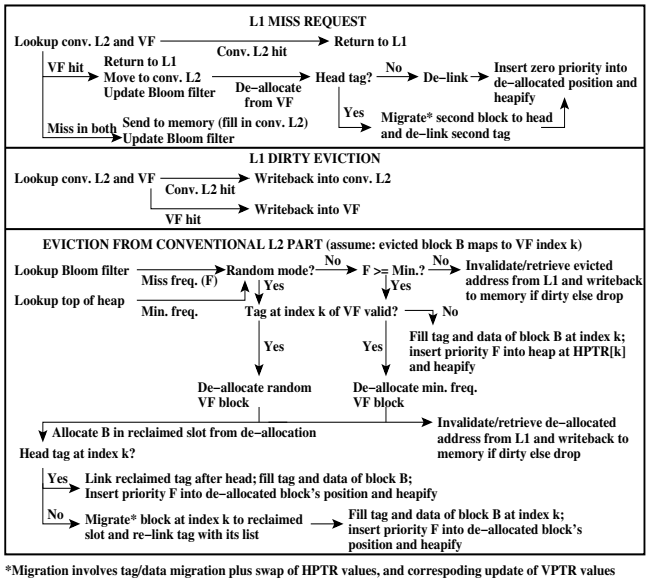

Figure 9. Flow of actions in Scavenger.

2.3.4 Physical Implementation of VF

Before concluding this section, we briefly mention the physical implementation of the VF. The upstream and downstream pointers are maintained in UPTR and DPTR RAMs, each entry of which is $\lceil \log(k) \rceil$ bits wide where k is the number of entries in the VF. Each entry of the HPTR and HEADPTR RAMs also has the same width. The COUNT RAM has 3-bit entries (records the length of a list without the head). All the RAMs have k entries. We note that banking any of these RAMs (including the VF tag and data RAMs) does not pose any problem. However, all the RAMs should be banked in a similar way to have uniform indexing. In a banked VF tag RAM, a tag list may span several banks. Since at any point in time only one tag is accessed, a single read and a single write port per bank continue to be sufficient. For swapping the contents of two entries in HPTR during a heapify operation, we do not need write ports because the contents to be swapped are related by parent-child relationship (i and $2i$ or i and $2i + 1$). As a result, for example, if entry A (containing i) is to be swapped with entry B (containing $2i + 1$), the contents of entry A are shifted left and a one is ORed at the LSB position while the contents of entry B are shifted right.

[4] A single register is required to maintain this maximum.

Table 1. Simulated architecture.

Processor	
Frequency	4GHz
Fetch/issue/commit width	4/4/6
In-flight branches	24
Branch predictor	32,768-entry GAg+Bimodal 15-bit GHR
RAS entries	32
Inst. window size	48 int.+mem., 32 FP
ROB entries	128
Integer/FP registers	160/160
Integer FUs	4 ALU, 2 Mult., 2 Div.
Floating-point FUs	4 Add., 2 Mult., 2 Div.
Load/Store units	2
Load/Store queue entries	48/32
Hardware prefetcher (between L2 cache and main memory)	16-stream stride pref., maximum stride 256B
Memory subsystem	
MHT entries	16 L1, 16 L2
L1 i-cache & d-cache	32kB/4-way/64B/LRU
L2 (unified) cache	1MB/8-way/64B/LRU
Cache ports	1 iL1, 2 dL1, 1 L2
L1 and L2 cache hit latency	0.75ns, 2.25ns
System bus bandwidth	8 GB/s
Memory latency	121ns (load-to-use)
Scavenger-specific	
Conventional L2 part	512kB/8-way/64B/LRU
Victim file	8,192 blocks×64B/blk.
Conventional L2 hit latency	2ns
Victim file tag, data latency	0.5ns, 0.75ns
Priority queue size, latency	8,192 entries, 0.5ns
Misc. RAMs size, latency	8,192 entries, 0.5ns each
Bloom filter size	33.3kB, 5-way banked
Bloom filter latency	0.5ns

Each HPTR row is internally equipped with a one-bit up/down shifter. The up/down shift control is generated by the priority queue logic. Note that HPTR still requires the adequate number of row address decoders to enable the shifters in the appropriate rows. We note that the embedded shift-OR logic and the extra row decoders do add to the area overhead of Scavenger. Finally, HPTR needs only one read port to read out the starting priority queue entry index at the beginning of a heapify operation caused by a VF allocation or de-allocation. HPTR requires one write port for initialization and for the swap operation during a block migration (two writes are assumed to proceed serially).

We summarize the flow of actions in Scavenger in Figure 9 for an inclusive cache hierarchy. The actions correspond to the three major commands that the Scavenger's L2 cache controller receives. The actions taken on external snoop commands are omitted from this figure. This case will have a lot in common with the L1 miss request handling. Also, the actions regarding COUNT and HEADPTR RAMs are not included for brevity.

3. Simulation Environment

3.1. Architecture

We use the SESC simulation environment [16] to evaluate our proposal. The baseline processor comprises a four-issue out-of-order CPU clocked at 4GHz, and with two levels of on-chip caches. We assume a 65nm process, and use CACTI 4.2 [7] to derive latency and silicon area for L1 and L2 caches. In all configurations, we assume that tag and data arrays are accessed in parallel in the L1 cache, but looked up serially in the L2 cache. We also assume an aggressive multi-stream stride prefetcher operating between the L2 cache and main memory. Table 1 summarizes the baseline architecture, which employs a 1MB 8-way set-associative L2 cache. We evaluate the following setups:

Scavenger. We evaluate two Scavenger configurations, both comprising a 512kB 8-way set-associative cache and an 8,192-entry victim file (VF). These configurations differ in that one employs a standard balanced Bloom filter (*StBF*) that divides a 26-bit L2 cache block address into 8-, 9-, and 9-bit partitions (in decreasing order of significance), while the other one uses a skewed Bloom filter (*SkBF*) as discussed in Section 2.1.

Using CACTI, we derive latencies for the 512kB conventional cache, as well as the VF. When optimized for performance, the 512kB cache can be accessed in 2ns. For the VF, we obtain the latencies of tag and data RAMs by modeling a 512kB direct-mapped cache (recall that, at any point in time, at most one tag is accessed in the VF) with one exclusive read port, one exclusive write port, sequential tag and data accesses, and two internal subbanks. We model one exclusive read port and one exclusive write port in the HPTR, UPTR, DPTR, HEADPTR, and COUNT RAMs. Since CACTI does not model block sizes of less than eight bytes, we assume that each RAM is designed such that multiple entries are combined into a single row to bring the row size to at least eight bytes. (Note that the total size of the RAM in bytes remains unchanged.) The priority queue has one exclusive read port and two exclusive write ports. We find that all these RAM structures can be accessed within 0.5ns. The largest bank of the Bloom filter has 32,768 entries and determines the critical path. We find that, with one exclusive read port and one exclusive write port and four internal subbanks, this bank can also be accessed within 0.5ns.

We also calculate Scavenger's area overhead. Table 2 lists component areas for baseline and Scavenger configurations, extracted using CACTI. We assume a 32-bit physical address. Each cache block is assumed to have three coherence states: M, E, and S (invalid state corresponds to M=E=S=0). We ignore the bits in the extended L2 tag needed for supporting a virtually indexed L1 cache [21]. CACTI yields a 7.8% area overhead compared to the baseline L2 cache (16.75mm^2 vs. 15.54mm^2).

16-Way. We explore an enhanced cache configuration with twice the associativity of the baseline. Using CACTI, we optimize this design for performance, achieving an access time of 2.75ns. According to CACTI's figures, this incurs a 70% area overhead over the 8-way set-associative baseline (26.4mm^2 vs. 15.54mm^2), which greatly exceeds Scavenger's area overhead. Our goal is to determine whether "giving back" Scavenger's area overhead to the baseline configuration in this way represents a competitive alternative.

512kB-FA-VC. We also explore an alternative configuration comprising a 512kB 8-way set-associative cache (same as Scavenger's), plus a 512kB fully associative victim cache (same storage as Scavenger's victim file) with random replacement policy. Although impractical in terms of area and energy consumption, the 512kB fully associative victim cache can be

Table 2. Storage and area requirement of baseline and Scavenger

Config.	Components	Details	Subtotal	Total
Baseline	1MB 8-way	Block size: 512 bits; tag: 15 bits; LRU state: 3 bits; M, E, S states; number of blocks: 16,384; 128-bit output.	15.54 mm^2	**15.54 mm^2**
Scavenger	512kB 8-way	Block size: 512 bits; tag: 16 bits; LRU state: 3 bits; M, E, S states; number of blocks: 8,192; 128-bit output.	10.02 mm^2	**16.75 mm^2**
	Victim file	Block size: 512 bits; tag: 13 bits; M, E, S, H, T states; number of blocks: 8,192; 128-bit output.	3.69 mm^2	
	Auxiliary RAMs	Row size in bits: UPTR 13, DPTR 13, HPTR 13, HEADPTR 13, COUNT 3, priority queue 8, VPTR 13; number of rows in each RAM: 8,192.	2.65 mm^2	
	Bloom filter	Number of counters in five banks: 8; 256; 32,768; 1,024; 64; counter size: 8 bits.	0.39 mm^2	

useful to understand the importance of Scavenger's replacement policy in its victim file. As in the case of the 1MB 16-way configuration, we use CACTI to pick the configuration that yields highest performance, obtaining an access time of 3.5ns.

3.2. Applications

We use sixteen applications from the SPEC 2000 benchmark suite (the remaining applications do not work with this simulation infrastructure at this time). All the applications are compiled for MIPS ISA and run on the *ref* inputs for a representative sample of one billion dynamic instructions extracted with SimPoint toolset [18]. We do not use cache warm-up while fast-forwarding, so that the effect of cold misses is also accounted for.

4. Simulation Results

In this section we present our simulation results. We begin the discussion with an analysis of the victim file performance in terms of tag access count per L1 miss request and block migration frequency. Then, we present performance results, comparing Scavenger with the other L2 cache organizations described in Section 3.1. We also present a study of Scavenger's energy per instruction (EPI) overhead. Finally, we conduct a comparison of Scavenger against recent related work of similar objectives.

4.1. Victim File Characteristics

In this section we characterize two performance-critical attributes of the VF. Table 3 shows the average, maximum, and the most frequent (mode or common case) number of tag accesses per L1 miss request. The percentage of requests requiring at most three tag accesses is also shown. The statistics are very encouraging: The average number of tag accesses is below 1.5 for 14 applications. Moreover, for 15 of the 16 applications, the mode requires a single tag lookup; note that, in this case, the VF enjoys the hit latency of an equally-sized direct-mapped cache. Furthermore, for 15 of the 16 applications, more than 90% of requests can be fulfilled with at most three tag accesses. One of the reasons for observing lists of small length in the VF is that it has three extra bits for indexing, compared to an equally-sized 8-way set-associative organization (13 vs. 10 bits). These extra index bits help spread out blocks that would otherwise map to the same set in the set-associative organization.

Table 3. Mean, maximum, and mode VF tag RAM access count per L1 cache miss request, percentage of requests requiring at most three accesses, and percentage of VF allocations/de-allocations requiring a block migration

Application	VF tag accesses per request				Block migr. (%)
	Mean	Max.	Mode	≤ 3	
164.gzip	1.31	6	1	99.8%	22.5
168.wupwise	1.38	10	1	91.4%	32.3
171.swim	1.42	7	1	99.5%	23.4
173.applu	1.21	6	1	99.9%	24.7
175.vpr	1.54	5	1	99.6%	0.6
176.gcc	1.39	9	1	99.5%	11.9
177.mesa	1.16	6	1	99.6%	24.7
179.art	1.39	15	1	99.9%	8.0
181.mcf	1.25	42	1	99.6%	15.3
183.equake	1.34	7	1	99.5%	24.7
186.crafty	1.47	5	1	99.4%	2.7
188.ammp	2.72	10	3	76.1%	2.2
253.perl	1.11	5	1	99.6%	5.9
256.bzip2	1.32	12	1	98.8%	14.3
300.twolf	1.48	16	1	99.8%	0.7
301.apsi	1.31	7	1	99.6%	23.7

Table 3 also shows the percentage of VF allocations and de-allocations requiring a block migration. With the exception of 168.wupwise, at most 25% of the allocations/de-allocations require a VF block migration. Thus, we expect the impact of block migration on the overall performance to be modest.

4.2. Performance Comparison

Among the SPEC2000 applications used in this paper, we consider an application to be memory-bound if the stall time reported in Figure 1 is 30% (15%) or higher for a 512kB (1MB) 8-way set-associative L2 cache configuration. Figure 10 presents speedups for these applications in the four configurations under study (Section 3), relative to the 1MB 8-way set-associative baseline. Nine applications fall under this category. The speedups for the other seven applications are shown in Figure 11 (but the geometric mean in this figure includes all sixteen applications).

The results show that Scavenger-SkBF achieves significant speedups for the majority of memory-bound applications, with a peak speedup of 63% (181.mcf) and an average (geometric mean) of 14.2%. On the other hand, Scavenger-StBF achieves much lower speedup–3.3% on average. In fact, in two applications (175.vpr and 300.twolf), Scavenger-StBF is slower than

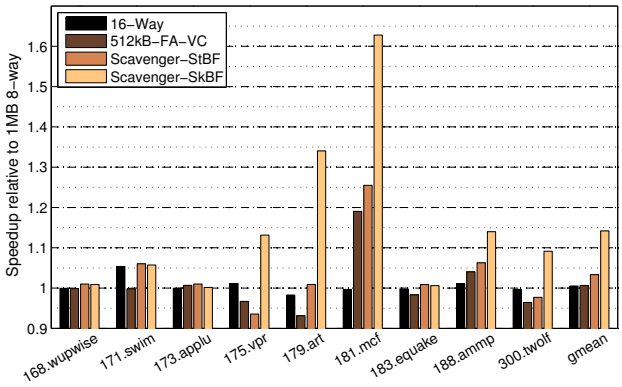

Figure 10. Speedup for memory-bound applications.

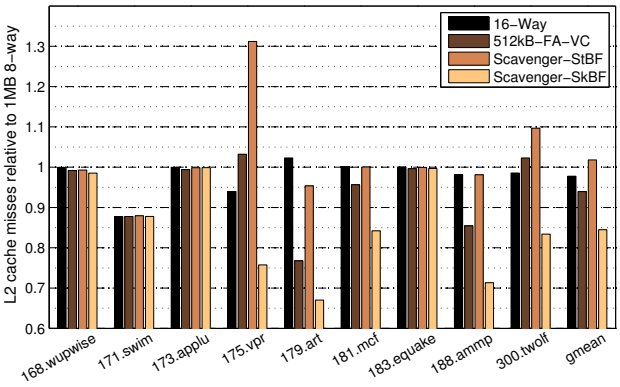

Figure 12. Normalized L2 cache misses for memory-bound applications.

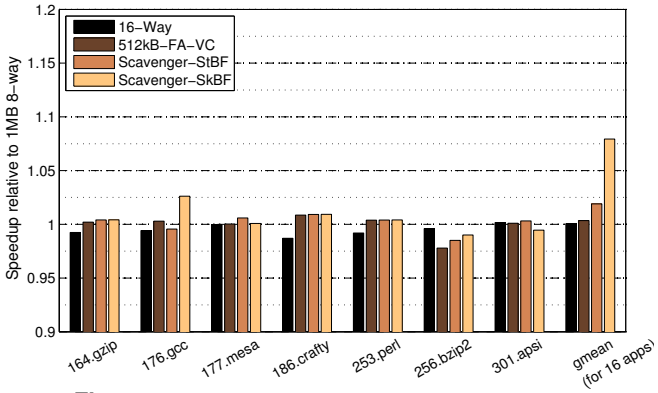

Figure 11. Speedup for non-memory-bound applications.

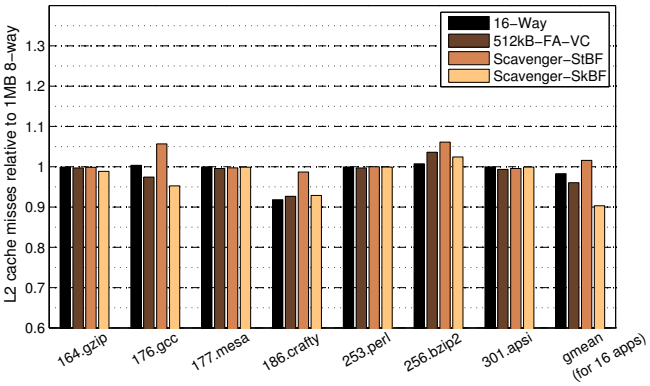

Figure 13. Normalized L2 cache misses for non-memory-bound applications.

the baseline. We empirically observe that, due to a high volume of false-positives in the standard Bloom filter, the miss frequency estimate has a high amount of error, leading to wrong selection of cache blocks to be retained in the VF.

When we increase the associativity of the baseline to 16 (16-Way), the results show that performance does not improve in most cases; only 171.swim enjoys a significant performance improvement (still under 10%). In fact, performance degrades in 179.art compared to the baseline. The primary reason for this is that increasing the associativity beyond a certain point brings only marginal improvements in miss rate while slowing down *all* cache hits.

Interestingly also, 512kB-FA-VC delivers significant gain only for 181.mcf, while four applications (175.vpr, 179.art, 183.equake, and 300.twolf) experience slowdowns due to the victim cache's high hit time. This brings out the importance of our victim file architecture, which is functionally equivalent to a fully associative victim cache, but offers a much lower hit time on average.

For 168.wupwise, 173.applu, and 183.equake, none of the L2 cache configurations provides any noticeable benefit, while these application suffer significantly from L2 misses (Figure 1). 168.wupwise suffers from a high volume of cold misses which cannot be addressed by any of the cache optimization techniques explored in this paper. The other two applications share the common characteristic that they have the largest eviction-reuse distances for L2 cache miss addresses (Figure 2). As a result, in the experiments they show relatively poor caching behavior that not even our victim file can capture sufficiently.

Finally, the results for the non-memory-bound applications are unremarkable as expected, except for the (important) fact that performance never degrades significantly with the use of Scavenger. With these applications factored in, Scavenger-SkBF's average speedup across all 16 applications is 8%.

To understand the performance results better, Figures 12 and 13 present the number of L2 cache misses in the four different architectures, normalized to the baseline. On average, for the memory-bound applications, Scavenger-SkBF saves about 15.5% of misses with respect to the baseline, whereas Scavenger-StBF *suffers* from 1.8% *extra* misses on average (31.2% for 175.vpr) compared to the baseline. As already discussed, this stems from high false-positive rates in the standard Bloom filter, leading to inaccurate victim caching. Recall, however, that Scavenger-StBF delivers an average 3.3% *speedup*. The primary reason for this apparent anomaly is that a large number of hits in Scavenger-StBF are faster than hits in the baseline. (From this point on, unless otherwise noted, we will refer to Scavenger-SkBF simply as Scavenger.)

On the other hand, the 1 MB 16-way set-associative configuration (16-Way) saves about 2.2% of misses on the memory-bound applications. This essentially means that, without a better replacement policy, increasing the associativity in this way is not helpful for this set of workloads.

Finally, 512kB-FA-VC saves about 6% of misses when using a random replacement policy on the memory-bound appli-

Table 4. Hit rate (HR) and miss count (MC) in Scavenger-SkBF

Application	VF HR	Conv. HR	Total HR	MC
164.gzip	0.00	0.65	0.65	0.20M
168.wupwise	0.01	0.15	0.16	1.65M
171.swim	0.40	0.30	0.70	19.28M
173.applu	0.00	0.03	0.03	7.98M
175.vpr	0.18	0.58	0.76	0.60M
176.gcc	0.03	0.57	0.60	1.09M
177.mesa	0.00	0.26	0.26	0.49M
179.art	0.28	0.13	0.41	48.82M
181.mcf	0.14	0.06	0.20	67.27M
183.equake	0.01	0.03	0.04	22.40M
186.crafty	0.01	0.96	0.97	0.04M
188.ammp	0.15	0.60	0.75	1.37M
253.perl	0.00	0.84	0.84	0.02M
256.bzip2	0.04	0.55	0.59	1.08M
300.twolf	0.21	0.44	0.65	2.47M
301.apsi	0.03	0.17	0.20	2.35M

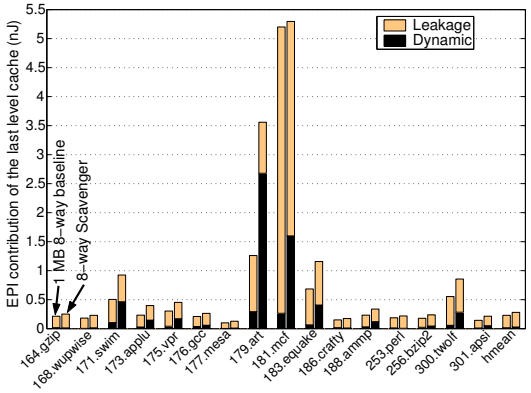

Figure 14. Comparison of EPI contributions from the last-level cache in the baseline and Scavenger.

cations. This alone justifies our global priority-driven insertion policy in the victim file, which achieves 15.5% savings in misses. (Recall also that the hit times are slower in 512kB-FA-VC's victim cache.)

Table 4 shows the local hit rate (number of hits over number of L1 misses) in the VF and the conventional L2 part for all the applications in Scavenger. It also shows the absolute number of L2 misses. For most memory-bound applications, the VF satisfies a significant number of L1 miss requests. In 171.swim, 179.art, and 181.mcf, the VF hit rate even exceeds that of the conventional half.

In summary, these results clearly establish the two major contributions of our proposal, namely, a selective victim caching scheme based on global priority determined by miss frequency, and a low-latency direct-mapped organization of a large fully associative victim file.

4.3. Impact on Energy per Instruction

In this section, we present the energy overhead incurred by Scavenger over the baseline. Our dynamic energy model for caches is derived from Wattch [3], but with considerably improved decoder circuitry and tag comparator. Also, we have modified CACTI [7] to compute the subarrays in the tag and data RAMs so that the energy-delay-squared is optimized. The dynamic power model used in this paper has been verified against the published results of the Alpha 21264 [5]. Our modeled peak power comes within 3% of the published results. We have developed a detailed leakage power model for SRAMs using subthreshold and gate leakage currents obtained from HSPICE simulations. Our SRAM schematic models the cell and all the peripheral circuitry including the sense amplifiers, the sense isolation circuitry, the write circuitry, and the precharge circuitry. We appropriately extrapolate these leakage components from smaller SRAMs to derive the leakage power of bigger SRAMs. In all these models, we assume a V_{dd} of 1.1V and a V_t of 0.18V. The rest of the parameters are scaled down appropriately to 65nm node from the Wattch distribution. In the following results, we do not include optimizations such as sleep transistors, drowsy cells, or power gating.

Figure 14 presents the EPI comparison between the baseline and Scavenger. The EPI of Scavenger includes the contributions from the conventional 512kB 8-way L2 cache, the 8,192-entry VF, all the associated SRAMs, and the Bloom filter. We break down the EPI contribution into dynamic and leakage components. As expected, across the board, Scavenger dissipates more EPI. On average (harmonic mean), this increase is relatively modest in absolute terms: 0.02nJ of dynamic and 0.21nJ of static EPI for the baseline, vs. 0.03nJ and 0.25nJ for Scavenger, respectively. Only for one application (179.art), the average EPI in Scavenger increases by more than 2nJ; in all other applications, it increases by at most 0.5nJ. Relatively speaking, in most of the applications, the dynamic EPI increases more significantly. The most noticeable increase in dynamic EPI is experienced by the memory-bound applications. We empirically observe that this is largely because these applications enjoy a relatively large number of hits in the VF, and a hit in the VF involves copying a cache block from the VF to the conventional L2 part and a re-adjustment in the contents of the priority queue. We also notice that, although 173.applu and 183.equake do not benefit from too many hits in the VF, they consume extra dynamic energy due to futile activity by Scavenger. These two applications have reasonable amount of address repetition (Figure 1), but fail to take advantage of that fact. In 181.mcf, the increase in dynamic EPI gets almost compensated by a decrease in static EPI. Scavenger dissipates lower leakage energy due to dramatically reduced execution time in this application. Finally, keep in mind that the energy savings in the DRAM, memory controller, and front-side bus due to fewer L2 cache misses in Scavenger may well outweigh this small increase in L2 cache EPI. Moreover, the reduced execution time is likely to decrease the total energy dissipation in the core pipeline.

4.4. Comparison with Recent Proposals

In this section we compare Scavenger with two recent related proposals, namely, dynamic insertion policy (DIP) [15] and V-way cache [14], in terms of L2 miss rates. DIP improves cache performance by deciding where to insert a newly allocated block. In traditional LRU algorithms, a new block is always made the MRU within a set. DIP explores the potential of dynamically choosing between the LRU and MRU positions for a new incoming block. A block inserted in the LRU position is promoted to the MRU position only after it is

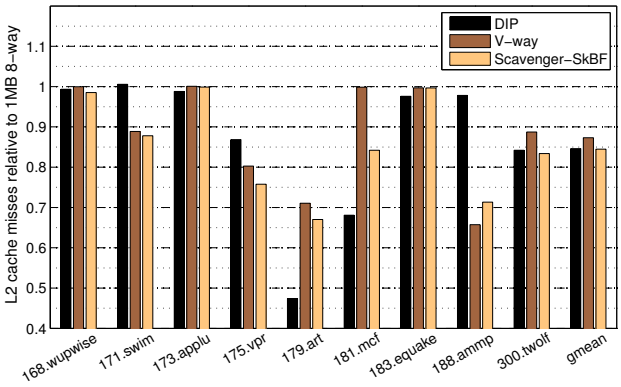

Figure 15. Comparison of dynamic insertion policy, V-way cache, and Scavenger in terms of the total number of last-level cache misses relative to 1MB 8-way baseline.

accessed once more. If the working set is much larger than the cache and exhibits a cyclic access pattern, such a policy succeeds in retaining some part of the working set in the cache. We simulate DIP in a 1MB 8-way L2 cache with 32 dedicated sets used for decision making and an epsilon of 1/32 for bimodal insertion (please refer to [15] for explanation).

The V-way cache doubles the tag store compared to a conventional set associative cache while keeping the decoupled data store size unchanged. As a result, the tag store gains one extra index bit allowing the conflicting tags to get distributed over two sets. Each tag maintains a pointer to the corresponding data block. Since the number of tags is twice that of the data blocks, within a set with a very high probability an invalid tag can be found. When replacing an invalid tag, the V-way cache implements a reuse-based global data block replacement policy where each data block maintains a small reuse counter. The block with the minimum reuse count is found by sequentially examining the counters. The authors impose an upper bound of five cycles for this search. We implement V-way in a 1MB 8-way cache.

Figure 15 shows a comparison of L2 cache miss rates among DIP, V-way cache, and Scavenger, relative to the 1MB 8-way baseline on the nine memory-bound applications. Scavenger significantly outperforms DIP in three applications (171.swim, 175.vpr, and 188.ammp) while loses to DIP in two (179.art and 181.mcf). These two applications occasionally operate on large working sets with significantly high reuse distance (Figure 2). Scavenger is able to retain only 512kB of the working set in the victim file, and it takes time to learn the repetition behavior as the top frequently missing blocks emerge gradually. On the other hand, DIP can retain almost 1MB of the working set, does not require any learning time, and as a result, beats Scavenger by a large margin. To understand this performance difference better, we simulated Scavenger with a perfect Bloom filter. This brought down the miss rates of 179.art and 181.mcf to 0.54 and 0.79 relative to the baseline, respectively. This result points to an inaccuracy of the Bloom filter when handling very large working sets. Next, to factor out the effect of the victim file's capacity, we simulated Scavenger with a perfect Bloom filter and the entire 1MB storage devoted to the victim file. This organization further brought down the relative miss rates of 179.art and 181.mcf to 0.42 and 0.77, respectively. Fundamentally, DIP attacks a capacity problem where the working set size exceeds the last-level cache size. Scavenger does not distinguish between capacity and conflict misses, but learns to retain the most frequently missing blocks. DIP and Scavenger will deliver similar performance if the most frequently missing blocks in a working set happen to belong to the part of the working set retained by DIP; otherwise, Scavenger will tend to deliver better performance. In the situation where all the blocks in a large working set miss with the same frequency, Scavenger will assign equal priority to all of them, and may fail to deliver performance similar to DIP depending on whether a random mode switch takes place, how effective this switch to the random mode is, and whether occasional hits in the VF re-arrange the priority values in the heap. Nonetheless, even in this situation, Scavenger will still be able retain a large portion of the working set.

The V-way cache is better than Scavenger in only one application (188.ammp). On average over these nine applications, relative to a 1MB 8-way baseline, Scavenger saves 15.5% of the L2 cache misses, the V-way cache saves 12.7% of the misses, and DIP saves 15.4% of the misses.

5. Related Work

Efficient cache architecture design has received decades of attention from the research community because of its critical role in determining the end-performance of computer systems. Broadly speaking, the innovations in this area can be classified into four categories, namely, smart indexing schemes, victim caching, novel replacement policies, and new overall organizations. In the following we focus on some of the studies done on victim caching and an organization called indirect index cache (IIC) [6] that shares some similarities with our victim file proposal. To avoid repetition, we do not discuss DIP and V-way cache here.

Small fully associative victim caches were introduced in [10] and have been used in several commercial processors. One relatively recent study exploring selective victim caching for L1 caches [8] detects blocks suffering from conflict misses by examining the dead time of the evicted blocks. In this paper we introduce a new organization of a large and fast victim file that can capture the large-scale temporal behavior present in the miss address stream of the last-level caches.

The IIC organization involves decoupled tag and data stores and a generational global replacement policy. The tags are organized into a primary 4-way associative hash table and a secondary direct-mapped table. On a primary table miss, the secondary table is used to walk the hash collision chain. The generational replacement policy maintains multiple queues of tags, each queue having a bulk priority for all the tags residing in that queue. If a tag is not accessed for a long period of time, it is gradually demoted to the lowest priority queue. A replacement tag is always selected from the lowest priority queue. While IIC relies on multi-level queues to capture temporal accesses to L2 cache blocks, Scavenger directly examines the L2 cache miss address stream and prioritizes the blocks that suffer from most misses for insertion into the victim file.

6. Summary and Possible Extensions

In this paper we propose a new last-level cache organization, namely Scavenger, which divides the total cache storage into a conventional cache and a victim file (VF). The VF retains only high priority cache blocks that have most frequently missed in the past and therefore are more likely to be used in the future.

Scavenger employs three new components: A Bloom filter that tracks the miss counts to cache block addresses, used as block priorities; a victim file that holds the high priority blocks evicted from the conventional part, and a fast pipelined priority queue that maintains the priorities of the VF blocks. A new block is eligible for allocation only if its priority is higher than or equal to the current minimum in the VF. The VF is organized as a direct-mapped hash table, while providing fully-associative buffering through chaining the tags that map to the same index. Thus, it offers low latency and fewer conflicts at the same time. Finding a block in the VF requires only a few tag accesses (at most three in most cases).

A 512kB+512kB Scavenger organization employed as a last level L2 cache improves IPC of nine memory-bound SPEC 2000 applications by up to 63%, and on average 14.2%, compared to a conventional 1MB 8-way set-associative L2 cache, with an aggressive multi-stream stride prefetcher enabled in both configurations. Across a larger set of sixteen SPEC 2000 applications, the average IPC improvement is 8%. We also present detailed analysis of dynamic and static energy overheads of the proposed architecture.

There are several possible extensions of this work. Given the performance potential of Scavenger, more energy-conscious implementations of the algorithms are worth exploring. Improving the accuracy of the frequency estimator is another challenging issue that has potential to further increase the performance. Finally, extending the current algorithms to a multi-threaded setting, especially in the last-level shared cache of a chip-multiprocessor, is important.

Acknowledgments

We would like to thank the Research I Foundation of IIT Kanpur for helping initiate this collaborative effort between IIT Kanpur and Cornell University. We thank Vijay Degalahal for helping us with HSPICE, and Jugash Chandarlapati for developing the leakage energy model. We thank Andreas Moshovos and the anonymous reviewers for suggestions to improve the paper.

The Cornell effort of this work was supported in part by NSF awards CAREER CCF-0545995, CNS-0509404, and CNS-0429922; by an IBM Faculty Award; by two Intel graduate fellowships; and by gifts from Intel.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 423–434, December 2003.

[2] B. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. In *Communications of the ACM*, 13(7):422–426, July 1970.

[3] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-level Power Analysis and Optimizations. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 83–94, June 2000.

[4] S. Cohen and Y. Matias. Spectral Bloom Filters. In *Proceedings of the SIGMOD International Conference on Management of Data*, pages 241–252, June 2003.

[5] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor. In *Proceedings of the 35th Design Automation Conference*, pages 726–731, June 1998.

[6] E. G. Hallnor and S. K. Reinhardt. A Fully Associative Software-managed Cache Design. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 107–116, June 2000.

[7] HP Labs. CACTI 4.2. Available at http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html.

[8] Z. Hu, S. Kaxiras, and M. Martonosi. Timekeeping in the Memory System: Predicting and Optimizing Memory Behavior. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 209–220, May 2002.

[9] A. Ioannou and M. Katevenis. Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks. In *IEEE/ACM Transactions on Networking*, 15(2):450–461, April 2007.

[10] N. P. Jouppi. Improving Direct Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, June 1990.

[11] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 234–245, June 2005.

[12] A. Moshovos et al. JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, pages 85–96, January 2001.

[13] J-K. Peir et al. Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching. In *Proceedings of the 16th International Conference on Supercomputing*, pages 189–198, June 2002.

[14] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-way Cache: Demand-based Associativity via Global Replacement. In *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 544–555, June 2005.

[15] M. K. Qureshi et al. Adaptive Insertion Policies for High-Performance Caching. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 381–391, June 2007.

[16] J. Renau et al. SESC simulator. *http://sesc.sourceforge.net*, January 2005.

[17] E. Safi, A. Moshovos, and A. G. Veneris. L-CBF: A Low-power, Fast Counting Bloom Filter Architecture. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 250–255, October 2006.

[18] T. Sherwood et al. Automatically Characterizing Large Scale Program Behavior. In *Proceedings of the 10th International Conference on Architectural Support on Programming Languages and Operating Systems*, pages 45–57, October 2002.

[19] K. Strauss, X. Shen, and J. Torrellas. Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors. In *Proceedings of the 33rd International Symposium on Computer Architecture*, pages 327–338, June 2006.

[20] D. H. Woo et al. Reducing Energy of Virtual Cache Synonym Lookup using Bloom Filters. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 179–189, October 2006.

[21] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. In *IEEE Micro*, 16(2):28–40, April 1996.