Counting Stream Registers: An Efficient and Effective Snoop Filter Architecture

Aanjhan Ranganathan[1], Ali Galip Bayrak[2], Theo Kluter[3], Philip Brisk[4], Edoardo Charbon[5], Paolo Ienne[2]

[1]System Security Group
ETH Zürich
CH-8092 Zürich, Switzerland

[2]School of Computer and Communication Sciences
Ecole Polytechnique Fédérale de Lausanne (EPFL),
CH-1015 Lausanne, Switzerland

[3]Department of
Engineering and Information Technology
Bern University of Applied Sciences
CH-3012 Bern, Switzerland

[4]Department of
Computer Science and Engineering
University of California, Riverside
Riverside, CA, USA 92521

[5]Department of Micro-electronics
and Computer Engineering
Delft University of Technology
2628 CD Delft, The Netherlands

***Abstract*—We introduce a counting stream register snoop filter, which improves the performance of existing snoop filters based on stream registers. Over time, this class of snoop filters loses the ability to filter memory addresses that have been loaded, and then evicted, from the caches that are filtered; they include cache wrap detection logic, which resets the filter whenever the contents of the cache have been completely replaced. The counting stream register snoop filter introduced here replaces the cache wrap detection logic with a direct-mapped update unit and augments each stream register with a counter, which acts as a validity checker; loading new data into the cache increments the counter, while replacements, snoopy invalidations, and evictions decrement it. A cache wrap is detected whenever the counter reaches zero. Our experimental evaluation shows that the counting stream register snoop filter architecture improves the accuracy compared to traditional stream register snoop filters for representative embedded workloads.**

Keywords-snoopy coherence protocol, snoop filter, stream register, counting stream register

I. INTRODUCTION

Broadcast-based snoopy hardware coherence protocols play an important role in small-scale multiprocessor systems. In a write-invalidate snoopy coherence protocol, whenever a value is modified in one of the processor's caches, a bus transaction is initiated to signal the change to the other caches. Each cache controller in the system snoops the bus: if the requested line of data is present (a hit) in the respective cache, the controller takes the appropriate action, depending on the nature of the request, and the status bits associated with the line (for instance, it may invalidate the cache line); if the line is not present in the cache (a miss), then no action is taken. A significant number of snoops miss in most of the caches; taken in aggregation, these misses consume excessive energy.

A snoop filter is a small cache-like structure that is placed in front of the cache itself, but provides inexact hit/miss information [13]. A snoop lookup either guarantees that the requested line is not in the cache, or returns a maybe signal, indicating that the line may or may not be present, and thus forwards the request to a cache. A snoop filter lookup consumes significantly less energy than a cache lookup.

Each snoop lookup that results in a guarantee yields a net energy savings over a cache lookup; however, each snoop lookup that returns a maybe consumes more energy, as the cache must then be probed. Snoop filters generally yield a net energy savings because a significant number of lookups can be avoided in most cases. The challenge is to design snoop filters that are highly accurate, while ensuring reasonable costs for maintaining and updating the data contained in the snoop filter.

One of the most effective snoop filter architectures is based on stream registers, which provide a compact set-based representation of a contiguous range of memory blocks [6, 16, 17]. Stream registers can track the blocks that are allocated to a cache, which allows them to act as snoop filters; however, there is no efficient method to update a stream register when a block is removed from the cache. Over time, the accuracy of a stream register degrades, as it records all of the blocks that have ever been loaded into the cache, many of which have been evicted, as opposed to the exact set of blocks in the cache at a given time. Snoop filters based on stream registers include additional mechanisms to overcome these limitations, which are described in much greater detail in Section III.

This paper introduces a new snoop filter architecture based on our notion of a counting stream register, which overcomes many of the limitations of existing stream register snoop filters. Our experiments demonstrate that counting stream registers filter a higher percentage of memory accesses compared to traditional stream registers, which improves energy savings.

II. INCLUSIVE AND EXCLUSIVE SNOOP FILTERS

Snoop filters can be categorized as being either inclusive or exclusive [1, 13]. An inclusive snoop filter records a superset of the blocks that are cached. A request that misses in an inclusive filter is guaranteed to miss in the cache, so there is no need to forward the request. On the other hand, a hit in an inclusive snoop filter may or may not hit in the cache, so the request must be forwarded. In contrast, an exclusive snoop filter maintains information about blocks that are not cached. A hit in an exclusive snoop filter guarantees that the cache does not contain the block, so there is no need to forward the request. A miss is inconclusive, so the request must be forwarded to the cache for processing.

A snoop filter lookup consumes less energy than a tag lookup in the cache. Therefore, a result indicating that a block is not in the cache saves energy. In contrast, an inconclusive result increases energy consumption, as both the filter and the tag array are accessed. Consequently, the system behavior must satisfy two key criteria for snoop filters to be effective:

(1) The vast majority of snoop lookups are cache misses.

(2) The snoop filters are effective, i.e., they detect would-be cache misses correctly the majority of the time.

If the first criterion is not satisfied, then most filter accesses return an inconclusive result, and the tag would be looked up anyway; a system without snoop filters would be more effective. If the second criterion is not satisfied, the majority of lookups do not hit in the cache, but the filter is ineffective; a system without snoop filters would be preferable, as it would eliminate a larger number of unnecessary snoop filter accesses.

III. Stream Register Snoop Filters

Researchers at IBM introduced a Stream Register-based snoop filter, which was used in the Blue Gene/P supercomputer [6, 16, 17]. This filter is inclusive, and uses stream registers to encode cache blocks stored in the cache. Each stream register (SR) is composed of a base register, a mask register, and a valid bit. Intuitively, the base register encodes the starting point of an array under traversal, while the mask register encodes the entries of the array that have been accessed as offsets of the base. The offsets are not represented explicitly, as this would require a separate register for each entry. Instead, the mask represents a superset of the offsets that have been accessed.

We explain the behavior of the SR-Filter with an example taken from IBM's papers. Two hexadecimal addresses are loaded: *0x1708fb1* and *0x1708fb2*. The first address is copied to the base register, and the mask register is initialized to all ones—i.e.,

$$base = 0x1708fb1$$

$$mask = 0x7fffffff$$

The two least significant bits of the two addresses differ. When the second address is loaded, the two least significant bits of Mask are set to zero, and Base is overwritten with the second address—i.e.,

$$base = 0x1708fb2$$

$$mask = 0x7ffffffc$$

The two least significant zeroes in the mask register indicate that the two addresses that have been loaded into the register differ in the least two significant bits. The SR indicates that the data cache may contain four addresses—*0x1708fb0*, *0x1708fb1*, *0x1708fb2*, and *0x1708fb3* is a superset of the two addresses contained in the cache. In this state, this SR can successfully filter any address other than the four listed above.

Over time, more and more unique addresses will be fed to a given SR, and more and more of the bits in the mask will be set to 0. Thus, the space of all possible addresses that the stream register can filter will decrease over time. Eventually, all of the mask bits become zero, and the SR filters no further addresses.

There is no general mechanism to remove a specific address from the SR without compromising correctness. Instead, the SR is reset whenever the cache has been completely replaced relative to some initial state. This is called a cache wrap. Active SRs cannot just be cleared; instead, their contents are copied to a history SR. A history SR is treated as a second SR for the purpose of filtering, but its contents are not updated until the next wrap occurs. This organization with both an active and a history SR guarantees correct functionality.

There is no need to limit the snoop filter to a single SR active and history pair. As shown in Figure 1, the actual filter contains a bank of SRs, along with cache wrap detection and update logic, which flushes the registers when it dectects a wrap. Detecting a cache wrap is not a trivial problem and may require significant storage and logic. The detector's design has been only cursorily addressed in literature [6, 16, 17] with one such implementation patented [7].

One efficient implementation, for instance, is circuitry tightly integrated into the cache, thus requiring a full-custom design methodology. Full-custom design is reasonable for high-performance computing, but is unreasonable for embedded systems, where designers are unlikely to have access to this type of custom memory. Memories are generally provided by vendors of intellectual property (IP) in the form of standard single- or dual-port memory generators, and individual macros cannot be modified without significant designer effort and additional cost to the system.

A simpler SR-filter could be reset periodically, but correct operation requires a cache flush, which would cost significantly in performance and in energy. Instead, for easy implementation in an ASIC flow, we introduce an important modification to the SR-filters and make cache wrap detection or flushes unnecessary without deteriorating their filtering performance.

IV. Counting Stream Register Snoop Filter

Here, we introduce the Counting Stream Register-based snoop filter CSR-filter, which addresses the shortcomings of the SR-filter discussed in the preceding section. The CSR-filter eliminates the cache wrap detection logic, replacing it with a direct-mapped update unit instead. In the CSR-filter, each stream register is augmented with a counter; the counter bitwidth is limited by the number of lines that can be stored in the cache: for an 8KB cache with 32-byte lines, no more than 8 bits would be required. The counter bitwidth is independent of the page size, which affects the bitwidth of the base and mask registers, as discussed in IBM's papers [6, 16, 17].

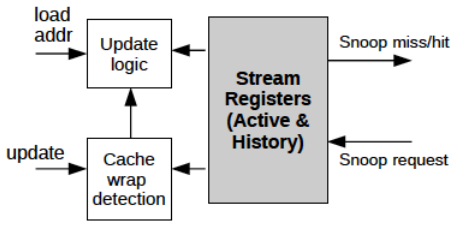

Figure 1. Architecture of a SR-filter.

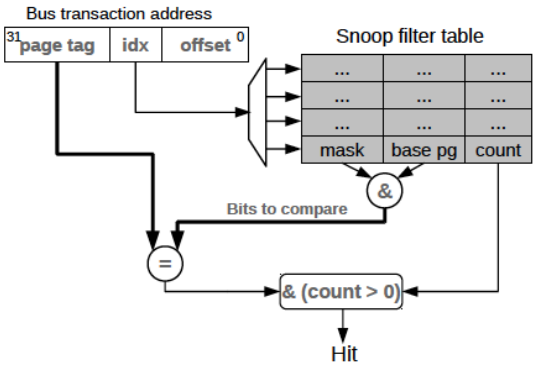

Figure 2. CSR-filter hit detection logic.

Figure 2 shows the snoop hit mechanism. The bus address is split into three parts: A set of cache lines (offset) is grouped into a page, and a set of bits (idx) is used to index the direct-mapped snoop filter table. The most significant remaining bits (page tag) are used as the tag for the base register. When a new cache line is loaded, the base page register is updated with the page tag, and all bits of the mask register are set to 1, indicating that all bits of the base register are valid. In addition, the counter, initially set to zero, is incremented. Similar to the example used in Section III, after loading both the addresses *0x1708fb1* and *0x1708fb2*, the contents of the base and mask registers are, *base = 0x1708fb2, mask = 0x7ffffffc*. The value of the counter is 2, since the counter is incremented for every new cache line load and two cache lines were loaded.

The counter is used as a validity checker and eliminates the need for the valid bit in the original SR-filter. Consecutive loads update the mask register and increment the counter. Cache line replacements, snoopy invalidations, and evictions decrement the counter i.e., an eviction or invalidation of the address *0x1708fb1* will decrement the counter to value 1. No modifications are necessary to the base and mask register contents. The stream register is effectively emptied when the counter resets to zero: this is the same functionality as cache wrap detection, but much simpler. The counter also eliminates the need to employ a cache/snoop filter flushing mechanism.

V. Experimental Setup

A. Experimental Platform

Our experimental platform was an internally developed FPGA-based soft processor emulation system running on a Xilinx Virtex-II FPGA. The processors in our system are 6-stage single-threaded RISC pipelines that implement the Open-RISC [14] instruction set. The size and associativity of the instruction and data caches for each processor in the system are configurable. An atomic bus interconnects the processors to one another and to the memory controller.

We instantiated a 5-core system that runs at 50 MHz. The system includes a 32 MB off-chip DRAM, which is used as a shared memory, and a variety of performance counters, whose measurements are used to generate results. All benchmarks were compiled using a "newlib"-based gcc 3.4.4 tool-chain for the Open-RISC.

Snoop filters were added to the OpenRISC cores thereby permitting us to evaluate the percentage of snoops that would be filtered. However, the filters sit in parallel to the L1 caches, and do not interfere with their operation. The performance numbers were estimated based on the penalty of an extra cycle during a "hit" in the filter for systems with SR and CSR-filters.

We modeled a system similar to IBM's Blue Gene/P, which maintains data integrity by using a write-invalidate cache coherence protocol with write-through L1-caches. In principle, a write-back cache configuration with a hardware coherence protocol would likely perform better; but, the configuration we used placed more stress on the snoop filters, which favored differentiation between the SR- and CSR-filters.

B. Benchmarks

We use the EEMBC MultiBench suite of parallel embedded workloads for our experimental evaluation. Table I lists the benchmarks that we used in our study. As our system DRAM capacity is limited to 32 MB, we had to limit the number of workloads executed and analyzed to about 75% of the total.

The EEMBC benchmarks are written using a generic API that is independent of an operating system. Developers who wish to use the EEMBC suite must first port the system APIs to a specific platform, operating system, and tool chain. Our platform, at present, lacks an operating system; it is programmed using a small library of rudimentary functions that perform memory management, I/O operations, and facilitate threads. Consequently, we modified the EEMBC suite to utilize our software library and to execute on a 5-core system.

The EEMBC MultiBench suite is multithreaded and is parallelized using a master-slave organization. One processor, the master, performs initialization, task management, and finalization; tasks are distributed to the slave processors, which perform the actual work. The OpenRISC processors are single-threaded, so each slave processor executes at most one task at a time, while the master processor queues future tasks.

The EEMBC benchmarks measure multicore performance across various degrees of computational intensity. The benchmark suites include workloads targeting the fields of image and video processing, cryptography, networking, encoding and automotive applications. Image and video processing involve continuous memory load/store operations. Cryptographic benchmarks such as MD5 exploit the system's computational resources and reveal memory bottlenecks, as several intermediate values of the ciphering are stored and retrieved during execution. This wide variety of the EEMBC workloads behavior enables us to convincingly generalize the results we have obtained to other embedded workloads.

TABLE I. EEMBC MultiBench Applications Used in Our Study.

Category	Benchmarks
Image Processing	Image rotation, RGB to CMYK conversion
Video Processing	H.264 video encoding
Networking	IP packet check, IP reassembly, TCP/IP network simulation
Coding	Huffman

C. Energy Model

A realistic estimation of the total system energy including the CPU pipeline, I/O and peripherals becomes specific to a given system, and hence, we present an isolated memory subsystem energy model. The model takes into account the energy consumption of the instruction and data caches, the interconnect bus, and the shared memory. It is important to recognize that the energy consumed by the memory subsystem is only a fraction of the total system energy.

CACTI 5.3 [20] provided per-access (read/write) energy estimates for each memory structure in our system. This information was collected into tables, and we used a standard profiling-based table-lookup methodology to estimate energy consumption, similar in principle to tools used in cycle-accurate software simulators. We used the 90nm technology node, which is popular in current embedded system designs, and determined the read energy, write energy, leakage energy, and snoop lookup energy for a variety of cache configurations. We considered both write-through and write-back caches.

The total energy consumption of the system was modeled considering the number of data and instruction cache write and read accesses: N_{ICR}, N_{ICW}, N_{DCR}, N_{DCW}; and the number of bus transactions, N_{BT}. The snoop energy for each data cache access was calculated based on the number of snoopable transactions, N_{STRANS}, and the energy consumed for a single data cache tag array look-up E_{TAGLU}. This snoop energy is summed into the total energy consumed by the data cache.

The energy to perform one data/instruction cache read/write are provided by CACTI, and are denoted by E_{ICR}, E_{ICW}, E_{DCR}, and E_{DCW}; the average energy consumed to access the shared memory unit is E_{SMRW}. Let N_P denote the number of processors in the system. The energy consumed by each instruction and data cache, E_{IC} and E_{DC} respectively, total bus energy, E_B, and total memory subsystem energy consumption, E, are:

$$E_{IC} = N_P(N_{ICR}E_{ICR} + N_{ICW}E_{ICW})$$

$$E_{DC} = N_P(N_{DCR}E_{DCR} + N_{DCW}E_{DCW} + N_{STRANS}E_{TAGLU})$$

$$E_B = N_{BT}E_{SMRW}$$

$$E = E_{IC} + E_{DC} + E_B.$$

Many implementations exist for a cache with a given size and associativity. For example the cache could be banked or non-banked; if it is banked, the number of banks may vary; etc. For each cache that we considered, we looked at all possible implementations, and, based on the results obtained from CACTI, selected the one that consumed the least energy.

D. Cache Design Space Exploration

As the behavior of workloads varies largely with different cache configurations, we desired to determine the best performing and most energy efficient cache configuration for each benchmark. The EEMBC benchmarks were run for various data and instruction cache configurations with varying sizes and associativity. We varied the cache sizes from 2KB to 16KB, and considered direct-mapped and 2- and 4-way set associative implementations for each size; all caches used the Least Recently Used (LRU) replacement policy.

The instruction and data caches, the interconnect bus and shared memory constitute the memory subsystem of the FPGA emulation platform. The energy consumed by the memory subsystem and the performance in terms of execution cycles is observed for all the cache configurations.

Figure 3 shows an example of a design space exploration for the Huffman Benchmark; we performed a similar design space exploration for each benchmark. The configurations that achieve the fastest execution time and lowest energy consumption are marked. The results demonstrate that cache parameters can significantly affect the observed results. Since the objective of snoop filtering is to reduce memory subsystem energy consumption, we selected the lowest energy consuming cache configuration for each benchmark; further experimental results are reported for this configuration alone.

VI. Experimental Results

A. SR- and CSR-filter Hardware Implementations

Using CACTI 5.3 [20], we modeled the area of SR- and CSR-filters containing 32, 64, and 128 SRs and CSRs respectively. For the SR-filter, we did not consider the area overhead of the update logic and cache wrap detector, which are considerably smaller than the registers themselves. Table II reports the results of the comparison.

The CSR-filters are marginally larger than the SR-filters, due to the extra counter bits; however, this does not account for the overhead of the update and cache wrap detection logic, so the SR-filter is expected to be larger than the CSR-filter. Further experimentation will demonstrate that the CSR-filter handles evictions, snoopy invalidations, and replacements more gracefully than the SR-filter, and without the custom wrap detection logic; as such, we consider it to be the better choice, especially for cost-constrained embedded systems.

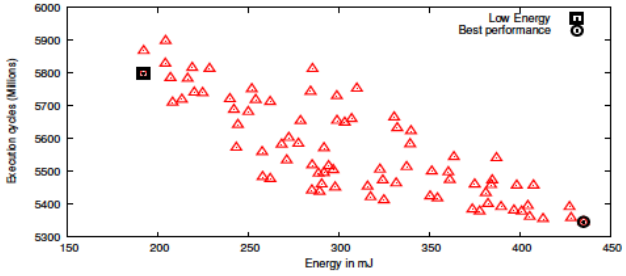

Figure 3. Complete cache design space exploration for the Huffman benchmark with energy and performance estimates for each of the configurations. The Low Energy configuration has a performance degradation of about 7% while the Best Performance configuration consumed almost 2.5 times that of the Low Energy configuration.

TABLE II. Area estimates for SR- and CSR-filters with 32, 64, and 128 SRs and CSRs respectively. The SR-filter area estimate does not account for the cache wrap detection and update logic.

Number of SRs	SR Area	CSR Area
32	0.040 mm^2	0.041 mm^2
64	0.047 mm^2	0.060 mm^2
128	0.078 mm^2	0.083 mm^2

B. Filtering Percentages

Next, we compare filtering effectiveness using SR- and CSR-filters. We considered snoop filters with 8, 16, 32, 64, and 128 SRs and CSRs respectively; Figure 4 reports the filtering percentage achieved by both filters. The CSR-filter achieved an equal or higher filtering percentage than the SR-filter in all cases, due to improved handling of snoopy invalidations.

CSR-filters with fewer registers can achieve higher filtering percentages than SR-filters with a larger number of registers. For example, observing the Huffman benchmark in Figure 4(a), the SR-filters show a gradual increase in filtering effectiveness as the number of SRs increases, while the CSR-filters display a consistent filtering rate of 100% across all filter sizes.

Similarly, in Figure 4(d) the CSR-filters achieve a filtering rate of 95% with 32 CSRs for the H.264 Video Encoding while the SR-filters require twice as many SRs to achieve the same filtering percentage. SR-filters equaled the CSR-filters in terms of filtering percentage for RGB-to-CMYK with 8 SRs/CSRs; and H.264 Video Encoding with 64 and 128 SRs/CSRs.

Next, we examine RGB-to-CMYK Image Conversion in greater detail, in order to see why CSR-filters are more consistent than SR-filters; pseudocode is provided as follows:

```
/* Calculate complementary colors */
c = 255 - R;      m = 255 - G;          y = 255 - B;

/* Find the black level K */
K = minimum (c,m,y)

/* Correct complementary color level based on K */
C = c - K;        M = m - K;            Y = y - K;
```

Fig. 5(a) and (b) show the filtering percentages of SR- and CSR-filters for 2MB and 4MB images for RGB to CMYK image conversion. For 4MB images, the filtering percentage of SR-filters starts at 80% for filters with 8 SRs, reduces linearly with 16 and 32 SRs, and then improves to 80% for 64 and 128 SRs. The 8 SRs are mapped through 3 bits of the physical address space, while a 9-bit offset accounts for all 4 MB of the image. As the number of SRs increases, more bits of the address space are required to address the SRs, and fewer bits are available for the image itself. The CMYK calculation is an inter-dependent two-step process where several invalidations and modifications occur, which the CSR-filters detect, but the SR-filters do not. The increases the uncertainty about the data present in the cache (the "maybe" condition), which results in a snoop filter hit, thereby reducing the filtering rate shown in Fig. 5(a). Increasing the number of SRs per filter to 64 and 128 eliminate the uncertainty.

Our CSR-filters act on the invalidations and modifications more effectively than the SR-filters, as shown in Figure 5(b). Like the SR-filters, the results are consistent for a 2MB image regardless of the number of CSRs in the filter; for a 4MB image, 8 and 16 CSRs appear to be insufficient, while the filtering percentage remains consistent for filters with 32 or more CSRs. For both image sizes, SR-filters achieve a maximum filtering percentage of 80%, while CSR-filters achieve a filtering percentage of 100% in most cases.

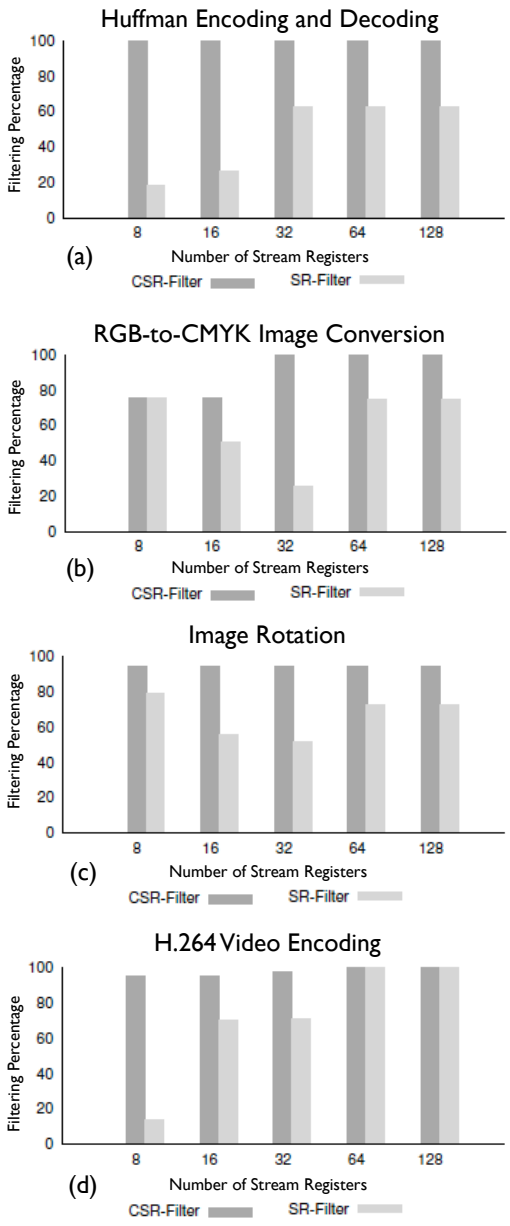

Figure 4. The CSR-filter consistently achieves a higher filtering percentage than the SR-filter.

Altogether, these results demonstrate that CSR filters are more robust to workload variability than SR-filters, while achieving a better overall filtering percentage.

C. Energy Consumption

Our experimental analysis considers the following three system configurations:

WT: Write-through caches without snoop filters.

WTSR: Write-through caches with SR-filters.

WTCSR: Write-through caches with CSR-filters.

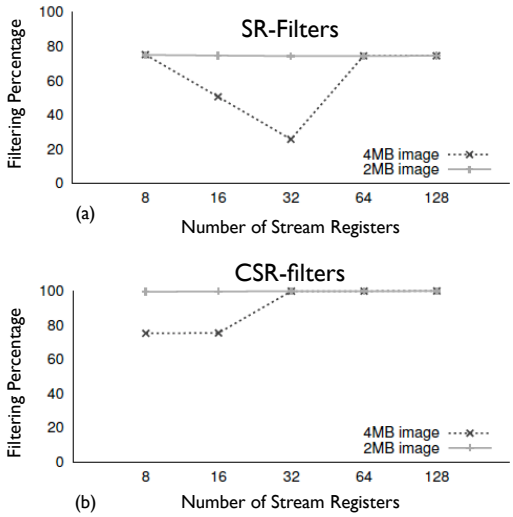

Figure 5. RGB-to-CMYK conversion with SR-filters (a) and CSR-filters (b). For SR-filters the filtering rate is inconsistent as the number of SRs increases for a 4MB image; for CSR-filters, the filtering rate is non-decreasing when the number of CSRs increases.

As mentioned earlier, the usage of a write-through protocol was motivated by IBM's Blue Gene/P supercomputer, which introduced SR-filters. Our goal was to show that CSR-filters could be more effective if used in a similar context (albeit, with different workloads and evaluated using a memory-limited emulation platform). Coherence with write-through caches is maintained implicitly by broadcasting invalidation messages for each write-through to main memory.

Figure 6 reports the memory subsystem energy for the EEMBC MultiBench suite using the configurations listed above. For most of the benchmarks, snoop energy was around 8-10% of the total memory subsystem energy without snoop filters. In many cases, SR-filters and CSR-filters were equally effective in terms of reducing memory subsystem energy; however, CSR-filters were clearly more effective for H.264 Video Encoding, Image Rotation, and IP Reassembly.

The granularity of the snoop filters also affects the overall energy consumption; RGB-to-CMYK Image Conversion is a typical example of this problem. The algorithm applies a sliding window to the image to perform the conversion. As the window moves, more and more addresses are added to the SRs, despite the fact that the working set at any given time is relatively small, namely the region of the window itself. Each new address adds more zeroes to the mask registers without a reset. Nonetheless, the counting mechanism of the CSR-filters is more effective than the cache wrap detection and update logic of the SR-filters.

These workloads do not stress the memory subsystem for two key reasons. First, the input data was relatively small, due to the fact that the system is limited to 32MB of off-chip SDRAM. Second, the parallelization scheme does not lead to complex data sharing arrangements, and the amount of data having multiple writers is quite limited. Consequently, we believe that a larger system with different workloads would increase the energy advantage of CSR-filters over SR-filters.

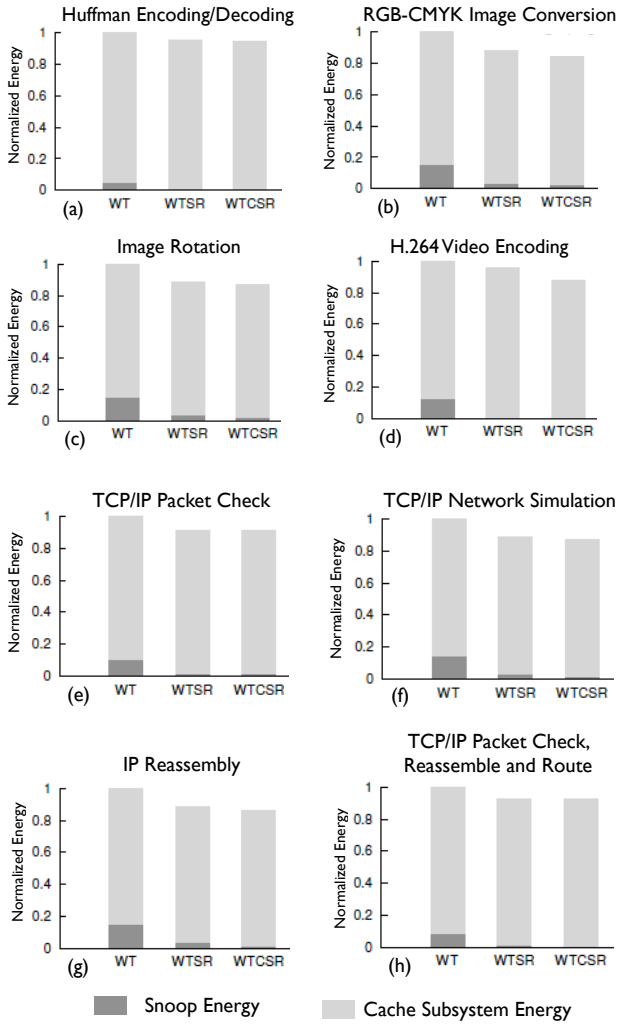

Figure 6. Energy consumption of write-through caches (WT), write through caches with SR-filters (WTSR), and write-through caches with CSR-filters (WTCSR) for the EEMBC MultiBench suite. Snoop energy was typically 8-10% of total memory subsystem energy for most benchmarks. CSR-filters were uniformly more effective than SR-filters across the benchmark suite.

D. *Performance*

Figure 7 reports the normalized execution time of the benchmarks for the WT, WTSR, and WTCSR snoop filtering schemes described in the preceding section. The write-through schemes incur a significant amount of bus and memory traffic. Snoop filters add an extra cycle to each memory access that hits in the cache [13].

The performance degradations we observed were negligible for most benchmarks, with a maximum of 3% for TPC/IP Packet Check. Altogether, our results demonstrate that the reduced energy consumption of snoop filters offsets the performance overhead.

VII. Related Work

The most comprehensive reference on snoop filters is a wiki maintained at the University of Toronto [1].

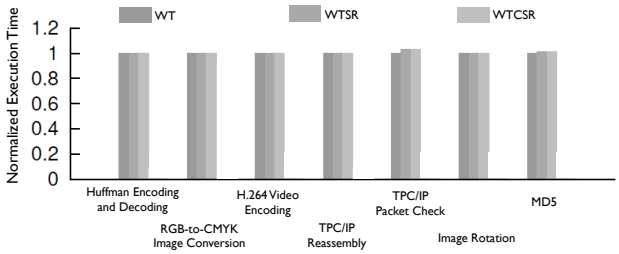

Figure 7. Execution time for each benchmark, normalized to WT. The worst-case performance overhead observed for WTSR and WTCSR was 3% for TCP/IP Packet Check.

Snoop filters generally fall into one of three categories:

Destination-based filters: attempt to eliminate tag lookups in response to a snoop broadcast.

Source-based filters: attempt to reduce the number of snoop broadcasts.

Other filters: rely on properties of the interconnect network, virtualization, or application-specific designs.

The SR- and CSR-filters discussed in detail in this paper are categorized as destination-based filters.

A. Destination-based Filters

Most destination-based filters are classified as either inclusive or exclusive, as discussed in Section II.

Inclusive filters can be further categorized as superset and subset filters. Superset filters identify a superset of all of the lines in the cache, while benefitting from a more space-efficient representation of this information than tracking every line. The SR-filters used in IBM's Blue Gene/P [6, 16, 17] are superset filters, as are several other designs that track blocks using counting Bloom filters. The inclusive JETTY [13], one of the first snoop filters introduced, performs L2 snoop filtering in SMP systems; to improve performance, it includes a small table to cache snoop requests that recently missed in the local cache; accesses that hit in this table can be filtered a-priori. Ballapuram et al. [5] described a similar snoop filter that focuses on L1 snoops and includes some features to support self-modifying code.

Strauss et al. [19] developed an inclusive subset snoop filter that adds a new coherence state for cache lines. A cache line is in the supplier state if it may provide a positive response to a snoop lookup; the subset filter tracks the subset of supplier blocks that are actually cached; it cannot filter snoops that access lines in other states. This particular filter was designed for a CMP with processors connected by a ring topology.

Examples of exclusive filters include the exclusive JETTY [13] and the range filter used as part of the Blue Gene/P Snoop filter [6, 16, 17]. The exclusive JETTY [13] maintains a table of addresses that have been recently snooped and return negative responses. Lines are removed from the table when they are loaded into the cache, or to make room for new lines when table capacity is exceeded.

The range filter [6, 16, 17] finds large range of addresses and specifies that address within (or completely outside of) that range are not in the cache, and can therefore be filtered. The range filter is useful in parallel applications where processors work on distinct and continuous ranges of memory.

One last destination-based filter, introduced by Atoofian and Baniasadi [3], is difficult to categorize as either inclusive or exclusive. Each cache maintains a table of saturating counters (one per core). When a core sends a request to the cache, the first step is to check the counter. If the counter is *not* saturated, then the cache returns a negative reply, regardless of whether it contains a copy of the data, under the speculative assumption that another core will be able to provide the line. If no core provides the line, then the processor re-issues its request and all cores perform cache lookups. This filter is area and energy-efficient, but it is only effective for workloads that exhibit supplier locality; it is ineffective for other workloads.

B. Source-based Filters

Source-based filters allow local caches to detect that all other remote caches do not contain the data, and therefore allows them to suppress snoop broadcasts.

A write-through cache places every write operation on the bus, and the increased bus traffic leads to increased snoop lookups at the caches. Write-back caches with cache coherence protocols are one of the simplest forms of source filtering. A write-back approach reduces bus traffic by *not* placing every cache write on the bus, and instead only writing back when a remote cache requests the data.

Coherence protocols used in conjunction with write-back caches play an important role in source filtering as well. For example, the MSI protocol categorizes each cache line as M(odified), S(hared), or I(nvalid), while the MESI protocol adds an E(xclusive) state. If the local cache contains a line in the exclusive state, there is no need to broadcast an access to that line, because no other cache contains a copy.

Atoofian et al. [4] developed a source-based snoop filter, which used saturating counters, sharing some principle similarities to their destination-based filters [3]. The filter predicts when remote caches are likely to supply data in response to a snoop broadcast; when the same processor services many subsequent requests, the requests are sent directly to the supplier, as opposed to snoop broadcasts.

Another approach is to provide augment instructions that access memory with a bit that can be set to suppress snoops [5]. Programmers or compilers could set the bits appropriately based on their knowledge of the application and its behavior. This approach requires minimal architectural support, but requires the ability for a programmer or compiler to understand the memory layout of the program, and possibly deal with issues such as pointer aliasing.

Other destination-based filters can be categorized as coarse-grained, wherein, the filter tracks whether or not at least one line in a larger region is in the cache; filtering is performed on the granularity of regions, rather than individual lines. Ekman et al. [9], for example, track sharing in the operating system on the granularity of pages.

Cantin et al. [8] took a similar approach with their Region Coherence Array (RCA), which tracks region sizes ranging from 512 bytes to 8KB, which offers greater flexibility than fixing the granularity to the page size. Multi-granularity snoop filters track coarse-grained regions, but within a larger page [15]. Subspace snooping records sharing information for each memory page in the page table itself, and therefore relies on operating system support [10].

RegionScout [12] and RegionTracker [22] sacrifice the precision of information about regions stored in the cache in order to achieve efficient hardware implementations. RegionScout maintains an incomplete list of regions that are not shared, and cannot answer precisely whether a region is shared. RegionTracker moves this information into the tag arrays of the cache, rather than storing it in external structures.

C. Other Filters

Serial snooping [18] distributes messages sequentially to processors, rather than via broadcast. The scheme is beneficial if the data is found early in the sequence, but there is a performance penalty if the data is found late, or is not found.

In-network filtering [2] distributes coarse-grained coherence information in routers throughout the network. When a broadcast occurs, the routers create a multicast tree to send the message to cores that contain the region. The major limitation is that in-network filtering does not appear to be compatible with non-adaptive routing protocols.

Virtualized workloads tend to have sharing limited to threads running on the same virtual machine, and that there is only need to snoop cores in the same virtual machine [11]. This approach involves the operating system and hypervisor, and special care must be taken to facilitate workload migration.

Zhou et al. [23] used a compiler analysis to disambiguate the memory space of an application into private and shared data regions; an operating system-supported mechanism was proposed to suppress all snoops, except for those that access shared data directly. Yu and Petrov [21] exploited the fact that in embedded systems, important a-priori knowledge is available regarding task allocation, sharing patterns of the processor nodes, and inter-processor communication. Both of these mechanisms take an application-specific approach to snoop filter optimization.

VIII. Conclusion

The CSR-filter architecture improves significantly over the SR-filter architecture introduced in IBM's Blue Gene/P supercomputer. The CSR-filter achieves a higher filtering percentage than the SR-filter, often doing so with a smaller number of stream registers. For the EEMBC MultiBench suite, the reductions in overall energy consumption were relatively small for two reasons: (1) our implementation platform limited the memory footprint; and (2) the workloads are parallelized into mostly independent tasks with little sharing of data. Nonetheless, our experiments clearly show that CSR-filters are more effective than SR-filters. We believe that these results generalize to other workloads and to L2 snoop filtering.

References

[1] http://www.eecg.toronto.edu/~moshovos/filter/doku.php?id=start

[2] Niket Agarwal, Li-Shuan Peh, Niraj K. Jha: In-network coherence filtering: snoopy coherence without broadcasts. MICRO 2009: 232-243

[3] Ehsan Atoofian and Amirali Baniasadi: Using supplier locality in power-aware interconnects and caches in chip multiprocessors. J. Systems Architecture 54(5): 507-518 (2008)

[4] Ehsan Atoofian, Amirali Baniasadi, Kaveh Aasaraai: Speculative supplier identification for reducing power of interconnects in snoopy cache coherence protocols. CF 2007: 259-266

[5] Chinnakrishnan S. Ballapuram, Ahmad Sharif, Hsien-Hsin S. Lee: Exploiting access semantics and program behavior to reduce snoop power in chip multiprocessors. ASPLOS 2008: 60-69

[6] Matthias A. Blumrich, Valentina Salapura, Alan Gara: Exploring the architecture of a stream register-based snoop filter. T. HiPEAC 3: 93-114 (2011)

[7] Matthias A. Blumrich, Alan G. Gara, Mark E. Giampapa, Martin Ohmacht, Valentina Salapura: Method and apparatus for detecting a cache wrap condition. US Patent US7386684B2, June 2008.

[8] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith: Improving multiprocessor performance with coarse-grain coherence tracking. ISCA 2005: 246-257

[9] Magnus Ekman, Per Stenstrom, Fredrik Dahlgren: TLB and snoop energy-reduction using virtual caches in low-power chip multiprocessors. ISLPED 2002: 243-246

[10] Daehoon Kim, Jeongseob Ahn, Jaehong Kim, Jaehyuk Huh: Subspace snooping: filtering snoops with operating system support. PACT 2010: 111-122

[11] Daehoon Kim, Hwanju Kim, Jaehyuk Juh: Virtual snooping: filtering snoops in virtualized multicores. MICRO 2010: 459-470

[12] Andreas Moshovos: RegionScout: exploiting coarse grain sharing in snoop-based coherence. ISCA 2005: 234-245

[13] Andreas Moshovos, Gokhan Memik, Babak Falsafi, Alok N. Choudhary: JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers. HPCA 2001: 85-96

[14] Damjan Lampret. OpenRISC 1000 Architecture Manual, Apr. 2006. http://www.opencores.org/

[15] Avadh Patel, Kanad Ghose: Energy-efficient MESI cache coherence with pro-active snoop filtering for multicore microprocessors. ISLPED 2008: 247-252

[16] Valentina Salapura, Matthias A. Blumrich, Alan Gara: Improving the accuracy of snoop filtering using stream registers. MEDEA 2007: 25-32

[17] Valentina Salapura, Matthias A. Blumrich, Alan Gara: Design and implementation of the Blue Gene/P snoop filter. HPCA 2008: 5-14

[18] Craig Saldanha and Mikko H. Lipasti: Power efficient cache coherence. WMPI 2001

[19] Karin Strauss, Xiaowei Shen, Josep Torrellas: Flexible snooping: adaptive forwarding and filtering of snoops in embedded-ring multiprocessors. ISCA 2006: 327-338

[20] Shyamkumar Thoziyoor, Naveen Muralimanohar, Jung Ho Ahn, Norman P. Jouppi: CACTI 5.1. Hewlett Packard Laboratories Technical Report HPL-2008-20, 2008

[21] Chenjie Yu and Peter Petrov: Low-power snoop architecture for synchronized producer-consumer embedded multiprocessing. IEEE TVLSI 17(9): 1362-66 (2009)

[22] Jason Zebchuk, Elham Safi, Andreas Moshovos: A framework for coarse-grain optimizations in the on-chip memory hierarchy. MICRO 2007: 314-327

[23] Xiangrong Zhou, Chenjie Yu, Alokika Dash, and P. Petrov. Application-aware snoop filtering for low-power cache coherence in embedded multiprocessors. ACM TODAES 13(1):1–25 (2008)