Exploring the Architecture of a Stream Register-Based Snoop Filter

Matthias Blumrich, Valentina Salapura, and Alan Gara

IBM Thomas J. Watson Research Center
Yorktown Heights, NY, USA

Abstract. Multi-core processors have become mainstream; they provide parallelism with relatively low complexity. As true on-chip symmetric multiprocessors evolve, coherence traffic between cores is becoming problematic, both in terms of performance and power. The negative effects of coherence (snoop) traffic can be significantly mitigated through the use of snoop filtering. The idea is to shield each cache with a device that can eliminate snoop requests for addresses that are known not to be in the cache. This improves performance significantly for caches that cannot perform normal load and snoop lookups simultaneously. In addition, the reduction of snoop lookups yields power savings. This paper describes Stream Register snoop filtering, which captures the spatial locality of multiple memory reference streams in a small number of registers. We propose a snoop filter that combines Stream Registers with "snoop caching", a mechanism that captures the temporal locality of frequently-accessed addresses. Simulations of SPLASH-2 benchmarks on a 4-core multiprocessor illustrate tradeoffs and strengths of these two techniques. We show that their combination is most effective, eliminating 94% - 99% of all snoop requests using only a small number of stream registers and snoop cache lines.

1 Introduction

As single-core performance becomes increasingly hard to improve, and marginal costs are growing, both in terms of complexity and power/performance inefficiency [1], the use of multi-core solutions to improve throughput in multi-threaded workloads has become increasingly attractive [2].

Unlike designs which target single-thread solutions with degrading power/performance efficiency, suitably scalable parallel workloads show little or no degradation in efficiency while delivering significant increases in performance through the use of multithreaded workloads. Using parallelism at the processor level also aligns with the limits of future technologies. Although performance growth has been driven by technologically-enabled increases in processor operating frequency for the past 20 years, it is increasingly hard to obtain with new technologies. One of the main reasons is the impact of wire delays as feature sizes are shrunk [3], requiring increasingly more sophisticated microarchitectures.

While faster transistors and wires are increasingly hard to obtain, the application of Dennard's CMOS scaling theory [4] is continuing to deliver improvements in density. Thus, multi-core solutions are based on a commercially viable exploitation of modern CMOS fabrication processes.

Several multi-core solutions have been introduced over the past few years, such as the IBM POWER4 and POWER5 servers, the IBM Blue Gene/P system [5], the Intel Quad Core processors [6], and the Cell Broadband Architecture [7]. Indeed, multi-core is now a well-established trend. A major challenge in the implementation of chip multi-processors is providing a suitable memory subsystem and on-chip interconnect that combines low average access latency with high bandwidth.

As the number of processors per chip rises, the coherence traffic per processor consequently increases. One solution to reducing the cost of coherence is to manage it in software; a solution adopted by both the Blue Gene/L [8] and Cell system architectures. In Blue Gene/L, software managed coherence is achieved by using one of two software abstraction models: virtual node mode, wherein each processor is a separate node in the Blue Gene/L-optimized MPI implementation, or coprocessor mode, where one processor is a dedicated computational node and a second processor provides I/O and system management functions. In the Cell Broadband Architecture, coherent DMA and the SPU-local store provide the necessary memory abstractions for building high-performance systems. Although software-managed coherence offers an attractive solution to achieving low-complexity memory architectures, it requires advanced compilation technologies and careful application tuning. While this is acceptable for high-end application-specific systems, providing low-complexity, coherent memory is an attractive solution for a wider range of systems.

To reduce the complexity of implementing coherence in chip multiprocessors, two component costs must be addressed:

- The bottleneck of a bus-based snoop implementation, which must be arbitrated between a high number of nodes.
- The cost of providing snoop ports to each processor's cache, or the cost of maintaining a central directory.

In this paper we have investigated the use of coherence request filtering (or snoop filtering) in a point-to-point coherence network to address these costs. The basic idea is to provide a mechanism which will significantly reduce the interference of coherence requests with processor operations without incurring costs of chip area, memory latency, or complexity inherent in existing hardware coherency support.

The contributions of this work include (1) a novel, highly efficient, point-to-point snoop filter architecture filtering in excess of 98% of all snoop requests, (2) significant reduction of area over a solution that duplicates cache directories to provide separate snoop directories, (3) an architecture to exploit the cache replacement policy to periodically re-train the snoop filters, increasing filtering effectiveness from an average of 30% to an average of 90% for the workloads studied, and (4) evaluation of the proposed architecture, including variations of several key parameters.

This paper is organized as follows. We begin with related work in Section 2. Then Section 3 describes the snoop filter architecture, and Section 4 gives a detailed description of stream registers. The simulation environment and methodology are presented in Section 5, followed by experimental results and analysis in Section 6. Finally, Section 7 concludes.

2 Related Work

In prior art, JETTY [9] is a snoop filter that combines two complementary filtering methods. The JETTY paper defines a characterization of filters as "include" or "exclude". An include filter tracks what *is* contained in a cache (or caches) while an exclude filter tracks what *is not*. The exclude filter consists of a cache of recently invalidated lines. A snoop that hits in the exclude filter is guaranteed not to be in the cache, so it can be filtered. The JETTY include filter captures a superset of a cache's contents. A snoop that hits in the include filter may be in the cache and should not be filtered. The include filter is like a simple bloom filter with direct-map hash functions applied to sub-fields of the address.

The JETTY paper makes an argument for snoop filtering as a means for power savings. However, our work was primarily motivated by the need to filter useless snoops that reduce performance. We also considered chip area and power consumption as significant constraints, causing us to look beyond the simple and accurate method of duplicating the cache tags as a filter. Because our simulation methodology and system organization differ considerably from the JETTY study, it is difficult to compare performance results. However, we simulated many of the same applications using the same problem sizes.

Several coherent network switches contain snoop filters that block unnecessary coherence requests from ever leaving a node. One such example is the Scalability Port Switch of the Intel E8870 chipset [10]. In this case, the snoop filter tracks the state of all cache lines within a 4-processor node for a system with up to four such nodes. Kant [11] modeled a similar system architecture with such a snoop filter. This architecture is also described in the Azusa system [12], which is based on Intel Itanium processors and may use an Intel chipset.

In [13], a HyperTransport network switch for use with AMD Opteron processors is described. The snoop filtering technique is basically the same as that of the E8870, including the fact that 4-processor nodes are supported.

A similar but more tightly-coupled architecture is evaluated in [14], where a single memory controller switch connects multiple multi-processor nodes and contains a snoop filter. The filter prevents unnecessary snoop requests between the nodes, and several variants are studied.

Snoop filters in tightly-coupled multiprocessors, such as chip multiprocessors (CMPs), can be located at each processor in order to squash unnecessary snoops without changing the overall coherence scheme. Ekman et. al. [15] describe a CMP architecture with Page Sharing Tables, which are exclude filters at the granularity of memory pages rather than cache lines. This architecture is a bit more involved in that the Page Sharing Tables coordinate to track page sharing rather than just presence.

The idea of preventing remote snoop requests from being broadcast can also be applied at the chip level as described in [16]. In this work, snoop filters keep track of memory regions, which can be quite large, and block remote snoops for memory that is known not to be shared.

3   Snoop Filter Architecture

In symmetric multiprocessor (SMP) architectures, coherency snoop requests represent a significant fraction of all cache accesses, but only a small fraction of snoop requests are actually found in any of the remote caches [9,17]. This is particularly true of supercomputing applications where data partitioning and data blocking have been performed to increase locality of reference and optimize overall compute performance. As a result, embedded cores with single-ported caches suffer significant performance loss due to unnecessary snooping because their caches are unavailable during snooping.

While data reference locality may make a coherence implementation seem unnecessary, there is a fine line between being able to prove that there is no data sharing at all, and the statistical observation that almost all data references are local. The former is a program correctness statement, the latter is a performance statement. Thus, providing an efficient snoop filtering implementation that can filter the vast majority of non-shared data traffic without burdening the cache bandwidth of every processor in the system provides a significant performance improvement over traditional cache-coherent systems. Snoop filtering and cache coherence also offer advantages over programs operating on fully disjoint data sets without hardware coherence by simplifying application porting and tuning – it no longer becomes necessary to eliminate all remote references under all circumstances, but most remote references for most situations, leading to increased programmer productivity by letting programmers focus on the common case.

This motivated us to introduce a simple hardware device that filters out incoming snoop requests, reducing the number of actual snoop requests presented to the cache, thus increasing performance and reducing power consumption. A snoop filter is associated with each of the four processors and is located outside the L1 cache. To make a snoop filter a viable implementation choice, it has to meet several requirements:

- Functional correctness – the filter cannot filter requests for data which are locally cached.
- Effectiveness – the filter should filter out a large fraction of received snoop requests.
- Design efficiency – the filter should be small and power-efficient.

In theory, a perfect snoop filter can be created by duplicating the cache tag directory and using it to determine exactly which snoops should be forwarded to the cache. However, this approach generally does not meet the design efficiency requirement because of practical limitations. In particular, the cache tag store is a highly optimized and integrated component that cannot easily be extracted, and a custom-designed equivalent with the same performance requires a large investment of time and expertise to complete. Furthermore, this paper will show that a highly-effective filter can be implemented in a fraction of the area needed for duplicate cache tags.

Ideally, a snoop filter will operate at the (typically lower) memory nest frequency to reduce power dissipation and design complexity. Lower frequency and reduced latch count reduces the number of state transitions and load on clock nets, which are the major contributors to power dissipation. Operating the snoop filter at a lower frequency simplifies the design, as transactions have to be less heavily pipelined, eliminating a variety of bypass conditions which have to be validated and tested. It also simplifies timing closure.

In some sense, a snoop filter trades off power consumed by cache lookups with power consumed by the filter. However, there is an additional overall energy reduction because the processor performance benefit resulting from reduced snoop interference causes applications to complete sooner. Therefore, energy, measured as power consumed in time, is reduced.

In this work, we only consider a write-through L1 cache, where data integrity between the processors is maintained with a cache coherence protocol based on snoop invalidates. Thus, every time a processor issues a store, snoop invalidate messages are generated at all other processors, and no other coherence messages are required. This approach could be extended to write-back coherence protocols, where both read and write snoops generally require responses from remote caches. At each remote cache, the snoop filter would quickly determine whether the snooped address was not present, and if so, immediately send the response without the need to snoop the cache.

3.1   Point-to-Point Snoop Filter Interconnection

Previous work on snoop filters has only considered bus based systems. In such systems, all cache controllers snoop a shared bus to determine whether they have a copy of every requested data block. A simple and common-place coherence protocol that utilizes snooping is "write-invalidate".

In a write-invalidate protocol, each write causes all copies of the written line to be invalidated in all other caches. If two or more processors attempt to write the same data simultaneously, only one of them wins the race, causing the other processors' copies to be invalidated. The use of the shared bus enforces write serialization.

For every write bus transaction, all cache controllers have to check their cache address tags (a.k.a. snoop) independently to see if they are caching the written line. With the increasing number of processors on a bus, snooping activity increases as well. Unnecessary coherency requests degrade performance of the system, especially impacting the supercomputing applications where only a small fraction of snoop requests are actually found in any of the remote caches.

In [9], several proposals for reducing snoop requests using snoop filters are described. While reducing the number of snoop requests presented to a cache up to about 70%, the performance of the systems are still limited because of the interconnect architecture and lack of support for multi-porting. The architecture described is based on a shared system bus, which establishes a common event ordering across the system. While such global time ordering is desirable to simplify the filter architecture, it limits the possible system configurations to those with a single, shared bus. Alas, such systems are known to be limited in scalability due to contention for the single global resource. In addition, global buses tend to be slow, due to the high load of multiple components attached to them, and inefficient to place in CMPs.

Thus, in a highly-optimized, high-bandwidth system, it is desirable to provide alternate interconnect architectures, such as star or point-to-point. These are advantageous, as they only have a single sender and transmitter, thereby reducing the load, allowing the use of high speed protocols, and simplifying floor planning in CMPs. Using point-to-point interconnects also allows several transmissions to be in-progress simultaneously, thereby increasing the data transfer parallelism and overall data throughput.

Another limitation of the bus-based architecture is the inability to perform snoop filtering on several requests simultaneously, because simultaneous snoop requests from several processors have to be serialized by the bus. Allowing the processing of several snoop requests concurrently provides a significant increase in the number of requests that can be handled at any one time, and thus increases overall system performance. However, this increased data throughput means that snoop filters for such interconnects must be designed to accommodate multiple requests simultaneously.

In this paper, we opt for a system incorporating snoop filters to increase overall performance and power efficiency without limiting the system design options to a common bus. We have designed a snoop filter architecture supporting systems using point-to-point connections that allows each processor's snoop filter to filter requests from multiple memory writers concurrently. Our high-performance snoop filter is implemented in a pipelined fashion to enable high system clock speeds. Figure 1 illustrates our approach.
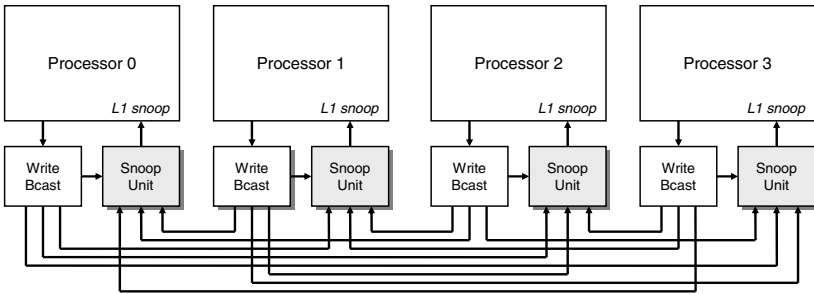
Fig. 1. Chip multiprocessor using snoop filters and a point-to-point interconnection architecture. All writes are broadcast by each processor to all remote caches for invalidation.

To take advantage of the point-to-point architecture and allow for concurrent filtering of multiple snoop requests, we implement a separate snoop filter block, or "port filter", for each interconnect port. Thus, coherency requests of all ports are processed concurrently, and a small fraction of all requests are forwarded to the processor. For example, each snoop filter in Figure 1 would have three separate port filters, each of which handles requests from one remote processor.

As the number of processors in a CMP scales up, the interconnect naturally transitions from point-to-point to a network-on-chip (NoC). At those scales, coherence protocols are likely to be based on directories, where filtering effectively takes place at the source. Therefore, our system architecture is most obviously applicable to CMPs of modest scale [18].

It should be noted that the snoop filters are basically transparent with respect to the behavior of the point-to-point interconnect. This interconnect poses particular design challenges, such as ordering between snoop invalidates, that exist regardless of the presence of snoop filters. In this study, we assumed a consistency protocol that is not sensitive to the order of invalidations coming from different processors. When necessary, a global synchronization would be used to enforce completion of all snoops.

3.2   Snoop Filter Variants

Early on, we decided to include multiple filter units which implement various filtering algorithms in each port filter block. The motivation for this decision was to capture various characteristics of the memory references because some filtering units best capture time locality of memory references, whereas others capture reference streams. We will show in this paper that the combination of filtering algorithms achieves the highest snoop filtering rate, reducing the number of snoop requests up to 99%.

We have explored a number of snoop filter variants, but have selected the combination of a stream register based filter and a snoop cache. The snoop cache is essentially a Vector-Exclusive-JETTY [9]. It filters snoops using an algorithm which is based on the temporal locality property of snoop requests, meaning that if a single snoop request for a particular location was made, it is probable that another request to the same location will be made soon. This filter unit records a subset of memory blocks that are *not* cached.

The stream registers use an orthogonal filtering technique, exploiting the regularity of memory accesses. They record a superset of blocks that *are* cached. Results of both filter units are considered in a combined filtering decision. If either one of the filtering units decides that a snoop request should be discarded, then it is discarded.

The snoop cache filter unit keeps a record of memory references which are guaranteed not to be in the cache. These blocks have been snooped recently (thus invalidated in the cache) and are still not cached (i.e. they were not loaded into the cache since invalidation). The snoop cache filter unit contains a small array of address tags. An entry is created for each snoop request. A subsequent request for the same block will hit in the snoop cache, and be filtered. If the block is loaded in the processor cache, the corresponding entry is removed from the snoop cache, and any new coherency request to the same block will miss in the snoop cache and be forwarded to the processor cache. There is one dedicated snoop cache filter unit for each remote memory writer (processor, DMA, etc.) to allow for concurrent filtering of multiple coherency requests, thus increasing system performance.

A single snoop cache contains M snoop cache lines, each consisting of an address tag field, and a valid line vector. The address tag field is typically not the same as the address tag of the L1 data cache, but is reduced by the number of bits used for encoding a valid line vector. The valid line vector is a bit-vector that records the presence of individual, consecutive lines within an aligned block. Thus, N least significant bits from the address are decoded to a valid line vector with 2^N bits, where each bit of the vector effectively utilizes the remainder of the stored address, significantly increasing the snoop cache capacity efficiently. In the extreme case when N is zero, the whole entry in the snoop cache represents only one L1 data cache line, and the valid line vector has only one bit, corresponding to a "valid" bit.

4   Stream Registers

The stream register filter unit was introduced in [19]. Unlike the snoop cache that keeps track of what *is not* in the cache, the stream register filter keeps track of what *is* in the cache (i.e. it is an include filter). More precisely, the stream registers keep track of at

cache invalidate
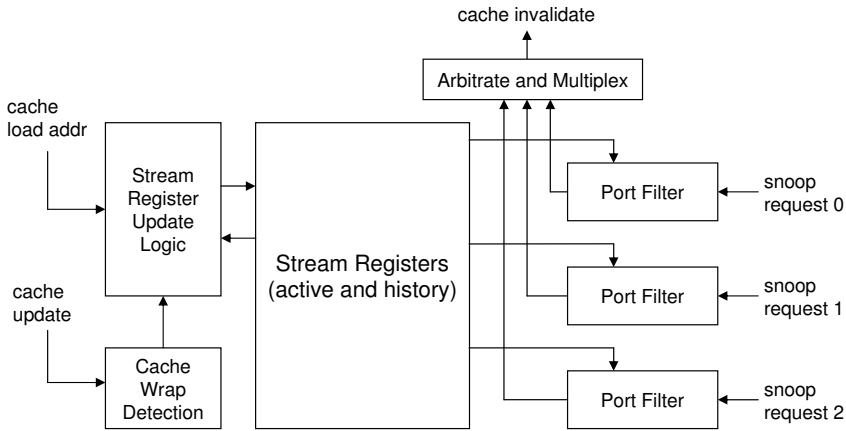
Fig. 2. Architecture of a snoop filter using stream registers. There are three port filters in order to handle snoop requests from all remote processors simultaneously.

least the lines that are in the cache, but may assume that some lines are cached which are not actually there. However, forwarding some unnecessary snoop requests to the cache does not affect correctness. The stream registers capture address streams, so they are advantageous for applications where too many spatially-distributed references overflow the snoop cache filter units.

The stream register filter unit consists of two sets of stream registers and masks (active and history sets), port filter logic, cache wrap detection logic, and stream register update logic, as illustrated in Figure 2. The stream registers and masks keep track of data which were recently loaded into the cache of the processor. One active stream register is updated every time the cache loads a new line, which is presented to the Stream Register Update Logic with appropriate control signals. A particular register is chosen for update based upon the current stream register state and the address of the new line being loaded into the cache.

Every remote snoop (snoop requests 0-2) is checked against the stream registers to see if it might be in the cache or not. This check can be performed in parallel because stream register lookups never change the state of the registers. Therefore, our architecture includes a port filter for each remote processor (or other snoop source such as DMA). Figure 2 shows three Port Filters. Snoop requests coming from one of the remote processors are checked in one of the Port Filters. Each arriving snoop requests address is compared with the state of the stream registers to determine if the snoop request could possibly be in the cache. In parallel, it is checked against a snoop cache, one of which exists in each Port Filter. If either the stream register or snoop cache lookup determine that the address is *not* in the L1 cache, then it is filtered out. Otherwise the request is forwarded to the Arbitration and Multiplexing interface and on to the cache. The Arbitrate and Multiplex logic shares the snoop interface of the cache between the Port Filters and queues unfiltered snoop requests, allowing for the maximum snoop request rate.

A stream register actually consists of a pair of registers (the base and the mask) and a valid bit. The base register keeps track of address bits that are common to all of the cache lines represented by the stream register, while the corresponding mask register keeps track of which bits these are. For example, if considering an address space of 2^{32} bytes with a cache line size of 32 bytes, a cache line load address is 27 bits in length, and the base and mask registers of the stream registers are also 27 bits in length.

Initially, the valid bit is set to zero, indicating that the stream register is not in use, and the contents of the base and mask registers are irrelevant. When the first cache line load address is added to this stream register, the valid bit is set to one, the base register is set to the line address, and all the bits of the mask register are set to one, indicating that all of the bits in the base register are significant. That is, an address that matches the address stored in the base register exactly is considered to be in the cache, while an address differing in any bit or bits is not.

At some point, another cache line load address will be added to this stream register. The new address is compared to the base register AND-ed with the mask register to determine which significant bits are different, and the mask register is then updated so that the differing bit positions become zeros in the mask. These zeros indicate that the corresponding bits of the base register are "don't-care", or can be assumed to take any value (zero or one). Therefore, these bits are no longer significant during comparisons to the stream register.

As an example, suppose the first two cache line load addresses are `0x1708fb1` and `0x1708fb2` (hexadecimal values). Then the contents of the stream register after these loads is:

Step 0: Base = `0x1708fb1`, Mask = `0x7ffffff`
Step 1: Base = `0x1708fb2`, Mask = `0x7fffffc`

As the second address and the base register differed in the two least significant bits, those bits are cleared in the mask register. At this point, the stream register indicates that the addresses `0x1708fb0`, `0x1708fb1`, `0x1708fb2`, and `0x1708fb3` can all be in the cache because it can no longer distinguish the two least significant bits.

Every cache line load address is added to exactly one of the multiple stream registers. Therefore, the collection of stream registers represents the complete cache state. The decision of which active register to update is made by the register update logic. In order to capture streams, we want addresses separated by the same stride to be added to the same stream register. We say that such addresses have an affinity for one another. We considered two policies for determining affinity and selecting which stream register to update:

- choose the stream register with minimal Hamming distance from the line load address (i.e. the stream register which will result in the minimum number of mask register bits changing to zero).
- choose the stream register where the most upper bits of the base register match those of the line load address.

Either mechanism guarantees that all addresses presented to the stream registers will be included within them.

Another issue is when to choose a new register instead of one that already contains a stream. We do this by assigning an "empty affinity" to unused registers and then including them in the update selection process. So when using the Hamming distance policy, for example, an empty register is chosen if the empty affinity is less than the affinity calculated for all used registers.

Over time, as cache line load addresses are added to the stream registers, they become less and less accurate in terms of their knowledge of what is actually in the cache. Every mask bit that becomes zero increases the number of cache lines that the corresponding stream register specifies as being in the cache, reducing the effectiveness of the stream register filtering. In the limit, the mask register becomes all zeros and every possible address is included in the register and considered to be in the cache.

Loss of accuracy is a disadvantage common to every snoop filtering technique that uses much less storage than the cache tag array. Snoop registers are specifically intended to remain accurate for strided streams, but they will not fare well with random addresses. To overcome the progressive loss of accuracy, the stream register snoop filter includes a mechanism for resetting the registers back to their initial condition. As there is no efficient way to remove an address from the stream registers and guarantee correctness, the filter clears the registers whenever the cache has been completely replaced, and they begin accumulating addresses anew. We call this complete replacement (relative to some initial state) a "cache wrap". The cache wrap detection logic monitors cache updates and determines when all of the cache lines present in the initial state have been overwritten. To do this, information must be provided by the L1 cache, either in the form of individual replacement notifications, or as a single event indicating that a wrap has occurred.

At that point, all of the stream registers are copied to a second "history" set of registers and masks and the "active" stream registers are all cleared to begin accumulating cache line load addresses anew. In addition, the state of the cache at the time of the wrap becomes the new initial state for the purpose of detecting the next cache wrap. The stream registers in the history set are never updated. However, they are treated the same as the active set by the Port Filters when deciding whether a snoop address could be in the cache. Their purpose is to make up for the fact that individual cache sets wrap at different times, but never survive two cache wraps.

The stream registers exploit periodic cache wrapping in order to refresh, and they rely upon knowing when the wraps occur. The second requirement is not difficult to implement, but does require a modified cache that either indicates all replacements, or tracks them from some point in time and indicates when all lines have been replaced.

Periodic cache wrapping is not guaranteed, but occurs frequently in practice. Caches with a round-robin replacement policy, such as those of the IBM PowerPC cores used in the Blue Gene/L and Blue Gene/P supercomputers, wrap relatively frequently. Specific wrapping behavior is a function of replacement policy and workload behavior. In either case, it is statistically possible that wrapping occurs infrequently, or never (i.e. a particular cache set is seldom or never used), but this pathological situation is not a threat to correctness. If this is a serious concern, then the stream register architecture could be extended to include a full cache invalidation and stream register reset when no wrap occurs for a long period of time.

5   Experimental Methodology

The experiments in this paper represent our top-down approach of validating the stream register ideas and showing that stream registers are effective for a broad range of applications. Our methodology trades detail for the ability to process huge traces of several applications. We feel that this is the best approach because the order of memory accesses is the critical factor in determining snoop filter effectiveness. That is, the order of a snoop request for some address relative to a snoop filter acquiring that address is what determines whether the filter rejects the address or not. Other papers (such as JETTY [9]) relate filter effectiveness to performance gain and power reduction. For example, the JETTY paper estimates that L2 cache snooping alone accounts for 33% of the total power of a typical 4-way SMP.

For our experiments, we used several applications from the publicly-available SPLASH-2 [20] benchmark suite. These are well known benchmarks containing shared-memory applications that have driven much research on memory system architectures and cache-coherence protocols. We have chosen to use these publicly-available codes because they are good representatives for a wide range of scientific applications, which is where we expect to see the most significant impact of CMPs. We have run the kernels (LU, FFT, Cholesky, and Radix), and some of the applications (Barnes, Ocean, Raytrace, and Fmm).

For each of the benchmarks chosen, we have simulated a full, four-processor application run and collected the entire L1 data cache miss sequence to determine coherence snoop requests. Table 1 shows the benchmarks used, the total number of accesses to memory, and the average percentage of misses in the L1 caches. Whereas the hit rate in the local cache of a processor is high, the percentage of hits in the caches of all other processors (we refer to such processors as "remote") is very low.

Table 1 shows that across all benchmarks virtually all coherency snoop requests will miss in the remote caches, this representing the total coherency snoop filter opportunity. Such small hit rates are due to the relatively small (32KB) first-level caches and highlight the importance of snoop filtering for CMP cache coherence, particularly when maintained between L1 caches. Although few snoops hit, the ones that do are essential and the ones that do not should be filtered. We also list the total number of snoop requests generated by all four processors collectively (i.e. the total number of writes).

We used a custom simulator written with Augmint [21] to collect the memory access traces. Augmint is a public-domain, execution-driven, multiprocessor simulation environment for Intel x86 architectures, running UNIX or Windows. Augmint does not include a memory backend, thus requiring users to develop one from scratch. We modeled the L1 data caches of four PowerPC 440 processors [22] and an ideal memory system below that (since we were only concerned with the order of accesses and their effect on snoop filtering rates). Then we collected a trace of all memory references, including the source processor and address.

We developed a custom back-end simulator to process the traces and produce the results in this paper. Because we wanted to measure the relative effectiveness of snoop filters over very long traces, we were not concerned with cycle accuracy. We were only concerned with the order of accesses and their effect upon the snoop filters and caches. Therefore, the trace entries are processed in order, and they have an instant, atomic

Table 1. SPLASH-2 benchmark characteristics. The low remote cache hit rate shows that almost all invalidation snoops are useless and can be eliminated.

Benchmark	Input parameters	Accesses to memory	Local cache hit rate	Remote cache hit rate	Total coherency accesses
Barnes	16K particles	1,602,120,476	99.73%	0.00047%	1,968,916,971
FFT	256K points	58,481,113	97.12%	0.0000057%	52,627,671
LU	512 matrix	202,643,933	99.24%	0.0000088%	204,434,958
Ocean	258 x 258 ocean	310,234,016	93.36%	0.03%	143,647,839
Cholesky	tk15.O	678,266,460	99.43%	0.00043%	614,572,560
FMM	16K particles	2,084,764,684	99.76%	0.00016%	2,976,937,884
Radix	10M keys	2,716,061,135	99.48%	0.00068%	3,491,931,132
Raytrace	car	404,977,091	98.43%	0.018%	358,731,051

effect upon the simulated caches and snoop filters. This simplification allowed us to compare many different alternative architectures, while exposing the significant trends. As a result, however, we could not measure execution times.

The back-end simulator models the private L1 data caches and snoop filters of four processors. We model the cache of the PowerPC 440 embedded processor, which is the building block for the Blue Gene/L supercomputer [8]. Each cache is 32KB in size and has a line size of 32 bytes. The caches are organized as 16 sets, each of which has 64 ways and utilizes a round-robin replacement policy.

While such high associativity may seem extreme, it is important to note that associativity is a characteristic that improves cache accuracy at the potential cost of cycle time. Overall, the gain in accuracy from high associativity can compensate for ever-increasing memory latencies, especially in a CMP where the overall performance is not as closely coupled to the cycle time as it is in a uniprocessor. Round-robin replacement is a viable, perhaps necessary, choice for a highly-associative cache because of its implementation simplicity (compared to LRU). Because we were only concerned with the order of accesses and their effect on snoop filtering rates, we did not model the memory system below the L1 caches.

The back-end simulator responds to loads and stores as follows:

- LOAD: Update the local cache. Then update the stream registers (insert the load address) and the snoop caches (delete the load address) of the local snoop filter.
- STORE: Update the local cache. Then update the three remote snoop filters, which includes a lookup to see if the snoop resulting from the store should be filtered, and a snoop cache update if it is not. Finally, propagate snoop invalidations to all remote caches for which the snoop request was not filtered.

6   Experiments and Simulation Results

We are interested in exploring the snoop filter design space to find the best compromise that yields a good filtering rate. We studied the impact of several design parameters. For stream registers, we considered their number, the replacement policy, and the empty affinity. For the snoop cache, we considered the number of entries and the size of the

valid line vector. The goal is to eliminate as many unnecessary coherence requests as possible, thus improving performance and reducing power dissipation. We also analyze the impact of using both snoop filter units together to exploit their combined strength, and explore several configurations to identify the optimal design point.

6.1   Stream Register Size

In order to determine the optimal number of stream registers, we have varied their number exponentially from 4 to 32, as shown in Figure 3. Not surprisingly, our experiments show that more stream registers filter a higher percentage of coherence snoop requests. But even when using only eight stream registers, we filter more than 90% of all snoop requests for three benchmark applications.

We observed that the effect of increasing the number of stream registers is not linear with respect to the snoop filter rate. Choosing only four stream registers is clearly a bad policy. For the SPLASH-2 benchmarks, selecting 8 or 16 stream registers seems to be the best compromise, whereas 32 stream registers (which doubles the area compared to 16 stream registers) only increases the snoop filter rate significantly for one benchmark.
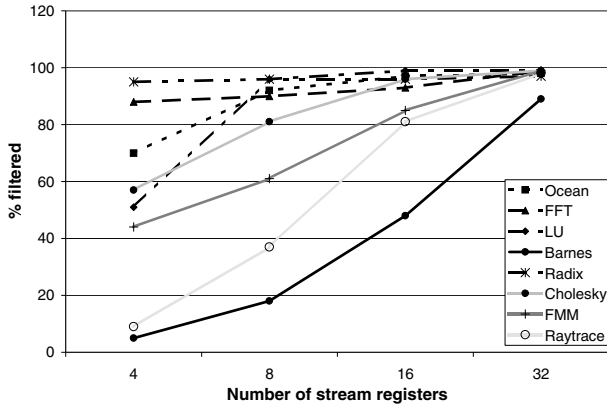
Fig. 3. Stream register filter behavior. Percentage of snoops filtered as the number of stream registers is increased.

6.2   Stream Register Update Policy

As previously described, every cache line load address is added to exactly one stream register. The register selected for update depends on the stream register update policy and the empty affinity value. We have evaluated two different selection policies:

– minimal Hamming distance, and
– most matching upper bits (MMUB).

For the minimal Hamming distance selection policy, we calculate the Hamming distance between each new load address and all stored values in the stream registers combined with their paired masks so that only relevant bits (i.e. bits that are 1 in the mask)
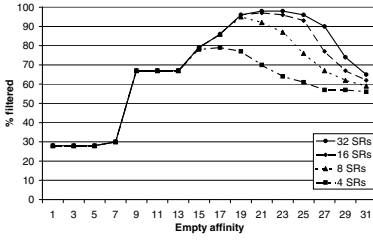
(a) Ocean benchmark
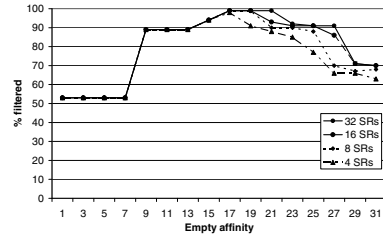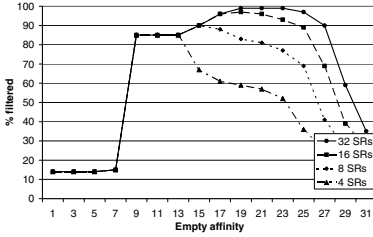
(b) FFT benchmark

(c) Cholesky benchmark
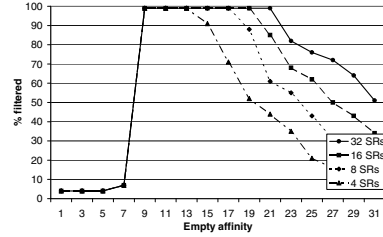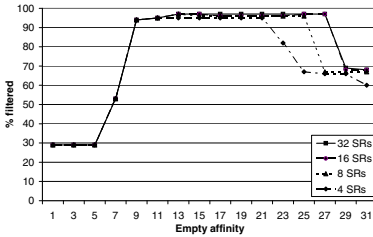
(d) FMM benchmark

(e) Radix benchmark
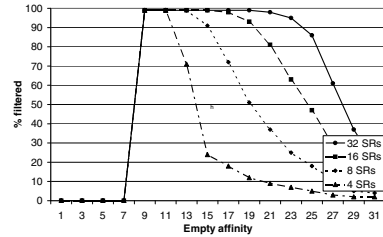
(f) Raytrace benchmark

Fig. 4. Percentage of snoops filtered as the empty affinity and number of stream registers is varied using the MMUB update policy

are considered. The affinity is the number of mask register bits that will be changed to 0, and the stream register with the minimum affinity is selected for update. In the MMUB update policy, we choose the stream register where the largest number of relevant upper bits match those of the line load address.

In our evaluation of stream register update policies, we have also varied the empty affinity value. As discussed in Section 4, empty affinity is the default threshold value assigned to an empty register. Figures 4(a) to 4(f) show the effect of varying the empty affinity for various stream register sizes using the MMUB update policy. If the empty affinity is set too low, empty stream registers are used to establish new streams even for memory accesses belonging to the same stream. As a result, the filter rate of the stream registers will be very low because few streams are captured. Similarly, setting the empty affinity value too high causes streams to share registers and obliterate each others mask bits, resulting in a low filter rate. When the empty affinity is increased

to more than 13, it starts to play a role in the filter rate, depending on the number of stream registers. For filters having a higher number of stream registers, a higher affinity value is advantageous because it allows for more sensitive stream determination. For configurations with a smaller number of stream registers, a lower affinity allows for the most effective stream discrimination. Overall, the optimal empty affinity value is about 19 for eight stream registers, and about 23 for 32 stream registers.
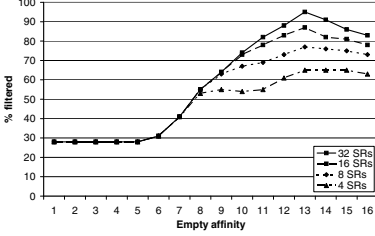
(a) Ocean benchmark
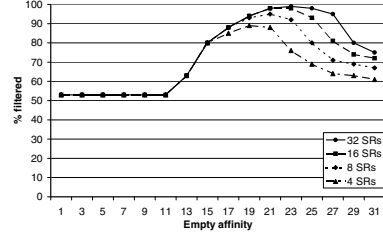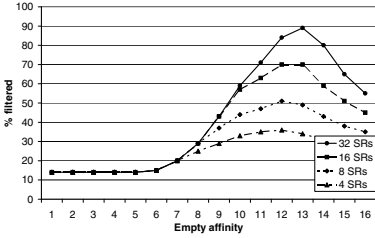
(b) FFT benchmark

(c) Cholesky benchmark
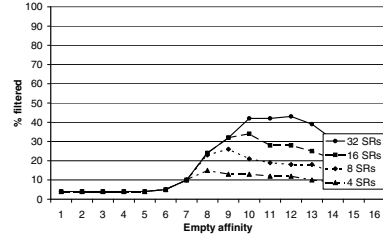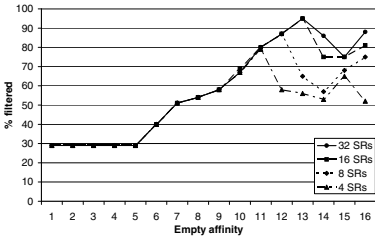
(d) FMM benchmark

(e) Radix benchmark
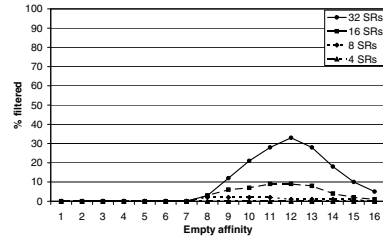
(f) Raytrace benchmark

Fig. 5. Percentage of snoops filtered as the empty affinity and number of stream registers is varied using the minimum Hamming distance update policy

Figure 5 shows the effect of varying the empty affinity for various stream register sizes using the minimum Hamming distance update policy. Similar to the MMUB update policy, setting the empty affinity too low or too high causes the filter rate of the stream registers to be low. The optimal empty affinity number is between 19 and 25, depending on the number of stream registers in the filter and the application's memory access pattern.

Across all applications, the sensitivity of the filter rate to the empty affinity value seem to be less for the MMUB update policy when compared with the minimum Hamming distance policy. In addition, for some applications - like Raytrace and FMM - the MMUB update policy significantly outperforms the Hamming distance policy. While MMUB achieves almost 100% filtering for these applications, the Hamming distance policy reaches less than 40%, even for the largest configurations.

The MMUB policy has the advantage of ignoring low-order address bits when establishing streams. The minimum Hamming distance policy results in well-correlated addresses that differ in their low-order address bits being mapped to different stream registers, thereby causing a kind of pollution which limits effectiveness.
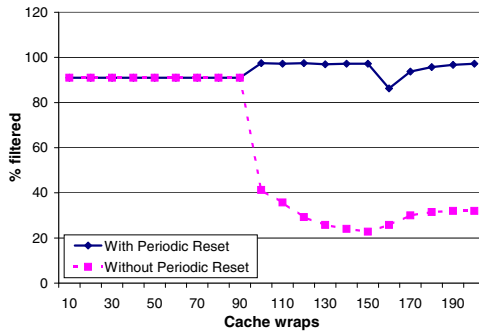
Fig. 6. Effectiveness of stream registers when implementing cache wrap for the FFT benchmark

6.3   Stream Register Clearing

As described in Section 4, the stream registers need to be cleared periodically in order to track changes in programs' L1 cache contents. The need for this is illustrated in Figure 6, which shows average, cumulative stream register effectiveness for a portion of the FFT benchmark running on a 4-processor CMP. During the course of the run, the caches wrapped 200 times.

Initially, the L1 working set is captured effectively by the stream registers and resetting provides no benefit. At around the 100th wrap, the working set changes, and the stream registers with reset can track the change. The stream registers without reset, however, forward many unnecessary snoops, causing their effectiveness to plummet. Although the stream registers without reset only become less accurate over time, the modest recovery in their effectiveness only indicates that the cache contents has changed in their favor.

6.4   Snoop Cache Size

In order to determine the optimal sizing for a snoop cache-based filter, we have varied two parameters: the number of lines, ranging from 4 to 64, and the number of bits used in the valid line vector, ranging from 1 to 64 (encoding 0 to 6 consecutive lines respectively). The results are illustrated in Figure 7 for several SPLASH-2 benchmarks.
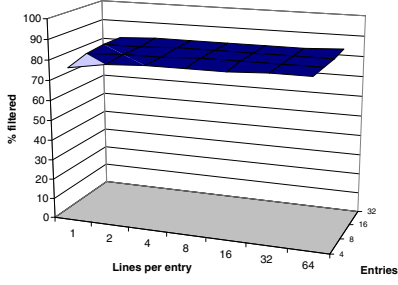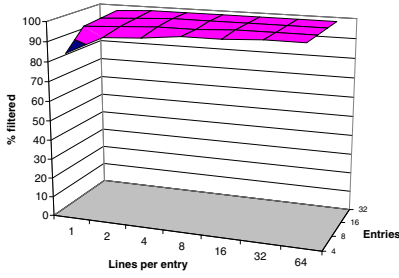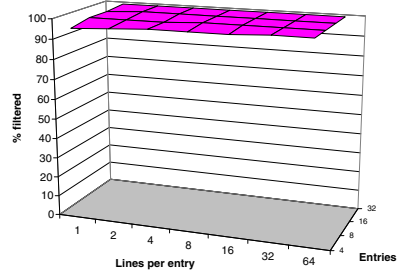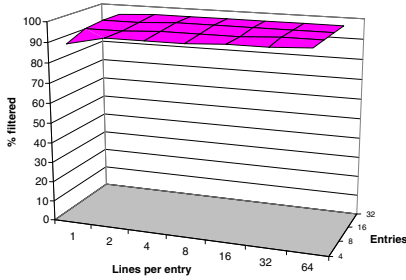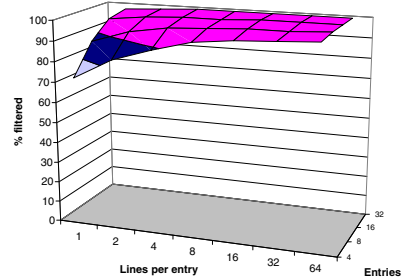
(a) Ocean benchmark
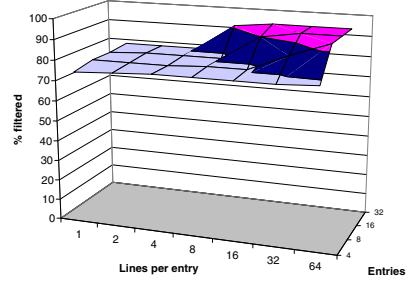
(b) FFT benchmark

(c) Cholesky benchmark

(d) FMM benchmark

(e) Radix benchmark

(f) Raytrace benchmark

Fig. 7. Snoop cache filter behavior. Percentage of snoops filtered as the number of entries and the number of lines per entry is varied.

Our experiments show that using filters with a greater number of snoop cache lines and/or a longer valid line vector are more effective at filtering snoop requests. But even for relatively small snoop caches having only 4 cache lines each, and with valid line vectors of 32 bits, we reach the snoop cache filter limit across the benchmark applications. The filter limit varies for various applications from 82% for Ocean to 99% for FMM.

We observe that for each application, depending of its memory access pattern, the shape of the cache size/valid vector line length surface differs. By increasing only the number of cache lines or only the length of the valid line vector, the maximum filtering possibility for a particular application can be reached. For example, Ocean (Figure 7(a)) reaches its maximum filtering rate of 83% for snoop cache filter units with almost minimal sizing, having only 4 cache lines and as little as 2-bit-long valid line vectors. Similarly, FMM (Figure 7(d)) reaches its filtering maximum of 99% with a small configurations of only 8 cache lines and 8-bit long valid line vectors.

On the contrary, the FFT benchmark reaches maximum filtering only for bigger configurations. As illustrated in Figure 7(b), at least 8 cache lines and 64-bit long valid line vectors are needed to reach the maximum filtering rate of 93%. The memory access pattern of FFT requires a higher number of cache lines because it has a high number of streams, and a higher number of bits in the valid line vector because these streams are longer.

The optimal snoop filter configuration achieves near maximum filtering across all benchmarks, requiring a minimal number of latches for its implementation. This translates directly to a minimal number of aggregated bits. The dependency on the number of bits required by each of the snoop cache sizes explored is illustrated in Figure 8.

We observe that the effect of increasing the number of cache lines and valid line vector length is not linear with respect to the area requirements. Whereas the snoop filtering rate varied symmetrically with the number of cache lines and the valid line vector length, the area requirement increases significantly when increasing the number of cache lines. Thus, snoop cache configurations with lower numbers of cache lines and longer valid line vectors are better design points. For the SPLASH-2 benchmarks, selecting 8 cache lines with 32- or 16-bit valid line vectors seems to be the best compromise.
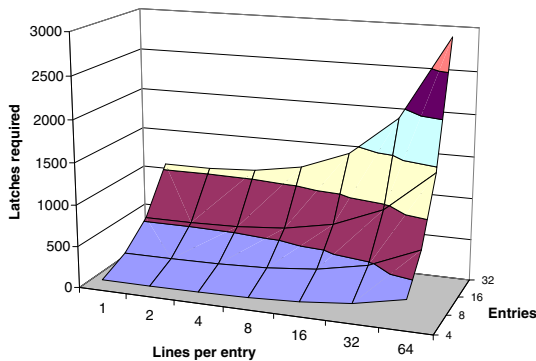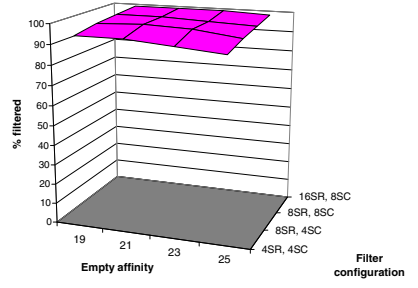

Fig. 8. Approximate number of latches required to implement a snoop filter unit with snoop caches, depending on the number of entries and the lines per entry
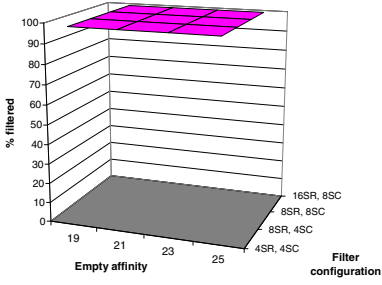
6.5   The Most Effective Combination

We have discussed and analyzed two snoop filters separately. As both filters cover different memory access patterns, the most effective filtering is achieved when putting the
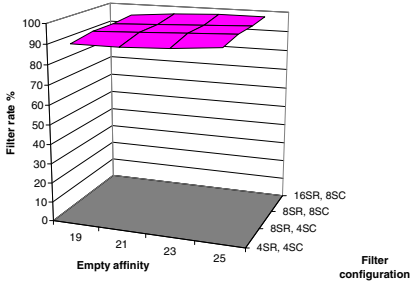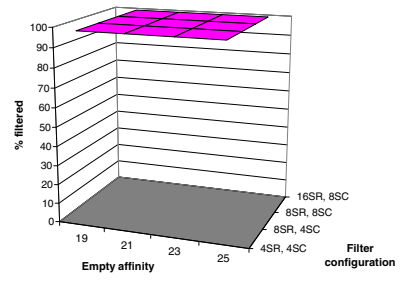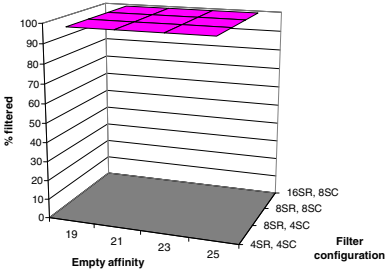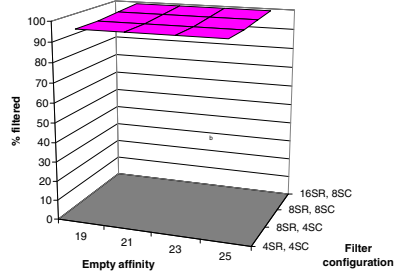
(a) Ocean benchmark

(b) FFT benchmark

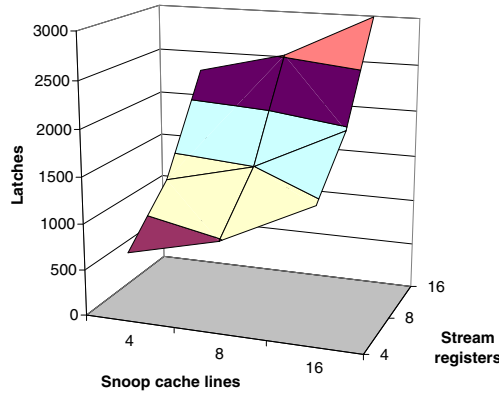(c) Cholesky benchmark

(d) LU benchmark

(e) Barnes benchmark

(f) Raytrace benchmark

Fig. 9. Combined filter behavior. Percentage of snoops filtered for several stream register and snoop cache configurations as the empty affinity is varied in its most effective range.

Fig. 10. Approximate number of latches required to implement a snoop filter unit with both snoop caches and stream registers, for several effective configurations

two filters together. We will show that using the combination of two filters, we can achieve high filtering rates even though each filter unit is quite small.

In order to determine the optimal sizing for our snoop filter, we have varied three parameters: the number of stream registers (4, 8, or 16), the number of snoop cache lines (4 or 8), and the empty affinity (in the most effective range from 19 to 25). We keep the snoop cache valid line vector at 32 bits in length. The results for several SPLASH-2 benchmarks are illustrated in Figures 9(a) to 9(f). Clearly, combining the two filtering techniques results in a highly effective combination across all of the benchmarks we studied.

Once again, we must consider the size of the combined filter in terms of latch count in order to determine the optimal configuration. Figure 10 shows how latch count varies with the number of stream registers and snoop cache lines for a valid line vector of 32 bits. The stream registers grow faster than the snoop caches, so the optimal configuration prefers larger snoop caches over more stream registers. The knee in the latch count and performance curves appears to be at 8 stream registers and 8 entries per snoop cache. By way of comparison, the L1 cache tags of the PowerPC 440 processor require more than an order of magnitude more latches.

7   Conclusion

With the emergence of commodity CMPs, we have entered the era of the SMP-on-a-chip. These high-performance systems will generate an enormous amount of shared memory traffic, so it will be important to eliminate as much of the useless inter-processor snooping as possible. In addition, power dissipation has become a major factor in chip density, so mechanisms to eliminate useless coherence actions will be important.

In this paper, we have described and evaluated a snoop filtering architecture that is appropriate for high-performance CMPs. Our architecture uses multiple,

complementary filtering techniques and parallelizes the filters so that they can handle snoop requests from all remote processors simultaneously.

We have described stream register snoop filtering, which uses a small number of registers to capture strided access streams. We explored the stream register and snoop cache design spaces using the SPLASH-2 benchmarks together with a custom trace generator and simulator.

Finally, we developed a highly-effective snoop filter that combines stream registers with snoop caches, and experimentally evaluated this design. We show that the combined filter can be very small in size, yet effective over all of the benchmarks we studied.

Acknowledgements

References

1. Srinivasan, V., Brooks, D., Gschwind, M., Bose, P., Zyuban, V., Strenski, P., Emma, P.: Optimizing pipelines for power and performance. In: Proceedings of the 35th Annual International Symposium on Microarchitecture, pp. 333–344. ACM/IEEE, Istanbul/Turkey (2002)
2. Salapura, V., et al.: Power and performance optimization at the system level. In: Proceedings of the 2nd International Conference on Computing Frontiers, pp. 125–132. ACM, Ischia (2005)
3. Gonzalez, R., Horowitz, M.: Energy dissipation in general purpose microprocessors. IEEE Journal of Solid State Circuits 31(9), 1277–1284 (1996)
4. Dennard, R., Gaensslen, F., Yu, H.-N., Rideout, V., Bassous, E., LeBlanc, A.: Design of ion-implanted MOSFETs with very small physical dimensions. IEEE Journal of Solid-State Circuits, 256–268 (1974)
5. IBM Blue Gene Team: Overview of the IBM Blue Gene/P project. IBM Journal of Research and Development 52(1/2) (January 2008)
6. Intel, Intel quad-core technology, `http://www.intel.com/technology/quadcore`
7. Gschwind, M., Hofstee, H.P., Flachs, B., Hopkins, M., Watanabe, Y., Yamazaki, T.: Synergistic processing in Cell's multicore architecture. IEEE Micro 26(2), 10–24 (2006)
8. Bright, A.A., Ellavsky, M.R., Gara, A., Haring, R.A., Kopcsay, G.V., Lembach, R.F., Marcella, J.A., Ohmacht, M., Salapura, V.: Creating the Blue Gene/L supercomputer from low power SoC ASICs. In: 2005 IEEE International Solid-State Circuits Conference on Digest of Technical Papers, pp. 188–189 (2005)
9. Moshovos, A., Memik, G., Falsafi, B., Choudhary, A.N.: JETTY: Filtering snoops for reduced energy consumption in SMP servers. In: Proceedings of the 7th International Symposium on High-Performance Computer Architecture, pp. 85–96 (2001)
10. Briggs, F., Chittor, S., Cheng, K.: Micro-architecture techniques in the Intel E8870 scalable memory controller. In: Proceedings of the 3rd Workshop on Memory Performance Issues, pp. 30–36 (June 2004)
11. Kant, K.: Estimation of invalidation and writeback rates in multiple processor systems, `http://kkant.gamerspace.net/papers/inval.pdf`
12. Aono, F., Kimura, M.: The Azusa 16-way Itanium server. IEEE Micro 20(5), 54–60 (2000)

13. Keltcher, C.N., McGrath, K.J., Ahmed, A., Conway, P.: The AMD opteron processor for multiprocessor servers. IEEE Micro 23(2), 66–76 (2003)
14. Chinthamani, S., Iyer, R.: Design and evaluation of snoop filters for web servers. In: Proceedings of the 2004 Symposium on Performance Evaluation of Computer Telecommunication Systems (July 2004)
15. Ekman, S., Dahlgren, F., Stenstrom, P.: TLB and snoop energy-reduction using virtual caches in low-power chip-multiprocessors. In: Proceedings of the 2002 International Symposium on Low Power Electronics and Design, pp. 243–246 (August 2002)
16. Moshovos, A.: Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In: Proceedings of the 32nd Annual International Symposium on Computer Architecture, pp. 234–245 (June 2005)
17. Saldanha, C., Lipasti, M.: Power efficient cache coherence. In: Proceedings of the Workshop on Memory Performance Issues (June 2001)
18. Salapura, V., Blumrich, M., Gara, A.: Design and implementation of the Blue Gene/P snoop filter. In: Proceedings of the 14th International Symposium on High-Performance Computer Architecture, pp. 5–14 (February 2008)
19. Salapura, V., Blumrich, M., Gara, A.: Improving the accuracy of snoop filtering using stream registers. In: Proceedings of the 8th MEDEA Workshop, pp. 25–32 (September 2007)
20. Woo, S., Ohara, M., Torrie, E., Singh, J., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 24–36. ACM, New York (1995)
21. Nguyen, A.-T., Michael, M., Sharma, A., Torrellas, J.: The Augmint multiprocessor simulation toolkit for Intel x86 architectures. In: Proceedings of 1996 International Conference on Computer Design, pp. 486–490 (October 1996)
22. IBM, IBM PowerPC 440 product brief (July 2006), `http://www-306.ibm.com/chips/techlib/techlib.nsf/products/PowerPC_440_Embedded_Core`