RC24376 (W0710-071) October 12, 2007
Computer Science

IBM Research Report

Design and Implementation of the Blue Gene/P Snoop Filter

Valentina Salapura, Matthias Blumrich, Alan Gara
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Design and Implementation of the Blue Gene/P Snoop Filter

Valentina Salapura, Matthias Blumrich, and Alan Gara

IBM Thomas J. Watson Research Center
Yorktown Heights, NY

Abstract

As multi-core processors 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 snoop filtering. Shielding each cache with a device that can squash snoop requests for addresses known not to be in cache improves performance significantly for caches that cannot perform normal load and snoop lookups simultaneously. In addition, reducing snoop lookups yields power savings.

This paper describes the design of the Blue Gene/P snoop filters, and presents hardware measurements to demonstrate their effectiveness. The Blue Gene/P snoop filters combine stream registers and snoop caches to capture both the locality of snoop addresses and their streaming behavior. Simulations of SPLASH-2 benchmarks illustrate tradeoffs and strengths of these two techniques. Their combination is shown to be most effective, eliminating 94-99% of all snoop requests using very few stream registers and snoop cache lines. This translates into an average performance improvement of almost 20% for the NAS benchmarks running on an actual Blue Gene/P system.

1. Introduction

Over the past 20 years, ever higher operating frequencies have been the main factor driving the performance growth of single-core processors. However, further increases in operating frequencies are increasingly hard to obtain with newer generations of technology [26]. One of the main reasons is the impact of wire delays as feature sizes continue to shrink [11]. To compensate, processors have adapted increasingly sophisticated microarchitectures [13], often at the cost of inefficiencies in power/performance.

While faster transistors and faster wires are increasingly hard to obtain, the application of Dennard's CMOS scaling theory [8] is continuing to deliver improvements in density. In response, several multi-core processors have been introduced over the past few years, such as the IBM POWER4 [1] and POWER5 [25] servers, the Intel Core Duo processors [10], the AMD Opteron quad-core processors [3], and the Cell Broadband Architecture [12]. On such multi-core processors, suitably scalable, multithreaded, parallel workloads show significant increases in performance with little or no degradation in the power/performance ratio [22].

IBM recently announced the Blue Gene/P supercomputer [14], successor to the highly successful Blue Gene/L machine [2, 6]. The Blue Gene family of supercomputers is based on multi-core nodes organized within a classic distributed-memory system architecture. The Blue Gene/P node builds upon its predecessor by doubling the number of cores to four, providing a completely coherent shared memory system, and more than doubling the network performance.

Like many embedded cores, the PowerPC 450 used in Blue Gene/P has an integrated first-level cache, and is designed to be used in conjunction with a multi-level cache, generally. Therefore, it does not provide a sophisticated hardware coherence protocol (such as MESI), but provides support for inclusion, consisting of a write-through mode and an input port for specifying addresses to be invalidated. In the case of the PowerPC 450, the first-level cache is single-ported, so lookups and externally-generated invalidations compete for access.

The Blue Gene/P node architecture implements small, dedicated second-level caches (one per core) and a shared third-level cache, so the hardware must maintain memory consistency between all of the first- and second-level caches. This is done by writing all stores through to the third-level cache, and invalidating all remote copies of every store in the first- and second-level caches. Thanks to the high bandwidth available on chip, the store addresses from any core can be broadcast in a point-to-point manner to the other three cores at full speed. However, the large number of invalidations received by each core could degrade performance for two reasons. First, there is

a physical bottleneck at the first-level cache invalidation port because it must be shared by all of the invalidations coming from three other cores and a network DMA engine. Second, the invalidations disrupt the normal cache behavior because the first-level cache is single-ported. The scientific applications that Blue Gene/P favors are generally written to avoid sharing between the cores. As a result, most of the invalidations applied to the caches are useless and could be eliminated.

Our solution was to introduce snoop filters to eliminate the vast majority of useless invalidations. There are two common classes of snoop filters: source-based and destination-based. Source based filters eliminate snoops (invalidations, in our case) before they are even sent to remote caches, while destination-based filters eliminate snoops at the remote destinations. Source-based filters are more appropriate to implement together with directory-based coherence because the directory tracks remote copies of cache lines. Conversely, destination-based filters are more appropriate to implement together with snooping coherence, where cache state is only kept local to each core, as is the case for Blue Gene/P. Therefore, we chose to implement a destination-based snoop filter for every core.

The remainder of this paper describes the design and implementation of the Blue Gene/P snoop filter, which utilizes our novel stream register technique [21]. Section 2 gives a brief overview of the Blue Gene/P system architecture, while Section 3 goes into the details of the snoop filter. Section 4 describes the simulations we performed in order to arrive at our design point, and Section 5 presents some preliminary performance measurements from an actual Blue Gene/P system. We comment on related work in Section 6 and conclude with Section 7. The contributions of this paper are to present our design methodology and to demonstrate a working implementation of a destination-based snoop filter.

2. Blue Gene/P System Overview

The Blue Gene/P supercomputer is a scalable, distributed-memory system consisting of up to 262,144 nodes. Each node is built around a single compute ASIC with 2 GB or 4 GB of external DDR2 DRAM. The compute ASIC is a highly integrated System-on-a-Chip (SoC) chip multiprocessor (CMP). It contains four PowerPC 450 embedded processor cores [15], each with private, highly-associative, 32 KB first-level instruction and data caches. Each core is coupled to a dual-pipeline SIMD floating-point unit and to a small, private, second-level cache whose principal responsibility is to prefetch streams of data. In addition, the chip integrates an 8 MB, shared third-level cache, two memory controllers, five network controllers, and a performance monitor, as illustrated in Figure 1.

The PowerPC 450 microprocessor is a high-performance, out-of-order industry-standard PowerPC processor originally targeted at high-end embedded systems. The processor supports 2-way superscalar instruction execution with a seven stage pipelined microarchitecture. The processor cores include 32KB first-level instruction and data caches organized as 16 associative sets with 64 ways per set.

A dual-pipeline, SIMD floating point unit is attached to each processor core. The floating point unit can execute two fused multiply-add instructions per cycle for a peak floating point performance of 13.6 GFLOPS/node. The floating point unit pairs two floating-point register files and two execution pipes. The primary and secondary register files are independently addressable, but they can be jointly accessed by SIMD instructions. SIMD execution exploits the data-level parallelism often present in high-performance computing workloads to reduce the number of instructions that must be executed, while increasing the number of operations completed.

Like its predecessor, Blue Gene/P provides five dedicated communication networks: the torus network, the collective network, the barrier network, 10Gb/s Ethernet, and IEEE1149.1 (JTAG). The network interfaces are integrated on the same chip as the processing units. The main network is the torus, which provides high performance data communication to nearest neighbor nodes in a 3D configuration with low latency and high throughput. The collective network supports efficient collective operations, such as broadcast and reduction, and serves as the I/O interconnect. A more detailed description of Blue Gene/P can be found in [14].

3. Snoop Filter Architecture

In symmetric multiprocessor (SMP) architectures, 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 [23][19]. This is particularly true of supercomputing applications where data partitioning and data blocking is performed to increase locality of reference and optimize overall compute performance. This motivated us to design a hardware device that filters out incoming snoop requests, reducing the number of actual snoop requests presented to the cache. In theory, a completely accurate filter can be created by duplicating the cache tag array and filtering out all snoops that miss. However, there are very significant technical realities that make this solution infeasible. For example, the SOC design flow makes it virtually impossible to modify a macro, such as the PowerPC core, or extract a piece of it, such as the cache tags. Furthermore, designing our own duplicate tag array with memory macros and gates

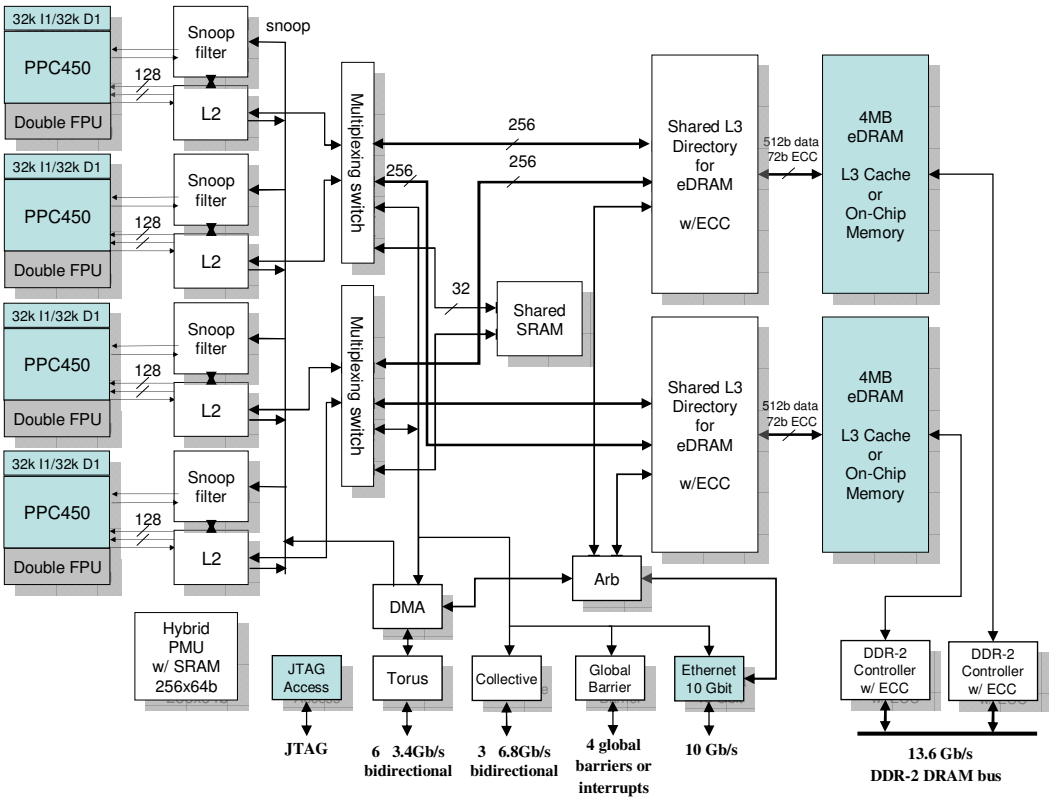


Figure 1. Blue Gene/P node architecture.

would never achieve the performance of the hand-placed L1 cache tags, so it would fall behind. Fortunately, it has been shown that very accurate filtering can be achieved with small designs that conservatively approximate the cache contents and operate at a reasonable frequency.

As mentioned in Section 1, data integrity between the Blue Gene/P cores is maintained with a cache coherence protocol based on write-invalidates, with all L1-caches operating in write-through mode. Every store not only updates the L1-cache of the issuing core, but also sends the write data via the L2 write buffer to the shared L3 cache. The L2s broadcast an invalidate request for the write address to ensure that no "stale" copy of the same datum will remain in the other L1s and L2s. We introduce our snoop filter at each of the four processors, located outside the L1 caches, as shown in Figure 2.

Each snoop filter receives invalidation requests from three remote cores and the network DMA by way of a point-to-point interconnect, so it must process requests from four memory writers concurrently (Figure 2). To handle these simultaneous requests, we implement a separate snoop filter block, or "port filter", for each interconnect port. Thus, coherency requests on all ports are processed concurrently, and a small fraction of all requests are forwarded to the processor. For example, each snoop filter in Figure 2 has four separate port filters (as shown in Figure 3), each of which handles requests from one remote processor or the network DMA unit.

Early on, we decided to include multiple filter units which implement various filtering algorithms in each port filter in order to capture various characteristics of the memory references. Some filtering units best capture time locality of memory references, whereas others capture reference streams. Through extensive simulation, we confirmed that the combination of various filtering algorithms achieves the highest filtering rate (reducing the number of snoop requests up to 99%, as shown in Section 4). We explored a number of snoop filter variants, and selected the combination of a snoop cache, a stream register filter, and a range filter.

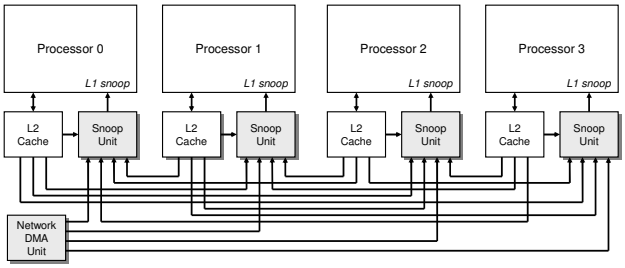


Figure 2. Blue Gene/P point-to-point snoop interconnect. All stores are sent by each L2 cache to all remote caches for invalidation. In addition, all network DMA stores are sent to all processor caches for invalidation.

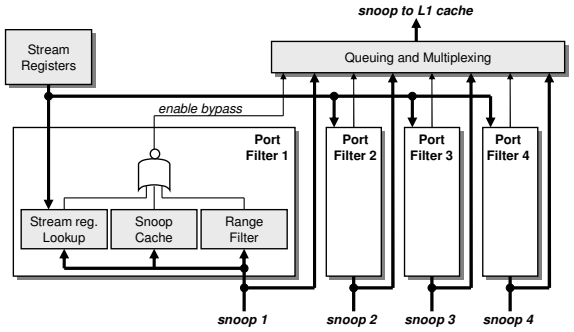


Figure 3. Architecture of a snoop unit including stream registers, four port filters, and queuing logic. Each port filter contains three separate filters, any of which can assert its output to filter the snoop. Otherwise, the snoop address bypasses the filter and is enqueued for invalidation.

The snoop cache is essentially a Vector-Exclusive-JETTY [19], which exploits the temporal locality property of some snoop requests. It records blocks that have been snooped recently (thus invalidated in the cache). It consists of a small, direct-mapped array, where an entry is created for each snoop request. A subsequent snoop request for the same block will match in the snoop cache and be filtered. If the block is loaded in the processor's L1 cache, the corresponding entry is removed from the snoop cache, and any new snoop request to the same block will miss in the snoop cache and be forwarded to the L1 cache. There is one dedicated snoop cache filter unit for each memory writer (three processors and the DMA) to allow for concurrent filtering of multiple coherency requests, thus increasing system performance.

Because the snoop cache is intended to capture spatial (as well as temporal) locality, storage efficiency is dramatically increased by using each entry to cache consecutive addresses in an aligned block. That is, each entry stores a base address together with a bit vector that indicates the presence of individual addresses offset from the base. The base address is essentially the address tag of the L1 data cache reduced by five bits that are used for encoding the presence vector. The presence vector encodes a group of 32 consecutive, aligned cache lines of the L1 data cache. Further details can be found in [21].

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. More precisely, the stream registers keep track of the lines that are in the cache, but may assume that some lines are cached which are not actually there. The stream registers capture address streams, so they are advantageous for applications where too many spatially-distributed references overflow the snoop caches.

The stream register filter has been described in detail in [21], and we provide a summary here. The heart of the filter is the stream registers themselves, of which there are a small number. One of these registers is updated with the line address every time the cache loads a new line. 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 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, each port filter includes logic to compare the shared stream register state to its unique incoming snoop addresses.

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. More precisely, the mask register indicates which bits are "don't-care" and which are not. An address matches a stream register if all of the bits which are *not* don't-care match. For example, a stream register with base 0x12345678 and mask 0xFFFFFF7F (where 0 means don't-care) matches addresses 0x12345678 and 0x123456F8.

Every load address (resulting from an L1 miss) is merged with one of the stream registers. Bits that differ between the register base address and the load address cause the corresponding mask bits to be changed to don't-care (if they are not already in that state). Therefore, two primary design issues emerge: how to choose which register to merge with,

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

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

and how to deal with the loss in accuracy (which becomes worse over time).

Our first impulse for deciding which register to merge with was to calculate the Hamming distance between the load address and each of the base registers (taking the mask into account), and then choose the minimum, thereby causing the smallest number of mask bits to be changed to don't-care. After careful consideration, we decided that the upper address bits should be favored in order to capture streams, where lower address bits would be expected to vary frequently. Therefore, we devised the "Most Matching Upper Bits" scheme which favors registers where the upper mask bits do not change. The basic idea is to choose the register with the longest matching string of consecutive bits, starting with the high-order bit. As shown in Section 4, this scheme was found to be superior and it is what we implemented. A related issue is when to choose a new register instead of one that already contains a stream. We do this by assigning a default distance (which we called the "empty affinity") to unused registers and then including that in the update selection process.

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. In the limit, some mask register becomes all don't-care and every possible address is considered to be in the cache and cannot be filtered. To overcome this, 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 registers are cleared whenever the L1 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 snoop filter contains logic that tracks cache wrapping based on notifications from the L1 cache every time a line is replaced. Because the PowerPC 450 cache uses round-robin replacement within sets, this logic basically consists of sixteen counters, one per set. The stream registers cannot simply be reset when the cache wraps because they contain all the addresses that caused the wrap. Therefore, they are copied into a duplicate "history" set that is never updated, but participates in lookups. Once the cache wraps a second time, it is safe to discard the history set, so it is overwritten on every wrap in a pipelined manner.

We added a third filter, called the range filter, which unconditionally filters all snoops within a specified address range (or optionally, outside the range). This filter is useful when the four processors are utilizing completely distinct and contiguous sections of physical memory because it insures complete filtering.

Results of all three filter units are considered in a combined filtering decision. If any one of the filtering units decides that a snoop request should be filtered, then it is discarded. Otherwise, the snoop request is queued and forwarded to the L1 cache, as shown in Figure 3.

4. Design Space Exploration

The experiments in this paper represent our top-down approach for finding the best snoop filter design point. For our experiments, we used several codes from the publicly-available SPLASH-2 benchmark suite [24, 27]. We chose to use these 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 Blue Gene/P. We have run the kernels (LU, FFT, Cholesky, and Radix), and some of the applications (Barnes, Ocean, Raytrace, and FMM). Table 1 shows the benchmarks used, the total number of memory accesses for all four processors, and the average percentage of misses in the L1 cache.

The analysis of the cache miss traces collected showed that the hit rate in the local cache of a processor is high, but the percentage of hits in the L1 data caches of all other processors (a.k.a. "remote" processors) is very low. Virtually all snoop requests will miss in the remote caches, representing the total 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

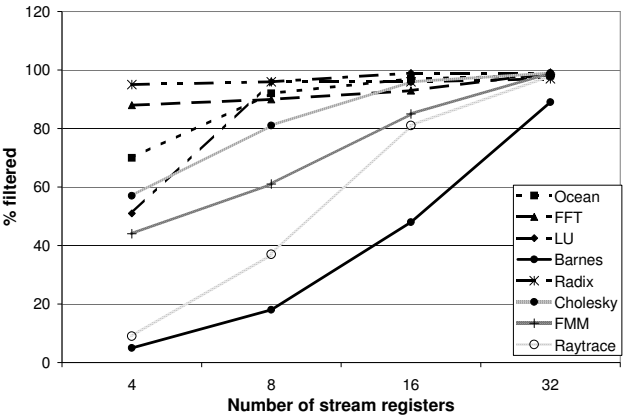


Figure 4. Percentage of snoops filtered as the number of stream registers is increased.

our architecture.

To collect the memory access traces, we used a custom simulator written with Augmint [20], a public-domain, execution-driven multiprocessor simulation environment. Augmint does not include a memory backend, thus requiring us to develop one from scratch. We modeled the L1 data caches of our four PowerPC 450 processors and an ideal memory system below that (since we were only concerned with the order of accesses and their effect on snoop filtering rates).

We developed a custom back-end simulator to process the traces and produce the results in this section. Because we wanted to measure the relative effectiveness of snoop filters over very long traces, we were not concerned with cycle accuracy, but only 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 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 actual execution times.

4.1 Stream Register Analysis

In order to determine the optimal number of stream registers, we have varied their number exponentially from 4 to 32, as shown in Figure 4. Not surprisingly, 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.

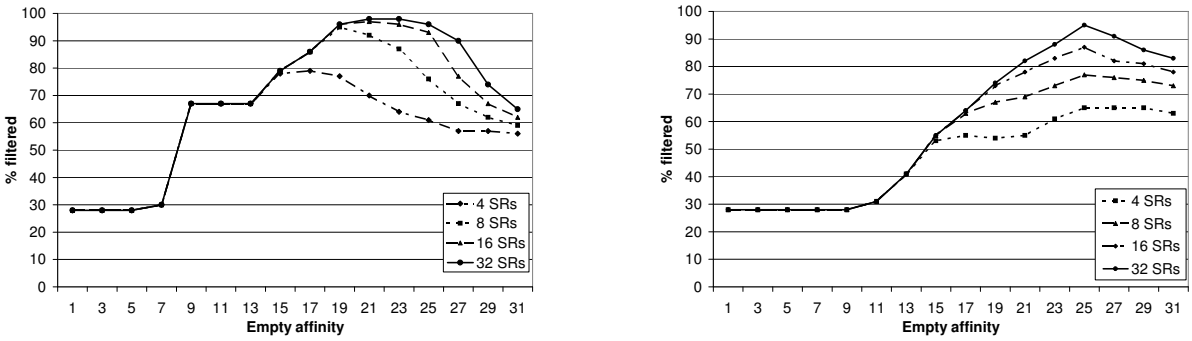
We observed that the effect of increasing the number of stream registers is not linear with respect to the snoop filtering rate. For the SPLASH-2 benchmarks, choosing only four stream registers is clearly a bad design point. Selecting 8 or 16 stream registers seems to be the best compromise, whereas 32 stream registers (which doubles the area compared to 16 stream registers) only increases the snoop filtering rate significantly for one benchmark.

We have evaluated two different selection policies to choose the stream register for update, as described in Section 3: minimal Hamming distance, and most matching upper bits (MMUB). Figure 5(a) shows the effect of varying the empty affinity for various stream register sizes using the MMUB update policy for the Ocean application. We illustrate only one application here due to space constraints, but the results for other benchmarks are similar.

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, resulting in a low filtering rate because few streams are captured. Similarly, setting the affinity value too high causes streams to share registers and obliterate each other's mask bits, resulting in a low filtering rate. When the empty affinity is increased to more than 13, it starts to play a role in the filtering 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. For example, the optimal empty affinity value is 19 for eight stream registers, and 23 for 32 stream registers.

Figure 5(b) shows the effect of varying the empty affinity for various stream register sizes using the minimum Hamming distance update policy for the Ocean application. The results for the other benchmarks have similar trends, although the maximal filtering rate achieved by Raytrace and FMM was substantially lower than the others (33% for Raytrace; 43% for FMM). Similar to the MMUB update policy, setting the empty affinity too low or too high causes the filtering rate of the stream registers to be low. Although the optimal minimum Hamming distance empty affinity value is fixed at 25 for the Ocean application, the general trend is the same as for the MMUB policy over all the codes we studied.

Across all benchmarks, the sensitivity of the filtering rate to the empty affinity value was less for the MMUB update policy. In addition, for Raytrace and FMM, the MMUB update policy achieves almost 100% filtering, while the Hamming distance update policy reaches less than 50%, even for the largest configurations. The MMUB policy has the advantage of ignoring low-order address bits when establishing streams in the stream registers. 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



(a) MMUB

(b) Hamming

Figure 5. Stream register filter behavior. Percentage of snoops filtered as the empty affinity and number of stream registers is varied for the Ocean application.

causing a kind of pollution. In this case, the effectiveness of stream registers is limited.

4.2 Snoop Cache Analysis

In order to determine the optimal sizing for a snoop cache-based filter, we have varied two parameters: the number of entries, ranging from 4 to 32, and the number of consecutive lines tracked by each entry, ranging from 1 to 64 (as determined by the length of the presence vector). The results for FFT, Ocean and Raytrace are shown in Figure 6. Results for the other benchmarks were similar.

Our experiments show that filters with a greater number of snoop cache entries and/or a longer presence vector are more effective at filtering snoop requests, primarily because of their larger capacity. The filter limit varies for various applications from 83% for Ocean to 99% for Raytrace. For each application, the shape of the cache size vs. presence vector size surface differs, depending on its memory access pattern.

FFT reaches its maximum filtering rate only for bigger configurations, while Ocean never exceeds a filtering rate of 83%. Raytrace is characteristic of several of the benchmarks that do very well with most snoop cache configurations.

4.3 Combining Both Filters

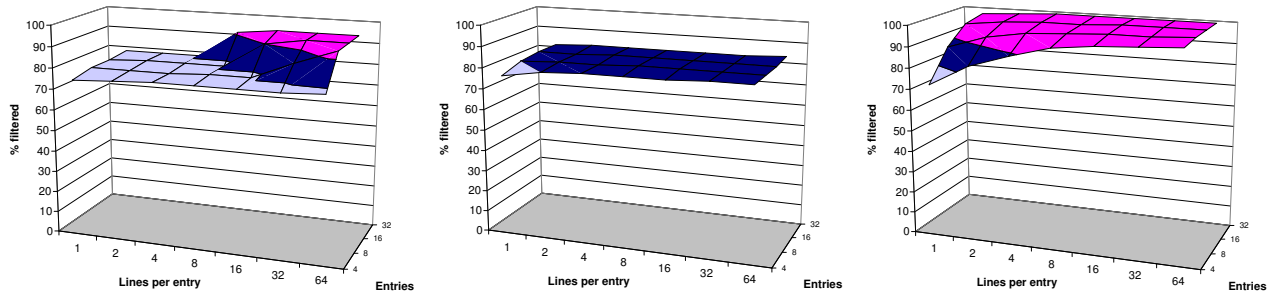
We have discussed and analyzed two snoop filters separately. As both filters cover different memory access patterns, the most effective filtering is achieved when putting the two filters together. We will show that using the combination of two filters, we can achieve high filtering rates even though 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 presence vector at 32 bits in length. Figure 7 shows results for the same benchmarks shown in the previous figures. Results for the other benchmarks are similar, showing a very high filtering rate for all configurations. Obviously, the two filtering techniques complement each other to obtain near-perfect filtering, even for filter configurations with a modest latch count.

5. Hardware Measurements

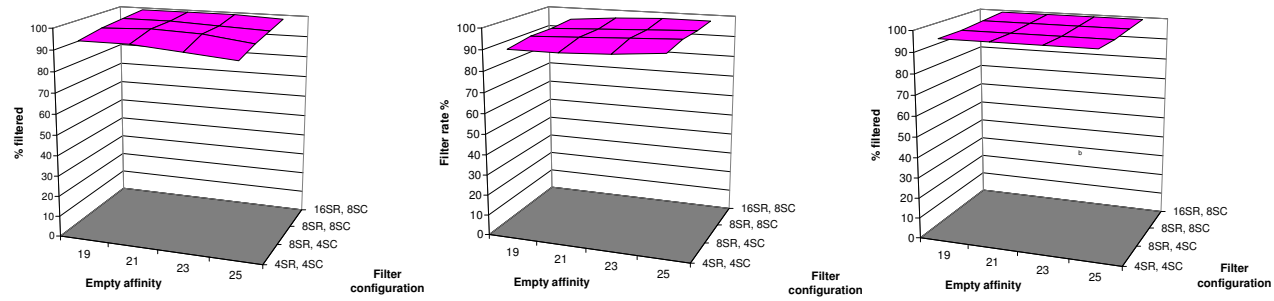
Based on our design space exploration, the Blue Gene/P system architecture implements the snoop filters in a point-to-point connection with four port filters, each having stream register lookup logic, a snoop cache, and a range filter. Based on our analysis, each port filter implements eight stream registers and eight snoop cache lines, each with a 32-bit valid line vector.

We recently brought up the first systems built with the Blue Gene/P compute ASICs, and we were able to make some preliminary hardware measurements. We measured runtimes for the NAS benchmarks both with and without the



(a) FFT (b) Ocean (c) Raytrace

Figure 6. Snoop cache filter behavior. Percentage of snoops filtered as the number of snoop cache sets and the number of lines per set is varied.



(a) FFT (b) Ocean (c) Raytrace

Figure 7. Combined filter behavior. Percentage of snoops filtered for several stream register and snoop cache configurations as the empty affinity is varied.

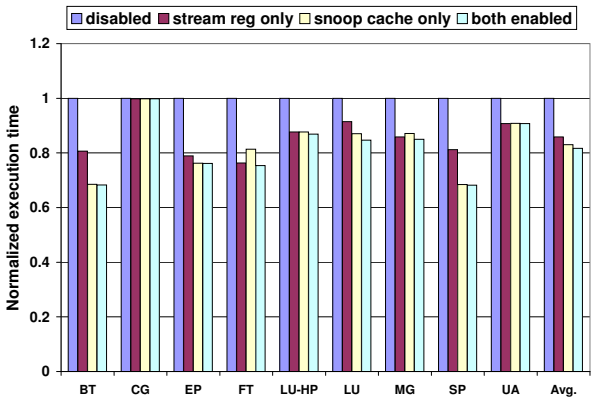


Figure 8. Normalized execution times of the NAS benchmarks on a Blue Gene/P node for various snoop filter configurations.

snoop filters enabled to see the performance effect. Figure 8 shows the normalized execution times for four snoop filter configurations: all snoop filters disabled, stream registers only, snoop caches only, and both snoop filters enabled.

Across all benchmarks, snoop filters reduce execution time. Most benchmarks benefit more from snoop caches (like BT and SP), while some get better performance with stream registers (like FT). The only exception is CG, whose execution time stays the same independent of the snoop filtering. For all benchmarks, the combination of the two snoop filters yields the best result, and reduces the execution time over 30% for BT and SP. On average, execution time is reduced by about 20%.

The hardware measurements confirm the effect of the significant reduction in coherence traffic shown by the simulations. The coherence traffic reduction translates into significant improvement in performance due to the use of the snoop filter. This confirms our simulation strategy to select an effective design point.

6. Related Work

Moshovos et al. [19] describes a snoop filter called JETTY that combines two complementary filtering methods. JETTY 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 cache, so it can be filtered. The include filter consists of several scoreboard arrays that track disjoint subsets of cache lines present in cache.

Moshovos et al. argue for snoop filtering as a means for power savings. However, our work is primarily motivated by the need to filter useless snoops that reduce performance. We also consider chip area and power consumption to be significant constraints, causing us to look beyond the simple and accurate method of duplicating the cache tags as a filter.

Several coherent network switches contain source-based 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 [5]. In this case, the snoop filter tracks the state of all cache lines within a 4-processor node for a system with up to 4 such nodes. Kant [16] modeled a similar system architecture with such a snoop filter. This architecture is also described in the Azusa system [4], which is based on Intel Itanium processors and may use an Intel chipset.

In [17], 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 [7], 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 CMPs, can be located at each processor in order to shield each from unnecessary snoops without changing the overall coherence scheme. Ekman et al. [9] describe a CMP architecture with Page Sharing Tables that are exclude filters at the granularity of memory pages, rather than cache lines. This architecture is more complicated in that the Page Sharing Tables coordinate to track sharing rather than just presence.

The idea of preventing remote snoop requests from being broadcast can also be applied at the chip level [18]. 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.

7. Conclusion

With the emergence of commodity multi-core processors and 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 with increased chip density, so mechanisms to eliminate useless coherence actions will be important.

In this paper, we have described and evaluated a snoop filtering architecture for the Blue Gene/P supercomputer, and presented some preliminary performance measurements. Our snoop filter uses multiple, complementary filtering techniques, and parallelizes the filters so that they can handle snoop requests from all remote processors

simultaneously. We explored the design space using the SPLASH-2 benchmarks together with a custom trace generator and simulator. Our Blue Gene/P measurements confirm the direct positive effect that the snoop filters have on performance.

Acknowledgements

The Blue Gene/P project has been supported and partially funded by Argonne National Laboratory and Lawrence Livermore National Laboratory on behalf of the United States Department of Energy under Subcontract No. B554331. The authors would like to thank Ruud Haring, and Michael Gschwind for their valuable contributions to this paper.

References

[1] Special issue on the IBM POWER4 system. *IBM Journal of Research and Development*, 46(1), January 2002.

[2] Special issue on the IBM Blue Gene/L system. *IBM Journal of Research and Development*, 49(2/3), March/May 2005.

[3] Advanced Micro Devices. AMD multi-core technology. http://multicore.amd.com, 2007.

[4] F. Aono and M. Kimura. The Azusa 16-way Itanium server. *IEEE Micro*, 20(5):54–60, September/October 2000.

[5] F. Briggs, S. Chittor, and K. Cheng. Micro-architecture techniques in the Intel E8870 scalable memory controller. In *Proceedings of the 3rd Workshop on Memory Performance Issues*, pages 30–36, June 2004.

[6] A. A. Bright, M. R. Ellavsky, A. Gara, R. A. Haring, G. V. Kopcsay, R. F. Lembach, J. A. Marcella, M. Ohmacht, and V. Salapura. Creating the Blue Gene/L supercomputer from low power SoC ASICs. In *Digest of Technical Papers, 2005 IEEE International Solid-State Circuits Conference*, pages 188–189, 2005.

[7] S. Chinthamani and R. Iyer. Design and evaluation of snoop filters for web servers. In *Proceedings of the 2004 Symposium on Performance Evaluation of Computer Telecommunication Systems*, July 2004.

[8] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFETs with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, pages 256–268, 1974.

[9] S. Ekman, F. Dahlgren, and P. Stenstrom. 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*, pages 243–246, August 2002.

[10] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. Introduction to Intel Core Duo processor architecture. *Intel Technology Journal*, May 2006.

[11] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid State Circuits*, 31(9):1277–1284, September 1996.

[12] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, March 2006.

[13] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. A novel SIMD architecture for the CELL heterogeneous chip-multiprocessor. In *Hot Chips 17*, Palo Alto, CA, August 2005.

[14] IBM Blue Gene team. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1/2), January 2008.

[15] International Business Machines. PPC450Ax6 embedded processor core. *Users Manual, SA14-2754-04*, September 2006.

[16] K. Kant. Estimation of invalidation and writeback rates in multiple processor systems. http://kkant.gamerspace.net/papers/inval.pdf.

[17] C. Keltcher, K. McGrath, A. Ahmed, and P. Conway. The AMD opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, March/April 2003.

[18] A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 234–245, June 2005.

[19] A. Moshovos, G. Memik, B. Falsafi, and A. N. Choudhary. JETTY: Filtering snoops for reduced energy consumption in SMP servers. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, pages 85–96, 2001.

[20] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The Augmint multiprocessor simulation toolkit for Intel x86 architectures. In *Proceedings of 1996 International Conference on Computer Design*, pages 486–490, October 1996.

[21] V. Salapura, M. Blumrich, and A. Gara. Improving the accuracy of snoop filtering using stream registers. In *Proceedings of the 8th MEDEA Workshop*, pages 25–32, September 2007.

[22] V. Salapura et al. Power and performance optimization at the system level. In *Proceedings of the 2nd International Conference on Computing Frontiers*, pages 125–132, Ischia, Italy, May 2005. ACM.

[23] C. Saldanha and M. Lipasti. Power efficient cache coherence. In *Proceedings of the Workshop on Memory Performance Issues*, June 2001.

[24] J. Singh, W.-D. Weber, and A. Gupta. Splash: Stanford parallel applications for shared memory. *Computer Architecture News*, pages 5–44, March 1992.

[25] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5), July 2005.

[26] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. Strenski, and P. Emma. Optimizing pipelines for power and performance. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 333–344, Istanbul, Turkey, November 2002. ACM/IEEE.

[27] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36. ACM, June 1995.