Subspace Snooping: Filtering Snoops with Operating System Support

Daehoon Kim, Jeongseob Ahn, Jaehong Kim, and Jaehyuk Huh

Dept. of Computer Science, KAIST

{daehoon, jeongseob, jaehong, and jhuh}@calab.kaist.ac.kr

ABSTRACT

Although snoop-based coherence protocols provide fast cache-to-cache transfers with a simple and robust coherence mechanism, scaling the protocols has been difficult due to the overheads of broadcast snooping. In this paper, we propose a coherence filtering technique called *subspace snooping*, which stores the potential sharers of each memory page in the page table entry. By using the sharer information in the page table entry, coherence transactions for a page generate snoop requests only to the subset of nodes in the system (subspace). However, the coherence subspace of a page may evolve, as the phases of applications may change or the operating system may migrate threads to different nodes. To adjust subspaces dynamically, subspace snooping supports a shrinking mechanism, which removes obsolete nodes from subspaces.

Subspace snooping can be integrated to any type of coherence protocols and network topologies. As subspace snooping guarantees that a subspace always contains the precise sharers of a page, it does not restrict the designs of coherence protocols and networks. We evaluate subspace snooping with Token Coherence on un-ordered mesh networks. For scientific and server applications on a 16-core system, subspace snooping reduces 44% of snoops on average.

Categories and Subject Descriptors

C.1.2 [**Processor Architecture**]: Multiple Data Stream Architectures (Multiprocessors); B.3.2 [**Memory Structures**]: Design Styles—*shared memory*

General Terms

Performance

Keywords

cache coherence, snoop filtering, subspace snooping

PACT'10 September 11–15, 2010, Vienna, Austria.

1. INTRODUCTION

The two broad classes of coherence protocols, snooping protocols and directory protocols, have traditionally targeted different scales of systems. Snooping systems offer low-cost, simple coherence at small system scales. Directory protocols have been built to scale much higher, but at greater cost and complexity. In snooping protocols, there is no explicit storage, or directory, to track the sharing states of memory blocks, and coherence requests must be broadcast to all the nodes. Such broadcast-snooping allows fast two-hop cache-to-cache transfers, and eliminates the complexity of maintaining the directory. However, scaling snooping protocols has been difficult due to the overheads of broadcasting requests and looking up the cache tags of all the nodes for snooping.

However, a large number of snoop requests in snooping protocols are unnecessary as the majority of memory blocks are not shared by all the nodes. There have been several recent studies to filter such unnecessary snoop requests by tracking sharing states at region granularity. Using the sharing states, the filtering techniques remove unnecessary snoop requests at requesting sources [23, 9], at receiving destinations [24], or at intermediate routers during the transmission of the requests [2]. All the techniques use some on-chip tables to store the sharing states of the most recently accessed regions.

In this paper, we propose a new coherence filtering technique, called *subspace snooping*, based on the page table support of operating systems. To the best of our knowledge, this is the first study to use the page table to store potential sharer lists for snoop reduction. Subspace snooping filters snoops at requesting sources, reducing both snoop tag lookups at the destinations, and network traffic for transferring snoop requests. Unlike prior work based on the dichotomy of private and shared spaces [23, 9], in subspace snooping, a set of sharers defines a subspace for a page.

In the rest of this paper, a sharer of a memory region is a node which accesses the region at least once during a certain time period. A subspace is a *stable* subset of nodes sharing a region of memory consistently. These subspaces can be dynamically evolving, so long as they are stable for sufficiently long to be useful. While subspace snooping does not achieve the sharing-list precision of directory protocols, it has the potential to offer many of the benefits of snooping protocols with a reduced number of total snoops.

Subspace snooping uses an OS-based mechanism to maintain subspaces at page granularity. The set of sharers for a page (the subspace of a page) is recorded in the OS page

table entry, and translation look-aside buffers (TLBs) also keep the subspace information. For a coherence transaction, requests are delivered only to the nodes in the subspace. The subspace is guaranteed to be the superset of the current precise sharers which have a copy of the requested block in their caches(*superset subspace property*). The property allows subspace snooping to be used with any type of snoop-based coherence protocols and interconnection networks.

Subspace snooping does not add a significant hardware complexity to existing snoop-based systems and is easily adaptable to novel coherence techniques such as Token Coherence and In-network Coherence Ordering [19, 3]. Compared to prior work to filter unnecessary snoop traffic, the contributions of this paper are as follows.

- Unlike the region-based source filtering techniques [23, 9], subspace snooping does not simply divide the address space to private and shared regions. In real workloads, a significant number of coherence accesses occur on partially shared pages, i.e. pages shared by more than one node, but less than the total nodes. Forcing broadcast-snoops for those partially shared pages incurs numerous unnecessary coherence requests. Subspace snooping can filter unnecessary snoops for partially shared pages by tracking all possible sharers.

- Subspace snooping requires a relatively small amount of extra hardware. Only the sizes of page table in the memory and TLBs may modestly increase to keep subspace information. Some systems with 64-bit page table entries have unused bits in the entries. For a small scale multicore(under 16 or 32 cores), subspace snooping may not require any increase in the page table size. On the other hand, the effectiveness of region-based snoop filters using hardware tables can be sensitive to the table capacity and the working set of applications. Furthermore, checking and updating the tables frequently for coherence transactions also consume power.

- As a coherence filtering technique, which removes unnecessary requests at requesting sources and supports the superset subspace property, subspace snooping is not dependent on the implementations of underlying coherence policies and interconnection networks. For example, In-Network Coherence Filtering (INCF) which filters requests at routers, requires a deterministic routing policy with a packet-switched network [2]. However, subspace snooping does not impose such a restriction on network implementations.

In this paper, we apply subspace snooping to Token Coherence, which provides simple ordering support on un-ordered networks, without indirection through home nodes [19]. However, any type of snoop-based coherence will work with subspace snooping. For scientific and commercial workloads, subspace snooping reduces the total global snooping by 44% on average for a 16-core system, compared to the base TokenB protocol. Filtering snoop requests at requesting sources reduces network traffic and power consumption significantly. The global network traffic is reduced by 27% for 16 cores.

A set of sharers at page granularity may change during program execution. Applications may have phases with different sharing patterns and the operating system may migrate threads to different physical cores. Such sharer changes may decrease the effectiveness of subspace snooping, as subspaces may contain nodes, which no longer access the pages. To address such dynamic subspace changes, we investigate a *subspace shrinking* mechanism, which removes obsolete nodes from the subspaces. Simulation results show that for most of the applications in our benchmarks, shrinking is not necessary with only 4% more snoop reduction from the base subspace snooping. For our benchmark applications with relatively short execution times, shrinking does not provide enough benefit to justify the extra design complexity. However, the shrinking mechanism will be necessary to support long-running applications.

In the rest of this paper, we first discuss the prior work on snoop filtering and other related work in Section 2. In Section 3, we present the sharing characteristics of applications at coarse-grained page unit. In Section 4, we describe the subspace snooping architecture with a subspace shrinking mechanism. In Section 5, we discuss the costs of implementing subspace snooping, and the integration of subspace snooping to existing cache coherence protocols and networks. In Section 6, we present experimental results. Section 7 concludes this paper.

2. PRIOR WORK

2.1 Snoop Filtering Techniques

There are several studies to reduce unnecessary snoops in snoop-based coherence, and subspace snooping is based on the prior work on snooping filtering techniques. *RegionScout* and *Coarse-Grain Coherence Tracking (CGCT)* prevent snoop requests from broadcasting for private data at the requesting source. Subspace snooping also filters out unnecessary snoop requests at the source, and thus is able to reduce both network traffic and snoop tag lookups. *RegionScout* records the states (private or shared) of coarse-grained regions in per-node tables [23]. Subspace snooping differs from RegionScount in that subspace snooping does not simply divide coherence space into private or shared states.

CGCT (Coarse-Grain Coherence Tracking) maintains an additional coarse-grained coherence protocol in addition to the conventional coherence at the cacheline granularity [9]. CGCT also relies on the dichotomy of coherence space into private or shared. Compared to RegionScout, CGCT is able to further reduce the requests by not broadcasting requests for clean-shared data. However, even if a clean copy exists in other on-chip caches, the clean data must be supplied from the external memory. Therefore, bypassing cache-to-cache transfers for the clean-data can reduce broadcasts, but it may increase miss latencies in multi-cores with much faster on-chip cache-to-cache transfer latencies than external memory latencies. Subspace snooping selectively sends requests only to the stable sharers of the requested page, and thus does not miss the opportunities for fast cache-to-cache transfers for clean data. Another difference between CGCT and subspace snooping is that CGCT needs to look up the region tags and change their states for incoming snoop requests. However, subspace snooping updates subspaces very infrequently.

Compared both to RegionScount and CGCT, the base subspace snooping architecture does not require extra hardware tables except for the additional information embedded

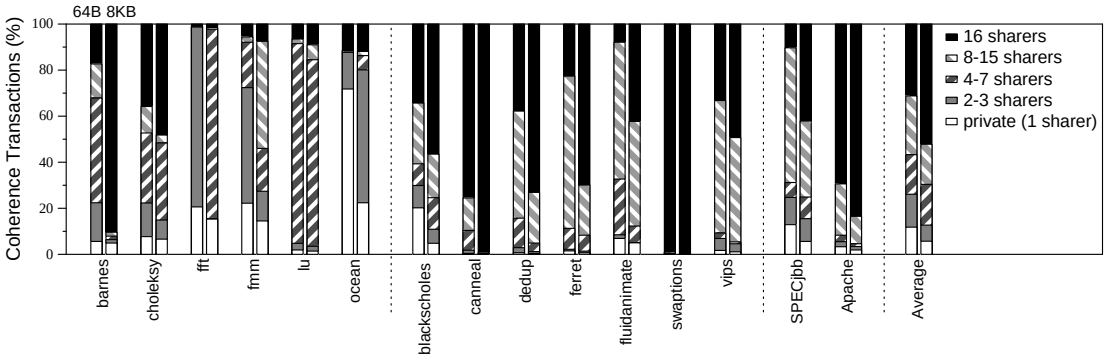

Figure 1: Sharing distributions of coherence transactions: 64B and 8KB granularity (16 cores)

in the TLB. We also explore a possible optimization for subspace snooping by allowing subspace shrinking, which uses counting bloom filters. However, the size of the per-node bloom filter is small compared to the region tables.

Unlike the aforementioned source-level filtering techniques, *In-Network Coherence Filtering (INCF)* filters out snoop requests at the routers [2]. Each router maintains a table for region-based sharing states, and does not forward a snoop request to a port, if the snoop request does not need to be delivered to the nodes reachable from the port. The mechanism requires a table for each router and the router pipeline must access the table. Furthermore, the routing algorithm must be deterministic and does not allow flexible adaptive routing. Unlike INCF, subspace snooping does not require any modification to underlying networks. *Flexible snooping* proposed by Strauss et al, provides adaptive forwarding and filtering of snoop requests, either for high performance or energy conservation [28].

Jetty filters out unnecessary snoops at the destination node, by maintaining a filter in each node [24]. Since the filter resides at the destination node, snoop requests must still be broadcast to all the nodes, consuming network bandwidth and power. It saves only the power consumption to look up snoop tags. *RegionTracker* proposes a mechanism to track cache states at both fine and coarse grain granularity with a two-level dual-grain tracking mechanism [31].

Ekman et al embedded sharing vectors in the TLB with virtually addressed caches [13]. The page-level sharing vector in the TLB is used to filter out unnecessary snoop requests. Subspace snooping keeps the subspace information in the TLB, but the original subspace information is stored in the page table. Also, subspace snooping does not track the sharing vector for every coherence transactions, which may require a constant update of the sharing vector in the TLB.

2.2 Other Related Work

Subspace snooping is extensively influenced by the prior work to improve coherence bandwidth, to support snooping on un-ordered networks, and to use OS support for managing on-chip caches.

Improving Coherence Bandwidth: Subspace snooping contains elements of both traditional snooping and directory protocols to improve coherence bandwidth. The bandwidth adaptive snooping has combined the benefits of snooping and directories adaptively in one system [21]. Multicast snooping [7] and destination set prediction [18] use prediction techniques which send coherence request only to potential sharers. Thus, these models must support a fallback mechanism to recover when a miss prediction occurs, whereas subspace snooping does not need it.

Coherence Ordering on Un-ordered Network: There are several recent studies to embed coherence capabilities into on-chip networks. Token coherence replaces the conventional cache states with tokens to remove synchronous updates of cache states [19, 26]. In-network cache coherence proposes the embedding of cache coherence protocols within the network [12]. The protocols send a request to the nodes in a virtual tree that consists of the sharers. Virtual tree coherence supports virtually ordered tree interconnects on unordered networks, and multicasts to the sharers tracked by region [14].

OS-based Cache Management: There are recent studies to use OS supports to manage on-chip cache placements. Fensch and Cintra proposed an OS-based approach for cache coherence in tiled CMPs by maintaining a single copy for each address [15]. Reactive-NUCA classifies data into private data, instruction, and shared data. Blocks are placed to a location that can improve performance by the classification [16]. In this approach, they record classification information in page tables.

3. FINDING SUBSPACES

Subspace snooping uses stable subspaces found in the sharing behaviors of parallel applications. In this section, we present the characteristics of sharing patterns in our benchmark applications. We first present the distributions of sharers at page granularity, and show that the majority of page accesses occur for partially shared pages. Secondly, we present the sharing pattern changes during the execution of applications. We use the Simics full-system simulator [17] with a cache model in the GEMS toolset [20]. The target system has 16 cores with a 512KB private L2 cache per core. We use parallel applications from SPLASH-2 [30] and PARSEC [6]. We also evaluate two server applications, SPECjbb and Apache. The details of the benchmark applications and system configuration are shown in Table 2 and Table 3.

Subspaces may contain more cores than the precise sharers for two reasons: i) *spatial* and ii) *temporal* aliasing. Spatial aliasing occurs, since subspaces are maintained at page granularity. The chance of false sharing with page granularity is higher than that with a cacheline unit. Secondly, temporal aliasing occurs, since subspace snooping does not update sharers for each coherence transaction as directory

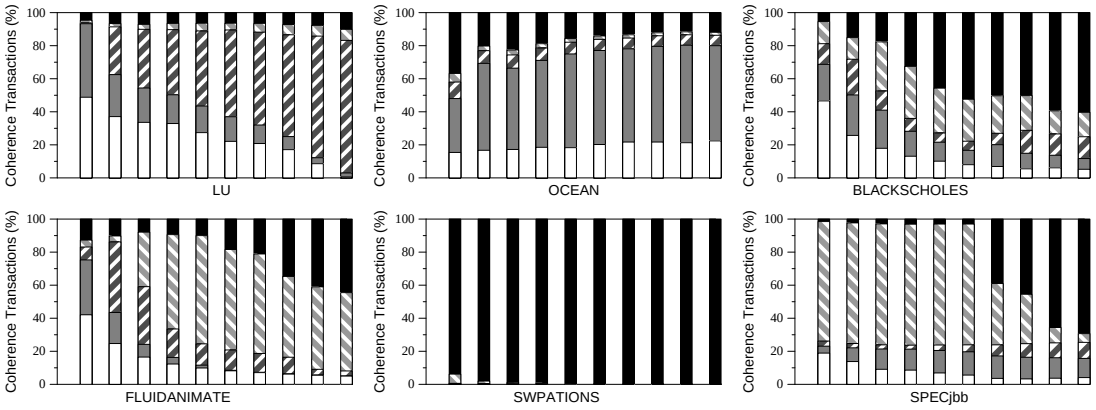

Figure 2: Cumulative sharing distributions for different time intervals (8KB granularity)

protocols do. Therefore, subspaces may have sharing cores which have accessed a page a while ago, but no longer have any cachelines of the page in their caches.

3.1 Sharing Behaviors at Page Granularity

Subspace snooping tracks sharers at page granularity and sends snoop requests only to the sharers marked in the page table. To reduce snoop requests, the number of sharing processors for each page must be small. Figure 1 presents sharing distributions for all coherence transactions, which include read, read-exclusive, and upgrade (from shared to modified) transactions in MOESI protocols. For each page, we track sharers cumulatively from the beginning to the end of simulation. In the figure, the number of sharers for a coherence transaction is the number of sharers collected for the accessed page until the transaction occurs. Sharers tracked in each page are never removed during execution. For each application, the figure shows two distributions, the first one with 64B granularity and the second one with 8KB granularity. The distributions of sharers are divided into 1 (private), 2-3, 4-7, 8-15, and 16 processor partitions.

As shown in the Figure 1, applications have a significant potential to reduce snoops, even at page granularity. On average, 5.7% of coherence requests are for private pages, 7.0% for 2-3 sharers, 17.8% for 4-7 sharers, 17.3% for 8-15 sharers, and 52.2% for 16 sharers. `Canneal` and `swaptions` in PARSEC show the worst sharing behaviors, with almost the entire coherence transactions sent to fully shared pages. A commercial workload, `SPECjbb` and `Apache` access fully shared pages at higher rates than the average rate.

The figure shows that, for the snoop requests from the L2 caches, 42% of coherence requests occur on partially shared pages with 2-15 sharers. This result indicates dividing the address space only to private and shared spaces, does not provide an effective snoop reduction. With such dichotomy, 94% of accesses occur on shared pages, and snoop requests must be broadcast.

Tracking sharers at page granularity causes spatial aliasing for some applications. The coherence transactions on fully shared pages in `barnes` increase from 17.3% (block granularity) to 90.4% (page granularity). `Ferret` also shows a significant increase of fully shared pages from 64B to 8KB granularity (from 7.6% to 59.5%). The rest of the applications show a modest increase of sharers with page granularity. Compiler or programming optimization to reduce false sharing at page granularity may be able to reduce the spatial aliasing, but the techniques are beyond the scope of this paper.

3.2 Sharing Behaviors at Different Intervals

Sharing distributions of applications may change over the program execution. Figure 2 shows cumulative sharing distributions at different time-intervals. We show six representative patterns in the figure. The total execution time of each application is divided into ten equal time periods. The graphs show sharing distributions, similar to Figure 1, during different time-intervals. In this figure, each bar shows the distributions of accesses to 1, 2-3, 4-7, 8-15, and 16 sharers (subspaces) for a time-interval. Sharers are not reset for each time interval. Instead, sharers are accumulated for each page cumulatively from the beginning of each application run.

For `lu`, fully shared page ratio does not increase as the application progresses, but partially shared page ratios (8-15 sharers) monotonically increase. In the later part of the program execution, high sharing (8-15 sharers) accesses dominate coherence transactions. `Ocean` shows a relatively stable pattern. `Swaptions` shows the ratio of accesses to fully shared pages is uniformly high from the beginning period. `SPECjbb` shows a stable access pattern with a large portion of 8-15 sharers till the sixth period, but fully shared accesses dominate after then.

There are some applications with stable patterns, either with a high ratio of fully sharing patterns (`swaptions`) or with a significant ratio of partially shared patterns (`ocean`, `fluidanimate`). However, other applications tend to increase the accesses to high sharing pages in the later part of program execution. It occurs for two reasons: Firstly, subspaces become larger for frequently accessed pages as more sharers are added over the runtime, due both to the sharing behavior and thread migration. Secondly, the program accesses highly shared pages more often in the later part of the execution than the early part. For the first reason, it may be necessary to adjust subspaces as the active sharers change during runtime. In Section 4.3, we will propose a shrinking mechanism to remove inactive sharers from subspaces.

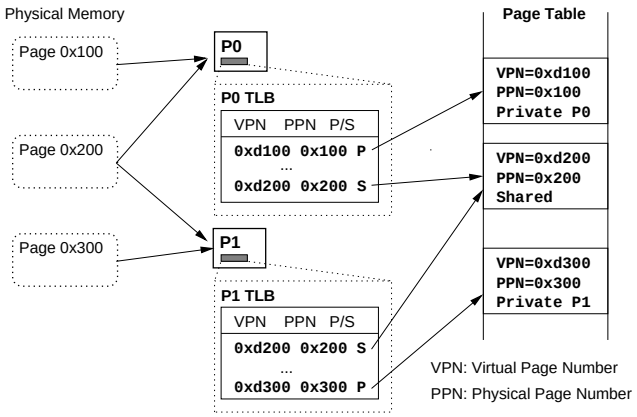

Figure 3: Bi-space snooping using a page table and TLBs

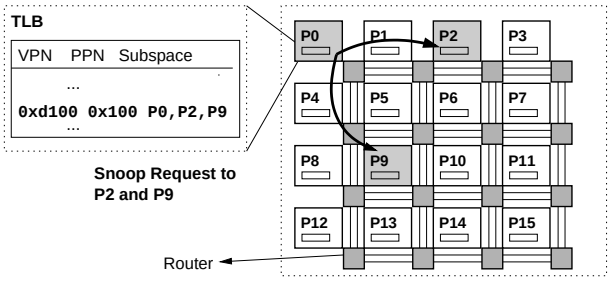

Figure 4: Subspace snooping overview: sending a snoop request to a subspace

4. SUBSPACE SNOOPING ARCHITECTURE

Subspace snooping maintains potential sharer lists in page table entries and TLBs. To describe the sharer tracking mechanism of subspace snooping, we first present bi-space snooping, which divides subspaces only to two spaces, private and shared. Bi-space snooping is similar to the prior coarse-grained source-level filtering, except for the use of OS page tables. In Section 4.2, we introduce a general subspace snooping protocol to track all the possible combinations of sharers.

As a program runs, nodes which have accessed a page, can be accumulated to the subspace of the page, and some nodes in the subspace may no longer use the page. In Section 4.3, to avoid the monotonic accumulation of sharers on each page, we propose an adaptive mechanism to shrink subspaces safely.

4.1 Bi-space Snooping Architecture

As a simplified subspace snooping, we introduce *bi-space snooping* that records the private or shared states of pages in page table entries. Before processors send coherence requests, the processors must look up the TLBs for address translation. With bi-space snooping, the processors can find whether pages are shared or private during address translation. For private pages, snoop requests do not need to be sent to the other processors. In conventional snooping protocols, if a cache miss occurs, the requesting processor does not know whether the cacheline is shared or private until all the other processors are snooped. Therefore, even if a page is private (no other processors have ever accessed the page), a processor must broadcast requests and all the other processors must snoop to verify whether the cacheline is in their caches.

Using page tables and address translation mechanisms, bi-space snooping classifies pages into private and shared pages to choose a different space. If a page has been accessed by only one processor, the page is set as a private page for the processor. If another processor attempts to access the page, the page state is updated to a shared state. For private pages, bi-space snooping does not broadcast requests, reducing unnecessary snoops.

Figure 3 shows the overview of a bi-space snooping protocol using a page table. To classify pages, bi-space snooping adds an extra bit in page table entries for representing a private or shared state. TLB entries must also have the extra state bit. In addition to the state bit, a page table entry maintains an owner identifier for private pages, and the owner identifier of a page is set when a processor accesses the page table entry for the first time to handle a TLB miss. Bi-space snooping updates the page sharing state in a page table entry when a TLB miss handler accesses the page table entry to fill TLBs. If a page is a private page already owned by another processor, the sharing state of the page should be changed to a shared state.

When a page state changes from private to shared, a critical constraint for correctness is that the original owner must also be notified about the change, before a new sharer accesses the memory location. To do that, the TLB miss handler must send a TLB update request to the current owner to update the status of its TLB entry to shared. To avoid any race condition, the TLB miss handler must not complete the TLB fill until the TLB entry of the current owner is completely updated. To ensure the correctness, the TLB miss handling of the newly joining core is delayed until the acknowledgment from the current owner is received.

4.2 Fine-grained Subspace Snooping

Subspace snooping extends bi-space snooping to support fine-grained subspaces for each page. Figure 4 presents the overview of subspace snooping. The sharing states in the TLBs and page table are extended to sharing vectors to record the sharers of each page. For each coherence transaction, a requesting processor finds a set of sharers (subspace) of the address from the TLBs. Snoop requests are delivered only to the processors in the subspace. In Figure 4, the subspace of page 0x100 is {P0, P2, P9}, and P0 sends snoop requests only to P2 and P9.

To maintain the list of sharers for each page, page table entries and TLBs are extended to hold a bitmap, *sharing vector* for all the processors, with each bit corresponding to each processor. For every coherence transaction, the requesting processor looks up the TLBs during address translation, and sends requests to only the processors marked in the sharing vector.

Subspace snooping uses a similar mechanism to bi-space snooping to keep the TLBs up-to-date with the latest subspace information. Whenever a new processor is added to a subspace, the sharing vector of the page table entry and the TLB entries of processors in the current subspace must be updated. To avoid any race condition, the update must be completed, before the new processor is permitted to access the page. In bi-space snooping, sharing state updates happen only at transitions from private to shared. However, in

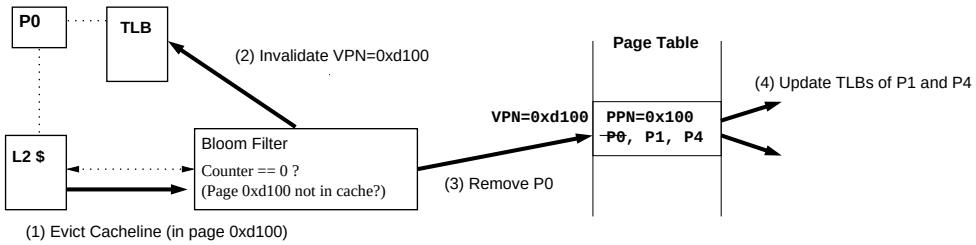

Figure 5: Shrinking subspaces: removing P0 for page 0xd100

subspace snooping, a subspace is updated whenever a new processor is added to the subspace.

Generalized subspace snooping may incur more storage and timing overheads than bi-space snooping. Firstly, subspace snooping must update the subspace of a page more frequently than bi-space snooping, since adding each new sharer must update the subspace. In the worst case, one page table entry can be updated N times when N is the number of processors. Such updates delay the TLB miss handling of a newly joining sharer. Secondly, with subspace snooping, the page table size may be increased in the main memory, and the TLBs require more space for sharing vectors than that with bi-space snooping. We will discuss the costs of supporting subspace snooping in Section 5.1.

4.3 Shrinking Subspaces on Phase Changes

In the base subspace snooping, sharers are added to subspaces during the run-time of a program, but never removed from subspaces. However, as a program runs, the sharing patterns may change, and the base cumulative subspaces may include obsolete sharers, which no longer access the pages. To adapt to sharing phase changes, subspace snooping supports a mechanism to remove sharers from subspaces, called *subspace shrinking*.

Shrinking a subspace can be done in the background. To shrink a subspace, a processor, which attempts to remove itself from the subspace, updates the sharing vector in the page table entry, and sends subspace shrink requests to the current sharers to update their TLBs. However, updating the TLBs of other processors can be delayed, without causing any correctness problem. As long as the sharing vectors cached in the TLBs are the supersets of the subspace in the page table, subspace snooping works correctly.

In this section, we propose a subspace shrinking mechanism, which can gradually re-adjust subspaces, if processors stop accessing certain pages. The shrinking mechanism tracks the number of blocks of a page residing in the caches of each processor. When a page is completely evicted from a processor, the processor is removed from the subspace of the page.

When subspaces are shrunken, one critical invariant must be maintained for correctness:

- *Subspace Invariant:* If a processor contains a cacheline in its caches or a page in its TLBs, the processor must be in the subspace of the page including the cacheline or the TLB entry.

To trigger a subspace shrink, the mechanism must know when the last cacheline of a page is evicted from the local caches. When no cacheline of a page exists in the caches, a processor can remove itself safely without violating the subspace invariant. The corresponding TLB entry must also be invalidated. Supporting the mechanism requires a fast page-level cache residence test function to check whether a page is cached or not. For the cache residence test, we use a counting bloom filter technique [8]. Each processor has a bloom filter indexed by a hash of a physical page number (PPN). The bloom filter entry has a counter and a virtual page number (VPN). Whenever a cacheline is inserted to the local caches, the counter in the corresponding entry is increased by one. When a cacheline is evicted or invalidated, the counter is decreased by one. A problem with the bloom filter is the saturation of counters, which should be a rare event. If the counter of an entry is saturated, we assume that all PPNs mapped to the entry have cachelines in the local caches. Infrequently, the OS can reset the bloom filter, when the entire local caches are flushed. If an aliasing, with two pages mapped to the same entry, occurs, only one of the pages can be shrunken, due to one VPN per entry.

Figure 5 describes the shrinking processes. A bloom filter is used to trigger shrink events. When a counter is decreased and becomes zero, it is guaranteed that pages mapped to the entry no longer exist in the caches. Using the stored VPN, the local TLB is invalidated and the page table is updated.

Shrinking subspaces too early can reduce performance by increasing the chance of subspace updates. For example, soon after a processor removed itself from the subspace of a page, the processor can access the same page again, adding itself to the subspace. In Section 5.1, we will present how many TLB misses cause subspace updates, and the possible increase of subspace updates by shrinking subspaces.

5. IMPLEMENTATION ISSUES

In this section, we discuss implementation issues for supporting subspace snooping. Firstly, we present how to maintain subspaces in page tables correctly, and how often subspace updates occur in our benchmark applications to show their performance impact. Secondly, we discuss how subspace snooping can be integrated to coherence mechanisms and networks. Finally, we discuss several other implementation issues, including DMA, HW prefetching, and page aliasing.

5.1 Maintaining Subspaces

If a TLB miss occurs and the processor is not in the subspace of the missing page, the processor must be added to the subspace before accessing the page. The TLB miss handling is not completed until the subspace change is updated to the page table and the TLBs of the processors in the current subspace. Hardavellas et al. also stores private or shared status of pages in page tables for cache placements [16]. In their work, once the second core accesses a

private page, the status in the page table entry changes to shared. The operation is similar to that of bi-space snooping.

5.1.1 Available space in page table entries

To record subspaces in page tables, a page table entry must have enough free space. As discussed in Awasthi et al [4], UltraSPARC-III has 64-bit address space, but uses only 44 bits for virtual addresses and 41 bits for physical addresses. The remaining 23 bits are unused and those bits can be used for storing subspace information. If the number of nodes exceeds 23, a possible solution is to cluster the nodes. 32-processor machine can use 16 bits, with clustering two processors to a cluster. Sharing vectors are set for cluster units, not for individual processors. If a requesting processor sends snoop requests to a cluster, all processors in the cluster must snoop the request.

5.1.2 Making TLBs consistent for subspace changes

In conventional shared memory multiprocessors, a page table entry is updated when the virtual-to-physical mapping is changed or the permission status is changed. In subspace snooping, changing subspaces also requires updating the page table. Updating a page table entry requires a mechanism to keep TLBs consistent [29]. Once the page table entry is updated, the existing copies in the other TLBs must be invalidated or updated. There are two common ways to keep TLBs consistent. A simple and generic way is using inter-processor interrupts (IPI) to force other processors to execute local TLB invalidation instructions. A more light-weight mechanism is to broadcast TLB invalidation messages. A TLB invalidation instruction not only invalidates the local TLB, but sends TLB invalidation signals to the other processors. Sun XD-Bus [27], PowerPC [1] and Intel's Itanium processors [5] support TLB coherence with such a remote TLB invalidation mechanism, which is much less costly than IPI.

Invoking IPIs is costly for subspace updates, as subspace updates may occur more frequently than page table updates occur in the conventional multiprocessors. To reduce the overheads of maintaining TLB consistency for subspace changes, subspace snooping uses a similar mechanism to the remote TLB invalidation mechanism. When a subspace is updated for a page, the update requests for the TLBs are multi-cast to the processors in the current subspace. The receiving processors must send acknowledgments to the requesting processor.

5.1.3 Updating page table entries

For a normal TLB miss, the page table entry is fetched either from caches or the external memory. If a subspace addition is necessary, an atomic read-modify-write is performed to the entry which is already in the local cache by the TLB fill. For SW-based TLB fills, the TLB fill handler will update the page table entry. For HW-based TLB fills, the TLB fill controller must be augmented for the support.

Unlike the page table updates to change address mapping or permission bits, adding a processor to a page table entry does not result in a significant complexity for avoiding a race condition. Each processor changes only its corresponding bit in the sharing vector, but it never modifies the other bits in the page table entry. However, it needs an atomic read-modify-write operation to set the bit safely without data races. Furthermore, adding a processor to a subspace is a relatively safe operation which does not cause any conflict with other operations on the page table entries. For example, adding a wrong processor to a subspace will only reduce filtering performance, since an extra unnecessary snooping message will be sent to the processor.

After the page table entry is updated, the requesting processor broadcasts subspace update messages to all the processors in the current subspace. The requesting processor can access the page only after it receives the acknowledgments from all the other processors. Compared to a normal TLB fill process, a subspace add operation requires two extra steps: *i)* firstly, it needs the execution of an atomic read-modify-write to set the corresponding bit in the page table entry. The operation may require a shared-to-modified state change for the cache block of the page table entry. The cacheline for the entry will be most likely in the local caches, since the TLB fill operation brings it into the caches. *ii)* The second step is to send TLB update messages to the other processors in the subspace, and wait for the acknowledgments. The latencies of each step are close to the latencies of broadcasting cacheline invalidation requests and receiving the acknowledgments.

Shrinking subspace can increase subspace updates, as it can remove a processor prematurely from a subspace, incurring subspace additions later. Unlike adding a new processor to a subspace, removing a processor from a subspace during shrinking is not on the critical path of TLB or cache miss handling. Shrinking can be processed on the background, as the delayed shrinking only increases unnecessary snoops.

	base	shrinking	
Workloads	add	add	remove
barnes	0.002	0.002	0.001
cholesky	0.006	0.007	0.003
fft	0.039	0.053	0.031
fmm	0.002	0.002	0.001
lu	0.002	0.002	0.001
ocean	0.004	0.012	0.008
blackscholes	0.007	0.007	0.000
canneal	0.053	0.174	0.127
dedup	0.014	0.016	0.002
ferret	0.004	0.005	0.003
fluidanimate	0.010	0.012	0.003
swpations	0.001	0.001	0.000
vips	0.004	0.009	0.007
SPECjbb	0.091	0.113	0.040
apache	0.103	0.189	0.097

Table 1: Subspace updates per 1000 instructions

5.1.4 Subspace update rates in applications

Table 1 presents the number of subspace updates per thousand instructions. The second column shows the number of subspace updates to add a new processor with the base subspace protocol. The base subspace snooping has a minor number of subspace updates. `FFT` has 0.039 updates per 1k instructions, and `canneal`, `SPECjbb`, and `apache` have 0.053, 0.091, 0.103 updates respectively. The rest of the applications have less than 0.01 updates per 1k instructions. The result shows that subspace updates are very infrequent. The third and fourth columns of Table 1 show the number of updates for adding and removing a processor with the shrinking support. An observation is that with shrinking, compared

Parameter	Value
Processors	16/32 in-order SPARC core
L1 I/D cache	32KB, 4-way, 64B block, 2 cycle access latency
L2 cache	512KB, 8-way, 64B block, 12 cycle access latency
I-TLB	L1 TLB: 16-entry, fully assoc. L2 TLB: 128-entry, 2-way
D-TLB	L1 TLB: 16-entry, fully assoc. L2 TLB: 2 x 512-entry, 2-way
Main memory	4GB memory, 8KB pages
Bloom filter size	1024 entries, 7 bit counter
Coherence	Token Coherence
On-chip Network	4x4, 4x8 2D mesh with 16B links, 4 cycle router pipeline

Table 2: Simulated system configurations

SPLASH-2	Dataset	PARSEC	Dataset	Servers	Dataset
barnes	65,536 particles	blackscholes	16,384 options	SPECjbb2k	16 or 32 warehouses
cholesky	tk29.O	canneal	200,000 elements	Apache	160 simulated users
fft	4,194,304 points	dedup	31MB data		
fmm	65,536 particles	ferret	64 queries, 13,787 images		
lu	1024 x 1024 matrix	fluidanimate	5 frames, 100,000 particles		
ocean	514 x 514 grid	swaptions	32 swpations, 10,000 simulations		
		vips	1 image, 2,336 x 2,336 pixels		

Table 3: Application input data and parameters

to the base subspace, the subspace add rates are increased almost by the same amount as the subspace shrink rates. It means most of the shrunken pages are accessed again later by the removed processors. The benchmark applications do not have many coherence requests on the pages with obsolete sharers in the subspace, making the shrinking mechanism ineffective.

5.2 Integration to Coherence Protocols

Subspace snooping can be integrated to any type of snoop-based coherence mechanisms, independent from underlying network topologies and from coherence mechanisms. Subspace snooping always maintains the superset of precise sharers in a subspace. This superset property allows subspace snooping to be used with any snoop-based coherence protocols without restriction.

Traditional snoop-based multiprocessors use a dedicated request bus to broadcast snoop requests. The broadcast bus can use various implementations, including physical buses, fat trees, or rings [10, 22]. When a dedicated broadcast bus is used for snoop requests, subspace snooping must send requests to all nodes through the bus anyway, with sharer vectors embedded in the requests. With this kind of fixed broadcast buses, subspace snooping reduces snoop tag lookups, although the network traffic cannot be reduced. However, subspace snooping can reduce the network traffic, if the broadcast network also supports multi-cast transmissions, as proposed by Bilir et al [7].

Subspace snooping can be applied to snoop-based protocols using packet-switched networks. In protocols like AMD HyperTransport [11], a snoop request is sent to the home node as an ordering point, and the home node broadcasts snoop requests. With subspace snooping, the home node, instead of broadcasting, can send requests only to the processors in the subspace. Other designs provide snoop ordering without the indirection through home nodes. In-Network Snoop Ordering provides snooping ordering on unordered networks by maintaining logical orders among coherence messages at routers [12]. Token Coherence eliminates the need for ordering by using tokens and persistent requests [19].

5.3 Other Implementation Issues

Subspace snooping relies on address translation to find subspaces. In this section, we discuss uncommon cases, bypassing of address translation and aliasing.

Handling DMAs: Direct Memory Access (DMA) by IO devices uses physical addresses without address translation by TLBs. Therefore, the IO devices cannot know the subspace of a page. However, many IO DMAs occur only on uncacheable memory areas, which is not made cache-coherent by hardware anyway. For the rest of DMA accesses on coherent addresses, DMAs can access memory as if all pages are fully shared. Delivering extra unnecessary snoop requests does not break the correctness of subspace snooping, and the DMA accesses are not frequent compared to other coherence transactions.

HW prefetchers: Hardware prefetchers are often trained by physical address streams, and they issue prefetch requests in physical addresses without address translation. Although it is always possible to send prefetch requests as if they are for fully shared pages, it can decrease the effectiveness of subspace snooping, since prefetch requests are much more frequent than DMA requests. There are two solutions to this issue: 1) use virtual address-based prefetchers with address translation, 2) add a map from a physical page number (PPN) to the sharing states for prefetchers.

Page sharing across address spaces (page aliasing): If the virtual pages of different address spaces are mapped to the same physical page, subspace snooping may not track the sharers correctly. A solution for correctness is to simply force broadcast snoops for the aliased pages, overriding subspace snooping. As the OS is always aware of such sharing and able to mark it on the page tables, subspace snooping does not cause any correctness problem.

6. EXPERIMENTAL RESULTS

To evaluate subspace snooping, we use Virtutech Simics [17], a full-system simulator, with a timing model from the GEMS toolset [20]. The GEMS memory model is augmented with the GARNET interconnection model [25]. Our parallel workloads consist of applications from SPLASH-

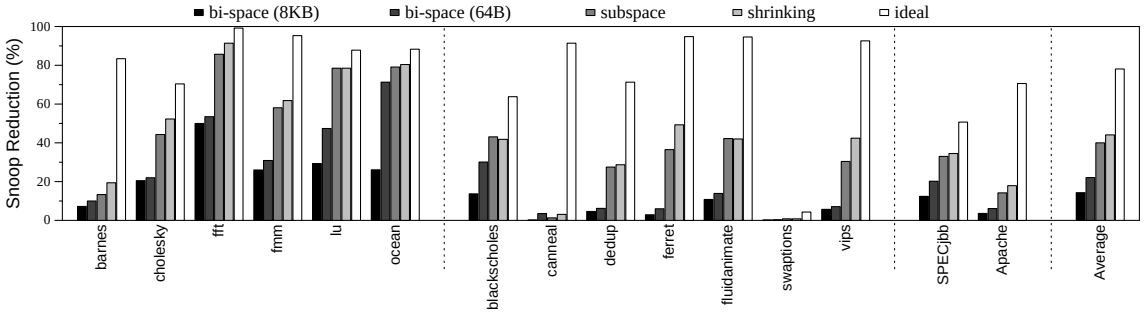

Figure 6: Snoop reduction with Token Coherence: 16 cores

2 [30] and PARSEC [6]. We also evaluate two commercial workloads, SPECjbb2000 and Apache. Apache is a web server workload with the Apache HTTP server, serving only static web pages. The details of workload parameters are shown in Table 3.

The simulated system is a 16-core CMP and each core is an in-order SPARC processor with a private L2. Table 2 shows the system configurations of the simulated system. We model 4x4 2D-mesh interconnection networks, with dimension ordered routing. We integrate subspace snooping to Token Coherence (TokenB), including the impact of supporting TLB coherence for subspace changes. Token coherence uses persistent requests as a fallback mechanism, if normal transactions exceed the time-out limit. We do not apply subspace snooping to the persistent requests to simplify our integration with TokenB.

6.1 Snoop Reduction by Subspace Snooping

In this section, we present the snoop reduction by various subspace snooping schemes. In a snoop-based coherence protocol, every coherence transactions generate snoop requests to all the nodes, and each node must look up their cache tags and send a response. In this section, we measure how many snoops occurring in each node can be reduced by subspace snooping. The dynamic power consumption of snoop tag lookups is proportional to the snoop rate. It has been shown that the power consumption of snoop tag lookups amounts to a significant portion of L2 dynamic power consumption, and as the number of cores increases, the power consumption by snooping will increase [24]. Subspace snooping reduces the snoop tag lookups, allowing the design of power-efficient many-cores.

Figure 6 shows the snoop reduction rates of four configurations: bi-space (8KB), bi-space (64B), subspace snooping without shrinking, and subspace snooping with shrinking. Bi-space (64B) is presented to show the result of ideal filtering using only two states (private and shared), without the negative effect of spatial aliasing. The rest of results use 8KB page granularity. The figure shows the reduction of total number of snoops occurring at all the cores, compared to the baseline TokenB protocol, which always broadcasts snoop requests. The last bar of each application shows the snoop reduction with an ideal protocol. In the ideal protocol, a requesting core always knows exactly which cores have a copy of the cache block requested for the coherence transaction, which is similar to a directory-based protocol. The requesting core can send requests only to the precise sharers.

Bi-space snooping has modest snoop reductions in several applications. `FFT` has the highest reduction with bi-space

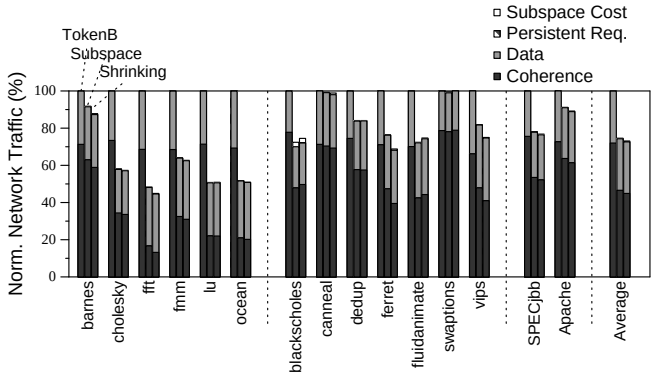

Figure 7: Normalized network traffic: 16 cores

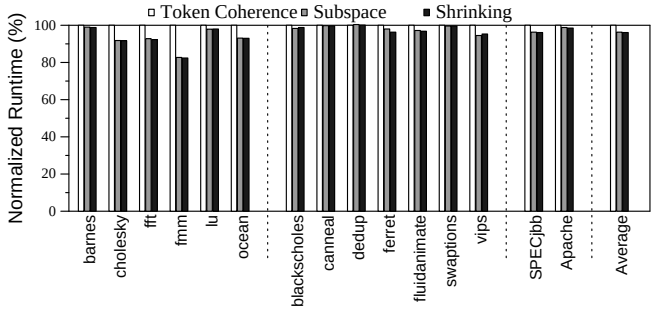

Figure 8: Normalized execution times: 16 cores

(50%). `Cholesky`, `fmm`, `lu`, and `ocean` have 21-29% snoop reductions by removing snoop requests for private pages. However, the rest of the applications have small reduction rates of up to 15% with bi-space snooping. Although the memory overhead of bi-space snooping is very small, its benefit is modest or minor, and fine-grained subspaces are necessary to further reduce snoops. Bi-space (64B), with average 22% reduction on average, is slightly better than bi-space (8KB) without spatial aliasing. However, the effect of spatial aliasing is not significant for bi-space snooping.

Using fine-grained subspaces, which can discern each sharing node, improves bi-space significantly. On average, the base subspace snooping without shrinking reduces snoops by 40%, reducing 25% more snoops compared to the 14% reduction of bi-space. The snoop reduction rates for scientific workloads are generally higher than those of commercial workloads, except for `canneal` and `swaptions`. `FFT`, `lu`, and `ocean` are close to the ideal since the sharing patterns of these workloads have most of the coherence transactions on 1-7 sharer pages.

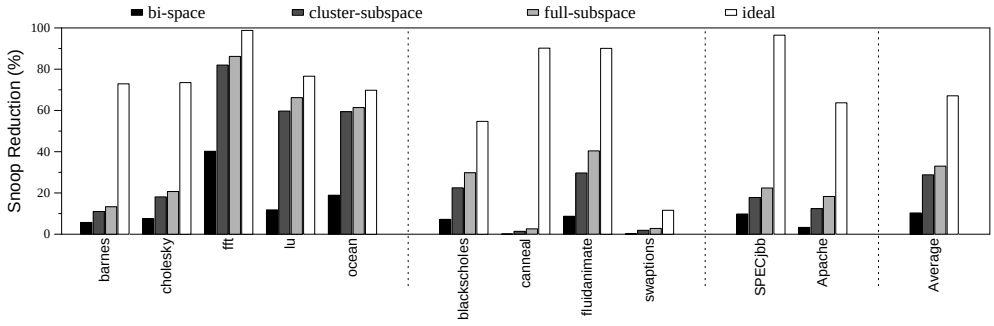

Figure 9: Snoop reduction with Token Coherence: 32 cores

Workloads	not pinned	pinned
blackscholes	43.1%	40.52%
canneal	1.3%	1.5%
fluidanimate	42.2%	82.0%
swpations	0.8%	0.6

Table 4: Snoop reduction by pinning threads to physical cores

The snoop reduction rates with the PARSEC applications are lower than those with the SPLASH-2 applications. Two applications with the worst reduction rates are `canneal` and `swaptions`. `Canneal` has very fine-grained sharing patterns, where threads access small data items in a random pattern. In such a fine-grained random sharing pattern, subspace snooping cannot form a stable subspace with small numbers of sharers. However, the ideal protocol can reduce the snoops by 92% for `canneal`. `Canneal` shows the inability of subspace snooping to track precise sharers, if many processors access shared pages frequently, but only a small subset of the processors have copies of the pages in their caches.

`Swaptions` also shows that most of the coherence transactions occur on the pages shared by all the processors, as shown in Figure 1. However, unlike `canneal`, even the ideal protocol cannot reduce snoops effectively. `Swaptions` has little communication among threads, but the majority of misses occur for instruction fetching, as the instruction working set does not fit in the private L2. Since the memory pages for instructions are shared by all the processors, when a cache miss occurs for a core, most likely other cores have the cache-line in their caches. However, for such read-only data, the coherence protocol can be optimized not to broadcast requests. Such optimization will be our future work.

Another factor to reduce the effectiveness of subspace snooping is unnecessary thread migrations. In Figure 6, threads are not bound to physical cores for PARSEC. Among the PARSEC applications, we picked four applications with static thread allocation, and pinned threads to cores. The performance actually improves slightly by binding threads to cores. For the four applications, three applications, `blackscholes`, `canneal`, and `swaptions`, do not improve snoop reductions. However, `fluidanimate` improves the reduction rate from 42% to 82% by pinning threads to cores. The results show that if the operating system scheduler limits thread migrations, the effectiveness of subspace snooping can be improved significantly for certain applications.

The fourth bar of Figure 6 shows the snoop reduction rates by the subspace shrinking mechanism. The results show the shrinking mechanism can increase the reduction rate modestly by 4% compared to the baseline subspace snooping. The benefit of shrinking is small in the benchmarks we used. As discussed in Section 5.1.4, in the benchmark applications, memory accesses to the pages with obsolete sharers do not occur frequently enough to make shrinking effective. One of the reasons for such small benefit of shrinking is the relatively short execution cycles of the benchmark applications. For long-running applications, we believe the shrinking mechanism will be necessary to mitigate the effect of sharing pattern changes and thread migrations.

6.2 Network Traffic and Performance

In this section, we present the reduction of network traffic by filtering snoop requests on a 4x4 mesh network. We also show the performance improvement by the traffic reduction.

Network traffic: By reducing snoop requests at requesting cores, subspace snooping reduces the network traffic in token coherence. Figure 7 shows the network traffic with subspace snooping normalized to that with the base token coherence. The three bars for each application show the network traffic with TokenB, subspace, and shrinking, respectively. For each bar, the traffic is divided into coherence, data, persistent and subspace update requests. For TokenB, 72% of the total traffic is for coherence requests and acknowelegements, and 28% is for data traffic. The traffic by persistent requests is negligible. Subspace snooping reduces only the coherence traffic.

With subspace snooping, `FFT`, `lu`, and `ocean` have about 50% less network traffic than TokenB. In `canneal` and `swaptions`, network traffic does not decrease, as requests are mostly to fully shared pages. The average network traffic is reduced by 25% without shrinking. Subspace shrinking reduces traffic only modestly. Except for `blackscholes`, most of workloads have negligible traffic overheads for subspace update messages, since subspaces are updated infrequently. However, even in `blackscholes`, the subspace update messages increase the traffic by 2%.

The reduction of network traffic leads to both the power reduction and performance improvement. The power consumption of routers and links contributes to a significant portion of the total chip power. Subspace snooping reduces not only the network power, but also the power consumption of tag lookups for snoops. Combining the two power reduction effects leads to a significant on-chip power saving for future many-cores.

Performance improvement: Figure 8 shows the normalized execution times of three configurations. For `cholesky`, `FFT`, and `ocean`, subspace reduces the execution times by 7-8%. The average execution time is reduced by 4%. The per-

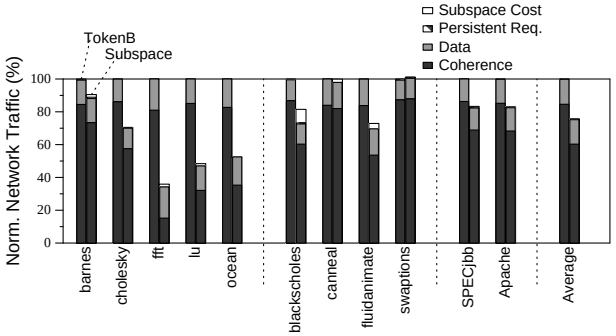

Figure 10: Normalized network traffic: 32 cores

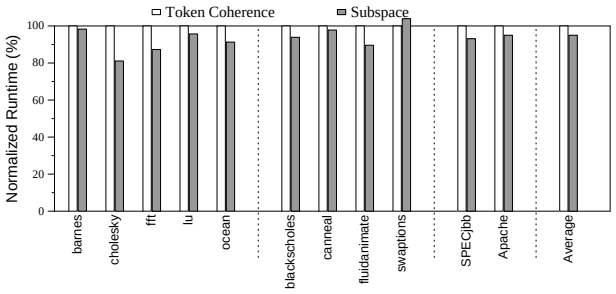

Figure 11: Normalized execution times: 32 cores

formance improvement is modest as the benchmark applications do not consume the network bandwidth intensively, and the network provides enough bandwidth even for TokenB. However, in this paper, we did not explore the performance improvement by other benefits of subspace snooping. For example, the reduced power consumption in the networks and snoop tag lookups allows the power budget to be used for a better performance. The processor clock speed can be increased using the saved power budget, or power-limited performance features can be used more aggressively using the budget. The traffic reduction can also allow narrow network links. The saved area with the narrow links can be reused to increase cache or other prediction table sizes.

6.3 Evaluation for 32 Cores

Figure 9 presents the snoop reduction rates with 32 cores. The results include a configuration, called cluster-subspace. The cluster-subspace bar shows the snoop reduction when only 16 bits are used in a page table entry for 32 cores. Adjacent odd and even number cores are clustered. Due to the limitation of simulation platform, 32-core results have a less number of applications compared to 16-core results. The average snoop reduction rate is similar to the 16 core results. On average, 34% of snoops are reduced by subspace snooping without shrinking. The reduced effectiveness of subspace snooping with 32 cores is partly due to the increased snoops by the kernel traffic. Clustering two cores does not significantly reduce the effectiveness of subspace snooping.

Figure 10 shows the network traffic for 32 cores. The first bar shows TokenB, and the second bar shows subspace without shrinking. Compared to the 16 core results, the portion of coherence traffic increases to 85% from 72% in the 16 core configuration. Figure 11 presents the normalized execution times for TokenB and subspace without shrinking. Even though snoop reduction rates in 32 cores are slightly less than those in 16 cores, the performance impact is higher in 32 cores, since the coherence costs become more expensive in 32 cores than in 16 cores.

7. CONCLUSIONS

In this paper, we proposed a novel coherence filtering technique called subspace snooping. Subspace snooping maintains the stable sharers (subspace) of a memory location in the page table entry, and snoop requests are issued only to the cores in the subspace. The proposed OS-based approach is not dependent upon a particular type of network topologies or coherence mechanisms, and is not limited by the hardware storage to trace the sharing states of commonly accessed pages. By integrating to Token Coherence on unordered networks, subspace snooping reduces snoops by 44% and network traffic by 27%.

The execution cycle improvement by subspace snooping was modest (4%), as the benchmark applications do not stress the networks intensively in our configurations. The performance gain would increase, as the bandwidth requirement increases with more demanding applications or different system configurations (for example with less private L2 caches). We expect the snoop and network traffic reduction to allow increased clock speeds and better performance features by exploiting the saved power and area budgets.

Shrinking subspaces dynamically leads to only a minor improvement for the set of applications we used. With the costs for bloom filters, and the added complexity for the unsafe page table updates, shrinking does not seem to be very useful for our benchmark applications. However, shrinking will become important to support long running applications, where sharing changes and thread migrations will occur. Improving the shrinking mechanism will be our future work.

8. ACKNOWLEDGMENTS

We would like to thank Doug Burger, who contributed a lot to the inital study for this work. We also thank the annonymous reviwers for their comments. This work is supported by the IT R&D Program of MKE/KEIT. [2010-KI002090, Development of Technology Base for Trustworthy Computing]

9. REFERENCES

[1] IBM PowerPC 750FX and 750FL RISC Microprocessor User's Manual, Mar 2006.

[2] N. Agarwal, L.-S. Peh, and N. K. Jha. In-Network Coherence Filtering: Snoopy Coherence Without Broadcasts. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*, pages 232–243, New York, NY, USA, Dec. 2009. ACM.

[3] N. Agarwal, L.-S. Peh, and N. K. Jha. In-Network Snoop Ordering(INSO) : Snoopy coherence on unordered interconnects. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[4] M. Awasthi, K. Sudan, R. Balasubramonian, and J. Carter. Dynamic hardware-assisted software-controlled page placement to manage capacity allocation and sharing within large caches. In *Proceedings of the 15th International Symposium on High Performance Computer Architecture (HPCA)*, pages 250–261, 2009.

[5] M. Azimi, F. Briggs, M. Cekleov, M. Khare, A. Kumar, and L. P. Looi. Scalability port: A coherent interface for shared memory multiprocessors. In *Proceedings of the 10th Symposium on High Performance Interconnects HOT*

Interconnects, page 65, Washington, DC, USA, 2002. IEEE Computer Society.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques(PACT)*, October 2008.

[7] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood. Multicast snooping: a new coherence method using a multicast address network. In *Proceedings of the 26th Annual International Symposium on Computer architecture(ISCA)*, pages 294–304, Washington, DC, USA, 1999. IEEE Computer Society.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[9] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving multiprocessor performance with coarse-grain coherence tracking. In *Proceedings of the 32nd annual international symposium on Computer Architecture(ISCA)*, pages 246–257, Washington, DC, USA, 2005. IEEE Computer Society.

[10] A. E. Charlesworth. The Sun Fireplane system interconnect. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, page 7, 2001.

[11] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 27(2):10–21, 2007.

[12] N. Eisley, L.-S. Peh, and L. Shang. In-Network Cache Coherence. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*, pages 321–332, Washington, DC, USA, 2006. IEEE Computer Society.

[13] M. Ekman, P. Stenström, and F. Dahlgren. 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 (ISLPED)*, pages 243–246, New York, NY, USA, 2002. ACM.

[14] N. D. Enright Jerger, L.-S. Peh, and M. H. Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. In *Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture*, pages 35–46, Washington, DC, USA, 2008. IEEE Computer Society.

[15] C. Fensch and M. Cintra. An OS-based alternative to full hardware coherence on tiled CMPs. In *Proceedings of the 14th International Conference on High Performance Computer Architecture (HPCA)*, pages 355–366, 2008.

[16] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture(ISCA)*, pages 184–195, 2009.

[17] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[18] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared memory multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture(ISCA)*, pages 206–217, June 2003.

[19] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Proceedings of the 30th Internalitonal Symposium on Computer Architecture(ISCA)*, pages 182–193, June 2003.

[20] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator GEMS toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, 2005.

[21] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth adaptive snooping. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2002.

[22] M. R. Marty and M. D. Hill. Coherence ordering for ring-based chip multiprocessors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture(MICRO)*, pages 309–320, Washington, DC, USA, 2006. IEEE Computer Society.

[23] A. Moshovos. RegionScout: Exploiting coarse grain sharing in snoop-based coherence. In *Proceedings of the 32nd International Symposium on Computer Architecture(ISCA)*, June 2005.

[24] 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 (HPCA)*, pages 85–96, 2001.

[25] L.-S. Peh, N. Agarwal, N. Jha, and T. Krishna. GARNET: A detailed on-chip network model inside a full-system simulator. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, April 2009.

[26] A. Raghavan, C. Blundell, and M. M. K. Martin. Token tenure: Patching token counting using directory-based cache coherence. In *Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture(MICRO)*, pages 47–58, Washington, DC, USA, 2008. IEEE Computer Society.

[27] P. Sindhu, J.-M. Frailong, J. Gastinel, M. Cekleov, L. Yuan, B. Ghnning, and D. Curry. XDBus: A High-Performance, Consistent, Packet-Switched VLSI Bus. *IEEE Compcon*, pages 338–344, 1993.

[28] K. Strauss, X. Shen, and J. Torrellas. Flexible snooping: Adaptive forwarding and filtering of snoops in embedded-ring multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture(ISCA)*, pages 327–338, Washington, DC, USA, 2006. IEEE Computer Society.

[29] P. J. Teller. Translation-lookaside buffer consistency. *Computer*, 23(6):26–36, 1990.

[30] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture(ISCA)*, pages 24–36, Santa Margherita Ligure, Italy, 1995.

[31] J. Zebchuk and A. Moshovos. Regiontracker: A case for dual-grain tracking in the memory system. Technical report, Computer Group, University of Toronto, 2006.