ALLARM: Optimizing Sparse Directories for Thread-Local Data

Amitabha Roy
EPFL
amitabha.roy@epfl.ch

Timothy M. Jones
University of Cambridge
timothy.jones@cl.cam.ac.uk

***Abstract*—Large-scale cache-coherent systems often impose unnecessary overhead on data that is thread-private for the whole of its lifetime. These include resources devoted to tracking the coherence state of the data, as well as unnecessary coherence messages sent out over the interconnect. In this paper we show how the memory allocation strategy for non-uniform memory access (NUMA) systems can be exploited to remove any coherence-related traffic for thread-local data, as well removing the need to track those cache lines in sparse directories. Our strategy is to allocate directory state only on a miss from a node in a different affinity domain from the directory. We call this ALLocAte on Remote Miss, or ALLARM. Our solution is entirely backward compatible with existing operating systems and software, and provides a means to scale cache coherence into the many-core era. On a mix of SPLASH2 and Parsec workloads, ALLARM is able to improve performance by 13% on average while reducing dynamic energy consumption by 9% in the on-chip network and 15% in the directory controller. This is achieved through a 46% reduction in the number of sparse directory entries evicted.**

I. Introduction

The projected move towards dies with a large number of cores has raised an interesting question about the need for cache coherence in such systems. One line of argument is that the cost of cache coherence is an unnecessary overhead in terms of both network traffic and die area needed to maintain directory state. Along these lines, some researchers have suggested moving away from shared memory models [16]. On the other hand, Martin et al. have argued that cache coherence can be scaled with the number of cores without undue impact on either network traffic or die area [18]. Specifically, they show that directory state can accurately track the state of lines with an asymptotic cost of $O(\sqrt{n})$ while network traffic due to coherence is an amortized constant independent of the number of cores. Mainstream systems today are cache coherent and will likely remain so in the foreseeable future.

In addition to this, there is no need to move away from robust shared-memory operating systems, as some have pointed out [9]; and we are at a point where mainstream programming languages are being enhanced with memory consistency models [8], and cache-coherence-linked abstractions, such as transactional memory, have appeared in hardware [1]. There is no need to discard these useful advances.

However, the area overheads and network traffic implications of cache coherence still remain high. Although the growth in coherence overheads may be manageable with an increasing number of cores, one would like as few overheads as possible from cache coherence, especially for workloads with small amounts of communication between cores. Given increasing wire delays and dark silicon, reducing network traffic and directory size can bring large returns in terms of performance and dynamic power.

This paper presents a technique to further these aims. It is called ALLARM, or ALLocAte on Remote Miss. It works by removing the coherence overheads for data that is local to a particular thread for the whole of its lifetime. Others have already pointed out the prevalence of thread-private data and proposed changes to coherence protocols to reduce its overheads in purely snoopy systems [15] and with sparse directories [12]. Unlike these approaches, however, ALLARM is completely compatible with existing software, requiring no changes to take advantage of it. Other than the microarchitectural support for ALLARM, all the ingredients for our proposal already exist and are deployed in systems today. ALLARM has the following features:

- Is transparent to existing software;
- Requires no directory entries for thread-private data;
- Creates no coherence network traffic when accessing thread-private data;
- Incurs only small and simple changes to the existing directory controller microarchitecture.

ALLARM allows the construction of systems where the area devoted to cache line tracking in the directory and the amount of network traffic generated depends solely on the amount of data that is shared between threads. We therefore reward programmers who invest effort in separating data that is accessed by different threads. This is already a concern for many developers, since fine-grained sharing of data leads to problems of synchronization. *We believe that investing time in data partitioning for a cache-coherent system is less demanding than managing data on a non-coherent one.* This is in line with recent arguments about disciplined shared-memory programming as a means to extract better performance from shared-memory hierarchies [10]. However, for applications that have not been optimized in this manner, ALLARM will continue to work seamlessly under the hood.

We evaluate ALLARM on a range of SPLASH2 and Parsec applications, showing that it is able to reduce the number of sparse directory entries evicted by 46% on average, leading to a 13% performance increase and dynamic energy reductions of 9% in the on-chip network and 15% in the directory controller.

978-3-9815370-2-4/DATE14/©2014 EDAA

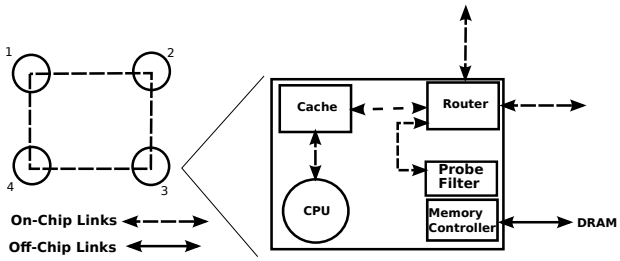


Fig. 1: A 2x2 mesh of nodes. Each node contains a core, cache and sparse directory (probe filter) attached to the memory controller.

II. ALLARM

ALLARM is targeted towards Non-Uniform Memory Allocation (NUMA) systems with sparse-directory-based cache coherence, which are standard in deployed systems. An example of such a system is shown in Figure 1.

In NUMA systems latency from the core to memory depends on the location of the core, so careful memory allocation is critical for reducing latency. This is done by partitioning the system into sets of cores and associated local memory called affinity domains. A popular (and, in fact, for most operating systems, default) mode of allocation is first-touch allocation. This policy allocates a page of memory from the affinity domain in which the first access to allocated memory is made, assuming that the first access is a good hint about which processor is likely to most frequently access the page.

A commonly-employed modification to first-touch allocation is next-touch allocation, which fixes problems with access patterns where a large amount of data is initialized by one thread (first-touch) but used (exclusively) by another thread (next-touch). ALLARM works seamlessly with both first-touch and next-touch allocation strategies.

A. Detecting Private Data

Given a policy of first-touch memory allocation, we assume that requests from remote cores are to shared data and requests from the local core are to thread-private data.

The first assumption holds true under first-touch allocation as thread-local data must be allocated in the local node. The second assumption holds true *in the common case*. Requests to shared data are more likely to originate from one of the remote cores rather than from the local core. We note that first-touch allocation is a best-effort policy. In the event that the operating system is unable to allocate requested memory in the local node, it is allocated at one of the remote nodes. The assumptions therefore hold true in the common case. *ALLARM does not depend on them for correctness, but depends on them for performance.* A nice property of this detection scheme is that it is stateless; we do not add any tracking state to any software or hardware structure, nor do we incur cost or complexity in updating any state. A positive aspect of ALLARM, in comparison to other proposals [12], [15], is simplicity. Requests from the local core are serviced without allocating a probe filter entry. A probe filter entry is therefore only allocated on a remote miss.

Within ALLARM, the probe filter continues to be the single point of reference for state, and we therefore lookup the probe filter on any incoming request (local or remote). If the requested cache line has an entry in the probe filter, we proceed as normal. If the requested line is not present in the probe filter, however, we follow one of two different paths depending on whether the request is from a local core or a remote one. If the request is from the local core, we do not allocate a line in the probe filter. On the other hand, if the request is from a remote core, we allocate a probe filter entry and probe the (unique) local core to determine the state of the line. The probe response determines the final state of the line and the request is then handled as normal.

B. Benefits of ALLARM

ALLARM eliminates probe filter entry allocation for requests from the local core. Since first-touch allocation attempts to allocate pages for thread-local data on the local affinity node, most thread-local data benefits from ALLARM. The elimination of probe filter entry allocation for thread-local data has the following benefits:

1) Reduced dynamic power in the probe filter;
2) Better cache hit rates since there are no probe filter evictions removing a needed line from underlying cores;
3) Reduced network traffic because a probe filter eviction would have necessitated an invalidate message being sent out to one or more cores.

Each probe filter eviction requires a read-out of tag and data for the replacement way. The replacement is then written into the probe filter. Both these operations consume dynamic power and we therefore expect a reduction in probe filter evictions to reduce the dynamic energy consumption of the probe filter.

A probe filter eviction also invalidates the cache line in all caches. For some workloads sensitive to cache misses this can cause a degradation of performance due to the increased miss rate. ALLARM, therefore, can improve performance by increasing cache hit rates.

Last but not least, each eviction requires at least one invalidate message to be sent out on the network, which is responded to by an acknowledgment message. Reducing probe filter evictions, therefore, also reduces network traffic.

C. Microarchitectural Changes

A key goal with ALLARM is to minimize the amount of microarchitectural change needed for implementation. ALLARM requires only the following microarchitectural changes:

a) Network: An extra message type is needed to be able to query a local cache about the current state of a line.

b) Cache: The cache needs to be able to respond to a probe message from a directory requesting the state of a line. The baseline coherence protocol would only support messages that are part of a request flow from a remote cache.

c) Directory: The most extensive changes necessary for ALLARM are in the directory controller. It must be modified for the ALLARM protocol. ALLARM has been deliberately structured as an addition to the existing protocol. This means that no additional flops are needed in the implementation of ALLARM, as it requires no extra storage to track in-flight requests, beyond those that already exist. The additions are only control logic, which means that area and power overheads are negligible and are subsumed by the reductions that we show in Section III.

An implementation of ALLARM could be optional, based on physical memory ranges. This could be configured at boot-time with range registers associated with each directory controller (analogous to the MTRRs present, for example, on current x86 microprocessors), specifying ranges on which ALLARM is active. We note that it is always possible to move from a non-ALLARM mode to an ALLARM mode at run-time, as the probe filter is always consulted, regardless of the mode of operation. Moving from an ALLARM to a non-ALLARM mode, however, would require flushing that range of physical addresses from the local core.

D. Impact on Remote Accesses

ALLARM introduces an extra step in the servicing of requests from a remote core when there is not already a probe filter entry allocated. Remote accesses that miss in the probe filter, therefore, see an increase in latency. This increase can be effectively hidden if the following two conditions hold true:

1) The probe of the local cache returns a miss, so the cache line is not cached locally;
2) The time to lookup DRAM is larger than the time to probe the local cache.

When both these conditions hold true, the DRAM lookup is the necessary critical path and hence the probe of the local cache happens in parallel and is hidden. The first condition is mainly true because the first access to a piece of shared data is more likely to come from one of the numerous remote cores than the local core. The second assumption also holds true on most systems. The local probe usually travels exclusively along on-die links and looks-up fast on-chip SRAM (access latency < 10 ns). The DRAM read, on the other hand, must traverse an off-die link to access slower DRAM (access latency > 40 ns). Hence, ALLARM is successful in hiding the impact on remote accesses. We confirm this fact in our evaluation.

E. Deploying ALLARM

ALLARM may only be enabled for a single core per affinity domain. This is not an unreasonable assumption as deployed NUMA systems are often configured as a memory controller per-die and a shared last-level cache handling coherence between cores on the die (for example, the AMD Mangy-Cours and Intel Nehalem EX processors). In this situation the directory sees the single last-level cache as the single core in its affinity domain, rather than individual cores. For systems where this is not the case, ALLARM may still be used by partitioning the physical memory into logical affinity domains and associating each core in the affinity domain with a single partition.

Core/Per-Core Cache			
Cores	16	Frequency	2Ghz
Block size	64 bytes	Access Latency	1ns
ICache	32kB, 4-way	DCache	32kB 4-way
L2Cache	256kB 4-way (exclusive)		
Directory/DRAM			
Directory	Tracks 512kB of cached data, 1ns access lat		
Memory	2GB, 60ns access lat		
OS	Linux 2.6.28, NUMA enabled		
Network			
Topology	4x4 Mesh	Flit size	4 bytes
Control Msg	8 bytes	Data Msg	72 bytes
Link BW	8 GB/s	Link Latency	10ns

TABLE I: Simulated system.

We have also discounted the effect of thread migration in the discussions above, although ALLARM still works when migration occurs. Thread migration is generally avoided by schedulers in NUMA systems because moving a thread causes its affinity to change, rendering previously-allocated memory on the local controller remote. In addition, high-end NUMA systems also support page migration, which allows locally-allocated memory to move to the new affinity domain. Hence, although thread migration is generally avoided in NUMA systems, when it does occur there are existing techniques to reduce the performance impact and ALLARM can take advantage of these too.

Finally, large-scale, deployed NUMA systems with sparse directories scale up to hundreds of thousands of cores. This is often accomplished through the use of hierarchical directories, which ALLARM would have no problem in accommodating.

III. Evaluation

We evaluate ALLARM using the GEM5 full system-simulator [7] with 16 cores, running the x86 ISA. We boot Linux on this system with the operating system configured for NUMA support. The system is organized as a 4x4 mesh, similar to Figure 1. The system is configured with 2GB of DRAM divided into sixteen 128MB blocks, each attached to a directory controller. The system runs the Hammer protocol [11] for cache coherence. Each node (memory, directory controller and core) is an affinity domain for the operating system. Each core has separate instruction and data caches and private L2 cache. Shared L2 caches would decrease the number of coherence agents and cause more lines to appear private to ALLARM, thus the private L2 we have evaluated represents a more challenging scenario. The size of the caches and latencies in the system are shown in Table I. The probe filter provides 2X coverage of an L2 cache (*exactly* the same as deployed AMD Hammer systems). The baseline performance model includes support for notifying the directory when an exclusively owned block is evicted from cache and thus represents an already optimized implementation. *Our baseline therefore accurately reflects real deployed systems.* We do not bind threads to cores, meaning that the Linux scheduler is free to migrate threads.

We evaluate ALLARM on a subset of the SPLASH2 [20] and Parsec [6] benchmarks, both well-known multi-threaded

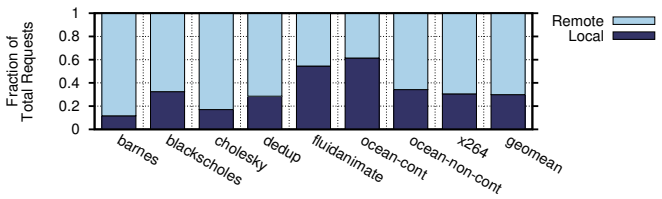


Fig. 2: Ratio of local to remote accesses.

benchmark suites, using standard inputs (simmedium for Parsec). Simulation time requires us to use smaller input sets but we scale cache sizes down as is standard in related work [12], [15]. Hence, as these works claim, the simulation results in this paper are valid for the larger input sets on the native systems.

A critical determinant of performance in ALLARM is the ratio of local to remote accesses. Figure 2 shows how this varies (averaged over all directories) for our benchmarks. We have deliberately picked benchmarks with the majority of accesses being remote, to ensure that we pick the most challenging workloads possible for ALLARM and test our hypothesis from Section II-D.

A. Multi-Threaded Performance

The first set of experiments we perform are 16-threaded runs of all the benchmarks.

1) Speedup: Figure 3a shows that on average we obtain a 12% speedup over the baseline performance. The gains vary across applications, with the *ocean* benchmarks showing the largest speedups of up to 40%. The SPLASH2 benchmarks show larger speedups than Parsec as they are more NUMA-friendly (Figure 2) and show better data isolation [4]. Although all benchmarks show a significant fraction of remote accesses, the only benchmark with a degradation of performance is *fluidanimate*.

The *fluidanimate* benchmark, has a large fraction of local accesses (Figure 2) but shows a slowdown with ALLARM (Figure 3a). This is because of its large working set [6] compared to the other benchmarks, leading to capacity misses dominating traffic to memory. This is also confirmed by relatively small change in cache miss rate, notwithstanding the reduction in probe filter evictions. ALLARM is unable to therefore balance its (modest) overheads through gains from reduced probe filter evictions. We note that ALLARM can be made optional both on a per-directory and per-physical-memory range basis. This can be used to avoid slowdowns in benchmarks like fluidanimate, where capacity misses rather than coherence misses dominate last level cache misses.

2) Probe Filter, Network, and Caches: The primary driver for improved performance is reduced probe filter evictions. Figure 3b shows the reduction in evictions over the baseline. On average we obtain a 45% reduction in probe filter evictions. This reduction correlates well with the fraction of local accesses seen by a directory (Figure 2). The larger this fraction, the better ALLARM is at reducing probe filter evictions.

A direct consequence of reduced probe filter evictions is lower network traffic. Figure 3c shows the reduction in network traffic (measured in bytes) achieved with ALLARM. On average we reduce network traffic by 12% across the benchmarks. This reduction is dependent on the average number of invalidation messages and responses per probe filter eviction. We plot the average number of messages sent per eviction in Figure 3d. number of messages often exceeds two, ALLARM also reduces cache misses due to unnecessary invalidations, shown in Figure 3e. On average, ALLARM reduces L2 cache misses by 9%.

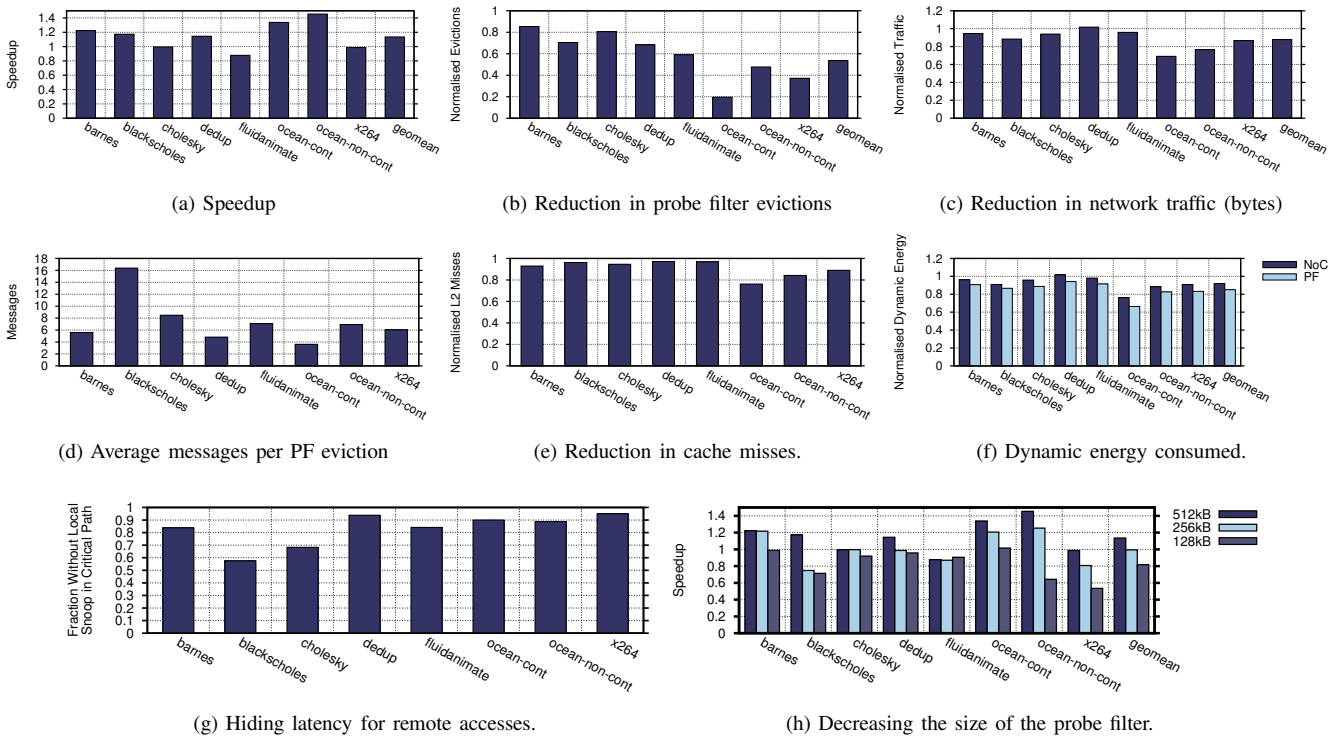
3) Dynamic Energy: Reducing the number of probe filter evictions and the amount of network traffic reduces the dynamic energy consumption of both components. We evaluate the dynamic energy consumption of ALLARM using McPAT [17], assuming a 32nm process. Figure 3f shows the reduction in dynamic energy achieved by ALLARM. On average we reduce dynamic energy consumption by 8% for the on-chip network and by 14% for the probe filter. The savings are as high as 30% in the case of *ocean-contiguous*. ALLARM, therefore, improves performance and reduces dynamic energy consumption, leading to lower energy consumption across the set of benchmarks.

4) Hiding the Latency of Remote Misses: An important component of ALLARM that enables it to perform well, even with a large fraction of remote accesses, is its ability to hide the extra latency for servicing a remote request at a directory (Section II-D). To test this hypothesis, we measured the fraction of requests where the extra local probe in ALLARM was *not* on the critical path, averaged across all the directories. The results, shown in Figure 3g, show that for a large fraction of remote requests (81% on average) the local probe is indeed not on the critical path to resolving the request.

5) Decreasing Probe Filter Size: Finally, we consider the performance of ALLARM when we decrease the size of the probe filter. Figure 3h shows these results, with each bar normalized to the baseline with a 512kB probe filter. With a 256kB probe filter, ALLARM maintains high performance for the majority of benchmarks. Only *blackscholes* is strongly affected, due to its pattern of sharing between threads. In this benchmark, a large amount of data is shared from the probe filter for CPU 0, which initializes the data for the other threads. Therefore, reducing the probe filter size for this benchmark causes significant reductions in performance. For other benchmarks the reductions are less pronounced. Reducing again to a 128kB probe filter size with ALLARM causes slowdowns on additional benchmarks (*ocean non-contiguous* and *x264* mainly). For *barnes* and *ocean contiguous* though, performance is only reduced to the baseline level, meaning that ALLARM can enable a 4$\times$ reduction in probe filter size for these types of workload.

B. Multi-Process Performance

Our key motivation with ALLARM was to optimize for workloads or application phases that are geared towards accessing thread-private data. In a multi-process scenario there is little sharing between the processes and is a common form of parallelism seen in data centers and programs communicating with MPI. We simulated this setup with the SPLASH2 benchmarks by running two copies of the benchmark, with each copy using a single thread. The copies are co-ordinated



(a) Speedup

(b) Reduction in probe filter evictions

(c) Reduction in network traffic (bytes)

(d) Average messages per PF eviction

(e) Reduction in cache misses.

(f) Dynamic energy consumed.

(g) Hiding latency for remote accesses.

(h) Decreasing the size of the probe filter.

Fig. 3: Speedup and impact of ALLARM on the system.

(using shared memory segments) to start their region of interest together and we measured the time taken for both copies to finish executing. Figure 4a shows that for the baseline, performance suffers with a decreasing probe filter size. On the other hand, Figure 4d shows that on enabling ALLARM, the execution of the tasks is largely unaffected by the size of the probe filter, showing only a minor degradation past 32kB.

The reason for the better performance of ALLARM is its effect on the number of evictions from the probe filter. As Figure 4b illustrates, the growth in the number of probe filter evictions is dramatic due to the pressure from the reduced probe filter size. In contrast, as Figure 4e shows (note different y-axis scale), the number of probe filter evictions with ALLARM only grows significantly when the size drops lower than 64kB: as capacity limitations at a single memory controller means some frequently used data needs to be allocated remotely. Reducing the directory size dramatically increases the eviction rate for these entries. However, note that a majority of data for the benchmark is allocated locally and therefore performance only gradually degrades with ALLARM.

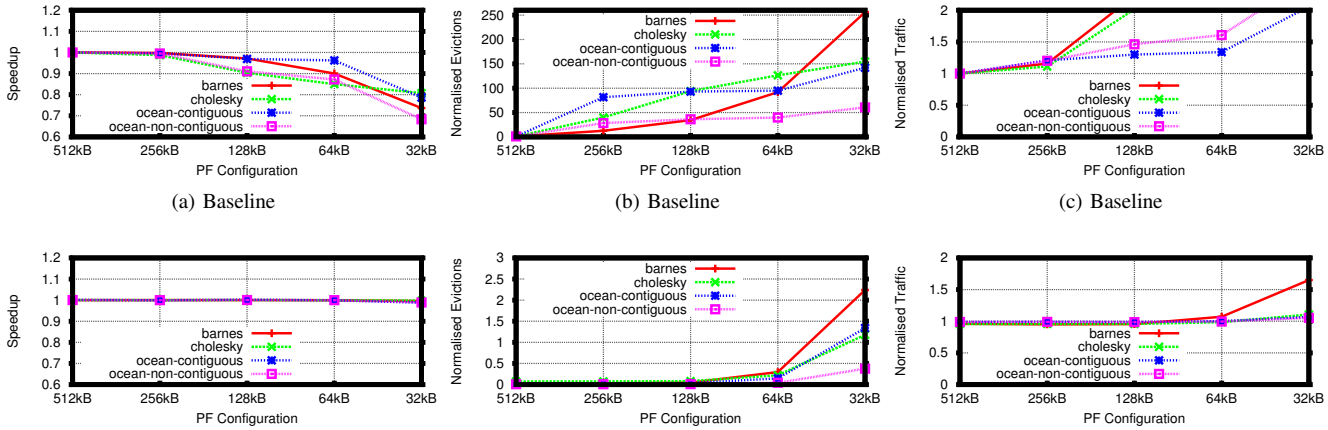
The same observation also extends to the amount of network traffic. On reducing the probe filter size, the amount of network traffic grows, as shows in Figure 4c. On the other hand, with ALLARM network traffic remains roughly constant and is unaffected by the size of the probe filter, as shown in Figure 4f. Again, we see a more moderated growth in network traffic with ALLARM. A interesting point to note is that network traffic growth, although correlated to the growth in probe filter evictions, is at a lower rate. This is because not all probe filter evictions lead to a writeback, while the triggering allocation at the directory always requires a 64 byte cache line transfer back to the requesting core. Therefore, an increase in invalidations has a scaled impact on overall traffic in bytes.

We note that current deployments of sparse directories in x86 microprocessors reserve a part of the last-level cache to serve as the probe filter. Reducing the size of the probe filter and returning the on-chip SRAM to cache can significantly boost benefits provided that the lower probe filter capacity does not impact performance. The table below quantifies the area benefits of ALLARM (using McPAT) as we vary the size of the probe filter, showing the amount of space that can be reused as cache again.

PF Configuration	512kB	256kB	128kB	64kB	32kB
Area (mm^2)	70.89	26.95	19.90	8.20	5.93

IV. Related Work

Distinguishing thread-private data at run-time and using it to improve processor performance has been explored by a number of researchers. Three closely-related pieces of work, are those of Kim et al. [15], Cuesta et al. [12] and Das et al. [13]. These approaches track sharing at a page granularity. In particular, Cuesta et al. also consider reducing coherence overheads in the Hammer protocol. Further, they also do not allocate probe filter entries for thread-private data. Both the sub-space snooping approach of Kim et al. and the coherence deactivation approach of Cuesta et al., however, track page sharing information using spare page table bits. The dynamic directories approach of Das et al. uses information encoded in



(a) Baseline (b) Baseline (c) Baseline

(d) ALLARM (e) ALLARM (f) ALLARM

Fig. 4: Speedup, evictions and NoC traffic for each workload, normalized to the baseline with a 512kB probe filter.

page tables to place directories close to the CMP tiles requesting it. These approaches are limited by the number of available bits and require operating system modifications in support. In addition, these approaches do not consider the impact of aliasing, a problem known to researchers who have considered the interaction of coherence and virtual addresses [5]. With virtual aliasing (a common feature on many modern operating systems due to copy-on-write), two page table entries can point to the same physical page. In contrast, ALLARM is stateless and requires no new features in the operating system.

A way to reduce overheads in general for directories is to reduce the cost of encoding sharer sets. We note that the Hammer protocol used in this paper, does not track sharers. Variants range from sharer pointers [2] to recent proposals for tagless directories using Bloom filters [21].

Another related direction of research is hardware structures to detect sharing at larger granularities than a cache line to filter coherence traffic, such as Region Scout [19]. A purely software-based is Fensch et al.'s [14] proposed coherence scheme that uses OS support for coherence. There have also been proposals to track sharing and filter redundant snoops in the network, such as Agarwal et al.'s in-network coherence filtering [3] that maintains some amount of state at each router in the on-chip network to filter snoops. In contrast to all these proposals, ALLARM requires only simple changes to the directory controller that can be optionally disabled on a per-controller, or even on a per-physical memory range basis.

V. Conclusion

We have presented ALLARM, a technique for sparse directories that provides significant savings in energy while simultaneously improving performance. ALLARM enables the construction of mainstream shared-memory systems where the area devoted to directories and amount of network traffic grows only in relation to the data that is actually shared.

Acknowledgments

This work was funded by the Royal Academy of Engineering and EPSRC.

References

[1] *Intel Architectures Instruction Set Extensions Programming Reference*, February 2012.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *ISCA*, 1988.

[3] N. Agarwal, L.-S. Peh, and N. K. Jha. In-network coherence filtering: snoopy coherence without broadcasts. In *MICRO*, 2009.

[4] N. Barrow-Williams, C. Fensch, and S. Moore. A communication characterization of SPLASH-2 and PARSEC. In *IISWC*, 2009.

[5] A. Basu, M. D. Hill, and M. M. Swift. Reducing memory reference energy with opportunistic virtual caching. In *ISCA*, 2012.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[7] N. Binkert et al. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2), 2011.

[8] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.

[9] S. Boyd-Wickizer et al. An analysis of Linux scalability to many cores. In *OSDI*, 2010.

[10] B. Choi et al. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *PACT*, 2011.

[11] P. Conway et al. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30(2), 2010.

[12] B. A. Cuesta et al. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *ISCA*, 2011.

[13] A. Das, M. Schuchhardt, N. Hardavellas, G. Memik, and A. N. Choudhary. Dynamic directories: A mechanism for reducing on-chip interconnect power in multicores. In *DATE*, pages 479–484, 2012.

[14] C. Fensch and M. Cintra. An OS-based alternative to full hardware coherence on tiled CMPs. In *HPCA*, 2008.

[15] D. Kim, J. Ahn, J. Kim, and J. Huh. Subspace snooping: filtering snoops with operating system support. In *PACT*, 2010.

[16] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5), 2006.

[17] S. Li et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*, 2009.

[18] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 55(7), 2012.

[19] A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *ISCA*, 2005.

[20] S. C. Woo et al. The SPLASH-2 programs: characterization and methodological considerations. In *ISCA*, 1995.

[21] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A tagless coherence directory. In *MICRO*, 2009.