Protozoa: Adaptive Granularity Cache Coherence*

Hongzhou Zhao†, Arrvindh Shriraman*, Snehasish Kumar*, and Sandhya Dwarkadas†
†Department of Computer Science, University of Rochester
*School of Computing Sciences, Simon Fraser University
{hozhao,sandhya}@cs.rochester.edu and {ashriram,ska124}@sfu.ca

ABSTRACT

State-of-the-art multiprocessor cache hierarchies propagate the use of a fixed granularity in the cache organization to the design of the coherence protocol. Unfortunately, the fixed granularity, generally chosen to match average spatial locality across a range of applications, not only results in wasted bandwidth to serve an individual thread's access needs, but also results in unnecessary coherence traffic for shared data. The additional bandwidth has a direct impact on both the scalability of parallel applications and overall energy consumption.

In this paper, we present the design of Protozoa, a family of coherence protocols that eliminate unnecessary coherence traffic and match data movement to an application's spatial locality. Protozoa continues to maintain metadata at a conventional fixed cache line granularity while 1) supporting variable read and write caching granularity so that data transfer matches application spatial granularity, 2) invalidating at the granularity of the write miss request so that readers to disjoint data can co-exist with writers, and 3) potentially supporting multiple non-overlapping writers within the cache line, thereby avoiding the traditional ping-pong effect of both read-write and write-write false sharing. Our evaluation demonstrates that Protozoa consistently reduce miss rate and improve the fraction of transmitted data that is actually utilized.

1. Introduction

Harnessing the many cores afforded by Moore's law requires a simple programming model. The global address-space supported by shared memory isolates the programmer from many of the hardware-level issues and frees the programmer from having to coordinate data movement between threads. Unfortunately, an efficient hardware implementation that satisfies the performance and power constraints remains a particular challenge. Prior research has addressed auxiliary directory overheads [35, 36]. A few works have proposed more streamlined cache coherence protocols [16, 19, 32] that eliminate both the traffic and complexity penalty of the coherence control messages. However, these alternatives present a significant departure from current state-of-the-art directory protocols [6] and introduce specific challenges of their own. A few have even advocated the abandoning of cache coherency in hardware and proposed scratchpad-based multicores [31] that rely on software [25]. In support of the position, "Why On-Chip Coherence" [15], we show that the overheads of cache coherence are not inherent to the model but are caused by the rigidity of the hardware implementations.

We find that the key impediment to scalability of cache coherent systems is the volume of data moved in the system. Across a range of applications in two different languages (C/C++ and Java) and three different programming models (pthreads, openMP, and Intel TBB) we find that between 68–82% of the total traffic is expended in moving data. We define unused data as words in the block that are fetched into the private cache but are untouched before the block is evicted. Within the data messages moved in the system we also identified that there is plenty of unused data (31–87%). Recent reports from industry [2] show that on-chip networks can contribute up to 28% of total chip power and can also suffer from contention. We seek to eliminate unused data movement to mitigate these issues.

The primary cause of the excess data volume is the rigidity of current cache hierarchies. State-of-the-art multicore cache hierarchies typically maintain coherence in coarse-grain (64 bytes) cache block units, the size of which is uniform and fixed at design time. This also fixes two important system parameters a) the **storage/communication granularity**: the granularity at which storage is managed and data is moved around and b) the **coherence granularity**: the granularity at which coherence read/write permissions are acquired from remote sharers. The mismatch between the applications' spatial locality and the storage/communication granularity leads to a large fraction of unused data being moved around in fixed granularity data blocks. The mismatch between the applications' sharing granularity, synchronization, and coherence granularity leads to false sharing related performance and bandwidth penalty. We discuss the specifics in more detail in Section 2.

In this paper, we present Protozoa, a family of coherence protocols that decouples storage/communication granularity from the coherence granularity and supports adaptive dynamic per-block adjustment of the granularity. Protozoa is built as an extension to the conventional MESI directory protocol and works with existing directories that track sharers at coherence granularity. It reduces the volume of data moved by supporting 1) per-block adjustment of storage/communication granularity so that the data transfer matches application spatial granularity and eliminates unused data, 2) adaptive granularity coherence so that readers to disjoint data in a conventional cache block can co-exist with concurrent writers, and 3) multiple non- overlapping writers within the cache line, thereby avoiding the traditional ping-pong effect of both read-write and write-write false sharing.

We constructed the Protozoa protocols in detail and analyzed the complexity in terms of # of new states introduced, and the extra transitions that arise from them. Our evaluation demonstrates that Protozoa consistently reduces application miss rates and improves the fraction of transmitted data that is actually utilized. We see significant reduction in data traffic and network utilization, with a 37% reduction in traffic at the L1 level. The corresponding reduction in network dynamic energy is 49% on average (geometric mean). The

*This work was supported in part by National Science Foundation (NSF) grants CCF-0702505, CNS-0615139, CNS-0834451, CCF-1016902, CCF-1217920, and CNS-0509270; and by an NSERC Discovery Grant, NSERC SPG Grant, NSERC CRD Grant, MARCO Gigascale Research Center, and Canadian Microelectronics Corporation.

ISCA'13 Tel-Aviv, Israel

adaptive granularity storage/communication support reduces the L1 miss rate by 19% on average and the adaptive granularity coherence support completely eliminates false sharing, resulting in up to a 99% reduction (Linear regression) in misses, with an average 36% reduction. Overall, we see a 4% improvement in performance compared to MESI since parallelism can hide the latency penalty of the extra misses.

Scope of Work.

Protozoa draws its inspiration from software-based disciplined race-free programming models that can dynamically vary the storage/communication granularity [5, 11], as well as from earlier work on adaptive software-managed DSM systems [18, 26, 27, 38] from the '90s that adapt data communication granularity to match access behavior. Unlike such systems, Protozoa does not rely on a specific programming model or software support. Building on existing MESI directory protocols and coherence directory hardware, Protozoa targets single-chip multicores, making the blocks more fine-grained to reduce the overall bandwidth overheads and eliminate false sharing. Protozoa effectively implements the Single-Writer or Multiple-Reader (SWMR) invariant [29] at word granularity without paying word-granularity implementation overheads. Finally, Protozoa enables the integration of adaptive granularity caches [8, 23, 33] into multicores; prior art has largely explored them in the context of single core systems.

The overall paper is organized as follows. Section 2 motivates the need for adaptive granularity storage/communication and coherence, discussing the impact on miss rate, false sharing, and data wasted as the fixed granularity is varied. Section 3 presents the details of *Protozoa-SW* and *Protozoa-MW*, coherence protocols that decouple storage/communication granularity from the coherence granularity. We discuss the state machine and the additions to a conventional MESI protocol. We also present a discussion on the additional races introduced by Protozoa. Section 4 presents a detailed evaluation of the system and compares the baseline MESI and the Protozoa family of protocols on a 16 core system. Finally, we discuss related work in detail in Section 5.

2. Motivation and Background

With hardware cache coherent systems, the cache block granularity influences many of the system parameters. A block is the fundamental unit of data caching and communication, the granularity of coherence operations, and coherence metadata management. While using the same granularity for these disparate components in the system simplifies the implementation, it imposes fundamental limitations on adaptivity to application behavior, causing inefficiency in one or more parts of the system. We discuss the impact of fine-grain (< 8 words) and coarse-grain ($>= 8 words$) cache blocks.

Storage/Communication Granularity.

A cache block is the fundamental unit of space allocation and data transfer in the memory hierarchy. Since programs access state in units of variables and data objects, any mismatch between the block granularity and program access granularity results in unused words in the block. These unused words increase network bandwidth utilization and dynamic energy consumption. These unused words constitute a dominant overhead even in parallel applications as shown in Table 1. Used data can be as low as 16% of transferred-data, causing up to 65% of the overall bandwidth in a conventional cache coherence protocol to be wasted on unused data; while only 19% is spent in control messages. Finer-granularity cache blocks have fewer unused words but impose significant performance penalties by missing opportunities for spatial prefetching in applications with higher spatial locality. Furthermore, they increase the % of traffic wasted on coherence control messages.

Coherence Granularity.

This defines the granularity at which the coherence protocol acquires read and/or write permissions. The coherence protocol sequences control messages and manipulates processor cache states to satisfy processor memory references. Given that this is strictly overhead, coherence protocols seek to reduce processing overhead by acquiring permissions for coarser blocks of data. However, when the application's sharing granularity is a few words and multiple threads read/write concurrently, coarse coherence granularity leads to the phenomenon of *false sharing*, where different threads read/write to distinct parts of a block. False sharing gives rise to two forms of overhead: a) the invalidation penalty to force a concurrent reader/writer to eliminate its current cache copy and b) the bandwidth penalty to move the block again on subsequent access.

State-of-the-art coherence protocols typically use the same fixed block size for the storage/communication granularity, coherence granularity, and metadata granularity. However, each of these system parameters shows best behavior at different block sizes. As we discuss below, the challenges are compounded due to the application-specific nature of the behavior.

2.1 Application Behavior

While applications access data storage in units of variables, objects, and data fields, cache-based memory hierarchies have moved data in fixed-granularity chunks known as cache blocks. To study the influence of the block granularity, we vary the block size from 16 bytes to 128 bytes. We do not present block granularities > 128 bytes since this leads to a significant drop in cache utilization and increase in false sharing. We focus on three metrics: Miss rate (MPKI), Invalidations (INV), and USED-Data %. We use applications with varying sharing patterns written in two different languages (C++ and Java) and two different programming models (pthreads, openMP). Table 1 shows the relative change in each of these metrics as we vary the fixed block granularity. The best block size is chosen as the one that minimizes MPKI and INV, while maximizing USED-Data %.

Blackscholes, *linear-regression*, and *string-match* show a significant increase in the % of false sharing when going from 16 to 32 byte blocks. For blackscholes and string-match, the absolute miss rate does not change much, indicating that false sharing is only a small part of the overall application. Nevertheless, as the per-thread working set reduces (with increasing threads), the false sharing component may influence performance. With linear regression, the primary type of sharing is false sharing and it directly impacts performance. *Bodytrack*, *cholesky*, and *h2* experience significant increase in false sharing at 64 bytes. Bodytrack seems to lack spatial locality and its miss rate does not decrease with higher block sizes; at 64 bytes only 21% of the total data transferred is useful. With h2 and cholesky, the coarser block sizes are able to achieve better spatial prefetching and miss rate drops. *kmeans*, *raytrace*, *string-match*, and *streamcluster* do not achieve optimality at a fixed granularity since they have both high spatial locality read-only data and fine-grain read-write patterns. Coarser blocks benefit the high-locality read-only data and there is a significant (>33%) reduction in misses. *mat-mul*, *tradebeans*, *word-count*, and *swaptions* all have minimal inter-thread sharing and have good spatial locality. We find that coarser block granularities (64 bytes or 128 bytes) exploit the spatial locality and reduce the # of misses. *fluidanimate*, *histogram*, and *raytrace* have a combination of read-only shared data and read-write data. Coarser blocks exploit the spatial locality and reduce the # of misses, but significantly increase the # of invalidations. *Apache*, *water*, and *jbb* will benefit from spatial prefetching with larger blocks, but still experience low data transfer utilization and some read-write sharing.

Table 1: Application behavior when varying block size for the baseline MESI protocol (see Section 4 for details).

	16→32		32→64		64→128			
	MPK	INV	MPK	INV	MPK	INV	Optimal	USED%
apache	↓	↓	↓	↓	↓	↓	128	37%
barnes	↓	≈	≈	↑	≈	≈	32	37%
blackscholes	≈	⇑	≈	≈	≈	⇑	16	26%
bodytrack	⇑	≈	≈	⇑	≈	⇑	16	21%
canneal	≈	↓	≈	≈	≈	≈	32	16%
cholesky	⇓	↑	⇓	⇑	⇓	↑	*	62%
facesim	↓	⇑	↓	≈	≈	⇑	32	80%
fft	⇓	↓	↓	↓	⇓	↓	128	67%
fluidanimate	↓	⇑	↓	≈	↓	↓	128	54%
h2	⇓	⇑	↓	⇑⇑	↓	⇑	*	59%
histogram	↓	⇑	≈	⇑	≈	↑	32	53%
jbb	↓	↓	≈	≈	↓	↓	128	26%
kmeans	⇓	⇑	⇓	⇑⇑	⇓	↑	*	99%
linreg.	⇑⇑	⇑⇑	⇑	⇑	≈	⇑	16	27%
lu	↓	↓	↓	≈	↓	≈	128	47%
mat-mul	⇓	≈	⇓	⇑	≈	≈	64	99%
ocean	⇓	↓	↓	≈	↓	≈	128	53%
parkd	⇓	≈	⇓	⇓	⇓	⇓	128	68%
radix	⇓	↓	≈	↑	↓	≈	*	56%
raytrace	⇓	≈	↓	⇑	↓	⇑⇑	*	63%
rev-index	⇓	≈	↓	≈	↓	≈	128	64%
streamcluster	⇓	⇑	⇓	⇑⇑	↓	↓	*	76%
string-match	≈	⇑	≈	⇑	≈	≈	*	50%
swaptions	⇓	⇓	↓	↓	≈	≈	64	64%
tradebeans	≈	≈	≈	↓	≈	≈	64	32%
water	⇓	⇓	↓	↓	↓	↓	128	46%
word-count	⇓	⇓	⇓	≈	⇓	≈	128	99%
x264	↓	↓	↓	↓	≈	≈	64	24%

MPK: Misses per kilo-instructions; INV: # of invalidations; USED%: % of DATA transferred used by applications. ↑ (Increase); ↓ (Decrease). For MPK and INV metrics, lower is better.
≈ < 10% inc., ↑ 10-33% inc., ⇑ > 33% inc. ⇑⇑ > 50% inc.. *: No application-wide optimal granularity.

Summary.

Overall, we see that the coherence/metadata/storage/communication granularity rigidity of state-of-the-art coherence protocols results in unnecessary traffic and reduced performance. While we can choose a coarse granularity block size for data parallel applications (e.g., matrix multiplication, histogram, and raytrace), in general, storage/communication granularity and coherence granularity need to be independently and carefully regulated per application. Coherence granularity needs to be fine-grain enough to eliminate false sharing overheads. Storage/communication granularity needs to be coarse enough to exploit spatial locality while not wasting space and bandwidth. These goals must be accomplished without paying an up-front fixed cost for fine-granularity coherence metadata.

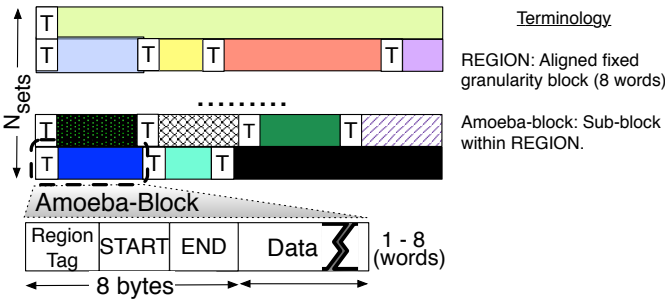
3. Protozoa: Adaptive Granularity Coherence

Protozoa coherence protocols provide variable granularity coherence that decouples storage/communication granularity from the coherence granularity. Protozoa seeks to match the data transfer granularity between the caches to the spatial locality exhibited by the applications while taking into consideration the granularity of sharing. Figure 1 illustrates the benefits of tailoring both coherence and storage/communication granularity using a toy OpenMP program snippet. The program declares an array of independent counters. Each counter is incremented by a separate OpenMP thread so that the program has no data races, showing a typical, albeit naive, example of false sharing. A conventional protocol would ping-pong the entire cache block between the processors even though only a single word is written by each thread. This results in misses at each processor caused by the invalidations and wasted bandwidth since the entire block is transferred even though each processor reads and writes only one word.

Figure 1 also illustrates the Protozoa coherence protocols proposed in this paper: a) ***Protozoa-SW*** supports adaptive granularity storage/communication but supports only a fixed coherence granularity. *Protozoa-SW* transfers and stores only the single counter word needed by each processor, saving bandwidth on both the write miss and writebacks. However, since it invalidates at a fixed block granularity, Core-0's write of item[0] invalidates Core-1's item[1], causing false-sharer induced misses. b) ***Protozoa-MW*** supports independent adaptive granularity selection for both storage/communication and cache coherence. Like *Protozoa-SW*, it transfers only the single counter word needed by each processor on a miss. Additionally, by invalidating at the granularity of the cached data, it allows Core-0 and Core-1 to cache separate words of the same block for writing at the same time, eliminating misses and coherence traffic altogether in this example. As in a conventional directory used with MESI, *Protozoa-SW* and *Protozoa-MW* track sharers at a fixed block granularity. In the next section, we describe the cache organization needed to support variable granularity storage/communication.

3.1 Variable-granularity Storage for the L1s

To support variable granularity cache coherence, we need the support of a cache that can store variable granularity cache blocks. There have been many proposals in the past that we can leverage, including decoupled sector caches [20, 28], word-organized caches [21], and most recently, Amoeba-cache [12]. We use the Amoeba-cache design in this paper as a proof-of-concept, but the coherence protocol support discussed here is applicable and portable to the other approaches to supporting variable granularity caching as well. We briefly describe only the details of *Amoeba-Cache* relevant to the paper and necessary for understanding the coherence protocol support.



Amoeba-Block: Variable granularity data block. 4-tuple consisting of block REGION identifier, START, END range markers, and the data block itself. T: Indicates tag collocated with the data.

Figure 2: Amoeba Cache Layout [12].

The *Amoeba-Cache* architecture enables the memory hierarchy to fetch and allocate space for a variable range of words (a variable granularity cache block) based on the spatial locality of the application. The *Amoeba-Cache* data array holds a collection of varied granularity *Amoeba-Block*s that do not overlap. Each *Amoeba-Block* is a 4-tuple consisting of `<Region Tag, Start, End, Data-Block>` (Figure 2). A `Region` is an aligned block of memory of size `RMAX` bytes. A region is analogous to a fixed-granularity cache block and is the basic indexing granularity for many coherence metadata (e.g., MSHRs and Directory). However, it is not the granularity of storage or communication. On cache misses, the Amoeba-cache requests a specific region's *Start*, *End* range of words from the coherence protocol. The boundaries of any *Amoeba-Block* (`Start` and `End`) always lie within a single region's boundaries.

The coherence protocol, Protozoa, requires the core-private L1

Listing 1: OpemMP Counter Example

```
/* Threads=2 and ITER = 1000 */
volatile int Item[MAX_THREADS];

/* Parallel Worker */
void worker (int index)
{
 for(int i=0; i<ITER;i++)
   Item[index]++;
}
int main()
{ ....
  /* Thread Creation */
   #pragma omp parallel for
    for (int i=0; i<THREADS; i++)
    {
      worker(i);
    }
}
```

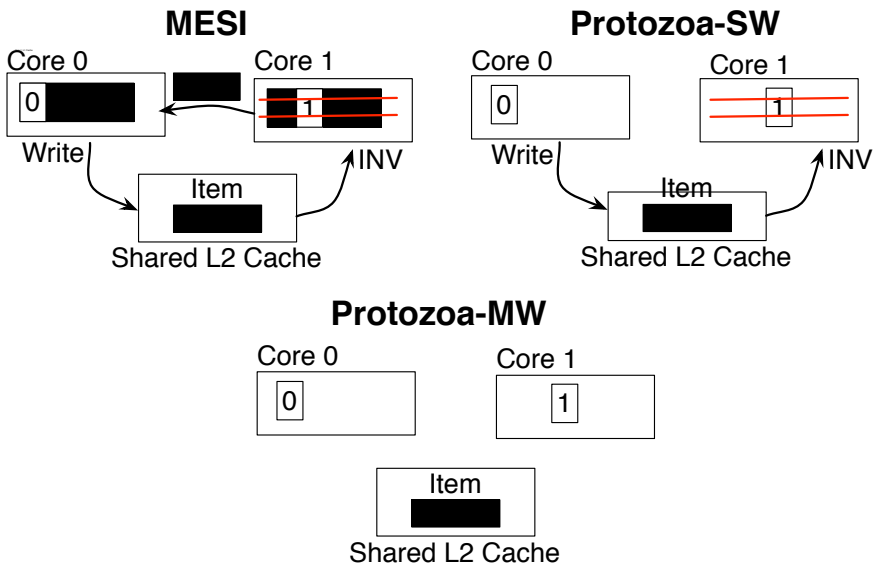
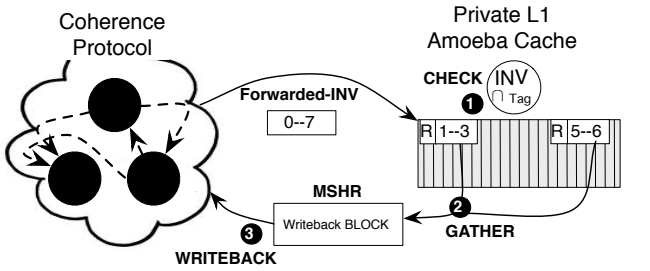


Figure 1: The impact of storage/communication and coherence granularity. MESI: Baseline block granularity protocol; Fixed granularity storage/communication and Fixed granularity Coherence; Protozoa-SW: Adaptive granularity storage/communication, but Fixed coherence granularity. Protozoa-MW: Adaptive granularity storage/communication and coherence granularity.



Example Writebacks: **Check:** Identify overlapping sub-blocks. **Gather:** Move relevant sub-blocks to MSHR. **Writeback:** Processed by protocol as a single coherence operation.

Figure 3: Amoeba-Cache and Coherence Protocol interaction.

Amoeba-Cache to provide support for variable granularity snoop operations (invalidations and writebacks). Since Protozoa decouples the coherence granularity from the storage/communication granularity supported by *Amoeba-Cache*, it gives rise to multi-step snoop operations. We illustrate this using a writeback example (Figure 3). Initially, the cache has two variable granularity dirty blocks from the same `REGION`, R. One block holds words 1–3 and the other holds words 5–6. The coherence protocol requests a writeback of words 0—7 to satisfy a remote write request. On the forwarded writeback request, *Amoeba-Cache* needs to remove both blocks; however, this is a multi-step operation. We keep the logical step of a writeback request separate from the number of blocks that need to be processed within the *Amoeba-Cache*. *Amoeba-Cache* interacts with the coherence protocol using a 3 step process. In **CHECK** (❶), *Amoeba-Cache* identifies the overlapping sub-blocks in the cache. In **GATHER** (❷) the data blocks that are involved in the coherence operation are evicted and moved one-at-a-time to the MSHR entry. The coherence protocol itself is completely oblivious to this multi-step process and is simply blocked from further actions for this `REGION` until the gather step completes. Finally, the protocol processes the **WRITEBACK** (❸), treating the gathered blocks in the MSHR as a single writeback.

3.2 Protozoa-SW

In this section, we describe the *Protozoa-SW* coherence protocol that leverages *Amoeba-Cache* to support adaptive granularity storage/communication, while continuing to use fixed granularity cache coherence. The fixed granularity we adopt for cache coherence is a `REGION` (64 bytes). Note that variable granularity *Amoeba-Block*s requested by the *Amoeba-Cache* never span region boundaries and *Protozoa-SW* invokes coherence operations only on blocks within a region. *Protozoa-SW* supports read sharing at variable granularities between the cores; it can also support variable granularity dirty blocks in the L1 cache. Requests from private Amoeba caches will reflect spatial locality, as well as the read-write sharing granularity in multithreaded applications.

Protozoa-SW supports only a single writer per `REGION` and maintains the Single-writer or Multiple-reader invariant at the `REGION` granularity, i.e., if there exists at least one writer caching any word in a `REGION`, then all sharers of any word in `REGION` need to be invalidated. We describe the protocol with a 2-level hierarchy: each core possesses its own private L1 *Amoeba-Cache* and shares the L2 cache, with coherence implemented at the L2 cache level.

Directory.

Protozoa-SW can operate with a conventional directory. *Protozoa-SW* organizes the directory entries at region granularity (64 bytes), with each directory entry representing the information for all the sub-blocks in the region cached at the various L1s. For instance, if Core0 holds sub-block range <0–3> (where A is the base region address), and Core1 holds sub-block range <4–7>, the directory entry would be represented as Region@(Core0 and Core1), indicating that some word in region A is possibly cached at both processors; it does not represent information about which word is cached by which processor. Since *Protozoa-SW* is a single owner protocol, the directory precisely indicates which processor is caching a sub-block for writing. We leverage the inclusive shared L2's tags to implement an in-cache directory. The directory sharer information is associated with the fixed-granularity L2 data block.

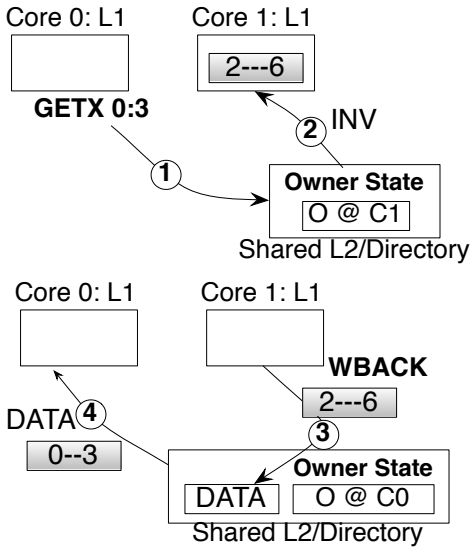
Operation.

Protozoa-SW uses the same stable states as a conventional direc-

tory protocol at the shared L2; we refer the interested reader to Section 3.6 for a discussion on the additional transient states. Table 2 shows the stable states in *Protozoa-SW* at each private L1 and the directory at the shared L2. Protozoa supports 2-hop and 4-hop transactions (when a remote core contains dirty data).

Table 2: Cache states in *Protozoa-SW*

L1 Stable States	
M	Dirty. No other L1 may hold a sub-block.
E	Clean and exclusive.
S	Shared. Other L1 can hold overlapping sub-blocks.
I	Invalid
Shared L2/Directory Stable States	
O	Atleast one word from REGION dirty in one or more L1s.
M	L2 data block dirty, no L1 copies. Need to writeback to memory.
M/S	L2 data block dirty, one or more L1 copies. Need to writeback to memory.
SS	Atleast one word from REGION present in one or more L1s.
I	Invalid



Core-0 issues GETX for words 0–3. Core-1 is an overlapping dirty sharer caching words 2–6. ❶ Requestor sends GETX to directory. ❷ Request is forwarded to Core 1. ❸ Core 1 writes back block including all words whether overlapping or not. ❹ L2 sets new owner as Core-0 and provides DATA 0–3.

Figure 4: Write miss (GETX) handling in *Protozoa-SW*.

We use an example (Figure 4) to illustrate the adaptive storage/-communication granularity of *Protozoa-SW* while performing coherence operations similar to a baseline MESI protocol. Initially, *Core-1* has requested permissions for words 2—6. The shared L2 supplies the data and has marked *Core-1* as the owner in the directory for the entire region. *Core-0* makes a write request (GETX) to the shared L2 for range 0–3. The directory detects that *Core-1* is marked as the owner (Owner state) and forwards a downgrade to *Core-1*. In ❸, *Core-1* evicts the entire block even though only words 2– 3 are required. This helps deal with the coherence request in a single step without having to further segment an existing cache block. If multiple blocks corresponding to the incoming downgrade need to be written back, then the cache employs the steps outlined in Figure 3 to send a single writeback to the directory. ❹ When the DATA arrives at the directory, directory will update the shared L2 with the words in the writeback and forward only the requested words (0–3) to the requestor. Compared to a conventional MESI protocol, *Protozoa-SW* has the same control message sequence including the sharer being invalidated completely. The DATA message sizes are reduced to include only the words requested (or cached, in the case of a writeback) by the core based on the spatial locality and sharing pattern of the application.

3.3 Add-ons to a conventional MESI protocol

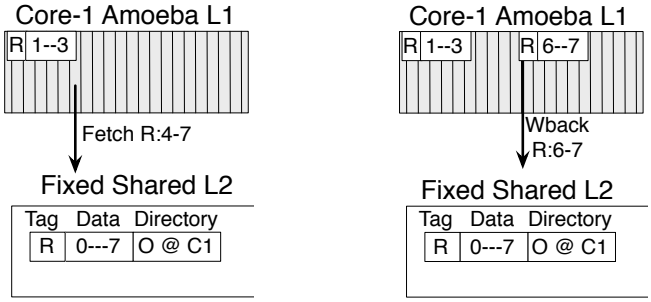
With *Protozoa-SW*, multiple distinct cache blocks from the same region can co-exist in the private L1 cache simultaneously, while the directory tracks sharers at a region granularity. This results in a few extra messages that are forwarded to the L1 cache and/or directory compared to a conventional MESI protocol. We discuss some issues below:

Additional GETXs from Owner.

With conventional MESI protocols, the directory does not expect write misses from the owner of a block, since it is expected that owners do not silently evict lines cached previously. In *Protozoa-SW*, since the directory and coherence granularity are tracked at REGION granularity, it is possible for an owner to issue secondary GETXs. As Figure 5 illustrates, a core holding a sub-block A:1–3 in the REGION could be marked as an O(Owner) at the directory. Now an additional write miss at this private L1 requests words A:4–7. The baseline MESI would blindly forward the request back to the existing owner, resulting in a protocol error. *Protozoa-SW* requires an additional check when a request arrives to see if the request is from the existing owner, and if so, return the requested data.

Multiple Writebacks from Owner.

With conventional MESI protocols, when a writeback is received the sharer is unset at the directory. With variable granularity storage, however, multiple blocks from the same core could be dirty at the L1. When one of the blocks is evicted (Figure 5), it shouldn't unset the sharer at the directory as there could be other sub-blocks from the same REGION in the L1. At the time of a writeback, Protozoa checks the L1 to see if there are any other sub-blocks from the REGION in the cache, and if so, send back a simple WBACK message that does not unset the sharer at the directory. When the final block from the REGION is evicted from the cache (R:0–3 in the example), it sends a WBACK_LAST message, which informs the directory that it can now safely stop tracking the sharer.



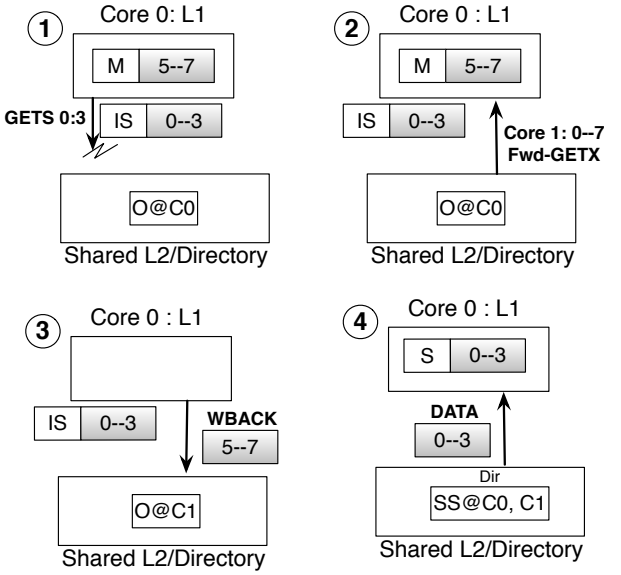
Multiple GETXs **Multiple WBACKs**

Figure 5: Multiple L1 operations to sub-blocks in REGION.

Races between GETX/Ss and INVs/Fwd. GETXs.

We use this race to illustrate how the L1 protocol handles coherence-related operations in the midst of outstanding misses. Note that a similar race occurs in the conventional MESI protocol as well [29, page 152] (due to false sharers), but *Protozoa-SW* does introduce a few specific challenges. Figure 6 illustrates one specific race. In ❶, *Core-0* holds a dirty (M state) block with words 5–7, and issues a read miss (GETS) for words 0—3. In ❷, with the GETS yet to reach the directory, a concurrent write (GETX) appears from

Core-1. Since *Core-0* is the owner at the directory (because of the gray block), the GETX gets forwarded. The challenge is that with a GETS already outstanding for a sub-block in the region, a racing forwarded request has arrived for another sub-block in the same region. The directory activates only one outstanding coherence operation for each `REGION` and in this case has activated the GETX already. In ❸, the sub-block corresponding to the forwarded invalidation is written back to the directory, which supplies *Core-1* with the requisite data 5–7 from the WBACK and 0–4 from the Shared L2. Finally, the GETS is activated and the directory downgrades *Core-1* to a sharer state before supplying the requisite data 0–3.



Core-0 is the owner of 0–7 holding block 5–7 in M. ❶ Core-0 issues an additional read request 0–3. ❷ Request 0–3 by Core-0 hasn't arrived at directory. Directory forwards Core-1's GETX 0–7 request to Core-0. ❸ Blocks in 0–7 are gathered at Core-0. Dirty words 5–7 are written back. Directory is owned by Core-1 after data 0–7 forwarded to core-1. ❹ GETS 0–3 from core-0 is processed at Directory. Directory downgrades to Shared (sending messages to Core-1) and supplies 0–3.

Figure 6: Race between GETS and Fwd. GETX (write miss) in *Protozoa-SW*.

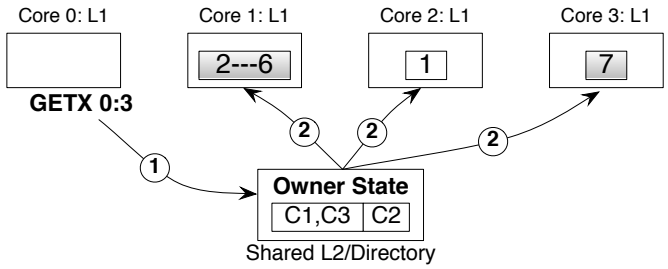
3.4 Protozoa-MW

Protozoa-MW augments *Protozoa-SW* to support both adaptive storage/communication granularity and adaptive coherence granularity. Adaptive granularity blocks in the cache provide an opportunity to differentiate true sharing from false sharing. On incoming coherence requests, the storage can determine whether the incoming range overlaps with any block in the cache (see Figure 3). *Protozoa-MW* takes advantage of this opportunity to support adaptive granularity cache coherence. In comparison, *Protozoa-SW* maintains coherence at a fixed `REGION` granularity and all sharers of a region are invalidated (even non-overlapping sharers). In many cases, *Protozoa-SW* will result in two non-overlapping sub-blocks invalidating each other, resulting in the well-known ping-pong effect (see OpemMP example in Figure 1). *Protozoa-MW* supports adaptive coherence granularity to effectively eliminate the ping-pong effect of data transfer due to false sharing.

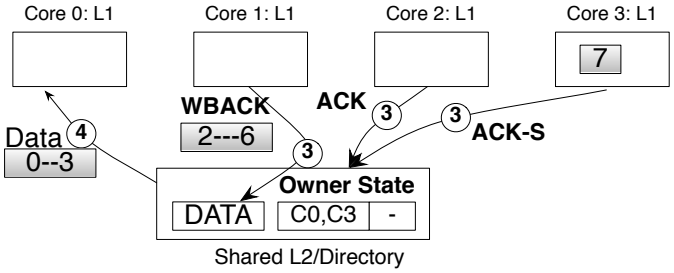
Protozoa-MW requires minimal changes to the protocol itself, requires no additional coherence states (at the directory or the L1), and uses the same in-cache fixed-granularity directory structure as MESI or *Protozoa-SW*, with the exception of tracking readers and writers separately. The key idea that enables *Protozoa-MW* is a **multiple-owner directory**. With MESI or *Protozoa-SW*, only a single writer to a cache block or region can exist at any given time and the writer is tracked precisely at the directory. With *Protozoa-MW*, we relax this constraint. When the directory is in the owner state in *Protozoa-MW*, multiple writable sharer copies and/or read-only sharer copies of non-overlapping sub-blocks could potentially exist. All such reader/writer sharers are tracked by the directory. **When a sub-block in a `REGION` is cached for writing, then no other overlapping reader/writer sub-blocks can exist at any other cache, i.e., we maintain the cache coherence invariant effectively at the word granularity.**

Operation.

We illustrate *Protozoa-MW* operation with an example (Figure 7) that supports write-write sharing of non-overlapping blocks and eliminates false sharing. *Protozoa-MW* uses a similar strategy to also support read-write sharing of non-overlapping blocks. Initially, Core-1 and Core-3 cache dirty non-overlapping sub-blocks. Core-2 caches a non-overlapping read-only copy. The directory has marked Core-1 and Core-3 as the owners and Core-2 as a sharer. In Part 1: Core-0 experiences a write miss (GETX), which is forwarded to the directory, requesting words 0–3. The directory forwards an invalidation or downgrade to all sharers (just as in *Protozoa-SW*).



Part 1: Request Forwarding



Part 2: Response and Directory Update

Core-0 issues GETX for words 0–3. Core-1 is an overlapping dirty sharer. Core-2 is an overlapping read-only sharer. Core-3 is a non-overlapping dirty sharer. ❶ Requestor sends GETX to directory. ❷ Request/invalidate is forwarded to writers/sharer. ❸ Dirty overlapping sharer (Core1) does writeback and invalidate. Clean overlapping sharer (Core 2) invalidates and sends ACK. Non-overlapping dirty sharer (Core 3) continues to be owner and sends ACK-S. ❹ L2 provides DATA for requested range.

Figure 7: Write miss (GETX) handling in *Protozoa-MW*.

In part 2, the cache responses vary based on the sub-block overlap. Since Core-1 contains some of the dirty data (words 2–3) needed by Core-0, it sends a WBACK (writeback) to the shared L2, which patches it into the shared cache block. Since the shared L2 is a fixed

granularity cache consisting of REGION-sized blocks, any WBACK will need to update only a single block. Core-2 as an overlapping read-sharer invalidates itself and sends back an ACK. Unlike in *Protozoa-SW*, Core-3 notices that there is no overlap between the dirty data cached (word 7) and the remote writer (words 0 –3), so it sends an ACK-S to the directory. ACK-S tells the directory that the invalidation is acknowledged, but the core should continue to be marked as a sharer. Finally, the shared L2 supplies the DATA back to Core-0. In final state, Core-0 caches words 0–3 for writing while Core-3 caches word 7 for writing.

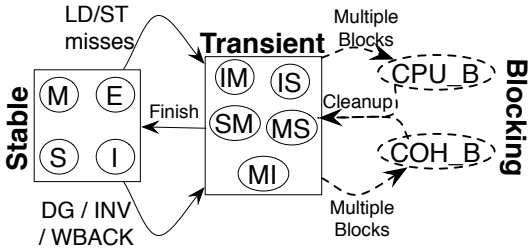


Figure 8: L1 State transitions of Protozoa Coherence.

Directory.

Note that *Protozoa-MW* uses the same directory structure as MESI and *Protozoa-SW*: the sharers are tracked at REGION granularity with the L2 completely oblivious about what sub-blocks each sharer holds. While this keeps the system comparable in space overheads to a conventional MESI, it leads to write misses being unnecessarily forwarded to false sharers, which need to send back ACK-S messages. Figure 7 illustrates this issue. Even though Core-3 is a non-overlapping sharer, the directory is unaware and probes Core-3. Note that this additional traffic only occurs at the time of a miss and once the sub-block is cached (Core 0), no additional messages are needed or generated. *Protozoa-MW* does require additional directory storage to accurately represent readers and writers separately. If no distinction is made between readers and writers in the metadata, a reader would generate unnecessary traffic by being forced to query other readers in addition to the writers.

3.5 *Protozoa-SW+MR*

Protozoa-SW supports the single writer invariant per block region while *Protozoa-MW* supports potentially multiple concurrent writers and readers. It is possible to consider an intermediate protocol realization, *Protozoa-SW+MR*, that supports multiple readers concurrently with one writer as long as they are non-overlapping, but not multiple writers. *Protozoa-SW+MR* provides interesting tradeoffs under different sharing behaviors. *Protozoa-SW+MR* will cause more write misses than *Protozoa-MW* in the presence of concurrent writers to the same block. On the other hand, consider the example in Figure 7. When *Core-0* performs the write, *Protozoa-SW+MR* will revoke Core-3's write permission, ensuring subsequent readers do not need to ping Core-3. In contrast, *Protozoa-MW* will continue to track Core-3 as a remote owner and continue to forward a subsequent requester to Core-3 since the directory does not keep track of what words are cached at Core-3, which increases control messages in some applications (see Section 4).

3.6 Protocol Complexity

Protozoa's complexity is comparable to well understood 4-hop MESI directory protocols. The differences arise in new events and message types as listed in Table 3 (no change to the size of control metadata is required, which is 8 bytes in the base protocol). For example, in Owner (O) state, an additional miss form the owner must be handled as described in Section 3.3. Our MSHR and cache controller entries are similar to MESI since we index them using the fixed REGION granularity; the L1 controller also serializes multiple misses on the same REGION. For *Protozoa-SW*, each directory entry is identical in size to the baseline MESI protocol. Assuming a P-bit vector representation of sharers, *Protozoa-MW* doubles the size of each directory entry in order to separate readers from writers. *Protozoa-SW+MR*, on the other hand, needs only *logP* additional bits to uniquely identify the single writer. Figure 8 shows the abbreviated L1 state transition diagram; add-ons to MESI are highlighted in broken lines.

In general, there are two types of requests: requests from the CPU side, and requests from the coherence side. Since Protozoa decouples storage/communication granularity from the fixed coherence granularity, a coherence operation may need to process multiple blocks in the same cache set. This situation is similar to the Sun Niagara's L2 eviction, which may need to forward invalidations to 4 L1 blocks, since the L1 block size is 16 bytes while the L2 block size is 64 bytes. If there is only one sub-block cached from a requested region, the behavior is similar to the baseline MESI. When multiple sub-blocks need to be processed before the coherence operation or cpu-side operation can be handled, the controller moves into a blocking state until the cache completes its lookup of each block (see Figure 3). This adds two transient states to handle such cases — CPU_B handles misses from the CPU side, while COH_B handles Downgrades and Invalidations from the Coherence side.

Table 3: Additional events and message types in Protozoa

Additional message types	
SW	LAST_PUTX
MW	Nonoverlapping_ACK (ACK-S in Figure 7)
Additional cache controller events	
SW	L1: LocalDG L2: PUTXLast, AdditionalGETS/GETX/UPGRADE
MW	L1: FwdNonlapping L2: L1Nonlapping/NonlappingLAST, L2LocalDG

Correctness Invariants.

We have tested protozoa extensively with the random tester (1 million accesses) and full system runs. Both Protozoa protocols build on the widely distributed MESI_CMP_directory protocol. Correctness for coherence protocols typically means implementing the SWMR invariant, which requires that, for a datum of some granularity (typically a cache block), at any given instant, the datum can be cached for read-write by a single core or cached for read-only operations by zero or more concurrent cores. If we assume that MESI implements the single-writer or multiple-reader (SWMR) invariant, then by extension: i) Protozoa mimics MESI's behavior when only a fixed block size is predicted, i.e., its transitions precisely match MESI's transitions on each event. ii) Like MESI, *Protozoa-SW* implements the SWMR invariant at the region granularity; when a single writer exists for any word to a region (64 bytes), no concurrent readers can exist., and iii) *Protozoa-MW*'s state transitions effectively extends the granularity to a word, implementing the SWMR invariant at a word granularity.

4. Evaluation

We use the SIMICS-based [14] execution-driven simulator with GEMS [17] for modeling the cache and memory system. We also substitute the SIMICS emulator with a faster trace-driven emulator (using Pin [13]) to enable longer runs. The applications compiled using gcc v4.4.3 and the -O3 optimization flag, and are traced on a 2x12-core AMD system. We model the cache controller in detail, including all the transient states and state transitions within GEMS. The parameters for our baseline system are shown in Table 4. All the protocols we evaluate employ the same directory structure. Since our L2 is shared and inclusive, we implement an

in-cache directory that requires 16 bits to precisely track the sharers. We evaluate a two-level protocol with private L1s and the shared L2, the coherence point being the L2. At the L1 level, Protozoa uses *Amoeba-Cache* [12] for storing variable granularity blocks. To determine the granularity of the blocks, we also leverage the PC-predictor discussed in the Amoeba-cache paper [12]. We evaluate a wide range of parallel benchmarks, including 7 scientific workloads from SPLASH2 [34] using the recommended default inputs, 2 commercial workloads (SPECjbb and Apache) [1], 9 workloads from PARSEC [3] using *simlarge*, 2 Java-based workloads (h2 and tradebeans) from DaCapo [4], 7 workloads from Phoenix [22], and an implementation of parallel k-D Tree construction [5].

Table 4: System parameters

Cores: 16-way, 3.0 GHz, In order Amoeba-Cache [12] Private L1: 256 sets, 288B/set, 2 cycles
Shared, Inclusive, Tiled L2 Cache
16 Tiles, 2MB/Tile. 8 way, 14 cycles
Interconnect: 4x4 mesh. 1.5Ghz Clock. Flit size: 16 bytes. Link latency: 2 cycles.
Main Memory : 300 cycles

Table 5: Benchmarks

Suite	Benchmarks
SPLASH2	barnes, cholesky, fft, lu, ocean, radix, water-spatial
PARSEC [3]	blackscholes, bodytrack, canneal, facesim, fluidanimate, x264, raytrace, swaptions, streamcluster
Phoenix [30]	histogram, kmeans, linear-regression, matrix-multiply, reverse-index, string-match, word-count
Commercial	apache, spec-jbb
DaCapo [4]	h2, tradebeans
Denovo [5]	parkd

We compare the following coherence designs: i) **MESI**, a conventional fixed granularity 4-hop directory-based protocol. The results presented here assume a 64 byte granularity. ii) *Protozoa-SW*, which supports adaptive granularity storage/communication, but fixes the coherence granularity, supporting only a single writer at a time per cache line. iii) *Protozoa-MW*, which supports adaptive granularity storage/communication and adaptive coherence granularity, allowing multiple concurrent readers and disjoint writers per cache line, and iv) *Protozoa-SW+MR*, which allows concurrent readers with a single disjoint writer per cache line. *Protozoa-SW*, *Protozoa-SW+MR*, and *Protozoa-MW* use the same `REGION` size (64 bytes), which defines the granularity at which the directory tracks data and the maximum granularity of a data block. In *Protozoa-SW*, it also represents the coherence granularity.

4.1 Reduction of Traffic and Cache Misses

Protozoa coherence protocols can adapt coherence granularity to parallel application sharing behavior. Protozoa-SW reduces on-chip traffic by 26% relative to MESI, while Protozoa-SW+MR and Protozoa-MW reduce traffic by 34% and 37%, respectively, relative to MESI. Protozoa-MW is able to eliminate false sharing, reducing miss rate by up to 99% (linear regression).

Figure 9 shows a breakdown of the total amount of data sent or received in messages at an L1: **Unused DATA** are the words within cache blocks brought in to private L1 caches, but not touched before the block is evicted or invalidated. **Used DATA**: is the converse of unused data. **Control:** refers to all metadata communication, including control messages such as invalidations and acknowledgments, and message and data identifiers. In MESI, due to the use of a fixed coherence and communication/storage granularity, Unused DATA accounts for a significant portion of the overall traffic (34%), more than all control messages combined(22%). This percentage varies greatly across applications (1% for Mat-mul to 65% for Canneal). With support for adaptive granularity storage/-communication, *Protozoa-SW* eliminates 81% of Unused DATA and on average reduces traffic by 26%. Note that this improvement is more noticeable than even if all control messages were eliminated from MESI, i.e., a move to incoherent software-based systems with fixed granularity has a bounded improvement scope compared to a conventional MESI protocol. In applications with predictably poor spatial locality, *Protozoa-SW* almost completely eliminates Unused DATA ($\leq$ 5% unused data in blacscholes, bodytrack, canneal, water). Only apache, jbb, and tradebeans, with unpredictable access patterns, have noticeable unused data (15%). Two applications, histogram and swaptions, see some rise in relative traffic; both have very low cache miss rates and raw traffic as shown in Figure 13. With swaptions, most of the data is read-only and *Protozoa-SW* experiences additional misses due to smaller granularity reads, increasing both control and data traffic. Used DATA also increases in some applications (e.g., h2, histogram) compared to MESI. H2 and histogram suffer misses primarily due to false sharing. In such cases, *Protozoa-SW*, which uses a spatial locality predictor to determine the block granularity, may increase the # of misses by underfetching, resulting in more data blocks transferred. Higher data retention time implies more data classified as useful data, especially when read and written sub-blocks are evicted together, resulting in higher counts for useful data that is written back.

Protozoa-SW+MR and *Protozoa-MW* further reduce data transferred by allowing fine-grain read-write sharing. *Protozoa-SW+MR* shows a similar traffic reduction in most applications. In applications that exhibit prominent false sharing with multiple readers to disjoint words in a cache line (but only a single writer at a time), both *Protozoa-MW* and *Protozoa-SW+MR* reduce data transferred compared to *Protozoa-SW* by eliminating secondary misses. Compared to *Protozoa-SW*, they reduces traffic by 29% for h2, 80% for histogram, and 32% for string-match. One application that shows the benefits of *Protozoa-MW* over *Protozoa-SW+MR* is linear regression, due to the presence of fine-grain write sharing in the application. *Protozoa-SW+MR* is able to reduce traffic by 58% relative to *Protozoa-SW*, while *Protozoa-MW* shows a 99% traffic reduction. For applications such as linear regression with small working sets, once the cache is warmed up and the disjoint fine-grain data blocks are cached for read-write access, the application experiences no further misses (as illustrated in Figure 1). We can see the cascaded benefits of supporting multiple concurrent readers with a single writer under *Protozoa-SW+MR*, and additionally, multiple writers under *Protozoa-MW* in applications such as barnes and streamcluster, which exhibit both behaviors. In a few cases (e.g., X264), *Protozoa-SW+MR* has a larger traffic reduction because earlier evictions actually help the streaming behavior. On average, *Protozoa-MW* reduces data traffic by 42% compared to MESI, 15% compared to *Protozoa-SW*, and 5% compared to *Protozoa-SW+MR*. *Protozoa-MW* saves 37% of total traffic compared to MESI.

Protozoa-MW does not drastically reduce control traffic. Figure 10 shows the breakdown of the control traffic in Figure 9. On average, control traffic in *Protozoa-SW* is 90% of MESI, while for *Protozoa-SW+MR* and *Protozoa-MW*, the corresponding statistic is 86% and 82%, respectively. Most applications show lower control traffic under *Protozoa-SW*, with a few exceptions such as h2, linear-regression, raytrace, and swaption. Due to its lower miss rate, *Protozoa-SW* reduced the data request traffic compared to MESI. The remaining traffic categories (coherence traffic) are comparable

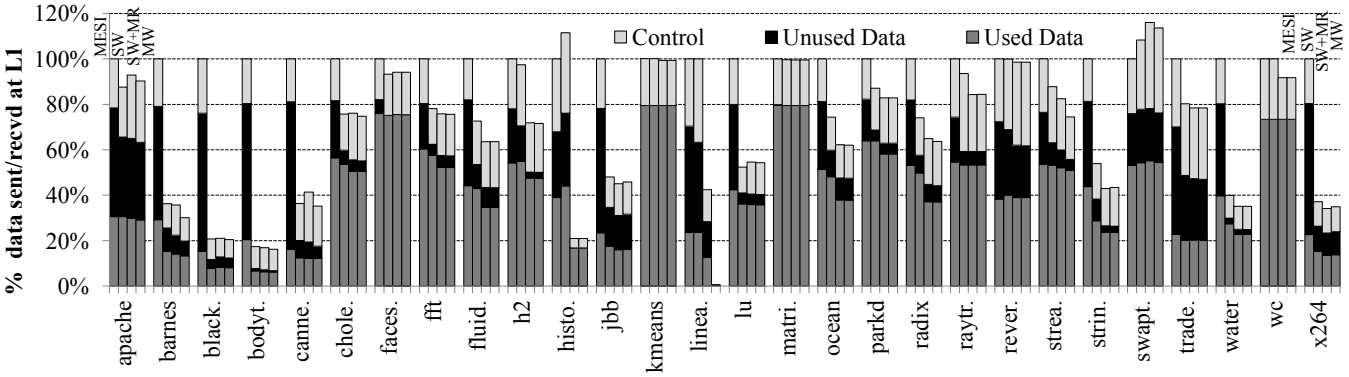


Figure 9: Breakdown of messages received/sent at the L1 (measured in bytes) by information type normalized to the total byte count for the MESI protocol. Four bars per application from left to right: MESI, Protozoa-SW, Protozoa-SW+MR, Protozoa-MW.

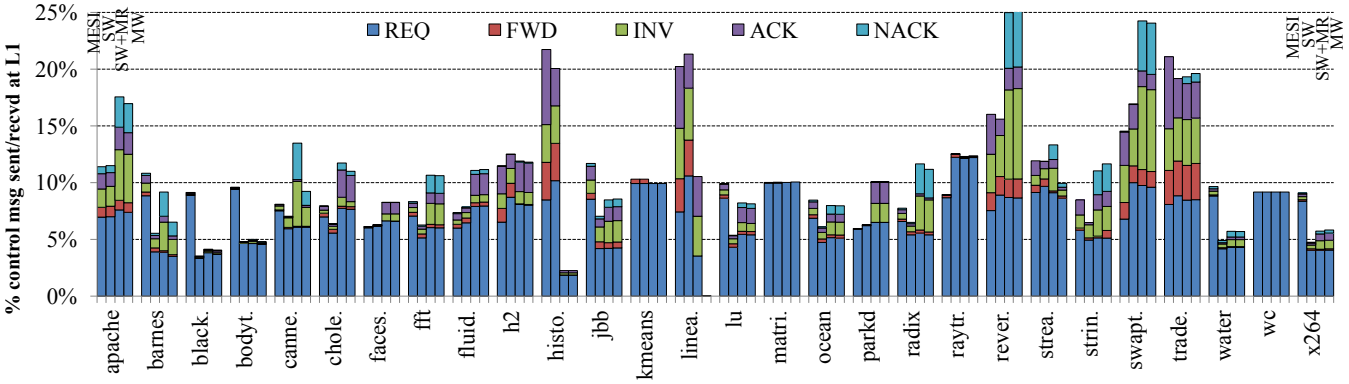


Figure 10: Breakdown of control messages (measured in bytes) sent/received at L1 by type, normalized to total data sent/received at L1 for MESI. Four bars per application from left to right: MESI, Protozoa-SW, Protozoa-SW+MR, and Protozoa-MW.

in *Protozoa-SW* and MESI. For applications such as histogram and linear regression, *Protozoa-MW* reduces directory requests significantly by eliminating eviction of blocks by non-conflicting writes. However, for some applications, the multiple owners and sharers result in new write requests needing to query every owner and sharer. Applications such as apache, rev-index, and radix all exhibit many more invalidation messages (which does not translate to more data traffic since disjoint data are not invalidated). Since there is little overlap in cached data, most of these invalidations will result in a NACK rather than an actual invalidation. Rev_index experiences a 4% to 8% increase in invalidation messages, while more than 65% of these invalidations trigger a NACK response. Even with these extra control messages, rev-index still experiences a decrease in overall traffic because of fewer misses. In barnes and canneal, *Protozoa-SW+MR* requires extra control messages compared to *Protozoa-MW*. Extra write misses are incurred because the protocol supports only one concurrent writer. Downgraded non-overlapping writers also remain as sharers, potentially adding to the number of invalidation messages at the next write miss.

Figure 11 visualizes the sharing behavior of blocks in the Owned state in *Protozoa-MW*. *Protozoa-MW* relies on recording multiple owners as well as sharers in the Owned state. However, too many sharers will increase invalidation traffic as discussed above. There are three applications without directory lookups in the Owned state. Matrix-multiply and wordcount are embarrassing parallel applications. Since linear_regression has a small working set, once the cache is warmed up, the elimination of false sharing by *Protozoa-MW* results in no additional misses. For raytrace, most of the directory entries are owned by a single writer without any fine-grain read-/write sharing (single-producer, single-consumer sharing pattern). In contrast, for string-match, more than 90% of the lookups in the Owned state find more than 1 owners, exhibiting extreme fine-grain sharing.

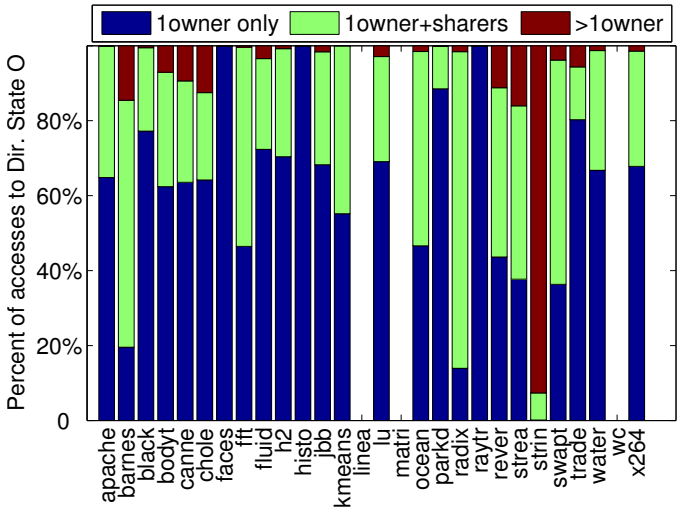


Figure 11: Percentage of accesses to blocks in different sharer count states. Number of owners and sharers is recorded every time a directory entry in Owned state is accessed.

Protozoa-SW only transfers useful data, freeing space in the L1 for other useful data. Miss rates are reduced by 19% on average for Protozoa-SW relative to MESI. Protozoa-MW and Protozoa-

SW+MR reduce miss rates by 36% on average relative to MESI by further eliminating unnecessary evictions caused by false sharing.

Figure 12 shows the block size distribution for each application using *Protozoa-MW* (distributions for *Protozoa-SW* and *Protozoa-SW+MR* are similar). Blackscholes, bodytrack, and canneal exhibit low spatial locality, bringing in a majority of 1–2 word blocks. On the other hand, applications such as linear regression, matrix multiply, and kmeans bring in a majority of 8-word blocks.

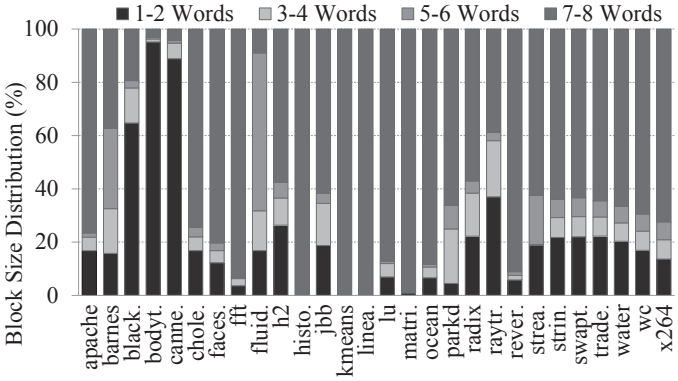


Figure 12: Distribution of cache block granularities in L1 caches (Protozoa-MW).

With smaller block sizes, each set in the L1 cache could cache more blocks in the space saved from unused words in a larger fixed granularity block. *Protozoa-SW* benefits the applications with high miss rates most: in 10 applications with MPKI ≥ 6 in MESI, *Protozoa-SW* reduces the miss rate by 35% on average relative to MESI. *Protozoa-SW+MR* and *Protozoa-MW* do even better, reducing miss rate by 60% on average relative to MESI by reducing traffic due to false sharing. The effect of *Protozoa-MW* can be seen clearly on 2 false sharing applications. In histogram and linear-regression, miss rates are reduced by 71% and 99%, respectively, while *Protozoa-SW* is unable to eliminate the misses.

4.2 Performance and Energy

Protozoa Coherence improves overall performance by 4%, and reduces on-chip network energy by 49%.

Figure 14 shows execution time relative to MESI for the Protozoa coherence protocols. On average, the impact on execution time is relatively small at 4%. The Protozoa variants do show a reduction in execution time by 10%—20% on barnes, blackscholes, and bodytrack, by significantly improving the miss rates (Figure 13). *Protozoa-MW* and *Protozoa-SW+MR* reduce execution time relative to MESI for histogram and streamclusters by 5%—8%, due to fewer misses. *Protozoa-SW* increases execution time by 17% for linear regression, while the speedup for *Protozoa-MW* is dramatic at 2.2X. This application has a small working set but suffers from significant false sharing, which is the dominant effect on performance. *Protozoa-MW* is also able to reduce execution time by 36% relative to *Protozoa-SW+MR* by allowing fine-grain write sharing. Due to apache's irregular sharing behavior resulting in the *Amoeba-Cache* predictor's inability to perfectly match its spatial locality, *Protozoa-MW* shows a 7% increase in execution time for apache.

Protozoa coherence greatly reduces dynamic power consumption in the interconnect due to the smaller number of messages. Figure 15 provides a relative measure of dynamic energy consumption in the on-chip interconnect during an application's execution by comparing traffic in terms of flits transmitted across all network hops. On average, *Protozoa-SW* eliminates 33%, *Protozoa-SW+MR* eliminates, 38%, *Protozoa-MW* eliminates 49% of the flit-hops across all applications.

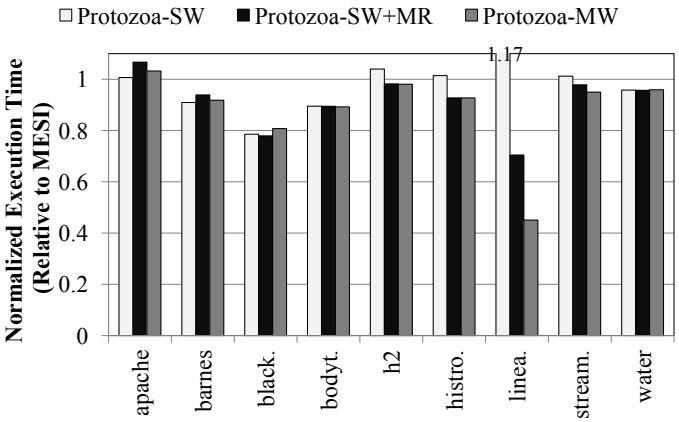


Figure 14: Execution time. Baseline: MESI. Only applications with > 3% change in performance are shown.

5. Related Work

Sector caches [10, 28], line distillation [21], and most recently, Amoeba cache [12], all aim at improving cache utilization and/or data traffic by only bringing in or keeping used data in the cache at the time of a miss/eviction. These designs, while not focused on coherence, provide the opportunity for reduced coherence traffic using variable granularity coherence.

Excess coherence traffic due to false sharing is a widely studied problem. Several coherence designs have been proposed to maintain coherence at a *true* sharing granularity in order to avoid unnecessary coherence traffic. Dubnicki and LeBlanc [7] propose to adjust block size by splitting and merging cache blocks based on reference behavior. The proposal is based on a fixed-granularity cache organization, focusing on coherence traffic rather than cache utilization, and requires coherence metadata overhead per word not only for reference bits but also for the split/merge counters. Kadiyala and Bhuyan [9] propose a protocol to maintain coherence at a subblock level, where the size of the subblock is dynamically determined by the write span of an individual processor. Coherence metadata overhead is fixed at design time and is linear in the maximum number of allowed subblocks per cache block. Minerva [24] is also designed to provide coherence at a subblock granularity. Sharing attributes are associated with each individual word-sized sector within a cache block. Coherence metadata costs are therefore linear in the number of words. A fetch is performed at a granularity of a sub-block, which can be anywhere from 1 word to the maximum block size, based on the block's (and thread's) access patterns. Coherence relies on a broadcast medium to invalidate individual words and to allow individual words within the sub-block to be fetched from potentially multiple locations (other caches or main memory). While these approaches use fine-grain blocks selectively based on false sharing patterns, Protozoa uses fine-grain blocks both to eliminate false sharing and to match data-specific spatial locality, without the need for coherence metadata at the fine-grain level. Denovo [5] enforces coherence at word granularity and uses self invalidation along with the data-race-free application semantics to eliminate invalidation traffic and directory storage requirements. While data is tracked at word granularity, communication is accomplished in bulk for multiple words. DeNovo, however, requires programmers to adhere to a disciplined programming model which requires rewriting of applications.

Variable granularity coherence has also been explored in the context of software distributed shared memory. Shasta [26, 27] allows

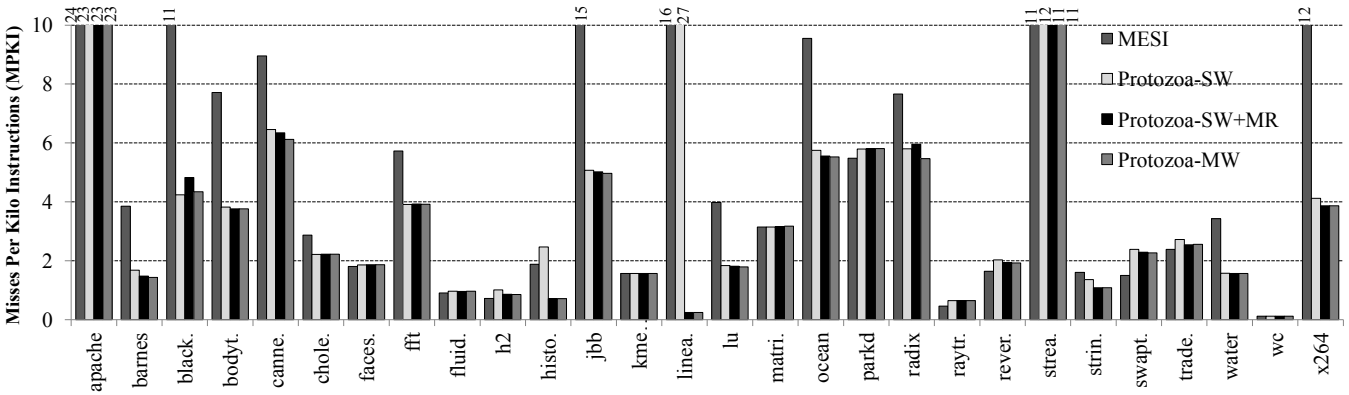


Figure 13: Miss rate (in MPKI).

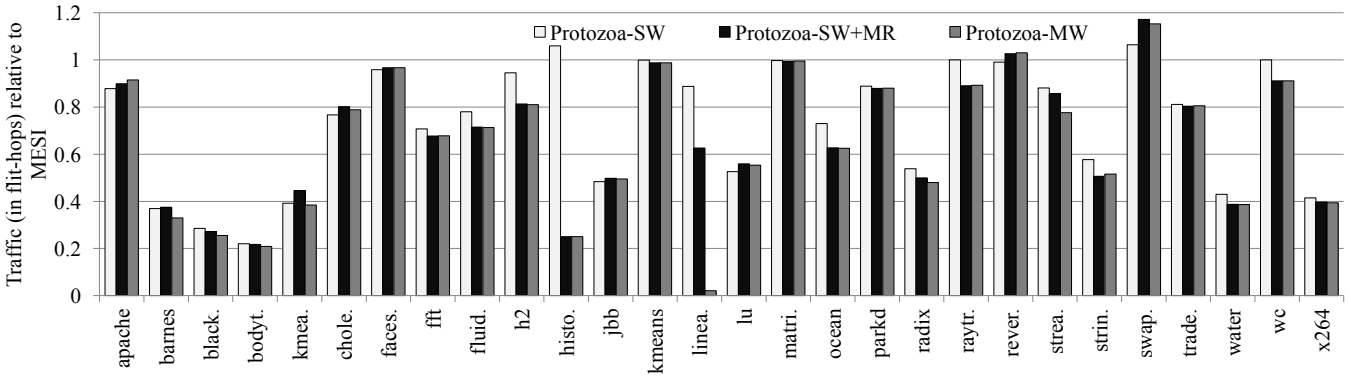


Figure 15: Relative measure of dynamic interconnect energy: traffic in terms of number of flit-hops normalized to MESI.

the coherence granularity to vary across different data within an application. However, programmers must specify the desired granularity. Adaptive Granularity [18] proposes a system that defaults to a fixed large granularity (e.g., page), using a buddy-style system to split the granularity successively in half when ownership changes, merging when adjacent ownership is the same. Granularity is thereby decreased when false sharing is detected, and increased when more spatial locality is expected. However, their proposal does not explore granularities smaller than a hardware-supported fixed cache line size.

6. Design Issues

3-hop vs 4-hop.

Our baseline in this paper is a 4-hop MESI protocol, from which the Protozoa protocols are derived. In the example shown in Figure 4, we could enable direct forwarding between the CPUs as opposed to transferring data through the L2. Both 3-hop MESI and Protozoa would need to deal with the case when a fwd request cannot complete at the owner. In MESI, this would occur when a block in E (Exclusive) is silently dropped (as, for example, in the SGI Origin-3000). In Protozoa, it could occur because the fwd request does not overlap, or partially overlap, with the owner (due to false positives at the directory). One option is to fall back to 4-hop for such corner cases.

Coherence directory.

In this paper, we implemented an in-cache directory in which the sharer vector is collocated with the shared cache blocks. Many current multicores, however, employ shadow tags that duplicate the L1 tag array to save storage. Implementing shadow tags for Protozoa is not straightforward since the number of amoeba blocks (and regions) at the L1 varies dynamically based on application spatial locality. A promising alternative is the use of bloom filter-based [35, 37] coherence directories that can summarize the blocks in the cache in a fixed space. The bloom filter can accommodate the variable number of cache blocks without significant tuning.

Non-Inclusive Shared Cache.

The Protozoa protocols we described in this paper utilize the inclusive shared cache to simplify certain corner cases. For instance, consider the example in Figure 4. The owner L1 supplies part of the block (2-3) required by the requestor and the inclusive L2 aware of the remote owner *Core-1*'s response picks up the slack and supplies the remaining words (0–1). If the shared L2 were non-inclusive, it may not have the missing words (0–1) and would need to request them from the lower level and combine them with the block obtained from *Core-1* to construct the final response. This situation would not arise in a conventional protocol that uses a fixed granularity for both the storage and coherence; if a remote L1 has a cached copy, it will be able to supply the entire block. In general, while Protozoa may need to assemble the data from multiple sources (remote L1, lower level cache), a conventional protocol can always find a single source capable of supplying the entire cache block.

7. Summary

Recently, many researchers have advocated the elimination of hardware cache coherence from future many-core processor [5,11]. These works cite poor coherence scalability based on traffic requirements, poor scaling of metadata storage, and complexity issues. In this paper, we show that the traffic overheads of cache coherence can be streamlined to reflect true application sharing and utilization. We propose Protozoa, a family of coherence protocols, that adapts both storage/communication granularity and coherence granularity to dra-

matically improve overall memory system efficiency. Protozoa exploits adaptive block sizing to minimize data movement and coherence traffic (37% reduction in traffic at the L1 compared to MESI) and enables fine-granularity reader-writer sharing, completely eliminating false sharing. Protozoa builds on a conventional MESI directory protocol and re-uses the conventional fixed-granularity in-cache coherence directory metadata structure.

8. References

[1] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, M. D. Hill, D. A. Wood, and D. J. Sorin. Simulating a $2m commercial server on a $2k pc. *Computer*, 36(2):50–57, 2003.

[2] D. Albonesi, A. Kodi, and V. Stojanovic. NSF Workshop on Emerging Technologies for Interconnects (WETI), 2012.

[3] C. Bienia. Benchmarking Modern Multiprocessors. In *Ph.D. Thesis. Princeton University*, 2011.

[4] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. of the 21st OOPSLA*, 2006.

[5] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *Proc. of the 20th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011.

[6] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. In *IEEE Micro*. IEEE Computer Society Press, 2007.

[7] C. Dubnicki and T. J. Leblanc. Adjustable Block Size Coherent Caches. In *Proc. of the 19th Annual Intl. Symp. on Computer Architecture (ISCA)*, 1992.

[8] A. González, C. Aliagas, and M. Valero. A data cache with multiple caching strategies tuned to different types of locality. In *Proc. of the ACM Intl. Conf. on Supercomputing*, 1995.

[9] M. Kadiyala and L. N. Bhuyan. A dynamic cache sub-block design to reduce false sharing. In *Proc. of the 1995 Intl. Conf. on Computer Design: VLSI in Computers and Processors*, 1995.

[10] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd˙Power7: IBM's Next-Generation Server Processor. In *IEEE Micro Journal*, 2010.

[11] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: a hybrid memory model for accelerators. In *Proc. of the 37th Intl. Symp. on Computer Architecture (ISCA)*, 2010.

[12] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon. Amoeba Cache : Adaptive Blocks for Eliminating Waste in the Memory Hierarchy. In *Proc. of the 45th Intl. Symp. on Microarchitecture (MICRO)*, 2012.

[13] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. of the 2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2005.

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

[15] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, pages 78–89, 2012.

[16] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: decoupling performance and correctness. In *Proc. of the 30th Intl. Symp. on Computer Architecture (ISCA)*. 2003.

[17] 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. In *ACM SIGARCH Computer Architecture News*, Sept. 2005.

[18] D. Park, R. H. Saavedra, and S. Moon. Adaptive Granularity: Transparent Integration of Fine- and Coarse-Grain Communication. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques (PACT)*, 1996.

[19] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian. SWEL: hardware cache coherence protocols to map shared data onto shared caches. In *19th Intl. Conf. on Parallel Architecture and Compilation Techniques (PACT)*, 2010.

[20] P. Pujara and A. Aggarwal. Increasing the Cache Efficiency by Eliminating Noise. In *Proc. of the 12th Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2006.

[21] M. K. Qureshi, M. A. Suleman, and Y. N. Patt. Line Distillation: Increasing Cache Capacity by Filtering Unused Words in Cache Lines. In *Proc. of the 13th Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2007.

[22] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proc. of the 2007 IEEE 13th Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2007.

[23] J. B. Rothman and A. J. Smith. The pool of subsectors cache design. In *Proc. of the 13th ACM Intl. Conf. on Supercomputing*, 1999.

[24] J. B. Rothman and A. J. Smith. Minerva: An Adaptive Subblock Coherence Protocol for Improved SMP Performance. In *Proc. of the 4th Intl. Symp. on High Performance Computing*, 2002.

[25] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson. Programming model for a heterogeneous x86 platform. In *Proc. of the 2009 Conf. on Programming Language Design and Implementation (PLDI)*, 2009.

[26] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain software distributed shared memory on smp clusters. In *Proc. of the 4th Intl. Symp. on High-Performance Computer Architecture (HPCA)*, pages 125–136, Feb. 1998.

[27] D. J. Scales, K. Gharachorloo, and C. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proc. of the 7th Symp. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 174–185, Oct. 1996.

[28] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost. In *Proc. of the 21st Intl. Symp. on Computer Architecture (ISCA)*, 1994.

[29] D. J. Sorin, M. D. Hill, and D. A. Wood. A Primer on Memory Consistency and Cache Coherence. In *Synthesis Lectures in Computer Architecture, Morgan Claypool Publishers*, 2011.

[30] J. Talbot, R. M. Yoo, and C. Kozyrakis. Phoenix++: modular mapreduce for shared-memory systems. In *Proc. of the second international workshop on MapReduce and its applications*, 2011.

[31] E. Totoni, B. Behzad, S. Ghike, and J. Torrellas. Comparing the power and performance of Intel's SCC to state-of-the-art CPUs and GPUs. In *IEEE Intl. Symposium on Performance Analysis of Systems & Software (ISPASS)*, 2012.

[32] D. Vantrease, M. Lipasti, and N. Binkert. Atomic Coherence: Leveraging Nanophotonics to Build Race-Free Cache Coherence Protocols. In *Proc. of the 17th Intl. Symp. on High Performance Computer Architecture (HPCA)*, 2011.

[33] A. V. Veidenbaum, W. Tang, R. Gupta, A. Nicolau, and X. Ji. Adapting cache line size to application behavior. In *Proc. of the 13th ACM Intl. Conf. on Supercomputing (ICS)*. 1999.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proc. of the 22nd annual Intl. Symp. on Computer architecture (ISCA)*, 1995.

[35] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A tagless coherence directory. In *Proc. of the 42nd Intl. Symp. on Microarchitecture (MICRO)*, 2009.

[36] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE: Sharing Pattern-based Directory Coherence for Multicore Scalability. In *Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2010.

[37] H. Zhao, A. Shriraman, S. Dwarkadas, and V. Srinivasan. SPATL: Honey, I Shrunk the Coherence Directory. In *Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011.

[38] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *Proc. of the 6th ACM Symp. on Principles and Practice of Parallel Programming (PPoPP)*, June 1997.