VIPS: Simple Directory-Less Broadcast-Less Cache Coherence Protocol

Alberto Ros and Stefanos Kaxiras
Department of Information Technology
Uppsala University, Sweden
{alberto.ros,stefanos.kaxiras}@it.uu.se

November 29, 2011

Abstract

Coherence in multicores introduces complexity and overhead (directory, state bits) in exchange for local caching, while being "invisible" to the memory consistency model. In this paper we show that a much simpler (directory-less/broadcast-less) multicore coherence provides almost the same performance without the complexity and overhead of a directory protocol. Motivated by recent efforts to simplify coherence for disciplined parallelism, we propose a hardware approach that does not require any application guidance. The cornerstone of our approach is a run-time, application-transparent, division of data into private and shared at a page-level granularity. This allows us to implement a dynamic write-policy (write-back for private, write-through for shared), simplifying the protocol to just two stable states. Self-invalidation of the shared data at synchronization points allows us to remove the directory (and invalidations) completely, with just a data-race-free guarantee (at the write-through granularity) from software. Allowing multiple simultaneous writers and merging their writes, relaxes the DRF guarantee to a word granularity and optimizes traffic. This leads to our main result: a virtually costless coherence that uses the same simple protocol for both shared, DRF data and private data (differentiating only in the timing of when to put data back in the last-level cache) while at the same time approaching the performance (within 3%) of a complex directory protocol.

1 INTRODUCTION

"For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it. ..." [30]

Among several definitions of Cache Coherence (CC), Sorin, Hill, and Wood propose and cite the above for its insight. To satisfy such a definition, coherence protocols react immediately to writes, and invalidate all cached read copies. This is a source of significant complexity. It requires a directory to constantly track readers and writers and to send invalidations or global broadcasts and snoops. It necessitates additional protocol states for performance (e.g., Exclusive, Owned), and causes an explosion in the number of transient states required to cover every possible race that may arise. For example, the GEMS [23] implementation of the MESI directory protocol –a direct descendant of the SUNfire coherence protocol– requires no less than 30 states.

Verification of such protocols is difficult and in many cases incomplete [1]. Complexity also translates into cost. Storage is needed for cache-line state, the directory (or dual-ported/duplicate tags for snooping), and the logic required by complex cache and directory controllers. Significant effort has been expended to reduce these costs, especially the storage cost [2, 8, 9, 26], but also verification cost [10, 31]. In terms of performance and power, complex protocols are characterized by a large number of broadcasts and snoops. Here too, significant effort has been expended to reduce or filter coherence traffic [17, 24, 33] with the intent of making complex protocols more power- or performance-efficient.

We take an alternate approach that eliminates the need for directories, invalidations, broadcasts and snoops. Indeed, this approach eliminates the need for almost all coherence state (besides the rudimentary valid/invalid and clean/dirty states). Our approach exploits a typical multicore cache hierarchy organization with private L1(/L2) caches and a shared Last-Level-Cache (LLC). Our motivation is to improve verifiability, by having very simple coherence, and improve power-efficiency, by considerably reducing the hardware cost (area), with negligible impact on performance and traffic.

Our targets include general-purpose multicores (few fat cores), manycores (many thin cores) or GP-GPUs with coherent caches, and shared-address space heterogeneous architectures with a multicore coupled to a manycore. Heterogeneous targets are of special interest in light of recent efforts by Microsoft to establish a shared memory model for all cores on a chip regardless of their designation as GP or special [29]. In particular, in the manycore cases, a simple and efficient implementation of coherence is of great importance

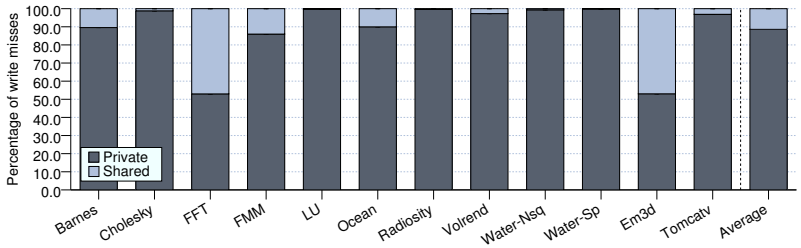


Figure 1: Percentage of write misses in a write-through protocol for both private and shared data.

to match the simplicity of the many thin cores. Furthermore, some manycore programming models (e.g., CUDA [21], CELL [16]) exercise explicit control of the ownership of shared data. This bodes very well for our proposal. In the interest of brevity, we will not make any further distinction between multicores and manycores but we will use the term *multicore* to describe both. The main focus of our work is the multicore cache hierarchy. Thus, the scope of this paper does not extend to inter-chip implementations which we discuss only briefly in Section 6.1. The well-known and well-understood coherence schemes employed widely today have been developed for multi-chip SMPs or distributed shared memory machines where the trade-offs are markedly different from a multicore cache hierarchy.

Our proposal is a simplification of multicore coherence to the bare necessities, but without sacrificing performance or power compared to more sophisticated versions. We achieve this in two steps:

1. **Simplifying the coherence protocol.** Significant complexity in current protocols comes from strategies for efficient execution of sequential applications (e.g., the E state in MESI) or optimizations to avoid memory (e.g., the O state in MOESI) ignoring the LLC between the cores and memory. A simple Write-Through policy to the LLC would obviate almost all coherence states (even the M state) but it is not acceptable as it sacrifices performance. The observation, however, that drives our approach is that *most write misses in a write-though protocol actually come from private blocks*, as Figure 1 shows. We propose a *Dynamic Write-Policy* for L1 caches that selects between Write-Back and Write-Through to the LLC depending on whether data are private or shared. For the private data there is no coherence and write-back is used for performance. For the shared data, a write-through policy simplifies the protocol to just two cache states (Valid/Invalid), eliminates the need to track writers at the (LLC) directory, and eliminates read-indirection through the directory (since the correct data are always found in the LLC). The distinction between private and shared data is determined dynamically at a page granularity utilizing the page table and the TLBs [14, 11] (Section 3.1).

2. **Eliminating the directory (or broadcasts/snoops).** Further, we *selectively flush* all shared data from the L1 caches on synchronization points. This is inspired by self-invalidation and read-only tear-off copies [17, 19] but we include copies that have been written too as in [10]. This step eliminates the need to track readers for invalidation, therefore obviating the need for a directory or for broadcasts and snoops. Finally, we perform write-backs by transferring only what is modified (*diffs*) in a cache line. Consequently, we allow multiple simultaneous writers per cache line and merge their writes correctly, as long as these writes constitute false sharing on separate words in a cache line and not a data race on some word (Section 3.2).

While write-through protocols and self-invalidation are not new, we combine them, enhance them with a dynamic division of data to shared and private at a page granularity, allow writable self-invalidating copies, and merge multiple simultaneous writers in the LLC. This leads to the main result we report in this paper: a minimal, virtually costless (directory-less & broadcast-less) coherence scheme that: i) uses the same very simple protocol for shared, Data-Race-Free, data and for private data, differentiating only in the timing of when to put them back in the LLC, ii) approaches the performance of a complex directory protocol, and iii) does not rely on any application involvement or guidance. Our work bears resemblance to the work of Choi *et al.* which inspired us to pursue a hardware approach as opposed to their application-driven approach [10]. We present the resulting differences in Section 7.

Another advantage of our approach is that it is *interconnect-agnostic*, meaning that our coherence protocol is exactly the same whether implemented over a bus, a crossbar, or a packet-based, point-to-point, NoC. This leads to seamless scaling from low-end to high-end parts or free intermixing of buses and NoCs on the same chip, e.g., in an heterogeneous multicore/manycore chip. In the limited space of this paper we cannot evaluate all the network options, so we limit our discussion to the most general and challenging case of an unordered, packet-based NoC.

There are two implications of our approach. First, the protocol resulting from the second step cannot support Sequential Consistency (SC) for data races. This is because without a directory or broadcasts, a core writing a memory location cannot invalidate any other cores that may be reading this location. This violates the definition of coherence but it is actually an *acceptable behavior for a weak consistency memory model* [30] (Section 3.3). Thus, our protocol is *incoherent* for data races but satisfies the definition of coherence for the important class of Data-Race-Free (DRF) programs. Similarly to SC for DRF [3], our approach provides

Coherency for DRF. It follows, that SC can be supported just for DRF programs. The second implication (a consequence of the first) is that, synchronization instructions (such as T&S or Compare&Swap) which inherently rely on data races, require slight modifications for a correct implementation. These modifications carry only a small impact on synchronization performance. Further, synchronization needs to be made explicit to the cache controllers in order to flush the shared data.

Data races are the culprits of many problems in parallel software. The benefits of data-race-free operation are well argued by Choi *et al.* [10] in conjunction with programming model guarantees –in their case Deterministic Parallel Java. We are inspired by their approach and fully subscribe to their views, but we do not tie our approach to any particular programming model or language, requiring instead just a simple data-race-free contract between the hardware and the software. In our framework data races, except those due to synchronization primitives, are software bugs.

Results. Our approach leads to a very simple coherence protocol that requires no directory, no state bits in the caches (other than the standard Valid/Invalid and Dirty/Clean bits), no broadcasts/snoops, nor invalidations, but actually performs as well as a directory protocol (within 3%). It minimizes control message traffic since there is no invalidation and minimizes data traffic by sending only cache-line diffs to the LLC from multiple simultaneous writers. Diffs are correctly merged in the cache-lines –in the absence of data-races– thereby solving the false-sharing problem. Our evaluation focus is on a tiled manycore architecture, reflecting our conviction that very simple coherence is especially appealing in this case.

2 BACKGROUND

2.1 Write-Through Caches and Coherence

A write-through policy for L1 caches has the potential to greatly simplify the coherence protocol [30]. Just two states are needed in the L1 cache (Valid-Invalid) and there is no need for a Dirty/Clean bit (so evictions do not need to write-back). Further, the LLC always holds the correct data so it can immediately respond to requests. This means that there is no indirection for reads and that there is no need to track the writers at the directory. Invalidation is still required, however, and the readers need to be tracked. Alternatively, with a Null Directory in the LLC and Dir_0B protocol [4], or in a bus-based multicore, broadcasts to all the caches are used for invalidation. Unfortunately, because the number of write-throughs far exceeds the number of write-backs, this results in abysmal performance, and significantly increased traffic and power, as we show in Section 5.

2.2 Private vs. Shared Data Classification

Recent work realizes the importance of classifying private and shared data. Some work uses hardware mechanisms for performing this task [28, 15], other rely on the operating system [14, 18, 11], and other on the compiler [20]. The advantage of hardware mechanisms is that they can work at a granularity of a cache line but can have prohibitive storage requirements. On the other hand, the techniques which employ the OS do not impose any extra requirements for dedicated hardware, since they store the information along with the page table entries (PTEs) working at a page granularity. This means that if a single block in the page is shared (or even if two different private blocks within the same page are accessed by different cores) the whole page must be considered as shared, thus, leading to misclassified blocks. Finally, the disadvantage of the compiled-assisted classification is that it is difficult to know at compile time if a variable is going to be shared or not.

On the other hand, different proposals use this classification to reach different goals. Some of them to perform an efficient mapping for NUCA caches [14, 20]. Others to reduce the number of broadcast required by a snooping protocol [18], or to reduce the size of the directory cache in a directory-based protocol [12, 11]. Finally, similarly to us, others use the classification for choosing among different behaviors for the coherence protocol [28, 15]. Our goal is to simplify coherence. Thus, the most appropriate way of classifying blocks is the one managed by the operating system, due to its simplicity, effectiveness, and lack of any requirements for extra hardware. It allows us to define two completely isolated protocols for private and shared data, that can be verified independently.

2.3 Self-Invalidation

Dynamic Self invalidation and *tear-off* copies were first proposed by Lebeck and Wood as a way to reduce invalidations in cc-NUMA [19]. The basic idea is that cache blocks can be *teared off* the directory (i.e., not registered there as cached copies) as long as they are discarded voluntarily before the next synchronization point by the processor who created them. As the authors note, this can only be supported in a weak consistency memory model (for SC, self-invalidation needs to be semantically equivalent to a cache replacement).

Lebeck and Wood proposed this as an optimization on top of an existing cc-NUMA protocol. Furthermore, they made an effort to restrict its use only to certain blocks through a complex classification performed at the directory. Their design choices reflect the tradeoffs of a cc-NUMA architecture: not applying self-

invalidation indiscriminately is because misses to the directory are expensive.

Self-invalidation was recently used by Kaxiras and Keramidas in their "SARC Coherence" proposal [17]. They observe that with self-invalidation, writer prediction becomes straightforward to implement. The underlying directory protocol is always active to guarantee correctness. Despite the advantage for writer prediction, their proposal increases the complexity of the base directory protocol with another optimization layer and leaves the directory untouched. Finally, Choi *et al.* use self invalidation instructions, inserted by the compiler after annotations in the source program, in their application-driven approach [10]. We discuss this further in Section 7.

3 PROTOCOLS

Our approach boils down to two successive steps: i) reduce protocol complexity by making a write-through policy to the LLC practical; ii) eliminate the directory from the LLC and all invalidation (including broadcasts). Here, we give the important details for these steps.

3.1 Step 1: Simplifying the Protocol

The centerpiece of our strategy for reducing the complexity of coherence is to distinguish between private and shared data references. For the coherence of the shared data, we rely on the simplicity of a write-through policy. However, the write-through policy does not have to be employed on the private data, for which a write-back policy can be safely used without any coherence support (apart from the one already required for uniprocessors). In the L1 a *Dynamic Write-Policy* distinguishes between private and shared data guided by page level information supplied by the TLB. Pages are divided into "Private" and "Shared" at the OS level, depending on the observed accesses. Because the resulting protocols have only two states (Valid/Invalid) and we differentiate between private and shared data, we call the overall protocol VIPS (Valid/Invalid — Private/Shared).

We will just dwell on single point: our proposal is minimally intrusive in the design of the core and its local cache. In fact, it leaves the L1 cache *unmodified*. We assume that each L1 line has the common Valid/Invalid (V) and Clean/Dirty (D) bits and that TLB entries use two bits from the reserved ones to indicate the *Private*/*Shared* (P/S bit) status of the page and to lock the TLB entry (L bit) when we are switching from write-back to write-through. The dynamic write policy is implemented outside the L1. The P/S bit of the accessed data controls whether a write-through will take place.

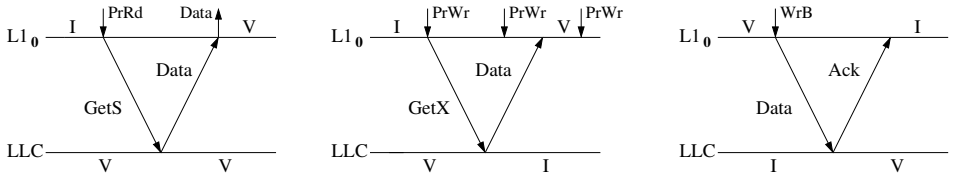


Figure 2: Read (PrRd), write (PrWr), and write-back (WrB) transactions for private lines.

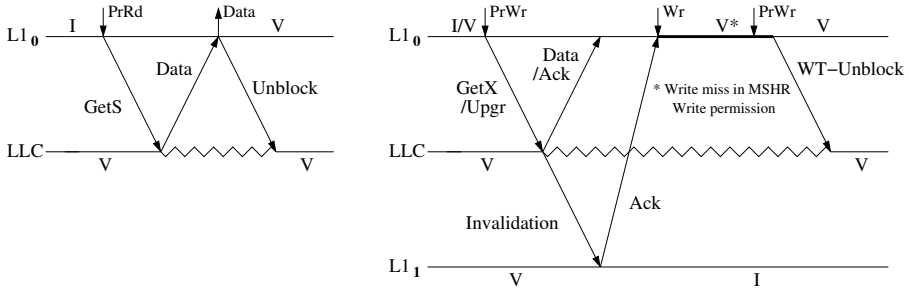


Figure 3: Read (PrRd) and write (PrWr) transactions for shared lines.

VIPS Protocol: Write-Back for Private Lines. The protocol transactions for private lines are simple, since no effort is expended on maintaining coherence. Figure 2 shows the read (PrRd), write (PrWr), and eviction (WrB) initiated transactions. For private request the LLC controller does not block the corresponding lines. The Write-Back transaction requires an acknowledgment for memory ordering purposes (so that fence instructions can detect its completion) in the corner case when a line changes designation from private to shared after it has been written.

VIPS Protocol: Write-Through for Shared Lines. The transactions for shared lines follow a write-through protocol. As in a MESI implementation, the LLC controller is blocked both for PrRd and PrWr transactions and requires *Unblock* messages to unblock it[1]. A PrRd transaction that misses in L1 cache gets the line and set its state to Valid. PrWr transactions send a *GetX* to the LLC controller that blocks the line (and gets the data if needed). Copies in other cores are invalidated (with the acknowledgments returning to the writer). When all acknowledgements arrive the write (*Wr*) is performed. The write-through can be delayed arbitrarily (we call this *Delayed Write-Through*) keeping the LLC line blocked. While the L1 line is in state V and dirty (i.e., there is still an entry for that line in the MSHR structure), it can be written repeatedly. The write-through (*WT-Unblock*) clears the L1 dirty bit, unblocks the line, and writes the new data in the LLC. A PrWr on a clean, valid, line initiates a write-through anew.

Transitions Between Write-Back and Write-Through. A page accessed by a single core starts as private in the page table, so a write-back policy is applied for every requested line within that page. When

[1]Note that MESI protocol also requires blocking the LLC controller for write-backs.

a second core accesses the same page it notices that it is tagged as private by another core. The first core needs to be interrupted and its TLB entry updated so it can see the page, henceforth, as shared. The write policy of the lines within this page will change from write-back to write-through. Consequently, those lines marked as dirty in the L1 cache for the core being interrupted need to be cleared by means of a write-back transaction. While this is an expensive operation, it is rather rare. The same technique for detecting private and shared pages has been used in recent work and in the interest of space, we will refer the reader to the excellent and detailed descriptions published elsewhere [14, 11].

Delayed Write-Throughs. The obvious optimization to any Write-Through cache is to reduce the amount of write-throughs by coalescing as many writes as possible. In the transactions presented above, the write-through can be delayed controller and no other transaction on it can proceed. In the meantime, the line can be written multiple times by the same core. This corresponds roughly to a MESI "Modified" state, but is strictly transient (exists only from the *GetX* to the write-through that unblocks the LLC line) and invisible to transactions from other cores. During that time, the address of the line is in one of the core's MSHRs. We assume that the MSHRs track only addresses and meta-information but do not carry a copy of the data. One simple implementation of the delayed write-through is to augment the MSHRs with a timer that causes the actual write-through to happen a fixed delay after the initial write.[2] Since the delayed write-through is transparent to other cores, this allows our protocol to have the same states as a traditional simple write-through protocol, thus significantly reducing the number of race conditions, and therefore, transient states with respect to a MESI protocol.

Comparison to MESI. It is straightforward to see that even with just the standard Valid/Invalid and Clean/Dirty bits (and the Private/Shared classification at the page level), we can implement a variant of the MESI protocol. Table 1 summarizes the correspondence. The E state corresponds solely to private lines and the shared lines follow an MSI protocol where the M state is does not exist as a stable state. One difference between our transient M state and the MESI M state is that ours cannot be downgraded by another core's read (it is invisible to other cores). Instead it "downgrades" automatically to a clean V state when it is written-through (possibly after a delay). Other cores' reads that find the line blocked at the LLC controller, because of this transient M state, have to wait the automatic downgrade.

[2] In the evaluation (Section 4) we assume 16 MSHRs, and a maximum delay of 1000 cycles in the absence of MSHR pressure. An alternate implementation is with with a separate small coalescing buffer on the side of the L1, draining slowly as new entries are allocated into it (possibly in FIFO order). This approach is more complex since both the buffer and the L1 must be updated on every write.

Table 1: Comparison to MESI

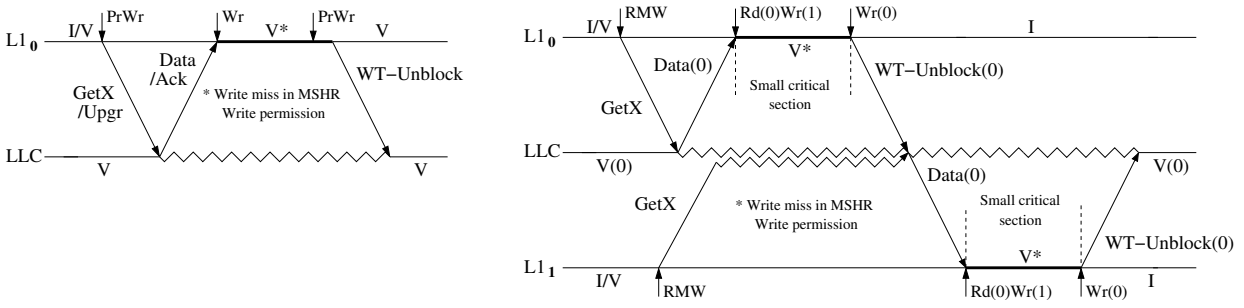
MESI States	V/I – P/S	Comment
I	I	Invalid is the same everywhere
S	(V, Shared) or (V, Private)	
E	V, Private	The E state is subsumed by the distinction between Private and Shared
M	(V, Private) or (V, Shared, Dirty)	The M state is subsumed by the V state in private lines; for shared lines the M state exists only transiently for the Delayed Write-Throughs —while the line is blocked at the LLC controller and the L1 dirty bit is set.

3.2 Step 2: Eliminating the Directory

Our next goal is to eliminate the directory. We have already removed the need for tracking the writer in the directory with the private-shared classification and the write-through policy. What is left is to get rid of the need to track the readers of a memory location just to invalidate them later on a write. Self-invalidation serves exactly this purpose [19]: readers are allowed to make unregistered copies of a memory location, as long as they promise to invalidate these copies at the next synchronization point they encounter. Our approach is similar but with a difference: *we allow writable tear-off copies* as in [10]. All shared data in the L1 caches whether read or written to –not just data brought in as Read-Only, e.g., as in [19] and [17]– are tear-off copies. A core encountering a synchronization point such as a lock acquire or lock release (as well as barriers, or wait/signal synchronization, and even fences) flushes all its shared data from the L1. Since we flush only shared and not private data, we call this *Selective Flushing*, (*SF*). Implementing selective flushing incurs very little change to the cache design. Valid bits are guarded by per-line *Private*/*Shared* (P/S) bits. The P/S bits are set when a line is brought into the L1. Subsequently a "flush" signal, resets all the valid bits guarded by P/S bits in state *Shared*. The implementation of the flush is straightforward when valid bits are implemented as clearable flip-flops outside the L1 arrays. As is pointed out in [19], and later in [17] and [10], self-invalidation, and by extension, selective flushing, implies a weak consistency memory model and works only with Data-Race-Free (DRF) programs, thereby providing SC for DRF [3].

Selective Flush (VIPS-SF) Protocol. Figure 4a shows the simple write transactions for the shared lines with selective flush (the protocol for private lines and the read transactions remain the same). The important change is that we do not have invalidation transactions any more. Blocking of LLC lines is still employed in this protocol and must be implemented at the LLC controller.[3] Another difference is that write-throughs cannot be delayed beyond a synchronization point since updates to shared data need to be made

[3]To support blocking, a Block/Unblock bit is needed only for the lines for which a Write-Through is in progress. The number of incomplete transactions is bounded by the number of cores and the number of MSHRs in each core.



(a) A single write transaction.

(b) Two cores acquiring a lock.

Figure 4: Write (PrWr) transaction for shared lines in the SF protocol.

visible to other cores after synchronization. Thus, we force any outstanding write-through to complete on synchronization. This translates to a flush of the MSHRs along with the selective flush of the cache.

Synchronization in VIPS-SF. Synchronization relies on data races. Instructions such as *Test&Set* or *Compare&Swap*, race to read-modify-write atomically a memory location if a condition is met (i.e., the "Test" or "Compare" parts). Otherwise, a copy of the memory location allows a core to spin locally in its L1 until the condition is changed by another core. In our approach, because we have no invalidations, a core cannot "signal" a change in the condition to the other cores that might be spinning, endangering forward progress. Therefore, atomic instructions always re-read the LLC copy.

The protocol is shown in Figure 4b for a simple atomic instruction such as Test&Set. Regardless of the existence of an L1 copy, an atomic instruction sends a *GetX* to the LLC. If the line is unblocked, its data are returned to the core, and if the test succeeds, the line is written with a new value (indicating, for example, that a lock is held by a core). Throughout the duration of the read-modify-write the line is blocked by the LLC controller; it is only unblocked by a final write-through. In the interim no other core can complete any transaction on that line (as core 2 in Figure 4b). Their requests enter a finite queue (bounded by the number of cores) managed by the LLC controller.

At first sight, bypassing the L1 and re-reading the LLC seems to make spinning very expensive. But this is not always so. By delaying the write-throughs of atomic instructions in the MSHRs, we are delaying the completion of a successful lock acquire. This may seem counter-intuitive but has a significant advantage. The more we delay the write-through of a winning lock acquire the more we reduce the LLC spinning of the other cores that are trying to get the lock at the same time. Other cores are blocked at the LLC controller and cannot even complete the Test part of the Test&Set. In fact, it is quite possible, that for a short critical section, the write-back of the Test&Set can be delayed in the MSHR for the whole duration of the

critical section as shown in Figure 4b. The lock release operation which is a simple write on the same lock, coalesces with the delayed write-through of the Test&Set. After the lock release, the delayed write-through must complete immediately to pass the lock to the next core in line. While we can eliminate spinning for short critical sections, in longer ones the write-through of the atomic instruction eventually completes and spinning resumes by the other waiting cores. This spinning in the LLC cache can increase traffic, so an exponential back off in software is essential to lessen it.

DRF Granularity and the Multiple-Writer-Merge (VIPS-M) Protocol. The VIPS-SF protocol is described above for write-throughs at a cache-line granularity. The implication is that it requires data-race-free operation at the granularity of a *cache-line*. This is because false sharing on a cache line –although not strictly a data race on a single memory location– is not handled correctly. To see this, consider two unsynchronized write-throughs from different cores, to two different words in the same cache line. This is a data race at the cache-line level but not at word level. Since the write-throughs happen at the granularity of a cache line, they can overwrite each other's new values, leading the system to a non-recoverable state. There are at least three solutions to the DRF granularity problem:

- Demand software guarantees for DRF operation at the cache-line level.
- Modify the protocol to always re-read the LLC line with every new write (every *GetX*), but this would considerably increase traffic.
- Perform write-throughs at a word granularity (which can also reduce the total number of words transfered to the LLC).

Our goal is for our approach to be as widely applicable as possible with minimal burden on the software, thus we cannot choose the first solution. Fortunately, the third one can be efficiently implemented.

Write-throughs at a word granularity require per-word dirty bits. This allows multiple concurrent writers on a cache line (false sharing) to write-through to the LLC just the words they modify but no other. Write-throughs from different cores are *merged* in the LLC. The important realization here is that immediately seeing the new values written by other writers is *not a requirement* in a weak consistency memory model already implied by self-invalidation.[4]

We call this version of the Selective Flush protocol Multiple-Writer-Merge and denote it with a simple M suffix: *VIPS-M*. An important implication of word granularity for the write-throughs is that it makes

[4] A word written by another writer cannot be read without synchronization –this would constitute a data race. Values written in the same cache-line by others become visible only after a self-invalidation which discards all shared data from the L1.

blocking of the lines at the LLC controller unnecessary. But this in turn, allows us to *equate the protocol for shared, data-race-free data to the protocol for private data.* At word granularity, we simplify the write transaction to just a write-through without even sending a write request (*GetX*) to the LLC (that would block the LLC line). It is the same transaction as shown in Figure 2 for private data write-backs, except that the LLC line remains valid. Thus, practically all data, whether shared (data-race-free) or private, are handled in exactly the same manner without *any* state in the LLC. The only difference is in *when* dirty data are put back in the LLC. Private data follow a write-back on eviction policy, while shared, data-race-free data follow a delayed (up to a synchronization point or an MSHR replacement) write-through policy. To stress the equivalence of private and shared DRF data, the latter write policy can be thought of as an *induced write-back policy*. Synchronization data, however, are still accessed following the blocking protocol described previously.

The overhead of the M version is that we need to track exactly what has been modified in each dirty line so we can selectively write back only the modified words to the LLC. One would assume that this means per-word dirty bits for every line in the L1. But per-word dirty bits are needed only for delayed write-throughs and are attached only to the MSHRs. No additional support is needed in the L1 or the LLC –other than being able to update individual words in the LLC.

3.3 Putting it all together: Memory Consistency and Coherence

Let us now return to the definition of coherence by Sorin, Hill, and Wood, called the Single-Writer/Multiple-Reader (SWMR)/Data-Value invariant. The definition (quoted from [30]) has two parts:

- *"For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it."*
- *"Data-Value Invariant: the value of a memory location at the start of an epoch is the same as the value of the memory location at the end of its last readwrite epoch."*

A coherence protocol that satisfies this definition is *invisible* to the underlying memory consistency model [30]. In other words, "correct" coherence cannot weaken a memory model. We use this definition to reason about the behavior of the two proposed protocols with respect to memory models. Table 3 summarizes the results.

Step 1: VIPS protocol. The protocol in Step 1 adheres to the SWMR definition without constraints. Even in the face of data races and/or false sharing, it allows only one writer at a time (because of the

Table 2: SC litmus test

Core C1	Core C2	Comments
S1: x = NEW;	S2: y = NEW;	Initially x = 0, y = 0
L1: r1 = y;	L2: r2 = x;	

invalidation) and guarantees that values are propagated correctly because of the blocking on the LLC line. It is therefore invisible to the memory consistency model and, thus, can support even the strictest model: SC.[5]

Step 2: VIPS-SF and VIPS-M protocols. The lack of directory and invalidations in Step 2 leads to a protocol that is *incoherent* for data races, but adheres to the SWMR definition for *data-race-free* operation.

It is easy to see how Step 2 violates the SWMR invariant for data races. Consider the following classic example for SC in Table 2. In an SC implementation r1 and r2 cannot be both 0, after the execution of the code shown for cores C1 and C2. A coherence protocol adhering to the SWMR/Data-Value invariant cannot change this. However, Step 2 does! Assume that y is cached in C1 before the execution of the C2 code. Since C2 cannot invalidate the cached copy of y, L1 will load 0 into r1. Similarly for x in C2, resulting in both r1 and r2 having the value 0, even after both writes have been performed. The single-writer invariant is violated.

It is equally straightforward to show that VIPS-SF is *incoherent* with false sharing. As we showed in Section 3.1, unsynchronized write-throughs to the same line can bring the system into a non-recoverable state. This is because two writers involved in false sharing, update not only their part of the line but also the other's, violating the single-writer invariant. Assume, however, that a program is DRF at a cache line granularity. Then VIPS-SF satisfies the coherency definition: i) there is only one writer per cache line at a time (otherwise there would be a data race), ii) the data value invariant holds because writes and reads are separated by synchronization, whereupon all shared lines are flushed. Thus, VIPS-SF is coherent for data-race-free cache lines, which means that it is invisible to *SC for DRF-cache-lines*.

Finally, the VIPS-M protocol satisfies the SWMR/Data-Value invariant for DRF operation *at a word granularity* because write-throughs are performed at this granularity. Assuming DRF operation at the word level, false sharing no longer violates the single writer invariant. Thus, VIPS-M is invisible to *SC for DRF*.

Similarly to the reasoning of SC for DRF [3], we implement Coherence for DRF. DRF satisfies by itself the single writer multiple reader invariant. All we have to do is guarantee the Data Value invariant and this

[5]Even the difference from the MESI protocol, the fact the Step 1 has automatic downgrades and not "forced" downgrades, is inconsequential, since there is at least one SC execution that corresponds to a small delay of the downgrade.

Table 3: Putting all together: Consistency and Coherence supported by the protocols. VIPS: Valid/Invalid Private/Shared, VIPS-SF: Selective Flush, VIPS-M: Selective Flush and Multiple-Writers-Merge

	Programs		
	Non-DRF	**DRF-word (Non-DRF cache line)**	**DRF-cache-line**
Step 1: VIPS	SC, Coherent	SC, Coherent	SC, Coherent
Step 2: VIPS-SF	Incoherent	Incoherent	SC for DRF_cache-line
Step 2: VIPS-M	Incoherent	SC for DRF_word	SC for DRF_cache-line

is achieved writing-through the correct data, and flushing the L1 at synchronization. This is why in VIPS-M we can equate the protocol for shared DRF data to the protocol for private data. The exception, of course, is due to synchronization data (accessed by atomic instructions) which inherently have data races. Changing the synchronization model to one not based on data races, would relieve us from the burden of even having a protocol just for these special cases.

3.4 Optimizations

In this section, we discuss optimizations to our protocols. So far we have been in the process of de-evolving coherence: we removed most coherence state (i.e., the M,O,E states) and the directory. It is therefore important that any optimization we introduce is neutral to complexity and hardware cost. We consider only very simple optimizations that adhere to this principle.

Read-Only Pages. Self-invalidation can cause needless misses on shared data that have not been modified. Complex techniques to exclude such data have been proposed [19]. In our approach, we simply tag pages as Read-Only (RO) if they are not written, and Read-Write (RW) otherwise. A page stars as RO but transitions to RW on the first write (there is no reverse transition). Because the page is shared, all the cores that have a TLB entry must be interrupted and notified of this change. Cache lines belonging to RO pages are spared from self-invalidation. Although a crude approximation to the optimization proposed by Lebeck and Wood, it yields good results and we use it in the evaluation. More sophisticated approaches can be explored in future work.

Relaxing Inclusion Policies. In a MESI protocol, where the directory information is stored along with the LLC entries, inclusion between the L1 and the LLC is enforced. When a line is evicted from the LLC, the directory information is evicted as well, and for maintaining coherence, all copies in L1 are invalidated. In the *VIPS* protocol, private lines do not require directory information, and therefore we can selectively relax the inclusion policy for them. An *exclusive* policy for private lines can make better use of the LLC storage, potentially reducing expensive off-chip misses. But, silent *conflict* evictions of clean L1 copies, result in

Table 4: System parameters.

Memory Parameters		Network Parameters	
Processor frequency	3.0GHz	Topology	2-dimensional mesh (4x4)
Block / Page size	64 bytes / 4KB	Routing technique	Deterministic X-Y
MSHR size / delay timeout	16 entries / 1000 cycles	Flit size	16 bytes
Split L1 I & D caches	64KB, 4-way (256 sets)	Data message size	5 flits (if MWM not used)
L1 cache hit time	1 (tag) and 2 (tag+data) cycles	Control message size	1 flit
Shared unified L2 cache	512KB/tile, 16-way (512 sets)	Routing time	2 cycle
L2 cache hit time	2 (tag) and 4 (tag+data) cycles	Switch time	2 cycle
Memory access time	160 cycles	Link latency (one hop)	2 cycles

subsequent misses in the LLC. One option is to victimize clean L1 copies into the LLC. This increases traffic, but saves latency in L1 conflicts. A middle-of-the-road approach that reduces this traffic is to opt for a *non-inclusive policy* where data requested by the L1 with read-only permission are also stored in the LLC. This way, evictions of clean lines can be silent both in the L1 and in the LCC. Because the LLC is not stressed by our benchmarks, we have seen that the later approach can reduce traffic requirements. In the *VIPS-M* protocol inclusion is not required for any line, since we do not have directory information in the LLC. Therefore, the same optimizations for private lines in the *VIPS* protocol, are applicable to all lines in the *VIPS-M* protocol.

4 EVALUATION METHODOLOGY

4.1 Simulation Environment and System Configuration

The evaluation of the protocols proposed in this work is carried out with full-system simulation using Virtutech Simics [22] and the Wisconsin GEMS toolset [23]. The interconnection network has been modeled using GARNET [5]. We simulate a 16-tile CMP architecture. The values of the main parameters used for the evaluation are shown in Table 4. Through experimentation we have found that only 16 entries per MSHR are needed to keep shared data as dirty for enough time to avoid most write misses. Likewise, a timeout between 500 and 2000 cycles for the delayed write-throughs (or up to eviction in the MSHR) offers a good compromise between the reduction in the number of extra write-misses and the time the LLC remains blocked. Cache latencies, energy consumption, and area requirements are calculated using the CACTI 6.5 tool [25] assuming a 32nm process technology.

We evaluate the five cache coherence protocols shown in Table 5. This table summarizes some of their characteristics. The first protocol (*Hammer*) corresponds to a broadcast-based protocol for unordered networks [27]. Since in this protocol invalidations are sent to all cores, there is no need for a directory. How-

Table 5: Protocols evaluated.

Protocol	Invalidations	Directory	Indirection	Requires inclusion	L1 base states	LLC tag area
Hammer	Broadcast	None	Yes	No	5 (MOESI)	0.0501 mm^2
Directory	Multicast	Full-map	Yes	Yes	4 (MESI)	0.0905 mm^2
Write-through	Multicast	Full-map	Only for write misses	Yes	2 (VI)	0.0905 mm^2
VIPS	Multicast	Full-map	Only for write misses	Only for shared blocks	2 (VI)	0.0905 mm^2
VIPS-M	None (self-invalidation)	None	No	No	2 (VI)	0.0501 mm^2

ever, this protocol generates a significant amount of traffic, which dramatically increases with the number of cores. The second protocol (*Directory*) corresponds to a MESI directory-based protocol where the directory information is stored along with the LLC tags. The directory information allows the protocol to filter some traffic, and therefore, save energy consumption. Inclusion between L1 caches and the LCC is enforced in this case. The main advantage of the third protocol (*Write-through*) is its simplicity, since it only has two base states for lines in L1 caches (as do our protocols). Although this protocol accelerates read misses by removing their indirection, the write-through policy increases the number of write misses and severely hurts performance. The fourth protocol is our SC protocol (*VIPS*). It only has two base states, and can relax the inclusion policy for private blocks. Despite being simple it still requires invalidations. Finally, the fifth protocol (*VIPS-M*) provides SC only for DRF applications. The main characteristic of this protocol is that it completely removes both the need of storing directory information and sending invalidations, as well as the indirection for all cache misses. The absence of a directory reduces the LLC tag area, and allows the protocol to implement any inclusion policy.

4.2 Benchmarks

We evaluate the described protocols with a wide variety of parallel applications. *Barnes* (16K particles), *Cholesky* (tk16), *FFT* (64K complex doubles), *FMM* (16K particles), *LU* (512×512 matrix), *Ocean* (514×514 ocean), *Radiosity* (room, -ae 5000.0 -en 0.050 -bf 0.10), *Volrend* (head), *Water-Nsq* (512 molecules) and *Water-Sp* (512 molecules) belong to the *SPLASH-2* benchmark suite [32]. *Em3d* (38400 nodes, 15% remote) is a shared-memory implementation of the Split-C benchmark. *Tomcatv* (256 points, 5 time steps) is a shared-memory implementation of the SPEC benchmark. *x264* (simsmall) is a media processing application from the PARSEC benchmark suite [6].

To accurately simulate our *VIPS-M* protocol we have instrumented the synchronization mechanisms (locks, barriers, conditions) used by the benchmarks so they are "visible" by the hardware. Synchronization points are signaled by *fences* preceding synchronization. In this way, processors can perform the selective

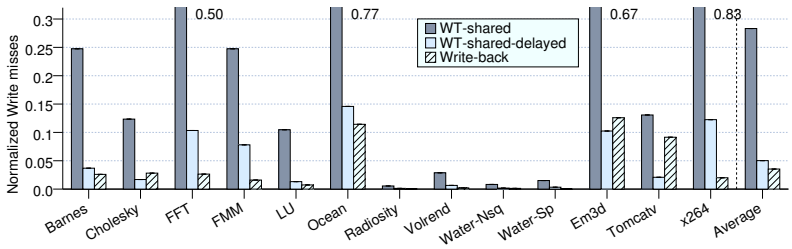


Figure 5: Reduction in write misses due to the private/shared classification and delayed write-through. Write misses have been normalized with respect to the simple write-through policy.

flushing on their caches when required. Data accessed by atomic instructions follow the synchronization protocol described in Section 3.2. We simulate the entire applications, but collect statistics only from start to completion of their parallel part.

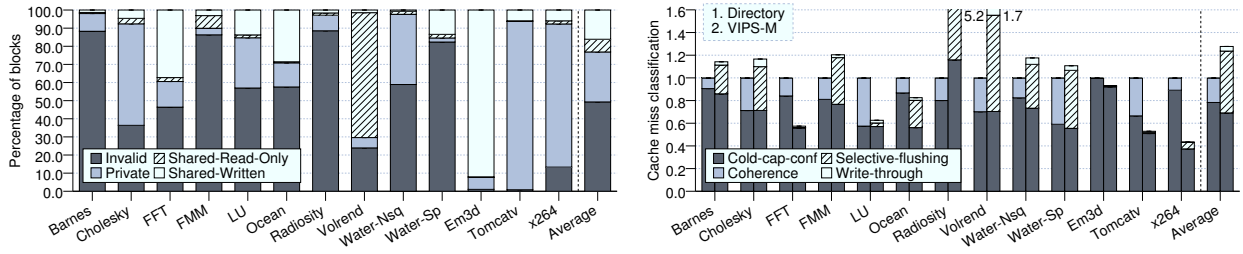
5 EXPERIMENTAL RESULTS

5.1 Impact on Write Misses

As discussed throughout this paper, the main drawback of a write-through policy is the significant amount of write misses it entails. In this section, we show the ability of both the private/shared classification and the delayed write-throughs to reduce this overhead. Figure 5 shows the number of write-misses for the different mechanisms normalized with respect to a write-through policy (not shown). Since most of the write misses in a write-through protocol come from private blocks (Figure 1), by switching to a write-back policy for private blocks we can save 72.7%, on average, of write misses (first bar). However, if we compare to a write-back policy (third bar), we can see that we still incur extra misses. Although the effectiveness of the private-shared classification in reducing write misses is noticeable, a delayed write-through mechanism is still necessary to be competitive to a write-back policy (a MESI directory protocol in this case). This is shown in the second bar (delayed write-throughs), where the number of write miss is reduced by 95.0% (on average) compared to a write-though protocol, thus making this number close to the one obtained by a write-back policy (96.5% on average).

5.2 Selective Flushing

The *VIPS-M* protocol relies on selective flushing at synchronization points to keep coherence (and provide SC) for DRF applications. We only flush lines whose page is being shared among different cores and modified by at least one of the cores (read-only optimization, Section 3.4). As shown in Figure 6a this selective flushing prevents about 68.2% on average of valid lines from being evicted from the cache. This



(a) Lines found in the cache upon a selective flushing. (b) Number of cache misses normalized with respect to *Directory*.

Figure 6: Impact of Selective Flushing.

significantly lessens the number of misses as consequence of self-invalidations. We can also observe that 16.1% of cache lines will be flushed. Most of them are silently invalidated because their copy is clean. This happens for lines brought in the cache as consequence of read misses, or lines that have performed a write-through (synchronization or DRF lines). Dirty lines for which a write-though has not happened will be sent immediately to the LLC. *Fence* instructions must wait for the acknowledgments of such write-throughs to guarantee proper memory ordering. Frequent synchronization results in parts of the cache already being invalid in the next flush.

Selective flushing prevents significant part of the cache from being needlessly invalidated and can be competitive to directory invalidations. Figure 6b shows the misses in a directory protocol and in *VIPS-M* classified by the event that caused them. The percentage of cold, capacity, and conflict misses (*Cold-cap-conf*), slightly decreases in *VIPS-M* due to the lack of write misses for DRF lines. For some applications, e.g., *FFT*, *LU*, *Em3d*, *Tomcatv*, and *x264*, the impact of the selective flushing on the miss rate is negligible. In *FFT* and *LU* this is because they have only a few barriers, so the selective-flushing is not frequent. In *Em3d*, *Tomcatv*, and *x264* the working set accessed between synchronization points is much larger than the cache size (few invalid lines are flushed, as shown in Figure 6a), thus, after a synchronization point misses are not due to self invalidation. On the other hand, applications like *Radiosity*, and *Volrend* incur numerous extra misses due to self-invalidation because of frequent locking. This impacts performance and energy consumption as we show in next section. For the remaining applications, the number of misses is comparable in both protocols.

5.3 Performance Results

Figure 7 shows the applications' execution time for the five protocols evaluated in this work (Table 5). The execution time has been normalized with respect to *Directory*. We can observe that the broadcast-based

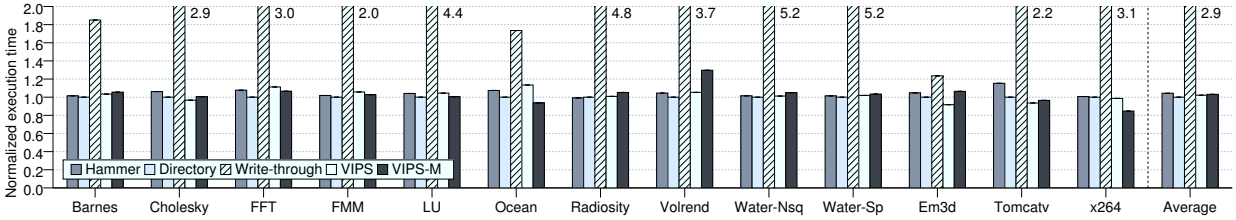


Figure 7: Normalized execution time with respect to *Directory*.

Hammer protocol slightly increases application's execution time with respect to *Directory*. The performance of the *Write-through* protocol is prohibitive due to the dramatic increase in the miss rate for writes. But this increase can be lessened if private lines are detected (and their write misses removed) and a delayed write-through mechanism is implemented. As we can observe, the simple protocol *VIPS* is just 2.2% slower than the complex *Directory*. Finally, *VIPS-M*, despite not having a directory structure, is just 3.1% slower than *Directory*. We can see that these two simple protocols achieve an average performance close to the one of a more complex directory (MESI) protocol. In fact, *Cholesky*, *Ocean*, *Em3d*, *Tomcatv*, and *x264* run faster with one of our two protocols.

5.4 Energy Consumption and Area

Figure 8 shows the energy consumed by the network and the LLC for the applications and protocols evaluated in this work. Broadcasting invalidations and receiving acknowledgments in *Hammer* leads to a significant increase in the energy consumed by the network (2x compared to *Directory*). Additional write misses in a write-through protocol result in a dramatic increase in traffic and LLC accesses. As with execution time, the detection of private pages, and the delayed write-throughs help to reduce L1 misses, and therefore, traffic and LLC accesses. Thus, *VIPS* reduces significantly the consumption when compared with *Write-through*, still consuming 7.5% more energy than a directory protocol due to the increase in the number of misses. However, *Barnes*, *Cholesky*, *Radiosity*, *Em3d*, and *Tomcatv*, actually consume less. Finally, although the selective flushing in *VIPS-M* causes extra L1 misses, thanks to the MWM protocol the amount of data written back into the LLC is reduced, which translates into a reduction in dynamic consumption of 4%, on average, with respect to *VIPS*, and just 3.6% additional average consumption with respect to *Directory*. *Cholesky*, *LU*, *Ocean*, *Em3d*, *Tomcatv*, and *x264* consume less than *Directory*.

With respect to area, CACTI reports for the LLC described in Table 4 and the protocols described in Table 5 a significant decrease in tag area. The tag arrays in the directory protocols *Directory*, *Write-Through*, and *VIPS* occupy 0.09mm^2 while in the directory-less protocols *Hammer* and *VIPS-M* occupy just 0.05mm^2.

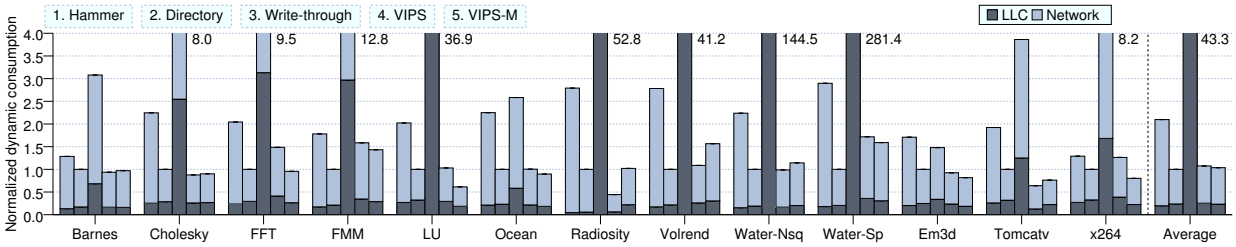


Figure 8: Normalized energy consumption with respect to *Directory*.

This corresponds to a reduction of 45%.

6 DISCUSSION

There are several issues concerning our proposal that we discuss in this section to argue for its applicability in real-world situations.

6.1 Inter-Chip Coherence

Extending our simple protocol to go beyond a single chip faces a different set of trade-offs. Naïvely, one could extend the same scheme outside the chip: write-backs would have to go through the LLC to the external memory and the entire LLC would need to be flushed of all the shared data on any synchronization. Because of limited bandwidth and high latency outside the chip both of these propositions are prohibitive. In particular, flushing the LLCs is bound to be a major performance issue because of the amount of shared data cached there. For this reason we need to selectively invalidate blocks in the LLC when another chip writes the block.

Consequently, an invalidation strategy is more appropriate for inter-chip coherence but comes at a cost. Here, we describe in broad strokes this strategy. Only the LLC of each chip needs to be interfaced to an invalidation-based protocol (e.g., MESI or MOESI), with state for LLC blocks and an inter-chip directory (or external broadcasts and snooping) –the rest of the on-chip cache hierarchy remains the same. Because there is no invalidation for the L1s, even with external invalidations the DRF semantics need to be preserved. Assume, for example that a write to the memory location A in chip M1, invalidates a copy of A in the LLC of chip M2. Although the LLC copy in M2 is invalidated, there is no way to forward this invalidation to the (M2) L1s. There are two cases: either there are copies of A in the L1s or they have been self-invalidated. The former case constitutes a data race if the L1 copies are read. This of course is forbidden. In the later case the system works correctly and coherence is extended across chips.

6.2 Multithreading

Until now, we have not distinguished between core and thread, assuming a one-to-one correspondence. There are, however, a few issues in simultaneous multithreading (SMT) mode, where two or more threads share the same core/L1: i) a page accessed in the L1 by multiple threads is classified as *core*-private but is actually shared; this leads to a subsequent PTE and page flush when a thread migrates or another core accesses the shared data, ii) threads flush each-other's data on any synchronization, and, iii) writes are visible to local threads before remote threads —the same issue exists even in the widely popular TSO (Total Store Order) memory model [30]. All these issues stem from not differentiating data and operations *per thread*. A well-known solution is to tag L1 lines with a thread ID and differentiate operations on them. This of course incurs the corresponding cost.

6.3 OS, Context Switches, Migration

Dealing with OS code and data is more complicated than with applications. One issue with OS is page classification. The OS executes in all cores. This means that even a simple migration from one core to another can (falsely) tag a private page as shared in our classification scheme. In addition, many a times, the OS executes concurrently on many cores, which leads to misclassification for data that are not actually shared, but their page appears to be. Although this also happens in applications, it is much more pronounced for OS pages. In fact, the amount of (truly) shared data in the OS relatively low, but most of its pages can be classified as shared. For us, this is an optimization issue. Better classification (possibly with reverse adaptation) or migration-aware classification by the OS itself (as sugested in [14]), could help alleviate OS page misclassification. Another issue is the flushing of shared data by the OS. In the implementation for this paper, context switches, system calls, I/O, etc., conservatively flush L1 shared data, so that updates are made visible to other cores. This may negatively impact both application and OS performance and could be optimized in any number of ways, including tagging the L1 lines as system or user (similarly to the thread ID tagging for SMT discussed above).

6.4 Verification

In VIPS-M the protocols for private and shared (DRF) data are the same. There are only two stable states (Valid/Invalid) and two transient states (IV and VI) in the L1s and no states at all in the LLC (no blocking). It is thus straightforward to verify and significantly more easier than MESI which requires extensive state

exploration [10]. The synchronization protocol (discussed in Section 3.2) requires more effort to verify, because it has LLC blocking and protocol races. This protocol is the result of our insistence to provide (for this work) the established memory semantics for synchronization primitives. We note, however, that not even Choi *et al.* implement synchronization in their protocols [10]. Alternative approaches to synchronization would eliminate the burden for any further verification effort beyond VIPS-M.

7 RELATED WORK

We have already discussed in passing most of the related work. Unfortunately, in the limited space of this paper we cannot do justice to all the work that influenced us. Here, we will just summarize the similarities and differences with the work closest to ours, the DeNovo work of Choi *et al.* [10]. Based on the properties of disciplined parallelism Choi *et al.* simplify coherence in a similar manner as we do. However, they rely on significant feedback from the application which must define memory regions of certain read/write behavior and then convey and represent such regions in hardware. This requires programmer involvement at the application layer (to define the regions), compiler involvement to insert the proper self-invalidation instructions, an API to communicate all this information to the hardware, and additional hardware near the L1 to store this information. We fully subscribe to their views for disciplined parallelism and data race-free operation. But, our goal is to proceed along the same motivation (simplify coherence) relying solely on an application-transparent approach. The only application involvement we may require (for the VIPS-M protocol) is DRF compliance.

There are, of course, many similarities to the resulting protocols in both approaches. Both rely on self-invalidation. The DeNovo approach self-invalidates the "touched" data in a phase, we invalidate the shared data on synchronization. However, the DeNovo approach still implements a directory ("registry") that tracks the writers (but not the readers), and cleverly hides it in the data array (since LLC data are stale in the presence of a writer). Although the directory storage cost is hidden, there is still directory *functionality* in the LLC. In our VIPS-M protocol the directory is fully abolished leaving behind a "passive" LLC. We rely on keeping the LLC up-to-date with write-throughs; fresh (non-stale) data are always found in the LLC. This also leads to another difference: the DeNovo approach has directory indirection for reads (to get the new values), but we do not. To eliminate this indirection Choi *et al.* use writer-prediction (in software or hardware) and revert to the registry on mispredictions.

The heavy reliance of the DeNovo approach on application involvement, unfortunately, prevents us from

replicating their results for a direct comparison of the two approaches. Without having access to the source code annotations for the regions, the same compiler for the self-invalidation instructions, and the API to convey information to the hardware, we found it exceedingly difficult to produce numbers for their coherence protocols. By inspecting the published results, however, we can make the following observations: i) both approaches are competitive to MESI. ii) DeNovo shows an advantage, but this is to be expected since it has access to much more in-depth information on application behavior — our page classification is coarse-grain in comparison, but completely transparent. iii) their results do not, apparently, include synchronization operations — for us, synchronization traffic significantly hampers performance; as with DeNovo, an alternative approach to synchronization would significantly improve our results. Overall, we consider the two approaches to be representative of the trade-off between application involvement and application-transparent implementation.

8 CONCLUSIONS

This paper goes contrary to the experience of more than three decades in coherence and takes us back to before the IBM centralized directory, Censier and Feautrier's distributed directory [7] or Goodman's write-once protocol [13]. We are inspired by many recent efforts to simplify coherence and/or reduce directory cost [10, 12, 11, 15, 17] but we go a step further towards eliminating the need for a directory. In a multicore cache hierarchy, as long as we treat private and shared data separately, performance need not be compromised even with the simplest protocol. Our approach is based on a dynamic write policy (Write-Through for shared and Write-Back for private data) and selective flushing of the shared data from the L1 caches (self-invalidation) upon synchronization. By separating private from shared data at the page level, we minimize the impact of the write-through policy, since many of the write-misses are due to private data.

We have shown that our protocols are efficient in many respects. VIPS (featuring a simple directory) approaches the performance, energy, and traffic of the more complex MESI directory protocol while maintaining compatibility with SC. VIPS-M removes the directory from the LCC and further simplifies the protocol for shared data to that of private data. It still performs quite well with respect to MESI. There are also weaknesses in our approach. The page classification is in many cases too coarse-grain and adapts only one-way (e.g., from Private to Shared, or from RO to RW). We believe that, there is room for considerable improvement and new optimizations for page classification. The synchronization protocol also resists simplification and costs us in traffic and performance.

Our approach puts into question some well-established concepts with respect to memory ordering and synchronization. Coherence as we know it bears a significant complexity burden to support data races just to be transparent to SC implementations. But for a weak consistency memory ordering that supports SC for DRF, data races for ordinary data are absent for correctly synchronized programs [3]. In the interest of programming clarity and correctness there is a growing consensus that data races should be banned. In our work, we show that in the absence of data races, we can simplify the coherence protocols for shared data to that of private data with a just a simple rule of when to write data back to the LLC. Data races, unfortunately, are still pertinent to synchronization operations. While we strive to maintain the memory semantics of atomic instructions (for compatibility) we note that this detracts from simplicity. In this light, we wish to point out that the multicore environment offers many as yet unexplored opportunities to devise new and effective synchronization mechanisms *not relying on data races* as for example in [31].

We view this work as a starting point, based on some interesting questions for multicore architectures: Is it worth supporting directories or broadcasts just so SC can be implemented in the presence of data races? And if data races are solely for synchronization, is it worth having coherence state for every possible cached address just so we can deal with these data races, or is there another better way to implement synchronization taking advantage this new environment of the multicore?

References

[1] D. Abts, S. Scott, and D. J. Lilja. So many states, so little time: Verifying memory coherence in the Cray X1. In *17th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, Apr. 2003.

[2] M. E. Acacio, J. González, J. M. García, and J. Duato. A new scalable directory architecture for large-scale multiprocessors. In *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 97–106, Jan. 2001.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996.

[4] A. Agarwal, R. Simoni, J. L. Hennessy, and M. A. Horowitz. An evaluation of directory schemes for cache coherence. In *15th Int'l Symp. on Computer Architecture (ISCA)*, pages 280–289, May 1988.

[5] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha. GARNET: A detailed on-chip network model inside a full-system simulator. In *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 33–42, Apr. 2009.

[6] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *17th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 72–81, Oct. 2008.

[7] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, 27(12):1112–1118, Dec. 1978.

[8] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *4th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 224–234, Apr. 1991.

[9] G. Chen. Slid - a cost-effective and scalable limited-directory scheme for cache coherence. In *5th Int'l Conference on Parallel Architectures and Languages Europe (PARLE)*, pages 341–352, June 1993.

[10] 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 *20th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2011.

[11] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *38th Int'l Symp. on Computer Architecture (ISCA)*, pages 93–103, June 2011.

[12] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi. Cuckoo directory: A scalable directory for many-core systems. In *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 169–180, Feb. 2011.

[13] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *10th Int'l Symp. on Computer Architecture (ISCA)*, pages 124–131, June 1983.

[14] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In *36th Int'l Symp. on Computer Architecture (ISCA)*, pages 184–195, June 2009.

[15] H. Hossain, S. Dwarkadas, and M. C. Huang. POPS: Coherence protocol optimization for both private and shared data. In *20th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2011.

[16] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589–604, July 2005.

[17] S. Kaxiras and G. Keramidas. SARC coherence: Scaling directory cache coherence in performance and power. *IEEE Micro*, 30(5):54–65, Sept. 2011.

[18] D. Kim, J. A. J. Kim, and J. Huh. Subspace snooping: Filtering snoops with operating system support. In *19th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, Sept. 2010.

[19] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 48–59, June 1995.

[20] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones. Compiler-assisted data distribution for chip multiprocessors. In *19th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 501–512, Sept. 2010.

[21] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, Mar. 2008.

[22] P. S. Magnusson, M. Christensson, and J. Eskilson, et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[23] M. M. Martin, D. J. Sorin, and B. M. Beckmann, et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *Computer Architecture News*, 33(4):92–99, Sept. 2005.

[24] A. Moshovos, G. Memik, B. Falsafi, and A. N. Choudhary. JETTY: Filtering snoops for reduced energy consumption in SMP servers. In *7th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 85–96, Jan. 2001.

[25] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0. Technical Report HPL-2009-85, HP Labs, Apr. 2009.

[26] B. W. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In *17th Int'l Symp. on Computer Architecture (ISCA)*, pages 138–147, June 1990.

[27] J. M. Owen, M. D. Hummel, D. R. Meyer, and J. B. Keller. System and method of maintaining coherency in a distributed communication system. U.S. Patent 7069361, June 2006.

[28] 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 Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 465–476, Sept. 2010.

[29] B. Smith. Personal communication. Oct. 2011.

[30] D. J. Sorin, M. D. Hill, and D. A. Wood. *A Primer on Memory Consistency and Cache Coherence*, volume 6 of *Synthesis Lectures on Computer Architecture*. Morgan & Claypool Publishers, May 2011.

[31] D. Vantrease, M. H. Lipasti, and N. Binkert. Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols. In *17th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, pages 132–143, Feb. 2011.

[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *22nd Int'l Symp. on Computer Architecture (ISCA)*, pages 24–36, June 1995.

[33] H. Zhao, A. Shriraman, and S. Dwarkadas. SPACE: Sharing pattern-based directory coherence for multicore scalability. In *19th Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages –, Sept. 2010.