Complexity-Effective Multicore Coherence

Alberto Ros
Department of Computer Engineering
University of Murcia, Spain
aros@ditec.um.es

Stefanos Kaxiras
Department of Information Technology
Uppsala University, Sweden
stefanos.kaxiras@it.uu.se

ABSTRACT

Much of the complexity and overhead (directory, state bits, invalidations) of a typical directory coherence implementation stems from the effort to make it "invisible" even to the strongest memory consistency model. In this paper, we show that a much simpler, directory-less/broadcast-less, multicore coherence can outperform a directory protocol but without its complexity and overhead. Motivated by recent efforts to simplify coherence, we propose a hardware approach that does not require any application guidance. The cornerstone of our approach is a dynamic, application-transparent, write-policy (write-back for private data, write-through for shared data), 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 from software. This leads to our main result: a virtually costless coherence that outperforms a MESI directory protocol (by 4.8%) while at the same time reducing shared cache and network energy consumption (by 14.2%) for 15 parallel benchmarks, on 16 cores.

Categories and Subject Descriptors

C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Parallel processors*; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms

Design, Experimentation, Performance

Keywords

Multicore, simple cache coherence, directory-less protocol, dynamic write policy, self-invalidation, multiple writers

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. ..." [31]

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 snooping or a directory to constantly track cached copies and send invalidations (or broadcasts). It necessitates additional protocol states for performance (e.g., Exclusive, Owned), which causes an explosion in the number of transient states required to cover every possible race that may arise. For example, the GEMS [24] implementation of the MESI directory protocol –a direct descendant of the SUNfire coherence protocol– requires no less than 30 states.

Complexity 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, 9, 10, 27], but also verification cost [11, 32]. 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 [19, 25, 34] with the intent of making complex protocols more power- or performance-efficient. Verification of such protocols is difficult and in many cases incomplete [1].

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 simplify coherence and practically eliminate the hardware cost (storage and logic), while at the same time achieving improvements in both performance and energy consumption.

Our proposal targets multicore/manycore[1] architectures where the relative cost of coherence is significant compared to the complexity of the cores. This includes many accelerators based on simple cores (e.g., Tilera [6]), standalone GPUs, but also cache-coherent shared virtual memory GPUs coupled to general purpose cores. We do not envision our

[1]In the interest of brevity, we will use the term multicore to describe both.

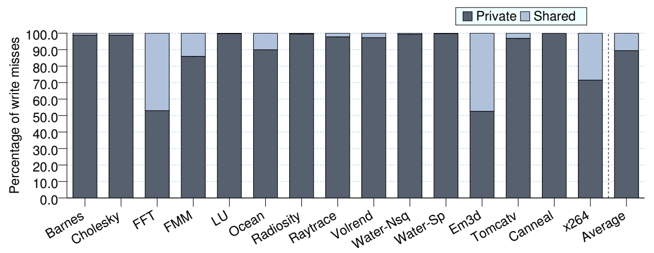


Figure 1: Percentage of write misses in a write-through protocol for both private and shared data

proposal for multicores based on few fat complex cores, where the relative cost of implementing a snooping or directory MESI protocol is not an issue, and such protocols are routinely implemented; nor do we explore workloads where performance is dominated by off-chip memory accesses rendering the on-chip coherence effects negligible (e.g., commercial workloads).

In addition, the 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.

Contributions. 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. We propose a **dynamic write policy**, a very simple albeit novel idea, that 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. 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) largely ignoring that in multicores there is an 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 (around 90%) in a write-through protocol actually come from private blocks, as Figure 1 shows. We select between Write-Back and Write-Through to the LLC depending on whether data are private or shared. The distinction between private and shared data is determined dynamically at a page granularity utilizing the page table and the TLBs [12, 15] (Section 3.1).

2. We **selectively flush** shared data from the L1 caches on synchronization points. Our approach is a generalization of *read-only* tear-off copies [19, 21] to all shared data (including writable copies). This step eliminates the need to track readers for invalidation, therefore obviating the need for a directory/broadcast or snoops. Write-throughs are performed by transferring only what is modified (diffs) in a cache-line, allowing multiple simultaneous writers per cache line (as long as their writes constitute false sharing on separate words in a cache line and not a true data race) (Section 3.2).

While write-through protocols and self-invalidation have been proposed separately in the past, we combine them and, for the first time, make them *practical* by applying them dynamically, based on a run-time division of data to shared and private at a page granularity. This leads to the main novel result we report in this paper: a minimal, very simple, virtually costless coherence protocol that does not require any application involvement or guidance. While our work bears some similarity to the work of Choi *et al.* we pursue an application-transparent approach, as opposed to their application-driven approach [11], we arrive at a truly directory-less coherence protocol while they still implement a "registry" in the LLC, and we provide support for synchronization without a directory or invalidations that is lacking in prior work.

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 [31] (Section 3.4). Thus, our protocol is incoherent for data races but satisfies the definition of coherence for the important class of Data-Race-Free (DRF) programs. Data races are the culprits of many problems in parallel software and the benefits of data-race-free operation are well argued by Choi *et al.* [11]. Thus, similarly to SC for DRF [3], our approach provides Coherency for DRF.

The second implication is that, synchronization instructions (such as T&S or Compare&Swap) which inherently rely on data races, require their own protocol for a correct implementation. We propose an efficient and resource-friendly **synchronization protocol** that works without invalidations (Section 3.3) and in many cases eliminates spinning.

Results. Our approach leads to a very simple cache 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, and outperforms a directory protocol by 4.8%, on average for 15 parallel benchmarks on a 16-core architecture with 32KB L1 caches. It minimizes control message traffic by not requiring invalidations or unblock messages, and it 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.

Results show that reductions in traffic and LLC power consumption can lead to energy saving of 14.2%, on average, with respect to a directory protocol.

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 [31]. 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 $\mathrm{Dir}_0\mathrm{B}$ 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 [16, 29], other rely on the operating system [12, 15, 20], and other on the compiler [22]. The advantage of hardware mechanisms is that they can work at cache-line granularity but can be prohibitive in their 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 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 non-uniform cache access (NUCA) caches [15, 22]. Others to reduce the number of broadcast required by a snooping protocol [20], or to reduce the size of the directory cache in a directory-based protocol [13, 12]. Finally, similarly to us, Pugsley *et al.* [29] and Hossain *et al.* [16] use the classification for choosing among different behaviors for the coherence protocol but both of these approaches rely on *directory* or *bus invalidation* coherence. 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 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 [21]. The basic idea is that cache blocks can be teared off the directory (i.e., not registered there) 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 trade-offs 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 [19]. 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 [11]. 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).

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.

3.1.1 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)

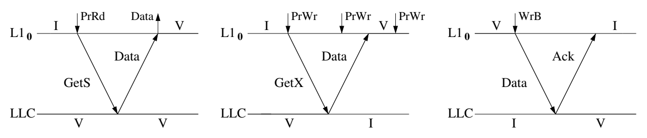


Figure 2: Read (PrRd), write (PrWr), and write-back (WrB) transactions for private lines

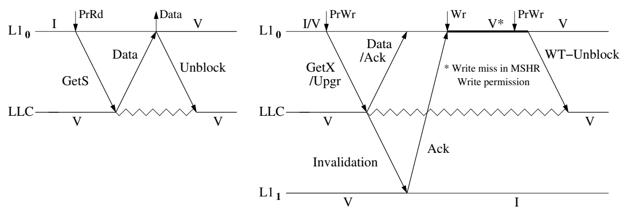


Figure 3: Read (PrRd) and write (PrWr) transactions for shared lines

initiated transactions. For private request the LLC controller does not block the corresponding lines, i.e., there are no transient states. 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 page changes designation from private to shared after it has been written.

3.1.2 VIPS Protocol: Write-Through for Shared Lines

The transactions for shared lines follow a write-through protocol (Figure 3). As in a MESI implementation, the LLC line is blocked both for PrRd and PrWr transactions and requires *Unblock* messages to unblock it. A PrRd transaction that misses in the L1 gets the line and set its state to Valid (Figure 3, left side). PrWr transactions (Figure 3, right side) 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 there is still an entry for that line in the MSHR structure and the line 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 and valid line initiates a write-through anew.

3.1.3 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 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 [12, 15].

3.1.4 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. 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. 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.

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 [21]: 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: *all* shared data in the L1 caches whether read or written to –not just data brought in as Read-Only, e.g., as in [21] and [19]– are tear-off copies. A core encountering a synchronization point (lock acquire/release, barriers, wait/signal synchronization) flushes 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 prior work [11, 19, 21], self-invalidation, and by extension selective flushing, implies a weak consistency memory model and only guarantees SC for Data-Race-Free (DRF) programs [3].

However, even with a DRF guarantee the lack of invalidations can cause problems. Consider two concurrent readers, each holding a valid copy of the same cache line. Assume that the two readers decide to write two different words in the cache line –false sharing– without any intervening synchronization. If their write-throughs happen at the granularity of a cache line, they can overwrite each other's new value, leading the system to an incoherent state. LLC blocking does not help in this case, since the two write transactions can be spaced sufficiently apart so they do not overlap.

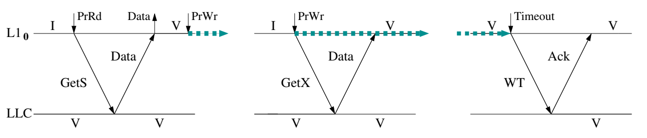


Figure 4: VIPS-M Read (PrRd), write (PrWr), and Delayed Write-through (WT at Timeout) transactions

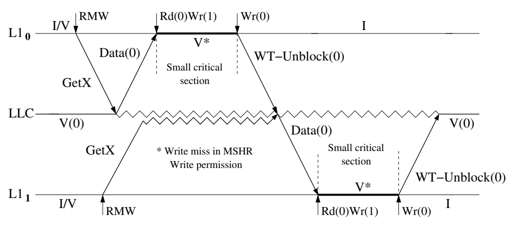


Figure 5: Atomic RMW transactions for shared lines

One solution would be to demand DRF guarantees at the cache-line level but that would place a heavy burden on software. Our solution is to perform write-throughs at a word granularity which has the additional benefit of reducing the amount of data transferred to the LLC.

Write-throughs at a word granularity require per-word dirty bits. This allows multiple concurrent writers on a cache line to write-through to the LLC just the words they modify but no other. Delayed write-throughs send their cache-line diffs which are then 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. We call this 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 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 (WT) followed by an acknowledgment (Figure 4, right side). If the line is already valid in the L1 (for example, as a result of a read –Figure 4, left side) an upgrade request (Upgr) is no longer needed, while from an invalid state, a GetX gets the data but does not block the LLC line (Figure 4, middle) which remains valid. Thus, shared-data write transactions become similar to private-data write transactions. Practically all data, whether shared (data-race-free) or private, are handled without any state in the LLC. The main 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. Synchronization data, however, still require a blocking protocol, described below in Section 3.3.

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.

3.3 Synchronization Without Invalidation

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 5 for a simple atomic instruction such as Test&Set. Regardless of the existence of an L1 copy, an atomic instruction invariably 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 5). 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 5. 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 can increase traffic, so an exponential back off in software is essential to lessen it.

3.4 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 [31]) 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 read–write epoch."*

Table 1: 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;	

A coherence protocol that satisfies this definition is invisible to the underlying memory consistency model [31]. 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.

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 invalidation). Similarly to MESI implementations [24], LLC blocking guarantees that values are propagated correctly.[2] It is therefore invisible to the memory consistency model, and thus, can support even the strictest model: SC.

Step 2: VIPS-M protocol. 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 1. 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.

However, for DRF operation, VIPS-M does satisfy the coherency definition: i) false sharing does not violate the single writer invariant because write-throughs happen at word granularity; thus there is only one writer per word 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. Note that on synchronization all outstanding write-throughs are sent immediately to the LLC and fence instructions must wait for the acknowledgments of the write-throughs for DRF lines to guarantee proper memory ordering.

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 to guarantee the Data Value invariant and this 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.

3.5 Optimizations

So far we have been in the process of de-evolving coherence: we removed most coherence state and the directory. It is therefore important that any optimization we introduce is neutral to complexity and cost. We consider only very simple optimizations that adhere to this principle.

[2]An old value cannot be accessed at the LLC if a write-through is in progress. This guarantees, for example, that only the new value of a write-through that started before a synchronization will be visible after the synchronization.

3.5.1 Classification Optimizations

Data classification is not the focus of this paper. We employ a very simple classification scheme that, nevertheless, allows our protocols to outperform more complex protocols. However, some classification optimizations are straightforward in our case.

Self-invalidation can cause needless misses on shared data that have not been modified. Complex techniques to exclude such data have been proposed [21]. 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 (e.g., accounting for thread migration to perform the page classification as suggested by Hardavellas *et al.* [15]).

3.5.2 Relaxing Inclusion Policies

In a MESI protocol, in which 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 all copies in L1 are invalidated to maintaining coherence. 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 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 LLC. 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 and System Configuration

The evaluation of the protocols proposed in this work is carried out with full-system simulation using Virtutech Simics [23] and the Wisconsin GEMS toolset [24]. The interconnection network is modeled using GARNET [5]. We simulate a 16-tile chip multiprocessor (CMP) architecture. The values of the main parameters used for the evaluation are shown in Table 2. 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.

Table 2: System parameters (between brackets: ranges for sensitivity analysis; in bold: main configuration)

Memory Parameters	
Processor frequency	3.0GHz
Block/Page size	64 bytes/4KB
MSHR size/Delay timeout	16 entries/1000 cycles
Split L1 I & D caches	[16KB, **32KB**, 64KB], 4-way
L1 cache hit time	1 (tag) and 2 (tag+data) cycles
Shared unified L2 cache	8MB, 512KB/tile, 16-way
L2 cache hit time	[**2**, 6] (tag) and [**4**, 12] (tag+data) cycles
Memory access time	160 cycles
Network Parameters	
Topology	2-dimensional mesh (4x4)
Routing technique	Deterministic X-Y
Flit size	16 bytes
Data message size	5 flits (if MWM not used)
Control message size	1 flit
Routing time	2 cycles
Switch time	2 cycles
Link time	2 cycles

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. MSHR timeout timers are implemented as cache decay hierarchical counters [18]. Cache latencies, energy consumption, and area requirements are calculated using the CACTI 6.5 tool [26] assuming a 32nm process technology. Likewise, the energy consumption overhead from extra L1 hardware (timers, selective flush support, etc.), is estimated to be a very small part (less than 1%) of the L1 energy.

We evaluate the five cache coherence protocols shown in Table 3. This table summarizes some of their characteristics. The first protocol (Hammer) corresponds to a broadcast-based protocol for unordered networks [28]. Since in this protocol invalidations are sent to all cores, there is no need for a directory. However, 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 LLC 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 relax the inclusion policy.

4.2 Benchmarks

We evaluate the five cache coherence protocols with a wide variety of parallel applications. *Barnes* (16K particles),

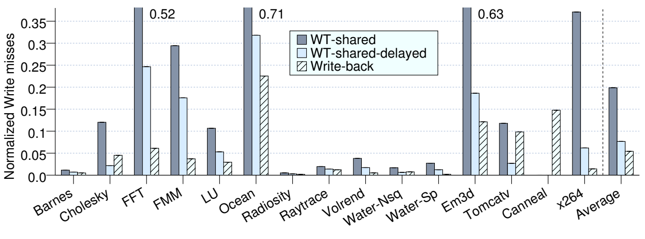


Figure 6: Reduction in write misses due to the private/shared classification and delayed write-through. Write misses have been normalized with respect to the simple write-through policy.

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), *Raytrace* (teapot, optimized version that removes unnecessary locks), *Volrend* (head), *Water-Nsq* (512 molecules) and *Water-Sp* (512 molecules) belong to the *SPLASH-2* benchmark suite [33]. *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. *Canneal* (simsmall) and *x264* (simsmall) are from the PARSEC benchmark suite [7].

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 flushing on their caches when required. Data accessed by atomic instructions follow the synchronization protocol described in Section 3.3. We simulate the entire applications, but collect statistics only from start to completion of their parallel part.

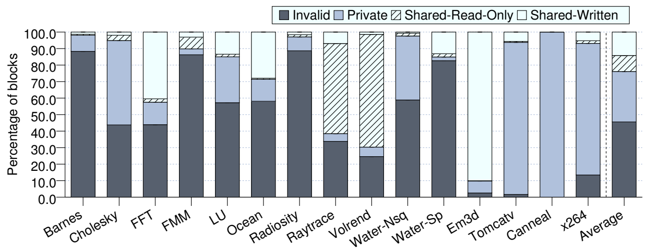
5. EXPERIMENTAL RESULTS

5.1 Impact on Write Misses

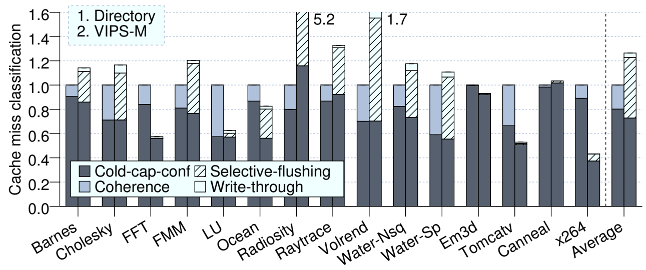
As discussed throughout this paper, the main drawback of a write-through policy is the significant amount of write misses it entails. In this section, we show the ability of both the private/shared classification and the delayed write-throughs to reduce this overhead. Figure 6 shows the number of write-misses 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 (WT-shared delayed), where the number of write misses is reduced by 90.4%, on average, compared to a write-through protocol, thus making this number close to the one obtained by a write-back policy (94.6%, on average).

Table 3: Evaluated protocols

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



(a) Lines found in the cache upon a selective flushing



(b) Cache misses normalized w.r.t. *Directory*

Figure 7: Impact of selective flushing

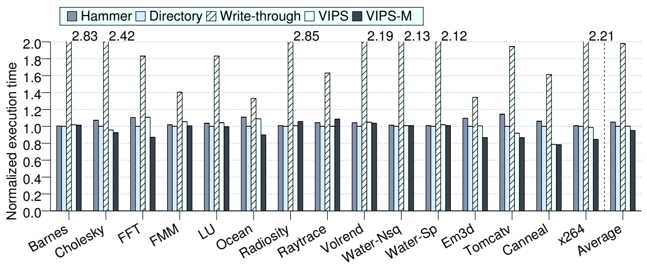


Figure 8: Normalized execution time w.r.t. *Directory*

5.2 Selective Flushing

VIPS-M 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 3.5.1). As shown in Figure 7a this selective flushing prevents about 73.9%, on average, of valid lines from being evicted from cache. This significantly lessens the number of misses as consequence of self-invalidations. We can also observe that 14.2% 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-through 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 so it can be competitive to directory invalidations. Figure 7b shows the L1 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 7a), 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 8 shows the applications' execution time for the five protocols evaluated in this work (see Table 3). The execution time has been normalized with respect to a directory protocol. The broadcast-based *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 VIPS protocol has similar performance to the complex directory protocol. In some applications it is faster, while in other ones slightly slower (but no more than 10%). VIPS-M, despite not having a directory structure at all, is faster in most applications (e.g., *Cholesky*, *FFT*, *Ocean*, *Em3d*, *Tomcatv*, *Canneal*, and *x264*, ranging from 7.7% to 21.5% faster). The exceptions being *Radiosity*, *Raytrace* (15.5% slower), and *Volrend* due to selective flushing. On average over the 15 applications, VIPS-M is 4.8% faster than Directory because of faster writes (no write-misses), faster reads (no directory indirection), and less traffic in the NoC.

Sensitivity Analysis. Figure 10a shows the average performance (over the 15 applications) of the two proposed protocols by varying the L1 cache size from 16KB to 64KB. The performance of VIPS with respect to MESI is not affected significantly by the cache size. However, VIPS-M flushes shared blocks from the L1 upon fence instructions and therefore, larger cache sizes impact its performance more. VIPS-

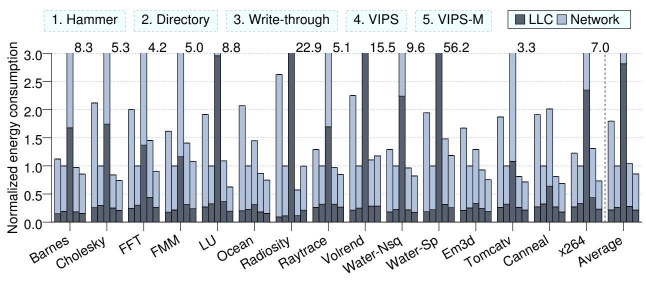
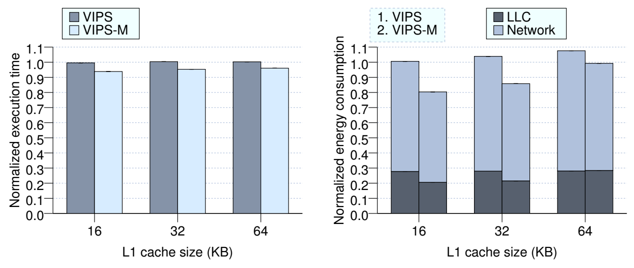


Figure 9: Normalized energy consumption w.r.t. *Directory*



(a) Performance 16KB–64KB L1 (b) Energy 16KB–64KB L1

Figure 10: Sensitivity analysis

M improvements still range from 6.2% (16KB), 4.8% (32KB), to 4.0% for the 64KB L1 cache. We have also studied larger sizes that would approximate private L1/L2 caches. VIPS becomes slightly slower than MESI for large private hierarchies (but less than 3% for 256KB). VIPS-M is still faster than MESI for 128KB caches, but becomes slightly slower (less than 2%) for 256KB caches. Tripling the latency of the LLC (to 12 cycles) does not affect the relative performance of VIPS and VIPS-M to MESI.

5.4 Energy Consumption

Figure 9 shows the energy consumed by the network and the LLC for the applications and protocols evaluated in this work for 32KB L1 caches. Broadcasting invalidations and receiving acknowledgments in *Hammer* leads to a significant increase in the energy consumed by the network (around 1.75× 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*, consuming less than a directory protocol for many applications (e.g., *Barnes*, *Cholesky*, *Radiosity*, *Em3d*, and *Tomcatv*). Finally, although the selective flushing in VIPS-M causes extra L1 misses, thanks to the Multiple-Writer-Merge protocol the amount of data written back into the LLC is significantly reduced, which translates into a reduction in dynamic consumption of 14.2%, on average, with respect to a directory protocol.

Sensitivity Analysis. Figure 10b shows the energy consumption of our protocols varying the cache size (from 16KB to 64KB) with respect to a directory protocol. Our protocols increase their traffic and accesses to the LLC relative to a directory protocol, for larger cache sizes. This is mainly due to two reasons: First, in VIPS and VIPS-M the rate of write-throughs is independent of the cache size (since they are always delayed the same) but in MESI, shared blocks stay longer in larger caches before they are written-back. Second, flushing in VIPS-M affects more lines in larger caches. This is why energy consumption (normalized to Directory) in VIPS-M increases faster than VIPS with cache size.

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. 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, –alternatively, 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 is forbidden. In the later case, coherence is extended across chips.

6.2 OS, Migration, and Multithreading

OS or thread migration from one core to another can (falsely) tag a private page as shared in our classification scheme. Better classification (possibly with reverse adaptation) or migration-aware classification by the OS itself, as described by Hardavellas *et al.* [15], alleviates these problems. Another issue is the flushing of shared data by the OS. In the implementation for this paper, context switches, system calls, I/O, thread migration, etc., conservatively flush all shared data. This can be optimized by tagging L1 lines as *system* or *user* lines. Similarly, tagging L1 lines with a thread ID and differentiating operations on them, avoids data classification or data flushing problems that may arise due to multithreading.

6.3 Debugging

VIPS-M can be incoherent in the presence of data races which may complicate debugging when a program is malfunctioning. Fortunately, a significant body of work on detecting data races already exists [30]. It is worth mentioning here some of the data-race debugging tools available that do not require any hardware support: *Velodrome*, *RecPlay*, *RaceTrack*, *FastTrack*, *RacerX*, *Eraser*, and *Goldilocks*, among others. With respect to debugging, we believe this is the way to go rather than relying on coherence being continuously present. Furthermore, eliminating the possibility of races at the programming model (e.g., as in DPJ or DRFx) would significantly help debugging.

6.4 Verification

VIPS has similar properties to MESI, only fewer states. Therefore it is easier to verify. In VIPS-M, the protocols for private and shared (DRF) data are nearly identical. There are only two stable states (Valid/Invalid) and two transient states (I–V and V–I) in the L1s and no states at all in the LLC (no blocking). Compared to standard protocols (e.g., MESI), there is a fundamental difference for the transient states: in VIPS-M there are no downgrades, invalidations, or data requests, so its transient states are invisible to the outside world.[3] They exist only in the MSHRs, and only to wait replies from the LLC. VIPS-M is thus simple to verify and significantly easier than MESI which requires extensive state exploration [11].

6.5 Speculation

Although we envision our coherence protocols as a better fit for simple cores, they can also be used to advantage with complex cores that support speculation and dynamic execution. In this case, a new possibility for optimization opens up. Selective flushing in VIPS-M leads to a small number of extra misses compared to a standard directory protocol. These misses are due to self-invalidated cache lines that are reused after synchronization, but without having been modified by any other core in the interim. With speculative execution mechanisms available, one can hide the latency of these misses by performing a Speculative Cache Lookup (SCL) and applying Coherence Decoupling as proposed by Huh *et al.* [17] on top of VIPS-M. Speculative cache lookups in VIPS-M, allow the cores to continue execution with self-invalidated data while the decoupled protocol reloads invalid cache lines with their latest version from the LLC. If the self-invalidated and LLC data differ (i.e., the line has been modified), speculation is squashed. In this case, the cost is just the wasted speculative execution. If, on the other hand, the data are the same (i.e., the line has not been modified), the speculation is verified and the latency of going to the LLC is completely hidden. The benefit of this optimization is obvious: it completely hides any performance penalty from selective flushing, but at the expense of some extra, discarded, speculative execution.

7. RELATED WORK

We have already discussed in passing most of the related work. Here, we will just summarize the similarities and differences with the work closest to ours, the DeNovo work of Choi *et al.* [11]. Relying on disciplined parallelism, Choi *et al.* take a similar approach to simplify coherence. However, in contrast to our approach, they require 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.

[3]With respect to verification this makes the treatment of the delayed write-throughs analogous to that of write buffers.

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, and therefore, directory indirection, 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, as proposed by Kaxiras and Keramidas, and revert to the registry on mispredictions. This adds complexity and cost. Finally, although Choi *et al.* simplify coherence for DRF, they do not address the issue of synchronization races.

The heavy reliance of the DeNovo approach on application involvement –and 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– prevents us from replicating their results for a direct comparison. By inspecting the published results, however, we can make the following observations: i) both approaches are competitive to MESI; ii) DeNovo performs well, 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. Overall, we consider the results of 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 complexity, and takes us back to before the IBM centralized directory, Censier and Feautrier’s distributed directory [8] or Goodman’s write-once protocol [14]. Inspired by efforts to simplify coherence and/or reduce directory cost [11, 12, 13, 16, 19] we go a step further and completely eliminate the need for a directory. In a multicore cache hierarchy, as long as we treat private and shared data separately, we can improve both performance and energy consumption 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 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.

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 just a simple rule of when to write data back to the LLC. The end result outperforms more complex protocols while at the same time consumes less energy. Data races, however, are still pertinent to synchronization operations. In our proposal, we maintain the memory semantics of atomic instructions with a simple synchronization protocol that does not need a directory or invalidations.

9. ACKNOWLEDGMENTS

This work is supported, in part, by the Swedish Research Council UPMARC Linnaeus Centre and the EU HEAP Project FP7-ICT-247615.

10. 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] S. Bell, et al. TILE64^{TM} processor: A 64-core SoC with mesh interconnect. In *IEEE Int'l Solid-State Circuits Conference (ISSCC)*, pages 88–598, Jan. 2008.

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

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

[9] 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.

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

[11] B. Choi, R. Komuravelli, and H. Sung, et al. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sept. 2011.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: Making use of incoherence. In *11th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, pages 97–106, Oct. 2004.

[18] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *28th Int'l Symp. on Computer Architecture (ISCA)*, pages 240–251, June 2001.

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

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

[21] 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.

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

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

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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 Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 465–476, Sept. 2010.

[30] A. Raza. A review of race detection mechanisms. In *1st International Computer Science Conference on Theory and Applications*, pages 534–543, June 2006.

[31] 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.

[32] 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.

[33] 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.

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