EFFICIENT, SNOOPLESS, SYSTEM-ON-CHIP COHERENCE

Stefanos Kaxiras[1] and Alberto Ros[2]

[1]Uppsala University, Sweden [2]University of Murcia, Spain

stefanos.kaxiras@it.uu.se, aros@ditec.um.es

ABSTRACT

Coherence in a System-on-Chip (SoC) introduces complexity and overhead (snooping caches/directory, state bits, invalidations, etc.) in exchange for a clean and uniform shared memory model. As it is typical today, a SoC comprises a variety of cores with local caches, accelerators with local memories, and some form of shared last-level cache (LLC), all interconnected with shared buses. We propose a very simple coherence protocol, fit for this environment, that eliminates L1 snooping and its associated complexity and costs (power). In essence, we remove all coherence decisions from local caches by simply determining at the LLC whether data are private or shared. This makes a write-through policy a practical and effective alternative to maintain coherence. In the local caches, we dynamically select between write-back for private data, or write-through for shared data. Self-invalidation of the shared data on synchronization points eliminates the need to snoop, with just a data-race-free guarantee from software. Our evaluation shows that this simple protocol outperforms a traditional snooping protocol while at the same time significantly reducing L1, shared cache, and bus energy consumption.

I. INTRODUCTION

While the benefits of a coherent memory system, especially for software development, are well known, there is significant concern over its cost, especially for cost-sensitive markets such as most Systems-on-Chip (SoCs). More importantly, concerns center on the constant overhead of coherence which, however, may be desirable only sporadically. It is not surprising that significant effort has been expended to reduce these costs, especially storage cost for directory protocols [2, 8, 9, 24], verification cost [1,10, 27], or coherence traffic and snooping cost (by reducing or filtering coherence) [14, 21, 29].

We take an alternate approach where there is practically no cost for when coherence is not needed and minimal cost when it is needed. We achieve this by eliminating the need for snoops; indeed, we eliminate all distributed coherence state in the caches (besides the rudimentary valid/invalid and clean/dirty states). Our approach exploits a typical SoC cache hierarchy organization with many local L1 caches and a shared last-level cache (LLC).[1]

Our proposal targets multicore/manycore[2] and SoC 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]), but also GPUs coupled to general purpose cores as most smartphone and tablet processors are today. We do not envision our proposal for multicores based on few fat complex cores, where the relative cost of implementing a snooping or directory protocol is not an issue, and where such protocols are routinely implemented.

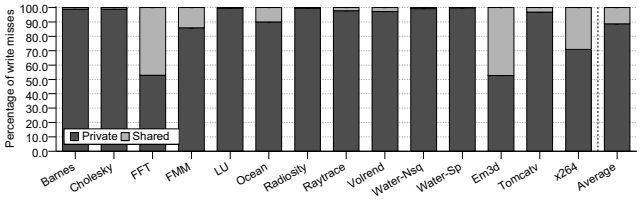

Figure 1. Percentage of write misses in a write-through protocol for private and shared data (cache-line granularity).

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 switch, or a packet-based, point-to-point, network-on-chip (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 this paper however, we focus on shared bus implementations and snooping coherence because they dominate the SoC implementations. A significant advantage of our approach for bus-based systems is that it allows easy scaling to multiple buses. (e.g., connecting cores with address-interleaved buses to multiple LLC banks), since there is no need to snoop on any bus.

Contributions:

1. We eliminate the need to constantly snoop global traffic just to track sharing information distributively at the L1s. Instead, sharing information is tracked at the LLC which naturally observes the request traffic from all the L1s. At the LLC we just determine whether data are private (cached in a single L1) or shared (cached in several L1s). We do not keep full directory information.
2. The benefit of a write-through policy is that it reduces the coherence protocol down to two states: Valid/Invalid. However, a simple write-through policy to the LLC would seriously compromise performance. The observation, that drives our approach is that most write misses in a write-though protocol actually come from private blocks, as Figure 1 shows. Motivated by this observation, we propose a **dynamic write policy** in the L1s: we dynamically select between write-back and write-through to the LLC depending on whether data are private or shared, according to the LLC classification. Write-throughs are performed by transferring only what is modified (*diffs*) in a cache line, allowing multiple simultaneous writers per line (as long as their writes constitute false sharing on separate words and not a true data race).
3. We **selectively flush** shared data from the L1 caches on synchronization points. Our approach is a generalization of *read-only* tear-off copies [14, 17] to all shared data (including written copies).

[1] Our approach extends to deeper hierarchies (e.g., with an L2 between the L1 and the LLC), but for clarity, in this paper, we will only discuss a two-level cache hierarchy.

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

978-1-4673-1295-0/12/$31.00 ©2012 IEEE

This step eliminates the need to snoop for invalidations.

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 classification of data to shared and private at a cache-line granularity. This leads to the main novel result that we report in this paper: a minimal coherence scheme that: i) eliminates snooping ii) uses very simple protocols for both shared (data-race-free) data and for private data, differentiating only in the timing of when to put them back in the LLC, iii) reduces energy consumption, iv) does not require application involvement, and v) allows effortless scaling with multiple buses.

There are two implications of our approach. First, our protocol does not support sequential consistency (SC) for data races. This is because without snoops, 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 [26]. 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. [10]. Thus, similarly to SC for DRF [3], our approach provides coherency for DRF.

The second implication is that, synchronization instructions (such as Test&Set, Compare&Swap, or Load-Linked/Store-Conditional) which inherently rely on races, require their own protocol for a correct implementation. We propose an efficient and resource-friendly synchronization protocol that works without snoops and in many cases eliminates spinning.

Results. Our approach leads to a very simple coherence protocol that requires no state bits in the caches (other than the standard Valid/Invalid and Dirty/Clean bits), no snoops for reads and writes, and can lead to significant energy savings with respect to a standard snooping protocol.

II. BACKGROUND AND RELATED WORK

A. Write-Through Caches and Coherence

A write-through policy for L1 caches has the potential to greatly simplify the coherence protocol [26]. Just two states are needed in the L1 cache (valid and 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 L1s do not need to snoop reads or writes on the bus to supply the latest data. However, invalidation is still required. Unfortunately, because the number of write-throughs far exceeds the number of write-backs, this results in abysmal performance degradation, and significantly increased traffic and power, as we show in Section VII.

B. Private vs. Shared Data Classification

Recent work realizes the importance of classifying private and shared data in hardware [25, 13], by the operating system [12, 16, 11], or by the compiler [18]. The advantage of hardware mechanisms is that they can work at a cache-line granularity, although the storage overhead can be prohibitive. To considerably reduce this overhead, we propose a hardware classification mechanism in the LLC that limits the storage to only the cached lines.

C. Self-Invalidation

Dynamic Self invalidation and tear-off copies were first proposed by Lebeck and Wood as a way to reduce invalidations in directory-based cc-NUMA systems [17]. 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 which 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).

Self-invalidation was recently used by Kaxiras and Keramidas in their “SARC Coherence” proposal [14]. They observed that with self-invalidation, writer prediction becomes straightforward to implement. Subsequently, Choi et al. took a similar approach to simplify coherence in their DeNovo approach, but relied on application-driven directory protocols [10]. 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. The heavy reliance of the DeNovo approach on application involvement prevents us from replicating their results for a direct comparison. In our work, for the first time, we use self invalidation to eliminate the need for L1s to snoop writes on a bus.

IV. PROTOCOL

Our approach boils down to three steps:

1. Classify cache lines in the LLC as private or shared.
2. Simplify coherence by dynamically using write-through for shared data. (1) and (2) together *eliminate snoops on reads*.
3. Eliminate *snoops on writes (invalidation)* by selective flushing of the shared data.

LLC Data Classification. The centerpiece of our strategy for reducing the complexity of coherence and eliminating snoops is to distinguish between private and shared data. For the coherence of the shared data, we rely on the simplicity of a write-through policy. However, private data employ a write-back policy, which can be safely implemented without any further coherence support.

Cache lines are divided into “Private” and “Shared” by the LLC, depending on the observed accesses. The Private/Shared (P/S) classification in the LLC is carried back to the L1s with the LLC responses. Each line in the LLC is tagged with a P/S bit and (if it is private) the ID of the L1 that “owns” it. An LLC line is private when all accesses to it come from the same L1. If we detect, a different L1 accessing a private line (i.e., the requestor is different from the current owner), we change the line to shared. Before the LCC responds to the new requestor, the former private owner must change its classification of the line.[1] We achieve this,

by forcing a snoop in the former owner. As a result of changing classification, the former owner may perform a write-back of the line (if dirty). Our classification is one way (private to shared) for simplicity.

The P/S bit and private owner field exist only for the LLC lines —they are not saved externally and are lost upon eviction. This creates the problem of the initial state when we bring a cache line into the LLC. While there are different design points depending on whether the SoC hierarchy is inclusive or non-inclusive, here —for simplicity— we assume an inclusive hierarchy. When we evict an LLC line, a *forced* snoop in all the L1s (if shared) or in the owner L1 (if private), also flushes the L1 copies. The solution for a non-inclusive hierarchy is to establish which L1 (if any) has the line when we bring a line in the LLC. This forces snoops on all the L1s.

Dynamic Write Policy. The LLC classification determines the write policy in the L1. On an L1 miss the sharing status of a line is unknown. The miss accesses the LLC and the outcome of the classification (Private or Shared) is carried back to the L1 with the response. The write policy is set to write-back (on eviction) for private lines, and write-through for shared lines.

Because in the L1 we have only two states (Valid or Invalid) and we differentiate between Private and Shared, we call the overall family of protocols VIPS. There is also a standard dirty bit (Dirty/Clean) that is used by the write policies. The resulting protocol is compatible with the protocol states defined in ARM AMBA-4 AXI Coherency Extensions (ACE) [5], as shown in Table 1.

Table 1. VIPS Protocol States

AMBA-4 ACE state	MOESI state	VIPS V/I – P/S – D/C			Comments
UniqueDirty	M	V	P	D	Use Write-Back
SharedDirty	O	V	S	D	***TRANSIENT***: this state is transient in our protocol and invisible to the outside (see Delayed Write-Throughs)
UniqueClean	E	V	P	C	Use Write-Back
SharedClean	S	V	S	C	Use Write-Through
Invalid	I	I	—	—	—

We will just dwell on single point: our proposal is minimally intrusive in the design of the core, L1 caches, and the LLC. In fact, it leaves standard L1 caches unmodified. We assume that each L1 line has the common Valid/Invalid (V/I) and Dirty/Clean (D/C) bits. An additional Private/Shared bit (P/S) bit is needed to store the LLC classification. (Equivalently, the ARM AMBA-4 ACE states can be used.) The P/S bit controls whether a write-through will take place. The dynamic write policy (as well as the P/S bit) can implemented outside the L1 using standard cache management functions ("clean line" or "write-back" [5]).

Delayed Write-Throughs. The `SharedDirty` ACE state (Table 1) does not exist in our protocols as a *stable* state. A write-through policy would immediately clean the line to a `SharedClean` state. However, the obvious optimization to any write-through cache is to reduce the amount of write-throughs by coalescing as many writes as possible. In VIPS, the write-through can be delayed (up to the first synchronization point). In the meantime, the line can be written multiple times by the same core. This corresponds to the ACE `SharedDirty` state (or MOESI "Owned" state), but in our case it is strictly transient. It exists only from the write miss to the write-through and is *invisible* to transactions from other cores —therefore introducing no complexity to the protocol.

One simple implementation of the delayed write-through is to insert the address of the line in one of the core's miss-status holding registers (MSHRs). Each entry is associated with a timer which causes the write-through to happen a fixed delay after the initial write. The write-through also occurs if the entry is replaced or the MSHRs is flushed.

Invalidation and Snoops on Writes. Our next goal is to eliminate invalidations, i.e., the snoops on writes. We have already removed the need for read snooping with the private-shared classification at the LLC and the write-through policy for shared data. What is left is to get rid of the need to snoop on writes just to invalidate possible L1 copies. Self-invalidation serves exactly this purpose [17]. Readers, after making a "*tear-off*" copy of a memory location, are allowed to ignore any write traffic, as long as they promise to invalidate their copy, *on their own*, 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 [17] and [14]— are tear-off copies. A core encountering a synchronization point (lock acquire/release, barrier, wait/signal synchronization, Load-Linked/Store-Conditional instructions, or memory ordering instructions such as ARM Data Memory Barrier —DMB— or Data Synchronization Barrier —DSB) 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 the per-line P/S bits, which are set, according to the LLC classification, 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 with selectively clearable valid bits [23]. As is pointed out in prior work [17,14,10], self-invalidation, and by extension selective flushing, implies a weak consistency memory model and only guarantees SC for DRF programs [3]. Note that there is always the option of using invalidations (snooping on writes) in which case a sequential consistent memory model is supported.

Read-only optimization. Self-invalidation can cause needless misses on shared data that have not been modified. Complex techniques to exclude such data have been previously proposed [17]. In our approach, we simply tag at the LLC shared cache lines as Read-Only (RO) if they are not written, and Read-Write (RW) otherwise. A line stars as RO but transitions to RW on the first write (there is no reverse transition —except through eviction). Because the line is shared, all the L1s must be notified of this change with a forced snoop. L1 RO cache lines are spared from self-invalidation.

Multiple Writers & Merge. Even with a DRF guarantee the lack of invalidations can cause problems. Consider two concurrent readers, each

[1] Fortunately, this concerns a single, known, L1 (since it was the *private* owner).

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. 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.

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) write-through policy. Synchronization data, however, still require a blocking protocol, described below in Section E.

The overhead 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 the outstanding delayed write-throughs and are attached only to those. No additional support is needed in the L1 or the LLC —other than being able to update individual words.

E. Synchronization Without Invalidation

Synchronization relies on data races. Instructions such as Test&Set or Compare&Swap, race to read-modify-write (RMW) 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.

To break this impasse we treat synchronization accesses differently. Synchronization instructions bypass the L1 and send their requests to the LLC. The LLC *blocks* the corresponding lines accordingly.

Regardless of the existence of an L1 copy, atomic RMW instructions (as well as load-linked/store-conditional instructions) invariably send a new request to the LLC. For the atomic RMW instructions, if the line is *unblocked* (or “*open*”) in the LLC, 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* (or “*exclusive*”) by the LLC controller; it is only unblocked by an ensuing write-through. In the interim no other core can complete any transaction on that line (as core 1 in Figure 2).

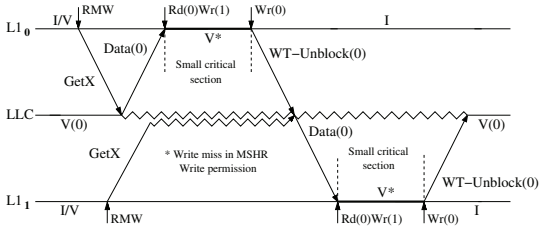

Figure 2. Atomic RMW transactions for shared lines

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 expensive. However, by delaying the write-throughs of atomic instructions, we can eliminate spinning by blocking other cores at the LLC controller. For a short critical section, the write-through of a Test&Set can be delayed for the whole duration of the critical section as shown in Figure 2. In longer critical sections the write-through eventually completes and spinning resumes by other cores. In this case, an exponential back off in software can be used to lessen it.

To maintain the semantics of the load-linked/store-conditional instructions we implement them differently than atomic RMW instructions. A load-linked instruction, blocks the LLC line and sets the private owner of the line to the issuing core. A store-conditional from the same core succeeds (and *unblocks* the line) only if the private owner field remains unchanged. Any other interfering core changes the private owner field thus causing the store-conditional to fail.

VI. EVALUATION METHODOLOGY

The evaluation of the protocols proposed in this work is carried out with full-system simulation using Virtutech Simics [19] and the Wisconsin GEMS toolset [20]. We account for the variability in multi-threaded workloads by doing multiple simulation runs for each benchmark and injecting small random perturbations in the timing of the memory system [4]. The variations are reflected in all the graphs with error bars.

The values of the main parameters used for the evaluation are shown in Table 2. Through experimentation we have found that only 16 MSHRs, with a timeout between 500 and 2000 cycles, are needed to keep shared data as dirty for enough time to avoid most write misses. MSHR timeout timers are cheaply implemented as cache decay hierarchical counters [15]. Cache latencies, energy consumption, and area requirements are calculated using the CACTI 6.5 tool [22] assuming a 32nm process technology.

Benchmarks. We evaluate the described protocols with a wide variety of shared-memory parallel applications that require coherence. Barnes (16K particles), Cholesky (tk16), FFT (64K complex doubles), FMM (16K particles), LU (512x512 matrix), Ocean (514x514 ocean), Radiosity (room, -ae 5000.0 -en 0.050 -bf 0.10), Raytrace (teapot), Volrend (head), Water-Nsq (512 molecules), and Water-Sp (512 molecules) belong to the SPLASH-2 benchmark suite [28]. 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

Table 2. System Parameters

Processor/Memory Parameters	
Processor Frequency & no. of cores:	3GHz, 16 cores
Split L1 I & D caches:	32K, 4–way
Cache Block:	64
MSHR size / timeout:	16 entries, 1000 cycles
L1 cache hit time:	1 (tag) and 2 (tag+data) cycles
Shared unified LLC cache:	8MB, 16-banks (512KB/bank), 16-way/bank
L2 bank hit time:	2 (tag) and 4 (tag+data) cycles
Off-chip access time:	160 cycles

(simsmall) is from the PARSEC benchmark suite [7]. Data accessed by atomic instructions follow the synchronization protocol described in Section E. We simulate the entire applications, but collect statistics only from start to completion of their parallel part.

VII. EXPERIMENTAL RESULTS

A. Impact on Snoops

Figure 3 shows the total number of *external* (non-core) L1 tag accesses. The accesses correspond to snoops (top bars) and LLC data responses to misses (bottom, darker bars). VIPS completely eliminates *coherence* snoops, but introduces a new kind of snoops: *forced* snoops. These occur in three cases. The first is when the LLC changes the classification of a line from private to shared. The second is when the LLC changes the classification of a line from Read-Only to Read-Write. The third is the eviction from the LLC, which also evicts the L1 copies. Forced snoops affect a single L1 in case of a private block, or all L1s in case of a shared block. The net result is shown in Figure 3: in VIPS, forced snoops are (on average) less than 0.3% of the coherence snoops, but LLC responses are slightly increased due to selective flushing (see below). VIPS saves (on average) 87% of the external (non-core) L1 tag accesses far exceeding state-of-the-art snoop filtering such as Jetty [21].

B. 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 that are being shared among different cores and modified by at least one of the cores. As shown in Figure 4 (top) this selective flushing prevents about 68.7% on average of valid lines from being evicted from the cache. This significantly lessens the number of misses as consequence of self-invalidations. We can also observe that 19% 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). 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 invalidations. Figure 4 (bottom) shows the misses in a MESI protocol and in VIPS classified by the event that caused them. The percentage of cold, capacity, and conflict misses (Cold-cap-conf), slightly decreases in VIPS 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

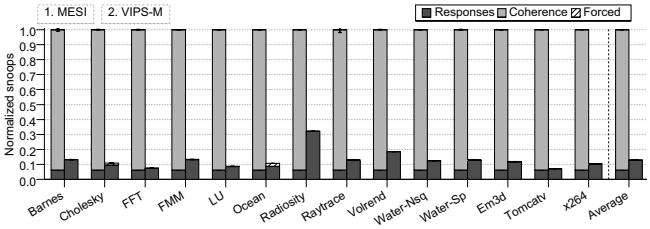

Figure 3. Reduction in snoops.

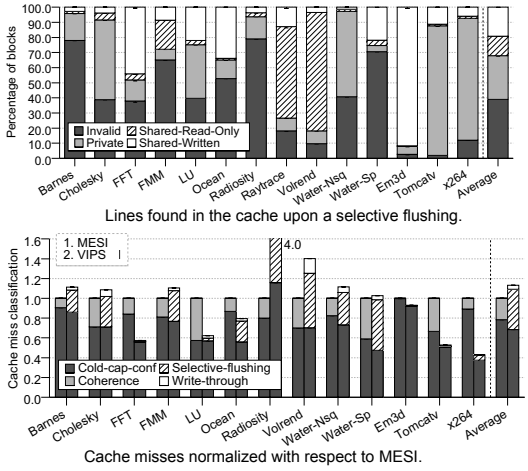

Figure 4. Impact of Selective Flushing.

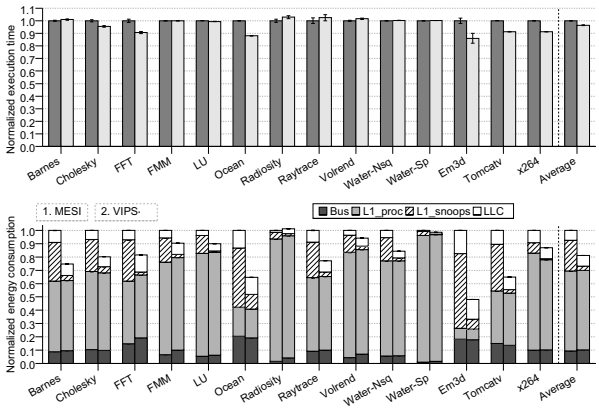

Figure 5. Normalized execution time and energy

this is because they have only a few barriers, so 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 4–top), 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.

C. Performance

There are two performance implications in our approach: The first, positive implication, is that in contrast to other snooping protocols, we have no snoops, thus no contention in the L1s from the bus,

competing with the core. Alternatively, for snooping protocols we would need to have dual tags or multi-ported tags, which would increase cost. The second, negative implication, is that read misses are always delegated to the LLC and cannot be satisfied by cache-to-cache transfers. Cache-to-cache transfers make sense when access to the next hierarchy level is very expensive. However, in a SoC cache hierarchy the LCC is relatively close. Furthermore, our results confirm that only a small percentage of read misses (on average 3.9% in MESI and 7.9% in MOESI) can be satisfied by cache-to-cache transfers, so the benefit would be minor. Figure 5 top shows the execution time (normalized to MESI). On average, our approach is 3.5% faster than MESI.

D. Energy Consumption

Figure 5 bottom, shows the normalized energy consumption of VIPS compared to MESI. We concentrate on bus and cache hierarchy energy (derived from Cacti and GEMS). For the cache hierarchy we show the energy spent on L1 accesses by the core, external (non-core) L1 tag accesses, and LLC accesses. Our results show that VIPS can be significantly more energy efficient. The external (non-core) L1 tag-access energy is reduced to less than 15% of the corresponding MESI energy. Variations in the number of misses (due to selective flushing) and write-through traffic occasionally increase bus and LLC energy slightly, but in some cases (Cholesky, Ocean, Water-NSQ, Water-Sp, Em3d, Tomcatv) VIPS is as efficient or even more efficient than MESI (for the bus and LLC). Overall, VIPS saves close to 20% of the energy in the on-chip memory system.

V. CONCLUSIONS

We propose a novel coherence approach, for SoC memory hierarchies, based on an efficient LLC classification of data (to private and shared), a dynamic write policy based on this classification (write-through for shared and write-back for private data), and selective flushing of shared data from the L1 caches upon synchronization. By separating private from shared data at the cache-line level at the LLC, we minimize the impact of the write-through policy, since many of the write-misses are due to private data. We achieve this by eschewing support for data races, which are considered harmful [10] and can be eliminated in malfunctioning programs using data-race detection approaches [24]. For synchronization, however, which does exhibit data-race behavior, we provide efficient extensions to our protocol. Our approach has several distinct advantages:

- eliminates all read and write snoops (leaving just a few *LLC-forced* snoops for classification purposes)
- incurs practically *no overhead* (performance or power) when coherence is not needed (e.g., for throughput workloads)
- improves performance and energy over standard snooping protocols for workloads that require coherence
- allows seamless scaling to multiple buses (since we do not need to snoop on any of them) and easily extends to packet-based networks-on-chip.

VI. ACKNOWLEDGMENTS

This work is supported in part, by the Swedish Research Council UPMARC Linnaeus Centre, the Spanish MICINN under grant TIN2009-14475-C04-02, and the EU HEAP Project FP7-ICT- 247615.

VII. 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, et al. A new scalable directory architecture for large-scale multiprocessors. In 7th HPCA, Jan. 2001.
[3] S. V. Adve, M. D. Hill, Weak ordering—a new definition, 17th ISCA, May 1990.
[4] A. R. Alameldeen, D. A. Wood "Variability in Architectural Simulations of Multi-threaded Workloads" In 9th HPCA, 2003.
[5] ARM AMBA-4 AXI Coherency Extensions, ARM 2011.
[6] Shane Bell et al. TILE64 Processor: A 64-Core SoC with Mesh Interconnect, IEEE Int'l Solid-State Circuits Conf., 2008.
[7] C. Bienia, et al. The PARSEC benchmark suite: Characterization and architectural implications. In 17th Int'l PACT, Oct. 2008.
[8] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In 4th ASPLOS, Apr. 1991.
[9] G. Chen. Slid - a cost-effective and scalable limited-directory scheme for cache coherence. In 5th PARLE, June 1993.
[10] B. Choi, et al. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In 20th PACT, Sept. 2011.
[11] B. Cuesta, et al. Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. ISCA 38, 2011.
[12] N. Hardavellas, et al. Reactive NUCA: Near-optimal block placement and replication in distributed caches. In 36th ISCA, June 2009.
[13] H. Hossain, et al. POPS: Coherence protocol optimization for both private and shared data. In 20th PACT, Sept. 2011.
[14] S. Kaxiras and G. Keramidas. SARC coherence: Scaling directory cache coherence. IEEE Micro, 30(5):54–65, Sept. 2011.
[15] S. Kaxiras, et al. Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power. ISCA 2001.
[16] D. Kim, J. A. J. Kim, and J. Huh. Subspace snooping: Filtering snoops with operating system support. In 19th PACT, Sept. 2010.
[17] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In 22nd ISCA, June 1995.
[18] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones. Compiler-assisted data distribution for chip multiprocessors. In 19th PACT, Sept. 2010.
[19] P. S. Magnusson, et al. Simics: A full system simulation platform. IEEE Computer, 35(2):50–58, Feb. 2002.
[20] M. M. Martin, D. J. Sorin, and B. M. Beckmann, et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. Computer Architecture News, Sept. 2005.
[21] A. Moshovos, et al. JETTY: Filtering snoops for reduced energy consumption in SMP servers. In 7th HPCA, Jan. 2001.
[22] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Cacti 6.0. Technical Report HPL-2009-85, HP Labs, Apr. 2009.
[23] Tadashi Okamoto et al. Memory device having valid bit storage units to be reset in batch. US Patent 4879687.
[24] B. W. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In 17th ISCA, June 1990.
[25] S. H. Pugsley, et al. SWEL: Hardware cache coherence protocols to map shared data onto shared caches. In 19th PACT, Sept. 2010.
[26] D. J. Sorin, M. D. Hill, and D. A. Wood. A Primer on Memory Consistency and Cache Coherence, Synthesis Lectures on Computer Architecture. Morgan & Claypool Pub., May 2011.
[27] D. Vantrease, M. H. Lipasti, and N. Binkert. Atomic coherence: Leveraging nanophotonics to build race-free cache coherence protocols. In 17th HPCA, Feb. 2011.
[28] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In 22nd ISCA, June 1995.
[29] H. Zhao, et al. SPACE: Sharing pattern-based directory coherence for multicore scalability. In 19th PACT, Sept. 2010.