A Hybrid Static-Dynamic Classification for Dual-Consistency Cache Coherence

Alberto Ros and Alexandra Jimborean

Abstract—Traditional cache coherence protocols manage all memory accesses equally and ensure the strongest memory model, namely, sequential consistency. Recent cache coherence protocols based on self-invalidation advocate for the model sequential consistency for data-race-free, which enables powerful optimizations for race-free code. However, for racy code these cache coherence protocols provide sub-optimal performance compared to traditional protocols.
This paper proposes SPEL++, a dual-consistency cache coherence protocol that supports two execution modes: a traditional sequential-consistent protocol and a protocol that provides weak consistency (or sequential consistency for data-race-free). SPEL++ exploits a static-dynamic hybrid classification of memory accesses based on (i) a compile-time identification of extended data-race-free code regions for OpenMP applications and (ii) a runtime classification of accesses based on the operating system's memory page management. By executing racy code under the sequential-consistent protocol and race-free code under the cache coherence protocol that provides sequential consistency for data-race-free, the end result is an efficient execution of the applications while still providing sequential consistency. Compared to a traditional protocol, we show improvements in performance from 19% to 38% and reductions in energy consumption from 47% to 53%, on average for different benchmark suites, on a 64-core chip multiprocessor.

Index Terms—Multiprocessors, cache coherence, classification of accesses, runtime, compiler, consistency model, data races.

1 Introduction

Given the prevalence of multi-core processors and the trend of continuously increasing the number of cores, scalability and efficiency of coherence protocols becomes crucial for performance. State-of-the-art coherence protocols seek to deliver scalability, performance and energy efficiency [1] by detecting and exploiting memory accessing characteristics of code with the goal of simplifying the coherence protocol [2], [3]. Despite the promising results, one of the main limitations of such protocols is that they fail at providing support for legacy code, which may prevent them from being integrated in commercial processors. Under these circumstances, emerging architectures (*e.g.*, Intel Xeon Phi [4]) still implement traditional directory-based cache coherence protocols, despite they perform sub-optimally on modern architectures. Furthermore, the inefficiency of traditional protocols increases with the number of cores in the system.

One source of inefficiency is that traditional coherence protocols provide the strongest consistency model (or memory model), namely sequential consistency (SC) [5]. While this design decision eases the development of the protocol by isolating the cache coherence protocol from the consistency model of the hardware [6], it brings significant performance penalties. The shortcomings become unjustified especially when the system provides a more relaxed consistency model [7]. In answer, recently proposed cache coherence protocols follow the sequential consistency for data-race-free (SC-for-DRF) model [8], which allows a simpler and more scalable design [2], [3] and improves performance [9]. However, despite the advantages of these cache coherence protocols, based on self-invalidating the cache content on synchronization points, they still have two main drawbacks: (i) they are not efficient when executing racy code, *e.g.*, when synchronizing threads, due to intense self-invalidation and (ii) they do not provide backwards compatibility with existing software that requires a consistency model stronger than SC-for-DRF. Although recent efforts have been done to optimize racy code in SC-for-DRF protocols [10], [11], they rely on exposing synchronization to the hardware, thus not providing backwards compatibility with existing software.

Another source of inefficiency of traditional protocols stems from not taking advantage of applications' behavior, thus missing potential performance improvements. To exploit this opportunity, numerous proposals revolve around identifying the nature of memory accesses as private or shared [12], [13], [14], [15], [16], [17], [18], [19], [20], [21] for optimizing the coherence protocol or, for example, enhancing data placement. While the optimization is efficient and enables novel coherence protocols to outperform traditional protocols, the underlying techniques for classifying accesses still lack accuracy (Section 7), due to the classification of memory accesses *based on the private or shared nature of the target data*. Performed at runtime, such a classification is either coarse-grained [12], [15], [16] or increases hardware complexity [14], [17], [21]. Performed at compile-time, it must conservatively handle memory accesses which cannot be fully disambiguated statically [13], [18].

This work proposes SPEL++, which stands for **S**calability, **P**erformance, **E**nergy efficiency and support for **L**egacy code. SPEL++ is a dual-consistency cache coherence protocol that supports two execution modes: a traditional SC protocol and a protocol that provides SC-for-DRF. SPEL++ actively selects a more relaxed SC-for-DRF protocol during the execution of data-race-free (DRF) code regions [22]. By relaxing the protocol SPEL++ achieves *high performance and scalability*, by executing racy code with an SC protocol SPEL++ achieves *energy efficiency*, and by delivering SC —due to the guarantee that only DRF code is executed with the SC-for-DRF protocol— it *ensures compatibility*

- *A. Ros is with the Computer Engineering Department, University of Murcia, 30100 Murcia, Spain.*
  E-mail: aros@ditec.um.es
- *A. Jimborean is with the Department of Information Technology, Uppsala University, 751 05 Uppsala, Sweden.*
  E-mail: alexandra.jimborean@it.uu.se

with legacy software.

Key to these properties is a hybrid static-dynamic classification of memory accesses. First, we perform a compile-time identification of *extended data-race-free (xDRF)* code regions (see section 2.1), where each xDRF region consists of a set of DRF regions. We complement the static classification per regions with a dynamic private-shared classification of memory accesses, by resorting to the operating system's memory management (OS-based). The hybrid classification entails three protocol modes that are designed to efficiently handle each memory access: (i) OS-private memory accesses require minimum coherence support and are highly optimized, (ii) accesses performed within xDRF regions can be executed under a high-performance and scalable SC-for-DRF protocol, while providing SC, and finally (iii) for accesses that are neither OS-private nor part of xDRF regions, coherence is ensured by a standard directory protocol (SC protocol), which is commonly optimized for executing racy code. The proposed design smoothly blends the three protocol modes, which can be simultaneously active: for instance, threads executing non-DRF regions with OS-shared accesses follow the SC protocol, while threads executing xDRF regions follow the SC-for-DRF protocol. Cache blocks transition from one protocol mode to the other on demand, thus maximizing performance (Section 3).

The compile-time classification differs from previous static classification schemes, which focus on classifying memory accesses as private or shared based on the nature of the accessed data. In this paper we target codes in which the compiler can unequivocally identify xDRF regions. Classes of codes amenable to compile-time identification of xDRF regions include both (1) already parallel applications and (2) sequential codes automatically parallelized at compile time. The first category refers to parallel codes with OpenMP annotations. We exemplify the second category with sequential codes that are statically analyzable and amenable to parallelization by applying polyhedral transformations [23], [24]. The compile-time classification is complemented by a standard OS-based classification (see Section 7) to increase the accuracy. The hybrid classification scheme represents a step forward in solving legacy issues automatically. Legacy codes which are not amenable to our compile-time classification do execute correctly under the proposed protocol, either in the standard directory protocol mode (SC protocol) or in the OS-private mode, but do not benefit from the optimizations designed for xDRF code regions. To enable xDRF-tailored optimizations, the role of the compiler can be subsumed by an expert delineating xDRF regions[1] and unleashing all the benefits of SPEL++.

We evaluate SPEL++ on a wide variety of applications from different benchmark suites, simulating a 64-core chip multiprocessor architecture similar to the Intel Xeon Phi co-processor [4] (Section 5). Experiments show an average performance improvement of 38%, 19%, and 20% for the Polybench, Rodinia, and SpecOMP benchmark suites, respectively, while reducing energy consumption by 48%, 47%, and 53% (Section 6).

Contributions: The proposal consists in an optimized coherence protocol that automatically "gels" to the code's behavior, namely to the type of each memory access. The type of accesses is identified by a hybrid static-dynamic classification that involves both the operating system and the compiler. The protocol automatically selects the optimal mode to handle each memory access efficiently, successfully addressing the three-fold goal: scalability, performance and energy efficiency [1], and ensuring compatibility with legacy code. This paper extends our previous proposal, SPEL [25], with the following contributions:

- Proposes a hybrid static-dynamic classification of memory accesses that delivers more opportunities for further optimizing the coherence protocol (Section 2);
- Thoroughly details the compile-time (Section 2.1) and the runtime (Section 2.2) techniques for classifying memory accesses and their interplay (Section 2.3);
- Extends the coherence protocol to optimally handle each memory access with respect to its nature, as dictated by the hybrid classification (Section 3);
- Proposes optimizations of the coherence protocol regarding the fetching of data upon write accesses (on demand vs. prefetching) (Section 3);
- Provides an extensive evaluation of the proposed optimizations, showing performance improvements and significant energy savings over our previous work (Section 6).

2 Classification of Memory Accesses

The proposed coherence protocol employs a hybrid classification scheme with two complementary techniques: a static, compiler-based classification of extended data-race-free (xDRF) regions and a runtime, OS-based classification of memory accesses.

2.1 Static: compile-time delineation of xDRF regions

SPEL++ is readily applicable on applications for which a compiler can precisely identify xDRF regions, either automatically or based on user-provided annotations.

An xDRF region consists of *a set of* DRF regions, which all together act as one unique DRF region. The DRF regions may be interleaved with non-DRF (nDRF) regions, but the nDRF regions do not belong to the xDRF region and do not break its data-race-free semantics. We denote such nDRF regions as enclave. An xDRF region must satisfy the following properties: (1) no memory access performed in an enclave nDRF region may alter data accessed in the xDRF region. As an exception, a thread is allowed to modify its thread local data, even during the execution of an nDRF region, since no other thread can access it; (2) all properties that hold in a standard DRF region [22], hold across the entire xDRF region. In short, it is guaranteed that parallel threads executing the same xDRF region do not access the same data (at least one access being a write), even if accesses are performed from within different DRF regions part of the same enclosing xDRF region.

The OpenMP programming model [26] is particularly well-suited for such a classification, as data sharing is entirely controlled by dedicated synchronization constructs (*e.g.*, atomic, critical), which are easily identified statically. Similarly, in automatically generated data parallel applications, the compiler has complete knowledge of the xDRF and nDRF code regions.

Once the parallelizing code transformations have been applied, the compiler delineates parallel from sequential regions. Sequential regions are considered xDRF, while parallel regions may contain a mixture of interleaved xDRF and nDRF regions. *Barriers* mark the beginning and end of an *xDRF region*, nevertheless, there may be several enclave nDRF regions (*i.e.*, synchronization points that protect shared data such as locks or critical sections).

1. Inserting directives provided for the programmer to enable the use of the xDRF classification for irregular, generic parallel codes, such as Apache, MySQL servers, etc.

Should synchronization mechanisms be used with moderation in scalable parallel programs, most of the parallel code represents xDRF regions.

2.1.1 Handling OMP directives and worksharing constructs

Identification of xDRF regions is based on the semantics of each OpenMP construct.

```
#pragma omp parallel for(sharedA)
for(int i=0; i<N; ++i)
  A[i] = i;

//Is internally transformed to:
Start_parallel_region();
 Thread_func(...);
End_parallel_region();

Thread_func(...){
  int this_th = omp_get_thread_num();
  int num_th = omp_get_num_threads();
  int my_start = (this_th  ) * N / num_th;
  int my_end   = (this_th+1) * N / num_th;
  for(int i=my_start; i<my_end; ++i)
     A[i] = i;  }
```

Listing 1. OpenMP *for* directive and the simplified transformed code

PARALLEL FOR: For example, the directive `#pragma omp parallel for` splits the for-loop such that each thread in the current team executes different loop iterations, as displayed in Listing 1. Such code transformations are performed blindly by the compiler, nevertheless, the programmer is responsible for avoiding data races, by ensuring that the loop iterations can run in any order (*i.e.*, there are no loop carried data dependencies and no parallel updates of shared variables).

In this simple example, since no synchronization is required, the whole parallel region is an xDRF region, thus the boundaries of the xDRF region coincide with the functions calls `Start_parallel_region()`, `End_parallel_region()`.

Consider now the example illustrated in Listing 2, which shows the pseudo-code generated by the compiler when `#pragma omp parallel for schedule(runtime)` is encountered. The OpenMP clause `schedule(runtime)` instructs the compiler to split the iteration domain of the parallel loop in *slices*. Thus, each thread is allocated a subset of iterations (slice), and as soon as it finishes its slice, the thread asks for more work, by calling `GOMP_next_slice()`. This distribution of work allows for better load balancing, but incurs more synchronization, since each request for a new slice requires a lock to update the number of not-yet-executed iterations (compared to the example in Listing 1). Each slice, excluding the call `GOMP_next_slice()` represents a DRF region, since the OpenMP paradigm guarantees that the loop iterations may be run in parallel, without incurring data races. The set of all slices, *i.e.*, the set of all DRF regions, builds up the xDRF region whose boundaries correspond in this example to the ones of the OpenMP parallel region. Hence, the xDRF region is non-contiguous since it is interrupted by calls to `GOMP_next_slice()`, which represent nDRF regions. Similarly, any call to an external library (here illustrated with libgomp [27]) that requires a lock is handled as an nDRF region enclave in the xDRF region.

SECTIONS: The `Sections` directive specifies that the enclosed section(s) of code are to be distributed among the threads in the team and each nested `Section` is executed once by one thread. Different sections may be executed by different threads. Hence, a section has the same semantics as the slice within a *parallel for loop*. As soon as the thread completes its section, it asks for another section to execute by calling `GOMP_sections_next()`. Thus, a section itself is DRF and the entire region of sections is xDRF, since the sections can be executed independently by different threads. The call to `GOMP_sections_next()` is nDRF.

TASK: `Sections` and `tasks` (their successor in OpenMP 3.0 and later) share many similarities, as well as differences. Similarly, the assumption is that all tasks can be executed independently, but the execution may be either immediate or delayed. To force threads to start executing tasks from the worksharing construct (*i.e.* task queue), synchronization points, such as `taskwait` or `barrier`, discussed later, must be used. From SPEL++s perspective, the parallel construct acts as a parallel region and it is by default considered an xDRF region. The creation of tasks with `GOMP_task` is an nDRF region, as it locks and updates the shared task queue, but the code of each task represents a DRF region within the xDRF region.

SINGLE: The `Single` block is executed only by one thread, while the rest of the threads in the team wait at the implicit barrier at the end of the `Single` block (unless `nowait` is specified). SPEL++ handles the parallel region until the `Single` block as xDRF, ended by the presence of the implicit barrier.

```
sdrf 0
drf.flush
Start_parallel_region();
 Thread_func(...);
sdrf 0
drf.flush
End_parallel_region();
sdrf 1

Thread_func(...){
sdrf 1
  int my_start, my_end;
  bool thread_has_work =
sdrf 0
  /* Takes a lock and updates the
   number of remaining iterations */
  GOMP_next_slice(my_start, my_end, N);
sdrf 1
  while ( thread_has_work){
    for(int i=my_start; i<my_end; ++i)
       printf(" %d", i);
    thread_has_work =
sdrf 0
    GOMP_next_slice(my_start, my_end, N);
sdrf 1
  }
}
```

Listing 2. Transformed code of a loop with OMP directive *#pragma omp parallel for schedule(runtime)*

2.1.2 Handling OMP synchronization constructs

CRITICAL, *ATOMIC*: Synchronization mechanisms that ensure atomicity (*e.g.*, `critical`, `atomic`) are identified and the corresponding code regions are marked as nDRF at compile-time. For example, the entire code section protected by a `#pragma omp critical` directive is nDRF, since threads manipulate data that must be visible to other threads at the end of the critical section. Although a critical section is, by definition, DRF, we opt for considering them nDRF. Our motivation is that coherence is required between threads when exiting and entering the same critical section. By considering critical sections as nDRF

regions, we avoid splitting an xDRF region into smaller xDRF regions, which, as we will see in Section 3, will introduce extra self-invalidation, and therefore, may increase execution time.

BARRIER, *TASKWAIT*: On the other hand, synchronization mechanisms that impose an order between threads (*e.g.*, `#pragma omp barrier`, `#pragma omp taskwait`) mark the end of the xDRF region, as threads executing code following the barrier are expected to access data updated before the barrier (possibly by other threads during the xDRF region), thus violating the xDRF properties.

ORDERED: In contrast, `#pragma omp ordered` specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor. Threads are ordered using the internals of the *libgomp* library (`gomp_ordered_sync`), waiting before executing their chunk of iterations if previous iterations have not completed yet. Since data is not expected to be made visible among the threads before the completion of the loop, the loop iterations build the xDRF region, while the synchronization dictating the order between the threads is nDRF.

MASTER: Constructs which indicate that a code region is executed only once by one thread (*e.g.*, `#pragma omp master`) are marked as DRF, part of the enclosing xDRF region. Since there is no implied barrier with this construct, while the master thread executes the block, the other threads continue doing other work. All updated data is communicated among the threads upon a (*#pragma omp barrier*), at the end of the xDRF region.

FLUSH: The `#pragma omp flush` directive identifies a synchronization point at which the implementation must provide a consistent view of memory, *i.e.* when thread-visible variables are written back to memory. This directive is translated by the compiler as a *fence seq_cst* [2], which is semantically equivalent to the end of the xDRF region.

2.1.3 Handling OMP data scope attribute clauses

In addition to synchronization mechanisms, the OpenMP framework provides several solutions to avoid data races, via clauses which explicitly define how variables should be scoped. *The privatization approach* (*e.g.*, `private`, `firstprivate`, `lastprivate` clauses) creates thread-local duplicates of shared variables, such that each thread can safely update its private copy. The private copies are initialized or may be copied back into the shared variables at the end of the parallel region, according to the semantics of the privatization clause. Internally, these threadprivate variables are declared in the code section executed by each thread and initialized with values transmitted as parameters to each thread. Hence the copy-in is transparent to our classification strategy, while the copy-out takes place outside of the xDRF region, *i.e.*, after a barrier.

A special case is the `#pragma omp threadprivate` directive, which indicates that values are privatized and maintained persistent between different parallel sections of code, *i.e.* a `threadprivate` variable will have the same value if accessed in a subsequent parallel region by the same thread, given that the number of threads is constant. `Threadprivate` variables are transformed at compile-time into `thread_local` variables [28], thus they will not be shared by threads (each thread will have a separated copy of the variable). The `copyin` clause provides a means for assigning the same value to `threadprivate` variables for all threads in the team. Since SPEL++ propagates all writes on a transition between adjacent xDRF regions (*e.g.* between a sequential and a parallel xDRF region), `threadprivate` variables are maintained persistent naturally by our protocol.

Note that accesses to data annotated as `shared` may belong to an xDRF region, if the OpenMP directive semantics indicates such accesses are safe, *e.g.* array A in Listing 1. In contrast, accesses to scalars declared as `shared` require synchronization, *e.g.*, critical sections, and are therefore handled as nDRF[3].

`Reductions` create a private copy per thread for each variable. At the end of the reduction, the values of the threadprivate variables are accumulated in the global shared variable, which is translated in the intermediate code to a write protected by a lock or an atomic operation. Private copies are written in parallel in the xDRF region, while the final update of the shared variable (via the reduction operation) is classified as nDRF.

2.1.4 Instructions delimiting regions of code

In order to delimit xDRF and nDRF regions, the code is compiled in two steps using LLVM [29]. First, the parallel code is generated in the LLVM intermediate representation. Next, a dedicated compiler pass inserts instructions delimiting the xDRF and nDRF code regions. There are two types of instructions inserted statically: `sdrf` and `drf.flush`.

The `sdrf` instruction (set SC-for-DRF coherence) delimits DRF regions which are part of the same xDRF region. The role of the `sdrf` instruction is to inform the processor whether it has to handle the subsequent memory accesses under SC-for-DRF or SC mode. For this purpose, it enables or disables a processor flag *DRF* (SC-for-DRF coherence), accordingly. Hence, `sdrf 1` sets the flag, indicating that coherence can be guaranteed by the SC-for-DRF protocol, while `sdrf 0` enforces the use of the SC protocol. Essentially, the implementation of the `sdrf` instruction is similar to a memory fence, which prevents reordering accesses across it and in addition updates the *DRF* flag.

The `drf.flush` marks the end of xDRF regions, where data modified in the SC-for-DRF protocol mode must be propagated. Since `drf.flush` instructions mark the boundaries of each xDRF region (and not of each DRF region), the number of flushes is considerably reduced, leading to better performance than previous SC-for-DRF protocols. The placement of these instructions is shown in Listing 2.

2.2 Dynamic: OS-based classification

We employ a standard operating system (*i.e.* OS-based classification) [12], [16], which is highly conservative in labeling memory accesses as *private*, but provides strong guarantees, thus enabling powerful optimizations of the coherence protocol. The OS-based classification labels a memory access as *private* if the entire memory page where the accessed data resides has been referenced by only one thread since the beginning of execution, up till the access is performed. More precisely, upon a first TLB miss for a certain page, the OS stores information about the page's owner along with the page's entry in the page table. The required information consists in an owner field, a cached bit and

2. *Fence seq_cst* is a fence to ensure sequential consistency, providing Acquire semantics for loads and Release semantics for stores. Additionally, it guarantees that a total ordering exists between all sequentially consistent operations [28].

3. It is the programmer's responsibility to ensure that multiple threads properly access `shared` variables

a private/shared (PS) bit [16]. The OS is aware of which cores access each page, as the OS is responsible for resolving TLB misses. If the page is only accessed by its owner core, the page is labeled as *private*. As soon as another core incurs a TLB miss for that particular page, the page becomes *shared*. In other words, once a memory page is accessed by a second thread, the page becomes *shared* and any access targeting this page will be labeled as *shared* until the execution completes or until the page is evicted from main memory.

2.3 Hybrid static-dynamic classification

To increase the accuracy of our classification, we exploit the advantages of each method (static vs. dynamic) applied in isolation and show how the two techniques become complementary when applied in conjunction.

On one hand, the OS-based classification operates at memory page granularity, leading to a high degree of false sharing reports. Moreover, once the memory page is labeled as *shared*, it remains shared until it is evicted from the main memory. In contrast, the xDRF-classification operates on a finer *granularity* – of accessed data – ensuring that throughout the execution of an xDRF region multiple threads cannot access the same byte. Furthermore, it resets the classifications at the boundaries of xDRF regions, providing *temporality*. This approach splits the set of *OS-shared* memory accesses into *xDRF* and *nDRF* accesses (corresponding to xDRF and nDRF regions, respectively).

On the other hand, the compile-time approach performs a bulk classification of the memory accesses residing in the same region (either xDRF or nDRF). In particular, an nDRF region may contain interleaved *OS-private* and *OS-shared* memory accesses. Given that the OS-based classification operates on *accesses*, rather than regions, such *private* accesses within nDRF regions are correctly identified. In addition, a compiler does not have access to dynamic information (*e.g.* thread affinity) and must conservatively delineate each barrier as bounding an xDRF region. Conversely, the OS-based classification is guided by dynamic sharing of data between threads and can therefore identify whether different threads access the same data (memory page) across a barrier. This information is useful in alleviating conservative actions that prove to be unnecessary for maintaining coherence (Section 3).

This hybrid classification scheme identifies three types of memory accesses, each type requiring a different coherence protocol, as explained in the following section:

- *OS-private*: accesses that find the PS bit set to *private* (0).
- *xDRF*: accesses that find the PS bit set to *shared* (1) and the xDRF flag set to 1.
- *Shared*: accesses that find the PS bit set to *shared* (1) and the xDRF flag set to 0 (nDRF).

3 Dual-consistency cache coherence

Relying on the presented hybrid static-dynamic classification of OS-private accesses and xDRF regions, we propose SPEL++, a dual-consistency cache coherence protocol, where *shared* memory accesses are kept coherent by a standard directory protocol that ensures SC by propagating writes immediately (Section 3.1), *xDRF* accesses follow an SC-for-DRF protocol that guarantees the visibility of the writes no later than the end of their xDRF region (Section 3.2), and memory accesses classified as *private* by the OS delay the propagation of writes until the page becomes shared (Section 3.3).

3.1 SC coherence protocol

Memory accesses considered *shared* are made coherent in SPEL++ with a traditional SC coherence protocol, which is an invalidation-based MOESI directory protocol [30] with a directory cache to track the memory blocks stored in the private caches.

Read misses are sent to the directory controller, where the directory cache keeps the information about the owner and about the sharers of the cached blocks. The directory controller forwards the requests to the cache owning the block, in case the owner is not the shared cache (co-located with the directory). Then, the owner sends a copy of the data block to the requester, which stores the block in its local cache and completes the read operation.

Write misses generate invalidation messages to all caches holding copies of the requested block. Each cache replies to the invalidation with a message of acknowledgment or with the data block in case of the block's owner. These messages are sent to the requester. Once the requester receives all messages, it can perform the write operation.

All transactions finish with an *unblock* message from the requester to the directory controller. The directory controller remains blocked from the point in time it processes the request until it receives the *unblock* message. Directory blocking avoids complex protocol races.

3.2 SC-for-DRF coherence protocol

Memory requests that are classified as *xDRF* do not modify the coherence status (*e.g.*, cache MOESI states, directory information, etc). Hence, blocks cached due to an *xDRF* access are not tracked by the directory and remain invisible to the coherence protocol. Instead, every memory block sets a *toFlush* (*F*) bit, that is kept along with every block in a private cache. For example, if a block is not present in the cache (*I* state) and is brought by an xDRF load, the state remains *I* and the presence bit is not added to the directory. Consequently, the coherence protocol cannot "see" the block and cannot invalidate it. However, the block resides in cache with the *F* bit set, such that the local processor can access it.

In contrast, the coherence status of a block (according to the coherence protocol) must be visible for xDRF accesses. For example, an xDRF read miss has to access the directory in order to locate the *coherent* copy of the block. Once the copy of the block is found, either in a shared cache or in a private cache, it is sent to the requester.

The key performance benefits for xDRF accesses, that are kept coherent through an SC-for-DRF protocol, over shared accesses are the following:

- Since the blocks marked as *F* are "invisible" to the coherence protocol, they do not require an entry in the directory cache, thus increasing the efficiency of the directory.
- As a consequence, invalidation requests performed by the coherence protocol do not affect the *F* blocks, thus reducing the cache miss rate (particularly, misses due to false sharing) and, as a consequence, reducing traffic.
- Thanks to the DRF properties, writes can proceed without waiting for write permission and do not perform invalidation of other copies, thus reducing the latency of write misses.
- Directory blocking is considerably reduced for xDRF misses since it is only required when the up-to-date copy of the block is in a private cache (owner state). Thus both (i) network traffic is reduced as less *unblock* messages will

be generated, and (ii) the waiting time of the subsequent requests is diminished.

3.2.1 Hardware support for handling false sharing

As opposed to real sharing where multiple threads update the same data, false sharing occurs when multiple threads modify different data residing in the same memory block. Traditional coherence protocols invalidate all remote copies of a written block, even in case of false sharing. This may cause important performance degradation due to frequent invalidations, increased traffic, etc. On the other hand, the management of false sharing in SC-for-DRF protocols requires extra information in order to maintain correctness.

Our compile time classification provides data-race free guarantees with a granularity finer than the cache block. Hence, during SC-for-DRF coherence mode several cores can write the same block (but access different bytes). The protocol ensures correctness by sending through the network "diffs" with only the written bytes and by "merging" them at the shared cache level. For this purpose, we add extra bits to mark the bytes written in SC-for-DRF mode (xDRF writes).

Ideally, in order to mark the written bytes of every cached block, the system would require one bit per cache byte. This represents a memory overhead of 12.5% of the effective L1 cache (in general, of the effective size of the private caches). However, this overhead can be reduced by storing information regarding the written bytes only for a subset of the cached blocks, namely, for the ones that are actually written during SC-for-DRF mode. In practice, most accesses are actually read operations. Therefore, we propose the use of a new cache-like structure for keeping track of the written bytes: the *written-bits cache*. This cache can considerably reduce the number of entries required to store the written bytes with respect to the number of entries in the cache.

Effectively, as we show in Section 6.3, a written-bits cache with only 32 or 16 entries is sufficient for obtaining similar performance to having as many entries as cache entries (512 in our case). In this implementation, when there are no available entries in the written-bits cache, some written-bits information has to be evicted. Since this information is lost, SPEL++ forces a write-back of the dirty bytes in the corresponding cache blocks, thus making them visible to the coherence protocol, as explained below.

3.2.2 Write propagation: From SC-for-DRF to SC

Writes performed under an SC-for-DRF protocol become visible to other threads either *on demand* or *forced* by the `drf.flush` instruction. The first scenario occurs when (i) dirty *F* blocks are evicted from cache, (ii) the corresponding entries in the written-bits cache are evicted, or (iii) upon *shared* write accesses by the local core. The second scenario occurs at the end of xDRF regions.

Upon the eviction of a dirty *F* block (or its written-bits entry), the modified data are written back to the shared cache. Previously all coherent copies of the same block, which are tracked by the directory, must be invalidated from the private caches and written-back to the shared cache, if dirty. The evicted *F* block is then merged with the current copy, and from this point on, it will be visible to the other threads, *i.e.*, it will be coherent. Fig. 1 details this process.

Upon a *shared* write access, remote copies are invalidated and the data block is sent to the requester. If there is already an *F* block residing in the cache of the requester, it is "merged" with the current copy, resetting *F* and the written bits. This way, the block is made coherent. Fig. 2 details this scenario.

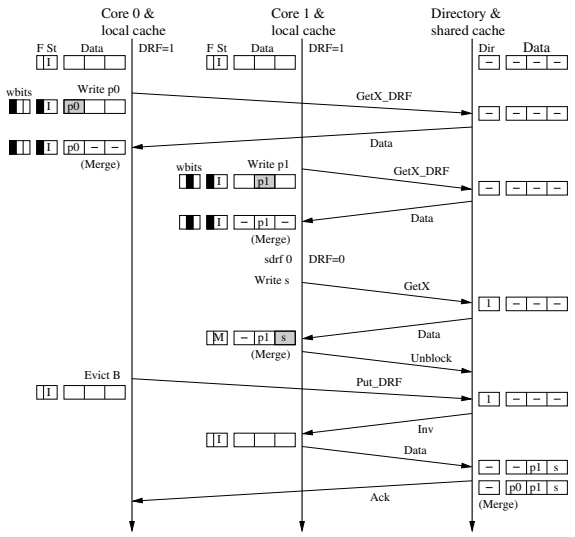


Fig. 1. Merging private data on evictions. Gray boxes indicate when the processor write is effective. Block B is initially stored in the shared cache, containing three data locations: p_0 will only be accessed by core 0, p_1 only by core 1, and s will be accessed by both cores. At the beginning, the *DRF* flag is set in both threads and we assume that the page is considered as *shared* by the OS. First, Core 0 writes p_0. The SC-for-DRF write is performed without waiting for permissions, and both the *F* bit for block B and the written-bit (*wbits*) corresponding to p_0 are set. The remaining data of the block is prefetched from the shared cache and merged appropriately with the write. Similarly, Core 1 writes p_1. Next, Core 1 sets the *DRF* flag to 0, so future requests from Core 1 will be kept coherent by the SC protocol. Then, Core 1 attempts to write s. Since the block has been accessed previously in SC-for-DRF mode, the directory does not track it and the copy in Core 0 is not visible to Core 1. Hence, Core 1 gets the block from the shared cache, merges it with its dirty data, clears both *F* and *wbits*, and writes s. Now, p_1's value is visible to any thread, because it is tracked by the directory (illustrated as 1 in the Dir field). When B is evicted from Core 0, the directory asks for a write-back of the copy in Core 1 (in case of more sharers, all of them should be invalidated). Once Core 1's copy arrives to the shared cache, it is merged with the dirty data from Core 0, and an acknowledgment is sent to Core 0. The evicted data is now also visible to other threads.

When the processor executes a `drf.flush` instruction, every dirty *F* block in its local cache is evicted. The `drf.flush` instruction does not commit until every eviction has been acknowledged (see Fig. 1). The overall performance penalty of the `drf.flush` instructions is negligible since they occur infrequently and they only evict *F* blocks.

3.2.3 Non-prefetching on xDRF write misses

As depicted in Fig. 1, xDRF write accesses that miss in the local cache request the data block from the shared cache. This is performed in SPEL [25] mainly as a prefetch for data co-residing in the same block. Since there is already a cache entry that stores the value written by the xDRF access, the remaining data can also be stored in the cache without requiring extra resources. A subsequent read to that block will be a cache hit, thus improving performance.

However, this prefetching transaction turns out to be unnecessary most of the times, because, in practice, there are seldom read accesses to the same block before the block is evicted. Hence, the prefetching introduces an additional energy expenditure, without performance improvements.

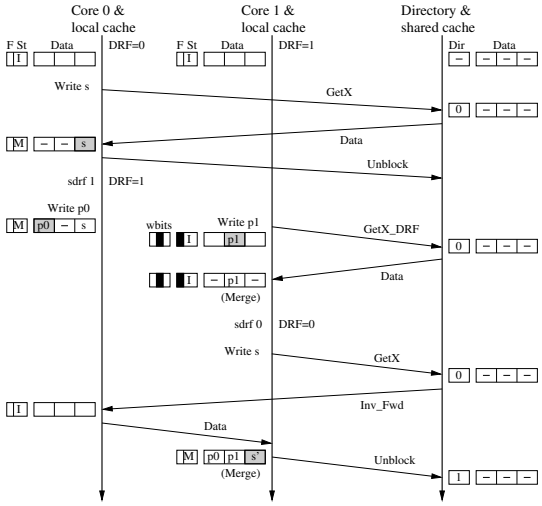


Fig. 2. Merging private blocks on writes to shared data. Core 0 starts with the *DRF* flag unset and Core 1 with the flag set. First, Core 0 writes s. Next, *DRF* flag is set in Core 0, and it writes p_0. Since the state of the block is M (modified), the SC-for-DRF write will be visible, and *F* and *wbits* are not set. Then, Core 1 writes p_1. As execution continues, the *DRF* flag is cleared in Core 1, and it attempts to write s. However, the directory tracks B in Core 0, so it asks Core 0 to invalidate B and send it to Core 1, where both blocks are merged and *F* and *wbits* reset. Then, s can be written (s').

Therefore, we implement in this work a non-prefetching version of the SC-for-DRF protocol. In this version a dirty *F* block can reside in the cache having the non-dirty bytes invalid. A read access to a non-dirty byte in this block will consequently cause a cache miss, and the remaining data of the block are fetched and stored in cache.

3.3 OS-based coherence protocol

OS-private accesses do not require coherence maintenance (as long as they preserve their private nature) [16] and follow what we call an OS-based coherence protocol. Under this protocol, *OS-private* accesses are not tracked by the directory and no extra cache information is stored, for example, in the written-bits cache. *OS-private* accesses simply perform a request-response transaction that does not block the directory.

Since an *OS-private* page is only accessed by one core while it is *private*, blocks stored in cache due to *OS-private* accesses do not set the *F* bit, and consequently, they are not flushed upon `drf.flush` instructions, thus reducing the cache miss ratio. This is the main advantage of *OS-private* accesses with respect to *xDRF* accesses.

3.3.1 Write propagation: From OS-private to SC

The only coherence action under the OS-private protocol is taken when a page transitions from *private* to *shared*. In this case, all blocks belonging to the page and stored in the local cache of the core holding the page as *private* must be flushed. Dirty blocks are written-back to the shared cache and clean blocks are invalidated, thus making visible the written values to the other cores. Flushed blocks transition to a non-cached state, and subsequent accesses, either *xDRF* or *shared*, will bring back the block to cache in a status according to their corresponding protocols.

TABLE 1
Characteristics of coherence protocol modes w.r.t. access type

Protocol	Type of access	Coherence information	Invalidation
SC	*Shared*	Directory cache and cache states	Write miss
SC-for-DRF	*xDRF*	Written-bits cache and *F* bit in cache	`drf.flush` instr.
OS-based	*OS-private*	Page table and PS bit in TLB	Page becomes *shared*

3.3.2 Non-prefetching on OS-private write misses

As for *xDRF* accesses, *OS-private* accesses may implement the non-prefetching optimization described in Section 3.2.3. However, when this optimization is implemented for *OS-private* accesses, *all* accessed bytes are marked in the written-bits cache, *i.e.* both written and valid blocks. This optimization reduces the interconnection traffic, at the small cost of using extra entries in the written-bits cache. Note that when an entry is evicted from this cache, only a "diff" with the written bytes of the corresponding block is sent to the shared cache.

3.4 Summary

Table 1 summarizes the three protocol modes described in this section: (1) the sharing information for *shared* accesses is stored at the directory cache and cached copies are invalidated upon write misses as in a traditional protocol; (2) bytes of blocks modified by xDRF writes are marked as dirty in the written-bits cache and *F* blocks are invalidated on `drf.flush` instructions; (3) finally, information about private pages, whose accesses are considered *OS-private*, is stored in the page table and invalidations are triggered when the page becomes shared.

4 Discussion

4.1 Thread migration

The operating system can decide at runtime to execute a thread on a different core, in which case, cache coherence is guaranteed in SPEL++ in the following way.

OS-private pages become *shared*, as they are accessed by a new core. When the page becomes *shared*, the corresponding blocks within the page are flushed from the private cache of the core that was previously executing the thread.

Cache coherence of blocks marked with the *F* bit, *i.e.*, due to *xDRF* accesses, is guaranteed by an *drf.flush* instruction executed by each thread before being de-scheduled. This instruction makes the *xDRF* writes "visible" to the coherence protocol, and the latest value is thus accessible from the new core.

4.2 Multitasking, simultaneous multi-threading, and syscalls

Applications compiled to expose xDRF regions may co-execute on the same core with other applications (either exposing xDRF regions or not). When two applications performing *xDRF* accesses coexist on the same core, an `drf.flush` instruction will write-back all *F* blocks from the cache and not only the ones of the requesting application. However, in practice, most write-backs correspond to blocks accessed by the current application, as other blocks have already been evicted.

Since the *DRF* flag is set per thread, when a thread is descheduled, the value of the *DRF* flag is recorded by the OS along with the process context and the newly scheduled thread resumes with the value of the flag corresponding to its own context. Systems supporting multiple hardware threads (SMT) require one *DRF* flag per thread. Upon scheduling an application thread to a hardware thread, the OS sets the corresponding *DRF* flag, similar to a multitask environment.

System calls can share data, and therefore, are always executed under SC coherence. When an OS exception is triggered, the *DRF* flag is cleared. Once the system call completes, the thread resumes with its corresponding flag.

4.3 Protocol scalability

Traditional protocols employ a sharing code (either as a n-bit vector, where n represents the number of cores, or compressed) to eliminate the need of broadcast requests or broadcast support. While this sharing code is still required in our SC protocol, our two specialized modes of SPEL++ do require specific information for each mode, as summarized in Table 1. For instance, the SC-for-DRF protocol requires a small number of written-bits entries for each private cache. Since the area overhead entailed by this structure does not depend on the number of cores, it is a scalable structure. Similarly, the OS-based protocol stores information about the owner of a block (plus two extra bits) along with the page table entry. The requirements therefore are $2 + log_2 n$ bits per page table entry, which implies high scalability.

Additionally, for *xDRF* accesses, SPEL++ eliminates most directory invalidations, forwarding requests, cache-to-cache transfers and unblock messages. The task of keeping coherence in SC-for-DRF mode is distributed among the cores in the system. This leads to less traffic and less communication between threads, thus improving both performance and scalability. Moreover, SPEL++ does not have any overhead with respect to a standard directory protocol when keeping coherence for *xDRF* accesses.

In traditional protocols, directories may become a bottleneck for large-scale systems, since they impose serialization of requests. The directory controller is blocked while processing a request, hence, as cores submit multiple simultaneous requests, they must wait for the controller to be unblocked. As the number of cores in the system increases, the waiting time can incur important performance degradation [31]. Both for *xDRF* and *OS-private* accesses, SPEL++ considerably reduces directory blocking during the resolution of cache misses and it does not track sharers. In consequence, directory availability and productivity are highly increased, preventing it from becoming a bottleneck.

5 Simulation Environment

We evaluate SPEL++ using the GEMS simulator [32], a cycle-accurate simulator for multiprocessor systems. The interconnection network has been modeled with GARNET [33], included in the GEMS toolset. We have modified GEMS in order to model both the classification and the protocols in detail, accounting for the cost of the instructions inserted by the compiler. We report energy consumption using the McPAT tool [34], assuming a 32nm process technology.

The baseline system used for the evaluation is a 64-tile chip multiprocessor that shares many similarities with the recently launched Intel's Xeon Phi co-processor [4]. For example, it implements a directory-based cache coherence protocol and it connects all cores through a high bandwidth bidirectional ring interconnect. We model in-order cores and provide sequential consistency. Processor techniques to improve performance based on relaxing the consistency model have been previously analyzed [19] and are complementary to this work. The focus of this work is the cache coherence protocol. The set of parameters employed in our simulations are shown in Table 2.

TABLE 2
System parameters

Parameter	Value
Cache hierarchy	Non-inclusive
Cache states	MOESI
Block / Page size	64 bytes / 4 KB
Split instr & data L1 caches	32 KB, 8-way (128 sets)
L1 cache hit time	1 (tag) and 2 (tag+data) cycles
Shared unified L2 cache	512 KB / tile, 16-way (512 sets)
L2 cache hit time	6 (tag) and 12 (tag+data) cycles
Directory cache	64 sets, 8 ways ($\times$1 L1)
Directory cache hit time	2 cycle
Memory access time	160 cycles
Topology	Bidirectional ring
Flit size, link time	72 bytes, 1 cycle

We compare three variants of our dual-consistency protocol (SPEL, SPEL+, and SPEL++) to a traditional SC protocol (Directory) and a state-of-the-art SC-for-DRF protocol (VIPS) [3]. SPEL represents our previously published proposal [25]. SPEL+ extends SPEL with the hybrid static-dynamic classification described in Section 2.3. Finally, SPEL++ adds on top of SPEL+ the non-prefetching technique for both *xDRF* and *OS-private* accesses described in Sections 3.2.3 and 3.3.2, respectively. All SPEL versions evaluated employ a 32-entry written-bits cache, except when performing the written-bits cache sensitivity analysis.

Our evaluation is carried out with a wide variety of applications from codes parallelized with OpenMP, SpecOMP 2012 [35] (352.nab, 359.botsspar, and 367.imagick – test input) and Rodinia [36] (backprop – 131072 elements; bfs – graph1MW_6.txt; btree – mil.txt, command.txt; hotspot – 1024 $\times$ 1024; particlefilter – 128 $\times$ 128 $\times$ 10, 10000 particles; and pathfinder – width 50000), to automatically parallelized applications from the Polybench benchmark suite [37] (adi, covariance, fdtd-2d, seidel, and trmm – small size; and mvt, bicg, and dynprog – medium size). The evaluated applications exhibit various data access patterns and cover a large number of OpenMP constructs and thread synchronization methods. Input sizes have been chosen in order to provide a representative behavior of the applications while keeping simulation time within a week. Statistics are collected from the beginning of the first parallel region until the end of the last parallel region.

6 Results

6.1 Effectiveness of hybrid classification

This section compares the xDRF compile-time classification of code regions employed in SPEL [25] to the hybrid run-time/compile-time classification proposed in this work (SPEL+). In particular, we focus on the amount of L1 cache misses for each type of access, since they are resolved involving different coherence mechanisms.

Results are plotted in Fig. 3 and represent the amount of L1 misses normalized with respect to the misses incurred when using an xDRF-only classification. The bars are split in different portions

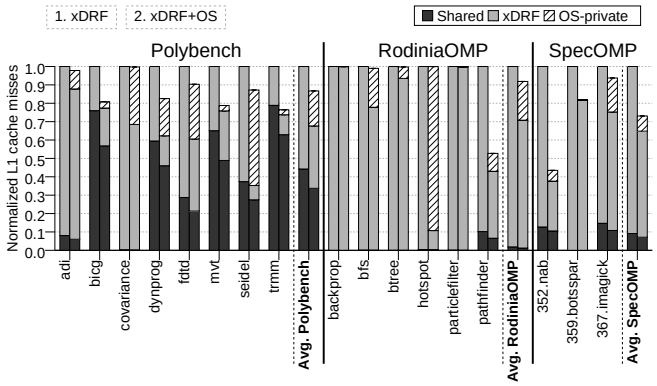


Fig. 3. Cache misses incurred per access type, classified with the xDRF and with the hybrid classification

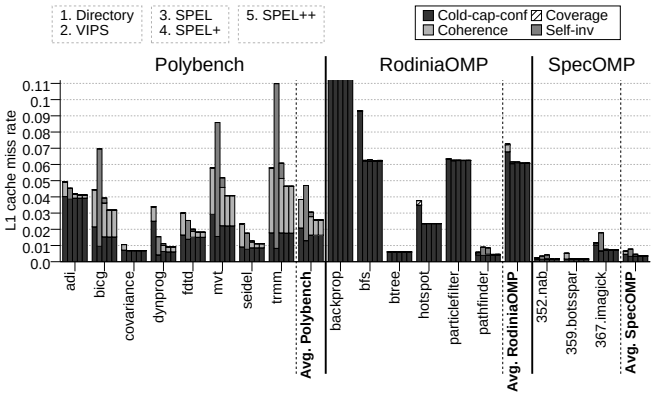


Fig. 4. Cache miss rates incurred in a directory, in a state-of-the-art and in the SPEL-family protocols

depending on the type of access and are labeled according to the nomenclature followed in the paper: *Shared*, *xDRF*, and *OS-private*.

The reader can observe that when using the hybrid (*xDRF-OS*) classification, not only that the overall number of L1 misses is reduced, but also a significant fraction of misses are identified as stemming from *OS-private* accesses: 22%, 23%, and 11%, on average, for Polybench, Rodinia, and SpecOMP, respectively (Fig. 3, second bar). A high number of *OS-private* accesses entails significant improvements in performance, as blocks cached by *OS-private* accesses are not flushed upon `drf.flush` instructions. We recall that *OS-private* accesses immune to any coherence action, except a page transition from *private* to *shared*.

Note however that using an OS-based classification without any compiler support would accelerate only 22%, 23% and 11% of cache misses (on average, for each benchmark suit respectively), while our hybrid classification accelerates between 61% (average for Polybench) to 99% (average for Rodinia) of cache misses. We conclude that the OS-based classification by itself is not sufficient, but the static and runtime classifications combined provide outstanding benefits.

6.2 Performance of SPEL, SPEL+, and SPEL++

SPEL++ is designed to "gel" to the code's behavior and handle each memory access in the most efficient manner, with respect to its type. Hence, SPEL++ optimizes race-free accesses as an SC-for-DRF protocol and racy access as a traditional SC protocol, always providing SC and support for legacy code. This section compares the three flavors of SPEL, namely SPEL, SPEL+ and SPEL++, to both a traditional SC protocol (Directory) and a state-of-the-art SC-for-DRF protocol (VIPS [3]).

Given that state-of-the-art SC-for-DRF protocols require that racy accesses are exposed, for comparison purposes, we rely on the proposed xDRF classification to delimit the racy code. Thus, non-DRF regions are guarded with memory fences for VIPS, which impose self-invalidation and self-downgrade (*i.e.*, a flush) of the cached shared blocks. This enables VIPS to execute codes which would otherwise not be accessible, as races are not exposed by default.

6.2.1 Cache miss rate

The cache miss rate varies in the evaluated protocols due to different techniques of performing invalidations and downgrades. For example, in SC-for-DRF mode, VIPS and SPEL (all three versions) do not require invalidations due to writes nor downgrades due to reads, but self-invalidations and self-downgrades triggered by fence instructions and `drf.flush`, respectively.

The L1 cache miss rate of the evaluated protocols is shown in Fig. 4. The first bar illustrates the miss rate of the L1 cache in a directory protocol, which is split into the *5C* classification of misses [38]: (i) cold or compulsory, capacity, and conflict misses (*Cold-cap-conf*, or *3C*); (ii) coherence misses (*Coherence*), as a consequence of invalidations and downgrades generated by remote writes and reads, respectively; and (iii) misses that stem from invalidations generated by directory evictions (*Coverage*). The second bar shows VIPS, which incurs no coherence or coverage misses, but a fourth category of misses due to *self-invalidation*. The third, fourth, and fifth bars show the three different optimization of SPEL, thus all type of misses can be encountered. When employing the hybrid classification, misses due to flushes in the transition of pages from private to shared are included in the *self-invalidation* category.

With respect to a Directory protocol, SPEL reduces the total number of misses, despite the additional misses incurred by self-invalidation. This is mainly a consequence of avoiding coherence misses due to false-sharing. Cache miss rate is reduced on average by 0.78% for Polybench and 1.10% for Rodinia. The cache miss rate for SpecOMP is very low and the variations obtained with SPEL are not significant. In some applications of the Polybench suite, such as *bicg*, *mvt*, and *trmm*, a noticeable fraction of the misses encountered in SPEL represent self-invalidation misses, which are converted into hits in SPEL+ and SPEL++, thanks to the hybrid classification. The L1 cache miss rate for *backprop* is very high compared to the rest of the applications and is therefore incompletely shown in the graph (the bars have not been scaled down in order to better emphasize the differences in the other applications). *Backprop* shows a $\approx$0.22 miss ratio for Directory and a $\approx$0.20 miss ratio for the other protocols, with a difference of $\approx$0.02 due to coherence misses.

With respect to VIPS, SPEL requires self-invalidation only in the boundaries of xDRF regions, while for nDRF regions it employs directory invalidations. Overall, SPEL reduces the cache miss rate with respect to both Directory and VIPS, emphasizing the advantages of a dual-consistency protocol.

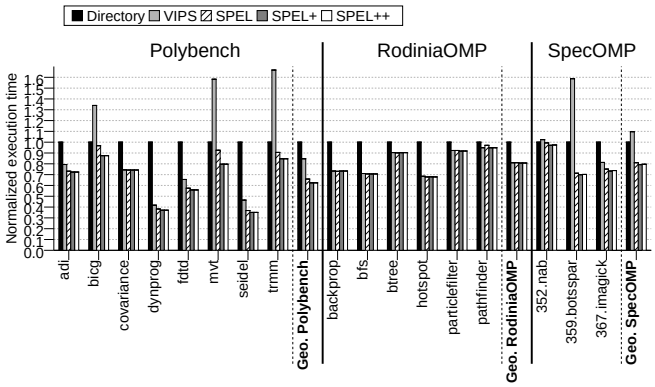


Fig. 5. Execution time improvements with respect to a directory protocol

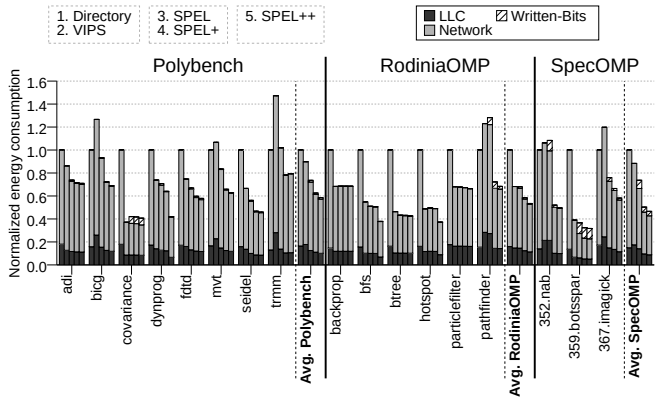


Fig. 6. Energy improvements with respect to a directory protocol

6.2.2 Execution time

By reducing the cache miss rate, SPEL++ achieves considerable improvements in execution time compared with both a pure SC protocol and a pure SC-for-DRF protocol. Fig. 5 shows the execution time normalized with respect to Directory and the geometric mean per benchmark suite.

The extra self-invalidation in the SC-for-DRF protocol increases execution time for some applications, as *bicg*, *mvt*, and *trmm*. For the Rodinia benchmarks, SC-for-DRF protocols work efficiently since the 3C misses clearly dominate the ones caused by self-invalidation. Hence, VIPS and all SPEL versions yield similar performance. However, for Polybench and SpecOMP, massive self-invalidation in VIPS impacts performance negatively, thus SPEL reduces the average execution time by 22% and 26%, respectively, compared to VIPS.

SPEL++ optimizes further the coherence protocol and obtains improvements with respect to the original SPEL protocol of 5%, on average, for Polybench and 2%, on average, for SpecOMP.

Finally, compared with the SC protocol, SPEL++ consistently improves performance for each application. On average, SPEL++ boosts performance by 38% for Polybench applications, 19% for Rodinia, and 20% for SpecOMP.

6.2.3 Energy consumption

Fig. 6 illustrates the energy consumption of VIPS, SPEL, SPEL+ and SPEL++ normalized with respect to Directory. Results show the energy consumption of the network, the shared last-level cache (LLC), and the written-bits cache. We account for the impact of self-invalidations of dirty blocks on the access to the written-bits cache, the extra traffic injected in the interconnect, and the extra writes in the LLC. Since VIPS maintains the information about the written bytes in the Miss Status Holding Register (MSHR) structure, we assume that its extra energy consumption is negligible. We also assume that the impact of resetting the valid bits for self-invalidating clean blocks is negligible.

Recall that SPEL reduces the number of L1 cache misses: (i) with respect to an SC protocol, by not invalidating private blocks upon writes, and (ii) with respect to an SC-for-DRF protocol, by reducing self-invalidation. Additionally, SPEL+ reduces the number of self-invalidations with respect to SPEL, and SPEL++ reduces the traffic with respect to SPEL+, by fetching data on demand only (non-prefetching write accesses).

Generally, with regard to the shared cache L2, the number of accesses increases in SC-for-DRF protocols due to extra fetches or write-backs. At the same time, SC-for-DRF protocols benefit from the merging of cache blocks containing SC-for-DRF data on a write-back, thus reducing the network traffic.

Therefore, SPEL++ consumes less energy than a directory protocol since it removes most of the coherence misses, and less energy than VIPS thanks to fewer self-invalidation and self-downgrade events. On average, SPEL++ exhibits a reduction in the energy consumption of 48% for Polybench, 47% for Rodinia and 53% for SpecOMP, compared to a directory protocol. Compared to VIPS, average reductions of 35%, 22%, and 47%, respectively, are achieved.

Most importantly, SPEL++ is consistently more energy efficient than the original proposal, SPEL. First, SPEL+ reduces the number of write-backs by lowering the pressure on the written-bits cache and reduces the number of flushes due to `drf.flush` instructions. SPEL++ further saves energy by not fetching non-requested data on writes. These two optimizations lead to energy savings of 20% for Polybench, 21% for Rodinia and 37% for SpecOMP, compared to the original proposal, SPEL.

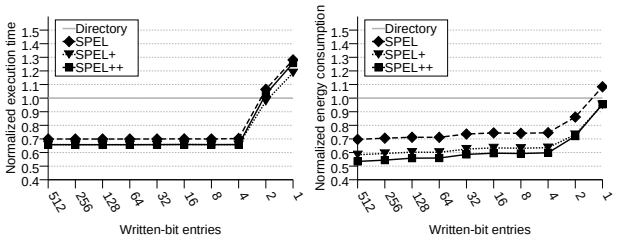
All in all, SPEL++ exhibits considerable energy savings not only with respect to traditional and state-of-the-art protocols, but also significantly improves the energy expenditure of its predecessor, SPEL. Furthermore, outstanding energy savings are achieved without compromising performance.

6.3 Area requirements of the written-bits cache

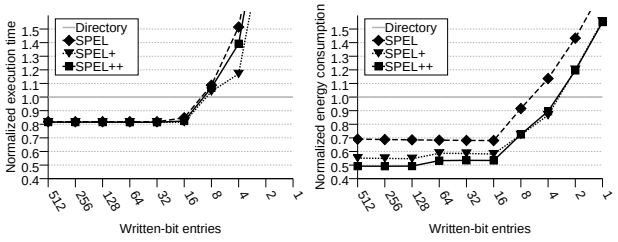
This section analyzes the area requirements of our proposal with respect to the written-bits cache employed in all SPEL protocols to track dirty data for *xDRF* blocks. We provide a sensitivity analysis of the number of entries required by this structure for each SPEL protocol.

Fig. 7 shows the consequences of reducing the number of entries (averages over each of the three evaluated benchmark suites and the three evaluated versions of SPEL). Values from 512 entries (corresponding to the number of entries in the L1 cache) to a single entry were evaluated. Results are normalized with respect to a directory-based protocol, which does not require a written-bits cache. Best results from performance and energy consumption viewpoints are obtained using 512 entries, since there are no penalties due to evictions in the written-bits cache.

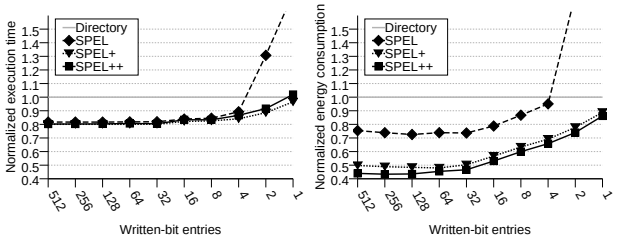
As the number of entries is reduced from 512, performance of the SPEL protocols (graphs on the left in Fig. 7) is unaffected up to only 16 entries for Rodinia and 4 entries for Polybench



(a) Polybench



(b) RodiniaOMP



(c) SpecOMP

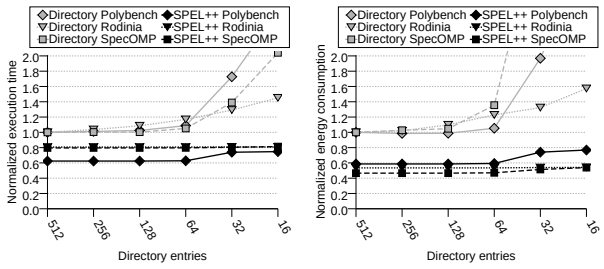
Fig. 7. Performance variations due to written-bits cache size

and SpecOMP. This is because there are not many blocks written during SC-for-DRF mode or they are already evicted due to cache capacity. The improvements of the two SPEL optimizations proposed in this work are more effective for SpecOMP, in which case, reducing the number of entries in the written-bits case to less than 4 does not cause a dramatic performance degradation, as happened in the original SPEL.

The energy consumption of the SPEL-family of protocols also increases as the number of written-bits entries is gradually reduced (graphs on the right in Fig. 7), but the difference is insignificant as the number of entries is reduced up to 32 for SpecOMP, 16 for Rodinia, and 4 for Polybench. While the energy expenditure of the three protocols SPEL, SPEL+ and SPEL++ follow the same trend with respect to the number of written-bits cache, SPEL++ is significantly more energy efficient on SpecOMP, even with very small written-bits cache structures.

The impact of reducing the size of the written-bits cache is visible on the applications' energy consumption earlier than on execution time. Execution time is less affected because the operations (evictions and allocations) in the written-bits cache are performed out of the critical path of the access, whereas energy is directly affected as the extra write-backs caused by the eviction of written-bits increase both network traffic and L2 accesses. Execution time increases when the limited capacity of the written-bits cache generates a bottleneck in the network, cache controller, and MSHR.

Polybench applications are optimized for data locality, therefore a small structure of 4 entries suffices, whereas for more complex applications such as Rodinia or SpecOMP, a larger structure is required.



(a) Execution time (b) Energy consumption

Fig. 8. Performance variations due to directory cache size

We conclude that, in general, a structure containing 32 entries yields results competitive to the ones obtained using 512 entries. Therefore, assuming per-byte information, the extra area requirements of the written-bits cache with respect to the L1 cache size is only 0.96% (0.37KB).

6.4 Directory cache effectiveness

One of the advantages of SPEL++ is that it reduces the number of entries required by the directory cache, since *OS-private* and *xDRF* cached blocks are not tracked by the SC coherence protocol. This section analyzes the impact of reducing the directory size on execution time and on energy consumption. We show that SPEL++ preserves performance, even with considerably reduced directory sizes.

Effectively, Fig. 8 shows the impact on execution time and energy consumption when reducing the number of directory entries from as many entries as the L1 cache (512 entries per bank, coverage ratio 1:1) to 16 entries per bank (coverage ratio 1:32). Regarding execution time (Fig. 8(a)), we can observe that directory protocols already lose performance when the number of entries is half the entries in the L1 cache. The degradation becomes more dramatic as the directory is reduced to less than 64 entries. This is due to an increase in the number of coverage misses. However, under our protocol, execution time is unaffected as the directory size is reduced to only 64 entries for the Polybench suite applications, and even when reduced to 16 entries for Rodinia and SpecOMP. In essence, the directory is employed in SPEL++ only for the data blocks holding synchronization variables or data accessed within critical sections, which represent a very small fraction of the blocks accessed by the applications.

Regarding energy consumption (Fig. 8(b)) we can observe a similar pattern. As expected, the extra invalidations due to directory evictions lead to extra network traffic and more cache misses, which finally impact energy consumption.

Based on these results we draw two conclusions. First, if the xDRF classification could be applied on all applications, the directory size could be reduced up to 8 times without harming performance, a property which would make SPEL++ scale even better. However, since the xDRF classification is so far limited to OpenMP codes, one cannot reduce the directory size without a negative impact on the performance of applications which cannot benefit from such a classification. The second conclusion is that even with a standard 512 entries/bank directory, the efficiency of the directory is increased when co-running applications compiled for SPEL++ and other applications. Since SPEL++ reduces the directory usage to a minimum, the co-running applications benefit from higher directory capacity.

7 Related work

7.1 Classification of memory accesses

While previous proposals classified memory accesses for guiding data placement [12], [18] or optimizing coherence protocols (*e.g.*, reducing self-invalidation [3], reducing directory pressure [16], [39]), SPEL++ relies on the hybrid classification to implement a dual-consistency protocol that minimizes coherence maintenance while providing SC guarantees for legacy code.

State-of-the-art proposals classify memory accesses as private or shared, either at runtime or at compile-time, based on the nature of accessed data. Examples of run-time classifications are OS-based [12], [15], [16], TLB-based [20], [40], or hardware-based [14], [17], [21], [41], [42], [43], [44] methods. TLB-based methods are able to capture more private pages, but at the cost of extra traffic and complexity. Hardware-based classification requires extra hardware support and increases storage costs, which become prohibitive if a large history of accessed blocks have to be tracked, but can be decreased by tracking only currently cached blocks. Had SPEL++ employed a hardware-based classification, numerous extra self-invalidations would have been caused by frequent shared-to-private and private-to-shared re-classifications.

Compile-time classifications [13], [18], [45] rely on standard static analysis, which is hindered by dynamic memory allocation and pointer aliasing, thus classification is either conservative or speculative. Both solutions lead to performance losses, either due to missed optimization opportunities or due to additional support required to recover from mis-speculations. A compile-time classification is presented by Li *et al.* [18] where, in addition to private and shared data, they propose a third category, "practically private", which embeds data that cannot be disambiguated statically, but is expected to be proven private at runtime or to incur minimal sharing. The classification is employed for designing efficient data placement optimizations, which affect performance, but not correctness. On the contrary, the design of a coherence protocol requires strict classification of accesses.

The private-shared nature of memory accesses is strongly connected in semantics with data-race-free properties of code. Classification of code regions has been addressed by Effinger-Dean *et al.* [46], reasoning about interference free regions (IFR) in DRF codes. IFRs are associated to variables (data) and guarantee that while a thread executes the IFR, no other thread can write to the shared variable accessed by the IFR, but there are no guarantees concerning the other variables, as in the xDRF classification. Moreover, xDRF expands as much as possible across synchronization points (locks) and includes non-overlapping and non-adjacent DRF regions, to maximize the granularity of safe xDRF regions.

Singh *et al.* [19] advocate hybrid classification and combine static and over-conservative, fine-grain classification with dynamic coarse-grain classification of memory accesses. Thus, a memory access is considered safe (private or read-only) if it is classified as such by at least one of the analyses (static or dynamic). The static analysis identifies as safe only data that is guaranteed to be thread-local or read-only. Hence, dynamically allocated variables, global or static variables are marked as unsafe. Moreover, if an instruction may access both safe and unsafe data (*e.g.*, a pointer dereference which can access both safe and unsafe data), would demote all safe data it may touch to unsafe. In consequence, safe data is restricted only to locations that are thread-local and can only be accessed by safe instructions. Vice-versa, memory accesses are safe if they access only safe data. This conservative static analysis is complemented by a dynamic classification of memory pages. The proposal extends the OS memory page protection mechanism with one byte per memory page per thread (instead of per processor), thus being able to keep track of the private, shared read-only and shared read-write memory pages. On one hand, compile-time classification is fine grained (with the granularity of the data accessed by each memory instruction), but it must be over-conservative due to pointer aliasing and dynamic memory allocation; on the other hand, dynamic classification can alleviate the limits of the static analysis using runtime information and can identify more accesses as safe, but has the drawback of a coarse granularity of a memory page. Such a classification is useful for enhancing commodity hardware with additional hardware resources to provide SC both for data race free and for racy programs.

To alleviate the problem of dynamic memory allocation or pointer aliasing, notoriously difficult for static analysis, we took a different approach. Rather than classifying memory accesses based on the nature of accessed data, we designed a compile-time classification per code regions in a class of applications obeying the paradigm that all memory accesses are private or read-only within the boundaries of a certain region: OpenMP and automatically parallelized applications. The compile-time classification is enhanced with runtime support provided by an OS-based classification.

7.2 Cache coherence techniques

Traditional coherence protocols are oblivious to the code's characteristics and enforce the Single-Writer-Multiple-Readers (SWMR) invariant for every memory block in the system, by invalidating all copies of a block upon write misses. Copies are detected by a directory that tracks every memory block stored in a private cache.

Cuesta *et al.* [16], [39] proposed to classify memory accesses and deactivate coherence for private data. The observation is that most of the accessed data is thread local, in consequence it does not require invalidation nor to be kept track of in the directory. Thus, the effectiveness of the directory is increased.

Recent proposals exploit DRF codes and propose relaxed consistency protocols in the shape of SC-for-DRF [8]. SC-for-DRF protocols rely on the guarantee that, within DRF regions, threads perform either private or read-only memory accesses [2], [3], [9], [10], [11], [47], [48], [49], [50]. Cache coherence is thus immune to the order of memory operations (within DRF regions), which enables more flexibility in the coherence protocol and leads to higher scalability, performance and energy efficiency. SC-for-DRF protocols exhibit significant advantages such as reducing access latency and directory pressure, alleviating blocking and diminishing protocol traffic. They perform self-invalidation at synchronization points: Lebeck and Wood use self-invalidation to limit the number of cache blocks registered in the directory [9], SARC coherence [47] employs self-invalidation and implements a writer prediction to avoid the directory indirection upon downgrades. In DeNovo [2], [10], [48] a compiler inserts self-invalidating instructions based on source code annotations. DeNovo implements a directory that tracks the writers, but not the readers, so it relies on downgrading *registered* copies upon read misses. VIPS [3], [11], [49], [50] employs a write-back policy for private blocks, which provides efficiency, and a write-through-policy for shared blocks, providing simplicity. It employs both self-invalidation and self-downgrade, thus removing the need of a

directory structure. To reduce self-invalidation, SPEL++ relies on the OS-based classification to identify the nature of the memory access (private/shared) and on the compiler to indicate the points of synchronization that indeed require self-invalidating cached data (*e.g.* barriers), which considerably improves the cache hit rate, performance and energy consumption.

More importantly, SC-for-DRF protocols cannot guarantee SC for non-DRF codes, leading to undefined behavior. This *breaks compatibility with legacy non-DRF software*, yielding such protocols impractical. As a dual-consistency protocol, SPEL++ provides a natural solution to this shortcoming, by relying on the compiler to identify *regions that can be safely executed under an SC-for-DRF protocol*, and ensuring support for non-DRF regions with a traditional SC protocol. Hence, compile-time delineation of xDRF regions plays a crucial role. Note that if the delineation of xDRF regions is not performed, e.g., for legacy applications, the code is still executed correctly, although without optimizing it.

Previous attempts to reduce self-invalidation require hardware support. Ashby *et al.* [51] perform selective self-invalidation of data that might have been updated by other cores using hardware Bloom filters. The bloom filters are reset only on barriers, which decreases their efficiency. DeNovoND [48] performs selective self-invalidation upon lock synchronization, using a hardware queue lock. However, both proposals have the drawback of (i) trading information accuracy for reducing hardware support and (ii) incurring very expensive self-invalidation since all cache tags must be matched against the filter. VIPS-M [3] and DeNovoSync [10] reduce self-invalidation in synchronization by using exponential back-off mechanisms for spin-waiting loops. Callbacks [11] reduce self-invalidation by employing specific loads that block at the LLC waiting for a new write to be performed. Although these techniques reduce the number of self-invalidations, they still require marking synchronization, thus not supporting legacy codes. In contrast, SPEL++ minimizes self-invalidation by using a precise classification, while still supporting legacy codes.

Finally, given the wide adoption of the Total Store Order (TSO) consistency model in commodity processors (*e.g.*, x86 or SPARC), TSO-CC [7] presents a scalable protocol that guarantees TSO. Although TSO-CC yields similar performance to a directory protocol, its advantage lies in the reduction of the area required by the directory structure, namely, a single pointer to the last writer. RC3 [52] improves TSO-CC by getting advantage of the information about synchronization in DRF applications, if exposed to the hardware. Both TSO-CC and RC3 can also benefit and from SPEL++ by relaxing the consistency model for private or xDRF accesses.

8 Conclusions

The dual consistency cache coherence protocol SPEL++ presented in this paper adapts dynamically to the code's behavior, switching between highly optimized and restrictive modes to guarantee the strongest consistency model while improving scalability, performance, and energy consumption.

SPEL++ optimizations rely on a static-dynamic hybrid classification. Memory accesses classified as private by the OS require minimum coherence support, compiler delineated xDRF parallel regions of code execute safely under an optimized SC-for-DRF protocol, while coherence for the remaining memory accesses is maintained with a standard directory protocol that manages efficiently data races and provides support for legacy code.

SPEL++ outperforms a traditional directory protocol from 19% to 38% and achieves savings in energy consumption from 47% to 53% (average for different benchmarks suites). All in all, SPEL++ achieves *scalability, performance* and *energy efficiency* and ensures compatibility with *legacy* software.

Acknowledgments

This work was supported in part by the "Fundación Seneca-Agencia de Ciencia y Tecnología de la Región de Murcia" under grant "Jóvenes Líderes en Investigación" 18956/JLI/13, as well as by the Swedish Research Council UPMARC Linnaeus Centre, the VR frame project "Efficient Modeling of Heterogeneity in the Era of Dark Silicon": 106201305/C0533201 and by the European 7th Framework Programme (EU ICT-287759) through a collaboration grant from HiPEAC.

References

[1] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why on-chip cache coherence is here to stay," *Communications of the ACM*, vol. 55, no. 7, pp. 78–89, Jul. 2012.

[2] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," in *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011, pp. 155–166.

[3] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 241–252.

[4] "Intel Xeon Phi Coprocessor," http://software.intel.com/en-us/mic-developer, Apr. 2013. [Online]. Available: http://software.intel.com/en-us/mic-developer

[5] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers (TC)*, vol. 28, no. 9, pp. 690–691, Sep. 1979.

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

[7] M. Elver and V. Nagarajan, "TSO-CC: Consistency directed cache coherence for tso," in *20th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2014, pp. 165–176.

[8] S. V. Adve and M. D. Hill, "Weak ordering – a new definition," in *17th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1990, pp. 2–14.

[9] 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)*, Jun. 1995, pp. 48–59.

[10] H. Sung and S. V. Adve, "DeNovoSync: Efficient support for arbitrary synchronization without writer-initiated invalidations," in *15th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2015, pp. 545–559.

[11] A. Ros and S. Kaxiras, "Callback: Efficient synchronization without invalidation with a directory just for spin-waiting," in *42nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2015, pp. 427–438.

[12] 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)*, Jun. 2009, pp. 184–195.

[13] J. Meng and K. Skadron, "Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling," in *Int'l Conf. on Computer Design (ICCD)*, Oct. 2009, pp. 282–288.

[14] 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)*, Sep. 2010, pp. 465–476.

[15] D. Kim, J. Ahn, 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)*, Sep. 2010, pp. 111–122.

[16] 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)*, Jun. 2011, pp. 93–103.

[17] 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)*, Oct. 2011, pp. 45–55.

[18] Y. Li, R. G. Melhem, and A. K. Jones, "Practically private: Enabling high performance cmps through compiler-assisted data classification," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 231–240.

[19] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-end sequential consistency," in *39th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2012, pp. 524–535.

[20] A. Ros, B. Cuesta, M. E. Gómez, A. Robles, and J. Duato, "Temporal-aware mechanism to detect private data in chip multiprocessors," in *42nd Int'l Conf. on Parallel Processing (ICPP)*, Oct. 2013, pp. 562–571.

[21] J. Zebchuk, B. Falsafi, and A. Moshovos, "Multi-grain coherence directories," in *46th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2013, pp. 359–370.

[22] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[23] A. Schrijver, *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., 1986.

[24] P. Feautrier, "Dataflow analysis of scalar and array references," *Int'l Journal of Parallel Programming (IJPP)*, vol. 20, no. 1, pp. 23–53, Feb. 1991.

[25] A. Ros and A. Jimborean, "A dual-consistency cache coherence protocol," in *29th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, May 2015, pp. 1119–1128.

[26] "OpenMP tutorial," website, 2015. [Online]. Available: https://computing.llnl.gov/tutorials/openMP/

[27] "GOMP library," website, Oct. 2005. [Online]. Available: http://gcc.gnu.org/projects/gomp/

[28] "LLVM Intermediate Representation," website, 2015. [Online]. Available: http://llvm.org/docs/LangRef.html

[29] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM Int'l Symp. on Code Generation and Optimization (CGO)*, Mar. 2004, pp. 75–88.

[30] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the IEEE futurebus," in *13th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1986, pp. 414–423.

[31] R. Fernández-Pascual, A. Ros, and M. E. Acacio, "Are distributed sharing codes a solution to the scalability problem of coherence directories in manycores? An evaluation study," *Journal of Supercomputing (SUPE)*, pp. 1–27, Dec. 2015.

[32] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[33] 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)*, Apr. 2009, pp. 33–42.

[34] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2009, pp. 469–480.

[35] Standard Performance Evaluation Corporation, "SPEC OMP2012," http://www.spec.org/omp2012. [Online]. Available: http://www.spec.org/omp2012

[36] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Int'l Symp. on Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.

[37] "Polybench," http://www.cse.ohio-state.edu/~pouchet/software/polybench/, Nov. 2011. [Online]. Available: http://www.cse.ohio-state.edu/~pouchet/software/polybench/

[38] A. Ros, B. Cuesta, R. Fernández-Pascual, M. E. Gómez, M. E. Acacio, A. Robles, J. M. García, and J. Duato, "EMC^2: Extending magny-cours coherence for large-scale servers," in *17th Int'l Conf. on High Performance Computing (HiPC)*, Dec. 2010, pp. 1–10.

[39] B. Cuesta, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Increasing the effectiveness of directory caches by avoiding the tracking of non-coherent memory blocks," *IEEE Transactions on Computers (TC)*, vol. 62, no. 3, pp. 482–495, Mar. 2013.

[40] A. Esteve, A. Ros, M. E. Gómez, A. Robles, and J. Duato, "Efficient tlb-based detection of private pages in chip multiprocessors," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, Mar. 2015.

[41] M. Alisafaee, "Spatiotemporal coherence tracking," in *45th IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2012, pp. 341–350.

[42] J. J. Valls, A. Ros, J. Sahuquillo, M. E. Gómez, and J. Duato, "PS-Dir: A scalable two-level directory cache," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 451–452.

[43] M. Davari, A. Ros, E. Hagersten, and S. Kaxiras, "The effects of granularity and adaptivity on private/shared classification for coherence," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 12, no. 3, pp. 26:1–26:21, Aug. 2015.

[44] ——, "An efficient, self-contained, on-chip, directory: DIR$_1$-SISD," in *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015, pp. 317–330.

[45] 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)*, Sep. 2010, pp. 501–512.

[46] L. Effinger-Dean, H.-J. Boehm, D. Chakrabarti, and P. Joisha, "Extended sequential reasoning for data-race-free programs," in *2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, Jun. 2011, pp. 22–29.

[47] S. Kaxiras and G. Keramidas, "SARC coherence: Scaling directory cache coherence in performance and power," *IEEE Micro*, vol. 30, no. 5, pp. 54–65, Sep. 2011.

[48] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient hardware support for disciplined non-determinism," in *18th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2013, pp. 13–26.

[49] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," in *40th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2013, pp. 535–547.

[50] A. Ros, M. Davari, and S. Kaxiras, "Hierarchical private/shared classification: the key to simple and efficient coherence for clustered cache hierarchies," in *21th Int'l Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2015, pp. 186–197.

[51] T. J. Ashby, P. Díaz, and M. Cintra, "Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters," *IEEE Transactions on Computers (TC)*, vol. 60, no. 4, pp. 472–483, Apr. 2011.

[52] M. Elver and V. Nagarajan, "RC3: Consistency directed cache coherence for x86-64 with RC extensions," in *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015, pp. 292–304.

Alberto Ros received the MS and PhD degree in computer science from the University of Murcia, Spain, in 2004 and 2009, respectively. In 2005, he joined the Computer Engineering Department at the same university as a PhD student with a fellowship from the Spanish government. He has been working as a postdoctoral researcher at the Universitat Politècnica de València and at Uppsala University. Currently, he is Associate Professor at the University of Murcia. His research interests include cache coherence protocols memory hierarchy designs, and memory consistency for manycore architectures.

Alexandra Jimborean received her PhD from the University of Strasbourg, France, researching on compile-time analysis and optimization of loops using the polyhedral model to enable automatic, dynamic and speculative parallelization. During the PhD studies (2009-2012), she held an academic grant from the French Ministry of High Education and Research and received a Google Anita Borg Memorial scholarship in recognition of her research. She continued as a post-doctoral fellow in Uppsala University, Sweden (2012-2014), held a researcher position within the same department (2014-2015) and became Associate Senior Lecturer (equiv. Assistant Professor) in May, 2015.