2015 IEEE 29th International Parallel and Distributed Processing Symposium

A Dual-Consistency Cache Coherence Protocol

Alberto Ros
University of Murcia
aros@ditec.um.es

Alexandra Jimborean
Uppsala Universitet
alexandra.jimborean@it.uu.se

***Abstract*—Weak memory consistency models can maximize system performance by enabling hardware and compiler optimizations, but increase programming complexity since they do not match programmers' intuition. The design of an efficient system with an intuitive memory model is an open challenge.**

This paper proposes SPEL, a dual-consistency cache coherence protocol which simultaneously guarantees the strongest memory consistency model provided by the hardware and yields improvements in both performance and energy consumption. The design of the protocol exploits a compile-time identification of *code regions* which can be executed under a less restrictive, thus optimized protocol, without harming correctness. Outside these regions, code is executed under a more restrictive protocol which enforces sequential consistency. Compared to a standard directory protocol, we show improvements in performance of 24% and reductions in energy consumption of 32%, on average, for a 64-core chip multiprocessor.

I. Introduction

Given the prevalence of multi-core processors and the trend of continuously increasing the number of cores, the efficiency of coherence protocols becomes of utmost importance. However, not only traditional protocols perform sub-optimally on modern architectures, but their inefficiency escalates as the number of cores in the system grows. State of the art coherence protocols seek to detect and exploit memory accessing characteristics of code with the goal of simplifying the protocol while delivering scalability and performance [1], [2]. Despite the promising results, such protocols exhibit limitations (for instance, fail at providing support for legacy code), and were therefore disregarded for being integrated in the emerging architectures (*e.g.*, Intel Xeon Phi [3]), which still implement traditional, inefficient, directory-based cache coherence protocols.

One source of inefficiency is that coherence protocols have been traditionally designed to provide the strongest consistency model (or memory model): sequential consistency (SC). This design decision eases the development of the protocol by isolating the cache coherence protocol from the consistency model provided by the multi-core. However, it comes at the cost of performance limitations, especially when the system provides a more relaxed consistency model [4]. In answer, modern coherence protocols follow the sequential consistency for data-race-free (SC for DRF) model [5], which allows a simpler and more scalable design [1], [2] and improves performance [6]. Nevertheless, a major drawback is that such protocols do not provide backwards compatibility with existing software that requires a stronger consistency model.

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 [7], [8], [9], [10], [11], [12], [13], [14], [15] for optimizing the coherence protocol or, for example, enhancing data placement. While these optimized coherence protocols outperform traditional protocols, the underlying techniques for classifying accesses still lack accuracy (Section II), 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 [7], [10], [11] or increases hardware complexity [9], [12], [15]. Performed at compile-time, it must remain conservative, due to memory accesses which cannot be fully disambiguated statically [8], [13].

This work proposes SPEL, a dual-consistency cache coherence protocol that provides SC, while relaxing the coherence protocol only during the execution of data race free codes. SPEL stands for **S**calability, **P**erformance, **E**nergy efficiency and support for **L**egacy code. By relaxing the protocol, SPEL achieves *high performance and scalability*, and, by delivering SC, it *ensures compatibility with legacy software*. Key to this ability is a compile-time identification of *extended data-race-free code regions (xDRF)*. Each xDRF region consists of a set of data-race-free (DRF) [16] regions that can be executed under a high-performance and scalable SC-for-DRF protocol, thus providing SC (Section III). For non-xDRF regions, coherence is ensured by a standard directory protocol (SC protocol). The proposed design smoothly blends both protocol modes, which can be simultaneously active: threads executing non-DRF regions 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, to maximize performance (Section IV).

Unlike previous approaches which focus on classifying memory accesses as private or shared based on data classification, 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 exhibit parallelization opportunities, by applying polyhedral transformations [17], [18].

1530-2075/15 $31.00 © 2015 IEEE
DOI 10.1109/IPDPS.2015.43

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 coprocessor [3] (Section V). Experiments show an average performance improvement of 24% while reducing energy consumption by 32% (Section VI).

II. Background and Motivation

Modern coherence protocols focus on a three-fold goal: scalability, performance and energy consumption [19]. This paper describes an *optimized SC-for-DRF coherence protocol* to efficiently maintain coherence during the execution of xDRF regions and successfully address the three-fold goal. The optimized protocol is complemented by a traditional protocol, enabled during the execution of non-xDRF regions, maintaining compatibility with existing code. Next subsections discuss the advantages and drawbacks of SC-for-DRF protocols and of previous techniques for classifying memory accesses, and describe our approach for overcoming their shortcomings.

A. Sequential consistency for DRF protocols

SC-for-DRF protocols rely on the guarantee that, during DRF regions, threads perform either private or read-only memory accesses [1], [2], [20]. A memory access is private if it targets a memory location that is only accessed by one thread during the execution of one DRF region; and is read-only if the location is not written within the DRF region. Cache coherence is thus immune to the order of memory operations during 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 (Section IV-B). Given the granularity of a memory access finer than a cache block, coherence of DRF regions can be maintained by an SC-for-DRF protocol [5], if false sharing problems are handled appropriately (Section IV-B1).

Limits and solutions: Previously proposed SC-for-DRF protocols conservatively impose self-invalidation of cached data in *each synchronization point* (regarded as boundaries of DRF regions). This excessive invalidation limits their performance [1], [2]. In contrast, SPEL reduces self-invalidation, by relying on the compiler to indicate the points of synchronization that indeed require self-invalidating cached data. SPEL considerably improves the cache hit rate, and consequently, performance and energy consumption (Section VI).

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.

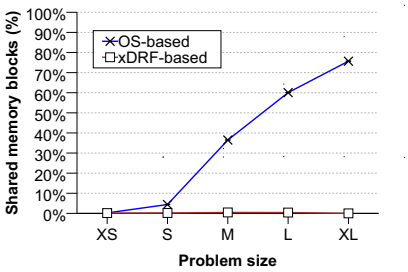


Fig. 1. Shared blocks w.r.t. problem size (average of all Polybench apps)

B. Classification of memory accesses

Data-race-free properties are strongly connected in semantics with the private-shared nature of memory accesses. While previous proposals classified memory accesses for guiding data placement [7], [13] or optimizing coherence protocols (*e.g.*, reducing self-invalidation [2], reducing directory pressure [11], [21]), SPEL relies on the delineation of xDRF regions to minimize the coherence maintenance while providing SC guarantees, *for any code*. Next we review the most widely adopted approaches for memory access classification.

1) OS-based classification: The operating system (*i.e.* OS-based classification) [7], [10], [11] classifies a memory access as private if the *entire accessed page* is referenced only by one thread, *during the entire execution*. Hence, in practice, the majority of memory accesses are finally classified as shared.

Limits and solutions: OS-based classification is highly over-conservative. To unleash optimizations, SPEL relies on a classification of xDRF regions, where a memory access is private (or read-only) if the *target address* is accessed by a unique thread *only within the xDRF region*. Therefore, we consider both a finer granularity (size of accessed data vs. memory page) and temporality (*i.e.*, the classification is valid during the execution of the xDRF region, see Section III).

Preliminary experiments on the Polybench benchmark suite [22] evaluate the effectiveness of these classification strategies. Fig. 1 clearly emphasizes the limits of the OS-based classification for large inputs, as the programs run longer, the majority of memory accesses are marked as shared, whereas the xDRF-based classification identifies such accesses as private. Had SPEL relied on the OS-based classification, a large part of code (if not all) would have been classified as non-DRF, missing enormous optimization opportunities.

2) Standard compiler-based classification: Previous approaches propose compile-time classification of memory accesses, based on the nature of accessed *data* [8], [13], [23].

Limits and solutions: Standard compile-time classification is hindered by the classical problem of static disambiguation of individual data. In contrast, we target *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 (Section III). In consequence, our proposal is not hindered by dynamic memory allocation or pointer aliasing.

III. 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. The DRF regions may be interleaved with non-DRF (nDRF) regions, but the nDRF regions do not belong to the xDRF region. 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 a nDRF region, since no other thread can access it; (2) all properties that hold in a DRF region, hold across the entire xDRF region.

A. xDRF regions in OpenMP codes

The OpenMP programming model 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, critical sections) . Should synchronization mechanisms be used with moderation in scalable parallel programs, most of the parallel code represents xDRF regions.

Identification of xDRF regions is based on the semantics of each OpenMP construct. 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 dependences and no parallel updates of shared variables).

```
#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

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()`.

The OpenMP framework provides several solutions to avoid data races. *The privatization approach* (*e.g.*, private, thread-private, firstprivate, lastprivate clauses) uses the copy-in / copy-out strategy to copy the value of a shared variable into a thread-local, such that each thread can safely update its private copy. Finally, the private copies may be merged back into the shared variables at the end of the parallel region, according to the semantics of the privatization clause. Internally, these thread-private variables are declared in the code section executed by 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.

Reductions provide the means to accumulate the values of the thread-private variables in a shared copy. In the transformed code, reduction operations are either protected by locking mechanisms or the shared variable is updated with an atomic instruction. Such code regions are classified as nDRF.

Similarly, other *synchronization mechanisms* (*e.g.*, critical, atomic, barrier directives) 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.[1] On the other hand, regions corresponding to directives or work-sharing constructs which indicate that the code is executed only once by one thread (*e.g.*, $\#pragma\ omp\ single$, $\#pragma\ omp\ task$) are marked as xDRF, and the updated data is made visible to the other threads via synchronization mechanisms ($\#pragma\ omp\ taskwait$), which are nDRF.

Note that accesses to *data annotated as shared* by the programmer via the OpenMP $shared$ clause, 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 a *scalar* declared as shared require synchronization, *e.g.*, $critical\ section$, and are therefore handled as nDRF.

B. Instructions delimiting regions of code

The code is compiled in two steps using LLVM [24]. 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

[1]Although a critical section is DRF, it cannot be embedded in a larger xDRF region, since coherence is required between threads entering the same critical section. In this proposal, critical sections are handled as nDRF, to avoid splitting the enclosing xDRF region into smaller xDRF regions. Larger xDRF regions reduce self-invalidation, and therefore, execution time.

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.

The *drf.flush* instruction delimits xDRF regions, where data modified in the SC-for-DRF protocol mode must be propagated. Since *drf.flush* instructions marks 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 proposals.

An example is 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 more work 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.

```
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 from Listing 1 when the clause schedule(runtime) is specified

IV. Dual consistency cache coherence protocol

Relying on the presented compile-time classification of xDRF regions, we propose SPEL, a dual consistency cache coherence protocol, where memory accesses within xDRF regions are kept coherent by an SC-for-DRF protocol, *i.e.*, writes are guaranteed to be made visible no later than the end of the xDRF region, whereas memory accesses in a nDRF region follow a standard directory protocol ensuring SC, *i.e.*, writes are propagated immediately.

A. SC coherence protocol

Our SC coherence protocol is a traditional invalidation-based MOESI directory protocol [25] 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 the sharers) of the cached blocks. The directory controller forwards the request 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 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 an acknowledge message, or with the data block in case of the owner of the block. These messages are sent to the requester. Once the requester receives all messages, it can perform the write.

All transactions finish with an *unblock* message from the requester to the directory controller, which remains blocked while the request is processed, until it receives the *unblock*.

B. SC-for-DRF coherence protocol

In SC-for-DRF (*DRF*=1), memory requests do not modify the coherence status (*e.g.*, cache MOESI states, directory information, etc). Hence, blocks cached during the SC-for-DRF mode 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 cache block. 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 SC 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 required by the SC protocol must be visible to the SC-for-DRF protocol, so on a read miss, the SC copy of the block is sent by the owner of the block.

The key performance benefits of SPEL in SC-for-DRF mode 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 making a better use of the directory.
- As a consequence, invalidation requests performed by the SC protocol do not affect the *F* blocks, thus reducing the cache miss rate (particularly, due to false sharing misses) and, in consequence, reducing traffic.

- Thanks to the DRF properties, writes can be performed without waiting for write permission or invalidation of other copies, thus reducing the average memory latency.
- Directory blocking is considerably reduced, since it is only required when the up-to-date copy of the block is in a private cache. Thus both (i) network traffic is reduced, as less *unblock* messages will be generated, and (ii) waiting time is diminished.

1) 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 cache block. In traditional SC protocols, the block is marked as invalid and updated later, which causes important performance degradation, but SC-for-DRF protocols must handle false sharing appropriately to maintain correctness.

Our compile time classification ensures data-race freedom with a granularity finer than the cache block. Hence, during SC-for-DRF coherence mode several cores can write on 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 during SC-for-DRF mode.

2) Hardware support for handling false sharing: Ideally, in order to store 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. However, this overhead can be reduced by storing 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. A new cache-like structure is required for keeping track of the written bytes: the *written-bits cache*. While this alternative increases the width of the structure due to the addition of a tag field, it can considerably reduce its height. Effectively, as we show in Section VI-B, a written-bits cache with only 32 or 16 entries is sufficient for obtaining similar performance to having as many entries as cache entries.

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 block, thus making them visible to the SC protocol, as explained below.

C. From SC-for-DRF to SC: Propagating SC-for-DRF writes

Writes performed in SC-for-DRF mode 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, (ii) the corresponding entries in the written-bits cache are evicted, or (iii) upon write operations in SC mode. The second scenario occurs at the end of xDRF regions.

Upon the eviction of a dirty *F* block (or its written-bits entry), data are written back to the shared cache. Previously all SC copies of the same block, which are tracked by the directory, must be invalidated from the private caches and written-back, if dirty, to the shared cache. The evicted *F* block

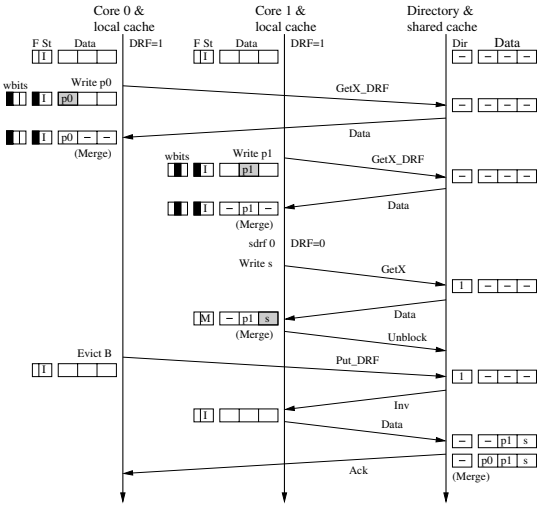


Fig. 2. Merging private data on evictions. Gray boxes indicate when the processor write is effective. Block B is initially stored in the shared cache, containing three data locations: p_0 will only be accessed by core 0, p_1 only by core 1, and s will be accessed by both cores. At the beginning, the *DRF* flag is set in both threads. 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 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 acknowledgement is sent to Core 0. The evicted data is now also visible to other threads.

is then merged with the current copy, and from this point on, it will be visible to other threads. Fig. 2 details this process.

Upon a write in SC mode, tracked copies are invalidated and the data block is sent to the requester. If there is already an *F* block residing in the requester's cache, it is "merged" with the current copy, resetting *F* and the written bits (Fig. 3).

When the processor executes the *drf.flush* instruction, every dirty *F* block in its local cache is evicted and the instruction does not commit until every eviction has been acknowledged (see Fig. 2). The overall performance penalty of the *drf.flush* instructions is negligible since they occur infrequently and they do not evict blocks stored in cache by the SC protocol.

D. Thread migration

The operating system can decide at runtime to execute a thread in a different core. We guarantee cache coherence by forcing threads to execute an *drf.flush* instruction before they are de-scheduled. This instruction makes the SC-for-DRF writes visible to the SC coherence protocol, and therefore, they can be seen from the new core.

E. Multitasking, simultaneous multi-threading, and syscalls

Both applications compiled to expose xDRF regions, as well as other applications can execute on the same core. When two

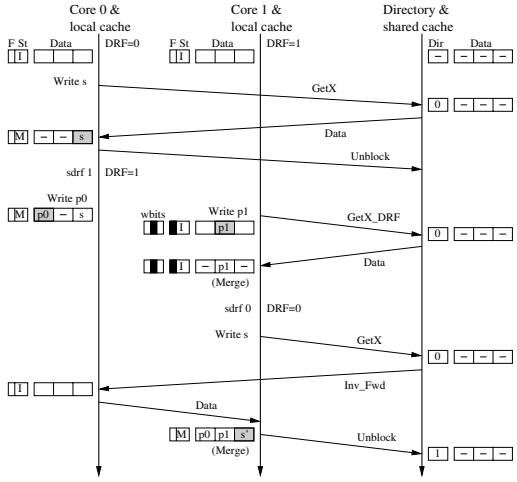


Fig. 3. Merging private blocks on writes to shared data. Core 0 starts with the *DRF* flag unset and Core 1 with the flag set. First, Core 0 writes s. Next, *DRF* flag is set in Core 0, and it writes p_0. Since the state of the block is M (modified), the SC-for-DRF write will be visible, and *F* and *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').

applications in SC-for-DRF mode coexist on the same core, an *drf.flush* instruction will write-back all *F* blocks 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 de-scheduled, 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 in its 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, they 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.

F. Protocol scalability

SPEL requires a number of written-bits entries for each private cache. The area overhead entailed by them does not depend on the number of cores, so it is a scalable structure.

When running in SC-for-DRF mode, 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 running in SC coherence mode.

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. In SC-for-DRF mode, SPEL considerably reduces directory blocking during the resolution of cache misses and it does not track sharers, thus directory availability and productivity are highly increased, preventing it from becoming a bottleneck.

TABLE I
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

V. SIMULATION ENVIRONMENT

We evaluate SPEL using the GEMS simulator [26], a cycle-accurate simulator for multiprocessor systems. The interconnection network has been modeled with GARNET [27]. We have modified GEMS in order to model SPEL in detail accounting for the cost of the new inserted instructions. To report energy consumption we have used the McPAT tool [28] 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 coprocessor [3]. 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 [14] and are complementary to this work. The focus of this paper is the cache coherence protocol. The parameters of the simulated architecture are shown in Table I.

We compare the proposed dual consistency protocol (SPEL) to a traditional SC protocol (Directory) and a state-of-the-art SC-for-DRF protocol (VIPS) [2] on a wide variety of applications from codes parallelized with OpenMP, SpecOMP 2012 [29] (352.nab, 359.botsspar, and 367.imagick –test input–) and Rodinia [30] (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 [22] (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

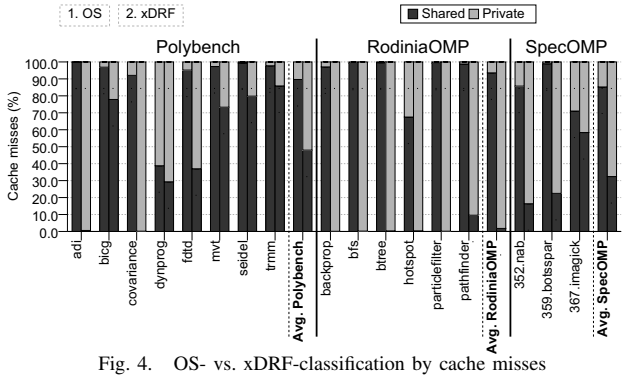


Fig. 4. OS- vs. xDRF-classification by cache misses

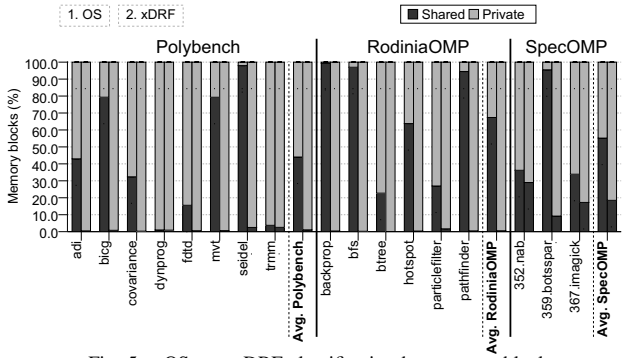


Fig. 5. OS- vs. xDRF-classification by memory blocks

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.

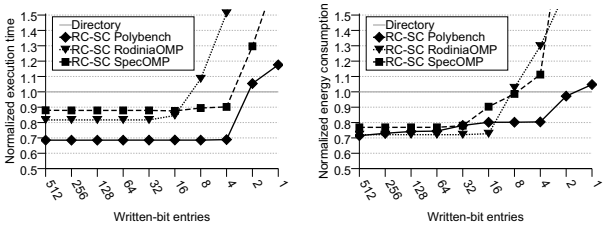
VI. Results

A. Effectiveness of the xDRF classification

This section compares the xDRF compile-time classification of code regions employed in SPEL to the run-time classifications based on data sharing, *e.g.*, OS-based classification, used by VIPS. The comparison is performed with respect to two key aspects for the protocol optimization: cache misses and accessed memory blocks. Results for 64 threads are plotted in Figs. 4 and 5. The *shared* bars represent the misses or blocks that cannot be optimized, *e.g.*, inside nDRF regions using the xDRF classification or within shared pages using the OS classification, while *private* represents misses or blocks that can be optimized.

The ratio of private misses is important because misses under the SC-for-DRF protocol are resolved faster and require less traffic. Fig. 4 shows that even with a large number of threads (64), when lock contention, and consequently, the number of shared misses increases, the number of optimized accesses is higher in the xDRF classification. Particularly, the fraction of shared misses is reduced by 42% for the Polybench applications, 90% for Rodinia, and 53% for SpecOMP.

The ratio of private memory blocks is also important since it affects directory utilization and contention. Fig. 5 shows that xDRF classification exhibits a higher rate of memory blocks of which the directory is not required to keep track, compared to *OS*. This is a consequence of the xDRF classification labeling data as private if it maintains this state during the execution



(a) Execution time (b) Energy consumption

Fig. 6. Performance variations due to written-bits cache size

of the xDRF region, while the OS approach requires data to be private during the entire application's execution. Therefore, the *xDRF* classification significantly outperforms the OS-based classification in both cases, thanks to a finer granularity (byte vs memory page) and temporality (xDRF region vs entire application). In particular, the ratio of private accessed memory blocks is increased by 41% for Polybench, 67% for Rodinia, and 37% for SpecOMP.

B. Area requirements

SPEL employs a written-bits cache that tracks dirty data for private blocks. This section performs a sensitivity analysis of the number of entries of this structure. Fig. 6 shows the consequences of reducing the number of entries (averages over each of the three evaluated benchmark suites). 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 protocol, which does not require this structure.

As the number of entries is reduced, performance of SPEL is unaffected up to only 16 or 32 entries for Rodinia and 4 entries for Polybench and SpecOMP, as Fig. 6(a) shows, since there are not many blocks written during SC-for-DRF mode or they are evicted due to cache capacity. The energy consumption of the network and L2 increases as the number of written-bits entries is reduced (Fig. 6(b)), but there is only a slight increase as the number of entries is reduced up to 32 for SpecOMP, 16 for Rodinia, and 4 for Polybench.

Reducing the size of the written-bits cache has an impact on energy consumption sooner than on execution time. This is because during SC-for-DRF coherence, writes do not wait for permission and can be performed immediately but the extra write-backs caused by the eviction of written-bits increase both network traffic and L2 accesses. When this traffic grows considerably, execution time increases due to a bottleneck in the network and cache controller. On the other hand, 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.

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 SPEL with respect to the L1 cache size is only 0.96% (0.37KB). In what follows, the results presented for SPEL consider a 32-entry written-bits cache.

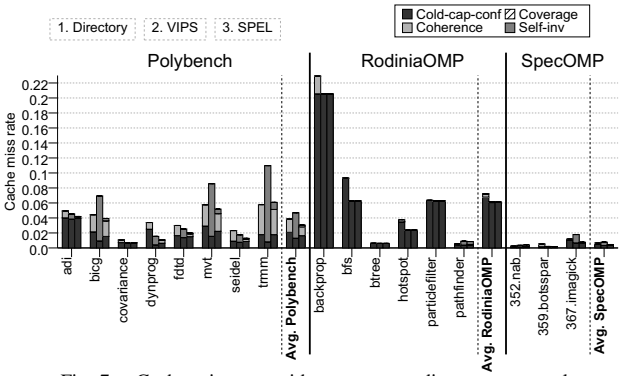


Fig. 7. Cache miss rate with respect to a directory protocol

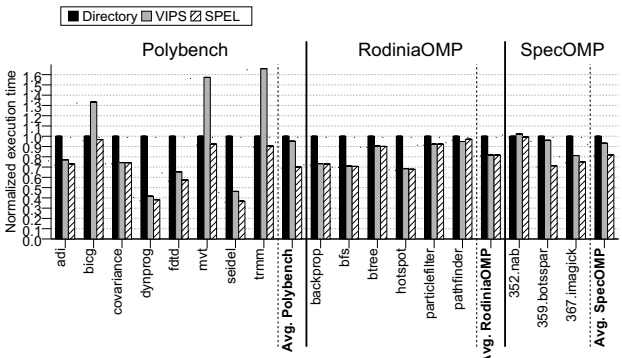


Fig. 8. Execution time improvements with respect to a directory protocol

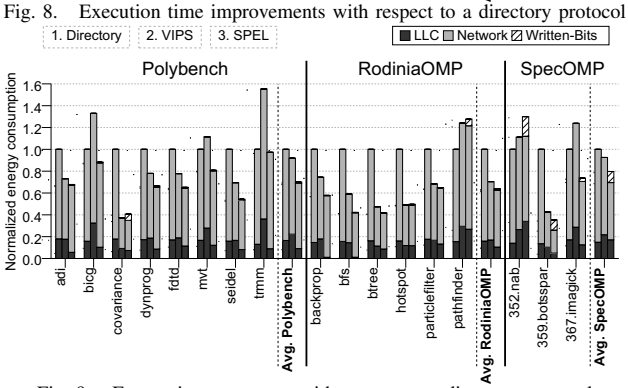


Fig. 9. Energy improvements with respect to a directory protocol

C. SPEL vs traditional vs SC-for-DRF protocols

SPEL is designed for optimizing performance as a SC-for-DRF protocol, while providing support for legacy code, as a traditional SC protocol. This section compares SPEL with an SC protocol (Directory) and a state-of-the-art SC-for-DRF protocol (VIPS [2]). Given that 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 of the cached shared blocks. This enables VIPS to execute codes which would otherwise not be accessible, as races are not exposed by default.

1) Cache miss rate: The cache miss rate varies in all three evaluated protocols. Both VIPS and SPEL in SC-for-DRF mode 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. In Fig. 7, the first bar illustrates the miss rate of the L1 cache in a directory protocol split into three categories: misses due to replacements in the cache (*Cold-cap-conf*), misses as a consequence of invalidations and downgrades generated by remote writes and reads (*Coherence*), and misses that come from invalidations generated by directory evictions (*Coverage*). The second bar shows VIPS, which involves no coherence or coverage misses, but a fourth category of misses due to self-invalidation. The third bar shows the proposed SPEL protocol, thus all type of misses can be encountered.

Compared with Directory, despite the additional misses incurred by self-invalidation, SPEL reduces the total number of misses, by 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. 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 the dual consistency.

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, as shown in Fig. 8. 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 misses caused by cache capacity clearly dominate the ones caused by self-invalidation. Hence, VIPS and SPEL yield similar performance. However, for Polybench and SpecOMP massive self-invalidation impact negatively performance, and SPEL reduces the average execution time by 26% and 12%, on average, compared to VIPS. Compared with the SC protocol, SPEL consistently improves performance for each application. On average, SPEL boosts performance by 30% for Polybench applications, and 18% for Rodinia and SpecOMP.

3) Energy consumption: Fig. 9 illustrates the energy consumption of VIPS and SPEL normalized with respect to Directory. Results show the energy consumption of the network and of the shared cache (LLC), Fig. 6(b), but also the energy consumed by the accesses to the written-bits cache. Since VIPS maintains this information in the MSHR structure, we assume that the extra energy consumption 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. 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. However, SC-for-DRF protocols also 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 30% for Polybench, 36% for Rodinia and 29% for SpecOMP, compared to a directory protocol. Compared to VIPS, average reductions of 24%, 9%, and 23%, respectively, are achieved by SPEL due to the reduction of invalidations and write-backs.

VII. Related work

Classifications: An efficient xDRF classification of code regions is key for designing SPEL. Classification of regions has been addressed by Effinger-Dean et al [31], 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 not to *any* shared variable, 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 the safe xDRF regions.

Previous proposals classify memory accesses as private or shared based on the nature of accessed data, either at runtime or at compile-time. Examples of run-time classifications are OS-based [7], [10], [11] (Section II-B1), TLB-based [32], or hardware-based [9], [12], [15], [33] 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 *all accessed blocks* are tracked, but can be decreased by tracking *only 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 [8], [13], [23] 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.

End-to-End SC [14] combine run-time OS-based and compile-time classifications. A memory access is considered safe (private or read-only) if guaranteed by at least one of the analyses (static or dynamic). This classification enables optimizations in the hardware itself, such as instruction reordering and out-of-order commits of accesses from the write-buffers. There are several synergies between SPEL and End-to-End SC: first, the two classifications are complementary: while xDRF classification is more accurate during the execution of xDRF regions, the static-dynamic approach could further classify accesses within nDRF regions; second, both proposals provide SC while enabling optimizations: End-to-End SC optimizes the processor design and SPEL the coherence protocol. As these techniques are orthogonal, a joint proposal would maximize the optimization opportunities.

Relaxed consistency protocols: SC-for-DRF consistency protocols [1], [2], [6], [34], [35] rely on self-invalidation at synchronization points: Lebeck and Wood use self-invalidation to limit the number of cache blocks registered in the directory [6], SARC coherence [34] employs self-invalidation and implements a writer prediction to avoid the directory indirection upon downgrades. In DeNovo [1] a compiler inserts self-invalidating instructions based on source code annotations. DeNovo implements a directory that tracks the writers, but not the readers, so they rely on downgrading *registered* copies upon read misses. VIPS [2], [35] 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. All these proposals require DRF applications for providing SC, otherwise, they provide SC-for-DRF, which makes them unsuitable for legacy codes, unless all data races are exposed.

SPEL is more general, being able to adapt to the code's characteristics and choose the suitable protocol, thus always providing SC. Additionally, xDRF regions, in contrast to DRF regions, enable the protocol to avoid many self-invalidations and self-downgrades. Thus SPEL outperforms the state-of-the-art SC-for-DRF protocols, as shown in the previous section.

TSO-CC proposes a scalable protocol that guarantees Total Store Order (TSO) given the wide adoption of the TSO consistency model in commodity processors (*e.g.*, x86 or SPARC) [4]. Although TSO-CC gets similar performance to a directory protocol, its advantage lies in the reduction of the area required by the directory structure, which only requires a pointer to the last writer. TSO-CC can also benefit from the proposed dual protocol by relaxing the consistency model for accesses in xDRF regions, and thus, being able to achieve performance improvements.

Hardware support: Ashby et al. [20] perform selective self-invalidation of data that might have been updated by other core by relying on inexact information using Bloom filters. The bloom filters are reset only on barriers, which decreases their efficiency. DeNovoND [36] 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. In contrast, SPEL minimizes self-invalidation by using a precise classification.

VIII. Conclusions

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

SPEL provides optimizations for applications in which the compiler can identify parallel regions of code that can execute under an SC-for-DRF protocol (xDRF). Coherence of nDRF regions is maintained with a standard directory protocol. SPEL

outperforms a traditional directory protocol by approximately 30% for Polybench applications, and 18% for Rodinia and SpecOMP, on average; achieves savings in energy consumption from 29% to 36%, on average; and requires negligible extra hardware resources.

All in all, SPEL achieves *scalability, performance* and *energy efficiency* and ensures compatibility with *legacy* software.

ACKNOWLEDGMENT

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, by the Spanish MINECO, by European Commission FEDER funds, under grant TIN2012-38341-C04-03, as well as by the Swedish Research Council UPMARC Linnaeus Centre and by the VR frame project "Efficient Modeling of Heterogeneity in the Era of Dark Silicon": 106201305/C0533201.

REFERENCES

[1] 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.

[2] 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.

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

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.

[36] 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.