Cache Coherence for Phase-Concurrent Programs

Anson Kahng, Charles McGuffey, and Sam Westrick

May 2017

1 Introduction

In an effort to simplify the complexity and improve the scalability of cache coherence schemes, some researchers have begun looking at the benefits of restricted forms of parallelism. For example, it could be reasonable to assume that parallel programs are *data-race free* (DRF). Under this restriction, it is possible to obtain improved performance with a simplified protocol [2, 3].

However, while DRF programs are common, there are some programs which benefit from "well-behaved" data races. For example, consider the following algorithm, based on the Sieve of Erastosthenes, which generates all primes less than n in parallel.

```
function primes(n) {
  if n < 2 then return {};
  let isPrime = [true : 0 $\leq$ i < n];
  foreach p in primes($\sqrt{n}$) {
    foreach k where 2 $\leq$ k < $\lceil n/p \rceil$ {
      isPrime[p $\cdot$ k] := false;
    }
  }
  return {$i$ : 2 $\leq$ i < n | isPrime[i]};
}
```

This algorithm relies on multiple processors writing to the same flag at the same time (write-write races), and is therefore not DRF. However, it does not exhibit read-write races. In this sense its memory locations are phase-concurrent, meaning that they each alternate between concurrent read and concurrent write phases. During a location's read phase, it is read-only. During a write phase, it is write-only. Note that phases are not global: it is possible for a process to be reading from one cell while writing to another without synchronizing in between.

In the example algorithm above, there is an added guarantee that concurrent writes to the same location will always be writing the same value. In general, we will not require this, as will be discussed in more detail later.

The term "phase-concurrent" is inspired by Shun and Blelloch's work on concurrent hash tables [4].

2 Definitions and Semantics

A *program* consists of P dynamic streams of instructions, where P is the number of processes. An *execution* is an interleaving of those P streams produced by some scheduler, where at each time step, the scheduler lets one stream advance by a single instruction.

Two distinct instructions A and B are *concurrent* if there exist two executions E_1 and E_2 such that $A \prec B$ in E_1 and $B \prec A$ in E_2, where the relation $X \prec Y$ means X appears before Y in a specific execution. In other words, if the relative order of A and B is not constant over all executions, then A and B are defined to be concurrent.

We distinguish between three classes of atomic memory operations: read, write, and read-modify-write (such as compare-and-swap, fetch-and-add, etc). We say that a multi-threaded program is *phase-concurrent* if all concurrent operations at the *same* memory location are of the same class. In our implementation, we discretize memory locations at the level of cache lines; we later describe how to extend this framework to the granularity of individual memory locations.

Under this definition, phase-concurrent programs must rely on read-modify-write (RMW) operations to synchronize effectively. For example, here is a phase-concurrent program using two processes to sum an array a of length $2n$ and print the result. Both processes run the same code. The value p is the identifier of the process (in this example, 0 or 1). Assume there is some shared cell s initially set to 2. We use r to store intermediate results, and the RMW operation sub-and-fetch (which atomically subtracts and fetches the new value) to synchronize.

```
function sum_array(a) {
  // initially s == 2
  r[p] := 0;
  for i where p · n ≤ i < (p+1) · n {
    r[p] := r[p] + a[i];
  }
  if (sub-and-fetch(s,1) == 0) {
    // exactly one process will execute this
    printf("%d\n", r[0] + r[1]);
  }
}
```

Note that this process is actually DRF; we haven't yet discussed how to handle concurrent writes; we do so below.

2.1 Concurrent Writes

When two or more processes concurrently write to the same location, we need to specify what value will be returned at the next read. We propose non-deterministically "choosing a winner".

Specifically, consider a particular execution and a read operation of interest. Identify the write w which occurred most recently before the read, and let W be the set of all writes that are concurrent with w. Partition W into sets W_p containing only the writes issued by process p. For each of these, identify the write w_p which is the last write in W_p. The winning write is chosen non-deterministically from $\{w_p : 0 \leq p < P\}$.

The above specification is a bit nasty, but (we claim) actually quite intuitive. For each memory location, at the end of each of its write phases, we non-deterministically choose one written value to be visible in the following read phase. The value which is chosen must be the "last" write of some process within that phase.

For example, we can modify the array-sum example to print not only the sum of the array, but also the identifier of the process which finished their portion of the array first. To illustrate the requirement that the winning write be the "last" write of some process, consider the assignment $w := 42$ below. In both processes, within the same write phase of w, that value is overwritten. So it will never be visible.

```
r[p] := 0;
for i where p · n ≤ i < (p+1) · n {
  r[p] := r[p] + a[i];
}
w := 42; // this value will never be visible
w := p;
if (sub-and-fetch(s,1) == 0) {
  printf("sum: %d\n", r[0] + r[1]);
  printf("winner: %d\n", w);
}
```

3 WSI Cache Coherence

One of the primary challenges in developing a cache coherence protocol for phase-concurrent programs is managing winners and losers at writes. Some existing techniques can be recycled here. For example, in DeNovo, Choi et al.

proposed reusing the shared LLC as a directory, allowing them to track the "owner" of a modified cache line with no asymptotic space overhead [2]. We will utilize a similar trick. **From now on, we will refer to the *shared LLC* and the *directory* interchangeably.** Note that this approach requires inclusive caching: every line in an L1 cache must also be in the shared LLC.

At the end of a write phase, we need to commit the winning write back to the directory so that it is visible to other processes. Although we could rely on programmer-directed cache self-invalidations, we would rather not do so. Instead, we will conservatively guess that each synchronization instruction is a barrier which signals the end of a phase.

Our first approach to phase-concurrent cache coherence was predicated on the following axioms. Each writer either "wins" or "loses" — as a simplest approach, the first writer wins; the shared LLC acts as a directory, as in DeNovo [2]; the winning writer's data must be flushed at a barrier; and the state can transition due to reads, writes, barriers, and directory interactions. This led to our first cache coherence protocol, which we termed WSI, with three states: **W**inner, **S**hared, and **I**nvalid.

Intuitively, WSI ensures that data can only be in the Shared state during a read phase; during a write phase, data can be in either Winner or Invalid, depending on whether or not the corresponding processor won the write-write race.

For a particular line in an L1 cache, the events which cause state transitions are as follows.

1. **(Wr) Write**: a write to this line, issued by the local processor.
2. **(Re) Read**: a read of this line, issued by the local processor.
3. **(Ba) Barrier**: a read-modify-write of any line, issued by the local processor.
4. **(Co) Conflict**: a message from the directory indicating a write to this line by some other processor.
5. **(Fo) Forward**: a message from the directory indicating a read of this line by some other processor. The cache has to respond to this directory by sending the contents of the line.
6. **(Ev) Eviction**: an eviction of this line due to the local caching policy.

Table 1: WSI cache transitions.

	W	**S**	**I**
Wr	W	W/I	W/I
Re	S	S	S
Ba	I	I	I
Co	W	-	-
Fo	-	S	-
Ev	I	I	I

Although WSI is a correct cache coherence policy for phase-concurrent programs, it suffers from many performance issues because it does not take advantage of privacy, does not track "losers" in each write phase, and sometimes invalidates data too early to be of use; each of these issues is addressed below.

Private data ($W \to D, C, W$): In this protocol, we must always conservatively "reset" (i.e., transition to Invalid) at all barriers. However, if a processor has exclusive access to a memory location, then interacting with the directory and transitioning into Shared or Invalid are inefficient, the intuition being that if no other processor is trying to access the data, there is no need for communication overhead or unnecessary transitions to states that implicitly assume that more than one processor has access to the data. In order to deal with exclusive data, we created two new states, D and C, corresponding to Exclusive Dirty and Exclusive Clean, respectively, in which processors have private ownership of the data.

Losers ($I \to L, I$): By the semantics of phase concurrency, if a processor has lost a write-write race in a write phase, then none of its writes can ever be accepted as valid in this phase. However, because the failed writer will be in Invalid, it has to query the directory upon each attempted write to determine whether or not it won the race. This is clearly quite inefficient and led us to create a new state, **Loser**, from whence all write attempts fail immediately without communication with the directory.

Back-to-back read phases ($S \rightarrow S, O$): Because WSI conservatively transitions to Invalid at every barrier, whether the barrier actually represents a phase transition for each cache line, there is potentially additional inefficiency when a barrier interrupts a continuous read phase, whereupon transitioning into Invalid is unnecessary. Therefore, one optimization is the creation of an Old phase, which itself downgrades to Invalid on a barrier, but can get "renewed" to Shared if the data is, indeed, read after a barrier.

Together, these optimizations transformed our original WSI protocol to DCWSOLI, where each old state in WSI is "expanded" into others as shown in Figure 1.

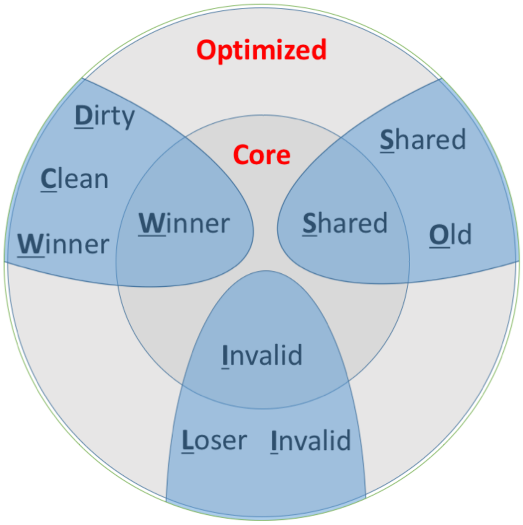

Figure 1: Basic cache coherence protocol and optimizations.

4 DCWSOLI Cache Coherence

With the above optimizations, the three-state WSI protocol turns into a seven-state protocol we term DCWSOLI (Exclusive **D**irty, Exclusive **C**lean, **W**inner, **S**hared, **O**ld, **L**oser, **I**nvalid). We now describe the states, transitions, and correctness of this protocol.

4.1 Local Caches

Each line in an L1 cache can be in one of seven states. We summarize them briefly here, where we list them from "most powerful" to "least powerful".

1. **(D) Exclusive Dirty:** this is the only valid copy of the line in the system, and it is dirty. No other processors have tried to concurrently write to (or read from) this line.
2. **(C) Exclusive Clean:** this is the only valid copy of the line in the system, and it is clean. No other processors have tried to concurrently write to (or read from) this line.
3. **(W) Winner:** this is the only valid copy of the line in the system, and it is dirty. At least one other processor has tried to concurrently write to this line.
4. **(S) Shared:** this is one of (possibly) many valid copies of the line in the system.
5. **(O) Old:** this is one of (possibly) many copies of the line in the system, but it may be invalid. Unless the data is refreshed by a read, this line needs to be self-invalidated at the next synchronization.
6. **(L) Loser:** this line is invalid due to some other processor having concurrently written to this line.
7. **(I) Invalid:** this line is not in the cache.

4.2 Directory

Each line in the directory can be in one of 5 states. The basic idea is to keep track of the status of the owner if possible.

1. **($\mathbf{D_p}$) Registered** p **as dirty**: the only valid copy of this line is in state **D** and is stored in processor p's local cache ($0 \leq p < P$). The local storage reserved for this cache line is overwritten by the identifier p.
2. **($\mathbf{C_p}$) Registered** p **as clean**: the only valid copy of this line is in state **C** and is stored in processor p's local cache ($0 \leq p < P$). The local storage reserved for this cache line is overwritten by the identifier p.
3. **($\mathbf{W_p}$) Registered** p **as winner**: the only valid copy of this line is in state **W** and is stored in processor p's local cache ($0 \leq p < P$). The local storage reserved for this cache line is overwritten by the identifier p.
4. **(V) Valid**: this is one of (possibly) many valid copies of the line in the system.
5. **(I) Invalid**: this line is not in the directory.

For a particular line in the directory, the events which cause state transitions are as follows. They are indexed by the processor identifier of the sender.

1. **($\mathbf{Wr_i}$) Write**: processor i wants to write to this line. The directory may have to respond to this message with either an accept or reject.
2. **($\mathbf{Ac_i}$) Acquire**: processor i wants to read from this line, and is not currently a sharer of it. The directory has to respond to this message with an accept or reject as well as the contents of the line.
3. **($\mathbf{Cl_i}$) Clean**: processor i is downgrading its status from **D** to **C**.
4. **($\mathbf{Sh_i}$) Share**: processor i wants to make its (dirty) copy of this line visible to other processors by downgrading its status from **W** to **O**. The processor also has to send the contents of the line.
5. **($\mathbf{Fl_i}$) Flush**: processor i wants to push this line out of its cache. The processor also has to send the contents of the line.

4.3 Cache Coherence Protocol

The above states and transition events are coordinated in a message-passing style as follows. Transitions marked - are impossible, and those marked ☹ violate phase-concurrency.

Table 2: Transitions for L1 cache i. All messages are sent to (and received from) the directory only. For transitions written A/B, we transition to A if the directory replies with an accept and B if it rejects.

	D	**C**	**W**	**S**	**O**	**L**	**I**
Wr	D	Send $\mathrm{Wr_i}$; D	W	Send $\mathrm{Wr_i}$; D/L	Send $\mathrm{Wr_i}$; D/L	L	Send $\mathrm{Wr_i}$; D/L
Re	D	C	☹	S	S	Send $\mathrm{Ac_i}$; Receive data; C/S	Send $\mathrm{Ac_i}$; Receive data; C/S
Ba	Send $\mathrm{Cl_i}$; C	C	Send $\mathrm{Sh_i}$; Send data; O	O	I	I	I
Co	W	I	-	-	-	-	-
Fo	-	Send data; S	-	-	-	-	-
Ev	Send $\mathrm{Fl_i}$; Send data; I	Send $\mathrm{Fl_i}$; Send data; I	Send $\mathrm{Fl_i}$; Send data; I	I	I	I	-

Table 3: Transitions for the directory.

	$\mathbf{D_p}$	$\mathbf{C_p}$	$\mathbf{W_p}$	**V**	**I**

$\mathbf{Wr_i}$	Assert $i \neq p$;
	Reject i;
	Send Co to p;
	$\mathrm{W_p}$
		if $i = p$ then $\mathrm{D_p}$;
		else: { Accept i;
		Send Co to p;
		$\mathrm{W_i}$}
			Assert $i \neq p$;
			Reject i;
			$\mathrm{W_p}$
				Accept i;
				$\mathrm{D_i}$
					Accept i;
					$\mathrm{D_i}$

$\mathbf{Ac_i}$	☹	Send Fo to p; Receive data from p; Reject i; Send data to i; V	☹	Reject i; Send data to i; V	Retrieve data; Accept i; Send data to i; $\mathrm{C_i}$
$\mathbf{Cl_i}$	Assert $i = p$; $\mathrm{C_p}$	-	-	-	-
$\mathbf{Sh_i}$	-	-	Assert $i = p$; Receive data from p; V	-	-
$\mathbf{Fl_i}$	Assert $i = p$; Receive data from p; V	Assert $i = p$; Receive data from p; V	Assert $i = p$; Receive data from p; V	-	-

Note that there are only five directory states, so there will be no asymptotic space overhead, as these states can be encapsulated in a constant number of specially treated bits (in this case, 3).

Some of the more interesting transitions are discussed below.

- Receiving a **Co** while in **D**: this occurs when another process issues a write after we already wrote to the line. In this scenario, we become the winner and the other line becomes a loser. This is one of the core mechanics of the protocol. It describes the mechanics of both losing exclusivity and managing winners for phase-concurrency.
- Transitioning from **W** to **O** at a barrier: note that it is also correct to transition to **C** instead of **O**. However, we predict that in the general case, a line which had conflicting accesses in the past will likely have concurrent reads in the futures. Thus transitioning to **O** reduces the amount of coherence traffic.
- Notifications for **D** → **C** and **C** → **D** transitions: notice that even for private data, the L1 caches send messages ($\mathbf{Wr_i}$ and $\mathbf{Cl_i}$) to the directory at writes and reads. This is to make sure the directory is always informed of the current state of the owner, which simplifies its logic. The performance penalty is small, because there will be at most 2 messages sent per barrier (the design of parallel algorithms typically aims to eliminate synchronization as much as possible, so barriers should be infrequent).
- Issuing writes while in **S**: it might at first seem in violation of phase-concurrency for a write to be issued while in **S**, because we arrive in **S** due to concurrent reads and therefore must be in a read phase. However, this is incorrect. The directory is not always able to know of exclusivity, and may need to send someone to **S** conservatively at an $\mathbf{Ac_i}$ message even when it would be correct to send them to **C**. For this reason, writes are permitted while in **S**, and the same logic as for **O** is used.

5 Simulations

Using a homegrown simulator, we have verified the correctness of the DCWSOLI protocol on two sample programs, summarized below. Our simulator implements DCWSOLI assuming that all transitions are atomic (in practice, there would likely be many transient states in the protocol to handle race conditions). On both programs, we ran the simulator with many different input sizes and a a varying number of processors.

- `primes`: as in the example given in Section 1, this program calculates all primes below a given number in parallel via the Sieve of Eratosthenes.

- `hashing`: this program constructs a hash set containing the elements of $[n]$ by, on each round, attempting to insert all remaining keys and retrying on the next round for those that do not succeed (due to losing a write-race). We use open addressing with linear probing.

Of course, these programs alone are not representative enough to prove that our protocol is correct in general. For example, both programs are statically scheduled in a BSP style. However, one could imagine writing a phase-concurrent work-stealing scheduler ([1]) in order to support dynamically scheduled, structured parallel computations.

Unfortunately, due to various technical difficulties, we were unable to conduct an empirical evaluation comparing our performance against state-of-the-art protocols such as MESI/MOESI.

6 Future Extensions

Sub-cacheline granularity: As mentioned previously, our current scheme interprets memory at the granularity of cache lines. It is possible to extend this scheme to sub-cacheline granularity, in a way similar to that of DeNovo, at a potentially linear factor of memory overhead in the size of each cache line [2]. Intuitively, supporting sub-cacheline granularity involves replicating each cache line and keeping track of part of each copy of the line. To be precise, if there are k memory locations in cache line L, create lines $L_1, \ldots, L_k$ and for line L_i, classify the entire line solely based on the state of memory location i. This therefore requires an extra $\log k$ bits for bookkeeping on each of $k-1$ new copies of each cache line, resulting in $k \log k$ times as much space per original cache line.

Larger hierarchies: So far, we only deal with private L1 and shared L2 caches; one potential future direction of this work is to extend the framework to accomodate larger cache hierarchies, as illustrated by Figure 2. Intuitively, this could be done by, for example, recursively viewing all higher-level caches as directories for their children. In particular, each higher-level cache would view the lower-level caches above it as a black-boxed version of the L1 cache in our current protocol. However, we would have to implement extended directory interactions; directories must be able to receive messages from lower- and higher-level directories in order to implement our coherence policy in this recursive manner.

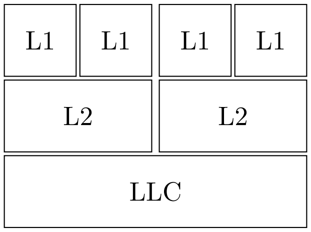

Figure 2: Higher-level cache hierarchy.

Barrier coalescing: Another further optimization is *barrier coalescing*, where two or more consecutive barriers at a memory location are grouped together and treated as two "effective" barriers. This is potentially useful in the case where there are no intervening memory accesses of the cache line *from the processor's point of view* between many consecutive barriers, and therefore repeatedly downgrading at barriers may be an unnecessary and inefficient policy. However, note that we cannot treat many consecutive barriers as a single barrier because of our transitions from S to O and then O to I on barriers; if we try to coalesce too many barriers with no intervening activity *for a single processor* into one, then we may keep data that has been invalidated by writes by other processors (i.e., to ensure correctness, we need to transition to I instead of O).

References

[1] Robert D Blumofe and Charles E Leiserson. "Scheduling multithreaded computations by work stealing". In: *Journal of the ACM (JACM)* 46.5 (1999), pp. 720–748.

[2] Byn Choi et al. "Denovo: Rethinking hardware for disciplined parallelism". In: *Proceedings of the Second USENIX Workshop on Hot Topics in Parallelism (HotPar)*. 2010.

[3] Alberto Ros and Stefanos Kaxiras. “Complexity-effective multicore coherence”. In: *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*. ACM. 2012, pp. 241–252.

[4] Julian Shun and Guy E Blelloch. “Phase-concurrent hash tables for determinism”. In: *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures*. ACM. 2014, pp. 96–107.