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 [1] [2].

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 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 [3].

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. [[TODO: actually do this – put in extensions / misc?]]

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.

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 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 [1]. 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.

3.1 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.

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.

3.2 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. **(Dp) 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. **(Cp) 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. **(Wp) 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.

3.3 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 X violate phase-concurrency.

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.

Table 1: Cache transitions

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

Transitions for the directory:

Table 2: Directory transitions

	$\mathbf{D_p}$	$\mathbf{C_p}$	$\mathbf{W_p}$	**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; $\mathrm{D_i}$
$\mathbf{Ac_i}$	XX	Send Fo to p; Receive data from p; Reject i; Send data to i; V	XX	Reject i; Send data to i; V	Retrieve data; Accept i; Send data to i; V
$\mathbf{Cl_i}$	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	-	-

4 Simulation Results

5 Discussion

6 Conclusion

7 Works Cited

References

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

[2] 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.

[3] 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.