Avoiding Undesired Behavior while Concurrently Accessing A Data Structure

Jeremy Khawaja

May 3, 2017

Abstract

The key to avoiding inefficiency and inconsistency in concurrently accessed structures is having a well-defined concurrent access algorithm. There are many approaches available to use. We devise a a method using spatial and temporal dependency graphs to manage access to arbitrary data structures.

1 Introduction

In this paper we would like to describe an algorithmic approach to avoiding undesirable behavior in systems that have multiple processes (threads) accessing a shared data structure. We should first supply a basic definition for a concurrent data structure.

Def. 1.1 (Concurrent Data Structure) - A *concurrent data structure* is a data structure that is concurrently accessed by multiple processes. The processes are allowed to read and write data as well as modify the structure itself.

There aready exist a plethora of approaches to managing access to a shared structure, of which we will make brief mention of only a few techiques e.g. *locks*, *fine-grained locking*, *combining trees*, etc.

For example, here is a basic definition of locking a data structure.

Def. 1.2 (Locking) - A *lock* is a device that enforces a temporary operation ownership of the concurrent data structure by a process e.g. a process in posession of the write lock will be the only process allowed to write data to the structure. Using a set of locks where each lock applies to only a substructure of the overall data structure is called: *fine-grain locking*.

In most programming languages a lock is called a *mutex* (mutual exclusion).

It is also important to specify what exactly we mean by *undesirable behavior*.

Def. 1.3 (Undesired Behavior) - The primary potential behaviors we will be focused on avoiding in this paper are: redundancy, data races, access blocking, and a lack of any consistency guarantee.

For example, some of the potential behavioral problems with locks can vary from things like taking too many locks or too few locks, to taking the wrong locks or taking locks in the wrong order.

Essentially we are trying to avoid problems with inefficiency and inconsistency. Namely, we would like to avoid overwriting values that have already been written. And we would also like to avoid problems that are usually seen as a lack of linearization e.g. a value being overwritten with an out-of-date value.

Of course, every access management implementation will require a degree of tradeoff. For example, are we allowed to read a value from our concurrent data structure at any time or must we always guarantee that all currently-in-progress writes have been completed? We will not focus on tradeoff choices in this paper. How a developer may choose to utilize the algorithm in this paper will depend on the developers personal choice of system requirements. But we do note that the algorithm will be rather flexible in varying the degree of consistency for concurrently accessing a structure.

One of the usual goals of concurrently accessing a single data structure is: computation efficiency i.e. speedup. Sometimes it is also used to avoid having multiple copies of the same structure and being forced to use a consensus algorithm to determine which one has the "true state" of the structure (or a sub-structure) at a given moment in time.

These goals/reasons for concurrent data structure design will not be the focus of this paper. Instead, we will focus on devising an (astoudingly rather simple) algorithm that will be able to 'monitor' and maintain the state of an arbitrary concurrent data structure for us to a degree of consistency that we would like it to have.

2 Dependency DDAGs

We begin our discussion around the crux of avoiding undesirable behavior: *Ordering* (in the literature: *linearization*).

In our definition of a CDS (1.1), we discussed reading and writing data to a structure as well as changing the structure of the CDS itself. We will label all of these processes as: *access types*. More specifically: these are procedural types. Every time we try to execute an access type we will call this an: *access procedure*.

We do not want to limit ourselves to some predefined set of operations such as "reads, writes, appends, removals, etc.". These do represent a sort of canonical basis for all possible access types, but we

want to be flexible enough to allow for any and all possible methods of accessing a CDS.

Thus, we need to find a way to ensure the linearization, or correct ordering, of access procedures.

For example, when and how will we guarantee that one access procedure completes before another one is started? The how is simple: we can sequentially order our access procedures i.e. apply a total ordering to all access procedures across our system.

But that can be accomplished by using only a single process without the necessity for concurrent processes. Thus we need to address the *when* of ordering access procedures using multiple concurrent processes.

Not all access procedures will need to be totally-ordered i.e. a complete (mathematical) linearization of our access procedures is not always a necessity. It is this *lack of necessity of a total ordering* that allows us to have the possibility of concurrency in the first place.

The degree to which our system can behave non-linearly is exactly the minimal degree to which it can be composed of concurrent processes. (Note: by "system" we are referring to a set of processes plus our CDS).

So, when is ordering a necessity? The only time two access procedures need an order is if one access procedure is *dependent* on the other. We do not specify the reason for the dependency, but if a dependency exists (at all) then it is easy to see that an ordering is necessary.

Since the necessity for an ordering is not always required, we see that a complete linearization of our system is not always a necessity. Rather, the access procedures which carry a dependency on other access procedures create a *partial ordering* for our system.

More specifically: all of our dependencies allows us to have a (disjoint)*dependency graph* for our access procedures.

All good, right?! Sort of. Sadly, this does not resolve all of our problems. We have not discussed exactly how we will order access procedures in terms of processes.

2.1 Spatial Dependencies

First, let's diverge quickly back into a discussion of our CDS. Let's try to formalize it:

Def. 2.1 (Concurrent Data Structure (2)) - A *concurrent data structure* is a graph $\mathcal{G} = \{\mathsf{N}, \mathsf{E}\}$ composed of $|\mathsf{N}|$ nodes and $|\mathsf{E}|$ edges, that is used to index data of type $\mathcal{T}$, and is allowed to be accessed by some set of processes $\mathcal{P}_i$ via some set of access types $\mathcal{A}_t$.

This helps us to remember that a data structure is a spatial object. It carries a certain topology and is invariant through time unless modified. Thus, it is completely possible that while access procedures may not carry a dependency: nodes of our CDS just might! The set of all nodular-dependencies will be called the: *set of spatial dependencies*. These dependencies may be the dependencies that an access procedure "adopts" in order to consider itself dependent on another access procedure.

For example: if access procedure A is trying to write to node $\mathrm{n}_1 \in \mathsf{N}$ and access procedure B is trying to write to node $\mathrm{n}_2 \in \mathsf{N}$, but $\mathrm{n}_2 < \mathrm{n}_1$ (where $<$ is a dependency arrow), then we would want to perform B before A.

This set of spatial dependencies only covers the dependency relationships between certain nodes in our graph. But what if we wanted sub-graphs of our CDS to be dependent or more importantly *independent* of other sub-graphs of our CDS?

Let's define this:

Def. 2.2 (Unique-Independent) - Any sub-graph $\mathcal{S} \subseteq \mathcal{G}$ that does not contain any internal spatial dependencies is called an *unique-independent* of our CDS.

Unique-Independents (UIs) can almost be seen as a *virtualization* of our CDS's spatial topology.

Unique-Independents can obviously have external dependencies i.e. be dependent on each other. Thus, the set of UIs of our CDS forms a dependency graph. Although, since UIs do not have to have other UIs as dependencies, we simply qualify this as a: dependencies list.

Def. 2.3 (Dependency List) - A *dependencies list*

for a CDS is the adjacency list of our disjointed graph of unique-independents.

Don't let the name "unique-independent" fool you. There is no requirement for an independent to be spatially-unique. In other words, a UI can share nodes from the CDS with other UIs. The only thing that makes a unique-independent "unique" is that it is in totality unique as to the nodes it applies too.

So while, a UI can be completely subsumed by another UIs spatially, it will never refer to the exact same set of nodes that another UI is referring too. And it is this notion of "uniqueness" that allows it to be an independent.

This may seem like a unique-independent that subsumes UIs with dependencies that are other subsumed UIs conflicts with our definition (2.2) of an unique-independent.

However, what we have to notice is that a unique-independent is evaluated separately from other UIs i.e. a UI that subsumes other UIs and their dependencies is only allowed to be addressed after all those other UIs have been addressed.

So, while an independent has "no spatial dependencies" it can still subsume UI-dependencies.

One of the most important points to take away from this section is the fact that: the structure of the CDS itself has zero semblance of how we design our concurrent accesses, only the structure of our UI-DDAG (disjoint directed acyclic graph) does.

Now that we have a way to virtualize a CDS using UIs (in a spatial manner), let's tie UIs back in with our previous discussion about ordering access procedures.

We can now use the UI (spatial) virtualization of our CDS to assign an access procedure to a particular UI. We can provide a brief pseudo-code description (written in the Go programming language syntax).

```
type AccessProcedure struct{
        A AccessType
        UID int // unique-independent id
}
```

2.2 Temporal Dependencies

Since our dependency list of UIs based off this virtualization represents all possible spatial dependencies in our CDS it can be used to help us apply an ordering to access procedures. However, a spatial dependency is not the only way access procedures can be ordered i.e. be dependent on each other. The other way access procedures can have an ordering is: *per UI*.

In other words, we can still order all access procedures that have been assigned to the same UI. For example, will read access procedures always take precedence over write access procedures for a particular UI? What if we wanted to be more specific (fine-grained) than that and give *certain* access procedures precedence over *certain* write access procedures?

It would be impossible to suggest that access procedures that apply to a specific UI could always be totally-ordered. Thus, we arrive at the same situation we were in before with trying to linearize our system. Thankfully, we have the answer: a dependency graph (list). Except, this time our dependencies list are access procedures assigned to the same UI.

Whereas we had an UIs dependency list that represented a virtualization of the spatial dependencies, we now have a procedures dependency list that virtualizes the temporal dependencies of our CDS (per UI). Every UI will have it's own temporal dependencies list and every node in this disjoint dependencies graph will be a specific access procedure.

In this way we can redefine our access procedure code from above to include a temporal ddag node assignment:

```
type AccessProcedure struct{
        A AccessType
        UID int // unique-independent id
        TID int // temporal ddag node id
}
```

2.3 Atomicity

We still have to come back around to combining our ordering of dependencies with the processes in our system. But, first we should consider when two ac-

cess procedures can be simply sequentially ordered. If some set of access procedures which apply to the same UI must be linearly ordered then we can consider the sequence of those procedures as being a new access procedure.

This can technically also apply to any sequence of access procedures (assigned to the same UI) that are not dependent on one another. If we want to reduce the total number of individual procedures in our system (i.e. reduce the number of nodes in our temporal dependency graph for a UI) then we can combine access procedures that are either linearly dependent or independent into a single access procedure.

The allowance for this combination is what we call: *atomicity*. Every access procedure must be *atomic*.

Def. 2.4 (Atomic Procedures) - Any access procedure or sequence of procedures that appear to the rest of the system as a single operation on a UI are called: *atomic procedures*.

All access procedures *should be* atomic procedures. But, arbitrary sequences of procedures are not necessarily atomic. We want to specifically classify sequences of access procedures that are atomic.

Def. 2.5 (Thread) - A *thread* is any sequence of access procedures that can be classified as a single atomic procedure. A thread can also be a sequence of atomic procedures (which may be sequences of access procedures in of themselves).

In terms of a dependency graph: a thread is assigned to a node and is also assigned to the node's dependencies iff the dependent node has only one dependency *and* the dependency has only one dependent. Any sequence of nodes each with at most one dependency and one dependent can be "combined" and considered: a single Thread.

Using threads makes our virtualization complexes more succinct, as any and all sequences of access procedures are either necessarily ordered or are considered single atomic procedures. It also becomes that every node in our disjoint dependency graphs are now a thread.

Prop. 2.1 (Nodes are Threads) Every node in our disjoint temporal dependency graphs is a thread (iff we maintain that all access procedures must be atomic procedures).

Cor. 2.1 (Thread DDAG) For every disjoint dependency graph (a disjoint directed acyclic graph i.e. DDAG) of access procedures there is an equivalent disjoint dependency graph of threads.

The fact of the matter is that the total number of UIs determines the minimum number of threads required to allow our CDS system to not be missing concurrency improvements (but is usually used when a maximum number of threads is a requirement).

If we choose to use a lesser amount of threads than we have UIs, then each thread will likely be assigned to multiple UIs? Can our system still offer a degree of undesired behavior avoidance in this case? The answer is yes, but that the management of behavior becomes internal to each thread.

In this case the threads will need to have internal sequential consistency. In other words, the ordering of dependent operations will only be based on the sequential ordering of access procedures within the thread. So, the thread should be well-ordered according to access procedures that apply to first: UI dependencies then UI dependents.

If the UIs that a thread is assigned to have no dependency relationship with each other (even composed dependency arrows) then the thread will not need to be ordered internally. Sometimes, ensuring this simple property makes the concurrency of the system simpler to implement and verify.

In the next section of this paper we will discuss how we will ensure atomicity for access procedures. But, for now, we simply hold this assumption as a given precondition.

2.4 Dynamic Dependencies

Now that we have methods for defining both spatial and temporal dependencies in our system, the question arises: what happens if a dependency has a lifecycle? What if a dependency needs to only hold true some of the time?

To solve these dynamic situations we can introduce

the idea of a *Virtual Dependency Graph* (VDG) which is a dependency graph that can apply to multiple UIs. The key is that is *must* have a limited lifespan. If it does not have a limited lifespan then it just becomes another spatial or temporal DDAG.

Def. 2.6 (Virtual DDAG) - A dependency disjoint directed acyclic graph that can apply to one or more UIs for our CDS, but whose lifespan must be less than the lifespan of the CDS (system). Used to define dynamic dependencies in a concurrent system.

A VDG will always have a root node that connects to each UI that the VDG involves. This root node represents the final atomic procedure that the VDG will finish before completing its lifecycle.

For example, what if a particular write on one UI needed to happen before a different write on another UI, but that this was *not* necessarily true for *all* writes for these two UIs?

The final write would be the VDG's root node, and the dependency write would be a single child node to the root node. This also means that the two nodes could be a single thread that sequentially ordered a write to the first UI then a write to the second.

So, VDGs can be assigned to a single UI or too multiple UIs.

But, *why* exactly must a VDG always have a limited lifespan (relative to the lifecycle of the system)?

Well, in our previous mentioned example, if that VDG that applied to two UIs lasted forever then it would be no different than a spatial dependency (since it would signify that all writes for the dependent UI must wait on all writes for the dependency UI). If the VDG had only applied to a single UI then it would become a permanent temporal DDAG for similar reasons. The only thing that makes a VDG dynamic is the fact that it's lifespan is always less then that of the CDS.

Similar to the discussion in the Atomicity subsection on assigning the same thread to multiple UIs, assigning a thread to multiple V-DDAGs can possibly lead to an underutilized concurrency potential. Threads that are assigned to V-DDAGs are essentially virtual themselves, as it is their lifecycle that is the lifecycle of a node in a V-DDAG.

The final dynamic property of a system we want to briefly discuss is the idea of a: dynamic UI.

If UIs are allowed a lifecycle shorter than the system then the UI dependency graph could potentially need to be constantly redrawn. The only way to outright avoid this scenario is to create another entity that would behave like the nodes in a V-DDAG (and that we specify must have a lifespan less than that of the CDS).

If a UI were to be dynamic, but have dependencies, there would have to be a way to say that it could not end its lifecycle until its dependencies ended their lifecycles.

Actually, this leads us to a very important (not yet discussed) issue: UIs must cover all nodes in the CDS i.e. every node in the CDS must be addressed by at least one UI at all times.

So, if we have dynamic UIs then we would have to check that before a UI could end its lifecycle that ever CDS node that it addresses must be addressed by some other UI. Seeing as this is just computationally inefficient, we thus note that dynamic UIs are only feasible if we create the secondary notion of: *Virtual UIs*.

This allows us to have: *Virtual UI dependency graphs*. But, then this also implies that every Virtual UI (VUI) could potentially have a V-DDAG attached to it (that must have a lifecycle less than or equal to the VUI node itself). All VUIs must have a lifecycle less than or equal to the VUI they are dependencies of. VUIs can apply to any set of nodes, but they must either have at least one real or one virtual dependent.

Every V-DDAG and VUI node is a thread that must make it's dependents aware of its existence. In fact, all nodes in any dependency graph (virtual or real) are just threads. The only difference is that real thread nodes do not control declaring themselves dependencies of one another, only virtual thread nodes can declare themselves as a (temporary) dependency of other thread nodes.

The final thing we would like to mention in this dependencies section is the fact that virtual (dynamic)

nodes can still have real nodes (spatial and temporal) as their dependencies.

2.5 Invariance

The final part of discussing disjoint directed acyclic graphs and their equivalent dependencies lists is determining if an access type is allowed or not allowed to be used on a specific UI (or even at a specific time).

Boundary Nodes on our UI dependency graph can be useful for determining what access types are allowed for a UI. For example, it might be simpler to allow CDS-structure modifying access types for boundary UI nodes only: specifically appending and pruning sub-graphs to the CDS.

But, we should first formalize what boundary nodes are and then look at an example of how this can be useful.

Def. 2.7 (Leaf Boundaries) - Any real DDAG node which has no dependencies is considered a *leaf boundary* node on that DDAG.

Every real (spatial and temporal) node can be represented by 4 numbers: spatial dependents, temporal dependents, spatial dependencies, and temporal dependencies.

```
type RealNode struct{
        SpatialDependents uint
        TemporalDependents uint
        SpatialDependencies uint
        TemporalDependencies uint
}
```

The basic difference between leaf-boundary and non-boundary real nodes can be expressed with these four variables succinctly:

- Spatial Nodes: {w, x, y, 0}
- Spatial Leaf Boundary Nodes: {0, x, y, 0}
- Temporal Nodes: {0, x, y, z}
- Temporal Leaf Boundary Nodes: {0, 0, y, z}

There is another kind of boundary node for real DDAGs:

Def. 2.8 (Root Boundaries) - Any real DDAG node which may or may not have dependencies but does not have any dependents is considered a *root boundary* node on that DDAG.

The classes for root boundaries can be expressed using our primary 4 variables succinctly as well:

- Spatial Root Boundary Nodes: {0, 0, y, z}

Notice, that we do not have Temporal Root Boundary Nodes defined. This is because of the following corollary.

Cor. 2.2 (No Temporal Root Boundaries) There are no Temporal root boundaries as every Temporal node must have at least one Spatial node dependency.

Example 2.1 (2-Regular Tree): We consider boundary nodes of a UI dependency graph for a 2-Regular Tree CDS.

In Example 2.1, we see that modifying the structure of our tree is dependent on what we want to preserve. Thus, we do not allow UIs with dependencies to have the ability to use access types that modify our CDS structure. But, we could if we wanted too. Thus, each UI could be assigned a list of *immutables*. More specifically, a list of nodes in the CDS that are *invariant* to certain access types.

Note: immutability is only a kind of invariance. A low-resolution invariance which blanket-denies all modification access types i.e. leaves the value of the data invariant under access procedures that are of a value modifier access type.

2.6 Review

Spatial Nodes are threads that last the lifespan of the CDS and are attached to a single UI (in this case a UI is a node).

Temporal Nodes are threads that last the lifespan of the CDS and are attached to a single UI (in this case temporal nodes represent all (permanent) threads beyond the initial thread attached to a UI).

Virtual Nodes are threads that have a lifespan shorter than the CDS (or Node that they are assigned to) and can be assigned to multiple UIs.

3 Algorithm

How are we going to assign unique-independent modifications to a thread? In other words, how will we ensure that a thread is only allowed to make modifications to a single unique-independent?

We will also address the atomicity of operations and sequence of operations as defined *internally* to a thread.

Since a leaf boundary node is a node without dependencies it can always iterate through its atomic operations infinitely often without waiting for other nodes.

4 Formal Verification

5 Memory

Can we utilize this new model to find a better method for Memory Management of Concurrent Data Structures (in runtimes that are not using a built-in GC)?

Garbage Collecting Virtual DDAGs becomes necessary.

6 Conclusion

References

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.