An Exploration of Concurrent Memory Models

Alexander Dean

Introduction:

Programming languages have a tendency to be syntactically fairly complex. However, semantically they can be even more so, unless, a Memory model is put in place to contain the semantic interpretation into a reasonable set of outcomes. A Memory model is a set of rules that govern how a language is to be interpreted by compilers when it comes to memory and operations on memory.

A memory model is mainly useful when dealing with languages that support concurrency. If one thread of computation wants to write to a particular memory address while another wants to read, what is read back? To know this, we would need to look at what the memory model allows and what it explicitly denies. Most of the time however data-races like the previous example are seen as an exception case where semantic behavior is explicitly undefined. The models of languages such as C or C++, who allow some unsafe behavior, just dictate that memory operations are, as long as the program is free of data-races, deterministic.

This idea of deterministic behavior is that the result of an operation is the same each time you run it with the same parameters. When talking about memory operations, we understand that in order for a read or write to be deterministic, separate threads cannot be allowed to affect the operation's ability to maintain its contract. Languages can thus provide methods for reducing the possibility of these data-races, otherwise they must resort to a weaker memory model that accounts for this.

However, memory models can express varying levels of determinism. Through the quarter it was seen that there is an explicit difference between what can be called “dynamic” determinism, which was highlighted by a project called Kendo, and “static” determinism, as seen in Deterministic Parallel Java. Although, there is one other axis of determinism, that of “hard” vs “soft” as seen with sequential consistency vs all other forms of concurrent memory models.

I will spend this paper discussing the relationship of these models of determinism and the choices language developers can make when designing their memory model. This quarter we looked at Deterministic Parallel Java among a few other languages. I will also briefly discuss other avenues of research we went on when exploring other memory model concentrations. For example, we looked at PLAID which is a type-state oriented programming language which uses a permission based model to look like a data-flow language and provide concurrency-by-default.

We also briefly touched on a few other projects that are focusing on particular research in concurrency such as MIT's Kendo project for deterministic multi-threading on Linux through a kernel module and library, and Erjang which is a Java port of the Erlang Virtual Machine onto the JVM using the Kilim library to support high-level “light” threads.

Understanding Determinism:

The argument for deterministic behavior in parallel languages is that it is easier to debug and to maintain because its easier to understand. Data races, deadlocks and other hard to debug erroneous behavior can be avoided if the programmer follows certain patterns in their code. However these patterns are often complex and are always dependent on the language's memory model.

The memory model enforces a particular method for how multiple threads will interact. Sequential consistency is one such model for our language; which says that an observer should not be able to tell the difference between a sequential and a parallel execution of a program for every intermediate state (disregarding side-channel differences). In other words, a trace of a parallel execution should coincide with a sequential execution. This contrasts from deterministic behavior in that sequential consistency can be seen as a method to provide determinism, but just because a program is deterministic does not mean it is sequentially consistent.

However, there is another definition of sequential consistency [BoehmAdve] that reveals the problematic nature of this approach in the real world. A program is considered sequentially consistent when two or more threads stay in lock-step or do not cause a memory conflict. A memory conflict is similar to what was mentioned before, it's when two operations on a particular portion of memory happen at once where at least one of them is a write.

This makes sequential consistency a difficult model to support. Disallowing two operations accessing the same portion of memory restricts developers' and reduces the effectiveness of a language. Sequential consistency isn't necessarily an advantageous restriction to put on a model to begin with as there are many optimizations compiler developers would be able to perform that would otherwise break it.

```
a = x;
b = x;    →   a=b=x;
```

Figure 1: Read-after-read optimization

One optimization that compilers can make which would invalidate sequential consistency is called read-after-read. In some languages when there are two sequential accesses to the same memory address, like in sequential assignments like in figure 1, a compiler would like to be able to only load the data into cache only once in order to not have to go to memory twice for each assignment. This makes a lot of sense but breaks sequential consistency since the value of the data at that memory address can change in between loads if there are two threads utilizing that data.

Therefore, models that simply guarantee sequential-consistency-like determinism when a program is data-race-free is more advantageous. This is what the Java Language developers have proposed and implemented. Their memory model relaxes the determinism to allow for some optimizations that could otherwise have been invalid, making it a “soft” determinism. However, with this added relaxation, some added concerns are raised. How should data-races be handled when they do appear and should the language even allow for them semantically?

Java takes a security based approach, which says that data-races can happen but should be taken care of safely (i.e. by throwing an exception or gracefully crashing), but data-races will never be introduced via optimization so that sequential consistency can be maintained for otherwise data-race-free software [Aspinall]. So it is possible to get a data race if you don't synchronize your threads properly but you will never get a random value out of the blue, it will always be some value due to some ordering of the operations. This means that two operations will never conflict to the point of over writing the same memory at a time. Deterministic Parallel Java takes a different approach, it goes so far as to say that data-races are impossible semantically as all valid DPJ programs must be free from conflicts. This can be seen as “static” determinism as it provides determinism as part of its language and doesn't do any compensation on the fly.

Why Determinism by Default?

As mentioned above, DPJ provides default support for “static” deterministic concurrency in Java. They did so by extending the Java language and adding support for region based type and effect checking to guarantee that the program returns the same values every single execution. It does not support hard determinism, as the intermediate states do not necessarily always stay in the same order from execution to execution.

By providing deterministic behavior by default, DPJ reduces the chances of data-races to zero and alters the style of programming so that it is easy to start out sequentially and then add concurrent sections later. For example, say I'm instantiating a bunch of objects using some list of values (see figure 2) where each instantiation could take a long time but doesn't ever affect the data in the list. To parallelize it, we could wrap it in a `cobegin` block, which would parallelize it as much as possible.

```
Obj x = new Obj( 1, argList );
Obj y = new Obj( 2, argList );
Obj z = new Obj( 3, argList );
```

```
cobegin{
  Obj x = new Obj( 1, argList );
  Obj y = new Obj( 2, argList );
  Obj z = new Obj( 3, argList );
}
```

Figure 2: DPJ sequential to concurrent example

However, in order to do know that `argList` isn't affected and wouldn't be affected in another thread separate to this instantiation thread, DPJ needed an effect checking system. First however, they made regional boundaries to indicate a separation between logical groups of data (i.e. logically disjoint regions). The result of this semantically means that if a region gets touched by two threads at once it can be considered a data-race. But since data-races aren't allowed in DPJ, a `cobegin` with two functions running an effect over one region is considered syntactically incorrect.

```
public interface Op {
   public <region R> void opOn( Node<R> obj ) writes R;
   ...
}
...
```

Figure 3: DPJ effect system example

The second step was defining and expanding upon the idea of regional effects. DPJ chose to define two types of effects over a region, both a subtype of a generic 'effect' class: `reads` and `writes`. A syntactic example of these regional effects can be seen in figure 3. The interface `Op` has a function `opOn` which functions on a generic region `R` of which `Node obj` is apart. The particular operation `opOn` will make on region `R` is a `writes`.

As an aside, effects can be compounded on a given function over the same or multiple regions. If a function takes two parameters, each over disjoint regions, the function could specify that it `writes` to region one and `reads` from region two. Being generic like this allows for libraries to be built that will always be deterministic even when other object types, regions or effects are substituted.

Other Semantic Models

Apart from determinism-by-default, this quarter I looked at a couple other concentrations language developers can keep in mind as well. For these we look at the PLAID language which takes a different route to concurrency. PLAID can be described as a “Typestate-oriented concurrency-by-default” programming language.

PLAID is made up of objects called typestates, which are essentially instances of classes at a specific point in time. Typestates explain the current state of the instance so that they can be operated on. Since a typestate is a temporal object, it changes over time. An example of a typestate could be a *ClosedFile* object, which could have a function '*open*' that changes it to a different typestate called *OpenFile*. This guarantees that any operation that requires a certain state of a class gets it. Any other typestate wouldn't uphold the contract with the operation as it wouldn't match the requirement.

PLAID (or at least an extension to the language called Æminium) also guarantees concurrency by default through the use of permissions. This makes it look similar to dataflow languages in that alterations to shared state are immediately pushed to other observers, and to DPJ in that the permissions act like the region and effect checking. PLAID implements these permissions as type declarations for all operations. An operation can declare required permissions on parameters and return values as 'shared', 'immutable', or 'unique'. For example a Queue's enqueue function would require a 'unique' queue object (no one else touching it) to enqueue a 'shared' object (other threads have access to it), and return a 'unique' queue with the new object in it.

The language's runtime can then use these permissions to automatically maximize concurrency in the language. This contrasts to DPJ's `cobegin` blocks as the concurrency here is implicit rather than explicit. In fact, if all your program used was unique objects for read-only operations, the entire application would be parallelized as much as it could be on your system. To add sequential blocks to the code, or require temporal separation, atomic blocks can be added as well as things called 'data groups' which act like a mutex over sets of operations.

```
if(a == null)
    synchronized(this){
        if(a==null)
            a = new Object();
    }
return a;
```

Figure 4: A common double synchronized check

I prefer this idea of implicit parallelization to anything else for a number of reasons. For one, safe code can be written quickly and logically followed. Littering code with locks and atomic operations can confuse and hinder not only beginning programmers but even experienced developers. For example, in some languages the double synchronized check (as seen in figure 4) is actually invalid as the variable can be altered in between the check and the sync call. To debug this and even protect against it would almost certainly require rewrites to portions of code and if found in the field or in critical libraries, this would be unacceptable. Also, these methods do not account for differences in hardware. Allowing the compiler or even an interpreter to account for these changes behind the scenes provides more stable results.

PLAID's use of type declarations is also a great idea. I'm a huge fan of self documenting code, and data access permissions seem to be a logical documentation requirement in non-sequential arenas. One could argue

DPJs 'regions' provide some semblance of documentation as to what objects are being synchronized on, but individual permissions in my mind provide a much more logical separation of potential and desired side effects. By this I mean, every call you make, you can rest assured that access to a particular data block is as described, whereas in DPJ all you know is that no alterations happen to one region in multiple threads at once. The difference here being that multiple entities within a region may be affected without necessarily knowing exactly what.

Concurrent Frameworks:

This quarter I was working on a personal project in Erlang. I was interested in how Erlang's runtime handled massive concurrency and Actor based message passing. Since I was focusing on DPJ so much I found a Java based clone of Erlang's BEAM compiler and interpreter called Erjang. The project used a library called Kilim for replicating (and even bettering) Erlang's "light" processes and message passing between them.

A quick explanation of the Kilim framework would be that its a post-processor for annotated Java byte code. The primary annotation Kilim extrapolates on is the `@pausable` annotation; which indicates that a given function is a light process that can be paused at any time. These processes require very little context switching as none of them require their own private heaps.

Another framework that caught my eye was MIT's Kendo project. I was unable to get my hands on the software to mess with it, but I read their paper on it. Kendo is a deterministic multi-threading module for the linux kernel and a C based library. It makes a guarantee of determinism for each computer it runs on. This provides an interesting contrast to the idea of using a virtual machine to provide cross-platform dynamic determinism.

The Kendo project argues that the if all memory accesses to shared memory were deterministic then the software would become so as well. However, Kendo defines a separate axis of determinism via "strong" and "weak" where strong means that all memory accesses to shared data is deterministic, and weak means that only the lock acquisitions are. Kendo guarantees what it calls weak determinism and states that for a program that is free of data races, "strong" and "weak" determinism provide the same guarantees.

Wrap-up:

Concurrent memory models are still a strong interest of mine and thankfully still a topic of current research. Had I a chance to repeat this course I would reduce the amount of papers to just the Boehm and Bocchino papers, and then finished it off with the Scott paper. It would have been advantageous to spend more time working with DPJ and PLAID to attempt to build something useful with it to compare their advantages to standard Java from more experience. As an aside, I read C. A. R. Hoare's "Communicating Sequential Processes" this quarter as well. So for a project ideas, perhaps adding Erlang style message passing on top of, or port Kilim to, DPJ would have been interesting tasks to consider. I would recommend this course for anyone interested in language level concurrency issues and language theory in general.

Bibliography:

- “You don't know Jack about Shared Variables or Memory Models”, Boehm et al.
- “Memory Models: A Case for Rethinking Parallel Languages and Hardware”, Adve et al.
- “Fixing the Java Memory Model”, William Pugh
- “The Java Language Specification”, Chapter 17 (mainly section 4)
- “On Validity of Program transformations in the Java Memory Model”, Jaroslav Sevcik et al.
- “Formalising Java's data race free guarantee”, Jaroslav Sevcik & David Aspinall
- “Parallel Programming Must Be Deterministic by default”, Bocchino et. al.
- “A Type and Effect System for Deterministic Parallel Java”, Bocchino et. al.
- “Safe Nondeterminism in a Deterministic-by-Default Parallel Languages”, Bocchino et. al.
- “Types Regions and Effects for Safe Programming with Object-Oriented Parallel Frameworks”, Bocchino
- “Concurrency by default: Using Permissions to Express Dataflow in Stateful Programs”,Stork et al
- “Typestate-Oriented Programming”, Aldritch et al
- “Toward a Formal Semantic Framework for Deterministic Parallel Programming,”, Scott et al
- “Implementation Tradeoffs in the Design of Flexible Transactional Memory Support”, Shriraman et al
- “Kendo: Efficient Deterministic Multithreading in Software”, Olszewski et al.
- “Kilim: Isolation-Typed Actors for Java”,Srinivasan, Mycroft

Links:

- Erjang Project: http://erjang.org
- Kilim: http://kilim.malhar.net
- Kendo: http://projects.csail.mit.edu/kendo

Hours:

- Typically spent 4-5 hours per week.
- Broken up on Tuesdays and typically Thursday and/or Friday