Clocked References in X10

Daniel Atkins
School of Engineering and
Computer Science
Victoria University of
Wellington
Wellington, New Zealand
Daniel.Atkins@ecs.vuw.ac.nz

ABSTRACT

1. INTRODUCTION

2. BACKGROUND

2.1 The X10 Programming Language

X10 is a programming language that is currently under development at IBM Research. It is a strongly typed, concurrent, imperative, and object-oriented programming language. Of those descriptors, *concurrent* is the most important, as X10 was designed with multi-core and clustered systems in mind [4, 3]. The goal of X10 is to allow programmers to easily produce code that can be distributed over multiple cores and/or machines, with good scalability [2]. This is becoming more and more relevant as manufacturers turn towards improving processors by increasing the number of cores, rather than raw speed. In addition, computers in general are becoming cheaper and available in larger numbers, and we begin to see that being able to take advantage of multi-core processors and clusters is a more and more attractive idea [7].

X10 is designed specifically to do this, and to make it easy, it contains several language constructs that allow programmers to readily, and easily, write concurrent code [3]. **Places**, which can be thought of as analogous to processes, provide a shared memory environment in which concurrent code can be executed. This memory is not shared *between* Places, which allows Places to be parallelised, and distributed across multiple machines. *Within* Places, concurrent execution is achieved by the use of **Activities**, which are analogous to Threads. (The specifics of these constructs will be dealt with in the next section).

X10 is built primarily as an extension to Java, using Polyglot to handle the translation from X10 source code to Java source code. X10 can also be compiled to C++ source code. The X10 Runtime is written primarily in X10 itself, giving it the ability to be compiled into one of several back-ends. Currently, there is a Java-based runtime environment (using the X10 Runtime as libraries for the JVM), a C++ based runtime environment, and a CUDA (Compute Unified Device Architecture—a parallel computing architecture developed by Nvidia, that can be executed on GPUs) runtime environment. This project deals solely with the Java back-end for X10, but could be easily applied to other back-ends, as the modifications were not implementation specific.

X10 source code bears a similarity to Java source code; Figure 1 shows a short source file that performs a Monte Carlo simulation to estimate a value for Pi. Since X10 is an class-based object-oriented language, it has the concept of a `Class` that defines an Object. Line 2 contains four syntactic differences to Java:

- X10 Methods are declared with `def` (public, static, and other modifiers retain their original meaning)
- Methods do not *require* a return type; if unspecified, type checking is performed at compile time to assign it a type.
- The types of parameters (and variables as a whole) are declared *after* the name; the two are separated by a `:`.
- Generics are declared inside square brackets (`[]`), not angle brackets (`<>`).

Other significant differences include the use of `val` and `var` when declaring local and field variables (lines 7-10,12,17-18); using a different array access notation (due to the inclusion of `operators`); the ability to create function types (line 8, `=>`); and the addition of syntactic sugar for looping over a range (line 11). Also note that primitive types are named with a capital letter (`Double` instead of `double`, line 17), and that types can often be omitted entirely from variable declarations—X10 is *strongly typed* however, so the types of such variables are determined and checked during compile time.

The keyword `val` is similar in meaning to Java's `final` modifier, meaning that once the value has been assigned, it cannot be changed. X10 performs checks to ensure such variables are actually assigned; if this cannot be determined, it is a compile time error. Variables marked as `var` are not final.

2.2 Clocks

```
1  public class MontyPi {
2      public static def main(args:Array[String](1)) {
3          if (args.size != 1) {
4              Console.OUT.println("Usage: MontyPi <number of points>");
5              return;
6          }
7          val N = int.parse(args(0));
8          val initializer = (i:Point) => {
9              val r = new Random();
10             var result:double=0.0D;
11             for(1..N) {
12                 val x = r.nextDouble(), y=r.nextDouble();
13                 if (x*x +y*y <= 1.0) result++;
14             }
15             result
16         };
17         val result = DistArray.make[Double](Dist.makeUnique(), initializer);
18         val pi = 4*result.reduce(Double.+,0)/(N*Place.MAX_PLACES);
19         Console.OUT.println("The value of pi is " + pi);
20     }
21 }
```

Figure 1: Estimating a value for Pi using a Monte Carlo simulation [1]

> "Many concurrent algorithms proceed in phases: in phase k, several activities work independently, but synchronize together before proceeding on to phase k + 1 . X10 supports this communication structure (and many variations on it) with a generalization of barriers called clocks. Clocks are designed so that programs which follow a simple syntactic discipline will not have either deadlocks or race conditions."
>
> —X10 Language Specification [9]

A Clock is an object that provides a programmer with a means of synchronizing concurrently executing threads—an incredibly important idea in a distributed system [6]. In X10, this synchronization is achieved through the use of a barrier-style structure somewhat based on Lamport's Logical Clock [6]; the clock object maintains a total count of the number of Activities (threads) that have registered with the clock, and a separate count of the number of activities that are currently active—i.e, not currently waiting for the clock to advance to the next phase. When an activity wishes to advance to the next phase, the clock first decrements the count of alive activities, and then if this is zero, atomically advances the phase of the clock. The calling activity is blocked by placing it in a loop until the clock advances (busy-waiting).

Clocks in X10 maintain an invariant `GlobalRef` field that refers explicitly to the original instance of the `Clock`, so that no matter where any copies may end up, they can always refer to the same `Clock` object. By forcing all updates to the internal fields of the clock to always execute at the root `Clock` object, the same state is seen by all copies of the clock at all times—an important part of ensuring proper synchronization!

2.3 Clocked References

The clocked references described in this paper are heavily derived from the design outlined in the X10 Design Document [8]. There, the intent is for only val and stack local variables to be able to be "clocked", as dealing with object references was considered "too hard" (CITATION NEEDED). The intent of this paper is show that this belief is not entirely accurate, and that it is possible to have *any* form of variable able to be clocked, be it a `val` or a `var`, or a local variable or a field. Extensions are proposed in the Design Document to allow methods, objects, fields and types to be clocked as well; but we do not consider the concept of a clocked method, object, or type in this project. We deal only with the idea of clocked references, and how interactions with them might proceed.

A clocked reference is functionally similar to a normal, unclocked reference—a location in memory in which a primitive value, or a reference to an object graph, is stored and can be accessed. However, in a clocked environment (such as a clocked finish or async), a clocked reference becomes quite different to an unclocked reference, in terms of how and when it can be updated and accessed.

Simply put, during a single clock phase, the value of a clocked reference (be it primitive or and entire object graph) remains **fixed**. If the reference is written to, or updated in some way, the change does not become visible until the **end** of the clock phase.

3. DESIGN OF CLOCKED REFERENCES

In this section, we outline the design for Clocked Variables. We begin by discussing what it means to be "clocked", and why we *want* to be able to clock variables in the first place. We then consider the case of clocked primitives, and how they should function. Taking the idea a bit further, we then explore the concept of a clocked reference (or non-primitive

type), and consider the complications that arise from clocking objects. Following that, we explore two different means of extending the language to allow for clocked primitives and references, and the issues that arise.

4. CLOCKED VARIABLES

The design for clocked variables is heavily derived from the design outlined in the X10 Design Document [8]. There, the intent is for only val and stack local variables to be able to be "clocked". For this project, the intent is to have *any* form of variable able to be clocked, be it a `val` or a `var`, or a local variable or a field. Extensions are proposed in the Design Document to allow methods, objects, fields and types to be clocked as well; but we do not consider the concept of a clocked method, object, or type in this project. We deal only with the idea of clocked variables, and how interactions with them might proceed.

A clocked variable is functionally similar to a normal, unclocked variable—a location in memory in which a primitive value, or a reference to an object graph, is stored and can be accessed. However, in a clocked environment (such as a clocked finish or async), a clocked variable becomes quite different to an unclocked variable, in terms of how and when it can be updated and accessed.

Simply put, during a single clock phase, the value of a clocked variable remains **fixed**. If the variable is written to, or updated in some way, the change does not become visible until the **end** of the clock phase. Figure 2 shows this interaction, and Figure 3 shows code that utilises this.

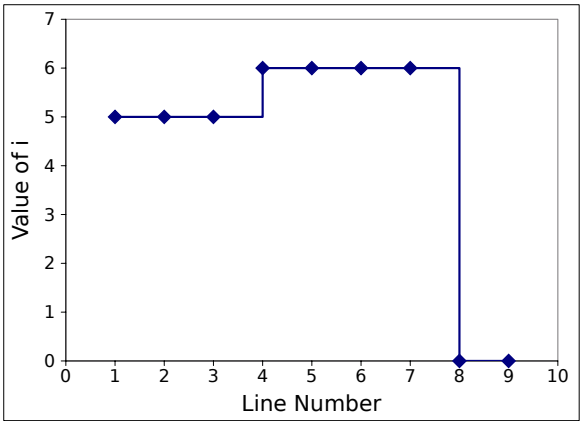

Figure 2: The Value of a Clocked Variable

```
1  clocked var i:Int = 5;
2  i = 6;
3  Console.OUT.println(i); //Prints 5
4  Clock.advanceAll();
5  Console.OUT.println(i); //Prints 6
6  i = 0;
7  Console.OUT.println(i); //Prints 6
8  Clock.advanceAll();
9  Console.OUT.println(i); //Prints 0
```

Figure 3: Example of Clocked Code

We require that clocked variables only be written to **once** during any given clock phase—writing to a clocked variable more than once in a given clock phase is a runtime exception. At first glance, this appears to be an odd design decision; primarily this was done to meet the proposal for Clocked Variables given in the X10 Design Document [8]. However, that document specifies this behaviour for variables marked with the keyword `val`. In this case, it **is** an error to write to the variable more than once per phase, as the variables are *final*. Under clocking, the original design allows such variables to be re-initialised once per phase. We did not adhere strictly to this design, as we allow the clocking of `var` variables, and allowing more than one write per phase was considered. However, this limit was deemed necessary to deal with some of the issues raised by clocking reference types, as discussed later.

Clocked variables may be *read* any number of times during a given clock phase, but we require that this value remains constant for the duration of the phase. If a clocked variable is written to, or updated in any way, the new value must take effect *between* the clock phases. The idea, then, is that clocked variables provide the same functionality as manually maintaining two separate states in a concurrent algorithm, but without all of the extra book-keeping that this currently requires.

4.1 Clocked Primitive Types

The design of clocked variables started with primitives, as they are conceptually easier to deal with than objects and references. Our design for clocked primitives is based on the outline for clocked vals given in the X10 Design Document, but has been extended to cover non-local vars and fields as required. We also depart from the Design Document in that *outside* of a clocked environment (ie: with a block encapsulated by `clocked(Clock)`, `clocked finish`, or `clocked async`) clocked primitives can still be used—they simply revert to behaving like an unclocked variable of the appropriate type.

The basic design is that of wrapper classes—instead of dealing with the primitive variable directly, all interactions are abstracted away by "Clocked Primitive" objects that sit between the primitive variable and the rest of the program. Thirteen of these wrapper classes are needed: one for each of the primitive types available in X10 (Table 1).

The design of the wrapper classes is reasonably simple. Each class contains two fields of the appropriate primitive type: one to hold the *current* value of the clocked variable, and one to hold the *next* value. Only two operations are supported on clocked primitives:

read returns the current value of the clocked variable. Can be performed any number of times.

write updates the next value of the clocked variable. Can only be performed *once* per clock phase.

As X10 supports user-defined operators (and operator overloading) these two operations are implemented as (for a ClockedVar x) `x()` for read, and `x() = newVal` for write.

Table 1: Primitives in X10

Numeric (Signed)	Numeric (Unsigned)	Non-Numeric
Byte	UByte	Boolean
Short	UShort	Char
Int	UInt	
Long	ULong	
Float		
Double		

The motivation behind this design is to keep code looking clean and as close to "unclocked" code as possible. Ideally, a programmer would be able to ignore these operators and use the variable as if it were just a normal, unclocked primitive, and the compiler would insert the appropriate operations to add clocked support. This was not implemented (see Chapter 4) for *primitives*, but was for references, as discussed below.

4.2 Clocked References

Like Clocked Primitives, Clocked References are expected to maintain a constant value during a clock phase, and then to update that value at the end of each clock phase. Unlike Clocked Primitives, this is not a simple matter of just executing `current = next`. Clocked References are not really dealt with in the design document, save for defining the concept of a "clocked field" that might exist inside such an object. Thus, the design for Clocked References are inspired by, but not particularly based on, the our work done for Clocked Primitives.

The first hurdle posed by Clocked References is that they can, quite literally, be **any** reference type. Where Clocked Primitives only had 13 different types to construct wrapper objects for, Clocked References have an infinite number. Thankfully, X10 supports the use of Generics, so this design issue is quickly overcome. So we then describe a class: `ClockedRef[T extends Object] extends ClockedVar`. As each subclass of ClockedVar must provide its own current and next fields, as well as the implementation of the `next()` method, we find that this new class fits directly into the class structure described in chapter 4. But how do we successfully update a Clocked Reference?

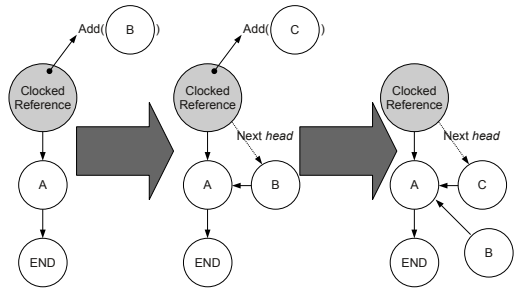

Figure 4: A clocked LinkedList behaves oddly under a call to add(node)

The answer is not a simple one. Figure 4.2 shows the behaviour of a clocked LinkedList during a call to add. Notice that we have a clocked reference to the root of the list (the head node). A call to add a node to the list, executed on the head node, adds a node to the list—but this alteration is not visible yet, as any *reads* of the graph are done from the current state, and the alteration was performed on the next state. Another add call is made; how do we resolve this? We need to ensure that we have access to a up-to-date version of the list, but the first change as not yet been commited. We cannot set the *next* field of the Node correctly, and the list enters an inconsistent state. Ideally, we would require all such alterations to be performed on a version of the graph that is kept up-to-date. It becomes clear that we cannot simply just maintain two states for the object being referenced by the clocked reference; we need to do this for the *entire* object graph it is connected to. But how, then, do we propagate changes to the current state when the clock advances?

There are many ways in which a Clocked Reference could update its value—the most simplistic of which is a deep-clone of the entire object graph. In fact, X10 readily provides a method to perform exactly that operation; one which even takes cycles into account. Thus the initial design of a Clocked Reference begins to coalesce. Other methods that were considered (and will one-day be revisited) include recursive clocking: in which each object referenced by the graph is individually clocked on the same clock as the root of the graph. To avoid issues caused by calling multiple updates on the graph in one clock phase, the write operator was limited to only one write per phase, as specified in the Design Document [8]. With the naïve deep-cloning method of updating Clocked References, however, it could be argued that this was unecessary, as the two states are completely separate object graphs. In the interest of exploring more interesting update mechanisms, however, we felt it was necessary to enforce this limit.

With the issue of updating a clocked reference dealt with, we turn to the final parts of the design: operations on a clocked reference. Immediately we can see that the operations used with clocked primitives are not going to suffice. `Read` still functions well enough, as it now just returns a reference to the current value of the clocked reference. `Write` proves a little more troublesome. We don't want to support an operation that replaces the next value wholesale—instead, we want to be able to give out a reference to the next value to allow programs to alter it in less destructive ways (such as updating a field, or calling a method, etc). After some consideration, it was decided that Clocked References would not support *any* operations, as there was no easy way to pass only the required changes to the graph as a parameter. Instead, direct access to the current and next values of the clock reference would be performed via method calls (`readableObject()` and `writableObject()` respectively).

4.3 Back-end Design

Having described the design of clocked primitives and references, we now describe the design of the actual clocking mechanism itself, and how it fits into the overall X10 architecture. This is not touched on at all in the X10 Design Document, and as such, is entirely our own design.

There are two main alternatives for the back-end of this system. The first puts the onus on the Clock to keep track of Clocked Variables and perform the updates. The second shifts this responsibility to the spawning Activity itself.

4.3.1 The GlobalRef Method

The first implementation of Clocked Variables uses the GlobalRef structure (See Chapter 2, Section 2.2.1) to ensure that all operations performed on the object are executed in the correct place—this is especially vital, otherwise the state of the object becomes inconsistent.

A list of GlobalRef objects is maintained by the Clock object. When a clocked variable is registered on that clock, its GlobalRef object is copied to the "root" of the clock (the `Place` it uses for its fields) and added to the list. Then, when the Clock advances from one phase to the next, it calls the next() method on all of the members of the list. Figure 5 illustrates this.

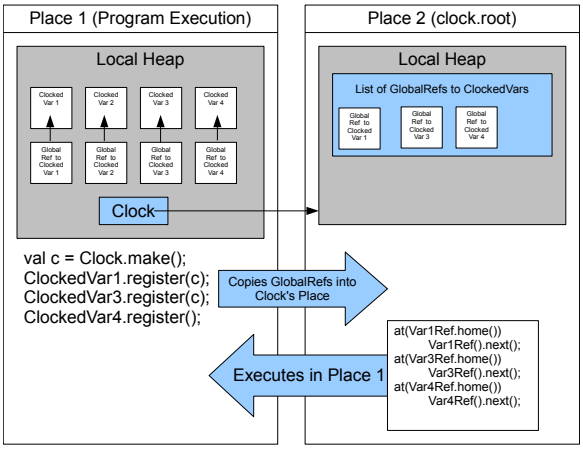

Figure 5: An activity in Place 1 using 4 ClockedVars under the GlobalRef method

4.3.2 The Map Method

For this design, the onus of keeping track of clocked variables falls on the activity in which they were declared. Each clocked variable is assigned an integer id upon construction, and a mapping from this id to the clocked variable is stored in a HashMap within the activity. The activity then registers the clocked variable on the same clock (if any) that the activity is registered on. This is accomplished simply by passing the integer id to the clock, which stores it in a ArrayList in the "root" Clock. Since the id is invariant, the fact that this value crosses places during this action does not raise any concerns.

At the end of the clock advancement step, the clock passes its internal list to each activity that it is associated with. Then, each activity scans the list for any ids that exist in the Map—if it finds any, the activity issues a call to next() on that clocked variable. Figure 6 illustrates this.

This design has the advantage of storing very little state within the Clock object itself: a list of primitive integers,

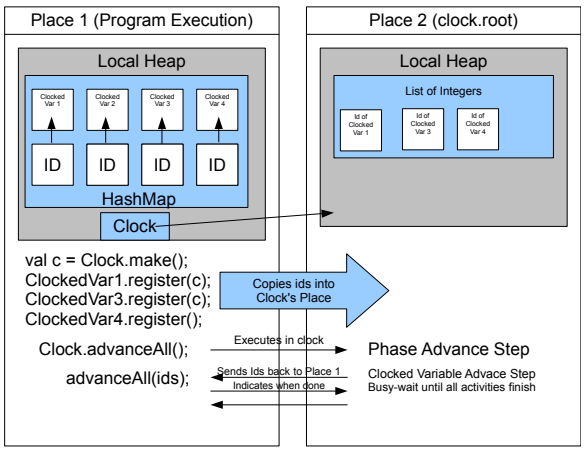

Figure 6: An activity in Place 1 using 4 ClockedVars under the Maps method

rather than a larger struct. Since the burden of updating the clocked variables falls on the activity itself, there is also no need to switch places in order to update them—and this way, clocked variables in different places (and thus Activities) are updated concurrently, rather than consecutively.

5. EXPERIMENTAL SETUP

5.1 Four Little Benchmarks

The performance of clocked references was measured through the use of four benchmarks, each of which tested a different form of reference type. Each benchmark was implemented in two different ways; using the clocked references described in Section 3, and not using clocked references. Care was taken to ensure that all versions of the benchmark programs operated correctly, and that the use of clocked/unclocked references were the *only* differences between the two version of each benchmark. An explanation of these benchmarks follows.

5.1.1 Conway's Game of Life

Conway's Game of Life is a fairly simple cellular automata, originally described by the mathematician John Conway [5]. The automata consists of a two-dimensional grid-based world, with each cell of the grid having two states (dead or alive). Cells live or die according to the following rules (neighbours are the 8 cells surrounding each grid location):

- Any live cell that has fewer than two live neighbours dies.
- Any live cell that has two or three neighbours lives.
- Any live cell that more than three neighbours dies.
- Any dead cell that has *exactly* three neighbours becomes alive.

The rules above are applied *simultaneously* to each cell in the grid. This is done in X10 by using the `async` structure to parallelise the application of the rules to each cell. Thus each

cell is given its own thread, the state of each cell is calculated concurrently with the state of each other cell. Naturally, unless this program is being executed on a machine with the same number of processors as cells in the grid, these calculations are not actually concurrent, as only a limited number of threads can be executed at once.

This was implemented in X10 using an array of integers to represent the grid. The clocked version used a single array of clocked integers, and the unclocked version used two arrays of normal integers (one to represent the current state, and one to represent the next state). The update mechanism for the unclocked version is essentially the same as for the clocked version (but coded manually): a loop copies the value from the next board state to the current board state. This may seem odd at first glance—surely it would be more efficient to simply re-assign the current board reference to point to next board and then simply re-initialise the next board reference as a new array. This is certainly the case for small boards, but for larger boards (400x400 and above) the loop method is more efficient. (NEED TO BENCHMARK THIS SO HAVE NUMBERS TO BACK THIS UP. CURRENTLY ONLY ANECDOTAL EVIDENCE. USING THIS METHOD, AN 850x850 BOARD TAKES 100s TO RUN TO COMPLETION; USING THE "more efficient" APPROACH, IT TAKES UPWARDS OF 30 MINUTES, PROBABLY DUE TO GARBAGE COLLECTION)

5.1.2 N-Body Simulation

An N-Body simulation is a physical simulation of many interacting *particles*, usually driven by the effects of gravity. N-Body problems are computationally intensive, as calculating the next state of a particle involves determining its interactions with each and every other particle in the system. Generally, these interactions take the form of forces exerted between the particles—usually gravitational (in the case of uncharged particles or large bodies, like planets) or electrostatic (in the case of charged particles) or both. Figure 7 shows a diagram of such a system.

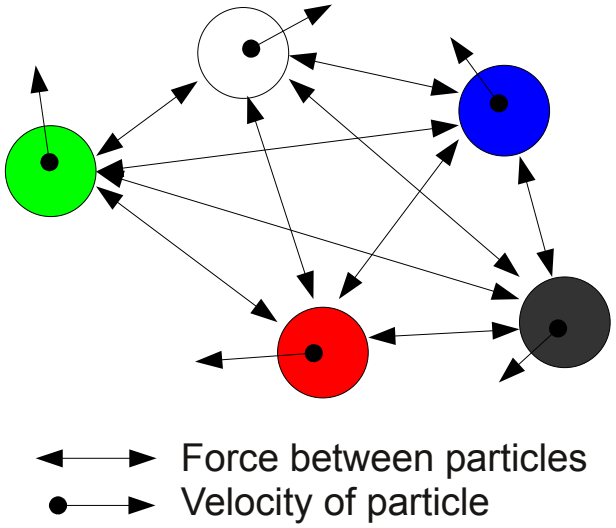

Figure 7: Interactions in a 5 particle system

This benchmark was implemented as a N-Body system of uncharged particles (i.e. the only interaction between the particles was gravitational). The particles were represented as a simple object with several primitive fields and an update method. In the clocked version of this benchmark, these particles were the objects that were clocked. The update method executed on the *next* state of the object, and wrote directly to the fields. In the unclocked version, two additional fields had to be added to hold the information required to update the particle, and a new method, `next()` was added to the Particle class so this update could be performed. Similar to Conway's Game of Life, this was done after ensuring that all of the next states had been calculated.

5.1.3 Sparse Matrix Convolution

A Sparse Matrix is a matrix in which the majority of the values are zero, thus allowing more compact storage by storing only the *non-zero* values. This was implemented using a linked-list style structure, in which each row of the matrix is represented by a single list. Rows are then linked by their first node, as shown in Figure (MAKE A FIGURE). This allows access to any cell within the matrix by following the links from the root node.

In this benchmark, this implementation of a sparse matrix was used to represent an image which then had three filters applied to it via convolution (DO I NEED TO EXPLAIN WHAT CONVOLUTION IS?). Much like Conway's Game of Life, the "next" (in this case "filtered") state of a given pixel in the image is calculated from the value of the pixel and its immediate neighbours, and this must be done "simultaneously" for each pixel. The difference here is one of representation; whereas Conway's Game of Life was an array of primitive integers, the images used in this benchmark are represented by a complex linked object structure. In the clocked version of this benchmark, the entire object graph is clocked via the reference to the root node of the matrix. In the unlocked version, it is necessary to update the current image state by replacing the reference with a reference to the next image state, and then re-initialising the next state to be an empty matrix.

Important Consideration: for this benchmark, the ability to write multiple times during a single clock phase was **enabled**. This is due to the nature of the calcuation being peformed—it can be reliably demonstrated that performing convolution on a matrix only updates each entry **once**, and any structural change to the object was performed atomically and was immediately visible to all threads. There will be further discussion on this topic in Sections 5 and 6.

5.1.4 Linked List Microbenchmarks

For this benchmark, the performance of a clocked data structure was tested against that of an unclocked data structure. The benchmark itself is quite simple; for linked lists of various sizes, the add and remove methods were executed a number of times. This benchmark mostly tests the overhead introduced by forcing the clocked list to be updated after *every* method call. Both were implemented in the exact same fashion; and both required the execution of `Clock.advanceAll()` after every method called on the list.

6. RESULTS

Each benchmark was executed 100 times for each input value. The results shown here are the average values of those executions.

6.1 Conway's Game of Life

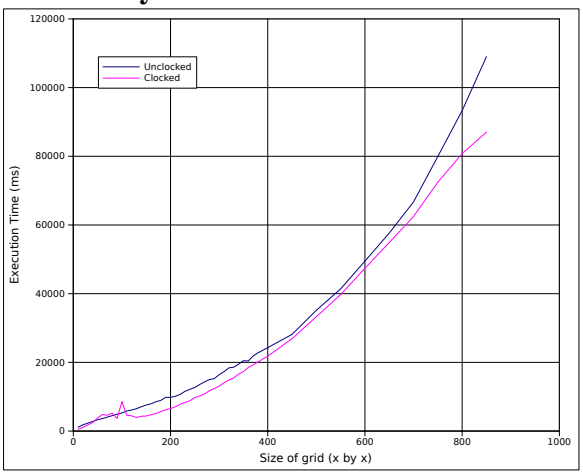

Figure 8: Conway's Game of Life: Clocked vs Unlocked execution times

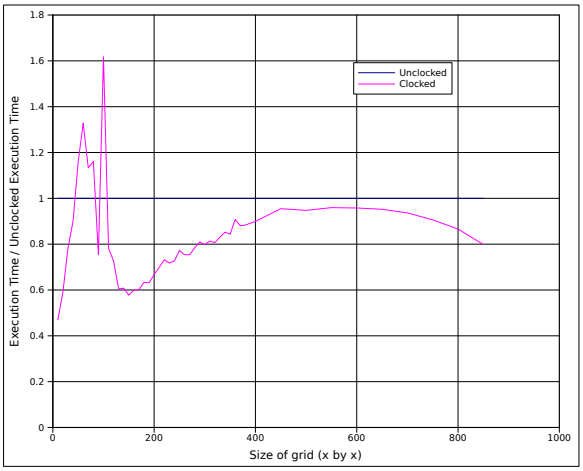

Figure 9: Conway's Game of Life: Ratio of $\frac{clocked}{unclocked}$

Figure 8 gives the results for the execution of the Conway's Game of Life benchmark for boards of various sizes. There is no significant difference between the clocked version and the unclocked version.

6.2 N-Body Simulation

6.3 Sparse Matrix Convolution

6.4 Linked List Microbenchmarks

7. DISCUSSION

7.1 Enabling Multiple Writes per Phase

It is obvious from the results presented in Section 5 that the performance of certain applications (i.e. Linked Lists) is heavily impacted by the inability to write to a clocked reference multiple times per phase. Why is this a restriction? If it can be shown that a given write is "safe", then what good

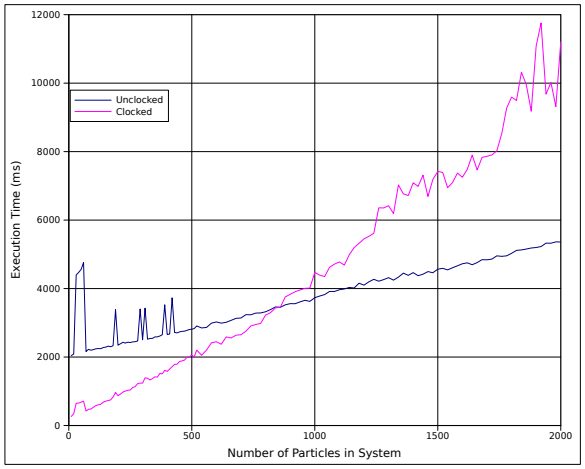

Figure 10: N-Body Simulation: Clocked vs Unlocked execution times

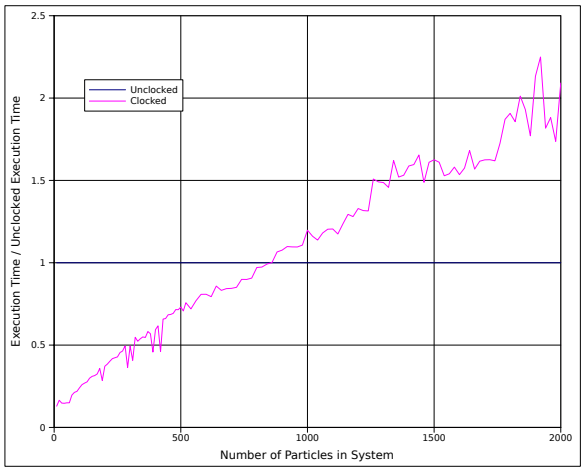

Figure 11: N-Body Simulation: Ratio of $\frac{clocked}{unclocked}$

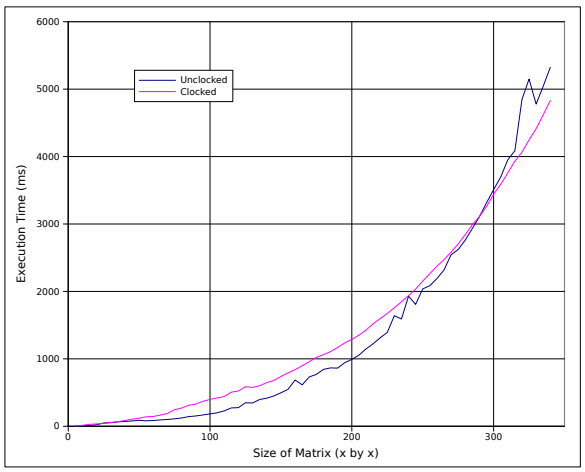

Figure 12: Sparse Matrix: Clocked vs Unlocked execution times

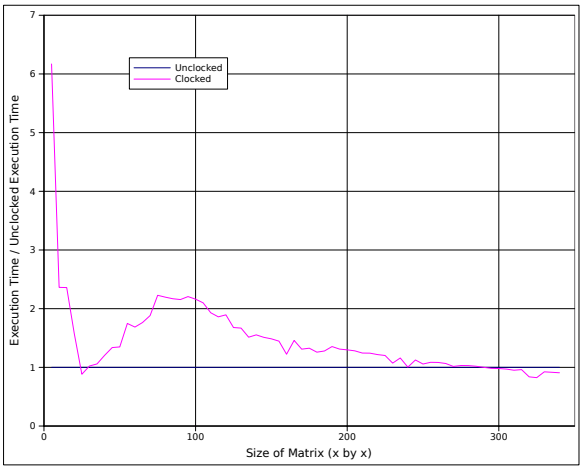

Figure 13: Sparse Matrix: Ratio of $\frac{clocked}{unclocked}$

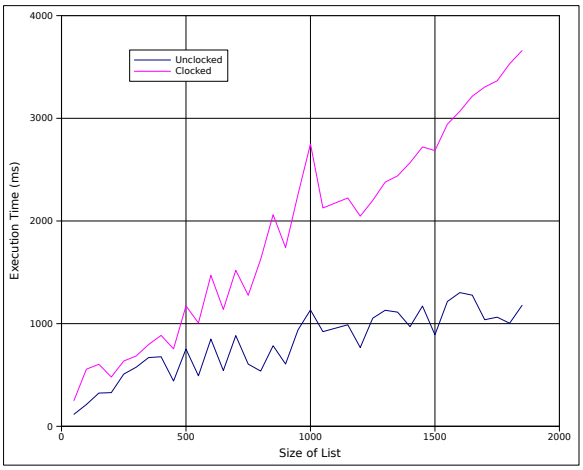

Figure 14: Linked List: Clocked vs Unlocked execution times

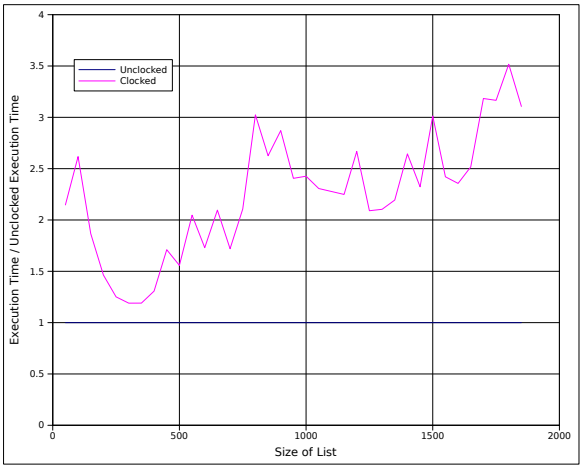

Figure 15: Linked List: Ratio of $\frac{clocked}{unclocked}$

reason is there for not allowing it? But before we can discuss that, we should look at what it means to be "safe".

A "safe" write is any write to any part of a clocked object graph that (1) does not change the **struture** of the graph, and (2) does not involve a value that has been written to already during this phase. For example, it would be *unsafe* to add or remove a node from a linked list of integers, but it would be *safe* to alter the integer value store within a node—provided that value has not already been changed this phase! From this, we can immediately see that the Sparse Matrix benchmark is **not** safe, as some operations change the structure of the object graph (setting the value of a previously zero entry to a non-zero value). This was taken into account in the benchmark, and all updates to the object graph are performed atomically and are immediately visible to all threads—but this will not always be the case!

Once this difference in safety has been established, we can amend the requirement of a clocked reference to only allowing one *unsafe* write per clock phase. The issue then becomes determining what is a safe update, and what is not. Ideally, this would be done automatically by the compiler with no extra work required on the part of the programmer—but this would be require a means of determining every possible interaction that could occur with an object. Certainly possible for very simple objects, but the difficult escalates quite rapidly.

One suggestion is the use of ownership types to isolate interactions into three different domains: "Globally Unsafe", "Locally Unsafe", and "Always Safe". This allows the programmers to specify which interactions fall into which domain by using annotations.

Executing a Globally Unsafe interaction would necessitate advancing the clock phase before **any** other unsafe interaction (local or otherwise) can occur. This could be something like sorting a list, applying a filter to an image, or changing the structure of a database.

A locally unsafe interaction only affects a certain sub-graph in the object graph, and thus multiple *non-intersecting* locally unsafe interactions can occur during a given clock phase. Examples might include adjusting the value in a single node of a linked list, altering the value of a single pixel in an image, or adding a new entry to a table in a database.

Always safe interactions, as the name implies, are always safe. This domain should be reserved for interactions that only ever read the object graph—if it peforms an update of any kind, it's probably unsafe.

7.2 Two Bad Approaches

Approach 1 (Figure 16) is unsafe. Consider the example shown in the Figure: adding an item into a linked list cannot be safely done more than once per clock phase, as the second add operation simply **cannot** know about the previous addition, as it uses the readable versions of the objects to determine the current state of the list—these versions of the objects do not have any links to the new node! Thus the add operation replaces the next pointer of the *old* last node with a pointer to the second new node, thus erasing

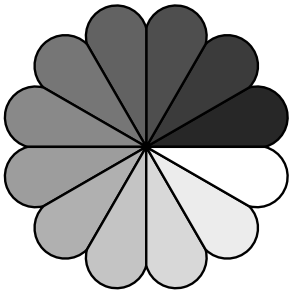

Figure 16: Approach 1: Clocking objects individually

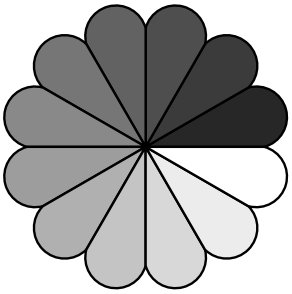

Figure 17: Approach 2: Single point-of-entry, clocking entire object graph

the *first* new node from the list. Multiple writes are unsafe under this approach.

Approach 2 (Figure 17) attempts to solve this problem by splitting the object graph into two disparate graphs: a writable graph and a readable graph. This is the approach to Clocked References used earlier in this paper. We can see that doing this solves the issue of data loss, as each write operation is performed on the writable object graph, which is always the most up-to-date version of the object. The add operation is safe here, as the entire operation uses the writable object graph. But what about other operations? If we were maintaining a *sorted* list, adding a new node may not be safe, especially if the location that a node must be inserted is determined prior to calling any methods on the writable graph—instead, the location would be determined by the readable object graph, and so multiple additions–while no data would be *lost*–may result in the list no longer being sorted. We also see that this approach is not thread-safe, as multiple threads attempting to add nodes to the list would be prone to the usual issues of concurrent lists. To eliminate this, we must then state that every thread that wishes to use the writable object graph must obtain a lock on the root node in order to proceed. Thus every write is atomic and uninterruptable—but we have sacrificed parallelisation. This becomes a large issue with problems like the Game of Life, or image convolution: if each thread is only updating one node, and no node is being updated by more than one thread, then why *shouldn't* the threads be able to do this concurrently!

7.3 Two "Better" approaches

Approach 3 (Figure 18) builds upon Approach 1 and attempts to make it safer. We first require that each thread lock the objects it needs to update. While these objects are locked, the thread uses the **writable** version of the object for **all** operations. This ensures that no data is lost, but brings new difficulties in ascertaining which objects a thread needs to lock in order to perform the operation successfully. It also raises concurrency issues: deadlock needs to be avoided, as it could be caused by two threads needing the same two objects, and locking them in different orders. Thus the threads attempt to lock the object that has been locked by the other thread, and deadlock ensues. Thankfully, there is a wealth of research on how to deal with this.

Approach 4 (Figure 19) attempts to solve this problem by providing a single point of entry. When a thread needs to lock objects, it first locks the root object; thus any thread that needs to lock objects within the graph can do so without interfering with any other threads. This doesn't solve the issue of multiple threads needing to write to the same object! In this case, such a thread must wait until the next clock phase to do so.

Thus, we have suggested a way to allow multiple *non-interfering* writes in a single clock phase.

8. CONCLUSION

9. REFERENCES

[1] Compute pi, monte carlo method using places & threads. `http://x10-lang.org/documentation/`

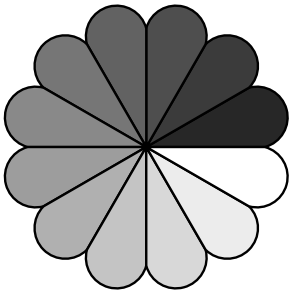

Figure 18: Approach 3: Clocking objects individually, with individual locks, and a flag on each object

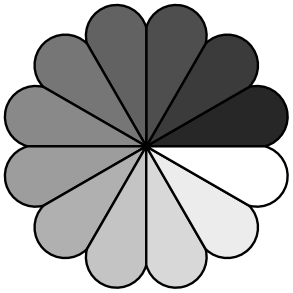

Figure 19: Approach 4: Single point-of-entry, with individual locks, and a flag on each object

code-examples/small-examples/225-monte-pi.html, April 2011.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[3] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: Programming for hierarchical parallelism and non-uniform data access (extended).

[4] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: an experimental language for high productivity programming of scalable systems (extended abstract). In *Workshop on Productivity and Performance in High-End Computing*, P-PHEC, February 2005.

[5] M. Gardner. The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, 223:120–123, Oct. 1970.

[6] L. Lamport. Ti clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[7] P. Murthy. Parallel computing with x10. In *Proceedings of the 1st international workshop on Multicore software engineering*, IWMSE '08, pages 5–6, New York, NY, USA, 2008. ACM.

[8] V. Saraswat. X10 design notes. `https://x10.svn.sf.net/svnroot/x10/documentation/trunk/x10.man/v2.2/design-notes/design-v08.txt`, April 2011.

[9] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 language specification. `http://dist.codehaus.org/x10/documentation/languagespec/x10-221.pdf`, September 2011.