The Design and Implementation of Clocked Variables in X10

Daniel Atkins[1] **Alex Potanin**[1] **Lindsay Groves**[1]

[1] School of Engineering and Computer Science,
Victoria University of Wellington,
Wellington, New Zealand
Email: {Daniel.Akins,alex,lindsay}@ecs.vuw.ac.nz

Abstract

This paper investigates the addition of Clocked Variables to the X10 Programming Language. Clocked Variables work well for primitives and objects with only primitive fields, but incur substantial performance penalties for more complex objects. We discuss ways to deal with these issues.

Keywords: X10, Concurrency, Clocks

1 Introduction

Distribution and parallelization are an important part of computing today; with the focus of processor manufacturers turning away from higher speeds, and towards larger numbers of cores, proper utilization of such resources is becoming more and more important (Saraswat et al. 2007, Murthy 2008). Unfortunately, many current programming languages don't provide the necessary support for easily writing thread-safe programs. To address this issue, IBM have been developing a new programming language called X10 (Saraswat et al. 2007, 2011, Charles et al. 2005). X10 is a strongly typed, concurrent, imperative, and object-oriented programming language, making it quite similar to popular existing languages such as Java and C++. X10 was designed with multi-core and clustered systems in mind. 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 (Murthy 2008). This means that concurrent programs become much easier to write, as the language has many built-in constructs to aid programmers in achieving their goals (Ebcioglu et al. 2005).

Many concurrent algorithms maintain two states; a current state and a next state. Operations are performed on the current state, and the results cause the next state to be updated. When the current state has been fully processed, the next state becomes the current state, and the algorithm continues. This can lead to code bloat, as maintaining these two states requires some extra book-keeping, and there is a risk of introducing bugs into a program by accidentally using the wrong state when performing an operation.

Clocked Variables allow variables to have different states depending on how they are used. This allows the prevention of race-conditions as each thread is guaranteed a consistent view of the clocked variables. This is achieved by requiring that updates to a clocked variable do not become visible until every thread has indicated that they are ready to progress to the next state. In the case of clocked Primitives, this reduces the requirement of having two explicit states to simply maintaining a single object in memory that automatically performs updates and state transitions at the appropriate times. More interesting is the case of Clocked References, which require a more complex clocking mechanism—it is these that we investigate in this paper. We implement Clocked Variables in X10, guided by the X10 Design Document (Saraswat 2011). While this paper is written with X10 in mind, the concepts it presents are applicable to any programming model that uses phased execution controlled by barriers.

The rest of this paper is structured as follows: Section 2 introduces the X10 Language, and details some of the language-specific constructs that are required to understand this paper. Section 3 discusses Clocked Variables, with a focus on how Clocked References are handled. Section 4 describes the experimental setup used to benchmark the performance of clocked variables, and Section 5 evaluates the performance of Clocked Variables. Section 6 gives further discussion of the results, and the paper is then is concluded by Section 7.

The main contributions of this paper are:

- Extension of the X10 programming language to include Clocked Variables,
- Case studies that use the new Clocked Variables,
- Performance evaluation of Clocked Variables in X10.

2 The X10 Programming Language

X10 contains several language constructs that allow programmers to readily, and easily, write concurrent code (Ebcioglu et al. n.d.). **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.

A Clock is an object that provides a programmer with a means of synchronizing concurrently executing threads—an important idea in a distributed system (Lamport 1978). In X10, this synchronization is achieved through the use of a barrier-style structure based on Lamport's Logical Clock (Lamport 1978); 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 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!

X10 is built primarily as an extension to Java, using Polyglot (Nystrom et al. 2003) 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.

3 Clocked Variables

The clocked variables described in this paper are based on the design outlined in the X10 Design Document (Saraswat 2011). There, the intent is for only val (final variables that can be altered once per clock phase) and stack local variables to be able to be clocked, as dealing with object references was considered too hard. The intent of this paper is to explore that claim and to investigate if it is possible to have *any* form of variable able to be clocked. Extensions are proposed in the Design Document to allow methods, objects, fields and types to be clocked as well; but we do not consider these in this paper. We deal only with the idea of clocked primitives and references, 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, is stored and can be accessed. However, in a clocked environment, a clocked variable becomes quite different to an unclocked variable, in terms of how and when it can be updated and accessed.

3.1 Design of Clocked Variables

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 1 gives an example of code that demonstrates this.

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. 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

```
clocked var i:Int = 5;
i = 6;
Console.OUT.println(i); //Prints 5
Clock.advanceAll();
Console.OUT.println(i); //Prints 6
i = 0;
Console.OUT.println(i); //Prints 6
Clock.advanceAll();
Console.OUT.println(i); //Prints 0
```

Figure 1: Example of Clocked Code

a concurrent algorithm, but without all of the extra book-keeping.

Only allowing one write per phase may appear to be an odd design decision; primarily this was done to meet the proposal for Clocked Variables given in the X10 Design Document (Saraswat 2011). However, that document specifies this behaviour for variables marked with the keyword `val`—that is, variables that are final, but when clocked, can be updated once per phase. 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 non-val 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.

3.2 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 clocked primitives can still be used *outside* of a clocked environment (ie: with a block encapsulated by `clocked(Clock)`, `clocked finish`, or `clocked async`)—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. One of these wrapper classes is needed for each of the thirteen primitive types available in X10.

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.

3.3 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 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 is entirely our own, and is based on the design of Clocked Primitives.

As a Clocked Reference must encapsulate a reference type (not a primitive), using generics to describe a general "Clocked Reference" was deemed the best approach. The bigger issue is that an object may contain references to other objects. Clearly updating such a complex structure would not be a simple task. So, how do we successfully update a Clocked Reference?

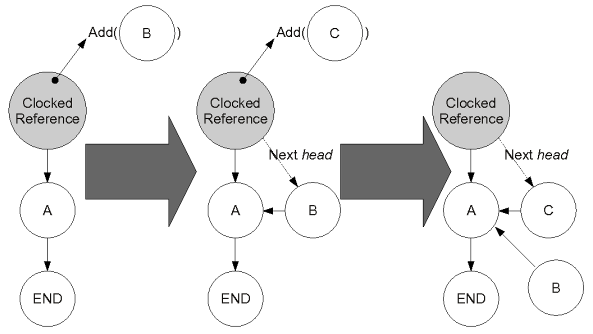


Figure 2: A clocked LinkedList behaves oddly under a call to add(node)

The answer is not simple. Figure 2 shows the behaviour of a clocked LinkedList during a call to add. Notice that we have a clocked reference to the head of the list. 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 is 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 has 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 simplest 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. This is the method used in the design of Clocked References within this paper, as time constraints meant other avenues could not be fully explored. 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 (Saraswat 2011). With the naïve deep-cloning method of updating Clocked References, however, it could be argued that this was unnecessary, 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.

Having chosen a solution to the problem of updating a clocked reference, we turn to the operations that can be performed 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).

3.4 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.

3.4.1 The GlobalRef Method

The first implementation of Clocked Variables uses the GlobalRef structure (an X10 type that can be used to access objects across Place boundaries) 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.

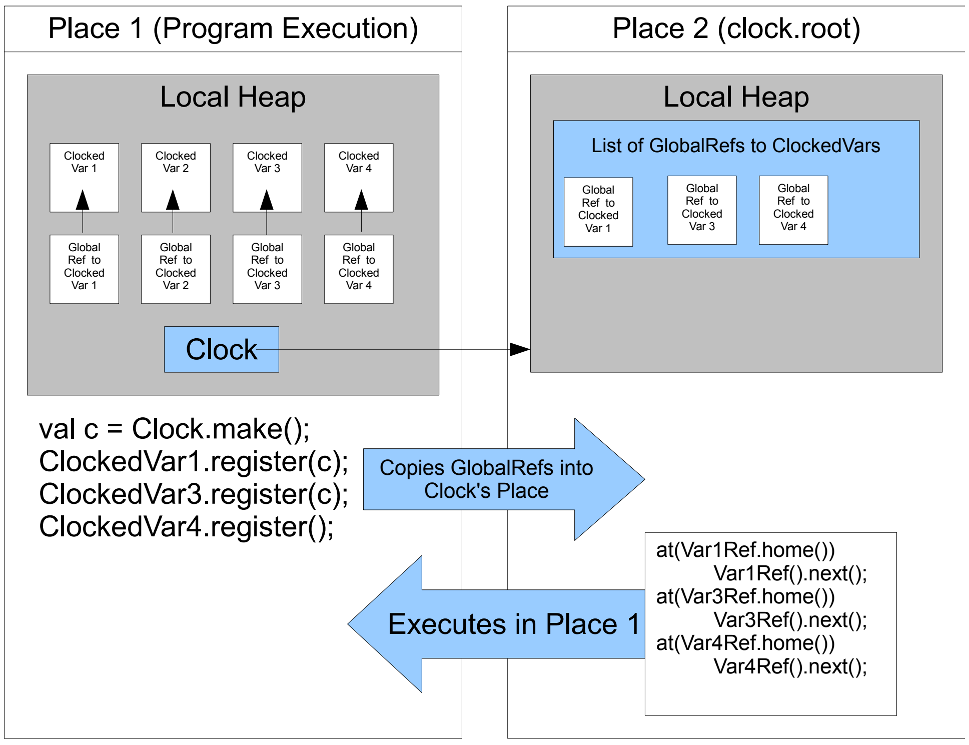
A list of GlobalRef objects is maintained by the Clock object. When a clocked variable is registered on that clock, its GlobalRef object is copied to the "root" of the clock (the `Place` it uses for its fields) and added to the list. Then, when the Clock advances from one phase to the next, it calls the next() method on all of the members of the list. Figure 3 illustrates this.

3.4.2 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 an 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 4 illustrates this.

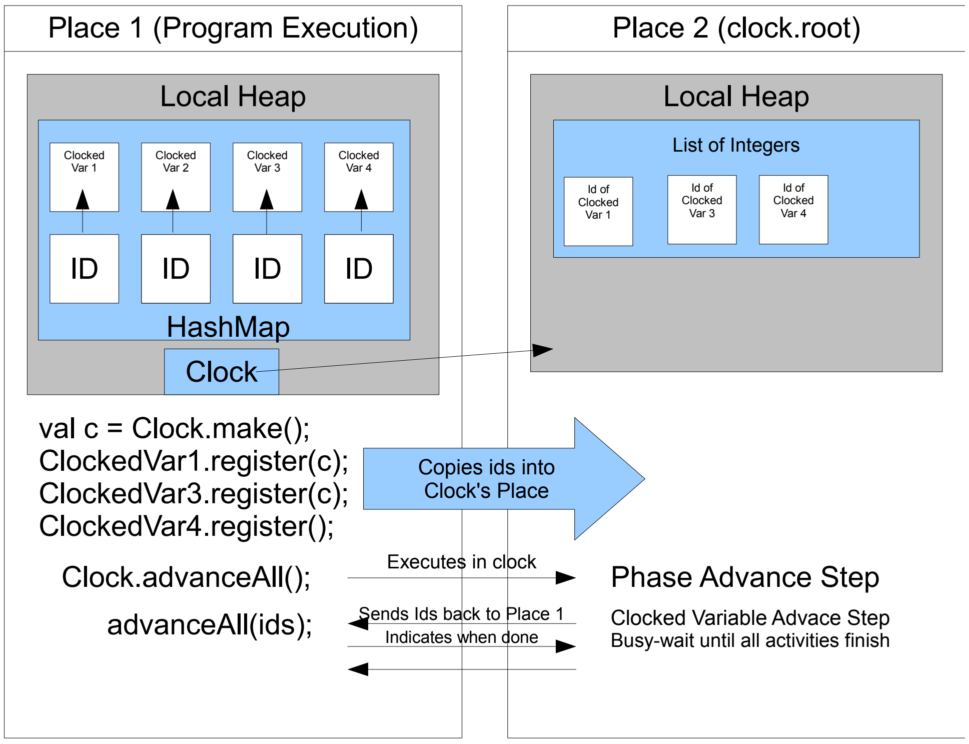
This design has the advantage of storing very little state within the Clock object itself: a list of primitive integers, rather than a larger struct. Since the burden of updating the clocked variables falls on the activity itself, there is also no need to switch places in order to update them—and this way, clocked variables in different places (and thus Activities) are updated concurrently, rather than consecutively


Place 1 (Program Execution)
Place 2 (clock.root)
Local Heap
Clocked Var 1
Clocked Var 2
Clocked Var 3
Clocked Var 4
Global Ref to Clocked Var 1
Global Ref to Clocked Var 2
Global Ref to Clocked Var 3
Global Ref to Clocked Var 4
Clock
Local Heap
List of GlobalRefs to ClockedVars
Global Ref to Clocked Var 1
Global Ref to Clocked Var 3
Global Ref to Clocked Var 4
Copies GlobalRefs into Clock's Place
Executes in Place 1

```
val c = Clock.make();
ClockedVar1.register(c);
ClockedVar3.register(c);
ClockedVar4.register();
```

```
at(Var1Ref.home())
        Var1Ref().next();
at(Var3Ref.home())
        Var3Ref().next();
at(Var4Ref.home())
        Var4Ref().next();
```

Figure 3: An activity in Place 1 using 4 ClockedVars under the GlobalRef method



```
val c = Clock.make();
ClockedVar1.register(c);
ClockedVar3.register(c);
ClockedVar4.register();

   Clock.advanceAll();

       advanceAll(ids);
```

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

Model	Dell Optiplex GX780
CPU	Intel(R) Core(TM)2 Duo @ 3.00GHz
RAM	4GB
OS	ArchLinux (3.2.4-1-ARCH)
HDD	250GB Serial ATA 7200rpm
Ethernet	Intel On-Board 1 Gigabit

Table 1: Hardware specifications for the benchmark applications

It was decided that the Map method would be used to implement Clocked Variables, as the cost of switching places is quite high in terms of efficiency. The Map method does this much less often than the Global Ref method (once per calling activity, rather than once per clocked variable).

4 Benchmarks and Evaluation

The performance of both types of clocked variable 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. For the "unclocked" case, synchronization and state updates were handled manually—the term refers to the absence of clocked variables, not the absence of clocks themselves. Care was taken to ensure that all versions of the benchmark programs operated correctly, and that the use of clocked/unclocked references was the *only* difference between the two versions of each benchmark. Each benchmark was executed 100 times, on a range of different values. The results shown here give the average values of those executions. Table 1 details the specifications for the hardware these benchmarks were executed on.

4.1 Conway's Game of Life

Conway's Game of Life is a fairly simple cellular automaton, originally described by the mathematician John Conway (Gardner 1970). The automaton 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 fixed rules that are only reliant on the current state of the board. At each step, the rules 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. Each cell is given its own thread, the state of each cell is calculated concurrently with the state of each other cell.

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.

Figure 5 gives the results for Conway's Game of Life for boards of various sizes. There is no significant difference between the clocked version and the unclocked version. This outcome was expected, as clocked and unclocked primitives are both updated via the same mechanism—directly copying the new value over the old value.

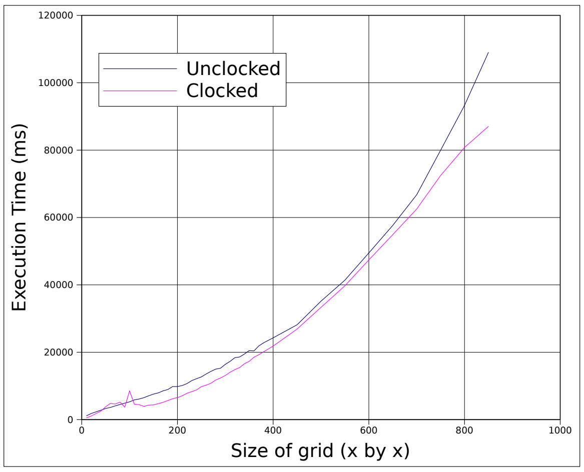


Figure 5: Conway's Game of Life: Clocked vs Unclocked execution times

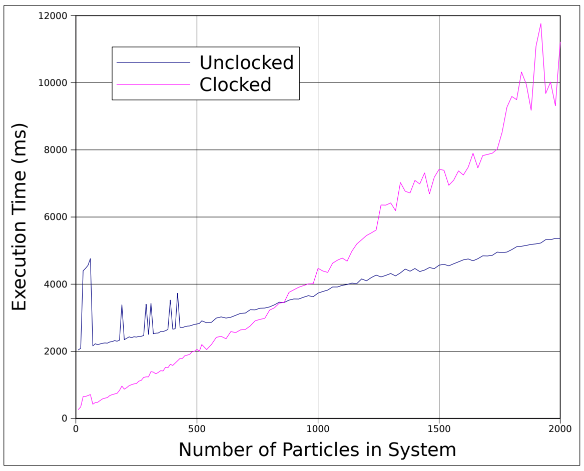


Figure 6: N-Body Simulation: Clocked vs Unclocked execution times

4.2 N-Body Simulation

An N-Body simulation is a physical simulation of a system of many interacting *particles*. N-Body problems are computationally intensive, as calculating the next state of a particle involves determining its interactions with 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.

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 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.

Figure 6 shows the results for the N-Body Benchmark, for various numbers of particles. As one would expect, execution time scales with the number of par-

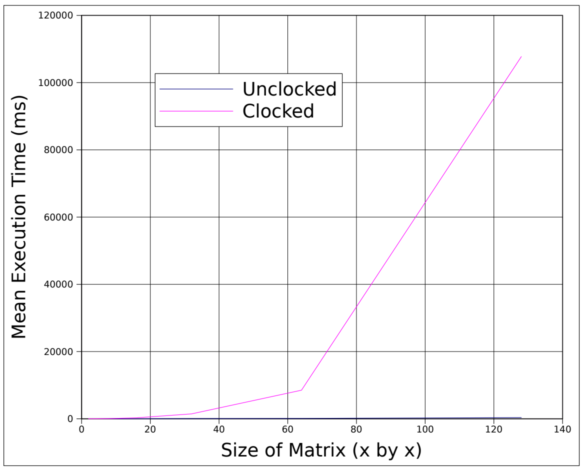


Figure 7: Sparse Matrix: Clocked vs Unclocked execution times

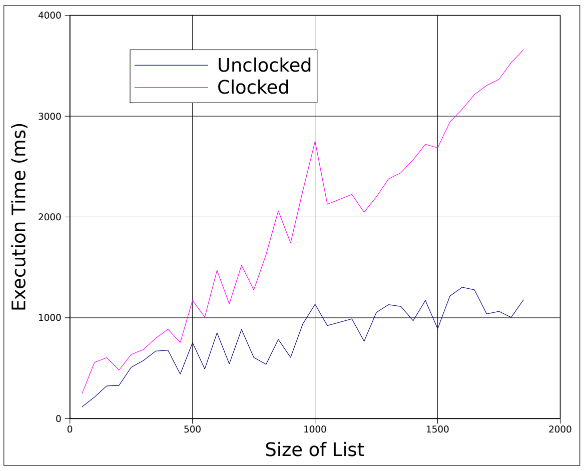


Figure 8: Linked List: Clocked vs Unclocked execution times

ticles present in the system (as this is an $O(n^2)$ algorithm). Interestingly, however, the clocked and unclocked versions clearly have a very different gradient. The update mechanism for the clocked version is a simple deep clone of the object (which only has primitive fields—essentially a struct), whereas the update mechanism for the unclocked version was method call on the object that performed two minor calculations and updated the fields. For smaller numbers of objects, the cloning method is much faster, but the time cost increases at a faster rate than the method-call update. The two methods are equal at around 800-850 objects, and the method-call update is faster for object numbers above that. From the graph, it appears that the method-call update has a constant cost associated with it (hence it starting at a much higher y value). This may be due to the update threads having to synchronize between the calculation phase and the update phase—something that doesn't need to happen in the clocked version.

4.3 Sparse Matrix Convolution

A Sparse Matrix allows more compact storage by storing only the *non-zero* values within the matrix. We used 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. This allows access to any cell within the matrix by following the links from the root node.

In this benchmark, a sparse matrix was used to represent an image which then had three filters applied to it via convolution. 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, the entire object graph is clocked via the reference to the root node of the matrix. In the unclocked 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.

Figure 7 gives the results for Sparse Matrix Convolution. Only a small number of points were sampled due to the very long execution time of this benchmark—but enough data were gathered to show that the clocked version of the Sparse Matrix is *vastly* slower than the unclocked version. Due to the single-write-per-phase nature of clocked variables, the Sparse Matrix was very slow to update. Each thread had to calculate the next value of its cells **before** any other thread could actually write to the matrix (as each cell insert necessitated a phase advancement, which—if peformed while threads were reading—breaks the convolution algorithm). After the values were calculated, each thread then inserts the new cell, advancing the phase after each insertion. Obviously this reduces the behaviour of the matrix to exactly that of the unclocked version—but with the high overhead of having to deep-copy the matrix at every clock advancement! Under a single-write-per-phase scheme, complex objects seem to perform quite slowly.

4.4 Linked List Microbenchmarks

For this benchmark, 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, as both were implemented in the exact same fashion, and both required the clock to advance after every method called on the list.

Figure 8 gives the results for clocked and unclocked Linked Lists. We can see that the clocked version of this data structure is much slower than the unclocked version. Every add, every remove—every operation that changes anything about the list—requires that the clock phase be advanced. This overhead simply does not exist in the unclocked version!

While clocked variables seem to offer some sort of benefit when used with primitives and objects with only primitive fields, they incur performance penalties with more complex data structures—at least, if we're restricted to one write per phase. Allowing multiple writes per clock phase might offer some performance improvements.

5 Alternate Approaches

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 reason is there

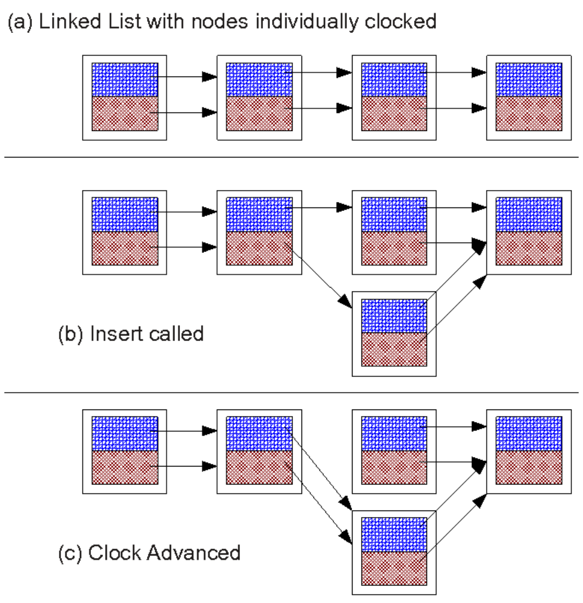


Figure 9: Approach 1: Clocking objects individually

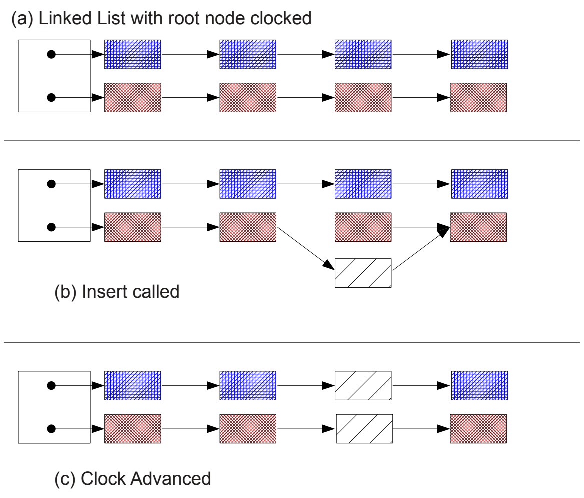


Figure 10: Approach 2: Single point-of-entry, clocking entire object graph

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 **structure** 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 difficulty escalates quite rapidly.

5.1 Two Possible Approaches

Consider Figure 9. Under this approach, each object is individually clocked, allowing multiple updates to occur to the list—provided the updates don't affect the same object twice. 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, erasing the *first* new node from the list. Multiple writes are unsafe under this approach.

A second approach (Figure 10) 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?

5.2 Two Better approaches

We can build on the first approach outlined above in order to make it slightly safer: we 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.

Our final approach attempts to solve this deadlocking problem by providing a single point of entry,

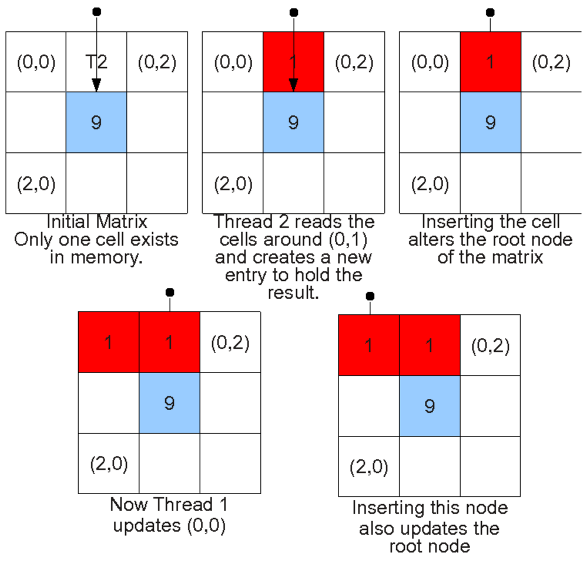

(0,0)
T2
(0,2)
9
(2,0)
Initial Matrix Only one cell exists in memory.
(0,0)
1
(0,2)
9
(2,0)
Thread 2 reads the cells around (0,1) and creates a new entry to hold the result.
(0,0)
1
(0,2)
9
(2,0)
Inserting the cell alters the root node of the matrix
1
1
(0,2)
9
(2,0)
Now Thread 1 updates (0,0)
1
1
(0,2)
9
(2,0)
Inserting this node also updates the root node

Figure 11: A Sparse Matrix Convolution

similar to the second approach outlined above. 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. So, what do these approaches look like in practice? A working implementation has not yet been developed, but Approach 3 lends itself well to *simulation*.

Figure 11 shows, we still cannot perform sparse matrix convolution—at least, not with this representation of sparse matricies. To understand this, we need to take a closer look at what is happening when a cell is updated. In the example shown, thread 2 is tasked with updating cell (0,1). To do this, it must first *read* cells (0,0), (0,1), (0,2), (1,0), (1,1), and (1,2). The result (1.00) is then written into cell (0,1)—but cell (0,1) does not currently exist in memory. So, the root of the matrix must be written to so that the cell can be inserted. The root holds a reference to the first non-zero cell in the first non-zero row, so currently it is pointing to cell (1,1). This reference needs to be updated, so we acquire a write-lock on the root node and insert the new cell.

Then, thread 1 attempts to update cell (0,0) via the same process. As this cell is before (0,1) in the row, the root needs to be updated again. Note that we have not yet advanced the clock. This requires obtaining a write-lock on the root, which throws an exception as the root has already been written to during this phase. We **cannot** solve this problem by advancing the clock before inserting (0,0), as this breaks the convolution algorithm. Cell (0,1) would be inserted into the matrix, and would thus affect any threads that have not yet read the old value of that location.

A solution could be to require all threads to perform their reads before **any** thread can write to the matrix. This would break each clock phase into two sub-phases—a read phase and a write phase. During this write phase, the clock can be advanced any number of times, as the old values are no longer required by the updating threads. However, this would result in the same level of performance as shown in Figure 7, as clock advancement is costly for a sparse matrix (Figure 12). It also renders the object unsafe to read from during the write phase, so any threads

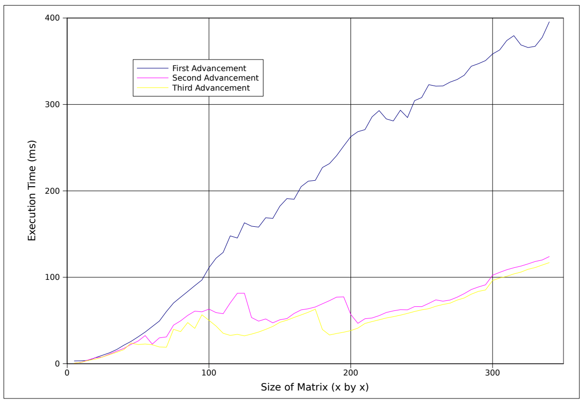


Figure 12: Clock Advancement Performance for a Sparse Matrix

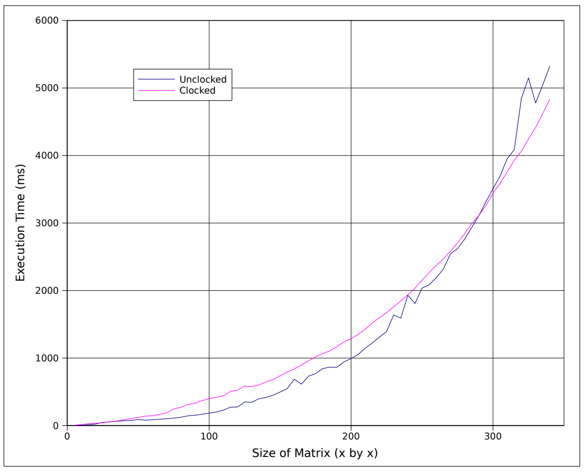


Figure 13: Using Approach 3 to solve the Sparse Matrix Performance Issue

external to this process would be forced to wait until the update process had finished in its entirety.

As we can see from Figure 13, the performance of Sparse Matrix Convolution is much improved—but this relies on a safe way to update the matrix.

Another solution, perhaps, is to implement things in a safer way. If a Sparse Matrix were implemented such that the first cell in each row was *always* present, even if zero-valued, a lock could be acquired on an entire row of the matrix, making structural changes safer. This would require that each row be updated strictly by one thread, so we have lost some concurrency here—but the performance would surely be better.

5.3 Related Work

The basic concept underlying clocked variables is not a new one. Software Transactional Memory (STM) (Shavit & Touitou 1995) provides database-like transactions for operations on shared memory. A transaction consists of one or more write operations performed on an object, which is then *committed* once the transaction is complete, causing an atomic update on the object to be performed. Transactions can be *aborted* at any time, and the pending changes are lost. This is similar, in many respects, to how Clocked Variables work—with some key differences. Both operate in a "phased" fashion; for STM, these phases are transactions, and for Clocked References, the phases are literal clock phases. Both maintain the old version of the memory location for reading purposes during these phases, and both "commit" changes to memory at the end of each phase.

For X10 specifically, there has been work to develop Phasor Accumulators (Shirako et al. 2009). These accumulators provide support for the accumulation of multiple values during a single clock phase (thus allowing multiple updates to a value) while maintaining the value from the previous state for reading purposes—similar to the Clocked Variables described in this paper. However, Phasor Accumulators were only designed with Number types in mind, and do not address Reference Types.

The use of revisions and isolation types (Burckhardt et al. 2010) offers a similar functionality to the scheme presented in this paper. Programmers can declare data they wish to be shared between tasks by using *isolation types*. Tasks are then executed and merged using *revisions*: isolated instances that can only read and modify the shared data locally. When the tasks are finished, the runtime merges the results, automatically resolving any conflicts that occur. The result is a concurrent programming model that can distribute and share data without concern over concurrent modifications, and successfully merge this shared data back into a coherent whole. Under such a scheme, it would be possible to split an array (such as in the Game Of Life case study) across multiple tasks, have each task read and update their assigned cells, and have the array merged successfully back into a consistent board state. Such a process may be used to provide high-performance concurrent programs (Burckhardt et al. 2011) that greatly improve upon the expected performance gained by parallelization alone. However, it is unclear how Revisions handle complex object graphs, as this has not been specifically addressed; nor does it seem to address the case where objects have reference types as fields.

6 Conclusion

Clocked Primitives are the most viable form of clocked variable presented in this paper, and offer no significant change in performance. The benefit gained from using them is a cleaner way of updating dual-state variables often found inside concurrent code.

Clocked References, however, were the main focus of this paper. While our initial attempts at solving this problem were not entirely successful, we have presented our results and offered insights into what could be done to solve this issue. There are many options for future work with Clocked References, and many new avenues to explore.

The implementation presented in this paper is available from
`http://ecs.vuw.ac.nz/~atkinsdani1/x10-clocked.tar.gz`.

References

Burckhardt, S., Baldassin, A. & Leijen, D. (2010), Concurrent programming with revisions and isolation types, *in* 'OOPSLA', ACM, New York, NY, USA, pp. 691–707.

Burckhardt, S., Leijen, D., Sadowski, C., Yi, J. & Ball, T. (2011), Two for the price of one: A model for parallel and incremental computation, *in* 'OOPSLA', Portland, Oregon.

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. & Sarkar, V. (2005), X10: an object-oriented approach to non-uniform cluster computing, *in* 'OOPSLA', ACM, New York, NY, USA, pp. 519–538.

Ebcioglu, K., Saraswat, V. & Sarkar, V. (2005), X10: an experimental language for high productivity programming of scalable systems (extended abstract), *in* 'P-PHEC'.

Ebcioglu, K., Saraswat, V. & Sarkar, V. (n.d.), 'X10: Programming for hierarchical parallelism and non-uniform data access (extended)'.

Gardner, M. (1970), 'The fantastic combinations of John Conway's new solitaire game 'Life'', *Scientific American* **223**, 120–123.

Lamport, L. (1978), 'Ti clocks, and the ordering of events in a distributed system', *Commun. ACM* **21**, 558–565.

Murthy, P. (2008), Parallel computing with X10, *in* 'Proceedings of the 1st international workshop on Multicore software engineering', IWMSE '08, ACM, New York, NY, USA, pp. 5–6.

Nystrom, N., Clarkson, M. R. & Myers, A. C. (2003), Polyglot: an extensible compiler framework for Java, *in* 'CC', Springer-Verlag, Berlin, Heidelberg, pp. 138–152.

Saraswat, V. (2011), 'X10 design notes', `https://x10.svn.sf.net/svnroot/x10/documentation/trunk/x10.man/v2.2/design-notes/design-v08.txt`.

Saraswat, V. A., Sarkar, V. & von Praun, C. (2007), X10: concurrent programming for modern architectures, *in* 'PPoPP', ACM, New York, NY, USA, pp. 271–271.

Saraswat, V., Bloom, B., Peshansky, I., Tardieu, O. & Grove, D. (2011), 'X10 language specification', `http://dist.codehaus.org/x10/documentation/languagespec/x10-221.pdf`.

Shavit, N. & Touitou, D. (1995), Software transactional memory, *in* 'PODC', ACM, New York, NY, USA, pp. 204–213.

Shirako, J., Peixotto, D., Sarkar, V. & Scherer, W. (2009), Phaser accumulators: A new reduction construct for dynamic parallelism, *in* 'IEEE International Parallel and Distributed Processing Symposium'.