The Design and Implementation of Clocked Variables in X10

Daniel Atkins, Alex Potanin, Lindsay Groves
School of Engineering and Computer Science
Victoria University of Wellington
Wellington, New Zealand
{Daniel.Atkins,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.

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 [11, 8]. 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 [11, 10, 3]. 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 [8].

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 [4]. 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 [7]. In X10, this synchronization is achieved through the use of a barrier-style structure based on Lamport’s Logical Clock [7]; the clock
Clocked variables provide the same functionality as manually maintaining two separate states in 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 [9]. 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 (i.e. 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)](image)

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 committed. 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 [9]. 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).

### 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.

#### 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 [6]. 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 parallelize 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 3: Conway’s Game of Life: Clocked vs Unclocked execution times

Figure 3 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.

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 4 shows the results for the N-Body Benchmark, for various numbers of particles. As one would expect, execution time scales with the number of particles 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 5 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 performed 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 uncomplicated 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 6 gives the results for clocked and uncomplicated Linked Lists. We can see that the clocked version of this data structure is much slower than the uncomplicated 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 uncomplicated version!

So it 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 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 7. 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.

Figure 5: Sparse Matrix: Clocked vs Unclocked execution times

Figure 6: Linked List: Clocked vs Unclocked execution times
A second approach (Figure 8) 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 object 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 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.

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, 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 9 shows, we still cannot perform sparse matrix convolution—at least, not with this representation of sparse matrices. 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 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 5, as clock advancement is costly for a sparse matrix (Figure 10). It also renders the object unsafe to read from during the write phase, so any threads external to this process would be forced to wait until the update process had finished in its entirety.

As we can see from Figure 11, 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) [12] 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 [13]. 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 [1] 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 [2] 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/~atkinsdan11/x10-clocked.tar.gz.

7. REFERENCES