Compile-Time Analysis and Specialization of Clocks in Concurrent Programs

Nalini Vasudevan[1], Olivier Tardieu[2], Julian Dolby[2], and Stephen A. Edwards[1]

[1] Department of Computer Science, Columbia University, New York, USA
{naliniv,sedwards}@cs.columbia.edu
[2] IBM T.J. Watson Research Center, New York, USA
{tardieu,dolby}@us.ibm.com

Abstract. Clocks are a mechanism for providing synchronization barriers in concurrent programming languages. They are usually implemented using primitive communication mechanisms and thus spare the programmer from reasoning about low-level implementation details such as remote procedure calls and error conditions.
Clocks provide flexibility, but programs often use them in specific ways that do not require their full implementation. In this paper, we describe a tool that mitigates the overhead of general-purpose clocks by statically analyzing how programs use them and choosing optimized implementations when available.
We tackle the clock implementation in the standard library of the X10 programming language—a parallel, distributed object-oriented language. We report our findings for a small set of analyses and benchmarks. Our tool only adds a few seconds to analysis time, making it practical to use as part of a compilation chain.

Keywords: Concurrency, Static Analysis, Synchronization, Clocks, X10, NuSMV

1 Introduction

The correct coordination and synchronization of concurrent tasks is one of the major challenges of concurrent programming. Low-level primitives, such as locks or compare-and-swap, can lead to optimum performance but they are hard to use and error-prone. In this paper, we consider higher-level concurrency constructs that are supplied in libraries and provide the user a richer, less error-prone abstraction. The usual disadvantage of general-purpose libraries is their generality: their implementation includes code to handle all possible cases, which slows down the relatively few cases each program uses.

We present an optimization technique that greatly reduces the performance penalty of a general-purpose concurrency library. We statically analyze the use of clocks—a form of synchronization barriers—in the Java-derived X10 concurrent programming language [1, 2] and use the results to safely substitute more specialized implementations of these standard library elements.

A clock in X10 is a structured form of synchronization barrier useful for expressing patterns such as wavefront computations and software pipelines. Concurrent tasks registered on the same clock advance in lockstep.

Our static analysis technique models an X10 program as a finite automaton; we ignore data but consider the possibility of clocks being aliased. We pass this automaton to the NuSMV model checker [3], which reports erroneous usage of a clock and whether a particular clock follows certain idioms. If the clocks are used properly, we use the idiom information to restructure the program to use a more efficient implementation of each clock. The result is a faster program that behaves like one that uses the general-purpose library.

Our analysis flow has been designed to be flexible and amenable to supporting a growing variety of patterns. In the sequel, we focus on inexpensive queries that can be answered by treating programs as sequential. While analysis time is negligible, speedup is considerable and varies across benchmarks from a few percent to a 3× improvement in total execution time.

In summary, our contributions are

- a methodology for the analysis and specialization of clocked programs;
- a set of cost-effective clock transformations;
- a prototype implementation: a plug-in for the X10 v1.5 tool chain; and
- experimental results on some modest-size benchmarks.

After a brief overview of the X10 language in Section 2 and the clock library in Section 3, we describe our static analysis technique in Section 4 and how we use its results to optimize programs in Section 5. We present experimental evidence that our technique can improve the performance of X10 programs in Section 6. We discuss related work in Section 7 and conclude in Section 8.

2 The X10 Programming Language

X10 [1, 2] is a parallel, distributed object-oriented language. To a Java-like sequential core it adds constructs for concurrency and distribution through the concepts of *activities* and *places*. An activity is a unit of work, like a thread in Java; a place is a logical entity that contains both activities and data objects.

The *async* construct creates activities; parent and child execute concurrently. The X10 program in Figure 1 uses clocks to recursively compute the first ten rows of Pascal's Triangle. The call of the method *row* on line 40 creates a new stream object, spawns an activity to produce the stream values, and finally returns the stream object to *main*. The rest of *main* executes in parallel with the spawned activity, printing the stream values as they are produced.

Spawned activities may only access final variables of enclosing activities, e.g.,

```
final int a = 3; int b = 4;
async { int x = a;     // OK: a is a final
        int y = b; }   // ERROR: b is not final
```

An X10 program runs in a fixed, platform-dependent set of places. The *main* method always runs in *place.FIRST_PLACE*; the programmer may specify where other activities run. Activities cannot migrate between places.

```
1  public class IntStream {
2    public final clock clk = clock.factory.clock(); // stream clock
3    private final int[] buf = new int[2]; // current and next stream values
4
5    public IntStream(final int v) {
6      buf[0] = v; // set initial stream value
7    }
8
9    public void put(final int v) {
10     clk.next(); // enter new clock phase
11     buf[(clk.phase()+1)%2] = v; // set next stream value
12     clk.resume(); // complete clock phase
13   }
14
15   public int get() {
16     clk.next(); // enter new clock phase
17     final int v = buf[clk.phase()%2]; // get current stream value
18     clk.resume(); // complete clock phase
19     return v;
20   } }
21
22 public class PascalsTriangle {
23   static IntStream row(final int n) {
24     final IntStream r = new IntStream(1); // start row with 1
25     async clocked(r.clk) { // spawn clocked task to compute row's values
26       if (n > 0) { // recursively compute previous row
27         final IntStream previous = row(n−1);
28         int v; int w = previous.get();
29         while (w != 0) {
30           v = w; w = previous.get();
31           r.put(v+w); // emit row's values
32         }
33       }
34       r.put(0); // end row with 0
35     }
36     return r;
37   }
38
39   public static void main(String[] args){
40     final IntStream r = row(10);
41     int w = r.get(); // print row excluding final 0
42     while (w != 0) { System.out.println(w); w = r.get(); }
43   } }
```

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 120 120 45 10 1

Fig. 1. A program to compute Pascal's Triangle in X10 using clocks

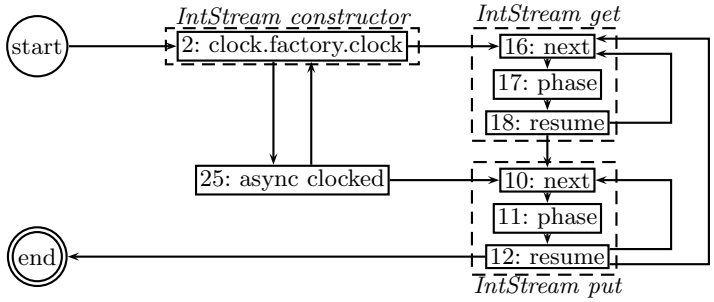

Fig. 2. The automaton model for the clock in the Pascal's Triangle example

```
final IntStream s = new IntStream(4);
async (place.LAST_PLACE) { // spawn activity at place.LAST_PLACE
        // cannot call methods of s if LAST_PLACE != FIRST_PLACE
        final int i = 3;
        async (s) s.put(i); // spawn activity at the place of s; s is local => ok to deref
}
```

Activities that share a place share a common heap. While activities may hold references to remote objects, they can only access the fields and methods of a remote object by spawning an activity at the object's place.

X10 also introduces *value classes*, whose fields are all *final*. The fields and methods of an instance of a value class may be accessed remotely, unlike normal classes. Clocks are implemented as value classes.

X10 provides two primitive constructs for synchronization: *finish* and *when*. *finish* p q delays the execution of statement q until after statement p and all activities recursively spawned by p have completed. For example,

```
finish { async { async { System.out.print(“Hello”); } } } System.out.println(“ world”);
```

prints "Hello world." The statement *when*(e) p suspends until the Boolean condition e becomes true, then executes p atomically, i.e., as if in one step during which all other activities in the same place are suspended.[3]

X10 also permits unconditional atomic blocks and methods, which are specified with the *atomic* keyword. For example,

```
atomic { int tmp = x; x = y; y = tmp; }
```

3 Clocks in X10

Clocks in X10 are a generalization of barriers. Unlike X10's *finish* construct, clocks permit activities to synchronize repeatedly. In contrast to *when* constructs, they provide a structured, distributed, and determinate form of coordination. While a complete discussion of X10's clocks is beyond the scope of this paper, the following sections will demonstrate that clocks are amenable to efficient and effective static analysis.

Figure 3 lists the main elements of the clock API. An activity must be registered with a clock to interact with it. Activities are registered in one of two ways: creating a clock with the *clock.factory.clock()* static method automatically registers the calling activity with the new clock. Also, an activity can register activities it spawns with the *async clocked* construct.

```
final clock clk = clock.factory.clock();
async clocked(clk) { A1; clk.next(); A2; clk.next(); A3 }
async clocked(clk) { B1; clk.next(); B2; }
async { C; }
M1; clk.resume(); M1_2; clk.next(); M2;
```

A clock synchronizes the execution of activities through phases. A registered activity can request the clock to enter a new phase with a call to *next*, which

[3] X10 does not guarantee that p will execute if e holds only intermittently.

blocks the activity until all other registered activities are done with the current phase, i.e., have called *next* or *resume*. For instance, in the program above, action A1 must complete before action B2 can start. In other words, A1 and B1 belong to phase 1 of clock *clk*; A2 and B2 belong to phase 2. C, however, does not belong to an activity registered with *clk*; it may execute at any time.

The *resume* method provides slack to the scheduler.[4] An activity calls *resume* when it is done with the current clock phase but does not yet need to enter the next. Unlike *next*, *resume* does not block the activity, and the activity must still call *next* to enter the next phase. In the example above, while M1 must terminate before A2 can start and A1 must terminate before M2 can start, M1_2 may start before A1 completes and continue after A2 starts because of *resume*.

In Figure 1, the value at the pth column and nth row of this triangle ($0 \leq p \leq n$) is the number of possible unordered choices of p items among n. One task per row produces the stream of values for the row by summing the two entries from the row immediately above. Each stream uses a clock to enforce single-write-single-read interleaving, so each task registers with two clocks: its own and the clock for the row immediately above. The clocks ensure proper inter-row coordination.

The *phase* method returns the current phase index (counting from 1). Figure 1 demonstrates this and also how activities can register with multiple clocks (using recursion in this example).

Finally, activities can explicitly unregister from a clock by calling *drop*. Activities are implicitly unregistered from their clocks when they terminate.

The operations of an activity on a clock modify the state of this activity w.r.t. that clock. Figure 4 shows the behavior. The activity may be in one of four states: *Active*, *Resumed*, *Inactive*, or *Exception*. Transitions are labeled with clock-related operations: *async clocked*, *resume*, *next*, *phase*, and *drop*. For example, an activity moves from the *Active* state to *Resumed* if it calls *resume* on the clock. If it calls *resume* again, it moves to the *Exception* state. Any operation that leads to the *Exception* state throws the *ClockUseException* exception.

3.1 Clock Patterns

We now describe the four clock patterns we currently identify. We believe that our techniques can also be applied to find other patterns.

Our first pattern is concerned with exceptions: can an activity reach the exception state for a particular clock? The default clock implementation looks for transitions to this state and throws *ClockUseException* if they occur. Aside from the annoyance of runtime errors, runtime checks slow down the implementation. We want to avoid them if possible.

Our algorithm finds that the clocks are used properly in the program of Figure 1, e.g., no task erroneously attempts to use a clock it is not registered with. Therefore, it substitutes the default implementation with one that avoids the overhead of runtime checks for these error conditions.

[4] The *resume* method is typically used in activities registered with multiple clocks.

```
/* Create a new clock. Register the calling activity with this clock. */
final clock clk = clock.factory.clock();

/* Spawn an activity registered with clocks clk_1, ..., clk_n with body p. */
async clocked(clk_1, ..., clk_n) p

public interface clock {
        /* Notify this clock that the calling activity is done with whatever it intended
         * to do during this phase of the clock. Does not block. */
        void resume();

        /* Block until all activities registered with this clock are ready to enter the next
         * clock phase. Imply that calling activity is done with this phase of the clock. */
        void next();

        /* Return the phase index. Calling activity cannot be resumed on the clock. */
        int phase();

        /* Unregister the caller from this clock; release it from having to participate */
        void drop();
}
```

Fig. 3. The clock API

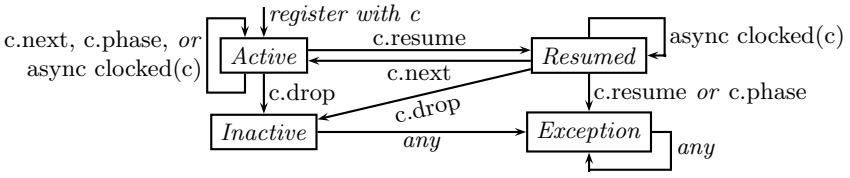

Fig. 4. The state of one activity with respect to clock c

We also want to know whether *resume* is ever called on a clock. This feature's implementation requires additional data structures and slows down all clock operations. We discuss this and other optimizations in Section 5.

Activities often use clocks to wait for sub-activities to terminate. Consider

```
final clock clk = clock.factory.clock();
async clocked (clk) A1;
async A2;
async clocked (clk) A3;
clk.next();
A4;
```

Here, if A1 and A2 do not interact with clock *clk*, *clk.next()* requires activities A1 and A3 to terminate before A4 starts executing and nothing else. In particular, A2 and A4 may execute in parallel. We want to detect sub-activities that are registered with the clock yet never request to enter a new clock phase.

Finally, the default clock implementation enables distributed activities to synchronize. If it turn out that all registered activities belong to the same place, a much faster clock implementation is possible. Our Pascal's Triangle program is a trivial example of this since all activities are spawned in the default place.

4 The Static Analyzer

In this section, we describe how we detect clock idioms. We start from the program's abstract syntax tree, compute its call graph, and run aliasing analysis on clocks. We then abstract data by replacing conditional statements with non-deterministic choice. From the control-flow graph of this abstract program, we extract one automaton per clock. This gives a conservative approximation of the sequences of operations that the program may apply to the clock.

To a model checker, we feed the automaton for the control-flow of the program along with an automaton model of the clock API and a series of temporal logic properties, one for each idiom of interest. For each property and each clock, the model checker either proves the property or returns a counterexample in the form of a path in the automaton that violates the property.

We use the T.J. Watson Libraries for Analysis (WALA) [4] for parsing, call- and control-flow-graph construction, and aliasing analysis. We have extended the Java frontend of WALA to accommodate X10 and extract from the AST the required automata in the form of input files for the NuSMV model checker [3].

We now describe the key technical steps in detail. We start with the construction of the automaton, then discuss the encoding of the clock API, the temporal properties, and finally aliasing.

4.1 Constructing the Automaton

Figure 2 shows the automaton we build for the clock *clk* in Figure 1. Each operation on *clk* in the text of the program becomes one state, which we label with the type of operation and its line number. Transitions arise from our abstraction of the program's control flow. We highlighted the fragments corresponding to the constructor and methods of the *IntStream* class.

methods Each method body becomes a fragment of the automaton. Each call of a method adds a transition to and from its entry and exit nodes. For example, since *get* may be called twice in a row (lines 28 and 30), we added the edge from its exit node "18: resume" to its entry node "16: next." It may also be called after put, looping from line 31 back to line 30, so we added an edge from node "12: resume" to node "16: next."

conditionals We ignore guards on conditionals and add arcs for both branches. For example, the *if* on line 26 runs immediately after the *async clocked* on line 25. The "then" branch of this *if* runs line 27, which starts with a call to *row* that starts by constructing an *IntStream* (line 24) whose constructor calls *clock.factory.clock()* (line 2). This gives the arc from node "25: async clocked" to "2: clock.factory.clock." The "else" branch is line 34, which calls *put*, which starts with a call to *next* (line 10). This gives the arc to "10: next."

async Because we are not checking properties that depend on interactions among tasks, we can treat a spawned activity as just another path in the program. When execution reaches an *async* construct, we model it as either jumping directly to the task being spawned or skipping the child and continuing to execute the parent. This is illustrated in Figure 5.

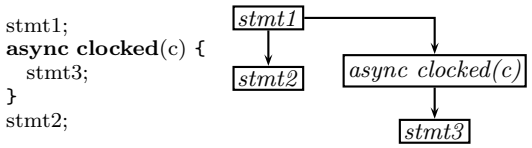

Fig. 5. Modeling *async* calls

In our Pascal's Triangle example, this means control may flow from the *IntStream* constructor exit point "2: clock.factory.clock" to the *async* construct "25: async clocked" or ignore the *async* and flow back via the *return* statement to the subsequent get method call in either *main* or *row*, i.e., node "16: next."

We give the NuSMV code for the automaton in an extended version of this paper [5].

We build one automaton for each call of *clock.factory.clock* in the source code, meaning our algorithm does not distinguish clocks instantiated from the same allocation site. So we construct only one automaton for our example, even though the program uses ten (very similar) clocks when it executes.

We have taken a concurrent program and transformed it into a sequential program with multiple paths. Thanks to this abstraction, we avoid state space explosion both in the automaton construction and in the model checker.

4.2 Handling Async Constructs with the Clock Model

Our model of clock state transitions—Figure 4—only considers a single activity, but X10 programs may have many. As explained in Section 4.1, we model *async* constructs with nondeterministic branches, so we have to extend the typestate automaton for the clock to do the same.

Figure 6 shows the additional transitions necessary for handling *async* actions. We consider two cases: when analyzing clock c and we encounter *async clocked(c)*, the new activity stays either *Active* or *Resumed*. By contrast, if we encounter an *async* not clocked on c, the new activity starts in the *Inactive* state (arcs labeled just *async*).

We give the NuSMV code for the complete automaton in the extended version of this paper [5].

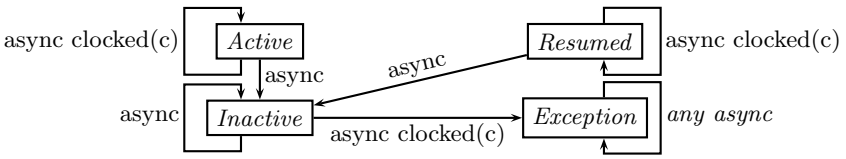

Fig. 6. Additional transitions in the clock state for modeling *async* operations

4.3 Specifying Clock Idioms

Once we have the automata modeling the program and clock state, it becomes easy to specify patterns for NuSMV as temporal logic formulas.

Three patterns are CTL reachability properties of the form

SPEC AG(!(target))

where *target* is either the *Exception* state, a *resume* operation, or an *async clocked(c)* node annotated with a place expression, that is, a remote activity creation. See the extended version of this paper [5] for details.

We check for the fourth pattern—whether spawned activities ever call *next* on the clock—by looking for control-flow paths that contain an *async clocked(c)* operation followed by a *c.next* operation. The LTL specification is

LTLSPEC G(c_next −> **H**(!async_clocked_c))

The extended version of this paper [5] gives the complete NuSMV input file for the Pascal's Triangle example.

4.4 Combining Clock Analysis with Aliasing Analysis

Clocks can be aliased just like any objects. Figure 7 shows an example of aliasing of clocks in X10. We create two clocks *c1* and *c2*. *x* can take the value of either *c1* or *c2* depending on the value of *n*.

```
final clock c1 = clock.factory.clock();
final clock c2 = clock.factory.clock();
..
final clock x = (n > 1)? c1: c2;
x.resume();
x.next();
c1.next();
```

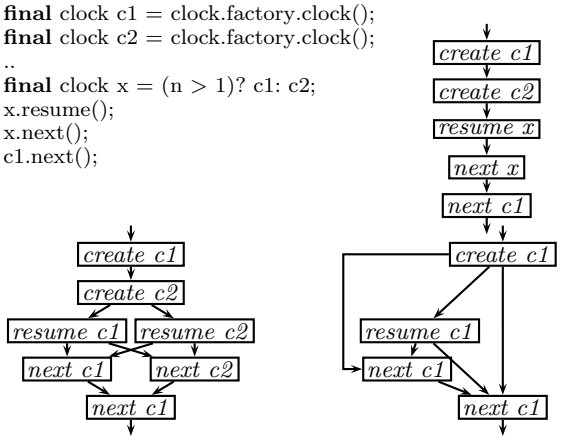

Fig. 7. *Top Left*: Aliasing clocks in X10, *Top Right*: the corresponding control flow graph, *Bottom Left*: our abstraction, *Bottom Center*: automaton for *c1*, *Bottom Right*: automaton for *c2*

We could abstract the program into two control paths, one that assumes $x = c1$ and one that assumes $x = c2$. However, this would produce a number of paths exponential in the number of aliases that have to be considered simultaneously.

Instead, we chose to bound the size of our program abstraction (at the expense of precision) as shown in the bottom three diagrams of Figure 7. We consider each clock operation on x in isolation and apply it non-deterministically to any of the possible targets of x as returned by WALA's aliasing engine.

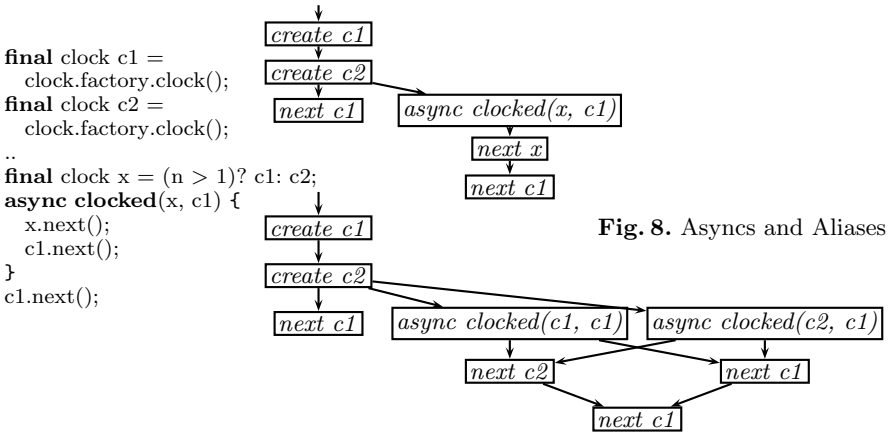

Fig. 8. Asyncs and Aliases

Figure 8 shows how we extend this idea to *async* constructs. Our tool reports that operations on clock *c1* cannot throw *ClockUseException*. However, it fails to establish the same for *c2* because our abstraction creates a false path—*next c2* following *async clocked(c1,c1)*.

5 The Code Optimizer

Results from our static analyzer drive a code optimizer that substitutes each instance of the clock class for a specialized version. We manually wrote an optimized version of the clock class for each clock pattern we encountered in our test cases; a complete tool would include more. Our specialized versions include a clock class that does not check for protocol violations (transitions to the *exception* state) and one that does not support *resume*.

There is one abstract clock base class that contains empty methods for all clock functions; each specialized implementation has different versions of these methods that uses X10 primitives to perform the actual synchronization. Our optimizer changes the code (actually the AST) to use the appropriate derived class for each clock, e.g., *c = clock.factory.clock()* would be replaced with *c = clock.factory.clockef()* if clock *c* is known to be exception-free.

The top of Figure 9 shows the general-purpose implementation of *next*. The *clock* value class contains the public clock methods; the internal *ClockState* maintains the state and synchronization variables of the clock. The *next* method first verifies that the activity is registered with the clock (and throws an exception otherwise), then calls the *select* function to wait on a *latch*: a data structure that indicates the phase. The *latch* is either *null* if *next()* was called from an *active()* state or holds a value if *next()* was called from a *resumed()* state. The *wait* function blocks and actually waits for the clock to change phase. The *check* method decrements the number of activities not yet resumed on the clock and advances the clock phase when all activities registered on the clock are resumed.

A basic optimization: when we know the clock is used properly, we can eliminate the registration check in *next* and elsewhere. Figure 9 shows such an exception-free implementation.

```
// The default implementation

class ClockState {
..
  atomic int check() {
    int resumedPhase = currentPhase;
    if (remainingActivities-- == 0) {
        // set the number of activities
        // expected to resume
        remainingActivities =
             registeredActivities;
        // advance to the next phase
        currentPhase++;
    }
    return resumedPhase;
  }

 void wait(final int resumedPhase) {
      when(resumedPhase != currentPhase);
  } }
```

```
value class clock {
..
  final ClockState state = new ClockState();
..
  void select(nullable<future<int>> latch) {
    if (latch == null) {
      async (state) state.wait(state.check());
    } else {
      final int phase = latch.force();
      async (state) state.wait(phase);
    }
  }

  public void next() {
    if (!registered())
        throw new ClockUseException();
    finish select(ClockPhases.put(this, null));
  } }
```

```
// An exception-free implementation

public void next() {
  finish
    select(ClockPhases.put(this, null));
}
```

```
// For when resume() is never used

void select() {
  async (state) state.wait(state.check());
}

public void next() {
  if (!registered())
      throw new ClockUseException();
  finish select();
}
```

```
// For when a clock is only in a single place

void select(nullable<future<int>> latch) {
  if (latch == null)
    state.wait(state.check());
  else
    state.wait(latch.force());
}

public void next() {
  if (!registered())
      throw new ClockUseException();
  select(ClockPhases.put(this, null));
}
```

Fig. 9. Various implementations of *next* and related methods

Accommodating *resume* carries significant overhead, but if we know the *resume* functionality is never used, we can simplify the body of *select* as shown in Figure 9. We removed the now-unneeded *latch* object and can do something similar in other methods (not shown).

Figure 9 also shows a third optimization. Because clocked activities may be distributed among places, synchronization variables have to be updated by remote activities. When we know a clock is only used in a single place, we dispense with the *async* and *finish* constructs.

Table 1. Experimental Results

Example	Clocks	Lines	Result	Speed Up	Analysis Time Base	Analysis Time NuSMV
Linear Search	1	35	EF, NR, L	35.2%	33.5s	0.4s
Relaxation	1	55	EF, NR, L	87.6	6.7	0.3
All Reduction Barrier	1	65	EF, NR	1.5	27.2	0.1
Pascal's Triangle	1	60	EF, L	20.5	25.8	0.4
Prime Number Sieve	1	95	NR, L	213.9	34.7	0.4
N-Queens	1	155	EF, NR, ON, L	1.3	24.3	0.5
LU Factorization	1	210	EF, NR	5.7	20.6	0.9
MolDyn JGF Bench.	1	930	NR	2.3	35.1	0.5
Pipeline	2	55	Clock 1: EF, NR, L Clock 2: EF, NR, L	31.4	7.5	0.5
Edmiston	2	205	Clock 1: NR, L Clock 2: NR, L	14.2	29.9	0.5

EF: No ClockUseException, NR: No Resume, ON: Only the activity that created the clock calls *next* on it, L: Clocked used locally (in a single place)

6 Results

We applied our static analyzer to various programs, running it on a 3 GHz Pentium 4 machine with 1 GB RAM. Since we want to measure the overhead of the clock library, we purposely run our benchmarks on a single-core processor. Table 1 shows the results. For each example, we list its name, the number of clock definitions in the source code, its size (number of lines of code, including comments), what our analysis discovered about the clock(s), how much faster the executable for each example ran after we applied our optimizations, and finally the time required to analyze the example. (The *Base* column includes the time to read the source, build the IR, perform pointer analysis, build the automata, etc.; *NuSMV* indicates the time spent running the NuSMV model checker. Total time is their sum.)

The first example is a paced linear search algorithm. It consists of two tasks that search an array in parallel and use a clock to synchronize after every comparison. The Relaxation example, for each cell in an array, spawns one activity that repeatedly updates the cell value using the neighboring values. It uses a clock to force these activities to advance in lockstep. The All Reduction Barrier example is a variant on Relaxation that distributes the array across multiple places. Pascal's Triangle is the example of Figure 1. Our prime number sieve uses the Sieve of Eratosthenes. N-Queens is a brute-force tree search algorithm that uses a clock to mimic a join operation. LU Factorization decomposes a matrix in parallel using clocks. We also ported the MolDyn Java Grande Forum Benchmark [6] in X10 with clocks, the largest application on which we ran our tool. Pipeline has three stages; its buffers use two clocks for synchronization. Edmiston aligns substrings in parallel and uses two clocks for synchronization.

The Result column lists the properties satisfied by each example's clocks. For example, the N-Queens example cannot throw *ClockUseException*, does not call *resume*, and uses only locally created clocks. Our tool reports the JGF benchmark may throw exceptions and pass clocks around, although it also does not call *resume*. In truth, it does not throw exceptions, but our tool failed to establish this because of the approximations it uses. This reduced the speedup we could achieve, but does not affect correctness.

The Linear Search, Relaxation, Prime Number Sieve, and Pipeline examples use clocks frequently and locally, providing a substantial speedup opportunity. Although our analysis found N-Queens satisfies the same properties as these, we could improve it up only slightly because its clock is used rarely and only in one part of the computation. Switching to the local clock implementation provided the majority of the speedup we observed, but our 5% improvement on the already heavily optimized distributed LU Factorization example is significant.

Our tool analyzed each example in under a minute and the model checker took less than a second in each case. Most of the construction time is spent in call- and control-flow graph constructions and aliasing analysis, which are already done for other reasons, so the added cost of our tool is on the order of seconds, making it reasonable to include as part of normal compilation.

7 Related Work

Typestate analysis [7] tracks the states that an object goes through during the execution. Standard typestate analysis and concurrency analysis are disjoint. Our analysis can be viewed as a typestate analysis for concurrent programs. Clocks are shared, stateful objects. We therefore have to track the state of each clock from the point of view of each activity.

Model checking concurrent programs [8, 3] is usually demanding because of the potential for exponentially large state spaces often due to having to consider different interleavings of concurrent operations. In contrast, our technique analyzes concurrent programs as if they were sequential—we consider spawned tasks to be additional execution paths in a sequential program—hence avoiding the explosion.

Concurrency models come in many varieties. Vasudevan et al. [9] showed that the state space explosion can also be avoided by carefully choosing the primitives of the concurrent programming language. Unfortunately, this restricts the flexibility of the language. Our work focuses on concurrency constructs similar to those advocated by Vasudevan et al., but features like resume and aliased clocks are absent from their proposal. We trade a more flexible concurrency model against the need for further approximation in modeling the programs.

Static analysis of concurrency depends greatly on the underlying model. Although X10 supports both message-passing-style and shared-memory-style concurrency (in the case of co-located activities), we focus exclusively on its message-passing aspects, as have others. Mercouroff [10] approximates the number of messages between tasks in CSP [11] programs. Reppy and Xiao [12] analyze

communication patterns in CML. Like ours, their work aims at identifying patterns amenable to more efficient implementations. They attempt to approximate the number of pending send and receive operations on a channel. Our work is both more specific—it focuses on clocks—and more general: our tool can cope with any CTL or LTL formula about clock operations.

Reppy and Xiao use modular techniques; we consider an X10 program as a whole. A modular approach may improve out tool's scaling, but we have not explored this yet.

Analysis of X10 programs has also been considered. Agarwal et al. [13] describe a novel algorithm for may-happen-in-parallel analysis in X10 that focuses on atomic sections. Chandra et al. [14] introduce a dependent type system for the specification and inference of object locations. We could use the latter to decide whether activities and clocks belong to the same place.

8 Conclusions and Future Work

We presented a static analysis technique for clocks in the X10 programming language. The result allows us to specialize the implementation of each clock, which we found resulted in substantial speed improvements on certain benchmark programs. Our technique has the advantage of being able to analyze a concurrent language using techniques for sequential code.

We treat each clock separately and model subtasks as extra paths in the program, much like conditionals. We abstract away conditional predicates, which simplifies the structure at the cost of introducing false positives. However, our technique is safe: we revert to the unoptimized, general purpose clock implementation when we are unsure a particular property is satisfied. Adding counter-example guided abstraction refinement [15] could help.

We produce two automata for each clock: one models the X10 program; the other encodes the protocol (typestate) for the clock. We express the automata in a form suitable for the NuSMV model checker. Experimentally, we find NuSMV is able to check properties for modestly sized examples in seconds, which we believe makes it fast enough to be part of the usual compilation process.

In the future, we plan to check for properties such as deadlock, which would involve considering interleavings rather than just the sequential analysis we currently use. For this reason we started with a powerful model checker like NuSMV. We also want to investigate other applications, such as using clock information from our static analyzer to refine pointer analysis of X10 programs.

Our current approach analyzes each clock as a whole. We may be able to improve the granularity by analyzing the program on a statement-by-statement basis. This would enable optimizing a clock operation at a particular line number differently from the same operation at another line number if we know more about the context of one operation compared to the other.

References

1. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. **40**(10) (2005) 519–538
2. Saraswat, V.A., Sarkar, V., von Praun, C.: X10: concurrent programming for modern architectures. In: PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2007) 271–271
3. Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV version 2: An OpenSource tool for symbolic model checking. In: Proceedings of the International Conference on Computer-Aided Verification (CAV). Volume 2404 of Lecture Notes in Computer Science., Copenhagen, Denmark (July 2002) 359–364
4. IBM, et al.: T. j. watson libraries for analysis (2006) http://wala.sourceforge.net.
5. Vasudevan, N., Tardieu, O., Dolby, J., Edwards, S.A.: Analysis of clocks in x10 programs (extended). Technical Report CUCS–052–08, Columbia University, Department of Computer Science, New York, New York, USA (December 2008)
6. Smith, L.A., Bull, J.M., Obdrzálek, J.: A parallel java grande benchmark suite. In: Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), New York, NY, USA, ACM (2001) 8–8
7. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering **12**(1) (1986) 157–171
8. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems **8**(2) (1986) 244–263
9. Vasudevan, N., Edwards, S.A.: Static deadlock detection for the SHIM concurrent language. In: Proceedings of the International Conference on Formal Methods and Models for Codesign (MEMOCODE), Anaheim, California (June 2008)
10. Mercouroff, N.: An algorithm for analyzing communicating processes. In: Proceedings of the 7th International Conference on Mathematical Foundations of Programming Semantics, London, UK, Springer (1992) 312–325
11. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Upper Saddle River, New Jersey (1985)
12. Reppy, J., Xiao, Y.: Specialization of CML message-passing primitives. SIGPLAN Notices **42**(1) (2007) 315–326
13. Agarwal, S., Barik, R., Sarkar, V., Shyamasundar, R.K.: May-happen-in-parallel analysis of x10 programs. In: Proceedings of Principles and Practice of Parallel Programming (PPoPP), New York, NY, USA, ACM (2007) 183–193
14. Chandra, S., Saraswat, V., Sarkar, V., Bodik, R.: Type inference for locality analysis of distributed data structures. In: Proceedings of Principles and Practice of Parallel Programming (PPoPP), New York, NY, USA, ACM (2008) 11–22
15. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proceedings of the International Conference on Computer-Aided Verification (CAV). Volume 1855 of Lecture Notes in Computer Science., Chicago, Illinois (July 2000) 154–169