The Analysis of Clocks in X10 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 Research, New York, USA
{tardieu,dolby}@us.ibm.com

Abstract. Distributed and concurrent programming languages are becoming more prevelant with the emergence of new parallel architectures. Reducing communication and synchronization costs is a major issue in concurrent languages.
In this paper, we deal with the X10 programming language, a distributed concurrent language that uses clocks for synchronization. Using these clocks can be costly because they provide synchronization between multiple tasks. To partially mitigate this, we use static analysis to determine properties of each clock that can be used to optimize its implementation. Experimentally, we find our procedure runs in only a few seconds for modest-sized programs, making it practical to use as part of a compilation chain.

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

1 Introduction

Concurrent programming constructs need to be designed and implemented carefully to keep overheads low. Typical sources include creating threads, communication, and synchornization. The X10 programming language is a distributed concurrent language that spawns concurrent tasks. Clocks in X10 synchronize tasks and resemble barriers. Most clocks are used in some idiomatic way. If can identify patterns in the behavior of a particular clock, we can fine-tune its implementation for better performance.

In this paper, we describe a method to classify clocks using static analysis. We use abstraction techniques to make the system finite. For the kind of patterns we want to detect, we do not need to analyze all possible interleavings of tasks, which allows us to use sequential analysis to classify clocks and minimize the complexity of our analysis low.

Not only does our tool provide input to the code optimizer but it also gives feedback to the user about the properties of each clock in the program.

2 Related Work

[?]

3 The X10 Programming Language

4 Clocks In X10

Clocks in X10 are generalizations of barriers. A single activity may register on multiple clocks. Mulitple activities may register on a single clock at many time. Creation of a clock is an implicit register in the activity. Clocks may be explicitly passed to activities using the *clocked* construct. An activity may block on a clock waiting for activities (that are also clocked on the same clock) to synchronize by executing *next()*. It may resume a clock by executing *resume()*. *resume()* indicates the end of thexecution of current phase. When an actvity finishes, all clocks registered with the activity are implictly deregistered. An activity can explicitly deregister with a clock by calling *drop()* on that clock.

```
clock c = clock.factory.clock()
async (clocked c) {
  ..
  c.next();
  ..
  c.next();
}
async(clocked c} {
  ..
  c.next();
  ..
  c.resume();
  ..
}
c.next();
..
c.next();
..
c.drop();
```

Fig. 1. Example of Clock in X10

Figure **??** shows an example of an X10 program with clocks. The main activity creates a clock *c* and forks two activites in parallel. Both the activities are clocked on c and have to synchronize on next(). The main activity is also registered with *c* and therefore will also take part in the synchronization. The activities run in parallel until the *next* call where they block to synchronize. During the first phase, all the activities synchronize on *next*. During the second phase, one of the spawned activities calls resume, indicating that it has finished the current phase and goes on with the execution of the rest of the code. The other two activities do not wait for the resumed activity until the next phase. Finally, the main activity drops the clock.

Clocks can be referenced like any Java object and can be nested within asyncs.

5 Obtaining the Intermediate Representation

6 Abstracting X10 Programs

Since we analyze the clock patterns in X10, we abstract away other instructions that are not relevant to clocks. Also, we abstract away the predicates of conditional statements and assume all branches can be taken at any time. When we check for properties like protocol violation, we leave open the possiblity that any path can be taken. Only if all paths in the program satisfy the property, we generate optimized code. Even if there is one branch in the program that does not satisfy, then we generate the generalized default implementation. Therefore our abstraction though not very precise, is safe.

7 Building the Activity Automata

```
1. c = clock.factory.clock()
2. c.next()
   if (n > 1)
3.   c.resume()
   else
4.   c. next();
5. c. next();
6. c.drop();
```

Fig. 2. Example of an activity in X10

Consider the snippet of code in Figure **??**. The activity calls *next()* first on *c*, followed by *resume()* or *next()* depending on the value of *n*. The number on the left denote line numbers.

We use *NuSMV*, a BDD and SAT based model checking tool. Modeling in *NuSMV* involves providing a model and list of properties to check. To provide a model, we translate the call graph of the activity to *NUSMV*. The translation is straightforward.

We create a finite state automaton in NuSMV as shown in Figure **??**, the clock is initially in the register state referring to the statement where it was created. NuSMV uses case statements to model transitions. For e.g.,from the register instruction, the activity calls *next* at line 2. From line 2, it can make

```
init (clock) = register;
next(clock) :=
case
(clock = register) : next_2;
(clock = next_2) : {resume_3, next_4};
(clock = resume_3) : next_5;
(clock = next_4) : next_5;
(clock = next_5) : drop_6;
1: clock;
esac;
DEFINE clock_next = clock in {next_2, next_4, next_5}
DEFINE clock_resume = clock in {resume_3}
..
```

Fig. 3. Modeling in *NuSMV*

non-deterministic choices to go to either *resume* at line 3 or *next* at line 4. The last case statement is the default condition, where it stays in the same place.

Also we provide *DEFINE* statements. For e.g,*clock_next* refers to all statements in the activity that call *next* on that clock.

We build this automaton for every clock in the program, independently of other clocks. We provide one sub-automaton for every function body in the program. Whenever there is a function call, there is a jump to the entry of corresponding sub-automaton of the function and back to the caller automaton from the function's exit.

8 Building the State Automaton

We also need to keep track of the state of the activity with respect to a particular clock. Figure **??**. Initially, before the clock is defined, the activity is in the *inactive* state. Now looking at the case statements, when the activity registers on the clock, the state changes from *inactive* to *active*. When the state is *active* and and the activity drops the clock, the state changes to *inactive*. When the state is *active*, and the activity calls *resume*, the state changes to *resumed*. From the *resumed* state, it moves to active, when the activity calls*next*.

The clock protocol does not allow the an activity to call resume twice consecutively. Suppose, the activity is in the *resumed* state, but calls resume, then it moves to the *resume_exception* state. Similarly, if an activity tries to call *next* on a clock after dropping the clock, then it moves to the *next_exception* state.

We build a separate state automaton for every clock in the program independently of each other.

```
init (state) = inactive;
next(state) :=
case
(state = inactive) & (clock_register) : active;
(state = active) & (clock_drop) : inactive;
(state = active) & (clock_resume): resumed;
(state = resumed) & (clock_next): active;
..
-- Exception cases
(state = resumed) & (clock_resume):
resume_exception;
(state = inactive) & (clock_next) : next_exception;
```

Fig. 4. State Automaton in NuSMV

9 Checking For Properties

We need to provide NuSMV with the properties or specifications that we need to check.

Suppose we want to check if a set of clock operations do not violate the clock protocol is exception free. We combine all exceptions such as *drop_exception*, *next_exception* etc into one *state_exception*:

```
DEFINE state_exception  = state in {drop_exception, next_exception, .}
```

Specifications can be expressed in Temporal Logic in NuSMV.

```
SPEC AG(~(state_exception))
```

The above specification checks for the freedom of exceptions. If all paths in the program, globally, are exception free, then the above expression returns true.

Similarly we could check for the absence of *resume* as follows:

```
SPEC AG(~(clock_resume))
```

10 Combining with Aliasing Analysis

Figure **??** shows an example of aliasing of Clocks in X10. Two clocks *c1* and *c2* are created. x can be either *c1* or *c2* depending on the value of *n*.

We can abstract the above snippet of code to Figure **??**. The problem with this kind of abstraction is that it increases the complexity exponentially with nested clocks. Applications that follow this behavior include data-flow algorithms.

We therefore reduce the complexity with some loss of precision by abstracting it in a different way as shown in Figure **??**. We branch each clock instruction independently of the other therefore making the expansion polynomial. By doing

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

Fig. 5. Aliasing of Clocks in X10

```
final clock c1 = clock.factory.clock();
final clock c2 = clock.factory.clock();
..
if(*) {
  c.resume();
  c.next();
} else {
  d.resume();
  d.next();
}
```

Fig. 6. Abstracting aliases

this, we are only adding extra paths in the program, still making our tool safe, because of over-approximation of the system.

Also, *wala*'s pointer analysis gives an over approximation of possible values for every variable used in each instruction, therefore giving us the feasibility to easily adopt the second abstraction.

11 Results

We ran our static analyzer on various X10 programs on a 3GHz Pentium 4 machine with 1GB RAM.

12 The Code Optimizer

The result of the static analyzer is fed as as input to the code optimizer. The code optimizer has a plugin that switches between different implementations based on the input. We provide a different clock implementation for each pattern of clock. Example of pattern based implementations include protocol-violation-free implementation, no resume implementation etc. . The code optimizer uses line and column number of definition of clock in the source code provided by the static analyzer to switch between implementations.

```
final clock c1 = clock.factory.clock();
final clock c2 = clock.factory.clock();
..
if(*) {
  c.resume();
} else {
  c.next();
}
if(*) {
  d.resume();
} else {
  d.next();
}
```

Fig. 7. Abstracting aliases in a less complex manner

Example	Lines of Code	No. of Clocks	Result	Analysis Time
Game of Life Algorithm	55	1	EF,NR	5.8
All Reduction Barrier	65	1	EF,NR	21.4
Sequence Alignment (Edmiston)	205	2	Clock 1: EF,NR Clock 2: EF,NR	20.0
LU Factorization	210	1	EF,NR	9.9
Java Grande Benchmark Suite	930	1	EF,NR	27.6

Table 1. Experimental Results. EF: Exception Free, NR: No Resume

13 Conclusion and Future Work

We have presented a static analysis technique for clocks in the X10 programming language.

References