Internal Report 2011–01 May 2011

Universiteit Leiden
Opleiding Informatica

Deploying active objects

onto

multicore

Behrooz Nobakht

MASTER'S THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Contents

Abstract

The performance of a program on multicore platform crucially depends on the scheduling of its tasks; existing high-level programming languages, however, offer limited control over scheduling. In this thesis, we develop Cacoj as an extensible tool set to transform Creol's active concurrent objects into Java to be deployed on multicore through standard Java Runtime Environment. The concurrent object paradigm is a promising trend for multicore programming because each object may conceptually encapsulate a processor. Cacoj introduces a higher-level abstraction of concurrency API and a Creol compiler in which the translated object in Java takes control over the scheduling of the incoming messages through a per-object approach in contrast with current mainstream trend. Cacoj brings about the required grounds to extend Creol syntax to additionally specify different levels of priority and integrate them into the notion of active concurrent objects.

Acknowledgements

Every work is successful with the effort of all invovled directly or indirectly in the path. I am wordlessly grateful to my parents and my sister and brother who constantly fed me with remote yet deep support and care.

I worked on this thesis for about a year under the supervision of Professor Frank S. de Boer. His support for my interests along with his constructive guidance through the research was the key for my susccess. I deeply thank him and wish him all the best in life for him.

Chapter 1

Introduction

1.1 The Problem

One of the major challenges of multicore programming in object-oriented languages is how to help the programmer optimally use potentially many cores. At the operating system level, efficiency is greatly affected by scheduling which is largely beyond the control of most existing high-level programming languages. Therefore, for optimal use of the hardware, we cannot avoid leveraging scheduling and performance related issues from the underlying operating system to the application level. However, the very nature of high-level languages is to provide suitable abstractions that allow the programmer to be free of irrelevant implementation details. The main challenge in designing programming languages for multicore processors is to find a balance between these two conflicting requirements.

Looking at a mainstream language such as Java, it introduces concurrency API in package `java.util.concurrent` to be used in multicore programming. However, the approach taken in this library does not seem to be in line with high-level concepts in object orientation; i.e. as an orthogonal supplement to other features of the language, the programmer has to get involved in the implementation details how to leverage the library in the domain of multicore programming. In contrast, languages such as Erlang [22] or Scala [49] take further steps to introduce, for multicore, interesting abstractions such as *actors*. In Scala [29], actors are scheduled using a scheduling manager that takes advantage of a thread pool of worker threads. Additionally, Scala introduces both *thread-based* actors and *event-based* actors. However, similar to Java, Scala uses a system-wide scheduling mechanism. Objects themselves do *not* control the scheduling policies used to select and execute the messages they receive; i.e. the priority management and the scheduling of the object's messages are *centrally* performed *outside* the objects.

In this thesis, we investigate how active concurrent objects in a high-level object-oriented language can be deployed onto multicore technology. We use

the notion of concurrent objects in Creol [35, 21]. An active concurrent object in Creol has control over one processor; i.e. it has control over a thread pool of execution of asynchronous messages. Thus, a concurrent object provides a natural basis for multicore deployment where each object ideally possesses one processor. The object itself manages the behavior in which messages are scheduled for execution. We develop Cacoj as an extensible tool set to translate active objects in Creol into their equivalent active objects in Java. To achieve the deployment on multicore, we need to think differently how to use the Java concurrency library to provide a higher-level abstraction for per-active-object scheduling control over the concurrency of messages.

The Cacoj tool set breaks down into two major components: an active object API based on Java's concurrency API and a compiler tool. The active object API is a generic architecture to be developed and leveraged to deploy Java objects onto multicore using a high level abstraction to allow each object take control over the scheduling of the asynchronous/synchronous messages for execution. The compiler component takes advantage of Cacoj API to translate Creol's objects into equivalent Java objects. Every active object in Creol is transformed to an object in Java that possesses a pool of execution threads on a separate processor. It may use a priority manager and scheduler to respond to the incoming messages from other objects. Besides, through this transformation, we allow the programmer to seamlessly use, in the original Creol program, Java's standard libraries including the data types. Thus, this moves Creol from just a modeling language to a full-fledged "programming" language. Through Cacoj, the programmer is provided with a higher-level abstraction as in Creol to model concurrency and then it is transformed to a mainstream language such as Java to deployed onto multicore.

1.2 Cacoj

In this thesis, we propose and develop Cacoj to deploy concurrent active objects onto multicore using Java platform. Essentially, Cacoj is designed in two major components:

1. Concurrency API is a generic architecture that can be used to deploy active objects onto multicore. It provides per-object priority and scheduling featuers and the support of asynchronous messages and future computation. We discuss the notation of thread binding to messages in this thesis and how they are used to realize the active object pattern. Additionally, we introduce an event-based mechanism to control how the processor is used and released.

2. Creol to Java compiler that provides a base ground to compiler Creol programs into concurrent Java taking advantage of the concurrency API in Cacoj. The concurrency API provides the grounds for a modeling language such as Creol to abstract away the implementation details of the concurrency API from the user. Moreover, in future, the extensions to

the Creol language will introduce syntax for priority scheduling that are implemented in Cacoj Concurrent API.

Cacoj project is freely available at `http://www.sourceforge.net/projects/cacoj`. All the user documentation and details on how to use and develop Cacoj can be found at the project page. In this thesis, we discuss the main ideas behind the design of Cacoj.

1.3 Organization of the Thesis

In this thesis, we introduced the general problem in Chapter 1. We elaborate on how Cacoj is designed and implemented in Chapter 2. We discuss two samples, Prime Sieve and Fibonacci, in Chapter 3. We provide an overview of the extensions planned for Cacoj in Chapter 4. We present an overview of related work in Chapter 5. Eventually, we conclude the thesis in Chapter 6.

Chapter 2

Cacoj

2.1 Overview

2.1.1 Creol

Creol [35] (Concurrent Reflective Object-oriented Language) is an object oriented modeling language designed for specifying distributed systems. A Creol object implicitly has a dedicated processor and thus encapsulates a line of execution. Different objects communicate only by asynchronous method calls, i.e., similar to message passing in Actor models [20]; however in Creol, the caller can poll or wait for return values or termination of the called method. This can be used to simulate synchronous method calls.

Creol objects are typed by interfaces, whereas classes can implement as many interfaces as necessary. An interface may restrict the type of objects that can call each of its methods via the notion of co-interfaces; thus knowing the type of the caller, a method implementation can safely call back. A co-interface is specified by the keyword `with`; Line 3 in Listing 1 uses `Any` to allow all types as co-interface. Creol supports multiple inheritance; inheritance among interfaces can be used to create sub-type hierarchies, whereas inheritance between classes is only used for code reuse.

Each object in Creol, upon creation, starts its active behavior by executing its `run` method if defined (e.g., in class `User`). When receiving a method call a new process is created to execute the method. Methods can have processor release points which define interleaving points explicitly. When a process is executing, it is not interrupted until it finishes or reaches a release point. Release points can be conditional, e.g. `await t?(res)` in Listing 1 checks for termination of `t!fib(n)`. If the guard at a release point evaluates to true, the process keeps the control, otherwise, it releases the processor and becomes disabled as long as the guard is not true. Whenever the processor is free, an enabled process is *nondeterministically* selected for execution, i.e., scheduling is left unspecified in Creol to enable more abstract modeling. In a future work, we extend Creol to support mechanisms for priority specifications of method calls in different levels.

Listing 1: Fibonacci in Creol

```
1  interface IFibonacci
2  begin
3    with Any
4      op fibonacci(in n: Int; out f: Int)
5  end
6
7  class Fibonacci(n:Int) implements IFibonacci
8  begin
9    op init == skip
10
11   op run ==
12     var res: Int; var t: Label[Int]; t!fib(n); await t?; t?(res)
13
14   op fib(in n:Int; out f:Int) ==
15     var a:Int; var b:Int; var s: Label[Int]; var t:Label[Int];
16     if n = 0 then
17       f := 1
18     else
19       if n = 1 then
20         f := 1
21       else
22         s!fib(n - 1); t!fib(n - 2); s?(a); t?(b); f := a + b
23       end
24     end
25
26 with Any
27   op fibonacci(in n:Int; out f:Int) ==
28     var t:Label[Int]; t!fib(n); t?(f)
29
30 end
```

2.1.2 Concurrent Java

Inside the classical object-oriented paradigm, Java proposes to think in terms of "tasks" in "objects". To utilize multicore, the programmer should spread the workload in different objects in terms of tasks. JVM then knows how to map different tasks to different processing cores. The concept of task is in analogy with the concept of `java.lang.Runnable` in multi-threading paradigm. Conceptually, the granularity of the task is dependent on the requirements and properties of the program.

Simply put, a task object is an implementation of `Callable`[1] to provide the `call` method returning an instance of `Future`. The `call` method to accomplish its job may use a set of Java statements, call other methods in the same object, or even invoke methods from as many objects required.

Concurrent Java introduces "thread pool": a repository of task objects (objects of type `Callable`) in the memory at runtime to store and schedule for execution on different processors if available. Along with thread pool, Java

[1] `java.util.concurrent.Callable`

introduces a central `ExecutorService`[2] interface that is used to manage, run or schedule different task objects. Based on the requirements of the tasks in the program, there may be the need to have different types of implementations of `ExecutorService`. For instance, a `ThreadPoolExecutor`[3] is an implementation that tries to run and spread the tasks on a queue-based thread pool to be executed in different cores.

2.2 Concurrency API

Each Creol class is translated to a corresponding Java class and each method is also translated to the equivalent method in Java that exposes the same behavior as the concurrent active object in Creol. To achieve this pattern in Java, we developed a concurrency API package based on `java.util.concurrent` package to enable a Java class expose asynchronous and active behavior properties. In this design, each active object is composed on the following components (Figure 2.1).

`MethodInvocation` is the central data structure that encapsulates an asynchronous method call. It holds a reference to the caller object. In future works, it will carry the priority information if provided by the caller. It is a subclass of `FutureTask`[4]. It holds the eventual result of the method call. It can be `await`ed or `signal`ed based on the program requirements during runtime; thus, it is aware of the corresponding "thread" in which it is running. When an instance is created by the caller, there are two options how to store the it in the callee's process store:

1. The caller calls a method from the callee to store the instance. This approach may have concurrency issues. When the method invocation is to be stored at the callee's process store, the callee may be busy doing something else; the caller should wait until the callee is free to store the instance of method invocation in its process store.

2. The callee directly stores the method invocation into the process store of the callee. Each active object exposes its process store as an immutable public property such that the clients of the active object may directly store the messages. Thus, the actual code executed to put the method invocation instance in the callee's process store is run in the execution thread of the caller. We adopt this approach.

`ProcessStore` is the storage data structure used to maintain the incoming method call invocations to an active object. The `ProcessStore` is an implementation of the `BlockingQueue` interface in `java.util.concurrent` package, the implementations of which are thread-safe, i.e. all methods in this interface operate atomically using internal locks encapsulating the implementation details for the user. It holds a reference to an instance of `PriorityManager` and `SchedulingManager`

[2] `java.util.concurrent.ExecutorService`
[3] `java.util.concurrent.ThreadPoolExecutor`
[4] `java.util.concurrent.FutureTask`

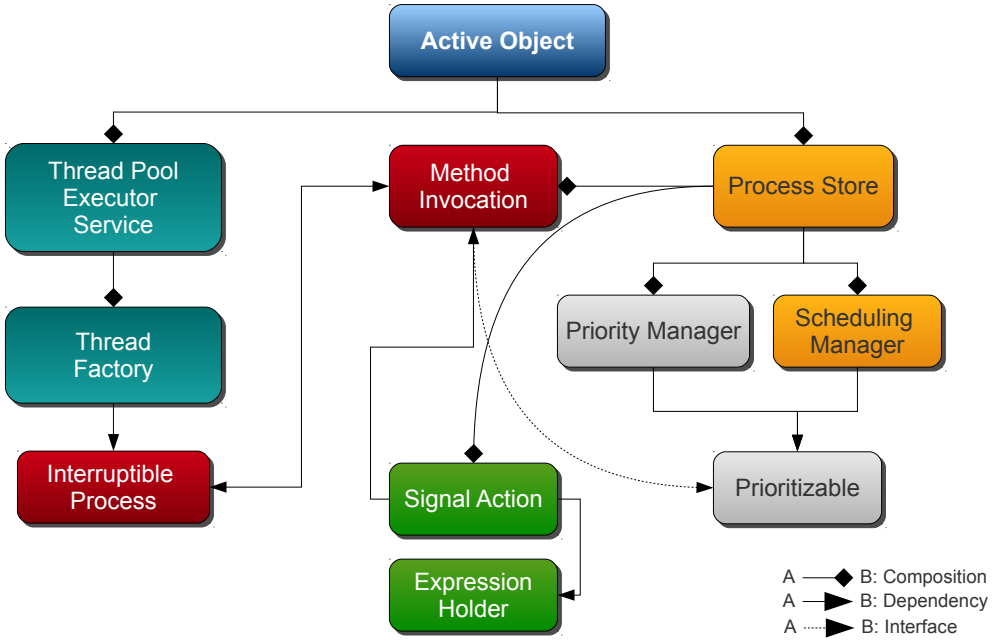

Figure 2.1: Cacoj Concurrency API Architecture

if specified (see below). It provides methods such as `preAdd` and `postAdd` along with `preTake` and `postTake` respectively to enable the further customization of the behavior before/after adding or taking a method invocation to/from the storage. These are extension points to enable the customization of priority or scheduling management of the method invocations. The `PriorityManager` component is a future extension to this work.

`PriorityManager` is an interface the implementations of which are supposed to provide a function to determine and resolve a final priority value in case there are different levels of priorities specified for a method invocation. This feature is used, if specified, by the process store to compute the "resolved priority" of the method invocation based on different levels of priorities specified and then add it to the storage. `PriorityManager` is a planned future extension of this work in line with the Creol language extensions.

`SchedulingManager` is an interface the implementations of which introduce a function to *select* a method invocation based on different possible criteria (such as time or data) that is either predefined or customized by the user. The scheduler manager is a component used by process store when asked to

remove and provide an instance of method invocation to be executed. Thus, the implementation of the scheduling manager is responsible how to choose one method invocation out of the ones currently stored in the process store of the active object. Different flavors of the scheduling manager may include time-based, deadline-based, data-centric, or a mixture.

`ExecutorService` is an implementation of `ExecutorService`[5] that is responsible for the execution of method invocations. Each active object holds one reference of one executor service instance. We develop an executor service to take advantage of a "thread pool" implementation; thus, each active object submits the selected method invocation by the scheduling manager to its executor service to be executed. The executor service takes advantage of the a thread factory to maintain a pool of threads to run the requested code of method invocations. The thread pool may use caching or other optimization techniques to minimize resource allocation and usage costs.

`ThreadFactory` is an interface used by the executor service to initiate a new thread when new resources are required. We cache the threads so that we can control and tune the performance of resource allocation. The thread factory create a new instance of an "interruptible process" upon its factory call.

`InterruptibleProcess` is an extension of Java `Thread`[6] class. A method invocation is assigned a Java thread for execution. In the original scenario in Java concurrency API, when a method is asked to `await`, it waits on the condition but actually *releases* the thread in JVM. This means that some of local information attached to the method execution and the thread are lost since the thread may be used for some other execution request. Thus, we need to have a version of thread that is aware of its running method invocation and vice versa. This relation helps us to preserve the thread through the time of awaiting so that the method invocation can continue just from the awaiting state. To achieve this, when the executor service is to delegate the execution to the method invocation, it notifies both the method invocation and the thread of each other's presence so that this relation will be preserved through the time till the method invocation is signalled to continue the execution.

`SignalAction` is an *event-based* mechanism to control how the method invocations are signalled after awaiting. When a method invocation is preparing to await, first, it adds an instance of a signal action to the active object's process store. The signal action gets selected sometime in future and its job it to check whether the original method invocation may continue execution based on some "conditional expression" provided by `ExpressionHolder`. If it can, it signals the original method invocation. Otherwise, it stores another copy of itself to the active object's process store as another future method invocation. The mechanism is event-based and non-blocking having the advantage that it is also transparent from the view of the scheduling manager and priority manager.

One of the significant ideas in the concurrency designed in Cacoj API is the relation amongst `MethodInvocation`, `InterruptibleProcess`, and `SignalAction`.

[5] `java.util.concurrent.ExecutorService`
[6] `java.lang.Thread`

From the user's viewpoint, an instance of a method invocation is *interruptible*; i.e. `MethodInvocation` exposes `await` and `signal` through `Interruptible` interface. Actually, however, these methods are delegated to the corresponding `InterruptibleProcess`. Besides, `InterruptibleProcess` takes advantage of `ReentrantLock` [7] and `Condition` [8] to expose the "await" and "signal" features (Listing 2).

Listing 2: InterruptibleProcess await and signal methods

```
1  public boolean await() throws RuntimeException {
2    try {
3      awaitLock.lock();
4      while (this.mi.isSuspended())
5        blockedCondition.await();
6    } catch (InterruptedException e) {
7      interrupt();
8    } finally {
9      awaitLock.unlock();
10   }
11   return false;
12 }
13
14 public boolean signal() throws RuntimeException {
15   try {
16     awaitLock.lock();
17     blockedCondition.signalAll();
18   } finally {
19     awaitLock.unlock();
20   }
21   return false;
22 }
```

In Listing 2, `mi` refers to the enclosing instance of `MethodInvocation`; thus, the interruptible process awaits as long as the corresponding method invocation is suspended. Other than that, it signals the method invocation to continue from the awaiting point of the execution.

In the same line, `SignalAction` takes an event-based approach to enable the pair of `MethodInvocation` and `InterruptibleProcess` act in a *non-blocking* manner. `SignalAction` holds a reference to the original `MethodInvocation` and an instance of `ExpressionHolder` the second of which is used to check whether the original method invocation is ready to continue the execution. If not, a copy of the current signal action is again stored in the process store of the corresponding active object for future checking. And, if the method invocation is ready to continue, the signal action "signals" it so that it can continue execution. This procedure is done in a *non-blocking* and *event-based* way. Listing 3 presents the code in which `executor` is the enclosing active object and `copy()` creates a copy instance of the same signal action instance.

Listing 3: SignalAction call method

```
1  public Object call() throws Exception {
```

[7] `java.util.concurrent.locks.ReentrantLock`
[8] `java.util.concurrent.locks.Condition`

```
2   if (this.expressionHolder.getValue()) {
3     this.originalMI.signal();
4   } else {
5     this.executor.getProcessStore().add(copy());
6   }
7   return null;
8 }
```

We now discuss how in overall an active object operates on a multicore architecture. We take a "client/server" approach here since each active object essentially behaves as a server to all the other objects as its clients that request a method to be executed.

In the client's viewpoint, a method call is needed to be published to the server but in an asynchronous way. First, the client needs to construct an instance of method invocation that wraps around the original method call from the server. Then, the client needs to directly fetch the `ProcessStore` of the server and `add`s the instance of the method invocation to the server. The method or policy used to store the method invocation in the process store of the server is totally up to the server's process store implementation details. Storing a method invocation in the process store of the server takes place in the client side thread of execution. Listing 6 presents a sample generated Java code for an asynchronous method call in Listing 1. Thus, through time, there are concurrent clients that are storing instances of method invocation into the server's process store and since the process store implementation encapsulate the mechanisms for concurrency and data safety, the clients have no concern on data synchronization and concurrency issues such as mutual exclusion.

In a parallel side of the story, when an active object comes to life, it constantly tries to fetch an instance of method invocation from its process store for execution. The process store uses its instance of `SchedulingManager` to choose one of the method invocations. Although, there are predefined scheduling managers in Cacoj, it can be easily developed and customized based on the requirements by the user. Listing 4 presents the main loop for the active object. Each `ActiveObject` also implements `Runnable` [9]. So, when an active object gets initialized, it simply adds `this` (itself) to its executor service component. This procedure *deploys* the active object onto the executor service corresponding to the underlying multicore platform.

Listing 4: Main run loop of an ActiveObject

```
1 public void init() {
2   this.es.submit(this);
3 }
4
5 public void run() {
6   for (;;) {
7     try {
8       MethodInvocation mi = this.ps.take();
9       this.currentProcess = mi;
```

[9] `java.lang.Runnable`

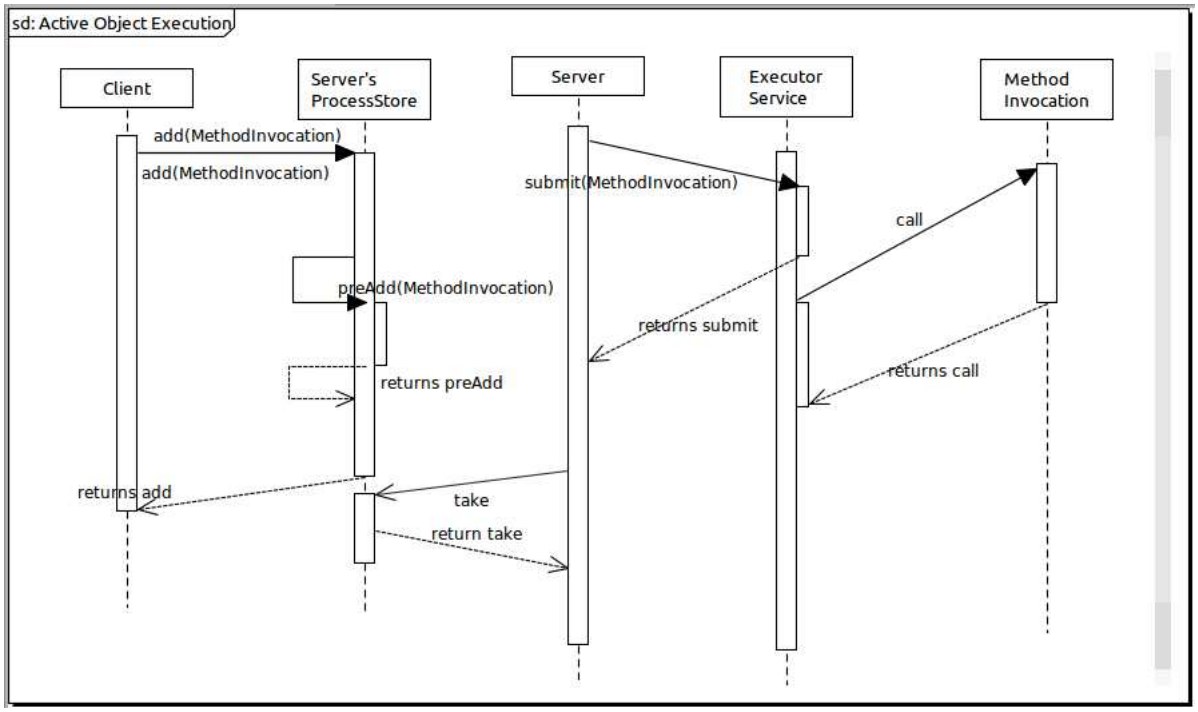

Figure 2.2: Sequence of Executions in Cacoj API

```
10        this.es.submit(mi);
11      } catch (InterruptedException e) {
12        e.printStackTrace();
13      }
14    }
15 }
```

The sequence of events in this API is depicted in Figure 2.2. It should be *noted* that the two parallel lines of executions are presented in the same diagram just to present the ideas but as it was described client and server run independently.

In this section, we elaborated on how the concurrency API in Cacoj is designed and developed. In the following sections, we describe how this API is entangled into Cacoj compiler to translate Creol programs into their corresponding Java code.

2.3 Cacoj Compiler

In this section, we first present how different constructs are translated to their equivalent Java ones. Then, we describe the overall compilation process in

Cacoj.

2.3.1 From Creol To Java

Each Creol class is translated to a corresponding Java class. The Java class contains a public immutable field `ps` of type `ProcessStore` to store the method invocations from the clients. The Java class also owns a private field `es` of type `ExecutorService`; the component in control of the thread pool of interruptible processes. Thus, generally, a class `Fibonacci` in Listing 1 will be translated into Listing 5 in Java:

Listing 5: General structure of the translated Creol Fibonacci into Java

```
1 class Fibonacci implements IFibonacci, ActiveObject {
2   private final AbstractProcessStore ps = new SimpleProcessStore
        ();
3   private final ExecutorService es = new
        InterruptibleProcessPoolExecutor(
4       new InterruptibleProcessFactory());
5   private MethodInvocation currentProcess;
6
7   // translations in between
8
9   public AbstractProcessStore getProcessStore() {
10    return ps;
11  }
12
13 }
```

Each method in the Creol class is translated to one method in Java. Each method invocation in Creol, on the other hand, will involve the construction of an instance of an anonymous implementation of the interface `Callable`. For instance, the method `fibonacci (n; f)` along with its method calls in Listing 1 is translated to the following Java in Listing 6:

Listing 6: Translated fibonacci method to Java

```
1 public Integer fibonacci(final Integer n) {
2   Integer f;
3   final Future<Integer> t;
4   MethodInvocation mi4 = new MethodInvocation(new Callable() {
5     public Integer call() {
6       return fib(n);
7     }
8   }, this);
9   getProcessStore().add(mi4);
10  t = mi4;
11  try {
12    f = t.get();
13  } catch (InterruptedException e) {
14    throw new RuntimeException(e);
15  } catch (ExecutionException e) {
16    throw new RuntimeException(e);
17  }
```

```
18   return f;
19 }
```

The variable `mi4`, as an instance of `MethodInvocation`, wraps the original method invocation and is added to the process store of the callee object (Line 9 using the `add` method in `getProcessStore()`). Note that `mi4` provides a `Future` return value that can be used to fetch the eventual result of the original method invocation. The corresponding redefinition of `call` method provides the actual trigger for the execution of `fib(n)` method (Line 6). As it can be seen in this sample, the concept of a `Label` in Creol in translated to an instance of a `Future` in Java. The `Future` in Java has the property that when `get()` (Line 12) is called, it releases the processor until the value is ready or the process is cancelled. Thus, the encapsulation corresponds to the same in Creol with `Label`.

In this regards, it is interesting to see how `await` in Creol is translated to Java using Cacoj compiler. The `init` method in Listing 1 makes a perfect sample to elaborate on it. In this method, there is a need to wait for the result of Fibonacci computation and then return the final result of this call. However, until the intermediate results are ready, the process should release and wait. The corresponding translated code in Java is shown in Listing 7.

Listing 7: await in Creol and SignalAction in Cacoj

```
1  Integer res;
2  final Future<Integer> t;
3  MethodInvocation mi1 = new MethodInvocation(new Callable() {
4    public Integer call() {
5      return fib(n);
6    }
7  }, this);
8  ps.add(mi1);
9  t = mi1;
10 while (!(t.isDone())) {
11   addSignalActionNotifier(currentProcess, new ExpressionHolder()
         {
         public Boolean getValue() {
12        return (t.isDone());
13     }
14   });
15   ((Interruptible) currentProcess).await();
16 }
17 try {
18   res = t.get();
19 } catch (InterruptedException e) {
20   throw new RuntimeException(e);
21 } catch (ExecutionException e) {
22   throw new RuntimeException(e);
23 }
```

In Line 11, an instance of `SignalAction` is created and stored in the process store for future execution. The expression that this instance should check is that whether the `Future` value for `t` is ready or not. Thus, whenever the signal action gets a chance to execute, it checks whether the final result is ready or not. If

not, it again queues a copy of itself to the process store. Otherwise, the result is ready and the method call finishes. *Note* that in Line 10, `while` is a not a busy wait loop. When the processor is given back, it tries to check the condition and if it is not satisfied it should again release the processor. Additionally, `currentProcess` is always the *current* method invocation that is being processed. On the other hand, the method `addSignalActionNotifier` is a template method that is included for every active object during translation (Listing 8).

Listing 8: addSignalActionNotifier template method

```
1 private void addSignalActionNotifier(MethodInvocation mi,
      ExpressionHolder expressionHolder) {
2   SignalAction sa = new SignalAction(mi, expressionHolder, this)
        ;
3   MethodInvocation saMI = new MethodInvocation(sa, mi);
4   this.ps.add(saMI);
5 }
```

In this section, we went through different samples of how language constructs in Creol are translated into their equivalent Java ones. In the next section, we introduce the general compilation process in Cacoj compiler.

2.3.2 Compilation Process

The Cacoj compiler is actually an aggregator and integrator of different components in the compilation process of Creol into Java. The compiler first uses ANTLR v3 [42] to generate the language recognizer. Then, it uses a transformation engine and a set of code templates to generate the equivalent target program such as Java based on the concurrency API:

1. The Creol language is expressed in a grammar in ANTLR. This is used to generate a standard ANTLR lexer and parser for the Creol language.

2. An instance of an abstract syntax tree (AST) is constructed when a Creol program is input to the Creol Parser.

3. The AST object is preprocessed to construct a set of intermediate language information, *compilation context*, that is used in the time of code generation. This phase is line with similar concepts in LLVM. This phase also introduces future extension points as it is not part of the standard ANTLR framework.

4. A tree grammar is developed for the Creol language used to walk through the generated AST tree of the Creol source and generate the equivalent Java code. A Creol tree walker is generated that walks through the generated AST and generates the final code.

5. Through code generation, the Creol walker takes advantage of the compilation context that is built before to fetch the required information including meta method information and class properties.

6. For code generation, a set of code template are developed in ANTLR framework using StringTemplate[43] library. Essentially, each construct in Creol has an equivalent code template snippet (as discussed in previous section) that given the required information outputs the equivalent Java construct. The Creol walker uses the set of code templates along with the compilation context to generate the final Java code.

The compilation process partially includes a set of standard routines for language compilation and transformation. However, the Creol compiler basically takes advantage of two loosely-coupled components in the process: the concurrency API and the code templates. These two make Cacoj an extensible and pluggable compiler tool for the Creol language. In the first place, the concurrency API designed for Creol in Java includes the fundamental data structures and services that are required to deploy an active object in Java using the utitilies provided in `java.util.concurrent` package. This is a major extension point for further development since the developer may introduce different flavors of *method invocation storage*, *method invocation priority assignment*, and *method invocation scheduling management*. These are the building blocks of the actual runtime environment based on which a translated Creol program into Java takes advantage of to be deployed onto multicore in JVM platform. Additionally, the set of code templates are actually maintained out of Creol compiler code. Thus, it provides another extension point for the users so that they can customize or develop different code templates based on their requirements in the compilation process. The separation of concerns in the design of Cacoj makes it an extensible and pluggable tool set.

Chapter 3

Case Studies

3.1 Prime Sieve

A prime sieve [51] or prime number sieve is a fast type of algorithm for finding primes. A prime sieve works by creating a list of all integers up to a desired limit and progressively removing composite numbers until only primes are left. One sample method to implement a prime sieve is "Sieve of Eratosthenes" [52].

To generate the prime numbers, we use the Creol code in Listing 9.

Listing 9: Prime Sieve in Creol

```
interface ISieve
begin
with Any
  op send(in x: Int; out r: Bool)
end

class Generator
begin
  var n: Int
  var s: Sieve
  op init ==
    gen := new Sieve(2);
    n := 3
  op run ==
    var r: Bool;
    !s.send(n; r);
    if n < 1000 then
      n := n + 1;
      !run()
    else
      s.printl(;)
    end
end

class Sieve(p: Int) implements ISieve
begin
  var next: Sieve
```

```
28
29   with Any
30     op send(in n: Int; out res: Boolean) ==
31       var d: Int := n / p;
32       var r: Int := n - d * p;
33       res := false;
34       if r = 0 then
35         res := false
36       else
37     if next /= null then
38       !next.send(n; res)
39         else
40           res := true;
41           next := new Sieve(n)
42         end
43       end
44
45       op printl ==
46         println(p + " ");
47         if next /= null then
48           next.printl(;)
49         end
50 end
51
52 class Test
53 begin
54   var g : Generator
55   op run ==
56     g := new Generator
57 end
```

In Listing 9, we generate a linked list of primes each of which holds a reference to the next prime number. In each step through generating the primes, in each `Sieve` object:

1. The field p is a prime number.
2. Parameter n in `send` method is received to test against the next prime number.
3. If $r = 0$ so n is *not* a prime.
4. If $r \neq 0$ and `next == null`, it means that we have found a new prime number that is the "next" prime number to p. We create the next prime number in the linked list.
5. If $r \neq 0$ and `next != null`, we delegate the computation of n being a prime to the next prime in the chain until either $r = 0$ or n will be a next prime in the chain.

Listing 9 proivdes a concurrent asynchronous approach to generate a sequence of prime numbers using asynchronous method calls such as in `!s.send(n ; r)` or `!next.send(n; r)`. At each step of the computation through time:

1. The object `g` may be composing a new message for the next n sent to object `s`

2. The object `s` is continuously receiving a messages with known n to be tested for a prime number

Considering these two criteria, the object `s` is faced with a pool of messages containing a random order of n's for the computation. Consequently, there is chance that larger n's may be processed before smaller ones are processed; this is a source of error since it may corrupt the prime number test. Thus, we need to guarantee that the incoming n numbers are processed in an *ascending order* no matter the order in which they are added to the message store of the object `s`.

Using Cacoj, we propose a two-step solution. In the first step, we propose a custom *priority manager* to prioritize the incoming messages to objects of type `Sieve`. A final priority value is assigned to each message (method invocation) being added to the active object's process store. In the next step, we introduce a custom *scheduling manager* to select the next method invocation from the active object's process store based on the previously-assigned priorities by the priority manager. We use the actual parameter n in `send` method for the basis of this prioritization and scheduling of the messages. We guarantee that the next message chosen for processing has the smallest available n among the method invocations in the process store of the active object.

We introduce `PrimeSievePriorityManager` that uses the parameter n to assign a priority value to each method invocation carrying a call to `send` method. Listing 10 demonstrates the code.

Listing 10: Prime Sieve Priority Manager

```
1 public class PrimeSievePriorityManager implements
      PriorityManager {
2   public Priority resolve(MethodInvocation<?> mi) {
3     MethodInvocationMetaData md = mi.getMetaData();
4     final Integer n = (Integer) md.getActualParameters().get(0);
5     return new Priority() {
6       public Number getValue() {
7         return n;
8       }
9     };
10  }
11 }
```

In Cacoj, each `MethodInvocation` carries a set of meta information about the actual method call including the actual parameters of the method call. We use the meta data to assign the priority of the method invocation. Additionally, we introduce `PrimeSieveSchedulingManager` that uses the priority assigned by `PrimeSievePriorityManager` to select the next method invocation instance that has *the smallest* parameter n. Listing 11 depicts the code.

Listing 11: Prime Sieve Priority Manager

```
1 public class PrimeSieveSchedulingManager extends
      AbstractSchedulingManager {
2   public Boolean isPrior(MethodInvocation<?> mi1,
        MethodInvocation<?> mi2) {
3     Integer n1 = (Integer) mi1.getResolvedPriority().getValue();
4     Integer n2 = (Integer) mi2.getResolvedPriority().getValue();
5     boolean result = n1 >= n2;
6     return result;
7   }
8 }
```

In Listing 11, the scheduling manager uses the "resolved" priorities assigned by the priority managet to select the next method invocation that will be processed. Since the implementation uses a minimum-heap, we use $n_1 \geq n_2$ to select the next smaller n. The case `result = true` means that `mi1` is the one to be selected; otherwise `mi2` is selected.

The case of `Sieve(p=2)` is of more interest. As presented in Listing 9 Line 16, the generator `g` is sending consecutive n values to $Sieve_2$. Then, $Sieve_2$ decides to delegate the computation of n being prime to the `next` sieve and so on until the next prime number is found that becomes the `next` sieve in the chain. However, in the case of $p > 2$, the sieve normally receives *one* message to test whether the current n is prime or not; so, the load of messages on the intermediate sieves are much more less. To evaluate the correctness of the prime sieve code generated by Cacoj, we modified the generated the code to create a situation in which $Sieve_2$ is faced with the pool of all messages contataining the test process for all values of n in the desired range. To achieve this way, we added the following manual code in the code generated for `init` method of `Generator`:

Listing 12: Prime Sieve Modification

```
1   private List<MethodInvocation<?>> mis = new ArrayList<
        MethodInvocation<?>>();
2
3   public void preRun() {
4     Boolean r;
5     nn = n.incrementAndGet();
6     final int ln = nn;
7     MethodInvocation mi1 = new MethodInvocation(new Callable() {
8       public Object call() {
9         gen.send(ln);
10        return null;
11      }
12    }, this);
13    mi1.initMetaData("send", null, nn);
14    mis.add(mi1);
15    if (nn < 1000) {
16      // go for the next value of n
17    } else {
18      Collections.shuffle(mis);
19      ((ProcessStoreAware) gen).getProcessStore().addAll(mis);
20      gen.init();
21    }
```

```
22  }
```

In Listing 12, `mis` is the collection of all method invocations that compose a test for some n being prime. All of the method invocation instances are sent altogether to $Sieve_2$ at once the collection up to the desired limit is ready. When `gen.init()` is activated. $Sieve_2$ starts to process all the messages. Thus, the generated sequence of n's from $Sieve_2$ determines the correctness of this approach.

We operated different runs of prime sieve with the modification described above and without it. We captured the output for the two different cases of the runs that is reprensented in Table 3.1:

Using priority and scheduling managers	Without using priority and scheduling manager
executing: 6	executing: 27
(n, p, r, next.p) = (5,2,1,3)	(n, p, r, next.p) = (25,7,4,11)
(n, p, r, next.p) = (5,2,1,3)	(n, p, r, next.p) = (25,7,4,11)
executing: 7	executing: 25
(n, p, r, next.p) = (6,2,0,3)	(n, p, r, next.p) = (23,11,1,13)
(n, p, r, next.p) = (6,2,0,3)	(n, p, r, next.p) = (23,11,1,13)
executing: 8	executing: 23
(n, p, r, next.p) = (7,2,1,3)	(n, p, r, next.p) = (19,15,4,)
(n, p, r, next.p) = (7,2,1,3)	(n, p, r, next.p) = (17,15,2,17)
executing: 9	executing: 16
(n, p, r, next.p) = (8,2,0,3)	(n, p, r, next.p) = (29,2,1,7)
(n, p, r, next.p) = (8,2,0,3)	(n, p, r, next.p) = (29,2,1,7)

Table 3.1: Prime Sieve Generated Sequence

3.2 Lazy Fibonacci

In this section, we study the sample of Fibonacci given in Listing 1. It is a *lazy* version of Fibonacci since to compute the value of $fib(n)$, we do *not* re-use the values computed for $fib(n-1)$ and $fib(n-2)$; we actually recompute them. This implementation of Fibonacci in a concurrent asychronous manner represents a case in which:

1. Since each method invocation is bound to a thread creation for processing, a large number of thread are created is in direct correlation with resource allocation such as memory.

2. Each computation $fib(k)$ should release the processor and wait for the computatio of $fib(k-1)$ and $fib(k-2)$. As there are a set of active

objects computing their own values, the case represents a good sample for synchronization of messages.

The hardware specification we used for this sample is as follow:

- A Centrino Duo Geniune Intel(R) CPU: 2 × T2500 @ 2.00GHz
- 2GB of Main Memory
- Java memory options for the heap: -XX:MaxPermSize=128m

We managed to compute the values up to $n = 20$ with the hardware. Figure 3.1 depicts how the number of threads generated for each method invocation computation rises in proportion to n.

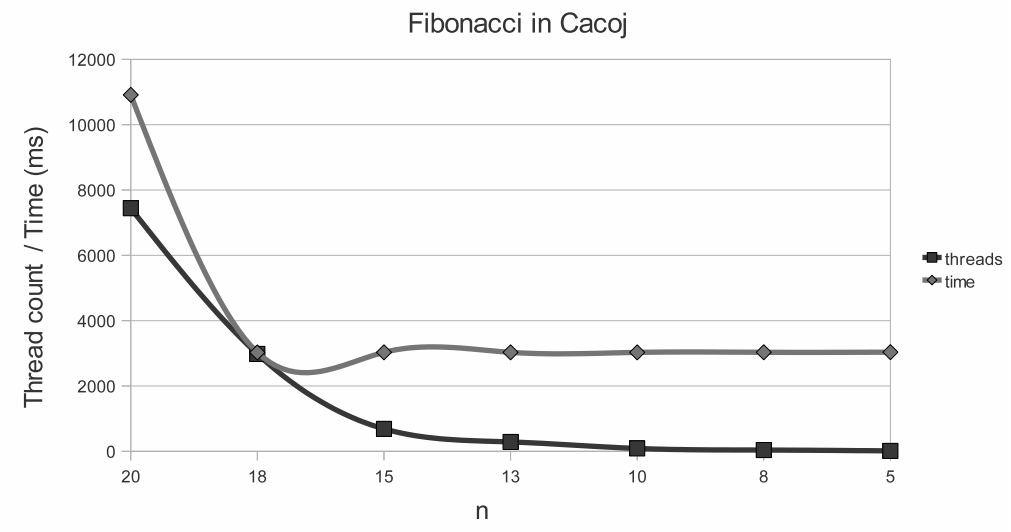

Figure 3.1: Thread Generation and Time with Fibonacci in Cacoj

Chapter 4

Extension Points

4.1 Prioritized Concurrent Objects

In this section, we first discuss how to extend Creol to provide different levels of priority scheduling by introducing high-level abstract constructs that can be used to constrain the nondeterministic selection of a method in an object for execution. Then, we briefly touch the other issue which is how to select one object for the purpose of resource allocation and load balancing.

4.1.1 Priority Scheduling

In this section, we investigate an extension to the Creol language that allows "priority scheduling" of messages in active concurrent objects. In this context, *scheduling* refers to the process of how to select a message in an object for execution based on a notion of priority. *Priority* is defined as a range of natural numbers to denote the importance of the scheduling requirement. In compliance with client/server model, we view each *caller* object as the client for the *callee* object as the server. Clients send messages that are queued in the server. The server is responsible to select a message at a time based on how different messages are prioritized for execution. We provide priority scheduling of method invocations through extensions to Creol language to express different levels of priorities in client and server objects.

Please note that the syntax used in this thesis is *exemplary* and is aimed to present the ideas. Throughout this section, we take advantage of the sample presented in Listing 13: an exclusive resource; i.e. a resource that can be exclusively allocated to one object at a time to behave as a mutual exclusion token.

On the client side, method calls may be naturally extended with priorities that are provided in the server interface as a range:

```
@priority(7) o !  request;
```

Listing 13: Exclusive Resource in Creol

```
 1 interface Resource begin
 2
 3   with User
 4     op request ()
 5     op release()
 6   with StatusPoller
 7     op isFree(; out isfree)
 8 end
 9
10 class ExclusiveResource implements Resource
11 begin
12   var taken := false;
13
14   op request () ==
15     await ~taken;
16     taken := true;
17   op release () ==
18     taken := false
19   op isTaken(; out isfree) ==
20     isfree := taken
21 end
22
23 class User begin
24   op run (Resource mutex) ==
25     await mutex.request();
26     skip;
27     release;
28 end
```

The priority on a method invocation is optional; so the default priority for the method calls without it is a default priority with minimum value.

On the server side, therefore, an interface may introduce a *priority range* that is available to all clients. For instance, in Line 2 of `Resource` in Listing 13, we can define a priority range:

```
priority range 0..9
```

If the method invocations are scheduled *only* on the basis of the clients' priority requirements, the server object would be completely at the mercy of its clients; i.e. clients that put their highest priority only on `request` will cause the server to fail to schedule `release` which means that it will block assuming a proper behavior of the clients; e.g. never calling `release` after a completed `request`.

In this particular example, we can solve this problem by allowing the server to define a further priority on the methods themselves. We propose the following sample syntax to be added to the implementation of the `ExclusiveResource` server:

```
priority method order ("release" > "request")
```

The above syntax means that `"release"` has a higher priority than `"request"`. A general high-level specification for reconciling the different levels of priorities will be discussed later in this section. First, we discuss more refined levels of priorities.

In terms of co-interfaces, each object may also impose a priority ordering on the types of its co-interfaces. If there are method invocations from different co-interfaces, the object may decide which to serve first based on the priority of the co-interface. For instance, we add the following to Line 2 of `ExclusiveResource` in Listing 13):

```
priority interface order ("StatusPoller", "User")
```

As a final example, the server may also introduce priorities on certain characteristics of a method invocation such as the kind of "release statement" being executed. This involves the definition of a priority range in the server code and assigning different priorities to corresponding release points. So, we may rewrite Line 15 of `ExclusiveResource` in Listing 13 by adding an identifying label to the `await` statement:

```
@@important:  await ~taken;
```

We assume the following extension of the above qualitative ordering of method with this new label for this labeling of the `await` statement.

```
priority method order ("release" > "important" > "request")
```

The priority used in this syntax involves a declaration of a priority range generally depending on the structure of the class.

We discussed different levels of application-specific priorities. Note that now each method invocation in the queue of the object involves in general a tuple of priorities corresponding to different characteristics of callee, method name, await statement, etc. Therefore, we now introduce a high-level specification of an "abstract priority manager" which synthesizes different levels of priorities. We define a general function, δ, to allow the object to map different levels of priority into one priority value. The mapped value is used to order messages in the object's queue:

$$\delta : P_1 \times P_2 \times P_3 \times \ldots \times P_n \longrightarrow P$$

In other words, δ is a function that maps different levels of priority in the object ($\{P_1, \ldots, P_n\}$) to a priority value in P that is internally used by the object to order all the messages that are queued for execution. In an extended version of δ, it can also involve the internal state of the object, S, including the values of the field of the object that allows the definition of dynamic priorities.

For example, in `ExclusiveResource`, we have two different levels of priorities namely the client's priority and the priority on the methods' names. The range of client's priority is $P_1 = \{0, \ldots, 9\}$. Also, in method name priorities, they are internally translated into a numerical range; i.e. `("release" > "important" > "request")` is respectively translated to $P_2 = \{0, 1, 2\}$. So, we define $\delta : P_1 \times P_2 \rightarrow P$ as:

$$\delta(p_1, p_2) = p_1 + p_2 \times |P_1|$$

As an example, if we have two method invocations such as $(p_1^a, p_2^a) = (5,$-`"release"`) and $(p_1^b, p_2^b) = (5,$`"request"`$)$, then $\delta(p_1^a, p_2^a) = 25$ and $\delta(p_1^b, p_2^b) = 5$ showing that how method name priorities are respected. It is obvious that the range of the final priority value is $P = \{0, \ldots, 29\}$.

Note that the abstract priority manager in general does *not* completely fix the "choice" of the method invocation to be executed. For example, there may be many method invocations with the same priority the selection of which one is still unspecified. Further, the abstract priority manager does not in general enforce strong or weak fairness. In our envisioned tool architecture, we include an extensible library of predefined scheduling policies such as strong or weak fairness that further refine the application-specific multi-level priority scheduling. The policies provided by the library are available to the user to annotate the classes. We may declare a predefined scheduling policy in Line 13 of `ExclusiveResource` in Listing 13:

```
uses scheduling policy StronglyFairScheduler;
```

These scheduling policies will also be represented in terms of a corresponding priority range in the δ function. In summary, there will be, first, priorities that directly stem from the model and are statically specified by the client or the server and, second, there will be priorities that dynamically change through time based on the runtime information of the application. In the latter case, a scheduling policy may use runtime information to re-compute the dynamic priorities and ensure properties such as fairness of the selected messages; for instance, it may take advantage of "aging" technique. Thus, δ will take into consideration all the different concerns when assigning a final priority value to a message.

4.1.2 Abstract Multicore Machine

In the previous section, we discussed how we extend Creol to provide different levels of priority scheduling in order to resolve the scheduling issue of selecting one method of a given object to execute. However, there is another issue of how to select an object for resource allocation.

In the application level, objects need to be monitored for runtime information to optimize resource utilization. Additionally, load balancing techniques are heavily useful in this context as they maintain runtime information about

objects. The "Abstract Multicore Machine" (AMM) provides a model of the shared resources, including the number of cores, caches, bandwidth, etc., and will be used to specify strategies for load balancing and scheduling of resources including memory components. At this level, we need to profile each object to maintain different information on its execution including memory consumption, I/O communications, processor utilization and such forth. This information is an oracle used by AMM in conjunction with operating system to manage and balance the load among different objects.

AMM is orthogonal but complementary to concurrent objects priority scheduling. As objects internally determine how to express different priority requirements they also externally provide a model to obtain various information for a higher-level model that uses such information to inter-operate with operating system in the task of load balancing. Moreover, this division of perspectives brings in a separation of concerns along the line of following two major research directions. First is to investigate how to involve and allow active concurrent objects to cooperate in the process of scheduling. And, second is how to develop and implement AMM in regards with operating system API.

We do *not* investigate AMM in this thesis. Figure 4.1 depicts the overall structure of a multicore deployment using Creol language.

4.2 Cacoj Compiler

4.2.1 Creol Extension's Implementation

4.2.2 Active Object Profiling

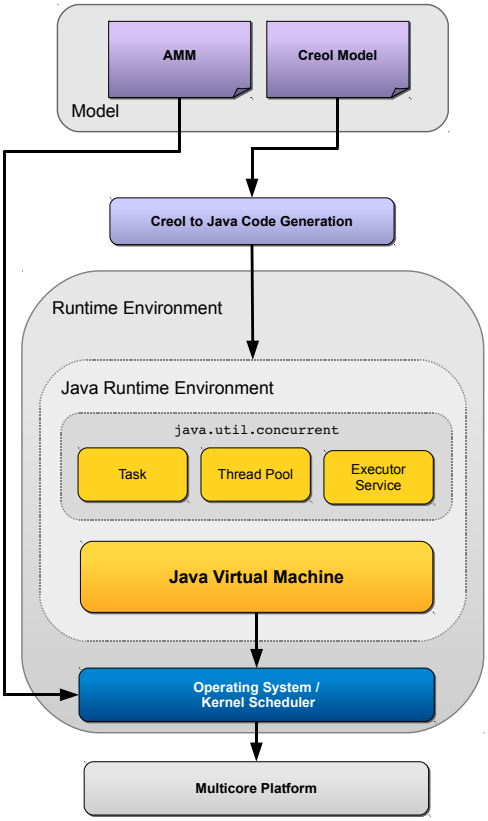

Figure 4.1: Abstract Multicore Machine

Chapter 5

Related Work

5.1 Concurrent Programming Paradigms

5.1.1 Actor Model

Agha, in [20], introduces "Actors" as an inherently concurrent programming model. According to [36], in the Actor model, systems comprise of concurrent and autonomous entities called *actors* and *messages*. Actors communicate by sending asynchronous messages to other actors for which they should be aware of the destination actor *name* (mailbox). Each actor in response to the receiving messages can display different behavior as [20] proposes:

1. Send a finite set of messages to other known actors;
2. Create a a finite set of new actors; and
3. Define how it will behave in relation to the next incoming messages

To realize the actor model characteristics, [20, 36] propose that each actor semantics should provide:

Encapsulation In this model, there are basically two concerns:

1. State Encapsulation: An actor cannot directly access the internal state of another actor. An actor may affect other actors' internal state through messages that it sends to them [20, 36].
2. Safe Messaging: As [36] mentions, there is no shared state between actors. Therefore, message passing should be done through call-by-value semantics.

Fair Scheduling [36] proposes the notion of *fairness* in actor model meaning that each message is eventually delivered to its destination actor. This infers that no actor can be starved for ever.

Location Transparency As described in [20, 36], actors communicate through their mail box address. Thus the actor's name should not be dependent on its physical address. This also affects the mobility of actors in case of location transparency.

Transparent Migrations This property falls into two types of mobility: strong and weak [20]. Strong mobility refers to the possibility of migrating both execution state and code while weak mobility refers to the movement of the actor code.

In actor model, there is an extensive usage of "pattern matching" for messages that are entrant to the mail box of an actor. This is also important in case of an internal representation of actor. When a programmer is writing an actor, she needs to specify the messages to which the actor will respond. Thus, actor model supporters also tend to take much advantage of "functional programming" concepts [25, 22, 50, 29].

[36] provides a good comparison of the frameworks implemented for JVM platform.

5.1.2 Software Transactional Memory

Shavit and Touitou [47] introduce software transactional memory as a novel method for supporting transactional programming of synchronized operations. They use STM to provide a general concurrent method for translating sequential object implementations to non-blocking ones. The most important property of STM is its *non-blocking* nature as opposed to lock-based programming models.

According to [47], the challenge of STM is to eliminate the deadlocks for transactions for which they propose the "helping" methodology. In helping methodology, each transaction tries to help the owner transaction complete its works to release the location others are waiting for.

Technically, a *transaction* is a finite sequence of local and shared memory machine instructions [47]. Transactions are either *read-transactional* or *write-transactional*; in the former, there is a read operation from a shared location while in the latter there is a write operation into a shared location [47]. Each transaction may either *fail* or *success*; in the case of success, the changes are atomically visible to other processes. Additionally, transactions are *isolated* and *atomic*; the former means that each transactions runs as if others are suspended while it runs and the latter means that they are all-or-nothing operations [19]. Thus, they give the programmer the illusion of a serial execution.

Accordingly, [47] proposes that a software transactional memory is *shared object* that behaves like a memory that supports multiple changes to its addresses by means of transactions. A transaction is a thread of control that applies a finite sequence of primitive operations to memory. This is why software transactional memory is believed to bring in the concept of database transactions on data into object-oriented paradigm. In this model, each object provides a set of *primitive operations* only through which the underlying object (shared memory)

can be manipulated. [47] introduces *wait-free*, *non-blocking*, and *swap-tolerant* transactions as types of STM.

To reason about STM, [47] introduces "real-time order" of processes in a system; i.e. operation `A` precedes operation `B` if `A`'s response occurs before `B`'s response. With this definition, two operations are *concurrent* if they are unrelated according to real-time order. A sequence of invocations and responses is called *history*. A *sequential history* is a history in which each invocation is followed by its corresponding response. The correctness requirement of STM is based on *linearizability* of a concurrent history being equivalent to some legal sequential history that is consistent to some real-time order induced by the concurrent history.

In an evaluation, in [23], they propose that the promise of STM may likely be undermined by its overheads and workload applications. They believe that STM introduces new issues into programming models:

Interaction with non-transactional codes that includes accessing data that is outside the transaction.

Exceptions and serializability that is a question on how to handle exceptions and propagate them within the context of transactions to outside.

Interaction with code that cannot be transactionalized that can be a requirement in specific circumstances.

Livelock that is a property to ensure that all transactions are making progress through time.

Based on these issues, they believe that STM has not yet matured and will not by itself solve the challenges of parallel programming, however, they are useful in the context of *concurrent data structures*. They also argue that STM introduces nontrivial drawbacks with respect to performance and programming semantics:

Overheads. STM approach inherently introduces overhead that is in direct opposition with performance.

Semantics that falls into:

1. Weak Atomicity that occurs when there is no detection of conflicts based on transactional and non-transactional regions.
2. Privatization: Being in the context of a transaction or being used in private by the underlying object can make the design decision more complex.
3. Memory Reclamation: Some designs restrict the use of memory accessing constructs directly that becomes a burden.

Legacy Binaries. STMs that take advantage of code instrumentation on the original code face difficulties to be used with legacy programs for which the original code is not available.

They conclude that there are a lot of challenges in STM field including lowering the overheads of STMs and the elimination of unnecessary read and write operations. Finally, they believe that TM programming model introduces complexities that limit the expected productivity gains and reduces the tendency towards its usage.

In another research, [19], they provide some implementation and evaluation in favor of STM. They start by proposing the concept and requirement for languages to introduce new constructs to support software transactional memory. They reason that unlike coarse-grained locking, STM provides fine-grained locking mechanisms and boost scalability through:

1. Allowing concurrent read operations on the same variable since basic mutual exclusion locks do not permit concurrent readers.

2. Allowing concurrent read and write operations on *disjoint* variables that may comprise two or more active threads of control.

However, they also mention that in modular software engineering fine-grained locks are not feasible when modules are composed together. They conclude that yet STM is not the best solution in parallel programming but with the help of other technologies such as task decomposition or mapping, it has taken concrete step in making parallel programming easier.

In another research, [39], they provide a good overview of design and implementation issues in STM and they introduce RSTM. They propose that major *design issues* include:

Metadata Organization that is mainly about how to maintain information about acquired objects in a transactional system that is referred to as *transactional metadata*. They compare two approaches that are *per-object metadata* as used in DSTM [31] and *per-transaction metadata* as used in OSTM [28]. RSTM uses per-object metadata.

Conflict Detection. There are two approaches that existing STMs use for conflict detection. In *eager* approach, the objects are acquired at the soonest open time (DSTM). However, in *lazy* approach it is delayed until the commit time of the object (OSTM). RSTM supports both. Each approach has its own advantages and disadvantages. With eager detection there could be prevention of unnecessary future operations in close future while also taking the chance to ruin related work that is already performed. And, obviously, lazy detection may have the opposite properties.

Contention Management introduces the concept of competing objects and their circular dependencies that may lead to deadlocks. As proposed in [47], *helping* is used, however, it may also result in heavy interconnect contention and high cache miss ratio. Thus, some use the concept of obstruction freedom [30] that prevents livelock and starvation; RSTM is obstruction free.

Validating Readers talks about the inconsistency that may be caused in lazy approaches when some private data objects are created to be used later in write operations and in case of abortion they may still expose inconsistent data to other transactions.

Memory Management. There is always need for memory reclamation in software transactional memory. A general purpose garbage collector helps, however, in languages such as C++ in which direct access to memory is present the reclamation policy should be decided by the programmer.

Also, they summarize that what could be the source of *overhead* in STM implementations:

Bookkeeping talks about the maintenance of the objects to facilitate the fetch and write by objects.

Memory Management for metadata and private date objects that cooperate in the implementation of STM principles.

Conflict Resolution. There should be some speculation over the avoidance and resolution along with the operations required for helping concept.

Validation. As discussed before validating reader may introduce new and costly operations for the STM.

Copying. When to-be-written data object are created the copying process may not be so costly, however, in the case of large objects that only require a small change this could become a heavy cost.

They conclude with strengthening features of RSTM over other implementations, however, as this is an optimization approach, it also implies that STM has much of overhead and challenges that should be faced with.

5.1.3 Data Flow Programming

Data flow programming [3], that can be considered a branch of flow-based programming [5], focuses on facilitating parallel computation through division, conquerence and merging of a program based on the data it processes. The division occurs such that parallel flows of computation can be managed and directed after which the results are again merged to yield the final output of the program. Much of the decisions in the phases are upon the programmer but not the synchronization and parallelization parts.

Based on data flow programming, in [26], they propose MapReduce that is basically based on one *map* and one *reduce* function. It is originally proposed and extensively used at Google.

The user writes two function. “Map” takes an input pair and produces a set of intermediate key/value pairs. When the computation is done, the library groups together all intermediate values associated with the same intermediate

key and passes them to reduce function. "Reduce" function accepts an intermediate key and all the grouped associated values and it is supposed to merge the values to possibly form a smaller set of values. For instance, in [24], they use MapReduce in Machine Learning purposes.

[44] introduces Phoenix as an implementation of MapReduce on shared memory. They do an evaluation of MapReduce feasibility on multicore and multiprocessor systems. They start by arguing that the main feature of MapReduce approach is its *simplicity*. The programmer thinks about the functionality rather than parallelization concerns. However, they propose the question that "how widely applicable is the MapReduce model" is not studied in the research. They conclude that MapReduce is a useful programming and concurrency management approach for shared-memory systems that are heavily data-centric.

5.2 Languages and Libraries

5.2.1 Actor Model

A number of languages have been developed that either specifically or as-a-part support actor programming model:

Erlang [22] is a dynamically typed functional language that was developed at Ericsson Computer Science Laboratory with telecommunication purposes [25]. Erlang has a process-based model of concurrency. Concurrency is explicit and the user can precisely control which computations are performed sequentially and which are performed in parallel. Message passing between processes is asynchronous, that is, the sending process continues as soon as a message has been sent.

SALSA [50] is an actor-based language for mobile and Internet computing that provides three significant mechanisms based on actor model: token-passing continuations, join continuations, and first-class continuations. Essentially, a *continuation* is an executable behavior that is used when an actor completes its asynchronous response to a message.

In token-passing continuation, a concept of "customer" actor is defined that is part of a message sent to an actor. When the actor completes processing the message, it will pass by the token to the customer actor that will continue the processing the result.

In join continuation, a customer actor receives an array with the tokens returned by multiple actors once they have all finished processing their messages.

First-class continuation is a smart way to delegate computation to a third-party independent of the message processing context. In this type of continuation, a continuation is assigned to an object to be used when some processing is finished.

E Language [4] is a lambda-language such as Smalltalk that mainly comprises of the language and ELib; ELib provides is a pure Java library that provides distributed programming concepts. It provides inter-process messaging with security and encryption, event-loop concurrency and deadlock-free distributed object computing. E runs on JVM.

Ptolemy [37] is an actor-oriented open architecture and platform that is used to design, model and simulate embedded software. Their approach is hardware software co-design. It provides a platform framework along with a set of tools.

Axum [40] is a language that builds upon the architecture of the Web and principles of isolation, actors, and message-passing to increase application safety, responsiveness, scalability, and developer productivity.

Another stream of effort has been into developing libraries and frameworks for existing languages:

Java: Scala Actors Library [29, 17] : Scala is a hybrid object-oriented and functional programming language inspired by Java in which a famous Actors library exists that mimics the Erlang implementation. The most important concept introduced in [29, 17] is that Scala Actors unifies *thread-based* and *event-based* programming model to fill the gap for concurrency programming. In this model, an actor is a thread that can also react to events that come from other actors; i.e. it provides both "receive" and "react" features. "React" is a compliment to "receive" that is proposed in the original actor model.

Java: Kilim [48] is a framework used to create robust and massively concurrent actor systems in Java. It uses a bytecode post-processor called Weaver [48]. Kilim takes advantage of code annotations on bytecode to provide the facility.

Java: ActorFoundry [36] is a Java framework that brings the actor model implementation to the developer through the use of code annotations. It provides fair scheduling, actor mobility, and safe messaging out of the actor semantics.

Java: Jetlang [45] provides a high performance Java threading library. The library is based upon Retlang [46] for C#. The library is a complement to the `java.util.concurrent` package introduced in 1.5 and should be used for message based concurrency similar to event based actors in Scala. The library does not provide remote messaging capabilities. It is designed specifically for high performance in-memory messaging.

Java: JavAct [33] platform is based on the actor model and open implementation principles. With JavaAct, users write high-level Java standard code without considering low-level mechanisms such as threads, synchronization, RMI, Corba, etc. JavaAct has been designed in order to be minimal

in terms of code and so maintainable at low cost, portable, and easy-to-use for Java junior programmers who know a little of actors. It does not need any preprocessing and can be used with any standard Java toolkit.[1]

Java: AJ [53] is a software system for writing distributed programs in Java, and is based on the actor model. In AJ, an actor is an extension of an object: where objects communicate by calling each others methods, actors communicate by sending asynchronous messages to each other. AJ provides the messaging layers that allow actors to communicate with each other, no matter if the actors are on the same computer, or scattered across a network. AJ design goal has been experimentation in terms of clarity and modularity rather than performance.

Java: Jsasb. The project homepage has been removed.

C++: Act++ [38] is a class library for concurrent programming in C++ using actors model. Theron [16] is a lightweight, portable C++ class library for developing parallel applications. It implements a simple service-oriented model of concurrent processing based on the Actor Model.

Other implementations include Broadway, and Thal for C/C++, Stackless for Python, MS Asynchronous Agents Library and Retlang [46] for .NET, Stage for Ruby, and Actalk for Smalltalk.

5.2.2 Software Transactional Memory

Mutiverse [15] is a Java based STM implementation that aims at seamless integration in the language and language independence in the form a framework.

Clojure [32, 2] is a dynamic programming language as a dialect of LISP that target the Java Virtual Machine. Clojure provides concurrent programming constructs based on STM concepts.

JVSTM [13] , another Java based STM library, introduces two core concepts as "transactions" and "versioned boxes". The goal is to allow transaction programming at the programming language level, independent of an external transaction manager.

Intel C++ STM Compiler [10] is a C++ platform that provides STM concepts of isolation and atomic executions in a C++ compiler. Transactional Locking II (TL2) [27] is an STM algorithm based on a combination of commit-time locking and a novel global version-clock based validation technique.

[1] The documentation seems to be only in French.

	Java	C++	Python	C#
Actors	Scala, Erlang, SALSA, Axum, Ptolemy, Kilim, ActorFoundry, Jetlang, JavAct, AJ	Theron, Act++	Stackless, V3.2+	Retlang
STM	Multiverse, Clojure, DSTM2 (Fortress), Deuce, JVSTM	TL2 (Sun), Intel C++ STM	Durus	SXM
Data Flow	Java 7+ JSR 166y Fork/Join, Hadoop	OpenMP, TBoost.STM, Hadoop Streaming	Disco (Nokia)	Dryad, Hadoop Streaming

Figure 5.1: Summary of different libraries and languages for different paradigms

5.2.3 Data Flow Programming

Java 7 [11] is supposed to provide language-level features for data flow programming as proposed in JSR 166y [12] including fork/join.

Apache Hadoop project [7] develops software for reliable, scalable, distributed computing proposing several frameworks among which is MapReduce [8] that is a framework for distributed processing of large data sets on computer clusters.

Figure 5.1 summarizes some of the most current works for languages Java, C++, Python, and C# in the three discussed paradigms.

5.2.4 All-in-one solutions

There are also some interesting languages that have tried to provide all the different approaches in one library or language including Haskell and GPars.

Haskell

Haskell [9] is an advanced purely functional programming language which, among others, seems to have implemented more than one approach towards providing *built-in* multicore programming features. These , according to [18, 41], include a primitive type `MVar` that is a primitive in the language to provide *asynchronous channels*, an ability to spawn new concurrent threads via `forkIO` primitive, and software transactional memory through `TVars` and `retry` and `orElse` primitives. Additionally, it provides an actor model implementation that is internally dependent on Haskell's STM implementation.

GPars

GPars [6] is a Groovy/Java library that aims to provide easier techniques to programmers to handle concurrency tasks. The main areas include concurrent collection processing and fork/join abstraction, asynchronous operations, Actor model, data flow concurrency data structures, agent-oriented programming primitives and a planned[2] STM implementation.

5.2.5 Hybrid solutions

In another perspective, some have started to create a blend of different approaches to provide solutions and techniques for multicore programming.

Akka

Actor model is based on asynchronous message passing. That is why Scala, for instance, favors the use of *immutable objects* in message passing mechanisms. However, immutable objects make it so hard to actually have a *shared state* so that multiple threads and objects can manipulate.

Software transactional memory (STM) makes it simple to synchronize shared state manipulation with object transactions. It also has the advantage that transactions are *composable*. However, asynchronous communication is complicated with STM.

Akka [1] is a platform that provides a simple way to develop concurrent and fault-tolerant applications using a mixture of Actors and STM. It proposes the notion of **Transactors** that are Actors which support STM in their behavior, so, they provide *transactional*, *compositional*, *asynchronous*, *event-based* actors. Akka already takes advantage of another library Multiverse [15] that is an STM implementation for Java, Groovy, and Scala. The platform is still under development but working.

MPI

The MPI [14] standard includes point-to-point message-passing, collective communications, group and communicator concepts, process topologies, environmental management, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, some miscellaneous topics, and a profiling interface. Language bindings for C, C++ and Fortran are defined. This standard has an assumption that it will be used on a *distributed memory* architecture. On the contrary, the multicore architecture is commonly assumed to be based on a *shared memory* architecture. The difference should be noticed in this line for the applications.

[2] As of the date of this report

Chapter 6

Conclusion

In this thesis, we developed Cacoj that introduces a concurrency API for active objects in Java and a compiler tool to translate Creol's active objects into Java. We adopt a per-active-object control over scheduling of messages in contrast with related languages and frameworks; that is also why we choose Creol since it provides a natural approach towards multicore deployment using concurrent active objects. We discussed how the overall architecture of Cacoj facilitates the deployment of Creol objects into concurrent Java Runtime Environment. Cacoj's project web could be accessed at `http://sourceforge.net/apps/trac/cacoj/`.

As the major future work, we work on the Creol language extension to enable the support to specify different levels of priorities for method invocation including method invocation priority, priority ordering on methods/co-interfaces, and other levels prioritizing method invocations on statement levels. We will integrate this language extension into Cacoj through a priority management mechanism in line with the current scheduling management features. In addition, we intend to extend and integrate into our tool set the model checking engine Modere [34] that was earlier developed for an actor-based language. Moreover, another future work may involve profiling and monitoring objects at runtime for optimization and improving performance.

Bibliography

[1] Akka Project. http://akkasource.org/.

[2] Clojure concurrent programming. http://clojure.org/concurrent_programming.

[3] *Data Flow Programming*. http://en.wikipedia.org/wiki/Dataflow_programming.

[4] *The E Language*. http://erights.org/.

[5] *Flow-based Programming*. http://en.wikipedia.org/wiki/Flow-based_programming.

[6] GPars. http://gpars.codehaus.org/.

[7] Hadoop. http://hadoop.apache.org/.

[8] Hadoop MapReduce. http://hadoop.apache.org/mapreduce/.

[9] Haskell. http://www.haskell.org/.

[10] Intel C++ STM Compiler. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/.

[11] Java 7. https://jdk7.dev.java.net/.

[12] JSR 166: Java concurrency utilities. http://www.jcp.org/jsr/detail/166.jsp.

[13] JVSTM. http://web.ist.utl.pt/~joao.cachopo/jvstm/.

[14] MPI. http://www.mpi-forum.org/.

[15] Multiverse: Software Transactional Memory for Java. http://multiverse.codehaus.org/overview.html.

[16] Theron. http://theron.ashtonmason.net/index.php.

[17] *Coordination Models and Languages*, chapter Actors That Unify Threads and Events. Springer Berlin / Heidelberg, 2007.

[18] *Multicore Haskell Now!*, DEFUN 2009. `http://donsbot.wordpress.com/2009/09/05/defun-2009-multicore-programming-in-haskell-now/`.

[19] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking concurrency. *Queue*, 4:24–33, December 2006.

[20] Gul A. Agha. *Actors: a model of concurrent computation in distributed systems*. PhD thesis, MIT, 1986.

[21] G.R. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[22] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[23] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51:40–46, November 2008.

[24] Cheng T. Chu, Sang K. Kim, Yi A. Lin, Yuanyuan Yu, Gary R. Bradski, Andrew Y. Ng, and Kunle Olukotun. Map-reduce for machine learning on multicore. In *NIPS*, pages 281–288. MIT Press, 2006.

[25] Fábio Corrêa. Actors in a new "highly parallel" world. In *Proceedings of the Warm Up Workshop for ACM/IEEE ICSE 2010*, WUP '09, pages 21–24. ACM, 2009.

[26] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[27] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In Shlomi Dolev, editor, *Distributed Computing*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer Berlin / Heidelberg, 2006.

[28] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25, May 2007.

[29] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202 – 220, 2009.

[30] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: double-ended queues as an example. In *Distributed Computing Systems*, pages 522–529. IEEE, May 2003.

[31] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, PODC '03, pages 92–101. ACM, 2003.

[32] Rich Hicky. Clojure. http://clojure.org/.

[33] S. Rougemaille J.-P. Arcangeli, F. Migeon. *JavaAct*. http://www.javact.org/JavAct.html.

[34] Mohammad Mahdi Jaghoori, Ali Movaghar, and Marjan Sirjani. Modere: the model-checking engine of rebeca. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC), Dijon, France, April 23-27, 2006*, pages 1810–1815, 2006.

[35] Einar Broch Johnsen and Olaf Owe. An Asynchronous Communication Model for Distributed Concurrent Objects. *Software and Systems Modeling*, 2007.

[36] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the JVM platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20. ACM, 2009.

[37] Edward A. Lee. Overview of the Ptolemy Project. Technical report, University of California, Berkeley, 2003.

[38] Loch M., Mukherji M., and Lavendar G. Act++ 2.0: A class library for concurrent programming in C++ using actors. *Journal of Object-Oriented Programming*, 1993.

[39] Vriendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Esienstat, William N. Scherer III, and Michael L. Scot. Lowering the overhead of nonblocking software transactional memory. In *Proceedings ACM SIGPLAN*, TRANSACT '01. ACM, 2006.

[40] Microsoft Corporation. *Axum Programming Language*. http://msdn.microsoft.com/en-us/devlabs/dd795202.aspx.

[41] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real World Haskell*, chapter (24) Concurrent and multicore programming. O'Reilly, 2008.

[42] Terence Parr. Antlr. http://antlr.org/.

[43] Terence Parr. StringTemplate Library.

[44] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 13–24. IEEE Computer Society, 2007.

[45] Mike Rettig. *Jetlang*. http://code.google.com/p/jetlang/.

[46] Mike Rettig and Graham Nash. *Retlang*. `http://code.google.com/p/retlang/`.

[47] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10:99–116, 1997.

[48] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-Typed Actors for Java. In *ECOOP 2008 Object-Oriented Programming*, volume 5142, pages 104–128. Springer Berlin / Heidelberg, 2008.

[49] Venkat Subramaniam. *Programming Scala: Tackle Multi-Core Complexity on the Java Virtual Machine*. Pragmatic Bookshelf, 1 edition, July 2009.

[50] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with salsa. *SIGPLAN Not.*, 36:20–34, December 2001.

[51] Wikipedia. Generating primes.

[52] Wikipedia. Sieve of Eratosthenes.

[53] Wililla Zwicky. *AJ: A system for building actors with Java*. PhD thesis, University of Illinois at Urbana-Champaign, 2008.