CONCURRENT POLYGLOT: AN EXTENSIBLE COMPILER FRAMEWORK

by

SAURABH SATISH KOTHARI

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2011

ACKNOWLEDGEMENTS

I would like to express my deep and sincere gratitude to my supervising professor Dr. Nathaniel Nystrom who has shown tremendous faith and patience in me. He has supported me throughout the work on this thesis and has given me priceless advice on many occasions. I have gained invaluable knowledge working with him, which will undoubtedly help me in the years to come.

I would also like to thank my committee members, Dr. Christoph Csallner and Dr. Bahram Khalili for giving me invaluable suggestions and advice.

I am grateful to all my friends in Arlington who have spent time with me during the course of my master's here. I am also grateful to all my friends back home in Bangalore who encouraged me through my studies. I would especially like to thank my roommates: Anjan, Ram and Sharath for their immense support, without which I would not have been able to work on this thesis in an efficient way.

I sincerely appreciate my parents' and my sisters' love and support, which has enabled me to come this far in my field of study. They have always encouraged me in every possible way.

Special thanks go to my friend, Sharda Murthi, at Georgia Institute of Technology, whose patient love and faith has always been with me.

I dedicate this work to all who have associated with me in some way in the present and in the past.

April 13, 2011

ABSTRACT

CONCURRENT POLYGLOT: AN EXTENSIBLE

COMPILER FRAMEWORK

Saurabh Satish Kothari, M.S.

The University of Texas at Arlington, 2011

Supervising Professor: Nathaniel Nystrom

We have today crossed the threshold of increasing clock frequencies as the dominant solution to faster computing, and it is imperative for software developers to be able to think in the concurrent and parallel paradigm. Important still is to equip programmers with the right tools to develop concurrent and parallel software.

Software concurrency is a widely researched area, and many of today's compilers concentrate on compiling for parallelism. However, although concurrent compilers have been explored since the 1970s, very few of them exist today that justify multicore environments. For instance, *gmake* [1] supports coarse grained parallelism through the –j option for building independent files in parallel. Our effort here brings into perspective a much finer grained approach: targeting the abstract syntax tree to extract opportunities for parallel and concurrent compilation.

Polyglot is an extensible compiler framework. It allows the extension of Java in domain-specific ways into languages that may define their own new constructs. Compilation in Polyglot is sequential. The compiler uses only one core, even if additional cores are present. In order to fully utilize the power of the underlying hardware, we present here our efforts in making Polyglot

concurrent. Parallelization of the compilation process can yield faster compilation times. It also presents the correctness and scalability challenges that make this a test bed for experimenting with various concurrency models.

As part of the larger Polyglot project, we evaluate and present here the performance of Concurrent Polyglot against benchmarks and compare with the performance of Sequential Polyglot. We also present correctness of the output produced by Concurrent Polyglot.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

Figure Page

LIST OF TABLES

Table Page

CHAPTER 1

INTRODUCTION

1.1 Context

Chip designers have made tremendous progress in the recent years in their quest to boost computing power. Much of this progress has been towards the design of parallel processing capabilities for reasons that have been studied and researched widely. For example, Hwang and Xu [2] mention computing applications like accurate weather prediction, the human genome and computer vision among many others which are complex and require intensive computational power. This power initially came from increasing operating frequencies of processors, which was a direct consequence of increasing the number of transistors on a chip. Gordon Moore observed this rate of growth in his paper [3] - the number of transistors that can be placed on a single chip doubles every eighteen months. While this law may still hold, increasing the operating frequencies of processors may not be the solution to increased computational power. Grama et al. [4] argue that converting these high frequencies to useful operations per second (OPS) is more important. Factors like memory and disk access speeds which have not necessarily kept up with this trend of increasing processor frequencies cause undesirable overheads and deteriorate computing power. Parallel computing provides one approach by which higher frequencies can be converted to useful operations for powerful computation.

Traditional software applications fail to utilize this capacity of modern hardware efficiently because they are designed to be sequentially executed on a single processor. Harnessing the power of parallel processing for enhanced application performance has been the focus of many researchers over the past decade. Evolution of libraries and new

programming languages that allow programmers to build parallel applications has been tremendous. MPI [5] is one such library for the C programming language. Development of libraries does not involve as much effort as developing a whole new programming language. One example is the X10 language [6,7]. X10 is a high-level, object-oriented language that has powerful features and a large and powerful library to support parallel programming. The focus of this thesis is Polyglot, the compiler framework on which X10's compiler is based. Before the work on this thesis began, Polyglot was a sequential compiler that worked to compile source files in a series of passes over the abstract syntax tree (AST). Linearization of passes was simple: one pass had to run at a time and only after its prerequisite passes had completed their work on the AST. In this thesis we explore opportunities for parallelizing this process of compilation. This thesis is part of the larger Polyglot project.

1.1.1 Challenges of Parallel programming

The behavior of concurrent systems is not the same as that of sequential systems. Concurrency and parallelism bring new challenges to the forefront that did not exist before. Maurice Herlihy and Nir Shavit [8] describe correctness, liveness and safety as being properties of concurrent systems that represent these challenges. Correctness is application dependent. Some orderings are important and some are not. For example, type checking should have completed before any subsequent pass starts. They also describe how liveness guarantees can be expressed as blocking and non-blocking in nature. Maintaining progress of each pass is important. A pass should not starve or be deadlocked. Safety ensures that no process interferes in the critical sections of any other process.

Another important aspect of parallel programs is scalability. Parallelizing an application may be difficult, but making it scalable is a completely different challenge. Scalability in basic terms means that if the underlying hardware increases in the number of processors, the

performance of the application must increase accordingly with a fixed workload. If the number of processors reduces, the performance must reduce accordingly.

1.2 Motivation and Goals

Our motivation for the work presented here has been to:

1. Take advantage of parallel architecture to improve compile speed: we believe modern parallel architecture with multiple cores on the same machine will help in reducing compile times. We want to build a compiler framework that will set standards for future compilers to be able exploit the power of parallel hardware.
2. Experiment with new concurrent programming models such as in X10: X10 provides a high-level abstraction for parallel programming with a Java like syntax, powerful libraries and its programming model specifically aimed at parallel architecture. Employing this model in building a concurrent and parallel compiler will show how effective the model is in handling the computationally intensive task of compilation.

Hence, our goals are:

With the above motivations in mind, the goals of this thesis are to demonstrate:

1. Parallelization of the passes over the AST: We want to demonstrate the ability of the compiler to run independent passes simultaneously.
2. Maintenance of correctness, liveness and safety properties of passes over the AST: We want to show that by running independent passes at the same time, the results of the compilation process are similar to that of the sequential compiler.
3. Scalability of the Compiler: An important aspect of parallel software is its scalability. We want to show that the compiler thus built, will scale effectively with hardware.

This thesis also contributes towards identifying granularity levels for parallel compilation. If parallel tasks are too fine - that is if they perform too little computation - performance will suffer because of the extra space and time overhead. On the other hand, if tasks are too coarse, concurrency is lost and behavior can be sequential.

We have experimented here with different task sizes:

1. Sequential
2. Source file level
3. Class definition level
4. Method level and
5. Expression level

We present in this thesis the results of measuring performance with these configurations.

1.3 Thesis outline

The rest of this thesis is organized as follows. In Chapter 2, we present an introduction to the Polyglot framework and a brief overview of the X10 language and its concurrency model. In Chapter 3 we will survey the mechanisms used in Concurrent Polyglot, namely: Using futures in the abstract syntax tree (AST), shared data structures and synchronization. Chapter 4 describes our experiments with Concurrent Polyglot. We use various configurations for our experiments to set the level of granularity. Chapter 5 examines related work. Chapter 6 presents possible future work and concludes the contribution of this thesis.

CHAPTER 2

POLYGLOT, X10 AND THE X10 CONCURRENCY MODEL

This chapter is divided into three sections. In the first section we present a brief introduction to the Polyglot compiler framework. In the process we will see what Polyglot is, what are extensions and how Polyglot can be used to define new Java like languages. We explain the concept of extension with an example: extending Java 1.4 by adding the enhanced for loop in it.

In the second section we will then present an overview of the X10 programming language. X10 has a powerful concurrency model, and a rich library to support it. Most of the work in this thesis is based on this concurrency model and we present an overview here. We briefly discuss the relevant concept of futures and the constructs of async and finish. These constructs are important as we use them in our mechanics to make Polyglot concurrent.

The last section presents an overview of the Funicular [10] library for Scala [11], which is based on the X10 concurrency model. We have used the Funicular library extensively in our work and we consider it the backbone of Concurrent Polyglot.

2.1 Polyglot

2.1.1 Sequential Polyglot and extensions

Polyglot is an extensible compiler framework for Java and languages similar to Java. It helps in the creation of compilers for languages similar to Java that have domain-specific applications. Nystrom et al. [9] define an extension as a domain-specific modification of an existing programming language. The original language is called the *base* language and the

modified language is called an *extension* of the base language. An extension then defines a new front end for the compiler. It tells the compiler how to compile new or modified constructs written in the extended language and convert them to Java. The compiler spits out Java code as the result of compilation. The Java compiler is then invoked on the Java files to compile them to byte code.

2.1.1.1 Job

A *job* in Polyglot represents is a compilation unit (or source file). It can be passed to the compiler on the command line, or it can be implicitly pulled in for compilation as a result of a reference. Suppose there are two classes as shown in Figure 3.1.

```
//E.java
package foo;
public class E {
        public static void main(String [] args) {
                F f = new F();
                System.out.println(f.getA());

        }
}
//F.java
package foo;
public class F {
        private int a;

        public F(){
                a = 0;
        }

        public int getA(){
                return a;
        }
}
```

Figure 2.1: Example of a Job. Compilation of class F is triggered because it is referenced by class E.

E.java is passed to the compiler on the command line. E has a reference to F, hence F gets pulled in for compilation.

A job also has references to the AST and other data structures. It contains all data that is associated with a file that is used by more than one pass.

Sequential Polyglot works in a series of passes over the AST to translate files from source language to Java. These passes are:

1. Parsing – parsing is the process of analyzing the tokens of a source program and building its structural representation in the form of a tree. This structural representation is called the Abstract Syntax tree.
2. Type initialization – is the process of recording all the type information in the symbol table. New classes defined and library classes referenced are pulled in and pushed onto the symbol table.
3. Type checking – is the phase where type compatibility is checked. Java being a strongly typed language, it is important that types used in expressions are compatible with the types that the expression expects to see. For example, in the assignment statement:

   ```
   A a = new B();
   ```

   type B must be a subtype of A, assuming both A and B are concrete classes.

   Whether or not this statement is valid will depend on what is recorded in the type initialization pass, when information that B is a subtype of A is recorded (or not recorded).
4. Conformance checking - is the phase where conformance rules of the language are checked to be followed. For example no two classes can have the same fully resolved name. Or no two methods can have the same signature.
5. Reach checking – This pass checks whether all statements in a scope can be reached by all paths of execution. For example, if we have two return statements one after the other, the second one will never be reached.
6. Exception checks – exception checking does extensive analysis on whether all exceptions are handled appropriately. All exceptions that are thrown, must be caught in

the same scope or in any of the outer scopes. If the current scope doesn't handle the exception, the scope must declare that it throws that exception.

7. Code generation – is the final stage where the nodes of the AST are translated into Java and written to appropriate .java files. Then the files on the command line of the compiler are handed to javac to compile the Java files and generate the class files.

2.1.1.2 *Node*

Alfred Aho et al. [10] define an abstract syntax tree as a condensed form of a parse tree. Each node of the tree represents a construct from the grammar of the language. Concurrent Polyglot's nodes preserve this definition. Each node has fields that give some information about the construct that the node represents. Along with these fields, a node also contains a field of the type Ref, which makes it possible to delay its evaluation.

The AST in Polyglot is made up of nodes that are generated by the parser. These nodes are represented by a hierarchy of classes. At runtime, nodes are created on demand within this hierarchy by the node factory. A scheduler schedules relevant passes to go over the AST. Each node accepts a pass in the form of a visitor. The visitor derives necessary information and stores it in the node and other relevant data structures like the symbol table. This information is used by subsequent passes to derive information relevant to them.

The use of a parser generator, the node factory and the visitor pattern allow for a tremendous ability to change the AST nodes in flexible ways at various stages. For example, before the parser generator produces a parser, the grammar of the language can be modified to accommodate a new construct like the enhanced for loop for arrays and iterable types, as explained in section 2.1.3 of this chapter. The parser generator will then generate AST classes acknowledging this change. The node factory can then be modified to produce instances of nodes representing this new construct when the compiler runs. Visitors can be told to expect

this node while visiting the AST, and be instructed as to what need be done should they encounter these nodes. As a result, a new extension to the base language is born.

2.1.2 Extensibility

As described by Nystrom et al. [9] Polyglot uses PPG, an extensible parser generator, which allows defining the modifications to the base grammar of Java. PPG allows adding, deleting or modifying existing production symbols for Java. Once these changes to the base grammar are defined, new or modified AST nodes can be defined by implementing the Node interface and / or extending existing AST nodes.

The compilation process is a series of passes over the AST. A pass scheduler determines the order of these passes over the AST. This order is called a schedule. Flexibility of the Polyglot framework allows an extension to choose to add new passes or delete / modify existing passes from the original base language schedule. Hence, each extension has its own scheduler. The result of all the passes is a translation of the AST to Java source files. These files are then passed on to javac for bytecode generation. Thus, a new language extension is essentially translated to Java by the compiler.

2.1.3 An example: The enhanced for loop for Java 1.4

The concepts explained thus far can be concretized through an example. Suppose we want a construct to iterate over all the elements of an array irrespective of its size. The Java 1.4 specification did not contain a construct for this purpose. Programmers had to use the ugly for loop, determining the size of the collection each time and writing extra lines of code to manage bounds. Using index variables to iterate over arrays and collections was error prone if the bounds of the collection or array were violated. Java 1.5 introduced the enhanced for loop for Java.

The base language of the Polyglot framework is Java 1.4. To demonstrate the extensibility of this framework, we present here the EFor extension for Java 1.4 which essentially plugs the

enhanced for loop feature into Java 1.4. The following sub-sections are the steps involved in creating this extension.

2.1.3.1 Extension of the PPG for EFor

PPG [11] is a parser generator for extensible grammars based on the CUP parser generator [12]. We use the *newext* script provided in the Polyglot distribution to generate all the files required for the extension. To extend the base grammar, we then modify the PPG file generated by the script to define new productions as shown in Figure 2.2:

```
non terminal EFor foreach_statement;                                    //1
                                                                        //2
start with goal;                                                        //3
                                                                        //4
foreach_statement ::=                                                   //5
FOR:n LPAREN formal_parameter:a COLON expression:b RPAREN:d             //6
        statement:c                                                     //7
        {: RESULT = parser.nf.EFor(parser.pos(n, d), a, b, c); :};      //8
                                                                        //9
foreach_statement_no_short_if ::=                                       //10
FOR:n LPAREN formal_parameter:a COLON expression:b RPAREN:d             //11
        statement_no_short_if:c                                         //12
        {: RESULT = parser.nf.EFor(parser.pos(n, d), a, b, c); :};      //13
extend statement ::= foreach_statement:a {:  RESULT = a;  :};           //14
extend statement_no_short_if ::=                                        //15
        foreach_statement_no_short_if {: RESULT = a; :};                //16
```

Figure 2.2: Listing of the PPG grammar for the enhanced for loop

Here EFor is the newly defined non terminal, denoted by foreach_statement. The node class representing this should also have the same name. It extends the existing statement non terminal as defined by the last two productions listed on lines 14 and 15. Lines 5 through 13 have two productions which result in the EFor loop. Hence, we have defined how the enhanced for loop will be structured grammatically. It is exactly same as the enhanced for loop in Java 1.5.

2.1.3.2 Defining new AST nodes

The next step is to define the node interface for the EFor non terminal. The enhanced for is a looping statement, hence we extend from the existing construct interface Loop. We define the rewrite() method which will translate the source code into valid Java code as shown in Figure 2.2.

```
public interface EFor extends Loop{

Node rewrite(EForTypeSystem typeSystem,
        NodeFactory nodeFactory) throws SemanticException;

}
```

Figure 2.3: Listing for the EFor interface

We then implement this interface with a class EFor_c. This class has two important methods:

- **public** Node typeCheck(ContextVisitor tc) **throws** SemanticException {}

  This method type checks the EFor node by accepting a context visitor. Consider the example shown in Figure 2.3.

```
int [] intArray = new int [10];          //1
for(int eachInt: intArray){              //2
}                                        //3
```

Figure 2.4: Example of an enhanced for loop

  typeCheck then ensures that the type of the local to the right of the colon on line 2 is an iterable or an array type. It also checks if the local to the left is the same as the type that can be contained by the iterable or array defined by the local to the right. In the example above, we check if intArray is an array or an iterable. And then check if eachInt is of type int (same as intArray's elements).

- **public** Node rewrite(EForTypeSystem ts, NodeFactory nf) **throws** SemanticException;

  This is the implementation of the method we defined in the EFor interface. It contains code that converts the enhanced for loop to a normal for loop of Java 1.4, thus allowing

the programmer to use the enhanced for loop without worrying about violating boundaries of arrays or iterable types. The rewrite method takes care of the bounds of the array and sizes of the iterable types.

2.1.3.3 Scheduling the passes

The next step in the extension process is to schedule the passes over the source AST. We write this in the ExtensionInfo.java class created by the newext script. We call our scheduler the EForScheduler. It extends the default JLScheduler, which is the scheduler for the Java base language. Our scheduler inserts the rewrite pass between the typechecked and the reassembleAst passes as shown in Figure 2.4:

```
static class EForScheduler extends JLScheduler {
        EForScheduler(ExtensionInfo extInfo) {
    super(extInfo);
  }

  public List<Goal> goals(Job job) {
    List<Goal> goals = super.goals(job);

    List<Goal> result = new ArrayList<Goal>();

    for (Goal g : goals) {
    //insert rewrite between typechecked and reassembleAst
      if (g == ReassembleAST(job))
        result.add(Rewrite(job));
      result.add(g);
    }

    return result;
  }

  public Goal Rewrite(final Job job) {
    TypeSystem ts = job.extensionInfo().typeSystem();
    NodeFactory nf = job.extensionInfo().nodeFactory();
        Goal g = new VisitorGoal("Rewrite", job, new
              EForTranslator(job, ts, nf)).intern(this);
    return g;
  }
}
```

Figure 2.5: Listing of the scheduler for EFor

This completes the implementation of the enhanced for loop. To run the compiler with this new structure, we hand the files passed to Main on the command line onto the start method along with an instance of the above scheduler as shown in Figure 2.5:

```
public static void main(String[] args) {
  polyglot.main.Main polyglotMain =
        new polyglot.main.Main();

        try {
              polyglotMain.start(args,
              new polyglot.java5.efor.ExtensionInfo());
         }
        catch (polyglot.main.Main.TerminationException e) {
              System.err.println(e.getMessage());
              System.exit(1);
        }
}
```

Figure 2.6: Listing of Main method of the EFor extension

Hence, Java 1.4 is now enabled with a new feature of the enhanced for loop.

The last step – scheduling the passes over the AST – is the most important one with respect to concurrency. The compiler is structured as a set of goals, e.g. Typechecked(file) etc. Goals have dependencies – goals that must be evaluated before the pass for this goal is run. For example InitializeTypes(file) must run before Typechecked(file). As we shall see, to parallelize, we can make the passes concurrent, as long as dependencies remain satisfied.

2.2 X10

X10 is a high-level, object-oriented language that has powerful features and an extensive library to support parallel and concurrent programming. Saraswat et al. [6] describe how the X10 programming model is based on the key idea of Partitioned Global Address Space (PGAS). They also describe how the PGAS makes it possible for global asynchrony between light weight threads, which are called *activities.* Activities are spawned at *places*. A place can be thought of as being a set of processors on a node with local memories. Places allow work global work distribution. In our work presented here, we only consider concurrency and not distribution.

2.2.1 X10 Concurrency model

X10's concurrency model is central to the work in this thesis and we present it here. At the core of X10 is the concept of places. In a cluster of computers, a place can be thought of as a node made up of a processor and memory local to it. Each activity is spawned at some place and it can access data that is local only to its place. To be able to access data that is at a place other than their own, activities must spawn new activities at that place to read and convey that data for them.

When an activity X spawns a new activity Y, X decides whether or not to wait for the new activity Y to complete its task. The new activity Y may or may not convey back or receive messages from the parent activity X. To make all this possible, X10 has several constructs that support concurrency [6]. We review some relevant ones here.

2.2.1.1 async

Concurrent execution of instructions is achieved with the async construct. async S makes a new activity that executes statement S. async returns as soon as this new child activity is spawned, to resume execution of the parent activity. It is similar to the fork() operation in C, but only to the extent that a new activity is spawned.

```
Console.OUT.println("Sum: " + add(array));
async Console.OUT.println("Product: "
        + multiply(array));
for(x in array) Console.OUT.println(a);
```

Figure 2.7: Example use of the async construct in X10

As illustrated in Figure 2.6 the main activity spawns a new activity to multiply the numbers in a single dimension array after completing the call to add. The new activity created starts off, and the main activity moves on to printing the numbers in the array, it does not wait for multiply to complete. Note that all the activities here act upon the same data, but do not modify anything in it. They just produce different results.

2.2.1.2 finish

Suppose now that in the previous example for some reason we want the multiplication to complete before the numbers in the array are printed. X10 provides this ordering of instructions through the finish construct. finish S spawns a new activity that executes the statement S, but unlike async, it waits till all the activities spawned by S terminate.

```
Console.OUT.println("Sum: " + add(array));
finish{
        async Console.OUT.println("Product: "
+ multiply(array));
}
for(x in array) Console.OUT.println(a);
```

Figure 2.8: Example usage of the finish construct in X10

As illustrated in Figure 2.7, the async spawns off a new activity for multiplying the numbers in the array and the finish waits till this activity completes to move onto the for-loop displaying the numbers in the array.

2.2.1.3 Futures

Futures are a way of handing over the responsibility of computing the result of an expression to another activity. Once this responsibility is handed over the parent activity is allowed to demand that value at any time. If the value hasn't been computed yet, the parent activity stalls until it receives the value. Futures thus provide on demand access of non-local data. Futures are similar to the async construct, in that they evaluate a value asynchronously. The value of the data can be demanded at any time by forcing the future, which makes it different from asyncs.

To understand the concept of futures better, suppose that we want to find the sum of a million integers stored in an array of ints. Also suppose that we have a cluster with 100 machines or places. We decide to do this task using the scan and reduction technique as explained by Hwang [2]. An activity can be spawned off at each place, and handed the task of

adding up 10000 of the numbers each. This can be done by spawning a future at each place. Then when the sum from each place is needed, the future can be forced, and the result obtained and passed onto the next stage.

2.3 The Funicular library

The Funicular library [13] is an implementation of X10's support for concurrency in the Scala programming language. It leverages the Fork/Join framework [14] for Java. Fork/Join framework for Java is based on the idea of recursively splitting up a task into independent sub tasks and computing their results in parallel and then combining all the results to form the result for the original task.

In our implementation, we have used the constructs provided by the Funicular library extensively. This has enabled us to control and modify the granularity of parallelism as desired. In the next chapter we present the mechanisms of how this has been done. Thus the objective of this thesis for fine grained parallelism is achieved by building on the X10 concurrency model with the help of the Funicular library.

The Polyglot framework stared off and has continued to build versions of implementations in the Java language. For incorporating concurrency, it would have been very tedious to re-implement it in X10 only for using X10's support for concurrency. One of the collective objectives of this project was to be able to use the original code base. X10 and Scala have similar syntax. Scala also has good interoperability support for Java code. Odersky et al. [15] describe the interoperability of Scala with Java. This drove the idea to build the Funicular library which we have used to bridge the gap between X10's concurrency model and Java.

We mainly have used the following constructs provided by the Funicular library.

1. async
2. finish
3. future

These constructs have the exact same meaning as they do in X10. They follow the same syntax and semantics as explained earlier in this chapter.

2.4 Conclusion

This chapter has established a basis of the workings of Concurrent Polyglot. We explained what the Polyglot framework is. We then explained what extensibility means and how it is achieved in the Polyglot framework. We examined the implementation of one extension, the enhanced for loop for Java 1.4. Next we gave an overview of the X10 programming language and its concurrency model and how the Funicular library leverages the fork/join framework for Java to provide a similar concurrency model.

In the next chapter we examine how these constructs of the concurrency model are used to build the mechanisms by which Polyglot exploits parallelism opportunities for compiling source code.

CHAPTER 3

MECHANISMS
USING FUTURES IN THE AST, SHARED DATA STRUCTURES
AND SYNCHRONIZATION

With the background on Polyglot, X10's concurrency model and the funicular library in the previous chapter we now present the mechanisms used in Concurrent Polyglot. We go beyond the file level and explore how the compilation process can be parallelized on the level of the Abstract Syntax Tree and its nodes.

Concurrency and parallelism bring new challenges to the forefront that did not exist in the sequential compiler. Communication between concurrent processes and the use of shared data structures makes concurrency and parallelism complex issues. This complexity gives rise to the need for maintaining correctness of data, synchronization of processes that access these data, liveness of these processes, and mutual exclusion of these processes. Therefore, we present in this chapter the shared data structures between the parallel components of polyglot. We also present how X10's concurrency model along with Java's concurrency model and data structures have been used in order to achieve synchronization and ensure the liveness, correctness and safety properties.

3.1 Granularity

The amount of work that is done between synchronizations of concurrent tasks is called the granularity of parallelism. Granularity is a deciding factor in performance. When small amounts of work is done between synchronizations it is termed fine grained parallelism and

when relatively larger amounts of work is done between synchronizations, it is termed coarse grained parallelism. One can see then that in finer grained parallelism communicating too often leads to an increased overhead and performance can deteriorate. On the brighter side, fine grained parallelism provides for better workload distribution. Similarly, coarse grained parallelism reduces communication overhead but also creates problems in workload balancing and in some cases concurrency can be lost. One goal of this thesis is to show that parallelism can be achieved at the fine grained level of the abstract syntax tree. In the following sections we describe how this is done.

3.2 AST and parallelism

The first phase of a compiler is the parser. If there are no syntax errors, the parser generates an abstract syntax tree, a tree representation of the compilation unit. This tree is passed to subsequent phases of the compiler for them to traverse and make their changes on it. Each phase is called a pass of the compiler. After all phases are complete, Polyglot produces Java code equivalent to the source code as the final result.

The abstract syntax tree is made up of nodes which represent syntactic constructs of the programming language. Each pass of the compiler deduces new information about the tree nodes and stores it into data structures that may be shared with other passes of the compiler.

3.2.1 Goals, Parallel compilation and Parallel AST passes

As mentioned in Chapter 2, Polyglot compiles files by scheduling passes over the abstract syntax tree. In the sequential version, each pass is executed after its previous pass has completed. Each pass has a *goal* [16] to reach and a set of *prerequisite goals*, that must be reached before that pass can begin. If a pass completes, we say it has reached its goal.

The first two goals, Parsing and Type initialized setup information used by subsequent goals. Conformance checking and Exceptions checked are independent of each other and are both dependent on the Type checked goal. Code generation should always happen after the program goes through all checks. Figure 3.1 shows the relationship between individual goals in the architecture of Concurrent Polyglot.

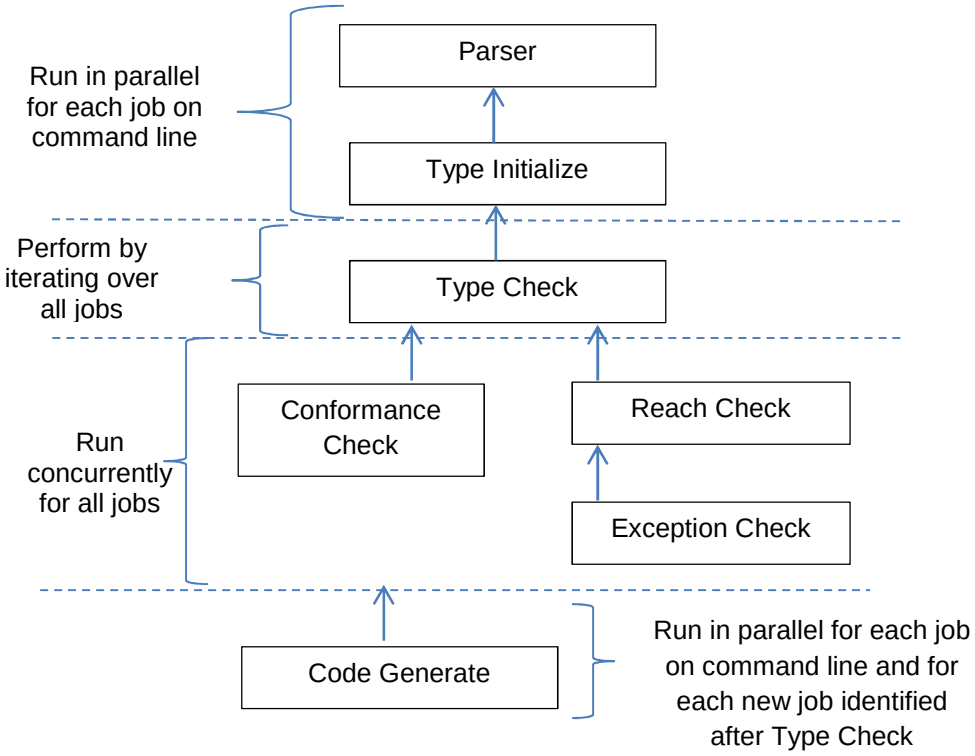

Figure 3.1: Overall architecture of Concurrent Polyglot

Parse and Type initialized goals are run in a separate finish block to ensure that they complete before any other pass starts execution. For each job passed to the compiler on the command line, a new activity is spawned that works on this first phase. Once all the jobs given

to the compiler on the command line are brought to the Types initialized goal, the compiler Type checks them. Type checking loops over all jobs passed on the command line, and all new jobs that are identified during the Type checking phase. When all jobs are at the Type checked goal, a new activity is spawned to run Conformance check and Exceptions check goals for each job independently. Reach check goal is a prerequisite to Exceptions checked goal, so Exceptions checked goal runs the Reach check goal first. After both these goals complete, Code generate goal is run independently for each job in a new activity. Hence, passes over the AST are made parallel for individual files. Multiple jobs specified on the command line are also compiled in parallel, making AST passes concurrent overall.

The following sub-sections explain concepts that have been used to in realizing the overeall architecture described in Figure 3.1.

3.2.1.1 Refs

A Ref represents a reference to an object of any arbitrary type T. Each node has a Ref set by the node itself on the request of a visitor (a pass of the compiler) to a ‘resolver’ that will evaluate the node. This resolver is either a Future or a LazyRef.

When the resolver is a LazyRef it does not spawn any new thread to evaluate the node. In fact, the thread that requested the value itself evaluates the node and updates its value. This is the sequential way of evaluating nodes.

A Ref can be set multiple times, usually over various passes on the AST. Mutual exclusion between the threads that set the ref is achieved through Locks as described later in this section.

LazyRefs are similar to Futures (explained next) except that a new thread of execution is not spawned and they do not start until they are forced. Once a LazyRef is created, it waits to be executed, which is done by calling force on it. Just like in Futures, LazyRef will return a value only when it is asked for through the get method. If the result of the computation is not available, the caller is blocked until the result is available.

3.2.1.2 Futures

Futures are the essence of simultaneous evaluation of AST nodes in Concurrent Polyglot. To understand futures, it is easy to think of them as light weight threads for asynchronous computation. When the resolver for a Ref is a Future type it is a new activity instance, pointed to by the future field of the Ref_c class. The get method in the Ref first checks if the ref has been computed. If it has, it returns that value. If not, it forces the future, i.e. it spawns a new execution thread and waits for its completion. When the new thread completes, the ref's future is updated to hold that value. If someone had called for the evaluation, the result is returned to that entity.

Each Ref has a future associated with it, which evaluates the ref when needed. This future is of type SettableFuture, described next.

3.2.1.3 Settable Futures

A SettableFuture supports the idea that a Ref can be evaluated more than once. Each of these evaluations must return a value that is greater than the old one in some predefined order. SettableFuture extends the functionality of the Future construct from the Funicular library, which was explained in the previous chapter. It adds fields for bookkeeping purposes and a set method which allows it to be set. Hence, as the name of this type suggests, the future of a settablefuture can be set to point to an instance of type callable, which is defined in the java.util.concurrent [17] library.

When the instance of a future that the settablefuture holds is forced, it is evaluated and returns the result of this evaluation to the thread who forced it. We call this type of a future a delayed future, because its evaluation is delayed to until when it is required.

The java.util.concurrent [17] library provides the method force on futures. We use the Funicular library's Force construct to wrap around the java.util.concurrent library's force which provides extra exception handling mechanism.

3.2.1.4 Resolution of Refs

Refs are used by visitors to a node in the AST to evaluate them. Resolution of a Ref happens every time a visitor requests the resolution for a node. As an example, consider a node that represents a field access, which is a node of type Field_c in the AST. The Type Initializer visitor traverses the AST and sets Refs of each node. When it reaches the Field_c node, it sets the Ref to a Future or a LazyRef (depending on the task size selected, which is explained in Chapter 4). When the Type checker visitor visits the Field_c node, it requests the evaluation of that node by forcing either the Future or the LazyRef. If a Future is forced, non-local computation takes place, which could cause a similar sequence of events if a new type is encountered when new classes are loaded. Hence the Ref set by Type initializer is resolved by the Type checker.

3.3 Shared Data Structures and Synchronization

Multiple passes over the AST derive varied information that needs to be communicated across passes. We present here the shared data structures that make this communication possible. We have used concurrent data structures like ConcurrentHashMap provided by the java.util.concurrent [17] library extensively for this purpose. ConcurrentHashMap fully supports concurrency for reads and writes and hence makes ideal storage equipment for our needs.

The presence of multiple threads doing multiple tasks and simultaneously modifying shared data necessitates the synchronization between these threads. We present synchronization mechanisms for Concurrent Polyglot used in this section.

Synchronization in Concurrent Polyglot is needed at two levels:

1. **Access to shared data structures** like the symbol table presented in this section needs to be synchronized. Mutual exclusion between threads that modify data is necessary so as not to corrupt data being modified, and to present correct data when it is read. Synchronized access to these data structures guarantees correctness of data at any given time.

2. **Evaluation of a Future** is considered its critical section. Given this and the fact that the future can be updated multiple number of times necessitates that the evaluation of the Future is synchronized. Mutual exclusion between activities that update the value of the future is achieved through the use of Locks. Every activity that wants to modify the value or read the value of a future, must obtain the lock on the future before doing so. We have extensively used here Locks that are provided by the Funicular library as a wrapper around ReentrantLocks in the java.util.concurrent [17] concurrency library.

In sequential compilation the symbol table is fully built when lookups happen. Unlike that, in concurrent processing of the AST by various passes, situation may arise when a particular symbol may be looked up by a pass and not yet be entered into the symbol table. Seshadri et al. [18] have explored this problem and call it the Do not Know Yet (DKY) problem, which means that when a symbol is looked up it may still not be in the table and may be potentially entered at a later stage. One of the solutions they proposed for this problem is that the lookup process / thread / activity block till the symbol is installed in the table. The approach we use in Concurrent Polyglot is to evaluate symbols on-demand, using LazyRefs and Futures. This is the approach we have implemented in Concurrent Polyglot. When a symbol is not found in the cache, the activity which is looking up the symbol is blocked and a new activity is spawned (by means of the async construct) to install the symbol in the cache. If the symbol is not found by the new activity, it simply unblocks the original one which then reports an error.

Following is a description of the shared data structures we have used:

3.3.1 ImportTable

The import table serves as a name resolver specific to a source file. It has a cache in which it stores names that are already resolved. This prevents future lookups from going to the

disk. The cache is a map from short names to long names. For example the String class is mapped with an entry:

{String=Some(java.lang.String)}

Hence, the import table has the additional responsibility of converting short names to their long forms, when needed.

Multiple futures working on different parts of the sub-tree corresponding to the source file may want to look up names at the same time. To support this we have synchronized access to the import table's cache which is a ConcurrentHashMap of names to classes found. For example, suppose two unrelated methods met1 and met2 are being type checked at the same time by futures f1 and f2. They both reference an object of the same type E. Assuming class E exists, and assuming that f1 and f2 almost simultaneously try looking up E in the import table, either of f1 or f2 are allowed to go first. Let's say f2 got to go first, and it doesn't find E in the import table, it does what's necessary to look up the class and then caches it in the import table. f1 goes next and finds E in the import table and doesn't have to do anything more. Hence, linearizability between accesses to the cache is achieved by using the *synchronized* structure provided by Java's concurrency library.

3.3.2 CachingResolver

The caching resolver caches results of other resolvers, so future lookups don't have to go to disk. For example, import tables that are specific to source files get cached here. Future passes don't have to go back to disk to look up information that was already resolved by a previous pass. This technique is called memoizing.

The cache is a map from a name to a Boolean. The Boolean indicates whether a previous pass was successful at resolving the symbol represented by the name. If the Boolean is set, then future passes don't have to go to disk for resolving symbols.

As explained above, more than one future at a time may want to access information this cache hold. This calls for synchronization of the access methods to the cache. In addition the cache itself is a ConcurrentHashMap of names to objects that are resolved.

3.3.3 SystemResolver

System resolver extends the behavior of the Caching resolver in that it is the main resolver for fully qualified names. It uses the import tables cached by the caching resolver to resolve fully qualified names.

The system resolver maintains a package cache in addition to the cache maintained by the caching resolver. The package cache is a map from name to Boolean that tells whether a particular name is the name of a package or not. For example, Figure 3.2 shows the state of the map during one of the passes over the dacapo benchmark [19] source code:

```
{
        java.lang=true, Harness=false,
        java=true, dacapo=true,
        java.lang.Object=false
}
```

Figure 3.2: State of the SystemResolver's cache

This says that java.lang is a package and so is java and dacapo. Harness and java.lang.Object are not packages. Harness is name of the harness class for the benchmark. When a particular name is not found, it is added to the cache. This makes it important to synchronize access to the package cache. Look-ups and update to the cache are synchronized and correctness is guaranteed through the use of the synchronized structure. In addition, like the caching resolver, the cache is itself a ConcurrentHashMap of name to Boolean.

3.3.4 TypeSystem

The TypeSystem object serves as a factory for types, methods, classes, fields etc., including all Java types. Each type that is defined gets cached in the TypeSystem. It maintains

a SystemResolver which resolves names as explained above. The Types system does not synchronize access to its data structures explicitly. It's all done implicitly by the data structures themselves.

The type system has many responsibilities which include creating instances of constructors, fields, methods, local instances and initializers. It also maintains legal flags for all these types of objects and helps in checking if the flags in a program are correctly applied to the corresponding entities.

The following data structures are part of the symbol table. Instances of these classes are created once for the corresponding entity they represent. For example, a class will have one ClassDef object created for it by the TypeSystem. Objects thus created get cached and are shared between futures. New information that is derived by passes gets stored in these objects to be read by passes that care for that information.

The need for synchronizing access to these objects is clear. More than one activity, possibly from different passes may need access to the objects at the same time. It could be either for writing newly derived information or just to read old information. Modifying old information is crucial from the view point of the futures that may need that information at a later point of time. We have synchronized all access methods in these objects that modify their state.

3.3.5 ClassDef

Classes that are defined in the source and parsed by the compiler end up being represented as instances of ClassDef.

The ClassDef has access methods that change the state of the object. These methods have been synchronized.

3.3.6 ConstructorDef

A ConstructorDef instance holds the definition of a constructor of a class. It wraps the ConstructorInstance class that holds type information for the constructor and hence provides synchronized access to this type information.

3.3.7 FieldDef

The FieldInstance class contains type information of a field in a class. The FieldDef provides synchronized access to this information. It also provides synchronized access to the initializer of the FieldDef.

3.3.8 InitializerDef

Each field or local must be initialized before it can be used. An InitializerDef is a wrapper around the InitializerInstance which holds type information for Initializers. InitializerDef then provides synchronized access to this type information.

3.3.9 MethodDef

Similar to the above definitions, a MethodDef is a wrapper around a MethodInstance that contains type information of a source method. It provides synchronized access to this type information.

3.3.10 LocalDef

A local is a variable defined inside the scope of a procedure. The procedure can be a method or a constructor or any other legal block. As all the above definitions, a LocalDef serves as a wrapper around a LocalInstance which holds the type information of a local's definition. Thus it provides synchronized access to this type information.

3.3.11 Ref_c

As explained previously a ref represents a reference to an instance of any arbitrary type T. Each node of the AST has a Ref which represents the node. The ref can be updated multiple number of times. The request to these updates can be triggered to happen simultaneously. Each new update must update the value of the ref to a greater value from the previous one, by some predefined order. To maintain correctness we synchronize the update procedure. Locks have been used to provide for mutual exclusion of process that try updation of the ref. A ref that is being updated has a callable (through the virtue of a future) that is in the process of evaluating the value of the ref. Hence, it needs to lock the ref till it can be updated with the value that is computed, and exclude all other updates during that time. This scheme provides for correctness as well.

Concurrent processing involves dealing with problems of mutual exclusion, starvation and producer-consumer problems. Mutual exclusion of activities is ensured through the use of Locks. Producer-consumer problems like installing of a symbol after its lookup fails can arise very often. This is avoided through the use of the synchronized construct and blocking till a looked up value is available. A pretypecheck pass ensures that no activity blocked on a lookup will ever be blocked unreasonably i.e. the activity knows that the symbol will be installed in the lookup table. Progress of activities is guaranteed and hence starvation is guaranteed not to happen, by ensuring there are enough number of worker threads, that eventually serve blocked threads and unblock them.

3.4 Conclusion

This chapter has explained the mechanisms that have been employed to make Polyglot concurrent. The most important of these mechanisms are the Futures and LazyRefs. Extensive

use of the robust framework provided by the funicular library's implementation of the X10 concurrency model has made these mechanisms possible.

In the next chapter we present experiments we conducted to measure the performance of the compiler.

CHAPTER 4

EXPERIMENTS

The previous chapter introduced the instruments with which Polyglot has been made concurrent. In this chapter we present the experiments we carried out with Concurrent Polyglot. In a parallel system, scalability is important. The aim of this chapter is to demonstrate how Concurrent Polyglot scales with processing power. This chapter is organized in four sections. In the first section we explain what scalable software is and present the setup of the experiments – the hardware and the tools used to measure performance. The second section presents the metrics that have been measured in the experiments. The third section explains the input to the experimental setup and the variation on this setup. The fourth section presents the results obtained. The final section concludes the chapter.

4.1 Setup of Experiments

The testing framework for our experiments has been setup on shared memory multiprocessor system. The Fermat server at The University of Lugano, Switzerland was used to test the compiler. The server is a Quad-Core Intel Xeon with eight cores and 8GB RAM. We used the processor affinity command *taskset* [20] to be able to choose the number of cores to use for each run.

4.2 Metrics Measured

Our experiments involved measuring the performance of the compiler in terms of the following metrics:

1. Time taken for execution, in milliseconds
2. Number of Futures created

3. Number of LazyRefs created

   The above two metrics are further classified into categories depending on the level at which futures are used. We explain it in the following section.

4. Size of the initial ASTs – in terms of the number of nodes created.
5. Number of files compiled – so we can see how the compiler scales vs. the size of the program and compare it to the sequential compiler.

The results will provide us with detailed insight about the behavior of Concurrent Polyglot. By analyzing the results we will further be able to make improvements to the performance of the compiler.

4.3 Input to the experimental setup

Our experiments were performed to:

1. Increase the workload and increase the number of processors to see how the compiler scales with respect to size of input.

Along with varying the size of the input, we have created variations in the task size to study the behavior of the compiler as it goes away from being sequential to being concurrent and fine grained. To be able to control the task size, we have included a command line option to the compiler '-usefuturesat' which tells the compiler at what level of the abstract syntax tree it must use futures to evaluate nodes. The following are possible:

1. Sequential - Using futures nowhere, and using LazyRefs everywhere
2. Source file level - Using futures at the source file level, and the others LazyRefs
3. Class definition level - Using futures at the class and all higher levels, and the others LazyRefs
4. Method level - Using futures at the method and all higher levels, and the others LazyRefs
5. Expression level - Using futures across the span of the abstract syntax tree, and no LazyRefs

It must be noted that all of the above are implemented with independent passes over the AST being concurrent. Hence the first one, Sequential, is sequential only with respect to the way Refs are resolved.

Our workload is the DaCapo benchmark [21,19] suite. The DaCapo benchmark suite contains many computation intensive programs, which pull in many standard Java libraries and hence serves as a good workload to measure the performance of the compiler. We used the DaCapo base drivers and the Antlr source code (which is part of the benchmark suite) to test the compiler.

4.4 Correctness of output

Concurrent Polyglot preserves the semantics of the program being compiled. We have verified this by a diff between the outputs of Sequential Polyglot and Concurrent Polyglot. Some of the results we obtained are shown in the figures that follow.

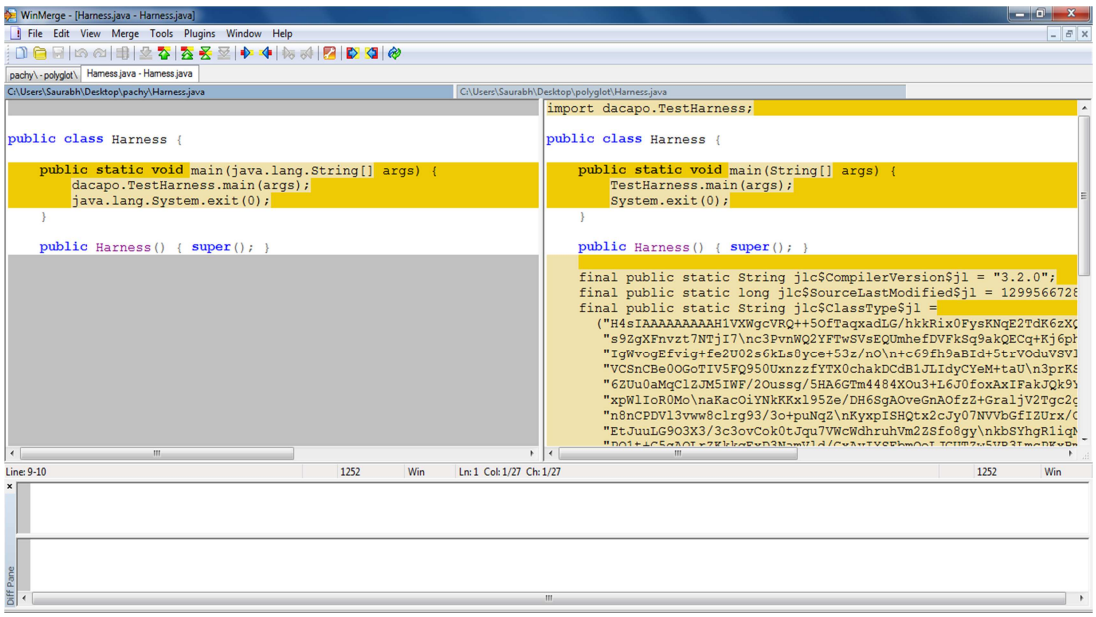

Figure 4.1 Diff of Harness.java (DaCapo base) produced by Concurrent Polyglot (left) with that produced by Polyglot (right).

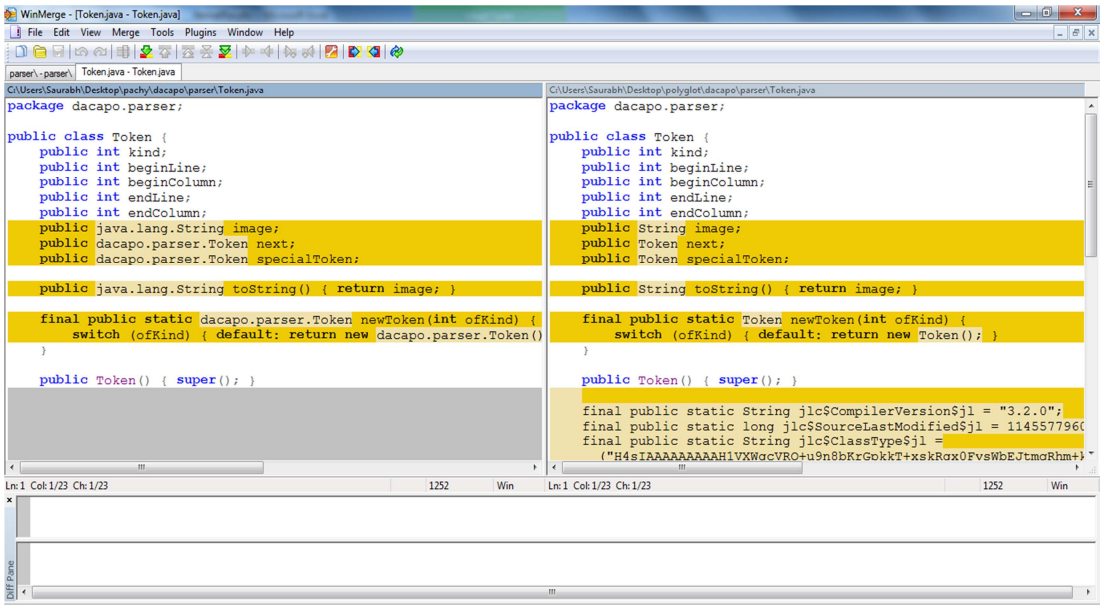

Figure 4.2 Diff of Token.java (DaCapo base) produced by Concurrent Polyglot (left) with that produced by Polyglot (right).

In Figure 4.2 and 4.3, the extra lines of code at the end of the output by Polyglot are the serialized type information, needed for separate compilation for language extensions. We have switched off the generation of this in Concurrent Polyglot, as seen at the right in the figure. Also, we have switched on the fully qualified name generation in Concurrent Polyglot, as is seen in the differences in the two figures.

4.5 Results

4.5.1. DaCapo Base [21]

We first compiled the DaCapo benchmark's base files (15 files) to measure the performance of the compiler on an Intel Core 2 Duo with 4GB RAM first. The following tables and charts show the results we obtained.

Table 4.1 Average compile time (ms) for Dacapo Base with 2 cores

Procs =2	Files = 15				
	All	File	Class	Method	None
	18040	18899	16070	15982	15639
	16130	21775	14019	14328	14811
	18768	23553	16558	16422	21635
	19779	25021	15872	19336	19568
	20918	25030	17968	20433	20208
	21225	25169	18316	20895	19790
	21272	24444	18045	20357	23225
	20305	26827	19442	22106	21303
	21908	25291	19457	20874	22260
	21198	26539	20024	20509	20169
	21222	26204	20366	22089	20230
Average	20272.5	24985.3	18006.7	19734.9	20319.9

The times shown in all tables here were obtained by running the compiler on the same input eleven times. The first time obtained is excluded from calculating the averages, so we don't time the runtime compiler, the JIT.

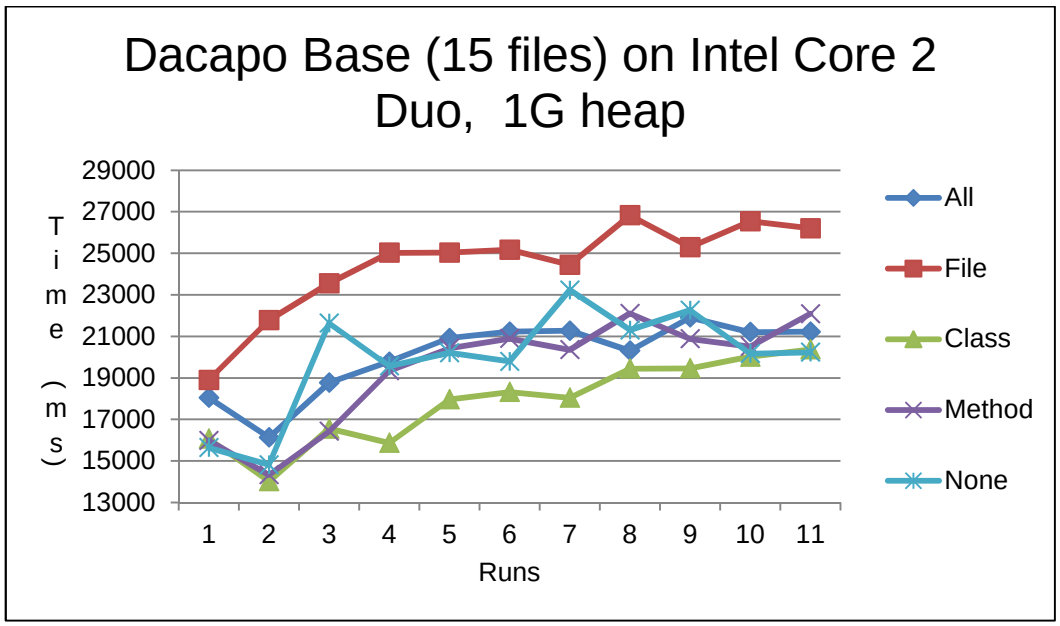

Figure 4.3 Graph showing run times of the initial test on DaCapo base files

As seen in the graph in Figure 4.3, a coarse grain size (task size File) performs worst. The other task sizes perform more or less equally well.

4.5.2 DaCapo Antlr

The performance of the compiler with larger workloads makes the difference of using Futures at the expression level (at all nodes in the AST) more apparent.

Parallelizing passes and using Futures to resolve each node of the AST, distributes the work of compiling the files on the command line more evenly among different threads available to the compiler. The only sequential part to the compilation process is then the Type Check goal. Type Check necessitates sequential execution because new Types could be added to the symbol table while type checking a file. New jobs could be identified during the process, and to keep the DKY problem from happening, this phase is sequentially executed.

Table 4.2 and Figure 4.4 show the runtimes of compiling the Antlr benchmark (with 152 files) in the DaCapo benchmark suite.

Table 4.2 Average compile time (ms) for Antlr with 8 cores

Procs = 8	Time in ms	Files = 152			
Task Size:	All	File	Class	Method	None
	203304	300019	276476	347798	327621
	192133	286816	281726	372093	359431
	196762	294798	269606	372929	360544
	192868	275671	273114	363495	357863
	189588	277744	276190	373790	359475
	189300	268900	282838	361941	357550
	186379	274095	273813	374929	347653
	187030	274138	262681	371982	357195
	184477	284203	262334	370718	350727
	185738	261199	276483	381381	350970
	188000	265152	262748	372084	352905
Average:	189227.5	276271.6	272153.3	371534.2	355431

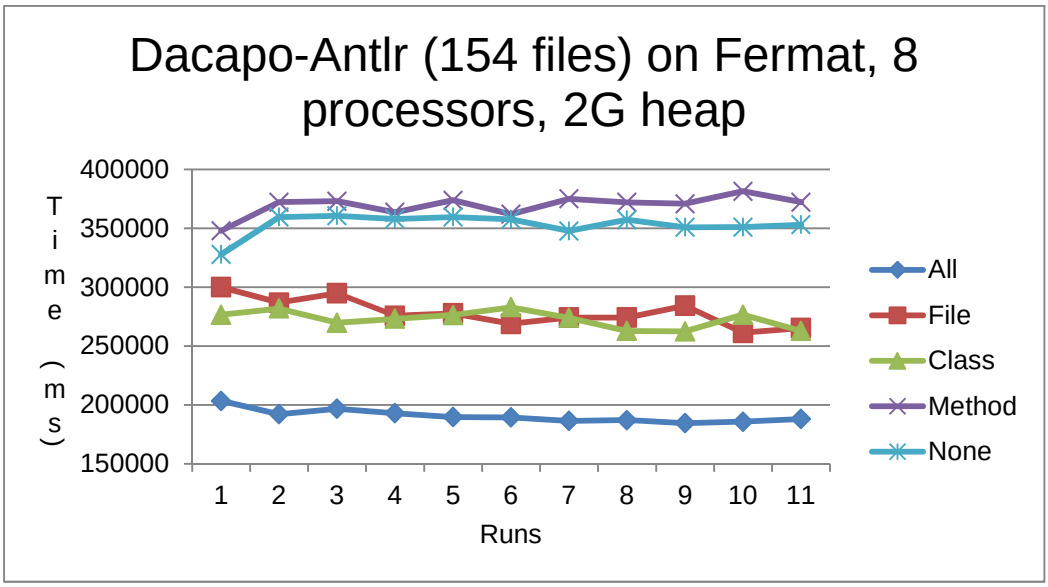

Figure 4.4 Graph showing run times for compiling the Antlr benchmark

As seen Figure 4.4, using Futures at all levels in the AST proves to be better than the other task sizes with increasing workload. It should be noted that all these levels also have parallelized passes.

Table 4.3 Number of Nodes, Futures and LazyRefs created compiling Dacapo Base

Files Compiled = 15	**Number of Nodes created**	**Number of lazyRefs created**	Futures Created
All-futures	25152	0	86137
FileLevel	25172	122141	15
ClassLevel	25172	122009	147
MethodLevel	25172	121022	927
None	25172	122156	0

Table 4.3 shows the number of futures and lazyRefs created along with the number of nodes created when compiling the DaCapo [19] base. These numbers are specific to the workload and remain constant over multiple runs.

CHAPTER 5

RELATED WORK

In this chapter we present work that has been done in the past and is similar to ours. We have learned from the experiences, concepts and techniques authors in the past have shared.

5.1 Early work on FORTRAN compilers

Concurrent compilers have been researched since the early 1970s. These early efforts concentrated mostly on the lexical analysis phase of the compiler. Krohn [22] mentions about work by Lincoln [23] in showing how lexical analysis can be performed on vector processors [24]. Krohn [22] demonstrates techniques for parallel code generation on Fortran like compilers. This early work talks about the decision to make syntactic and semantic analysis table driven. It then proposes to make these tables ‘sets of parallel vectors’. Early in the evolution of parallel computing, it recognized the importance of shared data structures in concurrent processing of source programs. Our approach too is similar in ways of maintaining shared data structures, which was presented in the third chapter.

The method suggested by Krohn [22] targets the code generation phase of the compiler. It first identifies statements which are not dependent on the order of evaluation of arithmetic expressions. The code for such statements is emitted first. Each class of these statements is evaluated in parallel. For example, all RETURN statements are processed in parallel. In the next stage, code for all arithmetic expressions is generated. And in the final stage, code for assignment statements is generated.

5.1.1 Dearth of Parallel Compilers

Despite a long history of parallel architectures and concurrent software, there have been surprisingly few parallel compilers. Part of the problem is complication of writing concurrent software. *gmake* [1] for example provides the –j option for parallelizing the build process. It compiles many files in parallel and thus makes the build process faster. Java and Java like languages have traditionally had compilers that are sequential in nature. Parallel compilers for Java that exist today mostly only exploit parallelism on the file level. Concurrent Polyglot proposes and demonstrates that parallel compilation of Java and extensions of Java on a much finer grain level is possible. Concurrent Polyglot also maintains the extensibility property that it inherits from the original sequential version. Hence, domain-specific Java like languages based on Polyglot's extension mechanism can now have parallel compilers.

Seshadri et al [18] propose an approach that splits the input stream into independent streams based on scopes for parallel compilation and relies largely on the concept of Do not Know Yet (DKY) for symbol lookups. Worton et al [25] implement this technique for the Modula2+ compiler. We use a similar approach as presented in Chapter 3.

5.2 Targeting syntax trees

Krohn [22] also mentions about work by Baer and Bovet [26] on parallel evaluation of arithmetic expressions by extracting a syntax tree: operations at the same level can be evaluated in parallel. For example, the expression

t = ((x + y) + (z*w)) / r

can be seen as a syntax tree as in Figure 5.1.

Figure 5.1 Syntax tree for the expression t = ((x + y) + (z*w)) / r

As can be seen in this example, expressions at the same level, say 7 can be evaluated in parallel, because they do not depend on each other.

Baer and Bovet's work [26] could be considered a milestone for concurrent compilation, for they recognized the opportunities to exploit parallel evaluation of the syntax tree. We use a similar approach here in keeping the idea of the syntax tree to exploit opportunities for parallel compilation.

5.3 Granularity on the function level

Gross et al. [27] demonstrate parallel compilation of functions on a cluster of computers infrastructure. They exploit the fact that programs written in Warp with different functions for the distributed memory architecture can be compiled in parallel. This was one of the earliest attempts at aiming for the distributed memory model and supported fine grained concurrency at the function level.

They explain the structure of Warp programs as being modular and being inherently based on the architecture of the machine. A Warp program has modules that have sections which describe the set of operations for a processing element. Hence, each section is executed by a processing element, and multiple sections are executed in parallel. Sections contain functions which are independent of functions in other sections and execute simultaneously with them.

This inherent hierarchical structure is exploited for parallel compilation of functions. The hierarchy is described to consist of three levels:

1. Master or Modular level
2. Section level
3. Function level

Each level has independent processes that compile the level independent of other processes. Processes at each level are forked by processes at their parent level. The parent level process is blocked until all forked children complete execution.

We use a similar idea of compilation at a fine grained level and in exploiting the hierarchical structure of programs. We go beyond the function level to be able to compile nodes of the AST in parallel. We have also demonstrated how this grain size (or task size) can be controlled to be set to different levels, even assigning a level to be able to run in a complete sequential mode.

.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In the first section of this chapter we present our conclusions with our work on this thesis. In the second section, we present possible future work.

6.1 Conclusion

The primary motivations of this thesis were to take advantage of parallel architecture to improve compile speed and to experiment with new concurrent programming models such as in X10. We have successfully demonstrated the use of X10 concurrency model, as presented in the third chapter of this thesis. We also showed in the fourth chapter the improvements in the compile times we achieved. Although, there is a huge overhead in using futures on the AST, we were able to demonstrate the control of task size, to experiment on various combinations of the number of cores / CPUs used and the task size. It still remains to be seen as to which combinations are the best and to generalize that in the use of Concurrent Polyglot.

We were successful in achieving the goals of this thesis which were:

1. Parallelization of the passes over the AST – as demonstrated in the third chapter
2. Maintenance of correctness, liveness and safety properties of passes over the AST – as demonstrated in the third and fourth chapters
3. Scalability of the Compiler – as demonstrated in the fourth chapter, with increasing workloads and the number of processors used (in some proportion to each other), concurrent polyglot performs much better than the sequential version.

6.2 Future work

The work on Concurrent Polyglot was proposed to be in three phases:

1. Refactoring the original sequential polyglot framework to make fine grained goals possible and to identify shared data structures.
2. Parallelization of compilation process making use of the X10 concurrency model.
3. Optimization to improve performance.

Concurrent polyglot at its current stage has gone through the first two main phases of development. The initial goal of making polyglot concurrent and working has been achieved. The challenges of concurrency and parallelism have been surpassed and we have been able to demonstrate that the compiler behaves much like the sequential version in preserving correctness and progress of all passes of the compiler.

The results we have obtained during our experiments with various workloads still leave much to be desired. We have identified the following future work for making Concurrent Polyglot a robust and reliable extensible compiler framework:

6.1.1 Optimizations

Rigorous optimizations need to be made to harness the full power of concurrent and parallel computer architecture. Currently there is a heavy overhead during the startup phase of the compiler which undermines all the benefits of concurrency (as was seen in the fourth chapter). Towards this we propose optimizations to be able to:

i) Control number of futures

Factors need to be identified to control the number of futures created. Uncontrolled growth in the number of futures makes it difficult to manage them. They bear a high overhead on memory too, making the memory requirements of concurrent polyglot unacceptable as compared to the sequential version. This can result in realistic changes and drastic improvements in compilation time reduction can be achieved.

ii) Memory Usage

Profile and identify classes that tend to instantiate more often than required and identify root causes and implement solutions for this problem.

iii) Removing Reflections based dispatch.

As opposed to Sequential Polyglot, Concurrent Polyglot uses reflections based dispatch (of visitors over the AST). This results in a higher compile time. Removing reflections based dispatch would mean a tradeoff between performance and ease of extensibility, which then brings out the need to work on deciding a good balance between the two.

6.1.2 Features

We suggest work on the following features to make Concurrent Polyglot an attractive compiler framework for developers:

i) Support for Java 5 (maybe even the current version of Java)

ii) Applying the ideas of polyglot to run-time compilers

The ideas of parallelized passes and using futures at various levels can be used in run-time compilers to achieve better JIT compile times.

iii) Using concurrency in language extensions

Currently, only the Java Language Compiler of the Polyglot framework has been made concurrent and has simultaneous passes. We intend to extend this functionality to all extensions within the Polyglot Compiler Framework. We will need to refactor the functionality from being inherent in the Java Language pass scheduler into higher level classes.

REFERENCES

[1] gmake manual. [Online]. http://www.gnu.org/s/hello/manual/make/Parallel.html#Parallel

[2] Kai Hwang and Zhiwei Xu, *Scalable parallel computing: technology, architecture, programming*.: WCB/McGraw-Hill, 1998.

[3] Gordon Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, number 8, no. April 19, 1965, p. 114 , April 1965.

[4] Ananth Grama, Anshul Gupta, Karypis George, and Vipin Kumar, *Introduction to Parallel Computing*, 2nd ed.: Addison Wesley, 2003.

[5] Message Passing Interface. [Online]. http://www.mcs.anl.gov/research/projects/mpi/

[6] Charles et al., "X10: an object-oriented approach to non-uniform cluster computing," *ACM SIGPLAN Notices*, pp. 519–538, 2005.

[7] X10 - Home. [Online]. http://x10.codehaus.org/

[8] Maurice Herlihy and Nir Shavit, *The art of multiprocessor programming*.: Morgan Kaufmann, 2008.

[9] N. Nystrom, M.R. Clarkson, and A.C. Myers, "Polyglot: an extensible compiler framework for Java," in *Proceedings of the 12th international conference on Compiler construction*, Berlin, Heidelberg, 2003, pp. 138–152.

[10] Alfred V. Aho, Jeffrey D Ullma, and Ravi Sethi, *Principles of compiler design*.: Addison-Wesley Pub. Co., 1977.

[11] Polyglot Parser Generator. [Online]. http://www.cs.cornell.edu/projects/polyglot/ppg.html

[12] CUP Parser Generator for Java. [Online]. http://www.cs.princeton.edu/~appel/modern/java/CUP/

[13] funicular - Funicular concurrency library for Scala - Google Project Hosting. [Online]. http://code.google.com/p/funicular/

[14] Doug Lea. A Java Fork/Join Framework. [Online]. http://gee.cs.oswego.edu/dl/papers/fj.pdf

[15] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*.: Artima Inc., 2008.

[16] Polyglot Website. [Online]. http://www.cs.cornell.edu/projects/polyglot/

[17] Java concurrency. [Online]. http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/package-summary.html

[18] V. Seshadri, S. Weber, Wortman D. B, Yu C. P, and I. Small, "Semantic analysis in a concurrent compiler," in *ACM SIGPLAN Notices*, New York, NY, USA, 1988, pp. 233–240.

[19] S. M, Blackburn et al., "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, Portland, 2006. [Online]. http://dacapobench.org/

[20] JProfiler. [Online]. http://www.ej-technologies.com/products/jprofiler/overview.html

[21] DaCapo Benchmark Suite beta 2006-08. DaCapo Benchmarks Home Page. [Online]. http://www.dacapobench.org/

[22] Howard E. Krohn, "A Parallel Approach to Code Generation," in *Proceedings of the conference on Programming languages and compilers for parallel and vector machines*, New York, NY, USA, January 1975, pp. 146-152.

[23] Neil Lincoln, "Parallel programming techniques for compilers," in *SIGPLAN Not. 5, 10* , October 1970, pp. 18-31.

[24] Roger Espasa, Mateo Valero, and James E. Smith, "Vector architectures: past, present and future," in *Proceedings of the 12th international conference on Supercomputing (ICS '98). ACM*, New York, NY, USA, 1998, pp. 425-432.

[25] David B Wortman and Michael D Junkin, "A concurrent compiler for Modula-2+," in *ACM SIGPLAN Notices*, New York, NY, USA, 1992, pp. 68–81.

[26] J L Baer and D P Bovet, "Compilation of Arithmetic Expressions for Parallel Computations," in *Proc. of IFIP. Congress*, 1968, pp. 34-46.

[27] T. Gross, Sobel A., and M. Zolg, "Parallel compilation for a parallel machine," in *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, 1989, pp. 91-100.

BIOGRAPHICAL INFORMATION

Saurabh Satish Kothari is a Master's student at the Department of Computer Science and Engineering, University of Texas at Arlington. Prior to taking up higher studies, he was a software developer with Wipro Technologies, India. He holds a B.E. degree in Computer Science and Engineering from the Visvesvaraya Technological Univresity, Belgaum, in the state of Karnataka, India. His research interests include programming languages, software engineering, artificial intelligence and parallel computing systems.