Improving the JCilk-1 Compiler and Runtime System

- Master's Thesis by M. Skou, A. Christensen, and P. Stilling -

- *Group d627a* -
Allan B. Christensen
Martin Skou
Per S. Stilling

Aalborg University

The Department of Computer Science
Aalborg University

Software Engineering, 10th semester

Title:
Improving the JCilk-1 Compiler and Runtime System

Project period:
1 February 2009 - 9 June 2009

Group:
d627a

Authors:
Allan B. Christensen
Martin Skou
Per S. Stilling

Supervisor:
Lone Leth Thomsen

Circulation: 5

Pages: 91

Appendices: A-K

Appendices on the CD: H-K

Abstract

This report deals with improving the JCilk-1 framework, a Java-based approach to the Cilk framework for C. The main features of JCilk-1 include call-return semantics for multi-threading using the `spawn` and `sync` primitives and implementation of a solution for passing exceptions from one thread to the parent that created it.
A number of the shortcomings in JCilk-1 are identified and a number of possible solutions are proposed. The advantages of each proposal are weighted based on the degree of improvement it will bring to the framework in its current state. Two proposals are chosen for implementation in the framework; simplification of the compilation architecture and increasing performance by lowering object allocation in the runtime system.
The architecture of JCilk is successfully simplified by removing the need for the intermediate language GoJava. The main concern of this change is the possible overhead that might arise from such a change, but tests show that the overhead is insignificant.
To increase performance, two proposals are described, an object manager and an object reuse system. The reuse system is chosen for implementation, and tests show significant performance gain using the system.

Preface

This report documents a project made by group d627a at the Department of Computer Science at Aalborg University. The report was written within the Database and Programming Technologies research group.

The report assumes that the reader has basic knowledge of programming languages and concurrent programming. Concepts within this field are used with the assumption that the reader understands them. In some cases, a brief description is given, however, and Appendix A defines some basic concepts used in the report. The first appearance of a concept which is described in Appendix A is marked in italic, such as *programmability*.

Names of code-specific elements such as classes, methods and variables are highlighted using `teletype` font. References to literature are written in square brackets. References to chapters, sections and appendices of the report are made in the format of: Chapter 2.

Many of the references in the project are to articles found on the Internet. This potentially raises the question of whether or not the sources are reliable, but we believe that we have chosen reliable sources and have been critical of which sources to use. Many references are to official sources on the subjects, such as MIT, Microsoft and Sun. Due to the nature of the project, much of the referenced data is quite new, which means that some of the points we reference have not been published in literature yet.

The report contains the following appendices. Appendix A–G can be found at the end of the report, and the appendices H–K are found on the enclosed CD:

- Appendix A: Basic Concepts
- Appendix B: Clone Example
- Appendix C: Object Allocation Test
- Appendix D: Object Access Test
- Appendix E: Handle_level Java Implementation
- Appendix F: JCilk Test Code
- Appendix G: Resume of this report in Danish
- Appendix H: Test Results in Excel format
- Appendix I: Compile Test Code, listed in Appendix F And The Runtime System
- Appendix J: JCilk Without Goto Compiler and JCilk With ReUse And No Goto
- Appendix K: Report in .pdf format

Project group d627a

Allan B. Christensen

Per S. Stilling

Martin Skou

Contents

Chapter 1

Motivation

This report documents our SW10 project, which is also our Master's Thesis. However, it is not our first work on this topic, and this chapter briefly presents our SW9 project, which is the project we worked on during the previous semester. This project is in part based on the findings of the SW9 project, which we review in this chapter. The SW9 project is referenced at [1].

The SW9 project was an examination of four concurrent languages with respect to their *concurrency models* and their *programmability.* The four languages examined were C#, Cilk, Fortress and Scala. We implemented two programs in each language; the Quicksort algorithm [2, PP 145-165] as well as the Sleeping Barber problem [3, PP 129-132].

We concluded that the concise *concurrency constructs* and syntax used in Cilk are useful to keep programmability high in languages while promoting concurrent programming. Additionally, we found that the model of thread task pooling and work-stealing used in Cilk and Fortress provides a good separation between the work specified in the program and the actual *threads* running on the system.

We also found that while the actor model used in Scala certainly has good applications, it is not the best model for general-purpose concurrent programming. Implicit *concurrency* as used in Fortress shows some promise, although there is the concern that it can create situations where it is difficult for programmers to understand what is going on in a program due to the lack of transparency in the concurrency of implicitly concurrent constructs.

We are mostly interested in developing a concurrent language extension for general-purpose OO languages such as C# or Java, and for this reason we choose the Cilk concurrency model to continue working on, since the results from SW9 indicate that this model is the one most likely to succeed in such a language. The SW9 project sums up the advantages of such a model over the existing concurrency models in these languages. These advantages include that the Cilk model scales better with hardware changes because of its dynamic work-to-thread mapping compared to the static mapping found in C# and Java, and the conciseness of the constructs used in Cilk.

This SW10 project follows up on the SW9 project by looking further into thread task pooling and work-stealing by examining related work in this specific approach to concurrent language design, and then proceed to choose one of those related works as the main focus of the project. We then analyze this work, and determine some way to improve upon it. We then implement some of these improvements, and finally test the improvements we have made in order to ensure that they are effective.

Chapter 2

Introduction

In recent years, hardware development has taken a turn away from the traditional single-core machines and towards multi-core architecture systems. In order to exploit the multi-core nature of these new systems, programming language development has begun to focus more on parallelism. Examples of this can be seen in the latest and upcoming versions of the two popular languages Java and C#. Java versions 5 and 6 introduced many new library features for assisting programmers in creating programs that run in parallel [4, 5], and Java 7 has even more of this, notably the new fork/join framework [6]. C# 4.0 will introduce a *task*/work-stealing library as well as other features for parallel programming [7].

A common attribute of all of these new features is that they are library-based features, which means that they facilitate easier writing of parallel programs within the same programming model. However, if the current trend continues, parallel programming will be a very large part of programming, and having the features embedded in libraries requires programmers to use them through an extra layer of abstraction which is not necessarily there in an extension. This could warrant the creation of language extensions in order to accommodate direct use of new features for parallel programming.

As mentioned in Chapter 1, this project is partly based on our SW9 project from the previous semester, which examined four concurrent programming languages with respect to their concurrency models, programmability and performance [1, Chapter 3]. The SW9 project resulted in proposals for future work projects in the area. This SW10 project adapts a combination of two of those proposals, the one for further investigation and the one which concerns implementing a Cilk-like concurrency model in an extension for a modern OOP language.

The Cilk model of concurrency is based on defining concurrent elements of work as tasks and leaving the scheduling of these tasks to a runtime system which uses work-stealing [8].

This task/work-stealing model will be described in more detail in Chapter 3. Rather than start development of an extension right away, we begin by looking at related work within the area. This examination will give us a better impression of common problems encountered in the area, as well as the possibility that we might be able to build upon an existing project rather than create our own extension.

We examine the original language using this concurrency model, Cilk, as well as several developments in OOP languages which introduce new concurrency features which share characteristics with the Cilk model of concurrency. We examine JCilk, a Java extension which implements the Cilk concurrency model into Java relatively faithfully, though there are some problems with transferring the model. JCilk is the result of the Master's Thesis projects of John S. Danaher and I-Ting Angelina Lee [9, 10]. Additionally, we examine the Java Fork/Join Framework being developed for Java 7 [6], as well as the Task Parallel Library for C# coming in C# 4.0 [7]. Finally, we examine Google's MapReduce programming model [11].

After this examination, we choose which of these projects it is best to build on, based on what will yield the best result, in terms of a good extension that programmers will want to use, or which is useful for further development.

2.1 Goals

The goals for this project can be divided into the following steps:

- Examine related work and identify common problems in this area.
- Choose a related work project to develop upon, or define our own project.
- Analyze the chosen project and its primary problems, in order to pinpoint our focus area(s) for further development of it.
- Define a plan for improving upon the project by solving one or more of its primary problems.
- Describe and implement solutions for the selected problem(s).
- Design test(s) for the implemented solution(s) in order to verify that they work as intended.
- Perform the test(s), and discuss the results.
- Evaluate and conclude upon the project.
- Finally, present proposals for future work in the area.

Chapter 3

Related Work

With the goals defined in the previous chapter in mind, we begin our examination of related work in this area with a look at the language which has inspired the trend we are working to continue, Cilk. After this, we review JCilk and how it combines the concepts from Cilk with Java. We then look at Java Fork/Join and the C# Task Parallel Library which are both currently in development, and finally, we examine Google MapReduce. After this, we choose one of these related works to develop further in this project, if we find that one is suitable.

3.1 Cilk

Cilk was developed at MIT by the CSAIL Supercomputing Technologies Group. The first article about Cilk was published in 1995. Cilk was developed following some work done on scheduling multi-threaded computations, especially the performance of work-stealing, which is the central theme of Cilk. [12]

Cilk is based on ANSI C, and introduces a number of new keywords to the language. The five most important of these are: `cilk`, `spawn`, `sync`, `abort` and `inlet`. An explanation of these keywords is given in Section 3.1.2.

Traditionally, parallel programming in the C family of languages has been done using message passing via the MPI interface, or with the OS-level mechanism of threads, where the programmer simply decides what is to be run in each thread, while the OS is responsible for scheduling and executing the threads themselves. If this scheduling is not acceptable, a runtime system can be inserted between the program and the OS. This gives a new level of control to programs, as they are now able to use other scheduling systems than the roundrobin time-share that most OS's use, where the only means to differentiate execution is by specifying a priority for threads [13].

To utilize multi-processor systems, it is necessary to divide the work in a program into multiple entities which can be executed concurrently. Dividing the program into a set number of threads according to an expected number of processors is not a scalable solution, since the number of processors in the host system may vary. Using less threads than there are processors means that some processors will be idle, while using more threads means wasting time creating threads and context switching between them, as well as using more memory than otherwise necessary. Even if there are exactly as many threads as processors, there is no guarantee that work is distributed equally among threads, which may mean that some processors are idle while others are working.

3.1.1 Tasks and Work-Stealing

Cilk offers a solution by providing a mechanism for dividing work into tasks rather than threads. This gives an abstraction that separates the definition of work from its execution. In Cilk, the programmer can remain oblivious of the thread abstraction, and focus on dividing work into tasks. Furthermore, this separation gives the programmer the opportunity to concentrate on structuring the program to expose parallelism without worrying about the overhead of thread creation or context switching.

Now that the tasks have been separated from the threads, a method is needed for assigning the available tasks to threads for execution. This is done by the scheduling technique known as work-stealing. With work-stealing, there are a number of workers, usually mapped one-to-one to the number of processors in the system, and each running in its own thread. Each worker has a queue which contains the tasks it currently has available for execution.

Each time a new task is created, it is added to the back of the deque belonging to the worker which created it. Whenever a worker finishes a task, it dequeues the task at the back end of its deque and begins executing it. If there are no tasks in the deque, the worker will 'steal' a task from the front end of a deque belonging to another worker. This means that when taking new work, workers treat their own deque like a stack (last-in-first-out), but they treat the deque of other workers as normal queues (first-in-first-out).

The reason that the deques are treated as stacks or queues based on which thread is using them is that this increases the possibility of exploiting the principle of locality. A worker taking its own tasks will take the newest created task, which also has the highest possibility of cache hits, while a worker taking tasks from other workers will take the oldest task, thus reducing the risk of causing cache misses for the 'victim' worker. [14]

Figure 3.1 illustrates work-stealing in practice. If Worker Thread 1 finishes executing its task (T1), it will take T5 next, as it is the most recently created task. If it creates a new task, it will be placed below T5 in the figure. If Worker Thread P finishes execution of T2, it will steal T3 from the deque of Worker Thread 1, as it is the oldest task in the deque.

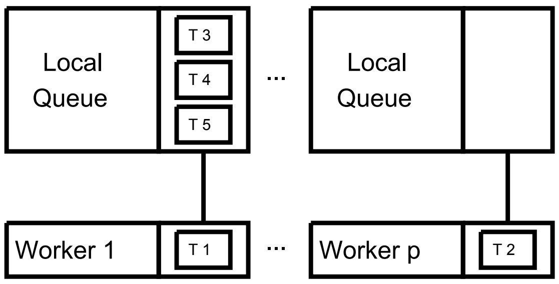


Figure 3.1: A snapshot of the work-stealing technique in progress.

3.1.2 Keywords

In order to provide syntax to describe, create and handle tasks, four main keywords have been introduced. `spawn` is used to create new tasks, while `cilk` marks functions which can be used to spawn new tasks. The `sync` keyword blocks execution until all child tasks have finished executing, and `abort` stops all child tasks.

```
 1  cilk void quicksort(int array[], int beg, int end)
 2  {
 3    if (end > beg + 1)
 4    {
 5      int pivot = array[beg], left = beg + 1, right = end;
 6      while (left < right)
 7      {
 8        if (array[left] <= pivot)
 9          left++;
10        else
11          swap(&array[left], &array[--right]);
12      }
13      swap(&array[--left], &array[beg]);
14      spawn quicksort(array, beg, left);
15      spawn quicksort(array, right, end);
16      sync;
17    }
18  }
```

Listing 3.1: Cilk example (Quicksort).

Listing 3.1 shows the Quicksort algorithm implemented in Cilk [1, Appendix A.2]. To specify it as a Cilk function, it is prefaced by the `cilk` keyword in line 1. This means that it can be used with the `spawn` keyword to create new tasks, thus acting as an identifier to help the compiler separate Cilk functions from C functions. This separation is useful such that C functions are not subject to the overhead of the Cilk framework.

Lines 14 and 15 show the `spawn` keyword in use, as they recursively create new tasks with the `quicksort` function. These tasks are added to the deque of the worker which executes the current task, and are then available to be executed after the current task, or to be stolen by other workers. Line 16 shows the `sync` keyword in use as a blocking operation to ensure that the array is fully sorted before the function returns. `sync` is a blocking operation that always ensures that all children of the current task have finished their execution before it returns.

Another keyword in Cilk is `inlet`. An `inlet` is an anonymous function nested inside a Cilk function. `inlet`s are used to `return` from `spawn`ed function calls atomically with regards to the rest of the function. For example, if a `spawn` calculates a value to add to a variable in the function, this addition can be specified as an atomic operation by an `inlet`.

3.1.3 Status

Cilk has been used for several award-winning applications, such as the computer chess program Socrates [15] and the Pousse program Cilk Pousse [16]. In general, Cilk is especially useful for programs where divide-and-conquer applies well. Traditionally, Cilk has been mostly used for high-performance computing, but now that multi-core processors are becoming mainstream, perhaps Cilk will see more general use.

Charles E. Leiserson and Matteo Frigo, two of the original creators of Cilk, have started a company, Cilk Arts, Inc, which is developing a commercial product called Cilk++, which is a C++-based version of Cilk. Additionally, the parallel programming features of C# 4.0 and Java 7 both use concepts similar to the tasks and work-stealing of Cilk.

3.2 JCilk

Like Cilk, JCilk was developed at MIT, and Charles Leiserson, one of the creators of Cilk, was also involved with this project. JCilk combines the concepts of work-stealing and tasks with Java. There are several advantages to using Java rather than C, such as automatic memory management and object orientation. The JCilk compiler is built using Polyglot and a modified version of the GNU Java compiler, Jgo. The runtime system is created in Java and runs on top of the Java Virtual Machine. [17]

Figure 3.2 shows the four involved languages in relation to each other. Cilk is based on C, and Java is also based on C, though C++ was a major step in between. JCilk is based on combining these two tangents of work into one language, combining all of the modern features of Java (object-orientation, memory management, generics, etc) with the elegant and efficient concurrency concepts of Cilk (fine-grained tasks, work-stealing). The idea is that this will introduce a language with better options for concurrency than Java while not requiring a lot to learn for users already familiar with Java.

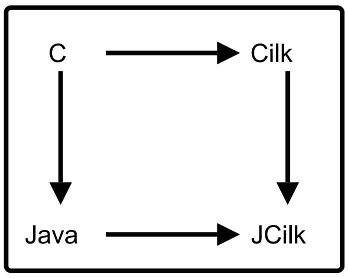

$$\begin{tikzcd}
\text{C} \arrow[r] \arrow[d] & \text{Cilk} \arrow[d] \\
\text{Java} \arrow[r] & \text{JCilk}
\end{tikzcd}$$

Figure 3.2: A diagram showing the convergence of two language tangents going out from C and ending up in JCilk. Based on a diagram from [9, Figure 1-1].

3.2.1 Status

The current version of JCilk is still a prototype implementation. As of this writing, the most recent article about JCilk was published in 2006, and it seems that there is not a lot of work being done on the system at the time of writing.

3.3 Java Fork/Join

The Java Fork/Join framework is being ported for inclusion in Java 7, which is scheduled for release in 2010 [18]. The framework is originally built by Doug Lea and presented subsequently in his white paper [19]. The goal of the framework is to simplify the definition of a parallel execution model for divide-and-conquer problems. The framework introduces two main concepts to facilitate this goal; fork and join. A problem is forked into sub problems, invoked in parallel and joined upon completion. The tasks which are created by the forks are managed by a group of threads using work-stealing as the scheduling mechanism, similar to Cilk and JCilk.

3.3.1 Library Specifics

Listing 3.2 shows the Fibonacci function implemented in the Fork/Join library. The first step when using the framework is to initialize a group of execution threads in line 17. Like other similar frameworks, the group-size is usually equal to the number of physical processors. In order to invoke a task on the work-group, the `invoke` method is used, as shown in line 19. Upon invocation, the `run` method is called for the initial `FJTask`. The `run` method spawns new `FJTasks` and executes them in parallel followed by a barrier for the two sub-tasks.

```
 1 class Fib extends FJTask {
 2   public int number;
 3    Fib(int n) { number = n; }
 4 
 5    public void run() {
 6       int n = number;
 7       if (n > 1) {
 8          Fib f1 = new Fib(n - 1);
 9          Fib f2 = new Fib(n - 2);
10          coInvoke(f1, f2);
11          number = f1.number + f2.number;
12       }
13    }
14 
15    public static void main(String[] args) {
16       int groupSize = 2;
17       FJTaskRunnerGroup group =  new FJTaskRunnerGroup(groupSize);
18       Fib f = new Fib(35);
19       group.invoke(f);
20       System.out.println("Answer: " + f.number);
21    }
22 }
```

Listing 3.2: Fork/Join framework example (Fibonacci). [19]

3.3.2 White Paper Conclusions

The white paper presents an extensive performance test, comparing the Fork/Join library to a number of other frameworks [19, pp 4-7]. The author reaches the conclusion that the Fork/Join framework shows that it is possible to create an efficient scalable parallel processing task-based framework in pure Java. Three other important observations are made in his conclusion; even though the garbage collector in the JVM scales well with parallelism, it does create an overhead proportional to the number of running threads due to frequent object creation. The second observation is that significant performance loss is only seen on machines with a very high number of processors in the range of 16 and 32 cores. The final observation is that the most important factor on how well an application will scale is the nature of the problem rather than the specific framework used to solve it.

3.4 C# Task Parallel Library

The Task Parallel Library is a part of the Parallel Extensions coming for C# 4.0 and Visual Studio 2010 [7]. No official white papers about the underlying mechanics of the Task Parallel Library have been published. Our current knowledge about the library is based on a web article and a Channel 9 developer presentation, both conducted by Daniel Moth, Senior Program Manager at Microsoft [20, 7]. From the article and presentations we have deducted that the Task library uses work-stealing as scheduling mechanism and from the examples given, it is clear that the library shares many of its general properties with the Java Fork/Join framework.

The library is described as an improvement to the `ThreadPool` library. `ThreadPool`s consists of a varying number of threads that are being assigned work either manually or through a producer-consumer data type and executes them concurrently on the threads assigned. The `ThreadPool`s are missing some useful functionality; they do not offer an effective scheduling model and they lack the ability to synchronize between assigned work chunks. To accommodate these requirements, the notion of a `Task` will be added in C#.

3.4.1 Library Specifics

`Task`s are realized through the means of `TaskManager`s which resemble the thread-groups in the Fork/Join framework for Java. A default manager is created for each application, and extra managers can be created explicitly as needed. The default number of threads assigned to each manager corresponds to the number of physical processors on the user machine. Listing 3.3 shows a compact way of creating the Fibonacci function with the new library.

```
public class TaskExample {
   static int fib(int number) {
      Future<int> result1 = new Future<int>((o) => fib(number - 1));
      result1.start();

      Future<int> result2 = new Future<int>((o) => fib(number - 2));
      result2.start();

      return result1.value + result2.value;
   }

   public static void Main() {
      Console.WriteLine("Answer: " + fib(35));
   }
}
```

Listing 3.3: Task Parallel Library example (Fibonacci). [20]

In this example, the `Future` class is used. The `Future` class is a generic result-value based class which wraps the task principle. A `Task` class also exists in the framework and it works similarly to the `FJTask` class in the Java Fork/Join framework. Creating a barrier before returning the result is not necessary as obtaining the value of the `Future` class is a blocking operation.

3.5 Google MapReduce

The MapReduce framework presents an alternative way of hiding the parallelization of problems. In the previously mentioned libraries, the main focus has been to start many instances of a specific task abstraction. In the MapReduce framework, this notion of splitting the work into tasks is taken one step further by requiring the user to do a more concise and elaborate mapping of the parts of each problem. [11]

The idea behind the framework is to look at each problem as an input consisting of a key/value pair. The programmer is required to define two operations for this input in order to solve the problem associated with the input; a `map` and a `reduce` function. The `map` function splits the value of the input into an intermediate set of key/value pairs. These pairs usually share a number of keys among them. The `reduce` function is called for each distinct key value after the mapping is complete. The reduction combines the values that have been derived in the mapping and returns the result.

3.5.1 Library Specifics

The current implementation of MapReduce is a library for C++. Listing 3.4 shows an example of how the mapping and reduction functions can be implemented. The example calculates the number of word occurrences in a given text. Firstly, the `Map` method is invoked and the given input text is split into words (lines 8-16). Each word is then emitted as a key/value pair, with the key being the word and the value being 1 (line 15). After the mapping has completed, the reduction can begin. The `Reduce` method is invoked once for every distinct key value, in this case each different word in the original input text. Upon invocation, the method iterates through each associated value and adds them together (lines 24-27). After adding the values, the result is emitted. The final result of this code example is a key/value list with words as keys and the number of occurrences as the values.

```
1  #include "mapreduce/mapreduce.h"
2
3  class WordCounter : public Mapper {
4     public:
5        virtual void Map(const MapInput& input) {
6           const string& text = input.value();
```

```
        const int n = text.size();
        for (int i = 0; i < n; ) {
            while ((i < n) && isspace(text[i]))
                i++;
            int start = i;
            while ((i < n) && !isspace(text[i]))
                i++;
            if (start < i)
                Emit(text.substr(start,i-start),"1");
        }
    }
};
REGISTER_MAPPER(WordCounter);

class Adder : public Reducer {
    virtual void Reduce(ReduceInput* input) {
        int64 value = 0;
        while (!input->done()) {
            value += StringToInt(input->value());
            input->NextValue();
        }
        Emit(IntToString(value));
    }
};
REGISTER_REDUCER(Adder);
```

Listing 3.4: Google MapReduce example (Word count) taken from [11].

3.5.2 Practical Use

The MapReduce framework requires the user to split the work in a significantly different way than the other frameworks presented. By requiring mapping and reduction functions, the framework is able to execute the written code on a cluster of computers rather than the cores associated to a single machine. One disadvantage of the framework is the strict manner in which the problem has to be defined, as many problems are not well suited for specification through mapping and reduction.

3.6 Choice of Subject

In this section, we make a decision about whether one of the related work projects is suitable to continue working on in this project. First of all, Cilk is not suitable, as it is implemented in C, which is not object-oriented. For both Java Fork/Join and C# Task Parallel Library, the source files are not available, which makes it impossible for us to continue working on the implementation. Google MapReduce is too low-level and problem-specific to use for a general-purpose language extension.

From the languages we have examined in this chapter, JCilk is the most promising to develop on, for several reasons. First of all, all of the JCilk source files are available. In addition, the Master's Thesis reports of I-Ting Angelina Lee and John Danaher give detailed descriptions of the system and the arguments for how it was created [10, 9]. Compared to working on the upcoming Java and C# features, with JCilk it is unlikely that we will be working in parallel with other developers. As such, continuing development on JCilk is a better choice than developing our own language extension, and this is what we do in this project.

An additional factor in choosing JCilk is that it is an extension rather than a library, as opposed to C# 4.0 and Java Fork/Join. This means that we will be exploring a different aspect of the field than the many researchers working at Sun and Microsoft. The main difference here is that an extension is able to introduce new language constructs and change language semantics, while a library must stay within the bounds of the language itself. Furthermore, an extension allows some freedom that a library cannot do, such as supporting method migration, which is not possible without a compilation pass of the program.

Chapter 4

JCilk

This chapter functions as our presentation of JCilk in its current public state, called JCilk-1, as of the two Master's Theses published on it [10, 9].

Firstly, a brief explanation of the new constructs in the JCilk language is given. This will be kept brief, as the constructs are actually rather simple in their function and usage, and are very similar to the Cilk constructs explained in 3.1. Additionally, this chapter contains a description of the JCilk compiler and of the runtime system. A list of strengths and weaknesses from Cilk is presented, and finally, this list is compared to JCilk.

4.1 JCilk Language Constructs

This section describes the constructs of the JCilk language. This is not meant as an exhaustive definition of the syntax and semantics of JCilk, but rather as an informative description of the keywords and their use.

The `cilk` Keyword

The most basic building block of JCilk is the `cilk` keyword, which indicates that a method can be spawned and can spawn other methods. A method marked by the `cilk` keyword is called a "‘Cilk method"’. Cilk methods can spawn other Cilk methods and call non-Cilk methods, but non-Cilk methods cannot spawn or even call Cilk methods. Line 1 of Listing 4.1 shows the `cilk` keyword in use.

The `spawn` Keyword

The `spawn` keyword is the construct that forks computation. A method call prefaced with the `spawn` keyword is created and computed as a separate task. `spawn` in JCilk functions exactly as in Cilk, with one exception. JCilk does not contain an explicit `inlet` mechanism, but there is a way to implicitly create inlets when using spawn with methods which return a value, by calling the method in an assignment, as shown in Line 3 of Listing 4.1. Line 4 shows the use of `spawn` without an implicit inlet.

Implicit inlets in JCilk act as `inlets` from Cilk, meaning that they run as separate tasks from both the spawned method and the calling method. Inlets act as a part of the calling method, and the assignment is guaranteed to be atomic with regards to other threads operating in the method.

```
1 public cilk testMethodA(){
2   int x = 0;
3   x += spawn testMethodC();
4   spawn testMethodB();
5   System.out.println(x);
6   sync;
7   System.out.println(x);
8 }
```

Listing 4.1: Example illustrating use of JCilk keywords.

The `sync` Keyword

The `sync` keyword is used to join computations. It blocks until all child computations are completed. Line 6 of Listing 4.1 shows the keyword in use. In Line 5 the printed number can be 0 or whatever value `x` has after `testMethodC` has finished executing. Since the computations have not been joined at this point, the printed value is nondeterministic. In Line 7, however, the `sync` keyword has been called, and the value printed will be the value that the implicit inlet of `testMethodC` has assigned it to be.

The Lack of the `abort` Keyword

The `abort` keyword from Cilk is noticeably absent in JCilk. The reason for this is that the exception mechanism in JCilk can perform the same function, as explained in Section 4.2 below. The `CilkAbort` exception is the JCilk equivalent of the Cilk `abort` keyword.

4.2 Exceptions in JCilk

Exceptions are an important part of Java, and JCilk extends the functionality of exceptions to work with the new concurrency model. Exceptions in Java are entirely sequential, and there is no semantics for concurrent exception handling, since an exception in Java never leaves the thread in which it is thrown. In JCilk, this functionality would be insufficient, since it would not allow for a spawned task to report exceptions back to the parent. Therefore, the exception mechanism has been expanded with the `cilk try` mechanism. Listing 4.2 illustrates the use of this mechanism, which closely mirrors the use of the Java `try` keyword.

```
public cilk testMethodD(){
  cilk try{
   spawn testMethodE();
  } catch(Exception e){
   //handle the exception.
  }
}
```

Listing 4.2: JCilk `cilk try` example.

When an exception is thrown in a tree of computations which may or may not be running in parallel, there is the issue of how to handle this. JCilk inherits Java's exception philosophy in that when an exception is thrown, computation is stopped and the exception is caught by the first dynamically enclosed `cilk try`-construct which catches the exception in question.

When an exception is thrown, all tasks in the same sub-tree of the computation tree are aborted. However, this abort does not happen synchronously, but rather semi-synchronously. Each task which is not currently running is aborted without problems. However, the ones which are running are not immediately aborted. Instead, they are aborted when they move to the next thread boundary after being notified of the abort. However, this does not guarantee that they will end up at the next thread boundary, since the exception may not have reached every thread in the tree by that time. Instead, computations are guaranteed to stop at *some* thread boundary, for every task in the sub-tree.

4.3 Compiler

This section describes the JCilk-1 compiler, which is the prototype compiler for JCilk created primarily by I-Ting Angelina Lee [10]. The compiler is largely based on the Cilk-5 [21] compiler [10]. In particular, the work-first principle and work-stealing algorithm was inherited, as well as the idea of having two clones, a fast and a slow one, for every Cilk method.

The implementation deviates, however, when it comes to the compilation process. Where the Cilk-5 compiler compiles from Cilk to C, the JCilk-1 compiler compiles first from JCilk to GoJava, an intermediate language created especially for this purpose, and then from GoJava to Java bytecode. The compiler from GoJava to Java bytecode is called Jgo.

4.3.1 The work-first principle

The work-first principle [21] states:

> *Minimize the scheduling overhead borne by the work of a computation. Specifically, move overheads out of the work and onto the critical path.*

This principle is the result of a proof from [21], and it is dominant in both the Cilk-5 and the JCilk-1 implementations. The principle means that the optimization should focus on minimizing the work overhead, even at the cost of a larger overhead to the critical path of the program.

4.3.2 The two-clone strategy

The compiler uses a two-clone implementation strategy. This strategy is designed to reduce the work overhead, which is what the work-first principle dictates. With this strategy, the compiler creates two clones of each Cilk method, a fast clone, and a slow clone. The fast clone is close to the Java code and limits the use of code to support parallelism. The slow clone has full parallel support.

The fast clone is used for methods which have not been stolen via the work-stealing algorithm, and since it contains less overhead for parallel support, it runs faster than the slow clone. The code compiled from the `sync` keyword is removed from fast clones since they are not parallel, and the `spawn` keyword simply puts the method it is executing into the work-stealing deque and then executes the method call of the `spawn` normally. When the method returns, if the method has not been stolen while the spawned method was executing, the worker simply removes the method from the deque and continues executing as it would in a sequential program.

The slow clone is only used for methods which have been stolen and are executed in parallel, and thus the slow clone has full support for parallel execution. Tasks can be stolen at all thread boundaries, and slow clones need to support this continuation, since Java itself does not support method migration. The way that continuations are supported is that each slow clone method starts with a `switch` which examines a program counter, telling it which number it needs to match. Each segment of the `switch` contains a `goto` which moves execution to the correct place in the code.

An example of the two-clone compilation strategy can be found in Appendix B. The appendix contains the source code of the `Fib` method, as well as simplified versions of the slow and fast clones of the compiled version of the program. All three methods are taken from John Danaher's Master's Thesis [9, Section 3.1].

4.3.3 JCilk-1's Two-Stage Compilation Process

The JCilk-1 compiler uses two stages to compile JCilk code. First, it compiles JCilk into GoJava, which is an intermediate language created for this compiler. Secondly, it compiles from GoJava directly to Java bytecode. The compiler for the first stage uses Polyglot [22], a compiler toolkit for creating Java extensions, which is further outlined in 6.3, while the second stage uses a modified version of the GNU Compiler for Java, an open source Java compiler.

The reason for this intermediary stage is that the slow clones use `goto` statements, which are not present in Java. The GoJava intermediary language was created to support this, and GoJava is exactly like Java except that it includes `goto` statements.

4.3.4 Example: Compiling a `spawn`

This section explains what happens in a spawn statement. At each spawn, the worker needs to save the program state of the parent method into the deque so that it can be stolen, and run the child method. When the child method returns, the program needs to check if the parent has been stolen while the worker was executing the child. This is where a thief would start execution, so after the steal check, there needs to be a label so that the thief worker can jump to that position in the program. Thus, a compiled `spawn` consists of the following steps [10, Section 3.3]:

1. Save local state into a Frame (See Section 4.4.2) and put that frame onto the deque of the worker.

2. Call and execute the child method.

3. Confirm that the parent was not stolen while the worker was executing the child. In case it was stolen, pass the value of the child to the parent via the runtime system.

4. Present a label for other workers to continue the method in case the method is stolen while the child is being executed.

4.4 Runtime System

The JCilk-1 runtime system uses work-stealing (Described in Section 3.1.1) to coordinate work between the `Worker` objects. `Worker` is one of three major classes in the runtime system, and each `Worker` object represents a `Thread` in the system. The second major class is `Frame`, which represents a task in the system and maintains a shadow of the call stack to allow work-stealing, and the third major class is `Closure`, which represents `Frame`s that have been stolen and allow children to return values to their parents.

4.4.1 `Worker`

Each `Worker` runs in its own Java `Thread`, and these are the only threads running in the runtime system. As such, the `Worker`s perform all of the work in JCilk programs. Each Worker object runs a loop of: find work, do the work, repeat. This work is the tasks in the system, and the first task is seeded into the system from the `main` method of the user code. This means that when the program starts, only one `Worker` will have work to do, and a JCilk program will always start out as single-threaded. Once more work is created by `spawn` calls in the user code, some of this work can be stolen by other `Worker`s, and thus the program begins to execute concurrently.

When a Worker encounters a `spawn` statement, it will first execute the child method, and then return to the parent method to continue execution normally. If a steal occurs when a thread has only the parent method in the deque, the parent method will be stolen. This means that when the `Worker` finishes executing the child method, it will have no work left, and will have to try to steal new work itself.

In case a `Worker` reaches a `sync` statement, it will check if all of the children of the current task have finished execution. If this is the case, the `Worker` passes the `sync` statement, but if this is not the case, the task will have to be suspended until all children have finished.

4.4.2 `Frame`

Whenever a task is stolen, the method of that task needs to be migrated to another thread. Since Java does not support method migration between threads, the runtime system needs to find a way to work around this. Calling the method itself is not too difficult, but there is the issue of the state of that method. This is what the `Frame` is for. Each `Frame` contains the local state for a given task, and represents that task on the deque. When a task is stolen, the `Frame` is taken from the corresponding deque by the stealing `Worker`.

When a Cilk method is compiled, a corresponding subclass of `CilkFrame` is created. This subclass contains a copy of every local variable of the method as an instance field. The subclass also overrides an abstract `cilkRun` method to call its corresponding method. This allows a thief to run the correct method and have access to the relevant local state.

4.4.3 `Closure`

Closures represent the returning end of methods. A closure has information about where to return values to, such that when a task finishes, it can find its parent to return a value, if any. Closures also track the number of outstanding children a task has, in order to know when it can proceed from `sync` statements.

4.5 Cilk

This section reviews the Cilk language in order to determine some of its advantages and disadvantages with regards to becoming a widespread language for general-purpose programming with good support for concurrency. Each of these advantages and disadvantages is briefly described, and finally a summary is given. The purpose for this is to create a basis for comparing Cilk and JCilk. The list of strengths and weaknesses in Cilk is then compared and contrasted with JCilk.

Tasks/Work-Stealing

For a description of how work-stealing and tasks work in Cilk, see Section 3.1.1. The concurrency model of tasks and work-stealing provides a model for dynamically assigning work to threads, which means that programs do not need to be optimized to a certain number of processors. Tasks and work-stealing form the foundation for the efficient concurrency in Cilk, and this model has been used in many languages and libraries since its adoption in Cilk, for example in Fortress [23] and C# 4.0 [7].

Provably Good Scheduler

The designers of Cilk have proven that a program running on the Cilk scheduler on an arbitrary number of processors will complete within a constant factor of the 'ideal' time it would take to run the program on an omniscient scheduler. The proof, as well as details on the implementation, can be found in [21].

Efficient Parallelism

A major strength of Cilk is that it is easy to see the impact that its concurrency model and implementation has on programs. This was tested in [1, Section 4.2], where Cilk was compared to C# and Scala and was found to scale much better with the number of processors than the other languages. Simply put, Cilk is a very efficient language for concurrent programs.

Thread Atomicity

Cilk guarantees atomicity between threads in the same function; that is, they never run simultaneously, and always execute in a sequence. This removes some of the need to be concerned about race conditions. However, race conditions may still occur between different functions, as the guarantee only applies within a function.

Speculative Computation

In some programs, it is beneficial to start up speculative branches of computation which can be aborted if they are found to be unnecessary. For example, the n-queens problem lends itself well to parallelized solutions which utilize speculative computation. A solution to the n-queens problem is a placement of n queens on an n by n chess board such that no queen could directly attack another, meaning that no two queens occupy the same row, column or diagonal.

Going through each row, the algorithm can spawn new branches of computation for every legal configuration. For example, in the first row of an 8-queens puzzle, simply branch computation 8 times, once for each possibility. Each of these branches will then branch out during the second row for each legal placement of the second queen, and so on. Now, when a branch of computation finds a solution, all other branches can be safely aborted, as the algorithm has already found what was needed.

Cilk supports speculative computation by using the `inlet` and `abort` mechanisms. An `inlet` is a local procedure which can be supplied for a `spawn`. The `inlet` will be run after the spawned procedure returns. Using `abort` in an `inlet` is an effective way to cancel all remaining subcomputations once they are found to be unnecessary. As an additional feature of inlets, they are guaranteed to execute atomically with regards to the spawning procedure, so there is no risk of race conditions in this respect.

Comparing the use of `inlet` and `abort` to the use of `spawn` and `sync`, however, they are not as elegant and simple. While the fact that Cilk supports speculative computation is good, and the functionality is effective at what it does, there is room for improvement.

Closely Based on C

And now we reach the major weakness of Cilk. Cilk is an extension of C, which means that no matter how good the constructs in Cilk are, the language still suffers many of the same weaknesses as C. This is a large hindrance, as most OO programmers are not willing to make the trade-off between Cilk and a language like Java or C#. While the gain in the parallel constructs may be big, the loss in terms of having to deal with things like pointers and memory allocations is simply too big, and as a consequence, Cilk has not reached mainstream use. It has, however, seen some use within the field of high-performance computing.

Summary

The concurrency model in Cilk is elegant, with the exception of the `inlet`/`abort` mechanism, and the implementation is highly efficient. The main weakness of Cilk is that it is based on C. The following section compares the list of strengths and weaknesses presented here to JCilk, in order to reveal whether or not the migration from C to Java has successfully retained the strengths while improving upon the weak areas of the language.

4.6 Cilk vs JCilk

Now that we have described both Cilk and JCilk, we compare the two in order to see how JCilk measures up against its predecessor. This is done step by step, and each step is a Cilk strength or weakness mentioned in Section 4.5.

Tasks/Work-Stealing

JCilk closely mimics the algorithm for scheduling used in Cilk. This concurrency model is the whole reason for creating JCilk, so it is obvious that it would be kept more or less intact.

Provably Good Scheduler

Since the same algorithm is used, JCilk's scheduling system is algorithmically provably good as well. However, there are some caveats here, since the performance of JCilk is suboptimal, as mentioned in both master's theses written about it [9, Chapter 8][10, Chapter 6]. From what we can deduce at this point, however, the problem is not with the algorithm but with the implementation. Whether or not the implementation can be made efficient remains to be seen, however. Since we conclude that the algorithm of the scheduler is not the problem, this point remains a strength in JCilk.

Efficient Parallelism

This point must be seen as the greatest weakness of JCilk, since it really is not efficient. Danaher concludes in his thesis that:

> *Until the efficiency can be improved, however elegant the exception mechanism might be, the language itself will not be useful.* [9, P 91]

As such, we see this point as the greatest weakness of JCilk. Since the efficient parallelism is supposed to be one of the great strengths of moving to a parallel language, this needs to be resolved.

Thread Atomicity

This constraint is upheld in JCilk as it is in Cilk. Closures ensure that threads within the same method operate atomically in relation to each other, and this point can be seen as fully fulfilled.

Speculative Computation

Cilk has support for speculative computation, but was found to be inelegant and clumsy. JCilk uses a wholly different mechanism for aborting speculative computations, which is the `CilkAbort` exception. The designers of JCilk think that this is a more elegant solution. It does work intuitively as opposed to the `inlet`/`abort` mechanisms of Cilk. However, this use of the exception mechanism in Java is not what was intended for it [24, PP 242]. It might work, but it goes against the design principles of Java. However, it is definitely more elegant and efficient than making extensive use of `inlet` and `abort`.

Closely Based on C

This is where JCilk makes a major leap forward. C was a reasonable choice for Cilk in 1995, but currently, much language development is focused around the major OOP languages Java and C#. Since Java and C# are so similar, the concept can be transferred to C# if desired.

In summary, JCilk has the advantage of using Java as its base as compared to the C base of Cilk, and has a different mechanism for speculative computation. However, it takes a big loss in performance. The next chapter will take a look at ideas for improving JCilk, and analyze them with regards to a potential implementation.

Chapter 5

JCilk Status Analysis

This chapter presents some of the possible improvements that can be made to the JCilk system in its current state and selects the improvements that will be implemented or analyzed further in this report. There are two origins of improvement ideas; those originating from the reports written by I-Ting Angelina Lee and John Danaher, and those proposed by our project group. Ideas originating from the thesis reports are described from this project groups' point of view and not necessarily faithful to the opinions put forth in the thesis reports. When an idea stems from one of these thesis reports, it is clearly marked with citations.

The improvements we choose to implement or analyze further are selected with the main goal of easing the future adoption of the JCilk framework. This means that we make a subjective evaluation of each improvement concept and decide which specific changes are most likely to increase the chances of a wider adoption of the framework. There are two main factors that need to be improved in the JCilk framework at its current state:

Performance The currently most significant shortcoming in the JCilk framework is the performance. As JCilk is a framework aimed at making use of all processors on a machine, it is very important that the performance gain is close to the gain that would be received by coding the parallelism without the task abstraction. Even though the tests performed by Danaher [9, Table 5.2] have shown that there is a speedup, there is still a long way to go before an acceptable degree of efficiency is achieved, as Danaher also mentions.

Installation and Use The current installation process of the library is tedious and the required disk drive capacity is very high compared to other similar frameworks. In order to install the compiler, it is necessary to download a number of modules to make a custom compilation of the GNU Java Compiler which handles the compilation of

GoJava files to bytecode. The modified GJC compiler and the other files associated with JCilk require a total of more than 1,5 GB of disk space when installed.

5.1 Connecting JCilk and Java

A problem with the current implementation of JCilk is that Java methods cannot call Cilk methods [9, Section 7.1][10, Chapter 6]. This property is similar to the Cilk language in that a C function cannot call Cilk functions. However, it would be desirable in JCilk to be able to do this, since this would enable a lot of interactions which currently are not possible, such as combining the standard library classes with JCilk classes. The problem stems from the fact that Java methods can not be suspended or moved between threads, which is necessary in order to make the JCilk scheduler work.

An example of a situation where this feature is useful is when making parallel extensions to already existing libraries. There already exists built-in sorting algorithms for enumerable structures such as arrays. It is desirable to create new classes extending the already existing types and overriding the sort methods using JCilk abstractions.

This restriction is not easily changed as it requires a complete rewrite of the JCilk architecture. As a solution to this problem, Danaher suggests rewriting the JCilk framework into an implementation of the `Executor` interface as a possibility [9, Section 7.1]. We have decided not to dig deeper into this problem due to two main reasons; removing this limitation requires a full re-write of the architecture which, rather than improving JCilk, induces a complete reimplementation, and secondly, it will change the idea behind JCilk as a language into a library implementation which goes against one the reasons for choosing JCilk, which is to support task extensions over library implementations as described in Section 3.6.

5.2 Modifying the Two-Stage Compilation Approach

The JCilk compiler architecture currently consists of two steps, where the JCilk source initially is compiled into GoJava code, followed by a compilation into bytecode using a modified version of the GNU Java Compiler as described in Section 4.3.3. This architectural choice stems from the fact that the compilation process is highly inspired by the one used in Cilk. Since the continuation mechanism in Cilk is made using `goto` statements, it was also decided to initially compile the JCilk language into an extension of Java supporting `goto` statements, GoJava. When defining the use of the `goto` statements in this new language, Angelina phrases the following assurance of the use of the `goto` statement, to assure correspondence between the resulting bytecode of Jgo and the allowed use of jump statements in Java bytecode:

"JCilk only uses `goto` statements to make local jumps, meaning jumps within the same method. Second, since the JCilk compiler inserts `goto` statements at the Java source-code level, the `goto` bytecode can only be inserted between two Java statements and never in the middle of a statement."

With these restrictions to the jump operations in mind it is possible to create a simulation of `goto` statements in Java code rather than at the bytecode level, as each conditional and loop-based construct in Java can be simulated with other Java code. The only concern when doing such a simulation is whether there is a significant performance loss associated with this change, as it is necessary to use more jump instructions than when compiling directly into bytecode.

By removing the need to compile into GoJava, JCilk can be compiled directly into Java code and the Java-to-bytecode step becomes optional, since any ordinary Java compiler can perform this step. The result of making this change is a simpler and more transparent compilation process. Detailed design and implementation of this improvement is described in Section 7 and a test of its performance compared to the `goto` solution is described in Section 8.

5.3 Using Modern Java Abstractions

JCilk was developed during 2004 and 2005. This was during the transition between Java 1.4 and 1.5. Some of the new concurrency mechanisms added in Java 1.5 and 1.6 can possibly improve the performance of JCilk. The following major concurrency additions have been made in Java 1.5 and 1.6:

Atomic The classes defined in `java.util.concurrent.Atomic` extend the notion of `volatile` variables. A `volatile` variable in Java is never cached thread-locally and its access is always automatically enclosed by a `synchronized` block. The atomic classes adds the possibility of reading and changing a variable in one single operation. In effect, hardware operations such as the compare-and-set instructions are exposed, which removes the need to synchronize Java code in a number of situations.

Locks The classes defined in `java.util.concurrent.Locks` offer a more flexible locking system than `synchronized` blocks provide. Important features include; the ability to specify a timeout when acquiring a lock, and to interrupt threads that are waiting on a lock and non-nested scoped locks.

Executors The `java.util.concurrent.Executor` interface creates a means to decouple the submission of `runnable` tasks and how they should be executed. This interface and its following utility classes makes it possible to define a scheduling mechanism for the submitted assignments.

The Atomic and Lock additions facilitate better performance on the cost of code simplicity, while the Executors bring a more flexible and general abstraction to thread work management. The currently biggest hindrance for a wider JCilk adoption is its performance [9, Chapter 8]. Adopting the `Executor` interface in JCilk is a structural improvement rather than facilitating any specific performance enhancements, thus making it a less interesting focal point. The new Atomic and Lock additions do, however, bring a potential improvement in performance, as they expose new functionality. Danaher has made an alternative implementation of the deque with Atomics to test whether it gives a noticeable performance increase [9, PP 77-78]. The result of implementing atomics is inconclusive, as some performance gain was achieved when testing with the Fibonacci function, while there was a performance loss when testing with the 14-Queens problem. Adding support for the new locks introduced in Java 1.5 can possibly increase performance of the system, but given the experience Danaher got when implementing atomics, the performance gain of changing the system to support relatively small improvements like locks are presumably not worth prioritizing over more extensive changes to the system.

5.4 Runtime System Performance

In order to see where we can improve the performance of the system, we start by examining the results of the performance test run by Danaher. All three tables in this section (Tables 5.1, 5.2, and 5.3) reflect data from John Danaher's JCilk performance test, which can be found at [9, Chapter 5].

Program	**Time/s**
Fib in Java	1.9
Fib in JCilk, 1 worker	58.9
Fib in JCilk, 2 workers	37.8
Fib in JCilk, 3 workers	41.1
Fib in JCilk, 4 workers	29.0
All-Queens in Java	60.3
All-Queens in JCilk, 1 worker	69.6
All-Queens in JCilk, 2 workers	34.9
All-Queens in JCilk, 3 workers	24.2
All-Queens in JCilk, 4 workers	19.0

Table 5.1: The overall results of the performance test [9, Table 5.1].

Table 5.1 shows the running times of the `Fib` and `Queens` programs in their serial Java elision (The Java code that comes from removing all JCilk keywords from the JCilk programs), as well as the running times of the JCilk programs with one, two, three and four workers. The `Fib` program calculates the nth Fibonacci number, in this case the 40th, and this program involves very little calculation. This program is meant to test the overhead introduced by the parallel computation model, as the program is very fast in its serial version. The `Queens` program calculates all solutions to the n-Queens problem for a given n, in this case 14. This requires a lot more calculation compared to the number of recursive calls made, and this program is used to test the speedup gained by using the parallel computation model of JCilk.

As expected, `Fib` runs slower in JCilk than in its serial version, but it runs quite a lot slower, which exposes a big performance loss in JCilk when spawning a lot of tasks. When running on a single processor, JCilk slows the program by a factor of 31 (58.9/1.9), which is quite problematic. The `Queens` program sees a slowdown as well, but only by a factor of 1.15 (69.6/60.3). This shows that when a number of calculations are made in each recursive call, the overhead becomes a less significant factor. Indeed, the `Queens` program only needs to run on two processors to be faster than the serial version. In comparison, we do not have access to a computer with enough cores to make the `Fib` program run faster in parallel. In fact, there is no guarantee that it will be faster even with access to an arbitrarily high number of processors, due to the significant overhead in the JCilk system compared to the extremely low number of calculations in the program.

	TS/T1	T1/T2	T1/T3	T1/T4
Fib	0.032	1.56	1.43	2.03
Queens	0.867	1.99	2.88	3.66

Table 5.2: The speedup of the `Fib` and `Queens` programs (running with n `= 40` and n `= 14`, respectively). TS is the serial elision of the program, and T1-T4 correspond to the JCilk program with 1-4 workers. [9, Table 5.2].

Table 5.2 shows the speedup of introducing more workers to the JCilk programs, thus using more cores on the computer. This table shows that the `Queens` program scales very well to multiple cores, with a speedup of 1.99 with the second core and 3.66 with all four cores, compared to the theoretical ideal gains of 2.00 and 4.00. This gain is comparable to the numbers we got when testing the speedup of Cilk in our SW9 project [1, Chapter 5]. For `Fib`, the speedup is much lower, however, achieving only a speedup of 2.03 when using four cores.

The preliminary test showed that there was a lot of overhead in JCilk, and the final test was made to analyze the sources of this overhead. Table 5.3 shows the running time of the `Fib` program divided into categories for what the time is spent on. By far, the most time is spent on frame allocation and frame deque usage. Frame allocation time is spent on creating and initializing the objects for each frame, while deque usage time is spent on *synchronization*. Both of these are overheads of the parallel version and are not present in the serial elision. As such, we can conclude that the overhead of JCilk lies in these two categories.

Modification	time/s
Serial Fibonacci	1.01
Runtime startup overhead	0.11
Frame allocation	25.7
Frame deque usage	13.4

Table 5.3: Performance running `Fib` with one worker [9, Table 5.3].

5.4.1 Frame Allocation

A weakness of Java performance-wise is the fact that every Java object starts off with 8 bytes spent on default functionality which may or may not be useful to the program [25]. This is not a problem in programs where performance is not critical, and most Java programs create a comparatively small number of objects (say, less than 10,000). The `spawn` keyword in JCilk encourages recursion, however, which results in the creation of many objects. Recursively calculating the 40th Fibonacci number using the `Fib` program creates over 300,000,000 objects [10, Chapter 6], which is more than 2,400,000,000 bytes of wasted allocation, since none of the data for the default functions are used. Additionally, the compiled `Fib` program has at least 20 more bytes of local data to keep track of the program counter and local variables used, so running the `Fib` program causes more than 7Gb ($(8 + 20) \cdot 3 \cdot 10^6$) of allocation to take place.

There are at least two ways to solve or at least alleviate this overhead problem. One solution would be to save and reuse the objects created for recursive calls. The other solution is to forego using objects to represent the frames altogether. Both of these methods are quite messy and obscure the actual intention of the system, though this all takes place at the runtime and compiler levels, which means that the change is invisible to the user, except for the performance change.

Reusing objects can be done by each worker maintaining an array with used objects which it can reuse later. This would mean that the full number of objects is created while traversing to the leaf nodes of the recursion tree, but once the first leaf node has been reached, no more new objects will be allocated. This means that instead of the number of allocated objects being relative to the number of nodes in the tree, it will be relative to the height of the tree. This takes an exponential number of objects created and turns it into a linear number (From 2^h (simplified) to h), which is a significant performance gain.

Avoiding the use of objects is more complicated, and would involve using a separate memory manager which uses array of primitive non-object variables such as integers in order to avoid the unnecessary allocation. This would mean that no frame allocation takes place at all, which is potentially a large performance gain. However, some allocation will still have to take place, since otherwise there would not be any way to store the information in memory.

There is a lot to gain performance-wise in this area, and both solutions have the potential to increase the performance of JCilk significantly.

5.4.2 Frame Deque Usage

There is a big problem here, in that it is hard to tell where the performance loss really comes from. The source of the overhead is the Java synchronization primitives, which is as low-level as Danaher goes in his thesis. There is the possibility to create a system for managing synchronization which does not use the Java primitives, but there is no guarantee that it will be faster. Since frame allocation is a bigger overhead, and has more readily available solutions, the synchronization problem would not be a good place to start reducing the overhead.

5.5 Summary and Selection

As this chapter shows, there are several areas in which JCilk can be improved. For our implementation, we choose to do two things, remove the intermediary language GoJava from the compilation process, and minimize the overhead involved in frame allocation. The next two chapters describe our work on these improvements.

Chapter 6

Lowering Frame Allocation Overhead

This chapter describes our work towards lowering the overhead involved in frame allocation in JCilk. We begin by describing the ideas with a higher level of detail, then proceed to describe the design, and finally the implementation.

6.1 Ideas

The frame allocation overhead stems from one very simple line of code that is present in every JCilk fast clone. An example from Line 4 of Listing B.2 in Appendix B:

```
Fib_fib_frame thisFrame = new Fib_fib_frame (n, 0, returnEntry );
```

The `new` keyword in the line of code is the costly one, since it causes the allocation of a new object in main memory. This is done for every Cilk method in a program, such that for every time a Cilk method is run, a new object is allocated. For `Fib(40)`, this means allocating more than 300,000,000 objects [10, Chapter 6]. Compared to the serial Java program which creates no objects, this is a large overhead, which is the largest contribution to the slowdown by a factor of about 25 that occurs in JCilk compared to the serial version.

We propose two different approaches to reducing this overhead. One solution is based on not allocating new objects at all, and instead using a separate memory manager which maintains large arrays which contain the data. This means that the data is stored as primitive variables rather than objects, and these primitive variables are allocated in large chunks at a time rather than being allocated as needed. The primitive variables will still need to be stored in objects, but each object can contain a large number of primitive variables. This means that much more data is stored in each object, which reduces the memory overhead for the default object data considerably. Additionally, garbage collection will be faster in this model, since there will be a factor 10,000 less objects to administrate. On the topic of garbage collection, this solution also requires an efficient way of knowing which variables are in use and which are free, since there is otherwise no way of knowing which to use. One way to create such a garbage collector, is to keep track of how many elements in one storage unit (the abstraction that keeps the data for e.g. 10,000 frames) that are no longer being used. When the number of unused elements reaches the capacity of the storage unit, it can be garbage collected.

A simple preliminary test showed an improvement of about a factor 30 in the time used for allocation with this solution (See Appendix C). This was a very simple test, however, and an actual implementation of the solution in JCilk will show if the solution works in practice. Also note that this test only considers allocation, and not the extra overhead that will be involved in referencing the data, since this will be more complex than simply saving a local pointer to an object. However, a test of the increased pointer overhead shows that the referencing overhead is insignificant. The code for this test can be found in Appendix D.

The other approach is based on reusing frames rather than letting the used frames be garbage collected and creating new ones. This involves each worker keeping an array of references to frames that represent data that is no longer needed. Whenever the worker needs a new frame, instead of creating that frame, the worker will check its array for a used frame it can reuse. Each frame allocated represents a method call, and for each method the frames have the same type. This means that a recursive algorithm like `Fib` will create many frames with the same type. This solution proposes to reuse those frames such that less frames will be created. For Fib(40), over 300 million frames are normally created. With frame reuse, this number will be greatly reduced. The number created with frame reuse is $p * n$ where p is the number of workers and n is the problem size. Since the number of workers is constant in any program, the overhead can be seen as $c * n$, or $O(n)$, where the original overhead was $O(2^n)$. Even assuming a very high number of processors, this model represents a vast improvement on the allocation overhead for recursive algorithms.

Most programs that can be efficiently parallelized by JCilk will have some recursion, which means that this model works on most JCilk programs. The reason that JCilk lends itself to recursion is that calling methods is the only way to have programs run in parallel, and calling them recursively is the only way to create dynamically parallel programs. This mechanism works best for simple recursion where the same method keeps calling itself, however.

There is the possibility of programs using *mutual recursion*, which is a more complex form of recursion where each recursive call can be made to one of several methods. In these cases, the reuse array needs to contain more than one type of object at a time. An example of the use of mutual recursion is in recursive descent parsers. Moreover, recursive descent parsers are an example of a case where mutual recursion grants desirable properties to the program, as the structure of the program closely mirrors the grammar it recognizes. We have concluded from later discussion that this is not a complete explanation of the subject,

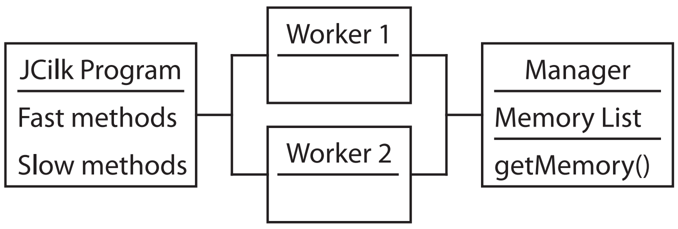


Figure 6.1: Architecture of the memory manager.

as described in Section 9.3.

6.1.1 Memory Manager

The idea of the memory manager is to use primitive Java variables to replace the objects used in the original JCilk implementation. Each frame is represented by a number of managed primitive variables placed in the memory manager rather than an object containing the data. Figure 6.1 shows the architecture for the memory manager, where there is one central memory manager object and then a number of workers which all use the memory manager to access the data for their frames.

Using such a low-level memory management mechanism requires the creation of algorithms which manage the variables properly, such that mutual exclusion is guaranteed and overhead is minimized. The problem is that each type of frame will require a variable amount of data, so statically sized blocks are not efficient. This all makes the memory manager class very complicated, and low-level programming like this is not something Java handles very well. Contrast this with the simple idea of reusing objects, which is the same concept as the memory manager, just on a different level, and we realized during development that this solution is inferior to the frame reuse system.

The memory manager reduces the amount of object metadata allocated, the 8 bytes of default data mentioned previously, by a large constant, for example 10,000. This reduces the amount of allocation overhead significantly. The object reuse system, however, goes one step further, and reduces the amount of *total* data allocated, by reusing not only the metadata but all of the allocated variables. For algorithms like `Fib` which derive recursively into a tree, this means that the amount of memory allocation that takes place is reduced from an exponential amount to a linear amount, which is a logarithmic order of improvement. This means that while the memory manager performs a constant reduction in only the unwanted memory allocation, the reuse system performs a logarithmic reduction in the total allocation, by far a larger improvement for those cases where the reuse system works (more about this in Section 9.3).

6.1.2 Frame Reuse Design

This section describes the design of the frame reuse system we have created for JCilk. Figure 6.2 shows the changed classes in the design. There are three changes here. The first is the frame storage `CilkFrameReUseArray`, in the `Worker` class, while the second change is the new `reConstruct` method in the `CilkFrame` class, which each subclass also implements. Finally, two methods have been implemented into the `Worker` class, one for storing frames for reuse and another for retrieving reusable frames.

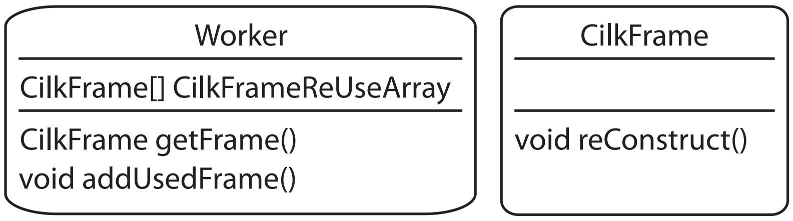


Figure 6.2: Class diagram illustrating the new local variable and the new methods in the `Worker` and `CilkFrame` classes.

Frame reuse occurs locally on `Worker`s such that there is no additional synchronization in the system. Essentially, when a `Worker` has finished using a frame, it puts it in the storage for reuse. When a Worker is going to create a new frame object, it first checks if there is an appropriate frame object in storage to reuse, and if there is, it will not create a new frame object but reuse the existing one instead.

6.2 Frame Reuse Implementation

The implementation process has been split into two phases. The first phase is exploratory, and consists of implementing the idea fully in the runtime system, but not in the compiler. Instead, we generate an example of the `Fib` program file 'by hand' as it is supposed to look after compilation. This gives us an opportunity to both test out the idea before fully implementing it, and to determine how the optimal end code should look before starting on the compiler modifications.

The second phase consists of implementing the system in the compiler as well. Figure 6.3 shows the process, and the versions of each part of the system in each step. Another advantage of this process is that there is a real product of implementation after the first phase, which is useful if there are problems during the second phase which mean that implementation is not finished. In this case, there will still a product to test at the end.

The first thing to determine, then, is what the changes to the runtime system consist of. First of all, there needs to be a storage for frames in the `Worker` class. This is where frames

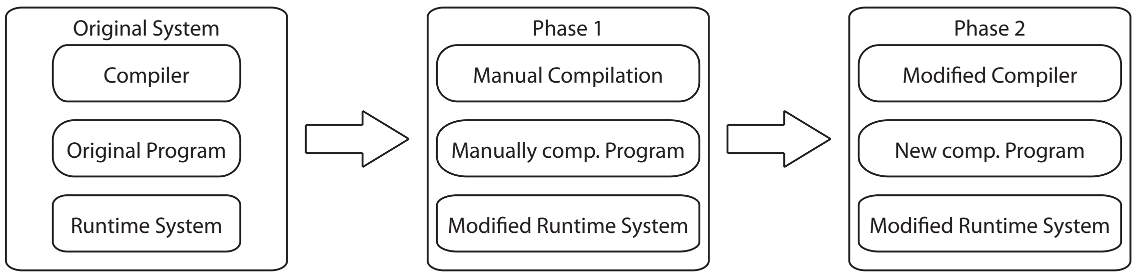


Figure 6.3: Diagram showing the states of the compiler, program and runtime system in each phase of our implementation process.

that are available for reuse are stored. Secondly, we need methods for storing and retrieving frames for reuse. Listing 6.1 shows the code for the frame storage and for saving frames for reuse, while Listing 6.2 shows the method for checking for reusable frames. Lines 1 and 2 show the local variables used to maintain frame storage. `top` is a counter used to indicate where in the `CilkFrameReUseArray` array the newest frame to be reused is stored. The frame storage works as a stack, which means that the object to be reused will always be the newest one added to the array. This is because JCilk encourages recursion, and this stack storage mechanism supports recursion best, as during a recursive algorithm, any frames saved before the algorithm is complete will be useless until it finishes. Lines 4-7 show the method for storing frames for reuse. The `top` counter is incremented in Line 5, and Line 6 stores the frame in the new `top` position in the array. Since the frame storage works within a single `Worker`, and since each `Worker` is single-threaded, there is no need to consider synchronization here.

```
1  private int top = 0;
2  private CilkFrame[] CilkFrameReUseArray = new CilkFrame[Size];
3
4  public void addUsedFrame(CilkFrame frame) {
5     top = (top + 1) % Size;
6     CilkFrameReUseArray[top] = frame;
7  }
```

Listing 6.1: Storage array for frame reuse as well as the method for storing frames.

Listing 6.2 shows the `getFrame` method. This method is called before creating a new frame to represent a method call, to check if there is an existing frame that can be reused instead. The `if`-statement in Lines 2-3 checks if the top frame is not `null`, meaning that the reuse array is not empty, and verifies that the top element is of the correct type. This means that Lines 4-8 are run when there is a frame object of the correct type on top of the frame reuse stack. Line 4 stores the frame to be reused in a temporary local variable, and Line 5 removes the reference to the frame from the stack. Lines 6 and 7 update the `top` counter, and finally Line 8 returns the frame to be reused. This covers the case where the top of the stack contains a frame of the correct type for reuse, and this is indeed the only case where reuse occurs. Lines 9-14 cover the case where the stack is not empty but the top frame is not of the right type, in which case the top frame is dropped, the `top` counter updated, and the method returns `null` to indicate that there is no frame to reuse. Line 15 covers the last case where the stack is empty, and again `null` is returned to indicate the lack of a frame to reuse.

```
 1  public CilkFrame getFrame(Class<? extends CilkFrame> class1) {
 2     if (CilkFrameReUseArray[top] != null
 3           && CilkFrameReUseArray[top].getClass() == class1) {
 4        CilkFrame temp = CilkFrameReUseArray[top];
 5        CilkFrameReUseArray[top] = null;
 6        top--;
 7        if(top==-1)top = Size-1;
 8        return temp;
 9     } else if (CilkFrameReUseArray[top] != null){
10        CilkFrameReUseArray[top] = null;
11        top--;
12        if(top==-1)top = Size-1;
13        return null;
14     }
15        return null;
16     }
```

Listing 6.2: The method for retrieving frames from storage.

This wraps up the functionality for object reuse in the runtime system. All remaining functionality is found in the compiler, which means that we now move on to creating the new compiled `Fib` class by hand. First of all, we determine which changes are necessary. There are three groups of changes to the compiled classes that need to be implemented for the system to work, and they are as follows:

- Used frames need to be saved in the `Worker` to enable reuse. This is done by calling the `addUsedFrame` method shown in Listing 6.1.
- When a new frame object would be created, instead check for a reusable frame first using the `getFrame` method shown in Listing 6.2. If that check fails, create a new frame. If it succeeds, reuse the object returned by calling its `reConstruct` method.
- Implement the `reConstruct` method for each frame class, to enable reconstruction of old objects.

Saving Frames for Reuse

To ensure that all used frames are saved for reuse, we need to save the frame using the `addUsedFrame` method before each `return`. The following line of code needs to be inserted before each `return`:

```
_cilkWorker.addUsedFrame(_cilkFrame);
```

Note that even though the `return` may use some of the data stored in the frame that has been stored for reuse, this is not a problem since it cannot be reused until the `Worker` proceeds to the next `spawn` to create a new method call and thus a new frame, which can not happen before the `return` is complete. Again the issue of synchronization is avoided by using separate, local frame stacks for each `Worker`.

Check for Frames to Reuse

This step is done when a new frame would be created. This is one of the initialization steps that happen at the start of each Cilk method. Listing 6.3 shows the code that represents this. Line 1 calls the `getFrame` method to retrieve an applicable frame for reuse. Lines 2-3 create a new frame if the `getFrame` call did not provide one, and Line 5 calls the `reConstruct` method to simulate the construction of a new frame object in the case where a reusable object was found.

```
1 Fib_calc_Frame _cilkFrame = (Fib_calc_Frame) _cilkWorker.getFrame(
      Fib_calc_Frame.class);
2 if (_cilkFrame == null) {
3     _cilkFrame = new Fib_calc_Frame(n, 0, _cilkRetEntry);
4 } else {
5     _cilkFrame.reConstruct(n, 0, _cilkRetEntry);
6 }
```

Listing 6.3: The code for calling the `getFrame` method to reuse an existing frame before creating a new one in case a reusable frame was not found.

Implement `reConstruct` Methods

This step consists of creating a `reConstruct` method in each `CilkFrame` class to mirror the constructors of those classes. The idea is that this method should do the same as the constructor does for the object. Thus, each `reConstruct` method starts by calling its `super.reConstruct` method, which invokes the simulated constructor of the `CilkFrame` class. However, this is not all that the method needs to do, as there is some initialization in each object that does not take place in the constructor; that of the local variables. Listing 6.4 shows the initialization of the local variables in Line 1, and this line also assigns the default values to these variables, which in this case is 0.

```
1 private int _n, _x, _y, _returnVal;
2
3 public Fib_calc_Frame(int _n, int _pc, int _returnEntry) {
```

```
4     super(_pc, _returnEntry);
5     this._n = _n;
6  }
```

Listing 6.4: The code for constructing a `Fib_calc_Frame`. Includes local variables and constructor (simplified).

Listing 6.5 shows the `reConstruct` method for the `Fib_calc_Frame` class. The method does three distinct things in order to simulate the initialization of a new object. First, it calls the `super` version of itself, which is done in Line 2. Secondly, it performs the code from the constructor, which is Line 3. Note that this line is mirrored in Line 5 of Listing 6.4, which shows the constructor. Finally, the remaining local variables are set to their default values, in Lines 4-6.

```
1  public void reConstruct(int _n, int _pc, int _returnEntry) {
2     super.reConstruct(_pc,_returnEntry);
3     this._n = _n;
4     this._x = 0;
5     this._y = 0;
6     this._returnVal = 0;
7  }
```

Listing 6.5: The reConstruct method for the `Fib_calc_Frame` class (simplified).

This concludes the first phase of implementing the frame reuse system, and it now works with the `Fib` program which is compiled by hand. The debugging phase was surprisingly short for this part, and a few preliminary tests show that the number of objects created has indeed been dramatically reduced, going from 317,269,119 objects in the unmodified JCilk version to 78 in the object reuse version. These numbers are for the pseudo-serial versions using one worker. When using more workers, the number of objects created in the reuse system is greater, and becomes a bit varied, as it depends on the number of steals performed in the program, which is nondeterministic. However, as noted in Section 6.1, the amount of frames allocated rises approximately with a factor of n proportional to the number of processors introduced.

Additionally, the time to compute Fib(40) with two workers has been reduced from 23 seconds to 11 seconds, which gives an indication that the overhead introduced (storing frames, checking for used frames to reuse) does not outweigh the advantage of less frame allocation, at least in this case. More tests will be performed later in Chapter 8, but for now this is enough for us to continue to the second phase, which is implementing the changes in the compiler.

6.3 Polyglot

Polyglot [22] is the base of the JCilk compiler as it is the underlying framework for performing the compilation from JCilk to GoJava. Both of the improvements implemented in this project need to make modifications and additions to the Polyglot project used for JCilk. This section is a short description of the framework.

Polyglot assumes that changes are being made to the Java language and its goal is to make the amount of work required to create a Java extension proportional to the amount of changes to make to the Java language. To achieve this goal, Polyglot starts as a Java to Java compiler, which is easily modified. Any new language constructs to be added to Java are defined in the corresponding Java CUP parser library [26], which is a LALR parser generator. CUP creates an abstract syntax tree based on the modified Java grammar. Polyglot can afterwards be configured to create any of a series of parses upon this AST. Each parse represents a useful analytic or transformation task which can be relevant to certain types of modifications to Java. Polyglot can be configured to do the following parses (not an exhaustive list):

type checking Performs a semantic analysis.

exception checking Semantic checking on whether the rules regarding checked/unchecked exceptions are upheld

reachability checking Checks the reachability of the code within each method defined by the code.

exit checking Checks that all paths through the methods defined by the code return the values that they are declared to return.

initialization checking Checks that all variables that are being accessed, have been initialized prior to this.

translation Transforms each AST node to a string and writes to a defined output.

To facilitate the translation parse, it is necessary to implement the logic which is applied to each AST node in the tree when it undergoes a transformation. As JCilk is mainly a translation into another language, part with the most code is the translation pass.

6.4 Compiler Implementation

To implement the reuse system into the compiler, we examine what changes we need. We need to implement three changes; those that we implemented into the `Fib` program as described in Section 6.2.

Saving frames for reuse is done by inserting the call to `addUsedFrame` before each `return`. This is very straightforward, the one exception being that the `return`s used to escape the method when work-stealing has been detected should not be considered, since the frame should not be reused when the task has been stolen; the stealing worker will need to use the frame, so reusing it would cause problems. Listing 6.6 shows the method which prints the code to add frames to the reuse system. This code is printed in front of every `return` that is not a work-stealing escape. Line 4 shows the actual writing, the other lines simply set up and finish the method. We see that it prints the name of the worker and concatenates this with a dot and the call to the `addUsedFrame` method of that worker. The parameter of the call is also printed, and it is the frame for the current task.

```
1  public void prettyPrint(CodeWriter w, PrettyPrinter pp) {
2        w.begin(0);
3        String worker = util.genWorkerFormal().name();
4        w.write(worker + ".addUsedFrame(" + frameLocal.name() + ");");
5        w.end();
6     }
```

Listing 6.6: The `prettyPrint` method of the `AddCilkFrame.java` class in the JCilk compiler.

Checking for frames to reuse is effectively done by replacing the line:

```
_cilkFrame = new Fib_calc_Frame(n, 0, _cilkRetEntry);
```

... with the code in Listing 6.3. This is done in exactly one place in each Cilk method, and the only changed code is the name of the frame class.

The final thing to do is create the `reConstruct` methods. First, the constructor code is inserted, and any local variable assignments done in their declaration is copied. Finally, any variables that were not assigned new values yet are assigned their default values.

This concludes the implementation of the frame reuse system. In Chapter 8, we test the performance of the new system in order to see if it has improved the JCilk framework.

Chapter 7

Simplifying the JCilk Compiler Architecture

This chapter describes our work on the removal of the second stage in the two-stage compilation process used in JCilk. We begin by describing the ideas with a higher level of detail, then proceed to describe the design of the algorithm as well as advantages and disadvantages, and finally the implementation details are presented.

7.1 Idea

The idea of this JCilk improvement is, as explained in Section 5.2, to remove the current necessity of the Jgo compiler. By removing this compilation step, we are able to simplify the compilation process and reduce the disk space required by the compiler significantly.

In order to remove the GoJava language, it is necessary to remove the need for `goto` statements. The `goto` statements are currently used for jumping within a method. Furthermore, the `goto`s are only used between statements, and do not jump within expressions. Since the functionality provided by `goto`s are necessary for continuations, it is necessary to simulate this functionality with Java constructs.

The idea proposed in this chapter eliminates the labels and `goto`s in compiled GoJava programs by splitting the language constructs in which they can occur into smaller pieces such that continuations are possible. In effect, we are adding an extra pass in the compiler which replaces the functionality of Jgo with corresponding Java code. Also note that it is only necessary to change code within the slow clones of Cilk methods, since this is the only place where `goto` statements are utilized.

7.2 Algorithm Design

Appendix B shows an example of the current way `goto` statements are used. The initial JCilk algorithm is translated into a new version (Listing B.3) where a `switch` statement determines what label to go to based on the current program counter, i.e. the continuation point. Removing the `goto` statements would be very simple in this given situation as it would just be a matter of moving the logic between each label into the `switch` statement. However, it becomes problematic to make such a transition when the labels are nested deeply within several compound statements. An example of such a situation can be seen in Listing 7.1. In this example a `spawn` statement is nested within a `for` loop and further into an `if` statement. To facilitate a continuation in such a case, it is necessary to create a more complex abstraction for continuations.

```
for(int j = 0; j < n; j++) {
   int[] nconfig = new int[n];
   System.arraycopy(config, 0, nconfig, 0, n);
   nconfig[i] = j;

   if(safe(i, j, nconfig)) {
      spawn nqueens(nconfig, i+1);
   }
}
```

Listing 7.1: Nested spawn statement.

We propose a solution where the language constructs are split into smaller chunks of code which facilitate continuation. These chunks are organized as cases in a `switch` statement with a `while` loop acting as program loop surrounding the `switch`. Listings 7.2 and 7.3 show a simple example with a `spawn` inside an `if` statement and how it will be translated with this algorithm.

```
switch(thisFrame._pc) {
   case 1:
      goto label_1;
}
/* code block 0 */
if(/* expression */) {
  /* code block 1 */
  label_1: ;
  thisFrame._pc = 2;
  /* code block 2 */
}
/* code block 3 */
```

Listing 7.2: Simple Jgo compilation result (irrelevant code omitted).

```
while(thisFrame._pc != -1) {
   switch(thisFrame._pc) {
      case 0:
         thisFrame._pc = 1;
         /* code block 0 */
         break;
      case 1:
         if(/* expression */) {
            thisFrame._pc = 2;
         }else{
            thisFrame._pc = 5;
         }
         break;
      case 3:
         thisFrame._pc = 4;
         /* code block 1 */
         break;
      case 4:
         thisFrame._pc = 5;
         /* code block 2 */
         break;
      case 5:
         thisFrame._pc = -1;
         /* code block 3 */
         break;
   }
}
```

Listing 7.3: The Jgo example where `goto` statements are eliminated from code containing an `if` statement (irrelevant code omitted).

The algorithm simulates the possibility of making a `goto` into an `if` statement by splitting the `if` statement such that the expression of the statement becomes a `case` in itself, and the consequent of the `if` statement is split into several other cases to facilitate an exact jump to the label. We have designed a general algorithm that can support all types of compound statements. This algorithm is shown in Listing 7.4.

```
function remove_jgo(SlowMethod method) {
   remove switch from method
   switch new_switch
   add (case 0: remaining code in method) to new_switch
   remove duplicate remaining code from outside the switch
   for each label : current_label in new_switch {
      construct child_list for current_label starting at switch root

      int entry, exit = -1
      statement next_code_block
      for each compound statement : current_child in child_list {
         if root level of switch {
            inherit entry
         }
15          current_stmt = isNull(next_code_block)?current_child:
                next_code_block
16          child_stmt = get_child_stmt(current_child, child_list)
17          child_child_stmt = get_child_stmt(child_stmt, child_list)
18          next_code_block, entry, exit = handle_level(current_stmt,
                child_stmt, child_child_stmt, entry, exit)
19        }
20    }
21 }
22
23 function handle_level(Statement current_stmt, Statement child_stmt,
       Statement child_child_stmt, int entry, int exit, SwitchStatement
       switchStmt) {
24    code_before_child = code between start of current_stmt and child_stmt
25    next_continuation_point = get_new_case_id()
26    add (case entry: pc = next_continuation_point, code_before_child) to
          switchStmt
27
28    int new_entry, new_exit
29    statement next_code_block
30    next_code_block, new_entry, new_exit = handle_child_stmt(current_stmt,
          child_stmt, child_child_stmt, next_continuation_point)
31
32    code_after_child = code between child_stmt and end of current_stmt
33    add (case new_exit: pc = exit, code_after_child) to switchStmt
34
35    return next_code_block, new_entry, new_exit
36 }
```

Listing 7.4: Pseudo code for the Jgo elimination algorithm.

The `remove_jgo` function takes as input the slow clone of a Cilk method. The first step of the algorithm is to locate the relevant code which will be the subject of the transformation. This is done by removing the original `switch` statement in (Line 2). The second step of the algorithm is to move the remaining code into `case 0` of a new `switch` statement (Lines 3-5). An example of the code inside `case 0` can be seen in Lines 14-33 in Listing B.3.

The next step the algorithm goes through is to find the labels and redefine the statements they are nested within. The outer loop (Lines 6-20) iterates through all the labels that the new `switch` statement contains. When a label is found, the first thing to do is to find its child list. The child list is a list of the statements leading to the label. Figure 7.1 shows a small code sample and the corresponding child list leading to the label.

Each element in the child list needs to be restructured in order to support a continuation within it. After the child list has been generated, three variables are initialized (Lines 9-10). The `entry` and `exit` variables hold the `case` ids which will be used in the `case` statements that enter and leave the level corresponding to an element in the child list. The `next_code_block` variable holds the current block of code which needs to be split up into a number of `case` statements.

```java
int i = 2;
while(i < 10) {
   if(i%3 == 2) {
      try {
         label1: ;
      } catch(Exception e) {
      }
   }
   i++;
}
```

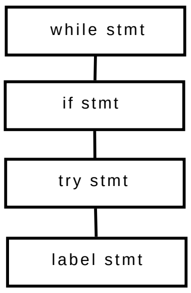


Figure 7.1: A GoJava code block and its corresponding child list.

The next step of the algorithm is to iterate through each element in the child list starting from the `case` in which the label has been found (Lines 11-19). In the first iteration, the element should inherit the `entry` and `exit` values already defined (Lines 12-14). In subsequent iterations, the values defined by the call to the `handle_level` function in the previous iteration (Line 18) are used. This ensures that the control flow is consistent after restructuring. The job of `handle_level` is to create the necessary new case statements in order to support a continuation to the beginning of the child of the current statement.

The `handle_level` function (Lines 23-36) consists of three parts. The first part generates a new case statement containing the code between the beginning of the current statement block and the child statement. In the beginning of this code block an assignment to the program counter is appended with a new unique case identifier (lines 24-26). The case id is defined as `entry` as this value represents the value of the program counter when this piece of code should be executed.

The second part handles the child statement, which consists of finding the next code block and new entry/exit case ids for the child list iteration loop (lines 28-30). How this is performed in `handle_child_stmt` for the different types of statements is defined in Section 7.2.1.

The last part of `handle_level` adds a case consisting of the code between the child statement and the end of the code block defined by current statement (lines 32-33). The case id of this code block is set to `new_exit` as this represents the destination when the code inside the child block has finished execution.

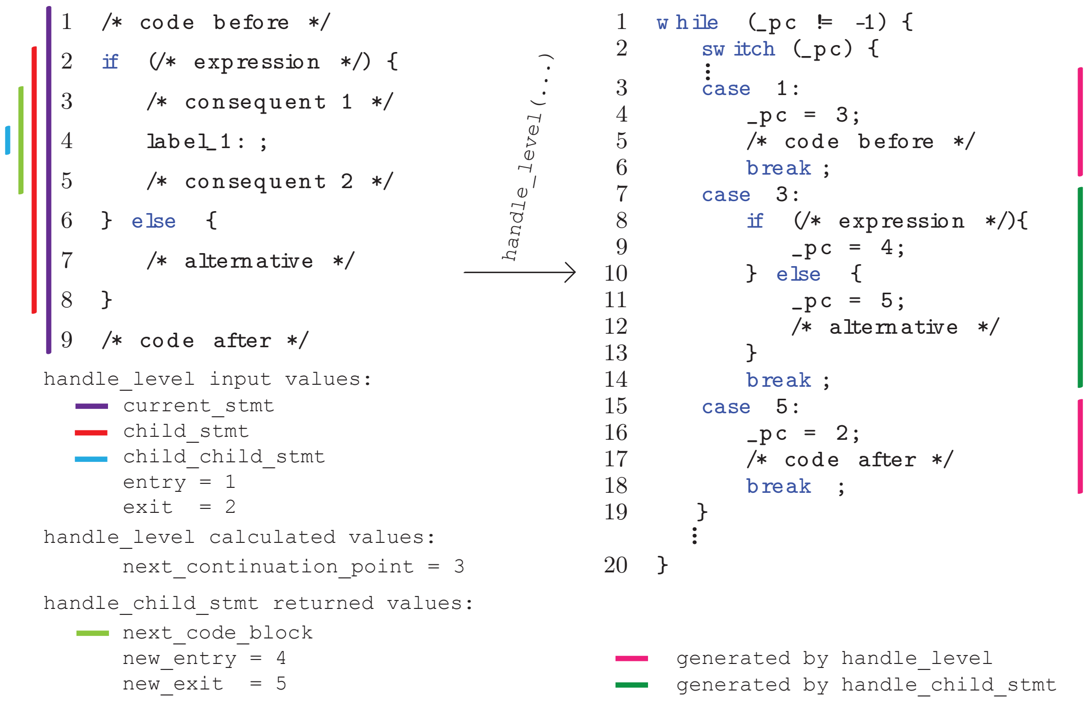
A requirement for the algorithm to work is that all variables that are created within the method to be transformed have to be defined such that they are available for the entire scope of the method. This precaution is, however, already necessary with the current Jgo implementation so an implementation of this requirement has already been made.

7.2.1 Restructuring Compound Statements

The algorithm presented in the previous section can be described as a template for how to handle each level that needs to restructured and bind them together. How each level is actually modified into `case` statements is described in this section. The general requirements for each type of compound statement is that they are given access to the child list and an entry program counter value which invokes their code. The result of restructuring the statement should be new `entry`/`exit` values and a new code block to use as `current_statement` in the next iteration. In addition to the statement-specific mechanisms, a detailed explanation of the interaction with the `handle_level` function is described for `if` statements.

`if` Statements

Figure 7.2 shows an example of how `if` statements are converted into code which facilitates continuations.



```
/* code before */
if (/* expression */) {
    /* consequent 1 */
    label_1: ;
    /* consequent 2 */
} else {
    /* alternative */
}
/* code after */
```

```
handle_level input values:
   🟪 current_stmt
   🟥 child_stmt
   🟦 child_child_stmt
      entry = 1
      exit  = 2
handle_level calculated values:
      next_continuation_point = 3
handle_child_stmt returned values:
   🟩 next_code_block
      new_entry = 4
      new_exit  = 5
```

```
while (_pc != -1) {
    switch (_pc) {
    ⋮
    case 1:
        _pc = 3;
        /* code before */
        break;
    case 3:
        if (/* expression */){
            _pc = 4;
        } else {
            _pc = 5;
            /* alternative */
        }
        break;
    case 5:
        _pc = 2;
        /* code after */
        break ;
    }
    ⋮
}
```

🟥 generated by handle_level
🟩 generated by handle_child_stmt

Figure 7.2: Compiler transformation of the `if` statement in `handle_level` and `handle_child_stmt`.

The example shows a situation where `current_stmt` is a block consisting of an `if` statement as child and some code surrounding it. Using the given parameters in entry and exit, `case 1` and `case 5` are generated by `handle_level`. The `if` statement is transformed such that the subblock containing `child_child_stmt`, which in this case is located in the consequent part of the `if` statement, will be used as `next_code_block` in `handle_level`. Since this part will be transformed in the next iteration, we just need to create a new label id and return it as `new_entry` such that the next iteration of `handle_level` knows at what case id to start its generated case statements. The other subblock of the `if` statement, in this case the alternative, can remain unchanged and simply refer to the `new_exit` case after execution.

`while` and `for` Statements

Since `for` loops can be converted into `while` loops by simply moving the initialization part before the loop, keeping the condition part, and moving the iteration step to the end of the loop body, it is only necessary to define a transformation for while loops. The transformation needs to do two things; facilitate the continuous nature of loops, and support breaking and continuing within the loops. Creating the loop behavior can be done by creating a case with an `if` statement with the same condition as the `while` statement with a program counter reference to the body of the `while` inside the consequent and a reference to the code after the `while` in the alternative. In the code block representing the body of the `while`, a reference to the newly created `if` statement is inserted. By continuously visiting the condition in the form of an `if` statement and the body whenever the condition is true, the same behavior as a `while` statement is achieved.

To support the `continue` and `break` statements within `while` loops, it is necessary to create a search for these statements within the `while` loop. The search needs to find all `continue` and `break` statements recursively within the children, without entering any other loops as the breaks within these will relate to another scope. Whenever a `break` is found, a reference to the program counter for the code after the loop should be inserted right before the `break`. In the case of `continue` statements, they need to be replaced with a reference to the beginning of the loop and the `continue` needs to be replaced with a `break`, to perform the necessary jump in the code. This approach can also be extended to labeled breaks as it is just a question of figuring out whether a loop has a label declaration and if so, search within other nested loops for `break` statements with the label attached.

`try-catch-finally` Statements

JCilk only supports continuations within the `try` block of these statements, so a solution for this compound statement only needs to support a continuation within this part. A continuation inside a `try` block can be made by making two copies of the statement where the only difference is that the `try` part in one of the statements consist of the code before the continuation, and the other statement contains the code after the continuation.

`switch` Statements

`switch` statements can be transformed such that each `case` within the original switch statement becomes a new `case` in the program continuation loop. Each new case consists of the body of the originating case following by an `if` statement. The `if` statement either jumps to the code after the original `switch` or to the next `case` from the original `switch`, in order to support fall through. The original switch is inserted in its own `case` where the code of each `case` is replaced with a program counter declaration corresponding to the `case` that piece of code has been moved to.

7.2.2 Overhead

This section describes the two types of overhead generated by the Jgo elimination algorithm. One type of overhead is the extra space taken by the creation of the new `switch` statement and all the `case`s. The other type consists of the actual computational overhead generated by having to make more jumps than before.

The space overhead can be defined as the following:

$$new\_cases = \sum_{i=0}^{m} \sum_{j=0}^{lvls(i)} k(j)$$

In this formula m is the number of spawns within the original method, and i is a given spawn statement. $lvls(i)$ returns the number of compound statement transformations that will be necessary in order to facilitate continuations at the spawn i, and j is one of these statements. $k(j)$ returns the number of new case statements that need to be generated in order to transform the statement j. The number of spawn statements within a method is usually pretty low ranging from 1 to 4. The number of levels leading to the code where the spawns are located will in practical cases never exceed 10 as any more levels most often would indicate bad program design. The number of additional case statements generated by the different types of statements is for the most commonly used statements, if and while, only 1. Combining these values shows that the algorithm increases the size of a program by a constant factor. It should also be noted that the algorithm does remove some of the original code, as label declarations are no longer necessary. This means that programs with a high number of spawns relative to the rest of the program might even take up less space.

Table 7.1 shows the lengths of the compiled programs. Each program has increased in length, but the increase for each program is less than 10%.

The new running time of a program can be defined as following:

$$runtime = (a + b + c) \cdot j + n$$

Program	GoJava	No Goto
Fib	368	369
Queens	307	327
QuickSort	363	371

Table 7.1: Number of lines in the GoJava versions and the versions without GoJava of three programs.

In this formula a is the time it takes to make one while iteration checking on the program counter. b is the time it requires to make one check and jump to somewhere within the switch statement, while c is the time it takes to make a single assignment to the program counter. a, b and c are all constants. j is the number of jumps performed during the execution of the program, while n is the original running time of the program. If we call the average computation per jump f, which is constant, we can express the following equation about the relation between n, j, and f:

$$f = \frac{n}{j} <=> j = \frac{n}{f}$$

We now have j expressed in relation to n, and the running time can be expressed as:

$$runtime = \frac{(a+b+c)}{f} \cdot n + n$$

$\frac{(a+b+c)}{f}$ is a constant since all of a, b, c and f are constants. We can now deduce the new running time of the program:

$$runtime = O(n)$$

We note, of course, that an increase of $\frac{(a+b+c)}{f} \cdot n$ in the running time does take place, but we can conclude that the computational overhead is linear. A problem that might occur for big methods is that the algorithm might spread otherwise consecutive lines of code, thus making the code unsuitable for caching. What these sources of overhead mean in terms of practical running time will be shown by the tests performed in Chapter 8.

7.3 Implementation

This section presents the current state of the implementation as well as suggestions for optimizing the performance of the system.

Since the goal of the Jgo elimination algorithm is to adapt the compiled code into plain Java which is compliant with ordinary Java compilers, the entire implementation has been made on the compiler, and no changes have been necessary within the runtime system. The current implementation supports compilation of `if` statements and `while` statements without `break` or `continue`, which are the two necessary statements in order to compile the three programs used for the tests, as described in Section 8.1.

The Jgo elimination algorithm is implemented late in the Polyglot compilation process. As described in Section 6.3, the different new productions added in JCilk to the original Java language are parsed through a number of source code iterations. The elimination algorithm is applied at the very end of this process, after all `spawn` and `sync` statements have been translated into code involving labels and `goto` statements. This choice was mainly made because it is necessary to keep the same program counter values at the same continuation points in order to convey to the program counter values set in the fast clones of the methods. This means that the compiler modification does not need to change anything in the fast clones.

To give an example of the transition from pseudo code to Java, Appendix E shows the Java implementation of the `remove_jgo` pseudo code function from Listing 7.4. The Polyglot statement abstractions are well suited for the mindset used in the pseudo code, and as a consequence, the mapping from pseudo code to Java methods has not caused any major problems. Examples of compilation output can be found on the Appendix CD in the test folder, where the three algorithms used in the performance test are located.

7.3.1 Optimizations

The runtime performance of the Jgo elimination algorithm can be improved by running two sanitation filters on the final output. One filter can remove the ”empty“ cases, which do nothing but point to another case in the `switch` statement. An example of such a case statement is shown in Listing 7.5. These statements can be removed by moving all logic to the case currently doing nothing but refer on to another case and remove the original case. All other references to the moved case need to be changed accordingly.

```
while(_cilkFrame._pc != -1) {
   ...
   case 9:
      _cilkFrame._pc = 8;
      break;
   ...
}
```

Listing 7.5: Example of an empty case resulting from the compilation process.

The other filter that can be applied is reordering the case statements such that case statements which are more likely to follow each other in the execution path are also placed sequentially within the `switch` element. This can be an advantage when executing long methods as there might not be room for the entire `switch` statement in the level 1 and 2 caches. Doing such an optimization is possible in this case because program counter assignments are deterministic and never incorporate any type of expressions when assigned. One way to make a filter is to create a directed acyclic graph where each case is a vertex and possible moves within the `switch` statement are represented by edges between these. The weight of each edge represents a probability factor for moving towards one vertex compared to other vertices. The weight of edges will be equal for some cases as it is impossible to predict any behavior, for example in `if` statements where it is impossible to know whether the consequent or alternative will be visited. An example where one edge will be longer than another is `while` statements, where the weight of the edge moving towards the body of the statement will be longer as the probability of this statement being visited more often is significantly higher. The algorithm starts at `case 0` and makes a depth-first search through the graph choosing highest weight of edges when possible. For each visited node, the corresponding `case` is inserted in the `switch` statement in the order they are visited. With this algorithm, all cases which run sequentially are placed after each other. This gives the possibility of removing the `break` statements in many cases and use fall-through in the `switch` statement.

Whether these two filters will have a noticeable impact on performance is dependent on the existing optimizations in the Java compiler and runtime system. An implementation and corresponding test of these filters is necessary to give a qualified estimate of their efficiency. However, performing these tests do not lie within the scope of this report.

Chapter 8

Performance Testing

This chapter describes the tests we have performed on the JCilk system. The purpose of the tests is to compare the performance of our modified version of the JCilk system with the original system.

8.1 Method

The testing method needs to ensure that a fair environment is configured for the tests. This is important in this case as JCilk-1 is a prototype implementation so there is no guarantee that all relevant algorithms will work correctly in the system. To ensure that this does not become a problem, the original algorithms used for testing JCilk will be reused. Furthermore, the same testing conditions will be replicated whenever possible. This includes aspects such as using similar numbers of processing cores as used in the previous tests. All in all, it means we should adhere to the settings used in the original JCilk tests to ensure reliability of our results. The original testing conditions can be seen in [9, Chapter 5].

Based on these considerations, the following sections describe the characteristics of the test, what programs we are testing the systems on, the systems we are testing and the environment used to perform the test.

Characteristics

Like the original tests, we will be making a performance test, testing the scalability of the subjected systems at a different number of used processors. The scalability of the system will be tested by increasing the number of processors assigned gradually and evaluate the performance change. The ideal of such a test is that there is a small gap between the serial elision of the system and the runtime system running the program with one thread. Any gap between these two measurements shows the overhead of using the runtime system. Furthermore, as the number of threads/processors are increased, the performance should increase at the same ratio. This means that a program ideally should perform four times better at four processors compared to one.

Test Program Selection

We choose the following three programs to test with; `Fib`, `All-Queens`, and `Quicksort`. The first two are chosen as these were used in the original JCilk tests while `Quicksort` is chosen by us because it lies in between the other algorithms in terms of the amount of computation per `spawn`. Furthermore, it adds a third reference point on the scale of computation-per-spawn when comparing the different versions of the system.

`All-Queens` is a modified version of the `Queens` program which finds all solutions to the problem. `All-Queens` is used over the traditional version since a test with `Queens` would be subject to too much random chance, since `Queens` only looks for a single solution, and the program nondeterministically examines the tree of possibilities. Usual implementations of `All-Queens` use `for` loops, but in order to support our compiler, these have been replaced with `while`s.

By using three computationally different algorithms for testing, a good impression of the system performance with different degrees of computation-per-spawn is obtained.

Test Systems

We test three different systems; the unmodified JCilk, JCilk without GoJava, and finally JCilk without GoJava and with frame reuse. Testing with JCilk without GoJava shows the impact of adding the extra jump statements required to simulate continuations. Finally, JCilk without Jgo and with frame reuse shows the performance of the two additions combined.

Originally, we planned to also test the original system with only reuse, to see the effects of the reuse system in isolation. However, the preliminary results of doing this showed that there was a problem with the test, giving results that were slower than the others by a factor of 10, which indicates a problem in the test. Since we were not able to solve this problem, the reuse system with GoJava is not included in the tests.

Environment

All three algorithms are tested with the four different systems. Like the original JCilk tests, each system will be tested using 1-4 processors. Furthermore, a test of the serial elision running time is made for each algorithm.

The programs all have a problem size input, or n. We run our tests with a large enough n that the tests take a significant amount of time, such that the whole system is put into action and runs for some time. The values of n used are 40 for `Fib`, 14 for `All-Queens`, and 100,000,000 for `Quicksort`.

We are running all tests on the following application server: Dell PowerEdge 2950, 2x2.5 GHz CPU (Quad Core Intel Xeon), 32 GB RAM. This is a public application server in our institute so no exclusive use can be given to its processing capabilities.

To ensure reliable results on the shared application server we run each test 25 times. The average running time is found by taking the median value of these tests. We choose median because it eliminates outliers and gives an accurate average, as we are performing such a high number of runs.

8.2 Results

This section presents the results from our test.

8.2.1 `Fib` Test

Threads:	1	2	3	4
Fib	25.07	13.42	9.31	7.22
Fib no Goto	25.06	13.49	9.39	7.77
Fib Reuse no Goto	24.16	12.48	8.36	6.33
Fib Serial elision	1.14			

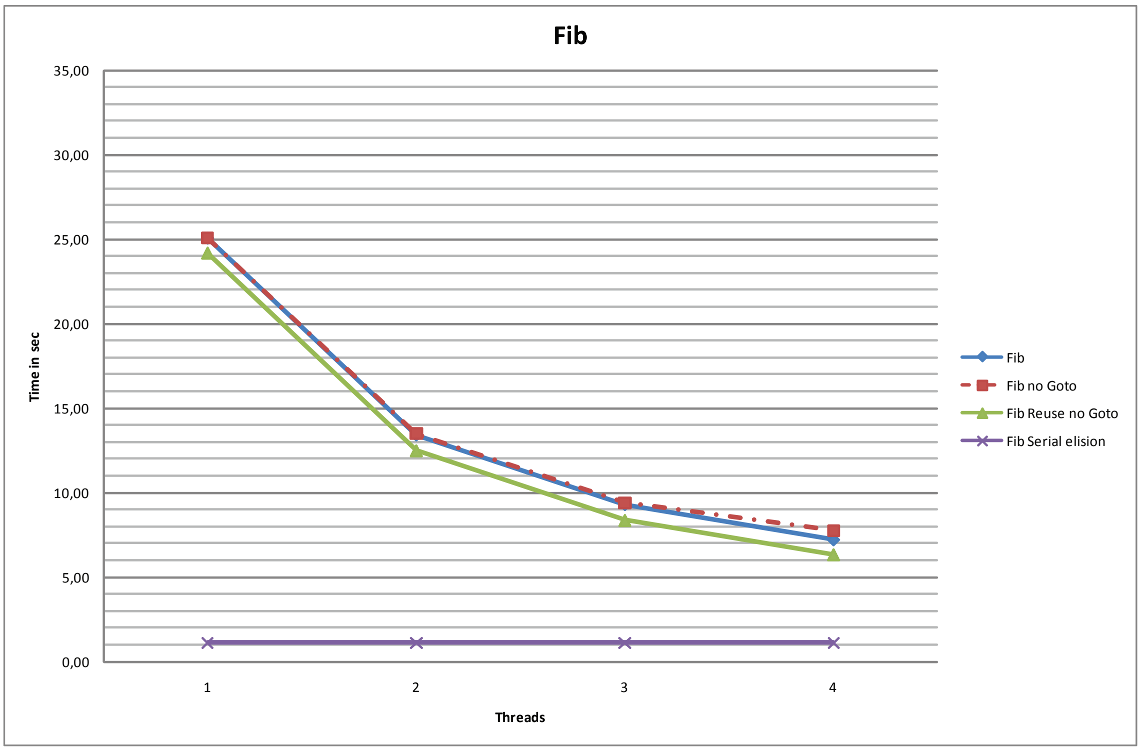


Figure 8.1: Fib Test Results

Figure 8.1 shows the results obtained by running the three JCilk configurations with `Fib`. A serial elision of the program added for comparison.

The `Fib` test shows that JCilk without GoJava has the same performance as with, except for the case of four processors where there is a slight gap in favor of the old system. We also observe that the reuse system gives an increasing advantage in performance for all settings. Like in the original tests performed by Danaher, we see that the serial elision is significantly faster than any of JCilk variants.

8.2.2 `All-Queens` Test

Threads:	1	2	3	4
Queens	22.37	11.86	9.18	7.98
Queens no Goto	22.26	11.80	9.29	7.99
Queens Reuse no Goto	22.26	13.80	10.29	8.60
Queens Serial elision	18.78			

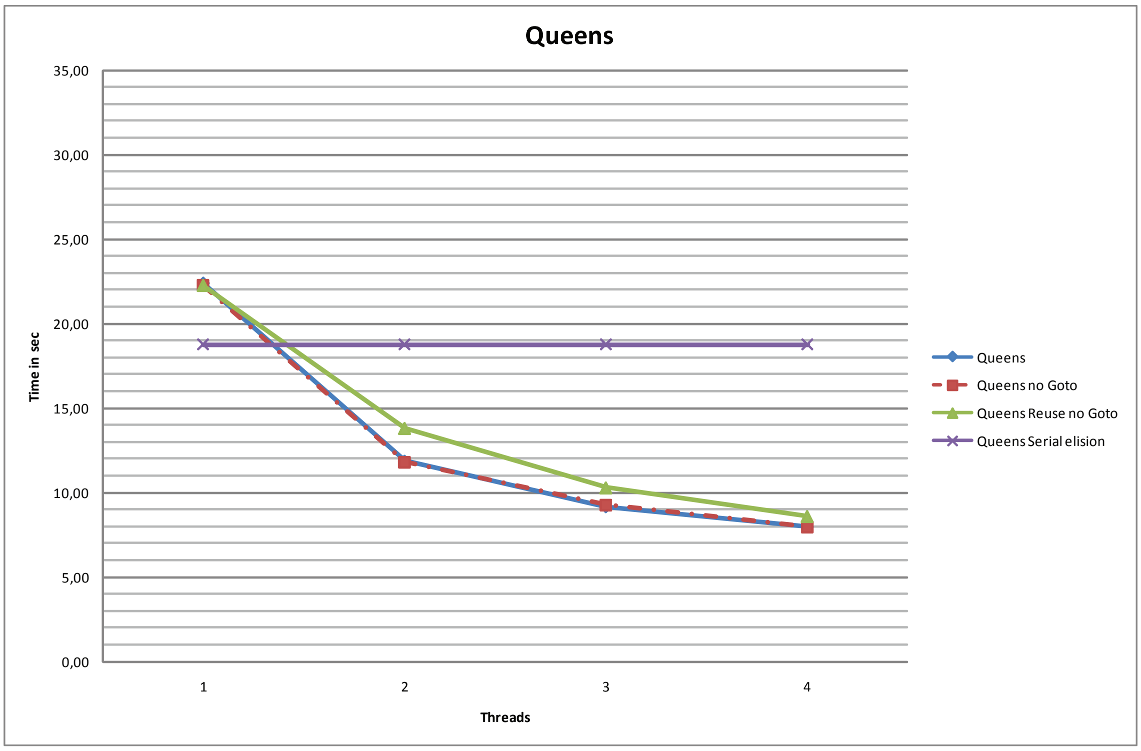


Figure 8.2: Queens Test Results

Figure 8.2 shows the results obtained by running the three JCilk configurations with `All-Queens`. A serial elision of the program added for comparison.

The `Queens` test also shows that there is only a small difference in the performance between the old system and the new system without GoJava. Furthermore, it shows that instead of a small increase in performance, the reuse system adds a small reduction in performance in this test. The serial elision is outperformed at two processors in this program.

8.2.3 `Quicksort`

Threads:	1	2	3	4
QuickSort	30.77	17.39	13.32	12.12
QuickSort no Goto	30.81	17.53	13.38	12.03
QuickSort Reuse no Goto	30.79	17.33	12.95	11.57
QuickSort Serial elision	19.76			

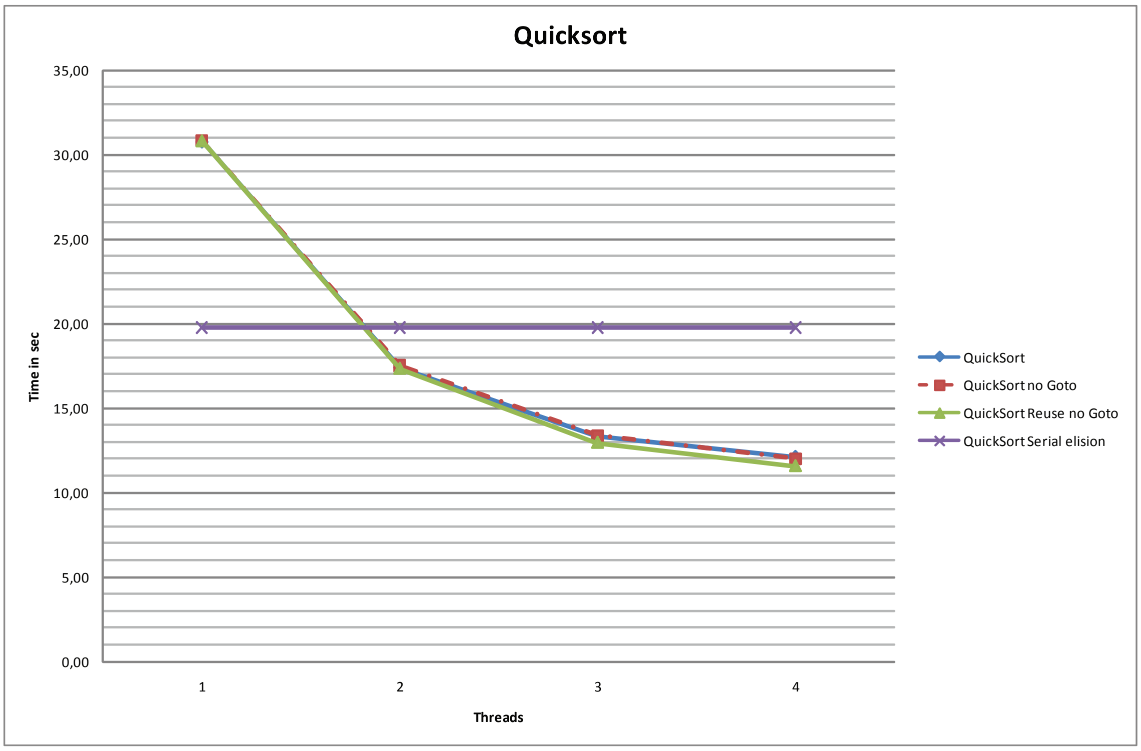


Figure 8.3: QuickSort Test Results

Figure 8.3 shows the results obtained by running the three JCilk configurations with `Quicksort`. A serial elision of the program added for comparison.

The `QuickSort` test shows the same pattern as the other tests with the original JCilk system and the system without GoJava. The reuse system is in this configuration close to the other programs, but shows a slightly better performance.

8.3 Discussion

All tests show that removing GoJava from JCilk has not introduced any significant performance overhead. In all tests, the results for the original system and JCilk without GoJava are very close.

The tests show that for the programs with a low ratio of computation-per-spawn, `Fib` and `Quicksort`, reusing frames has given a significant increase in performance. It does, however, also show that for the `Queens` program, which has a a high number of computations for each spawn, it caused an overhead rather than an improvement. Even though the reuse system might cause overhead on certain computation-heavy algorithms, this is not a big loss as JCilk already performs well in these cases. Currently, the most important thing to improve in JCilk is algorithms such as `Fib` and `Quicksort` where JCilk currently has problems outperforming the serial elision of the programs.

For current hardware systems, which usually have at least two cores, JCilk outperforms serial Java execution as long as there is a reasonable amount of computation per spawn. This is significant because systems will only gain more in time, and it is already reasonable to use JCilk over serial Java and will become even more so in the future.

Chapter 9

Evaluation

In this chapter, we go through some of the choices we have made during this project and evaluate on them. Each numbered section represents a number of decisions and their evaluation, and the order of the sections will follow the order of the decisions for the most part, such that the evaluation is done as chronologically as possible, though some decisions happened more or less in parallel, particularly during implementation.

9.1 Choice of Problem

This decision represents the outcome of the time leading up to starting the project as well as the early project period. There are several choices made here. The first choice was to work on the Cilk programming model and to integrate that into an OOP language. Later, we chose that this language would be Java, and finally we changed our intent to develop our own language extension to working on an existing one. We ended up choosing JCilk for this.

The choice to work on the Cilk programming model was a good one. It is an elegant, efficient way of modeling concurrency, and as we have seen from JCilk and our own work on JCilk, it is indeed transferable to the OOP paradigm, with minor changes. This topic gave us a basis where we could work on something that is useful for future language development, while also being something that interested us, which provides good motivation.

The early choice to use Java as the target language to create an extension to is debatable. While the choice of language is mostly arbitrary since the other obvious candidate, C#, is so similar and the extension is transferable, we limited ourselves unnecessarily at a very early stage by making this decision. On the other hand, JCilk ended up being such an obvious candidate once we chose to work on an existing extension that the choice to use Java did not really affect us except in how it limited our line of thought somewhat.

When we decided to work on an existing extension rather than develop our own, it was a complex decision. First of all, we had already spent some time investigating the possibilities for developing Java extensions, and this represented a time investment that would not yield high returns for us when going down this path. Secondly, we had found that much work had already been done in the field of Cilk-like OOP programming models, which meant that we would get to build on an existing foundation rather than reinvent the wheel. This was an important factor for us, as we think that contributing to an interesting project is worth more than creating a small project which would be less likely to see any use.

In the end, we chose JCilk, and it does a lot of the same things that we were going to do in our own extension, so it ended up being very good for us. It did, however, result in us spending a lot of time studying the JCilk source files and the reports and articles published about JCilk, which took up more time. In spite of the time cost that this decision represented, we think that this choice was very good compared to developing our own extension which would have done largely the same as JCilk.

However, one thing we could have done better is to realize this earlier on in the process. As it was, we did not choose JCilk before we were over a month into the project period, and making this choice during the first week, for example, would have given us more time to invest in improving JCilk further. We should have investigated the related work in more breadth before divulging into research on the tools for creating extensions.

The decision to work on a project created by someone else always has risks. There is the potential to waste time on pitfalls that they have already found but not shared in their writings. There is the difficulty in understanding their work, which can make continuing it problematic, especially when the work is not fully documented. Our choice to work on JCilk has had us examine some source code that was undocumented, and even some that the creators were not sure of the implications of. A couple of example of the JCilk source code comments we have encountered during implementation:

> *'Does the abort move the PC to -1?'*
> *'STILL: I'm not really sure why the !pcMoved check is here.'*
> (Both from `Closure.java`)

In fact, it seems like the source code has been used for communication between the developers, and it also seems like the project is unfinished, as are most software projects. This is only natural and to be expected when undertaking the task of continuing development on the software projects of others. We have spent some time figuring out the source code of JCilk, but in the end we think that this cost was worth the reward of getting the 'head start' of being able to continue where they left off rather than starting from scratch.

9.2 Choosing Areas for Improvement

We presented the potential areas for improvement that we found to be most relevant, chosen from our own ideas as well as from the suggestions in the Danaher/Lee Master's Theses. We believe that we presented the relevant areas, but as for suggestions for concrete improvements, we were not systematic or exhaustive enough to find the best ideas before beginning development. Especially with the frame allocation overhead reduction, if we had considered the possibilities more thoroughly, we might have thought of the idea for object reuse before starting implementation. As it happened, we thought of the idea while implementing the memory manager, and once we started implementing object reuse, it quickly became apparent that it was far superior to the memory manager as a solution to the frame allocation overhead.

9.3 Development Decisions

In general, we used a two-phased development model of implementing a change in the runtime system first, and testing it with source files created by hand rather than changing the compiler at first. Once we verified that the change worked well enough, we implemented the change in the compiler such that it could generate the appropriate source files. This model is very good for making fast prototypes for testing out an idea, and indeed it was in the first phase of development that the memory manager was discarded. Furthermore, it is an advantage in the cases where an implementation cannot be finished on time as there will still be a working prototype even though the compilation step has not been fully implemented. Since a compiler requires thinking on another level, effectively creating a program that creates programs, it is useful to create the change without this difficulty at first. However, a weakness of this approach is the risk of doing something in the hand-compiled code that is impossible to do in the compiler, or just something that could have been done in a better way in the compiler. However, we did not encounter these problems.

There was another problem with our approach, however. When we implement something in the hand-compiled source code for `Fib`, it does not test all the programming constructs. This lead to the pitfall of thinking that a change was much simpler to implement than it was in reality. For example, removing Jgo from the compiler is extremely simple for a flat program such as `Fib`, but when the control flow is nonlinear, this is much more problematic, as described in Chapter 7. In fact, we had a working `Fib` program that did not use `goto` statements in less than ten minutes, while the general-purpose JCilk compiler without Jgo we created took many hours of implementation, and is even limited to `if` and `while` constructs. This is an example of where there is a large mismatch between the amount of work required in the two phases, especially because of the fact that removing Jgo did not require any change to the runtime system. Even with this weakness, however, the approach is still better than trying to implement everything at once, which results in a lot of untested code being run at once. Dividing it into these two phases causes the overhead of the hand-implementations, which are of course redundant once the compiler is working, but this overhead is small compared to the advantage of having a working runtime system before starting the compiler-side implementation.

When we read the theses of Danaher and Lee, we were skeptical about their claims that GoJava was a required intermediary:

> *To support a continuation mechanism, I created an intermediate language called GoJava.* [10, Page 58]
> *Such support for goto is necessary to include a low-overhead continuation mechanism into Java, which in turn is necessary for the thread migration mechanism.* [9, Page 42]

We have shown that it is possible to simplify the compiler architecture and remove the GoJava intermediary language, and while this did introduce some minor overhead, it is worth the change, since the JCilk distribution no longer requires the end user to compile a modified Java compiler, but rather to compile JCilk into Java with a provided compiler, and then use any normal Java compiler to compile into Java bytecode. This resulted in a large improvement to JCilk, and we think that choosing to work on this area was a good choice.

We also worked on lowering the frame allocation overhead. This was the single largest source of overhead in JCilk, and we had ideas about how we could lower it significantly. While our first idea, the memory manager, was not successful, the second one, frame reuse, was. We presented the caveat that it only works for recursion, which is also where the largest source of overhead is.

However, this is not entirely true. We have concluded after some discussion that it is possible to optimize the frame allocation for other types of repetitive method calls such as loops and mutual recursion. However, this does require some modification of our implementation. This modification will mean that the frame allocation overhead is significantly reduced for all types of repetitive method calls. With this change, the frame allocation overhead would be reduced for all programs where it is needed, since the overhead is not a major contributing factor to computation time unless methods are called repetitively.

9.4 Testing

There are a couple of issues which question the quality of our test results. First, there is a basic result viability issue when comparing files compiled with an old compiler to those compiled with a new compiler. Some of the performance gain shown by using the new compiler will not be the result of our reuse mechanism, but merely the result of us being able to compile with the new Java compilers after removing the need for Jgo. In order to figure out more precisely how big the gain of implementing reuse is, we could have compiled the reuse system with the modified GJC compiler which they use for compiling Jgo. Doing such a test can possibly give a better indication of exactly how much performance gain the reuse mechanism gives. This would enable us to make a fair comparison to the original JCilk system, instead of having the system without Jgo as the most viable comparison.

Another interesting addition that could be made to our tests in order to get a better idea of the actual improvement gained from the reuse system, is using an old *Java Runtime Environment* (JRE). Testing the reuse system in the Java 1.4 JRE can give more clear results as to how efficient the reuse mechanism is for older Java systems. However, this is not very useful, since there is no realistic reason to use such outdated systems now.

A more complex test suite script could have ensured more viable results on the shared application server. With the current script, each test case is run 25 times in a row. To better accommodate for the changing load on the server, all tests could be performed in a completely random order. This would not make the results as prone to the varying load on the system while doing a specific test, as all tests would be evenly affected by the reduction in processing capabilities.

9.5 Adding a Third Compilation Step

This section evaluates on taking the approach of adding a compilation pass to the JCilk compiler to remove the need for GoJava. By doing so, there is currently little interference with the original JCilk system. This makes it easier for other developers that already know the JCilk project code to continue development, as they in many cases do not need to understand our code. Another advantage in this prototype version of the compiler is that inserting the addition at the end of the compiler did not require us to understand as much code as would be necessary if the solution was inserted as a more integrated part of the original compilation flow. The biggest disadvantage is that the current solution needs the GoJava compilation step, instead of having replaced it completely in the compilation process. Since we have shown that GoJava in unnecessary, a full fledged implementation of the system would remove the need for GoJava altogether. All in all, the addition of the extra compilation step has been a good choice as it ensured a possible implementation of the system within our scope, while still showing that GoJava is unnecessary.

9.6 Working with Polyglot

As with other similar frameworks, it took some time to learn the flow of the compiler but when a general understanding was gained, the structural decisions in the compiler seem intuitive and carefully thought out. The biggest disadvantage is the fact that it is coded in Java 1.4 which means that features such as generics are not used. To accommodate some of the shortcomings in Java 1.4, a class such as `TypedList` was created in Polyglot. It is essentially a decorator for the `List` datatype in Java, which is initialized with a certain class specification. If you try adding an element of a different type, it throws an exception. This class is essentially a way to incorporate some of the advantages of generics. However, doing an actual upgrade to the framework so it uses generics would facilitate finding a lot of the errors at compile-time rather than runtime.

Chapter 10

Future Work

This chapter presents some suggestions for future work in the area. The suggestions are all based on continuing from where this project ends. There are many aspects to examine in the area, but we have chosen the ones that we believe are the most interesting at this point.

10.1 Continue the Implementation

One suggestion we make is to continue implementation from where this project ends. This means completing the simplified (Jgo-less) compiler architecture by implementing the missing constructs mentioned in Chapter 7. Our prototype implementation has shown that it is possible to take this step, so all that remains is to complete it. Completing this allows JCilk to be distributed in a considerably smaller package, a factor of 5 at least, and also means that the package is ready to use rather than requiring compiling the modified Java compiler in order to use it. Additionally, since the JCilk distribution is now pure Java, the distribution is inherently cross-platform.

Additionally, there are some improvements that could be made to the frame reuse system to increase performance further. Currently, the system only supports repeated or recursive calls made to the same method, but with some modification, it would be able to support mutual recursion and repeated calls to any number of methods as well, enabling optimization for programs which are not currently able to take advantage of frame reuse.

10.2 Reduce Synchronization Overhead

While completing the Jgo-less architecture is important to improve the system, the frame reuse system is already capable of significant performance improvement, and perhaps it would be better to start looking at the other sources of overhead rather than optimizing the reduction of one source. The other major source of overhead in the runtime system is the synchronization overhead, and finding a way to greatly reduce this could provide a large performance gain to JCilk programs.

Danaher and Lee both present some ideas as to how this can be done [9, Section 5.3][10, Chapter 6], but there are many ways to go about this. The project would need to examine several approaches before choosing one or two to implement. Alternatively, this project can be combined with the previous one mentioned in Section 10.1 above.

10.3 Alternative Research

Another possibility is to step back and take a look at the larger picture. Currently, there is a lot of research going on within the field of Cilk-inspired OOP languages or language features. Perhaps it would be interesting to examine an aspect that the other projects are not necessarily considering, in order to benefit the other researchers as well, or to gain a competitive edge over them. One such possibility would be to consider alternatives to work-stealing, perhaps finding one that requires less synchronization, thus eliminating the currently greatest source of overhead. Another possibility would be to look at ways to integrate Cilk methods with Java methods such that they are able to be called both ways. This would make JCilk programs much more generally attractive, as it would enable all Java programs to use JCilk libraries, for example, or even allow mixing of Java and JCilk programs, much like the embedded code segments we know from C#.

Chapter 11

Conclusion

This chapter concludes on the report by reviewing each of the goals listed in Section 2.1 and concluding on whether we have reached that goal or not. Finally, we summarize the significant contributions that we have made in this project.

- Examine related work and identify common problems in this area.

Thanks to our SW9 project, we had a good idea about where to start in this area. We examined five different languages and/or projects, and we learned a lot about the common ideas within the task-based, or fork/join, type of concurrency model. This also served as a good preliminary step to choosing a project to continue work on.

- Choose a related work project to develop upon, or define our own project.

Once we were done with the examination, this was a simple task, as JCilk was the only real choice. This lack of options did not end up hurting us, however, since JCilk was a good project to work on, despite the minor problems we did encounter with it.

- Analyze the chosen project and its primary problems, in order to pinpoint our focus area(s) for further development of it.
- Define a plan for improving upon the project by solving one or more of its primary problems.

These two goals are put together here, since they are so closely related. First, we looked at the problems in JCilk and the possibilities for solutions, as well as some other ideas for improvement. We looked at suggestions from the creators of JCilk, as well as created our own ideas. In the end, we chose only to proceed with our own ideas, and both ideas were involved with improving upon the JCilk framework. None of the ideas involved changing the language itself, and the best place to improve JCilk is in the framework, which has some weaknesses, two of which we have worked on.

- Describe and implement solutions for the selected problem(s).

We started implementing the memory manager, but realized during development by preliminary tests that it was not working as well as we had thought. Handling memory usage in such a low-level manner in Java proved to be too troublesome. However, we then implemented our other idea of frame reuse, and it proved to be not only more intuitive and high-level in its memory model, but also simply worked better. We also implemented a prototype and showed that JCilk can indeed work without the intermediary GoJava language and, more importantly, without the Jgo modified Java compiler, which reduces the space required for the JCilk compiler considerably. The original system used 1500Mb of space, where the Jgo-less system takes up 28.3Mb, and that is including both the source files and compiled binaries. The only difference is that the system now requires a Java compiler, but users who have a JRE can be assumed to have a Java compiler.

- Design test(s) for the implemented solution(s) in order to verify that they work as intended.
- Perform the test(s), and discuss the results.

This was done, and we showed that the frame reuse system improved the performance by a reasonable margin. Additionally, we showed that the removal of the GoJava intermediary language from the system did not cause a noticeable performance loss.

- Evaluate and conclude upon the project.

This was done in Chapter 9 and in this chapter. We have discussed some of our choices, and found some ways we might have made better choices, perhaps ending up in a better result. However, some mistakes will be made during every project, and our mistakes did not prevent us from reaching good results. We have shown that it is possible to improve the performance of JCilk noticeably, while reducing the space required remarkably, by a factor greater than 50, without reducing the functionality in any way. While the GoJava-free implementation is not complete, we have shown that it is possible to do so.

The final goal, to present proposals for future work in the area, is completed in the next chapter.

11.1 Contributions

These are the most important contributions made in our master's thesis:

Reusing Frames in JCilk A system for reusing the frames in JCilk, instead of continuously creating new frames.

Supporting Java Continuations After successfully creating an algorithm which facilitates continuations in Java, the need for GoJava has been eliminated. The algorithm can possibly be used for other systems which require continuation support in Java.

These are the most important experiences gained by the project group:

Expanding other Projects The work on expanding the functionality of JCilk has been a great experience for the project group, as it has improved our ability to understand and expand the code of other developers.

Experience with Compiler Principles The project group's knowledge about the general mechanisms used in compilers has been expanded significantly. We have also gained further first-hand experience in working with compilers.

Research and Decision Making We have in a couple of situations taken decisions based on a narrow research foundation. One example of this is that we started examining compiler tools before properly examining if a language like the one we wanted to create already existed. In future projects, we will focus on finding the right ratio between research and decision making.

11.2 Final Words

We have improved upon the JCilk framework in two ways, and suggested other areas for improvement, as well as given suggestions for future work on JCilk and on Cilk-inspired languages.

With its elegant concurrency model inspired by Cilk, and the modern programming features from Java, we think that JCilk is a viable alternative to the standard thread model in Java. While it will take some adjustment to get used to this model of programming, this will be required anyway since both Microsoft and Sun are working on similar models in the upcoming versions of C# and Java. We think that JCilk is also a viable alternative to these upcoming versions, as it has the advantage of being an extension rather than a library, meaning that it uses new syntax rather than using the library-call method that Microsoft and Sun use.

Bibliography

[1] Allan B. Christensen, Martin Skou, and Morten Friis. Examining concurrency in languages. Project report, Aalborg University Department of Computer Science, January 2009.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* McGraw-Hill Science/Engineering/Math, July 2001.

[3] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[4] Sun Microsystems. New features in java 5. `http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html#threads`, 2004.

[5] Sun Microsystems. New features in java 6. `http://java.sun.com/javase/6/webnotes/index.html`.

[6] Sun Microsystems. New features in java 7. `http://www.slideshare.net/gal.marder/whats-expected-in-java-7-1116123`, 2009.

[7] Channel 9. Parallel programming for managed developers with visual studio 2010. `http://channel9.msdn.com/pdc2008/TL26/`, 2009.

[8] Massachusetts Institute of Technology. The cilk project. `http://supertech.csail.mit.edu/cilk/index.html`, 2009.

[9] John S. Danaher. The jcilk-1 runtime system. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, June 2005.

[10] I-Ting Angelina Lee. The JCilk multithreaded language. Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science, August 2005.

[11] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. pages 137–150.

[12] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, November 2001.

[13] Oliver Yang. Comparison of solaris, linux, and freebsd kernels. `http://cn.opensolaris.org/files/solaris_linux_bsd_cmp.pdf`.

[14] Yuli Zhou Michael Halbherr and Chris F. Joerg. MIMD-style parallel programming with continuation-passing threads. In *Proceedings of the 2nd International Workshop on Massive Parallelism: Hardware, Software, and Applications*, Capri, Italy, September 1994.

[15] Chris Joerg and Bradley C. Kuszmaul. Massively parallel chess. In *Proceedings of the Third DIMACS Parallel Implementation Challenge*, Rutgers University, New Jersey, October 17–19 1994.

[16] Cilk Pousse Team. Cilk pousse. `http://people.csail.mit.edu/pousse/`.

[17] John S. Danaher, I-Ting Angelina Lee, and Charles E. Leiserson. The jcilk language for multithreaded computing. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, San Diego, California, October 2005.

[18] Alex Miller. Year in review: What to expect in java se 7. `http://www.javaworld.com/javaworld/jw-12-2008/jw-12-year-in-review-2.html`.

[19] Doug Lea. A java fork/join framework. In *Java Grande*, pages 36–43, 2000.

[20] Daniel Moth. Parallel extensions to the .net framework. `http://www.vsj.co.uk/articles/display.asp?id=704`.

[21] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[22] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *In 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.

[23] Eric Allen, David Chase, Christine Flood, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, and Guy L. Steele Jr. Project fortress - a multicore language for multicore processors. *linux Magazine*, September 2007.

[24] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[25] Dr. Heinz M. Kabutz. Determining memory usage in java. `http://www.javaspecialists.co.za/archive/Issue029.html`.

[26] Technical University of Munich. Java cup parser library. `http://www2.cs.tum.edu/projects/cup/`, 2009.

Appendix A

Basic Concepts

This appendix presents a number of concepts used in the report. The concepts described here have ambiguous or confusing definitions, and are explained explicitly to avoid any misunderstandings when reading the report.

Thread

Threads are created by a process within the system and represents an abstraction which facilitates splitting the work performed by a process into several smaller pieces of code which can be run concurrently.

Concurrency

Concurrency is used in this report as a property of a language which allows execution of several pieces of code simultaneously. This does not necessarily mean that the code is being executed in parallel at the same time, but merely that the code can be split up and run by e.g. the use of context switches.

Concurrency Construct

A concurrency construct is a feature in a language which facilitates concurrency in the language. In the concurrency model used in Java, this includes constructs such as monitors and threads.

Concurrency Model

The concurrency model of a language denotes the language constructs implemented to support concurrent programming. In the case of a language such as Java, the concurrency model is based on the context of threads, with the advantages and disadvantages this brings.

Programmability

This term was defined in the SW9 project report [1, PP 12], and the same definition is inherited in this report. Programmability stems from the word program which is defined as a number of coded instructions that enables a machine to perform a desired sequence of operations. In our report this is realized through programming languages such as Java. In this report programmability denotes the difficulty with which these programs can be made, which in our case e.g. can be used as a way to compare languages.

Task

In this report tasks are referred to as an abstract pieces of programming work that need to be carried out. Furthermore, tasks in the context of Cilk and JCilk refer to a piece of work which independently can be run on a thread without necessarily having any affiliation with other threads.

Synchronization

When used in this report, synchronization refers to the handshake that threads or processes make in order to join up a certain point of execution. Synchronization can be achieved by using explicit concurrency constructs such as monitors or locks.

Appendix B

Clone Example

All of the examples in this appendix are taken from [9].

```
private cilk int fib(int n) {
   int x, y;
   if(n < 2) {
      return n;
   }

   x = spawn fib(n-1);
   y = spawn fib(n-2);

   sync;
   return x + y;
}
```

Listing B.1: The source code of the Fib method.

```
private int fib(Worker worker, int n, int returnEntry) {
   int x, y;

   Fib_fib_frame thisFrame = new Fib_fib_frame(n, 0, returnEntry);
   worker.pushFrame(thisFrame);

   if( n < 2 ) {
      return n;
   }

   x = this.fib(worker, n-1, 1);
   if(worker.popFrameCheck(new Integer(x))) {
      return 0;
   }

```

```
    thisFrame._x = x;
    y = this.fib(worker, n-2, 2);
    if(worker.popFrameCheck(new Integer(y))) {
       return 0;
    }

    return x + y;
}
```

Listing B.2: A simplified version of the fast clone of the compiled `Fib` method.

```
private void fibSlow(Worker worker, CilkFrame frame) {
    int tmp;
    Fib_fib_frame thisFrame = (Fib_fib_frame)frame;

    switch(thisFrame._pc) {
       case 1:
          goto _cilk_sync1;
       case 2:
          goto _cilk_sync2;
       case 3:
          goto _cilk_sync3;
       }

    _cilk_sync1: ;

    thisFrame._pc = 2;
    tmp = this.fib(worker, thisFrame._n-2, 2);
    if(worker.popFrameCheck(new Integer(y));
       return;
    } else {
       thisFrame._y = tmp;
    }
    _cilk_sync2: ;

    thisFrame._pc = 3;
    if(!worker.sync()) {
       return;
    }
    _cilk_sync3: ;

    retVal = x + y;
    worker.setReturnResult(new Integer(retVal));
    return;
}
```

Listing B.3: A simplified version of the slow clone of the compiled `Fib` method.

Appendix C

Object Allocation Test

The following class shows the test used to confirm that allocating big objects with array structures defining a certain number of integers is faster than creating an single object for each data set.

```java
import java.util.Date;

public class ObjectAllocationTest {
   private class SmallObject {
      private int i = 0;
      private int j = 0;
      private int k = 0;
   }

   private class BigObject {
      private int[] i = new int[10000];
      private int[] j = new int[10000];
      private int[] k = new int[10000];
   }

   public static void main(String[] args) {
      ObjectAllocationTest test = new ObjectAllocationTest();

      // warm up JVM
      for (int i = 0; i < 100000000; i++) {
         new Object();
      }

      // create small objects
      long time = new Date().getTime();
      for (long i = 0; i < 100000000; i++) {
         test.new SmallObject();
      }
      System.out.println("Allocating small objects:" + (new Date().getTime()
            - time));

        // create big objects
        time = new Date().getTime();
        for (long i = 0; i < 10000; i++) {
            test.new BigObject();
        }
        System.out.println("Allocating big objects:" + (new Date().getTime() -
            time));

    }
}
```

Listing C.1: The source code of the performance test class.

Appendix D

Object Access Test

The following class shows the test used to determine the overhead associated with hashing into an array of objects with a simple expression rather than accessing the object variables directly. The access expression is similar to the one we expect is required for the manager solution.

```
import java.util.Date;

public class ObjectAccessTimeTest {

   private class SmallObject {
      public int i = 0;
      public int j = 0;
      public int k = 0;
   }

   private class BigObject {
      public int[] i = new int[10000];
      public int[] j = new int[10000];
      public int[] k = new int[10000];
   }

   public static void main(String[] args) {
      ObjectAccessTimeTest test = new ObjectAccessTimeTest();

      // warm up JVM
      for (int i = 0; i < 100000000; i++) {
         new Object();
      }

      // access small object
      ObjectAccessTimeTest.SmallObject smallObject = test.new SmallObject();
      long time = new Date().getTime();
      int val = 0;
```

```
        for (long i = 0; i < 1000000000; i++) {
            val = smallObject.i;
        }
        System.out.println("Accessing small objects:" + (new Date().getTime()
             - time));

        // access big object with expression
        ObjectAccessTimeTest.BigObject bigObject = test.new BigObject();
        int rowSize = 10;
        int col = 5;
        int row = 4;
        time = new Date().getTime();
        for (long i = 0; i < 1000000000; i++) {
            val = bigObject.i[rowSize*row + col];
        }
        System.out.println("Accessing big objects:" + (new Date().getTime() -
             time));

    }
}
```

Listing D.1: The source code of the access time test class.

Appendix E

Handle_level Java Implementation

The following Listing shows the `handle_level` pseudo code function implemented in Java. The rest of the methods implementations can be found in the file `/jcilk/src/polyglot/ext/jcilk/ast/C` on the Appendix CD.

```java
private List remove_jgo_need(NodeFactory nf, TypeSystem ts, JCilkUtil util
    ,List stmts) {
  TypedList switchElements, switchStmts;
  switchElements = new TypedList(new LinkedList(), SwitchElement.class,
     false);
  addCase(switchElements, 0, stmts, util, nf);
  Labeled_c label;

  while((label = (Labeled_c)findLabel(switchElements)) != null) {

    int entry = -1;
    int exit = -2;
    Stmt nextStmt = null;
    Object[] res = null;
    int outerCaseValue = -1;
    int pcStmt = -1;
    Node_c parentBlock = null;
    int outerSwitchElement = parents.size()-2;
    for (int i = outerSwitchElement; i > 0; i--) {
     // if outermost case
     if(i == outerSwitchElement) {
        String caseValue = parents.get(parents.size()-1).toString();
        outerCaseValue = Integer.parseInt(caseValue.substring(5,
            caseValue.length()-1));
        entry = outerCaseValue;

        pcStmt = getPcStmtDefinition(cleanCodeBlock((Block)parents.get(
            parents.size()-2)));
        if(pcStmt != -1) {

             if(pcStmt > numPredefinedLabels || pcStmt == -2) {
                //System.out.println(pcStmt);
                exit = pcStmt;
                parents.set(outerSwitchElement, removeFirstPcStmtInSwitchBlock((SwitchBlock_c)parents.get(outerSwitchElement)));
                //System.out.println(parents.get(parents.size()-2));
             }
             //System.out.println(pcStmt);
             //System.out.println("pcStmt: " + pcStmt + " outerCaseValue: " + outerCaseValue);
          }
       }
       Stmt currStmt = nextStmt==null?parents.get(i):nextStmt;
       Stmt childStmt = parents.get(i-1);
       Stmt childChildStmt = i==1?null:parents.get(i-2);

       res = handleLevel(currStmt, childStmt, childChildStmt, entry, exit, nf, util, switchElements, parentBlock);
       if(i == outerSwitchElement) {
          removeCase(switchElements, outerCaseValue, util, nf);
       }

        nextStmt = (Stmt)res[0];
        entry = (Integer)res[1];
        exit = (Integer)res[2];
      }
   }
   return switchElements;
}
```

Listing E.1: The `handle_level` pseudo code function from Listing 7.4 implemented in the JCilk framework

Appendix F

JCilk Test Code

F.1 Fib

```java
/**
 ** Copyright (C) 2005, 2007 John Danaher, I-Ting Angelina Lee
 **
 ** This file is part of JCilk.
 **
 ** JCilk is free software: you can redistribute it and/or modify it under the
 ** terms of the GNU General Public License as published by the Free Software
 ** Foundation, either version 3 of the License, or (at your option) any later
 ** version.
 **
 ** JCilk is distributed in the hope that it will be useful, but WITHOUT ANY
 ** WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 ** FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 ** details.
 **
 ** You should have received a copy of the GNU General Public License along with
 ** JCilk.  If not, see <http://www.gnu.org/licenses/>.
 **
 **/

import jcilk_stub.*;

public class Fib {
  cilk public static void main(String args[]) {
    if( args.length != 1 ) {
      System.err.println("Usage: Fib <n>\n");
      System.exit(0);
    }

    int n = Integer.parseInt(args[0]);
    Fib f = new Fib();
    int res = 0;
    res = spawn f.calc(n);
    // entry = 1 here
    sync; // can be implicit
    // entry = 2 here

    System.out.println( "Solution: The fib number for " + n + " is " + res )
        ;
  }

  cilk private int calc(int n) {
    if( n < 2 ) {
      return n;
    }

    int x = spawn this.calc(n-1);
    // entry = 1 here
    int y = spawn this.calc(n-2);
    // entry = 2 here
    sync;
    // entry = 3 here

    return (x + y);
  }
}
```

Listing F.1: Fib

F.2 Queens

```
/**
 ** Copyright (C) 2005, 2007 John Danaher, I-Ting Angelina Lee
 **
 ** This file is part of JCilk.
 **
 ** JCilk is free software: you can redistribute it and/or modify it under
     the
 ** terms of the GNU General Public License as published by the Free
     Software
 ** Foundation, either version 3 of the License, or (at your option) any
     later
 ** version.
 **
 ** JCilk is distributed in the hope that it will be useful, but WITHOUT ANY
 ** WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS
 ** FOR A PARTICULAR PURPOSE.  See the GNU General Public License for more
 ** details.
 **
 ** You should have received a copy of the GNU General Public License along with
 ** JCilk.  If not, see <http://www.gnu.org/licenses/>.
 **
 **/

import jcilk_stub.*;

public class Queens {
    private final int n;
    private volatile static int result = 0;

    public Queens(int n) {
        this.n = n;
    }

    private synchronized void count(){
      result++;
    }

    private boolean safe(int[] q, int n) {
        for (int i = 0; i < n; i++) {
            if (q[i] == q[n])             return false;   // same column
            if ((q[i] - q[n]) == (n - i)) return false;   // same major diagonal
            if ((q[n] - q[i]) == (n - i)) return false;   // same minor diagonal
        }
        return true;
    }

    cilk private void nqueens(int[] config, int i) {

      if (i == n){
         count();
         return;
      }
          int j = 0;
          while(j < n){
             int[] nconfig = new int[n];
                System.arraycopy(config, 0, nconfig, 0, n);
                nconfig[i] = j;

                if(safe(nconfig,i)) {
                  spawn nqueens(nconfig, i+1);
                }

                 j++;
          }
```

```
            sync;

     }

     cilk public static void main(String argv[]) {

    if(argv.length < 1) {
        System.err.println("Usage: Queens <n>");
        return;
    }

    int n = Integer.parseInt(argv[0]);

    System.out.println("Running queens " + n);

    int[] config = new int[n];
    spawn (new Queens(n)).nqueens(config, 0);
     sync;

    if(result != 0) {
        System.out.println("Solutions: " + result);
    } else {
        System.out.println("No solutions.");
    }
     }
}
```

Listing F.2: Queens

F.3 QuickSort

```
import java.util.Random;
import jcilk_stub.*;

public class QuickSort {
   static int[] array = null;
    static int threadNumbers = 0;
    static int n = 0;

    cilk public static void main(String[] args)
    {

      if(args.length < 2) {
          System.err.println("Usage: QuickSort <n> <s>");
          return;
      }

        n = Integer.parseInt(args[0]);
        int seed = Integer.parseInt(args[1]);
        array = new int[n];
        Random rand = new Random(seed);
        int max = (n < 1000) ? 1000 : n;
        for (int i = 0; i < n; i++)
        {
            array[i] = rand.nextInt(max);
        }
        spawn new QuickSort().quicksort(0, n - 1);
        sync;
    }

    private void swap(int x, int y)
    {
        int t = array[x];
        array[x] = array[y];
        array[y] = t;
    }

    private int pivotFinderAndSorter(int left, int right, int pivot)
    {
        int pivotValue = array[pivot];
        swap(pivot, right);
        int temp = left;
        for (int i = left; i <= right; i++)
        {
            if (array[i] < pivotValue)
            {
                swap(temp, i);
                temp++;
            }
        }
        swap(right, temp);
        return temp;
    }

    cilk private void quicksort(int left, int right)
    {
        if (right > left)
        {
            int pivot = (right + left) / 2;

            int newPivot = pivotFinderAndSorter(left, right, pivot);

            spawn quicksort(left, newPivot - 1);
            spawn quicksort(newPivot + 1, right);

            sync;
       }
    }
}
```

Listing F.3: QuickSort

Appendix G

Resume (Danish)

Denne rapport omhandler arbejdet på speciale semestret udført af Allan B. Christensen, Martin Skou og Per S. Stilling på Institut for Datalogi på Aalborg universitet i forskningsenheden Database- og programmeringsteknologier.

Specialet bygger på et forspeciale der undersøgte fire forskellige sprog, C#, Cilk, Fortress and Scala, der hver især håndterer understøttelse af samtidighed på forskellige måder. Konklusionen af undersøgelsen var, at de principper der ligger til grund for den til dels implicitte samtidghed som Cilk understøtter igennem work-stealing og dens spawn/sync syntaks, giver et passende kompromis mellem hvad programmøren skal definere og hvad der implicit skal håndteres af programmeringssproget.

I specialet har vi valgt at kigge videre på sprogkonstruktioner og systemer der minder om dem brugt i Cilk, for at finde et objekt orienteret sprog eller bibliotek som vi kan arbejde videre på, eller inspirere os fra i opbygningen af et nyt objektorienteret system. Udover Cilk, gennemgår vi fire sprog som vi mener kan inspirere til vores arbejde i dette semester; JCilk, en Java implementation af principperne brugt i Cilk, Java Fork/Join Frameworket, et Java-bibliotek opbygget på mange af de samme principper der er brugt i Cilk og JCilk, såsom work-stealing og fork/joining af beregningerne, C#'s nye Task Parallel Library, et bibliotek til C# som ligeledes bygger på principperne bag Cilk samt understøttelse af andre samtidighedskonstruktioner såsom futures, og til sidst Googles MapReduce framework, der bygger på lidt anderledes principper idet det i høj grad er inspireret af at parallelisere MapReduce operationen fra funktionelle sprog.

Efter undersøgelsen af de eksisterende lignende systemer på området, vælger vi at fortsætte arbejdet på JCilk frameworket. Det vælges ud fra den iagtagelse at både Java og C#'s to nye biblioteker er under udarbejdning for øjeblikket og arbejde inden for disse rammer er derfor uhensigtsmæssigt. Abstraktionen som MapReduce frameworket tilbyder, bliver vurderet som værende på et for lavt niveau i forhold til vores mål. JCilk frameworket stemmer derimod overens med de egenskaber vi foretræker i forbindelse med valg af et system, som vi kan arbejde videre på; det er en sprog udvidelse, hvilket giver et alternativ til de biblioteker som der bliver arbejdet på hos Microsoft og Sun, og systemet er en direkte udbygning af Cilk i et objektorienteret sprog, hvilket passer med målet for projektet.

Efter valget af JCIlk, er næste skridt at undersøge den nuværende status af JCIlk-1, som det nuværende system kaldes. Et antal af forslag til forbedring og/eller observeringer af nuværende mangler i systemet bliver identificeret, både på baggrund af gruppens egne iagtagelser og baseret på de forslag der bliver givet i de rapporter der er udgivet i forbindelse med JCilk. Der bliver i forbindelse med denne del af rapporten fremlagt følgende større problemer med det nuværende system; sammenhængen mellem JCilk og Java er på nuværende tidspunkt kun en-vejs og det er ikke muligt at kalde JCilk fra Java kode, det bliver endvidere identificeret at den nuværende to-dels kompileringsprocess formodentligt er unødvendig. En anden optimering der kan laves til det nuværende system er at bruge nogle af de mere moderne sprog funktionaliteter i Java, der er introduceret efter 2004. Det sidste større problem som bliver identificeret, er at systemets performance på nuværende tidspunkt bliver svækket meget af objekt allokeringen og synkroniseringen i JCilks runtime system. Vi vælger at implementere to af disse forbedringer; afskaffelse af det sidste skridt i kompileringsprocessen af JCIlk, og optimering af den nuværende allokering i systemet.

Forbedringen af allokeringen i JCIlk bliver realiseret ved at modificere JCIlk således, at de objekter der ikke længere er i brug bliver gembrugt i stedet for at lade dem blive fanget af garbage collectoren.

Den anden forbedring, som består i at eliminere nødvendigheden af sproget GoJava i kompileringsprocessen, bliver implementeret ved at omdanne de `goto` statements og labels der bliver brugt i systemet til Java kode.

Den vigtigste faktor i forbindelse med en test af begge forbedringer er hvorvidt de har forårsaget en betydelig ændret i JCilks ydeevne. Der bliver derfor efter implementationen, lavet en test af den ændring som begge forbedringer har forårsaget i systemet. Konklusionen er, at elimineringen af GoJava ikke har haft bemærkelsesværdig negativ betydning. Det påvises endivdere at elimineringen af den høje objektallokerings frekvens har forbedret ydeevnen i JCilk betydeligt.

Konklusionen af projektet er, at vi med succes har lavet to forbedringer til JCilk som begge har forbedret systemet betydeligt.