Optimisation of Java Threads

MEng Individual Project Report
June 12, 2002

Andreas Martens

Project Supervisor: Dr Tony Field
Second Marker: Dr Paul Kelly
Abstract

The use of models to simulate and predict real world behaviour is becoming increasingly important to reduce development costs and locate flaws in systems. The main issue with these simulations is that they can be rather time-consuming.

This project aims to increase the performance of process-oriented simulations by an order of magnitude to be comparable to the performance of event-driven simulations.

A system has been built to perform this optimisation which translates Java class files into optimised class files. The correctness of the system has been verified and the and quantitatively evaluated, showing that optimised programs run 10–30 times faster than the original
Acknowledgements

I would like to thank my supervisor Dr Tony Field for invaluable assistance, brilliant ideas and time. Kwok Yeung for insights into the strange world of SOOT and further thanks to the MEng “sync” group for advice, comments and humour.
Table of Contents

1 Introduction .................................................. 1
  1.1 Objectives ............................................... 2
  1.2 Report Structure ...................................... 3
  1.3 Equipment ............................................. 3

2 Background .................................................. 4
  2.1 Simulation ............................................. 4
  2.2 Simulation toolkit .................................... 5
  2.3 Coroutines ........................................... 9
  2.4 Transformation ....................................... 9
  2.5 Java .................................................. 14
  2.6 Flyweight Pattern .................................. 19

3 Implementation .............................................. 20
  3.1 Changes to Toolkit .................................. 21
  3.2 Target Event Classes ................................ 22
  3.3 Block Structure ...................................... 23
  3.4 Translation process ................................... 26
  3.5 Auxiliary classes .................................... 36

4 Testing ...................................................... 37
  4.1 Test Cases ............................................ 37
  4.2 Correctness .......................................... 43

5 Evaluation .................................................. 45
  5.1 Overheads ............................................. 45
  5.2 Test-cases ........................................... 47
  5.3 Overall ............................................... 54

6 Conclusion .................................................. 56
  6.1 Further Work ......................................... 57

A Using the system ........................................... 59
  A.1 Installation .......................................... 59
  A.2 Running .............................................. 60
List of Figures

2.1 Virtual time-line for a Single Server Queue (Event Driven) (From [Fie]) .............................................. 5
2.2 Virtual time-line for a Single Server Queue (Process Driven) (From [Fie]) .............................................. 6
2.3 The main classes in the Simulation Tools Classes ............. 7
2.4 Translating coroutines to a single process .................... 10
2.5 A simple naive translation of a single process ............... 11
2.6 The main steps in a SOOT optimisation ...................... 17

3.1 New class structure ........................................... 23
3.2 Initial fragmentation of Arrival Process ..................... 24
3.3 Overview of the Optimisation Process using a BodyTransformer 27
3.4 Overview of the Optimisation Process using a SceneTransformer 28
3.5 Multiple calls to the same method .......................... 31
3.6 Class diagram of translated Single Server Queue ........... 35

5.1 Basic Test Case .............................................. 48
5.2 Single Server Queue ........................................ 50
5.3 Queueing Network .......................................... 52
5.4 Repair Model ................................................ 53
5.5 CPU model .................................................. 54
Chapter 1

Introduction

Simulation and modelling has a large range of applications, both academically and industrially, the usage of simulation and modelling ranges from simulating the traffic on a local network to determine where performance bottlenecks are to studying the effects of a new system not yet created. Thus application of simulation can be a real benefit to development of new systems.

There are two main simulation frameworks available; event-scheduling and process-oriented systems. The event system places “events” on a virtual time line and executes their code in order, whilst the process oriented system creates objects which interact with each other using blocking calls. These are both used a lot as the process oriented systems are easier to create and have a more logical structure, but the event systems are far faster, due to that the process oriented system needs to generate a thread per process and thus will be hit by overheads associated with threads. Thus many people are forced to create event-driven simulations when it would be easier to create an equivalent process oriented version.

In theory it is possible to transform one type of simulation to another, this would allow users to harness the speed of event driven simulations with the
1.1. OBJECTIVES

Java Thread Optimisation

ease of process-oriented simulations. This project started out with an aim of increasing the performance to be at least an order of magnitude faster, approaching that of event driven simulations. This was achieved, although the event driven simulations are still twice as fast in some cases. On average a speedup in the range of 10 to 30 was achieved in the simulation test cases.

The system is aimed at the end user so that they create their simulation (and check that it works) but before running it at length, they run it through the optimisation program which will output optimised classes. Since large scale simulation can sometimes take days, an order of magnitude shorter running time will make optimised process oriented simulations a viable alternative to event driven versions.

The system is available for download from
http://www.doc.ic.ac.uk/~am298/project

1.1 Objectives

The project has several aims, the most important of these are:

1. Investigate continuation passing techniques for thread reduction in Java Simulations.
2. Implement a system to perform thread reduction on Java Byte-code, for a specific Simulation API.
3. Provide an evaluation of the correctness and benefits of using continuation passing techniques to optimise processes.
4. Investigate the possibility of creating a general java coroutine optimiser
1.2 Report Structure

The following background section provides introductory information on the different systems and algorithms used, the implementation details the design decisions and workings of the program and the evaluation chapter checks that the program works both quantitatively and qualitatively. For people interested in the simulation aspect of the work, sections 2.1, 2.2, 3.1 and chapters 5 and 6 are probably of highest relevance. The sections 2.4 and 2.5 and chapters 3-6 will be of interest to people interested in program transformation.

1.3 Equipment

The project was written on machines of various speed, running SuSe Linux 7.2. JVMs used in programming and testing the system include IBM, Sun and Blackdown 1.3 and Sun 1.4. SOOT version 1.2.3 is required for the operation of the program and the tests were invoked by Perl. The code and report was written using emacs and \LaTeX. The graphs were generated by gnuplot and pstricks modified by hand, the diagrams by dia and xfig.
Chapter 2

Background

2.1 Simulation

A basic knowledge of simulation systems is assumed. The (very) short summary here is mostly taken from the Course Notes [Fie], the Handbook of Simulation [Ban98] and Ricki G. Ingalls Introduction to Simulation [Ing01].

There are four main methods of simulation, Event-driven, Process-Interaction, Activity Scanning and the Three-Phase method. Of these four, the two first are the most widely applied and the ones we will be concerned with.

Event Scheduling is based on having a virtual time-line (e.g. for an Single Server Queue, Fig 2.1) on which one progresses, thus if there is no activity occurring at the moment, one advances time along the line to the next event. This means that one can perform a simulation with a minimum of fuss.

Process interaction is a more intuitive way of creating a simulation, since it causes the computer to emulate the entities involved in the activity one is attempting to simulate. This is usually modelled through interacting processes and thus has extra processing for the context changes between the
Arrival $= A$          Departure $= D$

Figure 2.1: Virtual time-line for a Single Server Queue (Event Driven) (From [Fie])

interacting processes, of which there can be a lot in any given simulation (see Fig 2.2 for an example for a Single server Queue).

2.2 Simulation toolkit

The primary aim of the project is to be able to optimise the Java simulation classes described in [Fie]. These consist of classes for performing simulation of systems using either process or event driven models, with helper classes for performing measures and taking samples from various distributions. Although these classes are complete, they will need some modifications to make it easier to perform an automatic translation. There are other API’s that are also candidates for translation, but the problem is that these are usually either aimed completely at process-oriented simulation or on event-driven
computation, and thus the translation would have to be aimed at a completely different API, and would be far more complicated, whilst our target classes have distinct similarities between the types of simulation.

The main classes in the ic.doc.simulation.tools package are shown in Fig 2.3. Where the Sim and Event classes are used for event driven simulation and the PSim and SimProcess classes are used for Process-Oriented simulation. These are explained in more detail in [Fie], but briefly work as follows.

When using an event driven simulation one creates a subclass of Event which will perform the necessary computation for a specific part of the system, e.g. an arrival or departure (one class per event type). And one subclass of the Sim class which upon generation creates one Event object (e.g. an Arrival), schedules it (places it on the time-line) and then runs the execute block. The execute loop will continuously run the events until there are no more events on the time-line, new events are generated when other events are run.
Figure 2.3: The main classes in the Simulation Tools Classes

When creating subclasses to the Event class, one needs to implement the call() method, this method will perform all the necessary computation of the event and if necessary generate and schedule a new event. E.g. for an Arrival event one would write the following:

class Arrival extends Event {
    public Arrival( double t ) {
        super( t ) ;
    }
    public void call() {
        new Arrival( Math.random() ) ;
        n++ ;
        if ( n == 1 ) {
            wentBusy = now() ;
            new Departure( 0.25 ) ;
        }
    }
}

Here the call() first generates a new Arrival event, the call to super(t) will ensure that the event is inserted into the diary for future execution. The
call then increases the counter of number of items in the queue, and if there is only one item in the queue it causes the server to go busy and creates and schedules the completion of the item.

In the process-oriented approach, a new object is not created for every point in the time-line, but rather the process which hold the code for the event is activated each time the event is to occur. This is done using the activate() and passivate() calls.

Process classes must subclass the SimProcess class, for example, a customer in a single server queue can be modelled as follows:

class Customer extends SimProcess {
    public double arrTime ;
    private String name ;

    public Customer( double t ) {
        arrTime = t ;
    }

    public void runProcess() throws InterruptedException {
        q.enqueue( this ) ;
        if ( !worker.isActive() ) {
            worker.activate();
        }
        passivate() ;
    }
}

The constructor for the customer notes down the arrival time for reference purposes, the real computation is performed in the runProcess call (similar to the call() method in the Event type class). In this example the method places the customer on the queue and then checks to see whether the server is active, if it isn't it sets it active so that it can perform the necessary computation to let the customer depart. Finally the method makes a call to passivate() which ensures that the server gets a chance to serve the
2.3 Coroutines

A Coroutine is defined as “autonomous program which communicates with adjacent modules as if they were input or output subroutines” [Con63]. That is, a process which only communicates with other processes through blocking calls. Coroutines occur in many places, e.g. in the cooperation of a lexical analyser and a parser or the UNIX pipe operator [KM77]. Although [Con63] further notes that coroutines can be executed simultaneously on a truly parallel machine, this is not the definition I will use here, as the current idea of coroutines is that they cannot run concurrently.

From this definition we see that processes are coroutines if at any time only one process may be active. In the simulation toolkit, the process-oriented simulation ensures that only one processes can be active at any given time by the use of semaphores. This means that the processes in the simulation act as coroutines.

The benefit of knowing that the system uses coroutines is that it can be shown that any sequence of threads can be reduced to a single thread [GS96] as in Fig 2.4, thus removing the context-switching overhead. This single-threaded simulation is then equivalent to an event-driven sequence, and performs the exact same computations as the multi-threaded version.

2.4 Transformation

The main goal of the project is to create a transformation system which will transform one program into another. Similar projects have been attempted
Figure 2.4: Translating coroutines to a single process

before\(^1\) in other languages, and one can assume that it is possible in Java.

The transformation will fragment each thread into a number of parts, splitting them wherever an API call(hold or passivate) is made. Each of these fragments will become a continuation “block”, and can be used in a Continuation Passing System\([\text{Wan80}]\). Such a system will on each blocking call pass the next block with the call so that when the call is complete, the next block is run.

In its simplest form, translation will look like the example given in Fig 2.5. This represents a naive translation of a single process, this is naive because it will create some unnecessary blocks, such as “arrival\(_\text{end}\)” in the example, and some procedures could have been easily merged (such as “arrival” and “arrival\(_\text{continuation1}\)” in the example). This translation is quite intuitively performed, but it is difficult to prove that it is correct, although a similar proof has been completed for a dialect of LISP\([\text{Wan80}]\).

The program needs to be able to perform an automated translation from

\(^1\)Using compiler techniques for a C++ oriented simulation environment,\([\text{H108}]\)
Figure 2.5: A simple naive translation of a single process

one form to the other, this is similar to transforming source code in one programming language to source code in another programming language [GH94]. But some simplifications exist, as the source and target will be in the same programming language. The transformation is similar to that of an extended peephole optimisation [ASU85], in that it detects specific patterns in the input and translates them to another pattern in the output. The process used to translate source code in a more general manner is called Program Transformation, and its principles can be used when attempting to perform program transformation on the byte-code level.

**CCS/TCCS**

It is desirable to be able to formalise the transformation so it can be proven that the transformation outputs a program which performs the exact same computations as the input program. One such formalisation is the Calculus of Concurrent Systems. This is a process algebra described by R. Milner [Mil89], it is easy to use to describe communicating processes and their interaction, and thus works very well for a process-oriented model. As it is designed
for modelling communicating processes, and the event-driven simulation is
all held in a single, non-communicating process, the formalisation does not
manage to provide a good representation of the event driven program.

LOTOS

The Language Of Temporal Ordering Specification is a technique used to
formally describe models. It is based on CCS and CSP with some added
features. It is better suited for this kind of system as it has a good concept
of time.

Source Transformations

There are many examples of ways to perform a transformation on the source
code of a program ([Opd92], [TB95]). These all create a target version of the
input program which is either faster or more legible. This kind of transfor-
mation has been used in many areas of Software Engineering, from compiler
construction to document generation. They are however rather specialised
systems as a generalised system would require arbitrarily large amounts of
knowledge about the input program.

This however acts upon the source code, which can provide with complex
pattern matching problems. It would therefore be better to perform the
transformation on the abstract syntax tree.

Compiled code transformations

Rather than having to parse possibly invalid source code to be able to perform
a transformation one can use an Intermediate Representation. This has been
done in [RBJ97] for smalltalk and in [Eng97] for C code, and should also be
possible in Java. It would require a tree pattern matching system which can traverse the abstract syntax tree until it finds a match for a transformation that satisfies the precondition of the transformation. It is not explained in [RBJ97] how the refactorings are defined, and one may assume that they have been hard-coded into the program, this is not a very portable solution, but provides with higher speed and accuracy. In addition the refactoring browser described is interactive, requiring human input in some places, although this was a design choice rather than a necessity as it is desirable to have some non-automatic naming for code that is going to be re-used and modified.

picoThreads [BMS97] are an attempt at improving the performance of Java multi-threaded web architectures, such as “spiders”. This was done by transforming the process-oriented programs to event-driven programs, i.e. instead of blocking the process, the remainder of the computation is placed in its own method and scheduled for computation when the results of the call are completed. This was done by statically (at compile time), lazily discovering when a blocking call is made and splitting up the methods at these points.

The picoThreads attempt was moderately successful using java 1.1.5 on Solaris, but hampered by other performance problems with Java. Due to modifications and updates in later JVMs the performance gain of this approach was reduced. The approach is however similar to the approach attempted in this project.

Another similar project is Software Synthesis for Real-Time Information Processing Systems, described in [CTG94], where embedded systems are optimised based on static and run-time information about the timing of system calls within processes. For calls where enough information is known at compile time these are transformed into new “Thread Frames” and statically scheduled, whilst timing information which is unknown (often dealing with
user input) is dynamically (run-time) scheduled.

Of particular interest is Kwok Yeung’s *Automated Optimisation of Distributed Java Programs across Network Boundaries* [YK02]. This does not deal with optimising multi-threaded programs, but rather with network calls, but the approach to the problem is similar to the system above. The system works on Java byte-code using the SOOT framework, which makes it of particular interest to this project.

To use a method similar to the one described in [RBJ97] it may be necessary to generate an annotated parse tree for easier matching. This means that variables, methods and classes could be annotated with related methods and names.

## 2.5 Java

### Byte-Code

To aid in the correctness of the implementation and usage, it would be preferable that the tool modifies the Java byte-code rather than source code, as the byte-code has already been checked and compiled. This will also hopefully make it simpler to perform the translation as well, as there should be relatively few changes to be made to the syntax tree compared to the changes needed to the source code.

The class file format is defined in the Virtual Machine Specification [LY99], and is written as a stream of 8-bit bytes, i.e. all larger quantities (16, 32, 64 bit) are stored as multiples of 8-bit bytes. It is not advisable to start modifying the data directly, but one should access it through one of the available packages, such as SOOT.
The Byte-Code Engineering Library

The BCEL (available from http://jakarta.apache.org/bcel) is a tool for accessing and modifying class files directly with the byte-code. As shown in [Dah99] it allows users to operate with byte-code on a high level of abstraction without needing to handle all the details of the java class file format. The system only provides direct access to the byte-code in a stack based assembler notation. And through other projects such as the Class Construction Kit[Dah] one can access the class files through other levels of abstraction. There is however no Intermediate Representation which is amenable to whole-program optimisations.

SOOT

SOOT is a Java Optimisation Framework [VRLQ], it is a more established system than BCEL whilst giving a similar toolkit. It is more focused on a programming API than a GUI for end-user use. This allows easy integration with new programs and is therefore the best tool to use for the project.

There are three types of IR (Intermediate Representation) available through the use of SOOT, Baf, Jimple and Grimple. Baf is closest to byte-code and Grimple is closest to source code in their complexity level\footnote{BAF is a streamlined representation of the stack based byte-code, whilst Jimple and Grimple are stack-less}. Jimple is the most used interface as it provides a good balance between detailed low-level IR and higher level, more human readable representation.

SOOT presents two main methods of performing a transformation on the input code, \texttt{BodyTransformer} and \texttt{SceneTransformer}. A \texttt{BodyTransformer} is run on each method body as a visitor pattern [GHJV95]. This allows one to create a simple transformer without having to implement a way of picking out classes and their methods. The \texttt{BodyTransformer} is limited, however, in that
it becomes difficult to access the other methods and classes in the hierarchy. For this it is best to implement a SceneTransformer. The SceneTransformer is given a Scene object, which can be seen as the top-level element in the syntax tree. Through this element one can go through all of the classes associated with the given program (including library classes if the correct command-line arguments have been given), and perform transformations on methods or whole classes.

Figure 2.6 shows the main steps in a SOOT optimisation, where the Optimise step is where the main computation is performed. The “Simplify” step is the operation of generating the Jimple representation from the stack-based byte-code. The optimisation is performed through a pluggable interface, i.e. one can register a BodyTransformer object with the Scene and the transformer will be called automatically when needed. Thus several transformers may be registered and the system will call each of them on the relevant object in a predefined order. The system then generates BAF code from the Jimple code and performs some basic optimisations on this before calling the Jasmin compiler to generate the output byte-code.

In the Jimple representation each class is represented in a SootClass object, this holds information such as name, implemented interfaces and extended superclasses. In addition it holds all of the methods and fields (class variables) held in the class. Each field is represented in a SootField object which can be retrieved by name, whilst SootMethod objects are held in Chains which are Collections and can be looped over using an iterator.

The Java Virtual Machine

The Java virtual machine is the program which validates and executes the byte-code. This is now usually performed by Just-In-Time compiling the byte-code into an optimised version before the execution.
2.5. JAVA

Java Thread Optimisation

Figure 2.6: The main steps in a SOOT optimisation

Class file

Jimplify

Jimple code

Optimise

Optimised Jimple Code

Generate Stack Code

Baf Code

Generate Assembler code

Jasmin file

Jasmin
Threading

A thread is a lightweight process, and a Java Thread is a thread which has been created via the Runnable interface. A kernel thread or a process is a process which has its context known to the kernel. There are various forms of threading systems available to Java, in the early days of Java “green threads” were utilised. Green threads are threads created inside the VM and are as such not visible to the operating system. This was faster due to the fact that threads were not available in the Linux OS (they are in Solaris), and threads would otherwise need to be represented as full-blown processes. Due to further advances in the VMs, the benefits the green threading model had earlier have been surpassed and green threads are thus not considered an effective implementation any more\cite{Mic}. This is due to that the green threads cannot take advantage of multiple processors (since all the threads are kept within a single process), and since all system calls must be intercepted to prevent the whole JVM from blocking on the call.

One problem of threading under Linux is that it does not support lightweight threads\footnote{Linux only supports one-to-one threading, i.e. that each thread is mapped to a single kernel thread, as opposed to many-to-many, where many threads are run within the context of a single kernel thread (but to the kernels knowledge)}, and as such each thread must be instantiated as a full-blown process. This involves a large amount of overhead in the running of a threaded java program. Such programs can often spend 20\%-50\% of their CPU time performing kernel tasks (context switching, process creation and process deletion)\cite{BH00}.
2.6 Flyweight Pattern

Introduced in [GHJV93], flyweights are a method for object re-use and sharing. The usage of creating fragments of methods as flyweights is that it would reduce the amount of objects created\(^4\) and thus make the end result a lot faster. The only restriction on this is that the shared objects cannot have context-dependant state. This makes it difficult to devise the blocks as flyweight objects as there is often a great deal of context dependant state information. The way to avoid this problem is to pass the state to the object as an argument (an extrinsic state [Mag]), but the overheads of marshalling/demarshalling the state object could be higher than the benefit gained from using the flyweight pattern.

\(^4\)The objects would be declared static
Chapter 3

Implementation

After optimising several process-oriented simulations by hand it was determined that it is possible to perform the optimisation in an automated manner. The program to perform the automation must be able to produce a copy of the original program where all parts which are process simulations are converted and all other parts are left alone. There are three stages at which this conversion can take place:

1. Source code
2. Intermediary Representation
3. Byte-code

To be able to perform the translation on the source code level it would be necessary for the program to be able to understand and parse all valid forms of Java syntax. This is a lengthy task and hampered by the fact that one cannot be sure that the input is correct, i.e. that it compiles and runs.

Performing the translation on byte-code would ensure that the input is correct (probably runnable) and is presented in a logical manner. However, it is difficult to perform modifications / transformations on the code since
all references to the constant pool would have to be maintained in a correct form.

One can form an Intermediary Representation (IR) of the program by de-compiling the byte-code into a syntax tree. This IR can then be transformed and later re-compiled back into byte-code. This gives the power of a easily accessible logical structure and the assurance that the input is valid and working.

Of the two systems available which can retrieve an IR for a given program, BCEJ and SOOT, SOOT was chosen as it is more widely used and is somewhat more versatile. Since both of these programs are implemented in Java, and since the program is to work on Java code, it makes sense to create the program in Java, even though a C/C++ program might be faster.

A new package, called ic.doc.simulation.optimiser, was created alongside ic.doc.simulation.tools. This package performs the optimisation and contains the target classes of the optimisation.

3.1 Changes to Toolkit

To make it easier to translate process simulations to event simulations, some work was put into making the toolkit more structured and object oriented than its original form. This includes the following modifications:

- The old Event (Sim) and Process (PSim) classes have been modified so that they have a single superclass (Sim) holding the time object (Sim was renamed to ESim)
- Event and SimProcess now have a common superclass SimObject.
- The Diary in the event driven model and ProcList in the process driven model were combined to form a unified SimList which can con-
tain both Events and SimProcesses.
- The Time class was removed and the time added as a static variable in the Sim object.

The structure of the class hierarchy after the modifications is given in Fig 3.1, excluding Samplers and Measures, as these have not changed noticeably.

3.2 Target Event Classes

Initially an attempt was made to translate the process classes into Event classes. It was soon discovered, however, that there is not enough information in these classes to be able to run a converted process simulation. The Event objects are only scheduled once, at creation time. This means that the system would have to create a new object for every continuation each time it is called. For even a small number of fragments this would result in a large number of Events needing to be created. It was our desire to create the continuation once, and reschedule it every time it is to be run. A series of simulation classes were therefore created: OptControl which replaces the SimProcess classes and hold the API calls (hold, passivate and activate), OptEvent which holds the contents of each of the blocks and which can be placed in the diary (i.e. it is a subclass of the SimObject class), and OptSim, a subclass of Sim, which performs the scheduling. These classes are all part of the optimisation package and thus the optimisation package is needed to execute the optimised programs as well as performing the translation. This approach is favoured over placing the optimisation code in a separate package from the simulation classes to reduce the complexity of the end product.
3.3 Block Structure

The SimProcess classes each hold a single runProcess call which must be fragmented into blocks based on API calls. A typical fragmentation (without
control flow commands) is shown in Fig 3.2.

```
class Arrivals extends SimProcess {
    public void runProcess()
        while ( true ) {
            hold(Math.Random());
            Customer cust = new Customer();
            cust.activate() ;
        }
}
```

Figure 3.2: Initial fragmentation of Arrival Process

The process has been carved into four blocks, which are continuations for the execution of the method. Two of these are empty, due to the fact that the control flow of the loop boundaries needs to be held separately for the time being and then added later. Thus Block1 represents the start of the method and the start of the loop, Block4 represents the end of the loop and the end of the method and Block2 holds the API call whilst Block3 holds the continuation after the API call has executed. Each of these blocks need to be placed in the simulation diary for later execution if necessary. The original idea was to create a new method in the SimProcess class for each of the blocks. This cannot be implemented however, as one cannot dynamically access a specific method in a class. As an example consider the following code:

```
class Arrivals{
    nextBlock=block1;

    private block1(){
        nextBlock=block2;
    }
    private block2(){
```
3.3. BLOCK STRUCTURE

nextBlock=block3;
    hold(Math.Random());
}
private block3(){
    Customer cust = new Customer();
    cust.activate();
    nextBlock=block4;
}
private block4(){
    nextBlock=block1;
}

class Arrivals extends OptControl{
    OptEvent block1 = new Block1();
    OptEvent block2 = new Block2();
    OptEvent block3 = new Block3();
    OptEvent block4 = new Block4();
    nextBlock=block1;

    private class Block1() extends OptEvent{
        void call(){
            nextBlock=block2;
        }
    }
    private class Block2() extends OptEvent{
        void call(){
            nextBlock=block3;
        }
    }
    private class Block3() extends OptEvent{
        void call(){
            nextBlock=block4;
        }
    }
    private class Block4() extends OptEvent{
        void call(){
            nextBlock=block1;
        }
    }
}

In theory this could be run, as the program assigns a method to the variable nextBlock and the execute loop runs the method currently assigned to the variable. In Java however, only objects can be assigned to a variable. The solution is to place each block method into its own class, and then assign an object of the block class to the variable:
3.4. TRANSLATION PROCESS

```java
nextBlock=block3;
hold(Math.Random());
}
}
private class Block3() extends OptEvent{
    void call(){
        Customer cust = new Customer();
        cust.activate();
        nextBlock=block4;
    }
}
private class Block4() extends OptEvent{
    void call(){
        nextBlock=block1;
    }
}
public execute(){
    while(true){
        nextBlock.call();
    }
}
}
```

For this to work each of the blocks must implement the `call()` method, this is what the `OptEvent` class defines, so that the execute loop can be assured that each block implements the call. Each `Block` will be created once per process, it is possible that a further optimisation could be to make them entirely static, so that they can be accessed by all instances of the process (e.g. if there were three arrival processes, there would still be four blocks instead of twelve). This however requires passing the context information to the method at run time.

### 3.4 Translation process

The main stages of the initial optimiser design are given in Fig 3.3.
This was implemented using a BodyTransformer which is called on each method. However, since the system needs to create new inner classes, it needs to be able to access the class which holds the method it has been passed. This should not be a problem, but when the program tries to modify the class to add the inner classes, Java raises a ConcurrentModificationException. This is due to that the iterator which passes the methods to the BodyTransformer also accesses the class and Java will not allow a modification of the class structure whilst the iterator is still in use.

A SceneOptimiser class was therefore created to loop through all of the classes available in the Scene and create a BlockOptimiser object to go through the methods of the classes to modify them. Most of the computation therefore falls on this class in performing both simple transformations (e.g., translating superclass ic.doc.simulation.tools.SimProcess to ic.doc.simulation.optimiser.OptControl). The structure of the trans-
formation process is given in Fig 3.4.

![Diagram](image)

Figure 3.4: Overview of the Optimisation Process using a SceneTransformer

Once a class has been found that needs translating, a `State` object is created for it. The `State` object contains references to the method, the body of the method and a `HashMap` of all the blocks it contains (this map will be populated by the fragmentation process).
3.4. TRANSLATION PROCESS

Fragmenting Methods

The next step of the translation is to read in the body of the method and split it into blocks, based on the API calls made in the method (see Fig 3.2). For each block a BlockState object is created, this object holds references to the class which will hold the block and various other references. A Chain object is also created holding all of the statements in the block (these statements will need later translation, so they are not added directly to the body of the block). For the example of the Arrival process, these fragments are the statements in Block2 and Block3 in Fig 3.2. For each of these blocks a new class is created as an inner class of the original process. Thus the Arrivals class example would have the following structure:

class Arrivals extends OptControl{
  OptEvent block1 = new Block1();
  OptEvent block2 = new Block2();
  OptEvent block3 = new Block3();
  OptEvent block4 = new Block4();
  OptEvent nextBlock=block1;

  private class Block1() extends OptEvent{
    void call(){
    }
  }
  private class Block2() extends OptEvent{
    void call(){
    }
  }
  private class Block3() extends OptEvent{
    void call(){
    }
  }
  private class Block4() extends OptEvent{
    void call(){
    }
  }
}

29
3.4. TRANSLATION PROCESS

When the entire body has been split into blocks, the control flow between the blocks (which block follows which block) is stored for future reference. This information is stored in an overall State object held for the process.

Method Calls

When breaking up a method, one problem is handling calls to other methods which make an API call. This occurs in some instances of more complicated models, such as the Queueing Network, where the jobs call submit on a queue which will passivate the current job.

When this occurs the calling method needs to have a block boundary after the call, and the continuation stored so that it is called once the method has been completed. The called method then needs to be fragmented as usual. When the called method performs an API call it will need to store its continuation, but we have already stored the continuation of the first method, and thus, to avoid confusion we store the continuations on a stack. There are two ways to do this, explicitly or implicitly. The explicit way is to push the continuation on to the stack before making the method call and the implicit way is to utilise the return stack, i.e. replacing the return call with a call to the next continuation.

If only one method calls the second method the implicit way can be computed (as the call to the continuation must be defined at compile time). But if several methods can make a call to the method, as in Fig 3.5, the call to the continuation must be dynamic, as the continuation will be different depending on which method it was called from. Thus the continuation would have to be passed to the method at run-time. It is therefore simpler to use an explicit stack for the continuations.

When the method is called at run-time, the continuation of the caller will be placed on the explicit stack and the callee started. When the callee...
makes a call to `hold` or `passivate`, it will include its continuation as the argument to `hold` or `passivate`. Thus the scheduler will first run all of the continuations of the callee, and the pick the top element of the stack (the continuation of the caller) afterwards.

Detecting when such a method call is made is not very hard: The system checks each invoke statement in the method and when it finds a method which is not in a library class\(^1\) it opens up the body and looks to see if any of the statements are API calls. This will work recursively so that if a method calls another method which makes an API call it will be detected.

The implementation of this is somewhat complex, and has not been fully implemented. The system handles calls which have an API call as the last statement (for example the Queueing Network and Repair model), but may generate incorrect code if the called method has further computation after the API call.

Another possible way of reduce the complexity involved with API calls would be to in-line all or parts of the code of methods which make an API call so that only the calling method would need to be fragmented. This is however hard to implement, as the called method can reference variables within its own object, these references would have to be translated so they would still reference variables in the callee from the calling method.

\(^1\)Library (sun.*, com.ibm.* etc) classes are assumed not to make an API call
One limitation to the algorithm that the called method must not make a call, directly or indirectly, to the calling method, as loops would occur. This could be overcome by keeping a list of all the methods in the call hierarchy, and when a method which is already in the call graph is reached a loop has occurred. Since the algorithm will automatically return true if an API call has been detected, the detection of a loop implies that no API call is made in the called methods and thus they do not need to be fragmented.

Fixing statements

Java Object Referencing

The main problem which now arises is that statements in different blocks might access the same variable, i.e. the variable needs to be in scope for all statements which access it, even if they now are in different classes. There are two ways to overcome this. In [YK02] each variable was replaced with an entry in an array, and the array passed between the blocks so that the statements in the blocks can access the variable. Another method is to keep the variable as a field in the encapsulating class and reference this instead of a local variable.

This second method was favoured in the implementation as it avoids having to create an arbitrarily large array and pass it between the blocks. The changes to the statements is similar. In the first method, each statement has to be changed so that it uses an array reference in stead of a variable access, whilst in the second the statements need to be changed to perform a field reference.

The problem with this method is that it does not work in practice, as it is not allowed to reference a field in an outer class directly. The way to fix this is to make a local copy of the value stored in the field and then access
this (and save it back afterwards), this seems to be the general way the java compiler does the job. A further complication comes from the scope of fields in classes. In the source code a field in a class is accessible directly from the inner class. For example:

```java
class Outer{
    int var = 0;
    class Inner{
        void change(){
            var++;
            var++;
        }
    }
}
```

This is valid source code since var is in scope in the Inner class. However, this is not true for the byte-code when:

- A reference to the Outer class is held in a field (this$0) in the inner class
- A copy of this reference is read into a local variable
- A copy of the outer class field is created in a local variable, which is then accessed by the statement
- This copy is written back to the field afterwards

Thus the byte-code of Inner would look like this when translated back to source:

```java
class Outer$Inner{
    Outer this$0;
    public Inner(Outer ref){
        this$0=ref;
    }
    void change(){
        Outer r = this$0;
    }
}
```
3.4. TRANSLATION PROCESS

```java
int i0 = r.var;
i0++;
r.var=i0;
Outer r2 = this$0;
int i1=r2.var;
i1++;
r2.var=i1;
}
}
```

Furthermore, the compiler creates a new local variable for each statement which accesses the field in the outer class, even if it has already created a reference to the outer field it will create a new one, this is why there are two references to the `Outer` class and two copies of `var` in the example above.

The first implementation read in and wrote back all of the variables from the original method. This is obviously inefficient, as it would be reading and writing to variables which are not accessed in the block. However, SOOT provides a way of retrieving a list of variables defined or used in a set of statements. Using this information, the blocks will only read in variables used and write back variables defined in the block.

In all statements in all classes, all variables and classes of type `SimProcess` need to be changed to `OptControl` and `PSim` has to be changed to `OptSim`. This is to ensure that variables are of the correct type for the runtime validator.

At the end of this part the program will have a structure similar to that in Fig 3.6 (For the SSQ example).

**Scheduling**

Once the block classes have been properly created the control flow needs to be generated. In the simple cases where no API call has been made (e.g.
3.4. TRANSLATION PROCESS  
Java Thread Optimisation

Figure 3.6: Class diagram of translated Single Server Queue
start and end of a loop) a call statement can be inserted which calls the next block. In the cases where an API call is made the call is regenerated by making a call to the `OptControl` `hold` or `passivate` method with the next block as the argument to the call. This is equivalent to scheduling the continuation event in the event-driven simulation. To ensure that the new program will start, the constructor of the optimised program adds the first block to its block stack so that the first `activate` call will start the first block.

### 3.5 Auxiliary classes

To simplify debugging and output, an extra `Output` class was created. This has methods for outputting, debug outputting and exception handling. The print method sends the given string to `System.out.println` but has the added benefit that code can be inserted which will be run every time something is printed. For example a “verbose” or “logged” argument could be added to the command line, to enable output or redirection of the print statements. The debug method calls `print` with the string, but only if `DEBUGGING` is set to true, thus avoiding a need to go through the entire code to comment out any debug statements when necessary. The exception method is called with an exception and will print a stack trace if debugging is enabled and short message if not and then call `System.exit.`
Chapter 4

Testing

In this chapter we attempt a qualitative analysis of the program, to determine if the output of the system performs the same computation as the input. It would be desirable to have a formal proof of the system, this is however a time consuming task set aside for future work.

4.1 Test Cases

A range of test cases are proposed, designed to exercise all aspects of the simulation classes.

For each test case there is a process-oriented version and an event-driven version. The following JVMs are used to run the tests:

1. IBM Java 1.3
2. IBM Java 1.3 with green threads
3. SUN Java 1.4
4. SUN Java 1.3
5. Blackdown Java 1.3
4.1. TEST CASES

All of the tests are run on a single processor 1.8GHz AMD Athlon equipped machine, with 512 Mb of RAM and running SuSe Linux. With this amount of memory, there shouldn’t be a risk of having to swap out parts of the program whilst executing, which would cause speed tests to take longer.

**Basic Test**

To ensure that the basic functions are working, an initial class using a single looping process is used:

```java
Wibble{
    while (true)
    {
        hold(1);
    }
}
```

If this runs after being translated, it means that the basic functionality of the fragmentation and scheduling is working. It should also show how much the overhead of context switching is costing the Process driven model.

**Single Server Queue**

The Single Server Queue is a simple system with an Arrival, Worker and Customer process. The processes work as in the following pseudocode

```java
Arrivals{
    while (true){
        hold(random);
        cust = new Customer
        cust.activate();
    }
}
Customer{
```
4.1. TEST CASES

```java
enqueue;
if(!worker.isActive())
    worker.activate;
passivate;
}

Worker{
    while(true){
        while(queueLength>0){
            hold(0.25);
            c=dequeue;
            c.activate
        }
        passivate
    }
}
```

This system will spawn a large number of processes (one for the Arrivals, one for the Worker and one for each customer). The correctness of this test will show that the system transforms the input processes without mixing the calls (a possible failure scenario would be that the first customer block calls the second block of the arrival process instead of the second block of the customer process).

**Simple Queueing Network**

The queueing network model consists of three service centres, each holding a queue and a server. The arrival process spawns new Customer processes which submit themselves to a service centre according to the Network model.

The main complication which this program adds is that it makes a call to a method which causes the process to be held. This means that the optimisation program needs to be able to analyse all the method calls a method makes to ensure that all situations where it can be held or passivated
are caught.

The queueing network has processes with the following pseudocode:

```
Arrivals{
    while(true){
        hold;
        new Customer;
    }
}

Customer{
    while(current != final){
        centre[current].submit(this);
        current = network.transit(current);
    }
}
```

where the centres are service centres defined as follows:

```
Server{
    while(true){
        while(queuelength > 0){
            hold;
            Customer = dequeue;
            Customer.activate();
        }
        passivate();
    }
}

Submit(Customer){
    if(!Server.isActive){
        Server.activate();
    }
    Customer.passivate();
}
```

The “Network” determines which is the next service centre for the customer based on a set of probabilities, and “final” is the last service centre
in the network. If this test completes successfully, the system manages to transform method calls to API calling methods (i.e. the submit method above).

**Repair Model**

The repair model simulates a situation with a certain number of working and spare machines in action, whenever a machine stops working, a spare is used instead (if there are any) and a repair process fixes the broken machine (if there is a repair process free).

```java
Machine{
    machines.submit(this);
    while (true){
        repairers.submit(this);
        machines.submit(this);
    }
}
```

Where the machines and repairers are CentreNs defined thus:

```java
Server{
    available.enqueue
    passivate();
    while (true){
        while (queueLength > 0){
            Customer = dequeue;
            hold;
            Customer.activate();
        }
        available.enqueue;
        passivate();
    }
}
```

```java
Submit(Customer){
    enqueue(Customer);
}
```
if(!available.isEmpty){
    Server.activate();
}
Customer.passivate();

Similarly with the Queueing network, this model uses service centres and has method calls which perform a passivate. The service centre holds a queue of available servers and calls an available one to process the next job. The main purpose of this test is quantitative in that there is a fixed number of processes in the system, so the main load of the program is on context switching in stead of process creation / death.

CPU

This model concerns a system with a single CPU and two disk drives with a single arrival point. The purpose of the model is to measure the capacity of such a system and when the queues start growing out of control.

Arrivals{
    while(true){
        hold;
        new Job;
    }
}

Job{
    cpu.submit(this);
    while (random){
        disk[random].submit(this);
        cpu.submit(this);
    }
}

The arrival process is the same as previously used, but the Jobs will leave the system based on a given probability (in the test a value if $\frac{1}{12!}$ is used).
The disk and CPU servers work as in the “Network” example above.

The purpose of the model in regards to the optimisation program is to ensure that when there are multiple customers and service centres available to the system, that the optimisation program manages to translate all of the references correctly. In addition this system can involve a large proportion of context switching compared to process birth/death. This causes there to be a large number of processes running in the operating system, which is often a massive bottleneck which the optimisation should remove.

4.2 Correctness

Performance benefits are useless unless it can be shown that the program performs the same computation as the original program. There are two ways to ensure this: Either use a formal proof on the transformations used in the program to prove that for all programs the system will transform them into a new program which performs the same computation, and that all programs using a process-oriented simulation will be transformed into an event-riven simulation. This however is a very time consuming task, and beyond the scope of this project. Instead, by showing that each input program is transformed into a program which has the same state changes as the original, the system is shown to perform the correct transformation. Since the concept of state changes is individual to each program, this cannot be shown in a general case, but having shown that it works for a range of complex test cases we can assume that it works in the general case.

In most of the programs there is some use of random numbers, to be able to check whether different versions of the program have the same state changes the random number sequence must be predictable. This is done in Java by seeding the random number generator with the same number. Each
4.2. CORRECTNESS

state change is marked by a print statement with the time of the change and what the change is.

In each of the test cases, the output program gave an identical trace as the input program, and the event driven model exhibits the same changes.
Chapter 5

Evaluation

In this section is an evaluation of the performance of the system. The range of test cases proposed in Chapter 4 are run for the event-driven simulation, process-oriented simulation and optimised simulation. They are run multiple times to ensure that that the results are correct and run on multiple JVMs to ensure that the results are not exclusive to one specific JVM. The green threads JVM is interesting due to its different implementation (see 2.5).

5.1 Overheads

Before an evaluation of the program is given, an investigation into the overheads of process creation, process completion and context switching is needed. To do this a simple java program has been created. Initially run with the following code:

```
Loop{
    i=0;
    while(i<800){
        i++;
    }
}
```
5.1. OVERHEADS

Java Thread Optimisation

```java
}

This shows the time taken for the creation of the program and running a simple loop. Then a “new Thread()” statement is inserted into the loop:

```java
Loop{
    i=0;
    while(i<800){
        new Thread();
        i++;
    }
}
```

Thus measuring the amount of time it takes to create a new empty thread (but not starting it). Further tests include creating empty Events:

```java
Loop{
    i=0;
    while(i<800){
        new Simple();
        i++;
    }
}

Simple extends Event{
}
```

And finally, empty Processes:

```java
Loop{
    i=0;
    while(i<800){
        new Simple();
        i++;
    }
}

Simple extends SimProcess
}```
5.2 TEST-CASES

Since the structures in these objects is similar, it should show the difference in overhead in creating a simple object and performing a context switch.

Each test loops 800 times (it was initially attempted with 100000, but the system runs out of available threads at 1030) and has been completed several times to ensure that a correct estimate is achieved.

<table>
<thead>
<tr>
<th>Test</th>
<th>User time</th>
<th>Kernel time</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empty loop</td>
<td>0.23</td>
<td>0.07</td>
<td>0.30</td>
</tr>
<tr>
<td>Basic thread creation</td>
<td>0.26</td>
<td>0.07</td>
<td>0.33</td>
</tr>
<tr>
<td>Event creation</td>
<td>0.32</td>
<td>0.08</td>
<td>0.40</td>
</tr>
<tr>
<td>Process creation</td>
<td>0.38</td>
<td>0.53</td>
<td>0.92</td>
</tr>
</tbody>
</table>

As can be seen here, the overhead of creating a thread is minimal, the creation of a more complex object adds some time to the process, but the running time is almost doubled when a process is created and a context switch occurs. This means that there should be a large potential for speedup if one can remove the overhead of a context switch.

Some of the further tests given here have been run on different systems, so may not be comparable with each other. All runs of the same tests have been run on the same machine under the same load to ensure that each test gives correct results.

5.2 Test-cases

Each of the test-cases defined in the previous chapter will be evaluated quantitatively, giving the Speedup $\frac{\text{Process\ Optimised}}{\text{Event\ Optimised}}$ and Slowdown $\frac{\text{Event\ Optimised}}{\text{Process\ Optimised}}$ where available.
Basic Test

Since there is only one process which has been created, there should only be two processes in the system, the only context switch which can occur is between these two. The following table and diagram shown in Fig 5.1 shows that for a run of 1000000 increments, in all cases a purely event-driven model is far faster than the process-oriented version, and the optimised version is a little slower than the pure event driven.

<table>
<thead>
<tr>
<th>Java</th>
<th>Event</th>
<th>Process</th>
<th>Optimised</th>
<th>Speedup</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 1.3</td>
<td>3.56</td>
<td>86.82</td>
<td>4.78</td>
<td>18.2</td>
<td>0.74</td>
</tr>
<tr>
<td>Green</td>
<td>7.79</td>
<td>104.59</td>
<td>7.74</td>
<td>13.5</td>
<td>1.0</td>
</tr>
<tr>
<td>Sun 1.4</td>
<td>3.57</td>
<td>109.66</td>
<td>4.54</td>
<td>24.2</td>
<td>0.79</td>
</tr>
<tr>
<td>Sun 1.3</td>
<td>3.02</td>
<td>103.67</td>
<td>4.12</td>
<td>25.2</td>
<td>0.73</td>
</tr>
<tr>
<td>Blackdown</td>
<td>3.27</td>
<td>112.65</td>
<td>4.08</td>
<td>27.6</td>
<td>0.80</td>
</tr>
</tbody>
</table>

![Chart showing performance comparison](image)

**Figure 5.1: Basic Test Case**
5.2. TEST-CASES

Java Thread Optimisation

The speedup is much larger than the overhead tests should imply. The optimised version is also fairly close in performance to the event-driven version which shows that the system can produce good programs for the simple case.

Single Server Queue

In the single server queue there is an added overhead in the birth/death of the individual customer processes, as can be seen from the table and Fig 5.2 this has caused an even greater difference between the event driven and process driven versions. The test were run with an average inter-arrival time of 0.5 with a random distribution, the service times were constant at 0.25. This ran for 2000000 time units, thus there were approximately 4000000 processes created in this time period.

<table>
<thead>
<tr>
<th>Java</th>
<th>Event</th>
<th>Process</th>
<th>Optimised</th>
<th>Speedup</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 1.3</td>
<td>17.61</td>
<td>2515.04</td>
<td>94.89</td>
<td>26.5</td>
<td>0.19</td>
</tr>
<tr>
<td>Green</td>
<td>34.65</td>
<td>9059.16</td>
<td>171.02</td>
<td>53.0</td>
<td>0.20</td>
</tr>
<tr>
<td>Sun 1.4</td>
<td>13.92</td>
<td>3544.75</td>
<td>94.79</td>
<td>37.4</td>
<td>0.15</td>
</tr>
<tr>
<td>Sun 1.3</td>
<td>11.69</td>
<td>3182.45</td>
<td>94.23</td>
<td>33.8</td>
<td>0.12</td>
</tr>
<tr>
<td>Blackdown</td>
<td>11.54</td>
<td>3275.85</td>
<td>96.99</td>
<td>33.8</td>
<td>0.12</td>
</tr>
</tbody>
</table>

There is an enormous speedup occurring on the green threads JVM (and the test was run several more times to ensure the accuracy of the results). As the main component of the optimisation is the removal of a context switching overhead, this would lead to the conclusion that the green threads implementation is spectacularly poor in performing context switches.

The comparison between the Event-driven simulation and the optimised version is rather worrying, by being almost eight times slower on the last two JVMs the optimised version is rather lacking, the cause of this can only be explained in that the Process-oriented (and thus the optimised) version

49
has an Arrival, Worker and Customer process, the Event-driven model only models an Arrival and Worker, leaving the customer as a simple integer.

**Simple Queueing Network**

In the queueing network each customer will perform multiple submissions / context switches. This means that there will be more pressure on context switches than on process birth / death. One could therefore expect the green threads JVM to perform well on this test. The results are given in the following table and in Fig 5.3 for a run of 100000 time units with an average inter-arrival time of 0.5 with an exponential distribution. There is no event-driven model for this simulation available, so the test is only on the speedup of the process-oriented version.
5.2. TEST-CASES

<table>
<thead>
<tr>
<th>Java</th>
<th>Process</th>
<th>Optimised</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 1.3</td>
<td>29.35s</td>
<td>2.66s</td>
<td>11.0</td>
</tr>
<tr>
<td>Green</td>
<td>35.66s</td>
<td>2.32s</td>
<td>N/A</td>
</tr>
<tr>
<td>Sun 1.4</td>
<td>34.28s</td>
<td>1.98s</td>
<td>15.4</td>
</tr>
<tr>
<td>Sun 1.3</td>
<td>34.55s</td>
<td>1.94s</td>
<td>17.3</td>
</tr>
<tr>
<td>Blackdown</td>
<td></td>
<td></td>
<td>17.8</td>
</tr>
</tbody>
</table>

The system is unable to compute values for the process driven version for the green threads java as it ends with SIGABRT (Abnormal Termination) whenever it is run with any large time limit, the system will abort once the 1124th Customer starts executing.

The SIGABRT signal is described by IBM[Whi02] as occurring whenever the JVM detects a fault in the JVM, thus it is probably a bug in the JVM. It seems that it may be related to the amount of concurrent threads. The green threads JVM will manage when fed with a program that only has a few threads running at any given time, but when the amount of processes is increased it fails.

In the other JVMs the program works fine, and the speedup is between 11.0 and 17.8. It seems that when the call structure becomes more complex, the speedup is reduced, the servers are split up into many blocks and this causes a large amount of jumping between methods in the different blocks. If there were fewer blocks the results should be better.

**Repair Model**

The repair model does not have any birth/death overhead after the machines have been created, so the performance issues are only related to context switches. With a mean upTime and repairTime of 10.0 and 10 repairers and 10 machines (30 spare) and a time limit of 200000 units, the test is run for the process and optimised versions.

51
5.2. TEST-CASES

Java Thread Optimisation

Figure 5.3: Queueing Network

<table>
<thead>
<tr>
<th>Java</th>
<th>Process</th>
<th>Optimised</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 1.3</td>
<td>98.80s</td>
<td>10.83s</td>
<td>9.1</td>
</tr>
<tr>
<td>Green</td>
<td></td>
<td>16.73s</td>
<td>N/A</td>
</tr>
<tr>
<td>Sun 1.4</td>
<td>114.70s</td>
<td>11.93s</td>
<td>9.6</td>
</tr>
<tr>
<td>Sun 1.3</td>
<td>109.93s</td>
<td>10.94s</td>
<td>10.0</td>
</tr>
<tr>
<td>Blackdown</td>
<td>117.10</td>
<td>11.43s</td>
<td>10.2</td>
</tr>
</tbody>
</table>

Again the green threads JVM aborts on the process driven version. The speedup here is rather small, this may be due to the process creation overhead which is present in other tests but not in this test, as the processes are defined in the beginning and used throughout the simulation.

CPU

The CPU model has some birth/death of processes, but puts most pressure on context switches between a vast amount of processes. The results for a run
of 25000 time units with a mean inter-arrival time of 0.5, are in the following table and in Fig 5.5 are similar to the above test, quite possibly due to the rather similar structure in the simulation.

<table>
<thead>
<tr>
<th>Java</th>
<th>Event</th>
<th>Process</th>
<th>Optimised</th>
<th>Speedup</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 1.3</td>
<td>36.94</td>
<td>806.64</td>
<td>78.07</td>
<td>10.3</td>
<td>0.47</td>
</tr>
<tr>
<td>Green</td>
<td>51.89</td>
<td>0.00</td>
<td>104.22</td>
<td>N/A</td>
<td>0.50</td>
</tr>
<tr>
<td>Sun 1.4</td>
<td>28.72</td>
<td>918.51</td>
<td>75.55</td>
<td>12.2</td>
<td>0.39</td>
</tr>
<tr>
<td>Sun 1.3</td>
<td>24.99</td>
<td>887.17</td>
<td>74.58</td>
<td>11.9</td>
<td>0.34</td>
</tr>
<tr>
<td>Blackdown</td>
<td>26.45</td>
<td>938.66</td>
<td>78.11</td>
<td>12.0</td>
<td>0.34</td>
</tr>
</tbody>
</table>

Although the optimised version is more than 10 times faster than the process driven version, it is half as fast as the fully event-driven version, and on some JVMs only a third of the speed. This is most probably due to that there is an overhead in creating the blocks for the different processes, and the program has created 9 blocks for the service centre, 6 blocks for the jobs and
4 blocks for the arrival process. This number could be reduced by a more intelligent fragmentation and the resulting program could therefore be even faster.

![Bar chart showing CPU model](Image)

Figure 5.5: CPU model

### 5.3 Overall

The system manages to speed up process-oriented programs at a rate between 10 and 30 times (the 53 times speedup of the SSQ under green threads can probably be ignored, being due to the quality of the green threads implementation rather than the optimisation). This is significantly more than what the loop test conducted at the beginning of this section would imply (a doubling of the process time was seen). This is most probably caused by the context switch having to move more data. We also see that in the case where there are only two context switches per process (the single server queue)
there is more to gain than where there are multiple context switches (for example the queueing network), this would imply that the creation or completion of the SimProcess is more expensive than previously assumed, and that the transformation into events leads to removal of this overhead. Some more testing would have to be conducted to be sure of this result though.

The system still has some way before it can match the performance of the purely event-driven model, some of the gap can probably be closed by reducing the number of blocks created by the fragmentation process.
Chapter 6

Conclusion

This project set out to improve the performance of process-oriented simulation models. We have successfully achieved a speedup of at least an order of magnitude so that they are comparable in performance to hand-coded event-driven models. This was accomplished by transforming the processes into a continuation-passing system which runs between 10 and 30 times faster than the original.

Furthermore, we have found that:

- Green threads have become obsolete due to improved performance in other parts of the JVMs.
- The greatest speedup is achieved in simulations where there are few processes in the system at any given time.
- Translating mutually exclusive threads into a Continuation Passing System gives a large performance improvement.
- Optimisations are easier to perform on byte-code as one has access to an syntax tree representation of the code.
- The SOOT framework gives a very good API to create optimisation systems for byte-code.
6.1 Further Work

There are several extensions to this project which can be implemented:

- The read and write-back operations on the variables in the block classes can be optimised through a live-variable analysis to ensure that only live variables are read and written back.
- An interface can be created so that the translation is not hard-coded for a specific toolkit, but defined in a “schema” file. The system could then be used to transform other simulation toolkits.
- The system could create static continuation blocks, instead of instantiating one of each block for each object, as this should increase (possibly double) the performance.
- Detection of recursive methods to prevent the system from going into an infinite loop.
- Called methods which make an API call need to be fragmented to ensure that the execution order is correct.
6.1. FURTHER WORK

- Empty blocks can be optimised away, leading to faster execution due to less object creation overheads.
Appendix A

Using the system

A.1 Installation

In addition to the optimiser classes one needs the SOOT package. Installation proceeds as follows:

1. Download the soot package from
   http://www.sable.mcgill.ca/software/sootclasses-1.2.3.jar
2. Download jasmin from
   http://www.sable.mcgill.ca/software/jasminclasses-sable-1.2.jar
3. Download the optimiser package from
   http://www.doc.ic.ac.uk/~am298/project/optimiser.jar
4. Place these files in a useful directory
5. Modify your classpath so that these packages are in it (e.g.
   setenv CLASSPATH mydir/optimiser.jar:
   mydir/sootclasses-1.2.3.jar:
   mydir/jasminclasses-sable-1.2.jar:’’$CLASSPATH’’ in csh)
6. Modify your classpath so that the rt.jar file can be located (e.g. setenv
   CLASSPATH /usr/lib/jdk1.3/jre/lib/rt.jar:’’$CLASSPATH’’)
A.2 Running

The program can be set to overwrite your original classes or to output the new classes in a different directory, this second approach is advised as it makes it easier to ensure that you are running the correct version of your program.

Once your program is compiled it can be optimised by running the following command (from the same directory as the compiled program):

```
java ic.doc.simulation.optimiser.Main -d 'pwd'/optimised --app
```

Program

This will optimise “Program.class” and all its inner and related classes and place these in a subdirectory “optimised” the program will fail if this directory does not exist before it is run.

Remember to ensure that the CLASSPATH is set so that it will run the optimised programs in stead of the original programs when run, as they have the same name.
Bibliography


[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.


[Ing01] Ricki G. Ingalls. Introduction to simulation. In B. A. Peters, J. S. Smith, D. J. Medeiros, and M. W. Rohrer, editors, Proceed-


