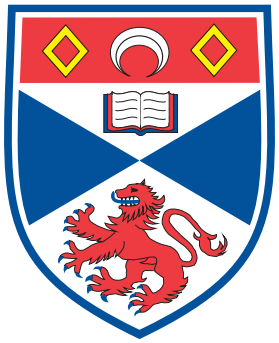
Understanding Thread Interactions



University
of
St Andrews

CS4099: Major Software Project

Ivan E. King

2014-04-04

Supervisor: Dr Graham Kirby

Abstract

Many methods of data visualisation for debugging exist but not many are used to any good effect and even less exist for visualising concurrency. The problem is that it is difficult to understand concurrency interactions, especially with the methods employed currently. This project identifies novel and intuitive ways to visualise information relating to concurrency from a range of sources as a proof-of-concept that visualisation is a viable approach. It targets the Java platform using both the Java Virtual Machine Tool Interface (JVMTI) and the standard Java API as data sources. A range of visualisations are used to display the data, so that a programmer using these tools will gain a better understanding of the threads, their states, the efficiency of the threads, information flow within a thread pool and any liveness issues present. This approach is evaluated against current tools that offer similar functions with the results incorporated into the initial design phases. An iterative development approach is used to help guide the progression of the project toward its stated objectives by utilising experience gained in previous iterations. The results show that visualisation is a viable approach to conveying understanding to the user and many intuitive visualisation methods exist. By leveraging thread pools, new specialised visualisation methods can be created.

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgment. This work was performed during the current academic year except where otherwise stated. The main text of this project report is 19,858 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

If there is a strong case for the protection of confidential data, the parts of the declaration giving permission for its use and publication may be omitted by prior permission of the Honours Coordinator.

Contents

Chapter 1

Introduction

Being able to write correctly functioning concurrent code has been a difficult task since its inception. Many languages support advanced concurrency features to help create highly performing code but at the cost of code complexity. Introducing concurrency drags in a new dimension that the programmer must be aware of and although mechanisms provided in such languages are powerful, many programmers still have difficulty leveraging this effectively. In an article by Vance Morrison he says "Programmers are simply not accustomed to the idea that other threads might be changing memory out...Worse when a mistake is made, the program will continue to work most of the time." [Morrison, 2005]. There is an obvious need for tools that help programmers understand concurrency interactions, increasing the chances that the code is functioning correctly. The aim of this project is to see whether there are better alternative ways to make this information known to the programmer by examining novel visualisation methods.

Chapter 2

Objectives

2.1 Primary Objectives

- Create a tool that allows information about threads to be displayed, with a focus on thread pools. A graphical visualisation approach will be used to show thread state information. The information will include the state of the thread, whether it is blocked, waiting, sleeping or running. It will be presented to the user using an intuitive graphical interface.
- The initial tool will then be extended to show information relating to locks on specific sections of code and include functionality that highlights when threads are waiting on locks. Threads that belong within a pool will be differentiated by threads from other pools, allowing the interactions between the pools to be seen.

2.2 Secondary Objectives

- Extend the scope of the primary objective(s) to cover programs that do not use the Thread Pool pattern.
- Extend the primary objective to highlight deadlock, starvation and potentially livelock situations to the user.

2.3 Tertiary Objectives

- Evaluate the program using human subjects via surveys
- Extend user evaluation to include comparison of other visualisation tools.

Chapter 3

Context Survey

Before starting the project, current tools and research had to be surveyed to determine the scope, objectives and approach taken to address the problem posed in the abstract. Most of the tools are not produced purely for concurrent programming purposes but this will be the main ability that will be looked at. Java and C# are two of the most popular languages, this makes them ideal candidates for profiling tools. It therefore makes sense to start the survey with the tools provided by Oracle. Higher level languages such as these also have complicated concurrency mechanisms to address issues that arise in multi-threaded application development.

3.1 VisualVM

The first tool that was surveyed was the VisualVM tool created by Oracle. This tool details information about the JVM and can be either integrated with an IDE (e.g. Eclipse) or can be run as a stand-alone binary. It contains a GUI that shows processes running within the JVM and allows the user to click on various tabs relating to JVM information. Not only does it show local information, it also allows remote connections to virtual machines. In addition to showing local and remote JVM's it also displays:

- CoreDumps
- HeapDumps
- Snapshots (Archives of data about a particular JVM process at a specific point in time).

Although VisualVM does provide some dynamic information on the threads, it only shows basic information related to their current states and nothing on for example, where in the code the threads are executing. It displays the thread states as a moving bar graph and uses colour to delineate the state of the thread. In addition, the thread states are misleading as it only shows a sub-set of the waiting and timed wait states. A thread that is sleeping is in the timed wait state but there are alternative ways a thread could enter this state, apart from invoking sleep. Using this display makes it difficult to spot deadlock within the threads, and more importantly, where in the code and what threads are locked. The situation where more than one thread is blocked at any one time is highly likely, and therefore is difficult / impossible to deduce which threads are blocked on each other using this method.

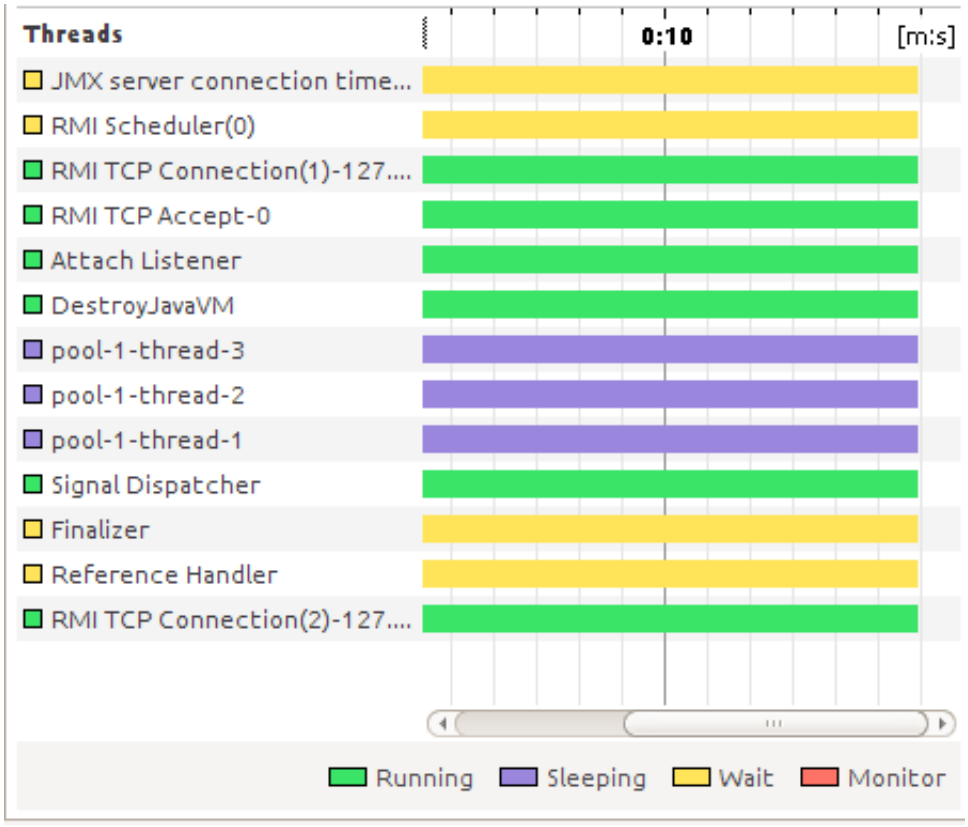


Figure 3.1: An example how a thread state visualiser looks like. Taken from VisualVM.

3.2 JIVE - Visualising Java in Action

This is another visualisation tool that contains information about threads, object information and displays method call traces. It uses a samping method to take 'snapshots' during the programs execution to minimise the amount of data gathered. The trace data for the program is generated by instrumenting the code using a special byte-code compiler. In the research paper by [Reiss, 2003a] it is easy to see that it has a similar layout to the information shown from VisualVM, but with some additional information on the active threads. This included whether a thread was running within a synchronized section and information about the blocking frequency of threads. The first obvious disadvantage to their approach is that the trace data is generated in snap-shots and so doesn't provide a complete picture of the dynamic nature of the program. This is especially important when looking at threads as there is much lower probability that a change is picked up by while sampling due to the long periods in between the sampling stages, missing out potentially important information.

The visualisation method used is similar to the one used in the VisualVM tool except the bars were vertical instead of horizontal. Each thread had their own window and the number of method calls per thread was denoted by the vertical size of the bar. Estimated memory usage was shown by the horizontal size of the bar and the thread state was denoted by the colour of the bar. Output from the trace tool was in the form of XML documents that contained all the information for that particular snap-shot [Reiss, 2003b]. It could either be sent over a network connection or loaded locally into the visualiser. Because the main aim of this tool is not to give detailed information on the interaction of

the threads, they prioritised performance over in-depth information. The impact on performance cannot be ignored completely and it would be ideal to minimise the affect on performance and this may influence the scheduling of the threads.

Figure 3.2: Visualisation example in JIVE

In figure 3.2, the thread windows can be easily seen on the right and the smaller windows on the left show classes that are involved in the execution of the program. The disadvantage to their visualisation approach is that its not aesthetically pleasing despite it being relatively easy to pick out relevant information. It also exhibits scaling issues which can begin to be seen from the figure above, because each class has its own window, as the program grows the windows will get smaller and it will become difficult to see the information picked up by the tool. Many of the approaches used by this tool have been approximations e.g. counters to approximate time spent in a specific class and exact thread information is never derived. This approach will be problematic in this project because accuracy of the interactions is important. Any gaps in collection of data could miss vital information about the behaviour of the threads. It is obvious to see from the result of the research in this paper that the objectives for the project focused mainly on creating a very simple but fast dynamic visualisation tool, with an emphasis on minimising the overhead in retrieving the trace data. They had seen that previous work in this area that collected data more accurately had imposed an unacceptable performance impact.

3.3 ThreadMon

ThreadMon is a dynamic analysis tool used to visualise the interactions between user-level threads and kernel level threads. According to the paper by Cantrill & Doeppner it allows direct deduction of any resource contention within a program and can show how the user-threads are mapped onto kernel threads within the operating system [Cantrill and Doeppner Jr, 1997]. It is a general program that targets the operating system thread implementations and is not targeted at specific language platforms. Both states of the user-level thread and the kernel threads are shown allowing the scheduling to be leveraged by the application developer. This tool is primarily targeted at gaining thread information for the user so no information relating to stack or heap allocation is available. Class information is also not available because it works at a lower level that the JVM and so could be used with any language or platform. CPU utilisation is the only other information apart from the threading information, that is shown.

The information is gathered from various operating system services and so could theoretically target any language that runs on that operating system. But it will be unable to look at language specific concurrency mechanisms such as monitors and semaphores. Information gathered by these services is then passed to a visualiser which then draws the information in some intuitive format. From the screen-shots it looks like the visualisation methods used are simple but show relevant useful information for the programmer in a clear manner.

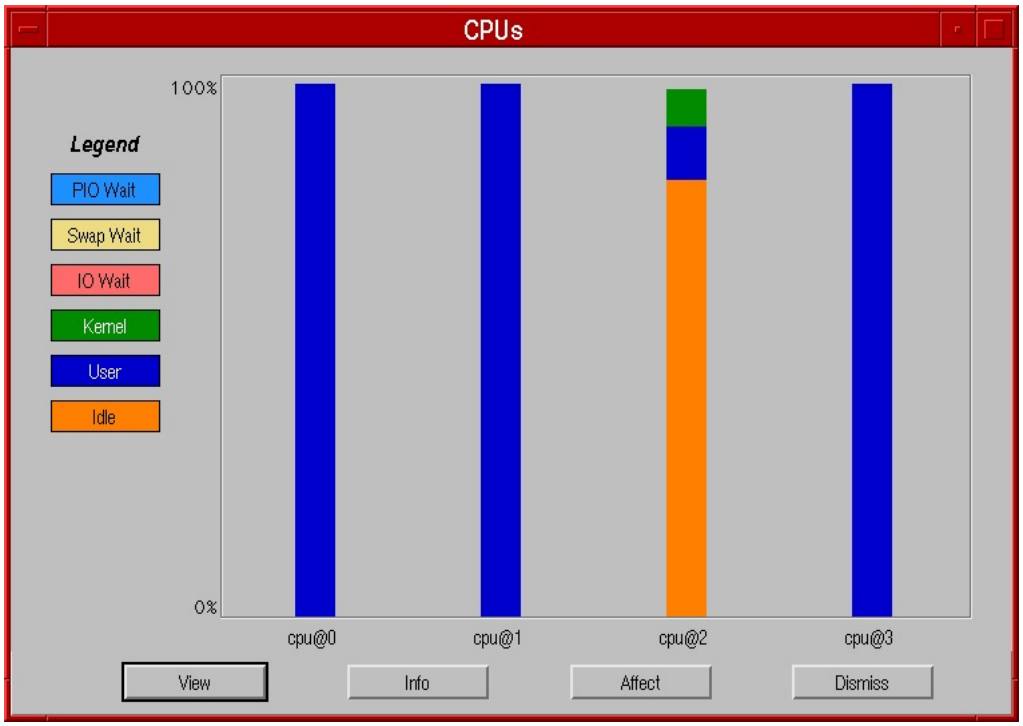


Figure 3.3: CPU Utilisation in ThreadMon.

Although this tool does not target the Java platform, their research has uncovered various other considerations to take into account when designing a thread tool. Scalability of the tool is one of the next goals for this research but for the purposes of this project, conversely, scalability will not be of primary importance in this project. This is because uses of the tool will be working on small sections of code and the overall interactions in a big system would be too massive to show inside a single tool. Another important aspect to take account of is the thread-model used in the target platform, Java has a one-to-one mapping between Java threads and OS threads hence the OS is responsible for scheduling. In this instance, the JVM will map a Java thread onto a native operating system thread and the operating system will handle the scheduling of the native threads. Other models like many-to-one and many-to-many would normally have their own scheduling done within the threading library as well as the scheduling of the native threads within the operating system. Resource contention can be deduced by using the tool but the way the user threads are mapped down restricts control of them.

3.4 YourKit Java Profiler

This is a fully-fledged profiling suite that gathers information about all aspects of an executing program and displays them in intuitive and colourful ways. With respect to synchronisation, it shows information relating to all currently executing threads and their states, includes a dead-lock

detector and details of a monitor lock history. The visualisation method used to display the currently executing threads is no different to any previous method and uses a dynamic bar graph to show each of the threads and a colour to denote which state that thread is in.

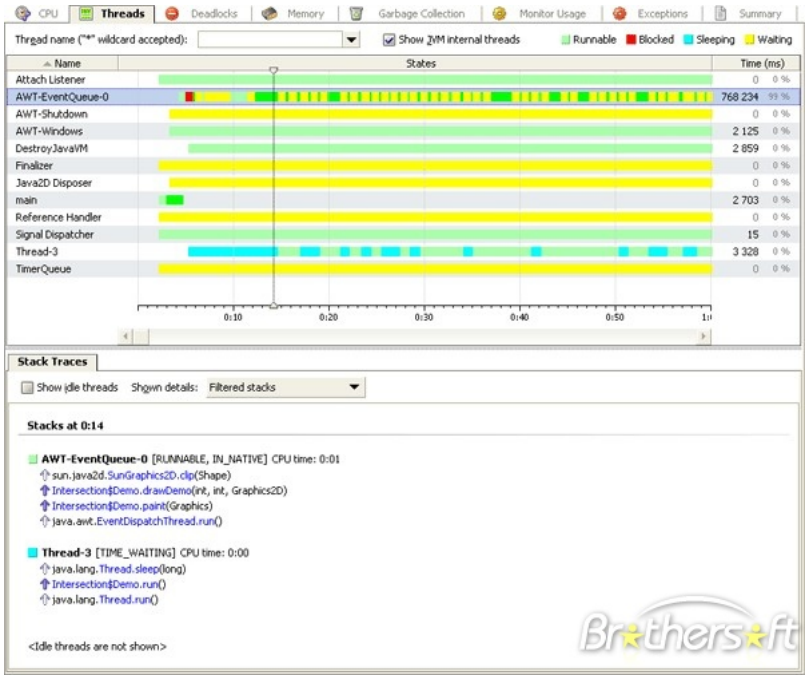


Figure 3.4: Currently executing threads in YourKit Java Profiler. Stack traces for each thread are also displayed.

One of the unique aspects of this profiler that has not been seen in previous tools, is its ability to record the acquisition of locks and calls to *wait()*. It can also record the number of times a lock has been previously held and the time spent waiting to acquire the lock. The dead-lock mechanism is also a distinctive trait of this program. Not only is it able to detect dead-lock but it is also able to show the threads involved and what objects the threads are dead-locked on. This tool is different from the rest in terms of it being a commercially available tool and not the result of research. It encompasses a wide range of aspects not only including thread information but also memory usage and object related information. It specifically targets the JVM but doesn't specify how it gets its data. It has the advantage of a large scope of data but shows most of this as statistics.

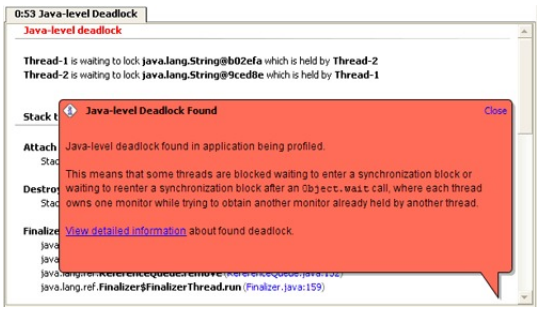


Figure 3.5: Dead-lock detection available in YourKit Java Profiler.

3.5 JProfiler

This is similar to the Java Profiler from YourKit, it is an extensive profiling suite that provides a lot of tools to optimise and enhance Java applications. On attaching to a JVM the user is able to select how invasive the tool is, selecting from full instrumentation to sampling. In terms of thread information, it provides all the information that is available in the YourKit Java Profiler but lacks dead-lock detection. It displays the data in much the same way as the YourKit tool does and also supports information related to locking histories and monitors. Most information relating to threads is shown as textual trees or a simple list of recorded information. Dynamic visualisation of thread state is done by a dynamic bar graph that correlates the bar colour to the state the thread is in, exactly the same way as the two research tools discussed earlier.

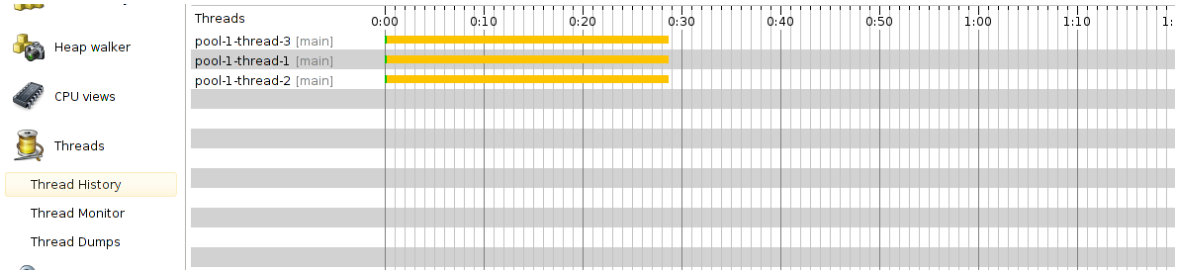


Figure 3.6: Threading information shown in JProfiler.

The method used to display information here is used in many other tools as well and is not unique to JProfiler.

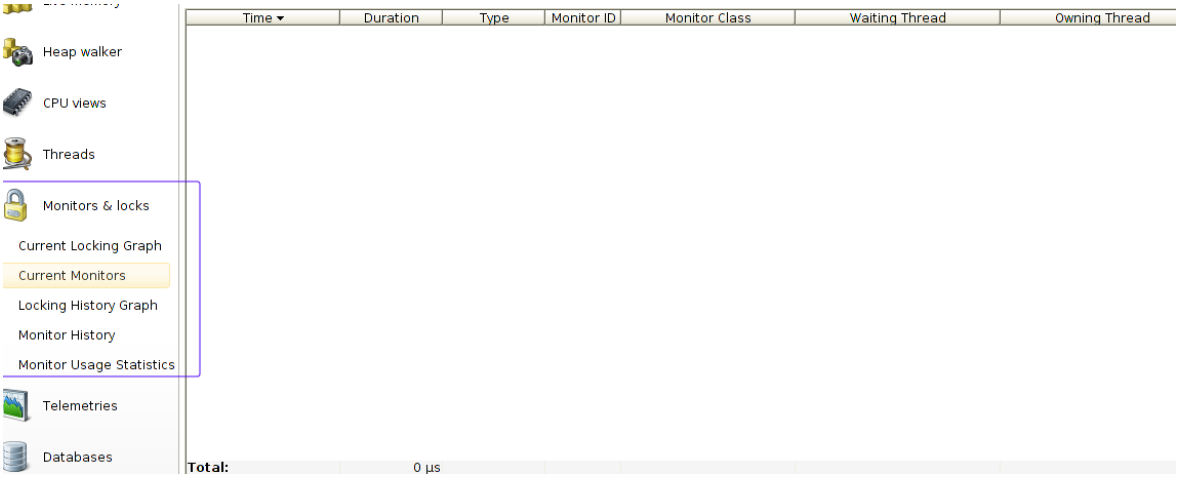


Figure 3.7: Monitor information would normally be shown in this table here. Tabs on the side direct to alternative information.

The various tabs seen on the left highlighted by the blue box allow information relating to locking and monitor history to be seen. Information can either be in a textual format or drawn as a graph. There is a lot of similarity with the YourKit tool except it lays it out in a different manner.

3.6 JVMMonitor

Another popular tool is actually a plug-in for the Eclipse IDE and itself, is written in Java. It is an open-source tool that allows dynamic inspection of the JVM and any Java programs that are running. It doesn't offer as much information as YourKit or JProfiler and requires the use of the Eclipse IDE where as the previous paid-for tools would easily integrate with the most popular IDE's or would run as a stand-alone program.

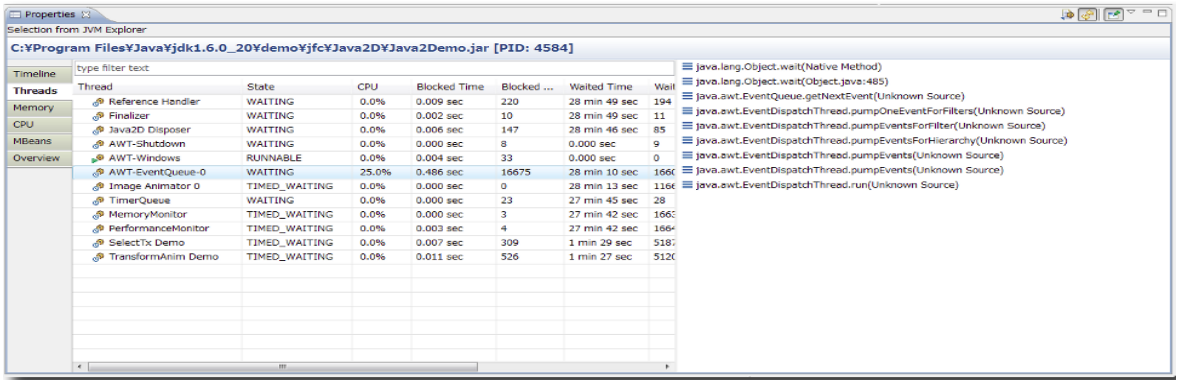


Figure 3.8: All the information relating to threads shown in the JVMMonitor tool.

Although the tool doesn't use advanced methods to display information like in the previous tools, it does show a lot of information within the table. Thread data such as blocked time, wait time and CPU utilisation can be seen. A stack trace of each thread is also available in the right hand window. Apart from this tabular format, there are no other display options for the data. The obvious downside to this tool is that it is a plug-in and requires other software.

3.7 AppPerfect Java Profiler

This is another stand-alone program that allows analysis of the JVM and all running Java processes. Like YourKit and JProfiler it doesn't just focus on threads but instead displays a large amount of information regarding the JVM, making it easier to fine-tune Java programs. It easily provides information on thread states, all active monitors and even has deadlock detection. Much of the information is available from previous tools and it doesn't offer anything unique. More importantly, it lacks visualisations for the data it collects.

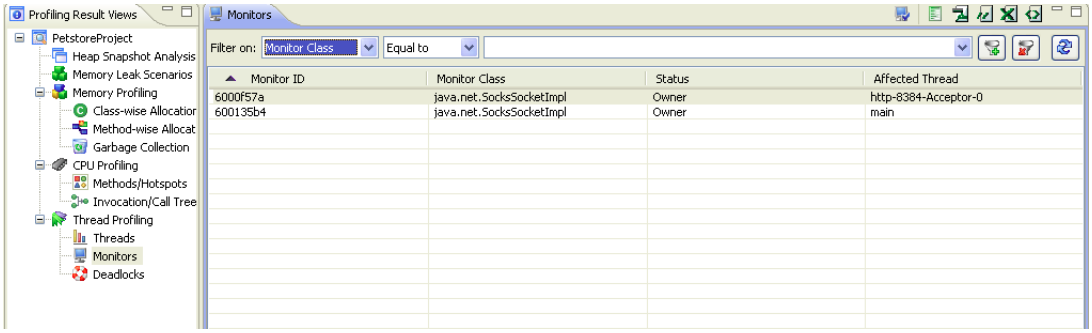


Figure 3.9: List of currently active monitors shown in AppPerfect Java Profiler.

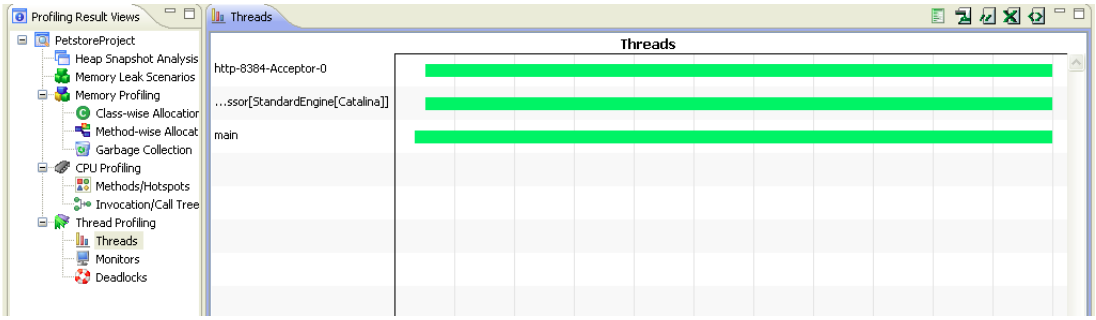


Figure 3.10: Threads view in AppPerfect Java Profiler.

3.8 Intellij IDEA Community IDE

One of the most popular IDE platforms for Java is the Intellij IDEA from JetBrains. It offers a comprehensive set of tools for developing in Java and a wide range of plug-ins. The standard community version of the IDE contains a debugging suite similar to the one seen in Eclipse and offers some information relating to concurrency, including the facility to produce a complete JVM heap dump. With respect to threads, the user is able to view a list of running threads in real-time including the threads state. There are currently 3 thread states that it reports, running, wait and unknown which correspond to the underlying Java thread states. In addition to the list of threads, the user can also select a thread and view its variables and stack frame while the program is executing. The method used to show this information is textual and does not use any visualisation methods [JetBrains, 2014].

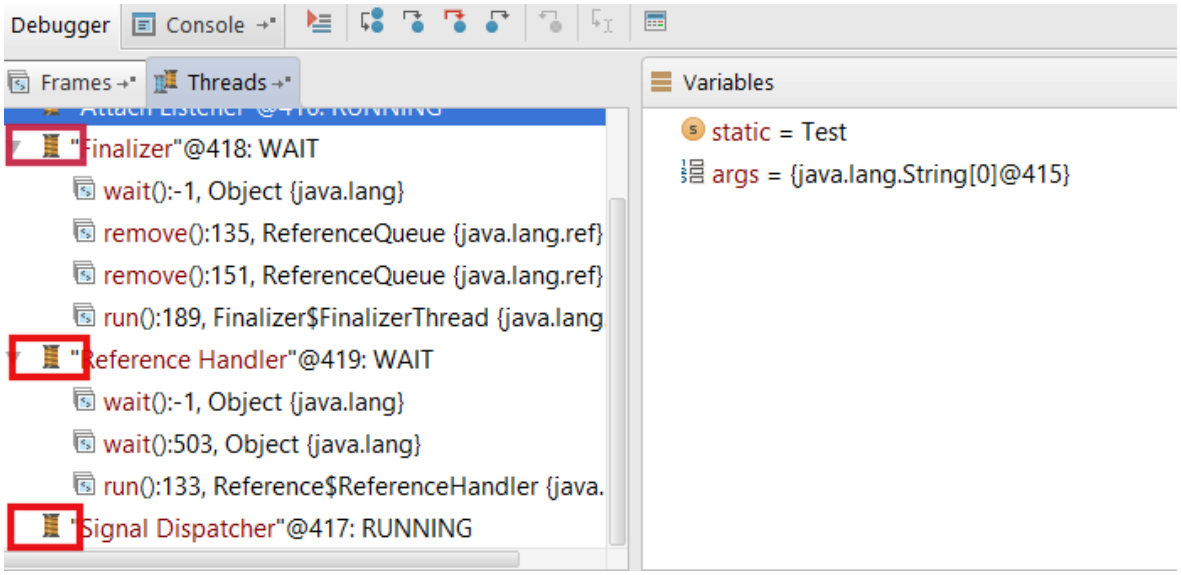


Figure 3.11: Debugging interface from the IDEA Intellij Java IDE

While other IDE's and profilers offer some sort of graphical feedback option, Intellij does not do this. It does have a small icon to show some information about the thread such as whether the thread is active, suspended, frozen or at a breakpoint and is the only graphical information shown in the debugging interface. These can be seen highlighted on figure 3.11. One possible reason why there

may not be any large graphical representation of such information is to do with a lack of space to show any visualisation. A brief look at the profiling section within the plug-in database revealed no credible alternatives to the debugging interface regarding thread information.

3.9 Eclipse IDE Java

Arguably the most popular IDE for Java is Eclipse from the Eclipse Foundation. It provides almost identical facilities to the IDE presented by JetBrains including a debugging suite. With respect to information regarding concurrency there is not as much as in the Intellij IDE, the only information in real-time is a list of currently executing threads, shown in the view seen in 3.12.

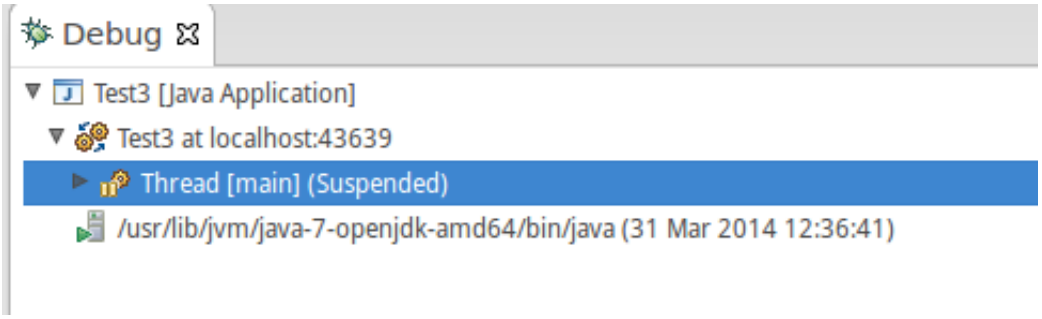


Figure 3.12: Debugging window shown in the Eclipse IDE.

Looking at the Eclipse Marketplace which is the main plug-in database for Eclipse, there is one plug-in that offers more information on threads. Lockness is a plug-in that allows analysis of Java thread dumps and while this technically does not allow it to monitor in real-time, it still offers interesting information. It provides more in-depth information than the thread information provided by the Intellij debugging suite or Eclipse. For example, in addition to the thread states shown, it provides potential reasons as to why they're in that state. The screen shot below shows all the possible thread states and is taken from the online documentation of the tool [cr, 2006]. Information on locks and waiting threads is shown in a tree format and it is also detects locks that are causing bottlenecks. A major disadvantage to this too is that it is a static analyser but more importantly,

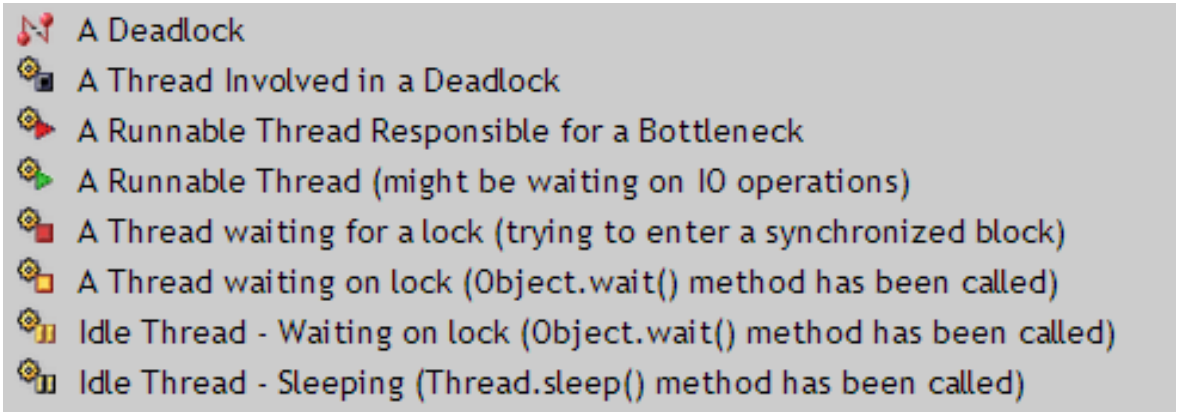


Figure 3.13: Showing possible thread states using the Lockness plug-in.

Eclipse, Intellij and Lockness all suffer from fundamental deficiencies in the way that they present

information to the user. Quite often when there is a lot of information available, presenting it in a textual format makes it hard for the user to find the required information and the effect is exacerbated in a dynamic environment. Combining these different sets of data into visualisations would make digestion of the data much more manageable for the user, hence making visualisation an interesting approach.

Chapter 4

Requirements Envisioning

Due to the nature of the project, the amount of initial documentation is small and more dispersed throughout the development of the tool.

4.1 Essential Use Cases

This is a high-level outline of how the user will interact with the tool and shows the general steps after a required action is selected. The *Actors* in these cases are the file system, the user and either a running or yet to be executed target Java program.

Essential Use Cases	System Responsibilities
Load target program	• Select program from file system. • Initialise tool and execute target program.
Select visualization	• Detect mouse or keyboard entry. • Select correct visualization and request data. • Render visualisation onto screen.
Attach to target program	• Detect all eligible running programs. • Attach to selected target program. • Initialize tool.
Execute target program	• Allow search of the file system. • Execute requested application

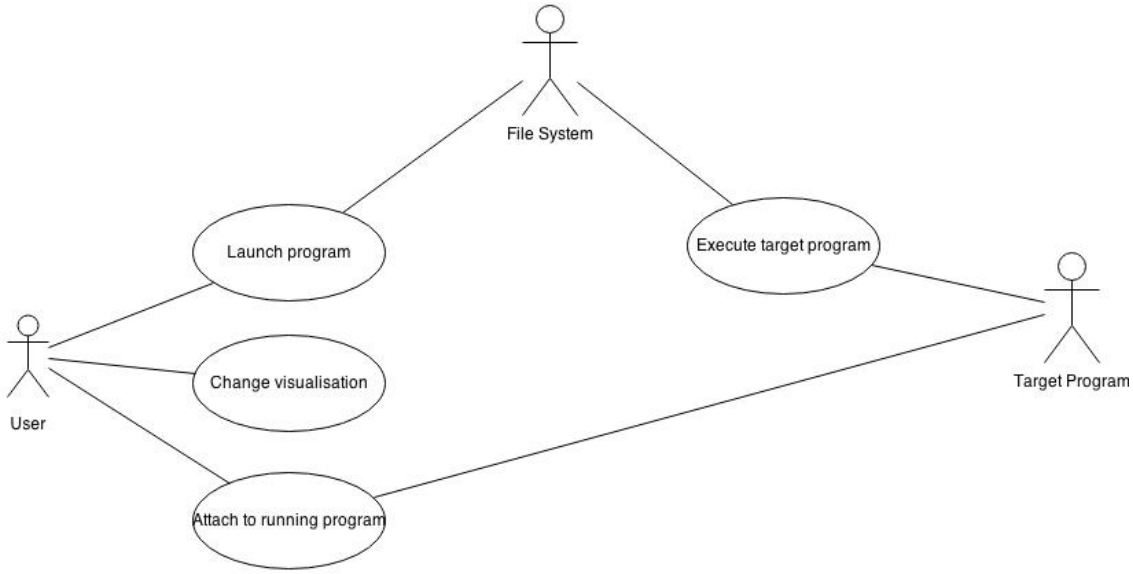


Figure 4.1: Use Case diagram.

This gives a high-level description of the typical interactions that will happen between the actors without settings any concrete details about how this will be done. This typically will not change throughout the development process unless further functionality is added to the system.

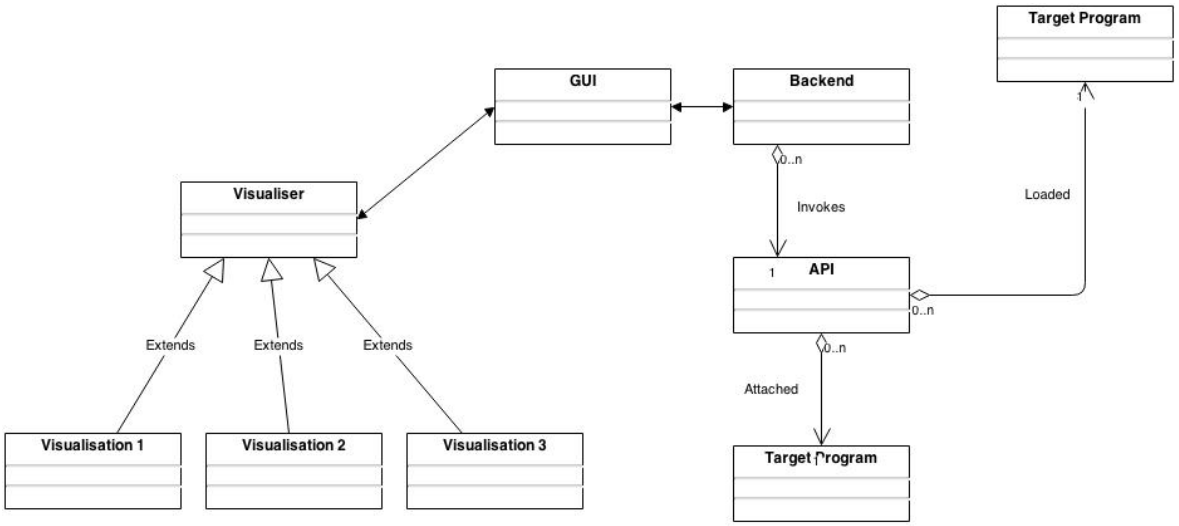


Figure 4.2: Domain model showing domain specific interactions.

The domain model consists of domain-specific entities, their relationships between other entities within the overall system and the associations between the entities. It provides a high-level structural envisioning of the system, although by no means is set in stone and is subject to change in later development stages.

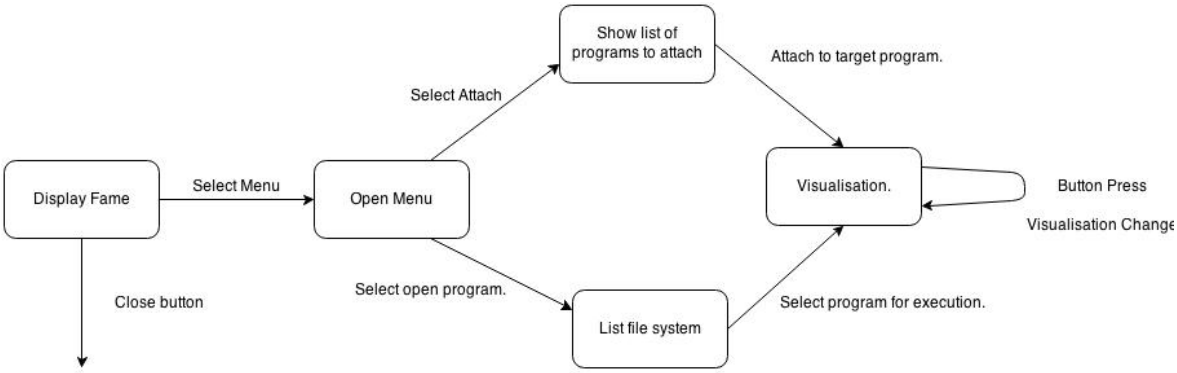


Figure 4.3: User interface flow diagram.

A user interface diagram specifies how the user is expected to interact with the system when it is completed. It outlines what actions are expected to take place and the responses in terms of what the user sees on the screen. Because this is a software debugging tool, the user interface is not a big priority as long as it shows all the required information and it is accurate.

Chapter 5

Software Engineering Process

The effect of the type of development methodology can be detrimental on the productivity and quality of the produced product if the wrong one is chosen. Arguably the worst result is building a product which is not the one that is wanted. For these reasons alone it is imperative that the right development technique is chosen according to the project domain and the business requirements.

This project is primarily exploratory in nature and so doesn't lend itself well to a traditional Waterfall phase-based approach. Changes in the requirements can be sudden, the exact long-term goal is unknown and the requirements may be ambiguous. Using a waterfall model in this situation increases the chances of delivering the wrong product because there is little feed-back between the stages. Testing is another stage that becomes condensed, although in this particular project, robust bullet-proof code is not a requirement.

While the focus of using Agile techniques revolves around the ability to adapt to changes in requirements, promoting development via an evolution of stages. It encourages working software over large amounts of documentation, and collaboration with customers as opposed to negotiation [Beck et al., 2001]. All of these aspects are much more applicable to an exploratory project such as this one, which is why it was chosen as the as the general approach to the project. In addition to agile, iterative & incremental design and implementation was used as the development model, which is common when an Agile approach is adopted.

Advantages to using an iterative & incremental approach to design is that it allows for feedback after each iteration. It also allows the chance to learn from the previous development cycle which is particularly good in this instance because unknown API's were being used with little to no examples. Feed-back is extremely important in a project such as this one because it facilitates the evolution of the project. Without feed-back, it would be difficult to decide whether the right approaches are being used and whether the project is heading in the correct direction. After each iteration, a working increment of the tool was created and verbal feedback was given including validation of the increment against the requirements. Notes for these discussions were kept and used as the basis for the next design and implementation cycle in the form of a simple to do list of high-level functionality to be added. Another side-effect of the incremental & iterative approach is the lack of upfront documentation regarding the design of the software, which is seen in the requirements specification part of this report. For such a small project, no other large bits of documentation are needed, with the source code presenting the major origin of the documentation. The first design and implementation phase took longer than the rest due to the unfamiliarity of the of the project area and APIs. This was probably the most important stage of all the iterations because it provided the

biggest boost in knowledge surrounding the domain. To support this development process, software version control was used in the form of Mercurial code repositories that were provided by the school. The Google Java code style was followed to make the Java code as clear as possible, which is largely based on Sun's own coding standards [Google, 2014].

Chapter 6

Ethics

Due to the tertiary objectives that relate to the possibility of using surveys for evaluation, special care will be needed to ensure the data stays safe. To this end, any information gained from the survey will be anonymous and encrypted on the virtual host provided by the school. An initial pre-assessment ethics form was completed and it was deemed that a full ethics assessment was not required.

Chapter 7

Background

I had no prior knowledge about the JVMTI or the thread pool pattern and so this section is a bit of background about what I've learnt regarding both these ideas. Both are mentioned due to their importance towards the project.

7.1 Java Virtual Machine Tool Interface

The JVMTI is a native API that allows external programs to inspect the state of a JVM when it is running. It also allows control over the threads and processes running within the JVM. It was introduced as an improvement to the previous interfaces that offered similar services such as the Java Virtual Machine Profiling Interface (JVMPI) and the Java Virtual Machine Debugging Interface (JVMDI) which have been discontinued since SDK 1.5.0. Some of the reasons behind the removal of the previous interfaces were because it was very invasive and therefore difficult to maintain, didn't scale that well and didn't provide any error information when things failed [Prasad et al., 2004]. The JVMTI proceeds by allowing a library to be loaded into the JVM which is then able to invoke functions. An events service is also provided by the interface that allows callback functions to be defined in case of a particular events occurring within the JVM. For example, you can register a callback for whenever an object is allocated, a method is called on an object or when the garbage collector is called. It also permits the implementor to control the threads from within the JVM, create new objects on-the-fly, invoke methods, change byte code associated with classes and even allows the implementor to intercept JNI function invocations and re-direct them.

A range of functions are provided by the JVMTI itself but there are also functions provided by the Java Native Interface (JNI) that can also retrieve information. JNI is the high-level interface that allows the JVM to call and be called by native libraries and applications. It is typically how cross-language support is implemented.The functions related to threads and thread groups allow the retrieval of information from the JVM in a clean and quick manner. For this reason and the fact that the JVMTI contains a great variety of capabilities, it was chosen as the approach to gain the information.

In a typical setup, a library known as an agent is created that contains calls to the JVMTI. The JVM has a choice whether to load the library at the same time as the JVM is started (i.e. when the Java program is started) or it can be loaded and 'attached' after the Java program has already started execution. Once the library has been loaded the actions it takes depends on the implementations of two functions with the following prototypes:

*OnAttach()*is invoked if the library is loaded into an already executing Java program and *OnLoad()* is invoked if the library is loaded at JVM start-up. The following pseudo-code is how the library is typically setup, regardless of when it is loaded into the JVM.

```
JNIEXPORT jint JNICALL Agent_OnAttach(JavaVM* jvm, char* options, void*
    reserved)
{
        //Get a reference to the JVM - This is how we access JVMTI
            functions.
        //Register for events of interest
        //Register callback functions by assigning function pointers
        //Setup a global monitor that can be used for synchronization
        //Return
}
```

Because the JVMTI is a native interface, manual memory management is required to manage memory allocated by the interface functions. For example, arrays that are returned by a JVMTI function must be freed by a function supplied by the JVMTI. Variables that refer to Java objects and variables within the JVM are generally local and only valid in the function that they're returned to. References that require a longer life time must be converted to a global reference using a JNI function. There are advantages and disadvantages to converting them to global references, it is regarded as better practice to, if possible, leave the references as local.

Agents can control other threads within the JVM, they are also able to create their own native threads that are able to handle events or communication. This is useful if you want to poll for information inside the JVM. Although the JVMTI doesn't provide events for everything, it generates events for integral actions inside JVM which include:

- Exceptions - Event fired whenever an exception is thrown within the JVM.
- Method Entry & Exit - Whenever a Java method is just entered and when it leaves.
- JVM state change - Fired when the JVM changes state, these include, JVM death, JVM initialization and JVM start.
- Class Loading - Whenever a class is loaded into the JVM this event is fired and enables access to the thread loading the class and the class itself. This facilitates instrumentation.
- Monitor Wait & Waited - Fires when threads begin waiting on an object monitor and just after a thread has finished waiting and is about to enter the monitor.

More information can be found from the JVMTI reference pages http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html .

7.2 Thread Pools

The thread pool pattern is a efficient solution to a common problem. The problem is that there are a lot of independent tasks that can be executed concurrently with each other. If a simple approach is used where a thread is instantiated for each task at once and permitted to execute, the performance of the machine may degrade owing to resource starvation. Another approach would be to limit the number of threads created and just instantiate more once the previous ones have been killed. This would also affect performance due to the high overhead of creating and destroying a thread.

A thread pool mitigates these disadvantages by creating a number of threads, normally a fixed amount, that will execute jobs they are given and then simply sleep until another job is in the queue. This reduces the overhead from creating lots of threads repeatedly and reduces the risk of resource starvation. Within the thread pool there is a queue where tasks are submitted to, and the worker threads dequeue these tasks once they are ready. If no tasks exist, the threads simply sleep until one becomes available. Common mistakes when implementing a thread pool include not making access to shared data structures thread-safe, thread leak, which is where references to a thread object are not released properly preventing it from being garbage collected and lastly the re-instantiation of threads when they are killed by the task they are executing. The complexity of these issues and commonality of the problem is the reason why many standard libraries (Java and C#) have implementations which allow some degree of customisation.

The disadvantage of using a library implementation is that a lot of things are hidden away under the interface and leveraging a thread pool well, requires knowing a bit about the behaviour of what is happening underneath. Common problems when utilizing a thread pool, include dispatching jobs that are 'too small' for the thread and not worth the overhead in creating the thread. The biggest problem and easiest to over look is when the tasks are not independent of each other may depend on other tasks up-or-down stream from its current position. In the worst cast this can lead to a complete program deadlock as all the worker threads are waiting on each other. A less severe repercussion is an IO bound task, in this case a CPU core is wasted while the thread it is executing blocks waiting for IO. A thread pool with an adaptable number of worker threads would assist these situations, although not without a small performance hit. In the case of the IO bound task, it can simply be swapped with another newly created worker thread that is able to execute while the other is blocked, and then switch the original back in once its finished waiting on IO. This does however enforce the need for the pool to 'know' when a worker thread has blocked on IO which may be difficult or impossible. While this cannot be forced because the OS scheduler controls things, it can be influenced by changing the thread priorities dynamically.

Java's version of the thread pool implementation is quite extensive and offers a lot of customisation with regards to the behaviour of the threads and the behaviour of the pool in certain situations. The Executor framework provides basic implementations of the standard thread pool pattern but also variations such as thread pools based on timers and work-stealing thread pools. For the purposes of this project, the standard thread pool implementation that is provided as part of the Executor framework will be used.

Chapter 8

Design

Throughout the development of the tool, the design and implementation stages were interleaved so the following sections on design will be normally followed by an implementation stage.

8.1 Initial Design

From research conducted in the context survey, many tools that provide dynamic capabilities do not have detailed concurrency information, and if they do it is in a textual form. The novel visualisation methods that are being investigated will help to inform programmers and make debugging concurrent programs less of an uphill battle.

The key hazards in concurrent programming are the non-deterministic scheduling of the threads, which can result in a range of invalid states if the threads are not controlled properly when they share data. This is commonly known as a race condition. Liveness is the notion of *making progress* within an application. If the wrong ordering occurs for synchronization it may lead to threads making no progress because they are waiting on each other, this is known as deadlock. The third hazard is related to the performance of the application that utilizes concurrency. Concurrency is difficult to analyse from a compiler's perspective and so this is the reason why many compilers will switch off performance optimizations when multiple threads are detected.

To aid the understanding of these influences and in an attempt to improve the understanding imparted on the user, the tool will help by visualising important concurrency information. Some of the properties that can be deduced from the implemented visualisations include:

- The states of each thread throughout the program.
- The performance of threads throughout the program.
- Threads that own a monitor as well as threads that are blocked.

With these aims in mind, it seemed sensible to start with the simplest case possible and build on this in an incremental fashion. Therefore, the initial design does not include any visualisations that exclusively deal with the thread pool pattern itself but focus primarily on a simple multi-threaded application. Once this approach has been seen to be feasible it can then be specialized to deal with thread pools. An alternative approach could have started looking exclusively at thread pools and then expand the visualisations to general multi-threaded applications. The reason this perspective wasn't pursued is due to the lack of familiarity with the thread pool pattern in general. It was a

further limited by only a basic understanding of the synchronization primitives in Java and their common uses.

8.2 Initial Design Decisions

Debugging tools are either active or passive in their analysis of a program. Active tools dynamically monitor the program as it is running and show this data in almost real-time, while passive ones collect data for analysis off-line, after the program is running. The information collected for off-line analysis is commonly known as a trace. Deciding whether the tool was going to be active or passive was the next decision to take, and depending on the type of profiling being done it can either facilitate this or make implementation more complex. For targeting the Java platform, there are two effective methods for obtaining information. Firstly, instrumentation at either the source or byte-code level and the second method uses callback methods and events on the JVMTI. Both of these can be adapted to create a trace of the running program or they could used to generate data that is used in real-time. A trace can be generated on one machine and then analyzed on another machine, removing the need to run the target program on a powerful computer. On the contrary, it requires the user to effectively run the program twice before the user can make any decisions on it. A dynamic tool also has the advantage that it can be stopped anytime or once it has passed the region of interest. There isn't any easy way to do this in a passive profiler without some sort of pre-processing. But having a dynamic tool raises other complications regarding the retrieval of the data and the analysis of it. The dynamicity places time constraints on the debugging tool as to how long it has to process the data before it gets a new set of data. For the purposes of this tool and the scale we are looking at, with time scales in the region of seconds, this should not be a problem.

In addition to the time constraints imposed by a dynamic analysis, the accuracy of the data and the performance degradation due to the extra method calls must also be considered. For the purposes here, accuracy will mean, is the data I am getting from the program the correct data, I.e. is it the latest set of data and not a set from 5 seconds ago when the update rate is every 2 seconds, if so does the visualization correspond to this data? Both of these aspects form part of the verification process in testing. Accuracy is dependent on the quality of the implementation and not on whether the tool is real-time or uses trace information. Another important aspect closely related to on line and off line debugging tools is that invasiveness of the methods used to extract the information and its affect on the performance of the target program. Regardless of whether code instrumentation is used or event hooking in the JVMTI, there will be a small over-head added to the execution time of the program but this is negligable. The sheer complexity of implementing bytecode instrumentation is unappealing in itself and while libraries do exist to assist in this matter, they may not be flexible enough and time restrictions on the project make this approach unattainable.

8.3 Implementation Language Decisions

The original choices of implementation language were between Java and C++. After some research into graphics libraries and techniques for extracting the required information, Java was picked as the implementation language. A data visualization library such as D3 was also briefly considered but ruled out due to the lack of skill in web development technologies. Using C++ while being competent in C, would of meant learning another set of features on top of working out how to visualize the data. The reason why C++ was considered was based on the motivation to learn a new language and the large support of graphical libraries that were present including Cinder, Cairo and OpenFrameworks. On reassessment of the available libraries it was decided that the functionality provided was far too much and the time investment needed to get to grips with the library were too high. C++ also lacks

a default thread pool implementation, adding more work and increasing implementation time. Once Java had been selected as the implementation language, a suitable graphics library that provided enough power to create complex visualizations had to be selected. A pre-condition on the selection is that it musn't be difficult to grasp. Common Java graphics libraries include Java's own Swing library but also Processing and a range of other primitive graph plotting libraries such as G and R via JRI. Swing is probably powerful enough but suffers from a lack of common abstractions, thus inhibiting productivity. It was suggested that a discussion with someone within the HCI department would be of use, and after consultation with Dr. Nacenta, Processing was picked as the library to implement the visualisations. It provides a lot of good abstractions over low-level event handling and has a huge wealth of documentation and examples to go with it but does not restrict you to only using these abstractions [Fry et al., 2014].

8.4 Architecture

A clear distinction can be made between the separate parts of the tool. At the front-end there is a GUI and multiple visualisations, while at the back-end there are data-structures and objects that represent entities of interest. There is also a third part of the program which is the interface that the backend communicates with to retrieve information from the target program. A strong de-coupling of the concerns of each of the sections naturally results in a model-view-controller type architecture. This separation allows the sections to work against an interface while their concrete implementations remain independent. While the implementations of parts of the tool were changed the overall architecture stayed constant throughout development.

8.5 Initial Visualisation Ideas

This section details ideas about the kind of information that would be helpful to a programmer and some of the reasons why it would be helpful. It forms the basis of the implementation efforts and helps guide the design for the visualisations. A majority of the visualisations here are for multi-threaded applications and could be applied to an application that utilizes thread pools.

- Lock History - This would be on a per thread basis, detailing every monitor lock that the thread has gained up until the current point in time. This can help pinpoint deadlock situations and identify high lock contention.
- Highlight un-synchronized code - This would effectively track a threads trace through an execution path and highlight sections of code, in particular memory accesses, where multiple thread access is unprotected by synchronization.
- Data/Race Condition detector - This would be helpful in highlighting structural issues and timing themes. It would be difficult to come up with a particularly effective solution because it requires a global view of the source code.
- Extension of Lock history - This would require extending the information about locks to highlight potential starvation situations between threads. A time metric would be used as a threshold value to decide whether the waiting time for a thread is acceptable or not, explicitly showing one consequence of thread interaction.

The following are potential visualization ideas that would implement some of the above.

- Thread tree - A diagrammatic view of all the threads in the executing program. It would give a global view of the actions of the thread throughout program execution. Areas where multiple

threads converge would show a lock contention between those threads, emphasizing potential bottlenecks.

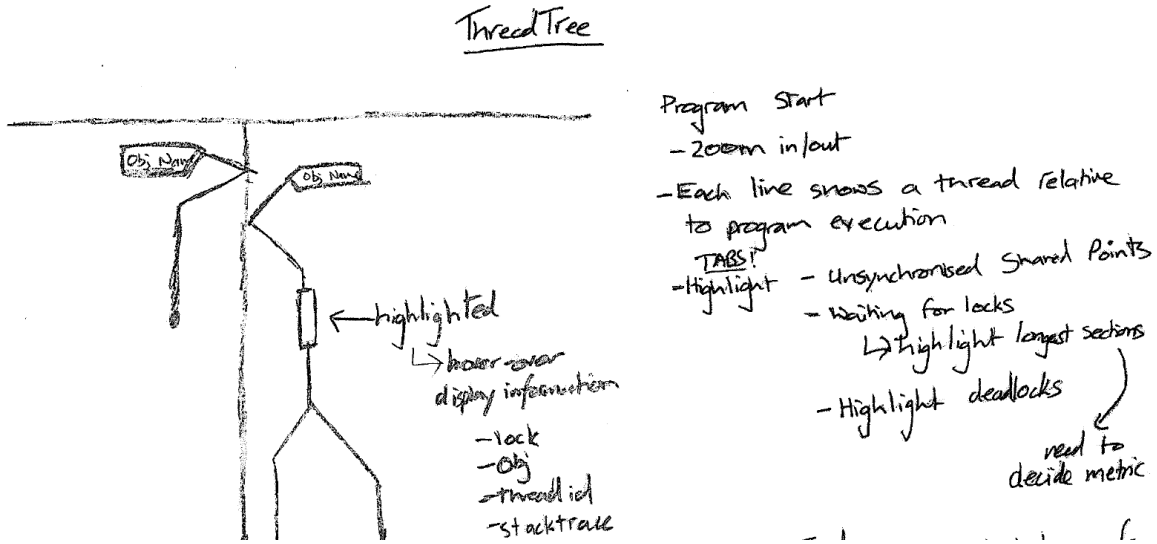


Figure 8.1: Example ThreadTree visualisation.

- Thread similarity graph - Allows the visualization of the amount of data that is common between the threads. It can guide the programmer to areas where synchronization should be properly implemented, which is not always the case. In situations where there is a lot of shared data, it can be difficult to track statically which threads have access to what.

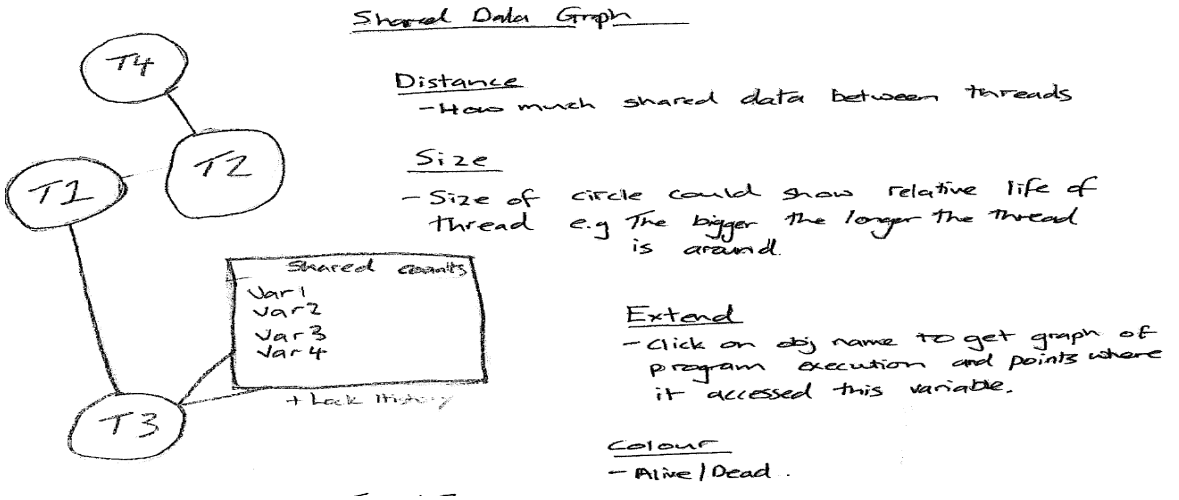


Figure 8.2: Shared data graph.

For the thread similarity graph, each thread is shown as a circle and the amount of shared data in common is indicated by the proximity of each circle to another. Scrolling over a circle will reveal other information such as a list of variables that the thread has touched, the names of monitor objects, and the names of other threads it has been waiting on. Another choice would be to have a

visualisation similar to the one below.

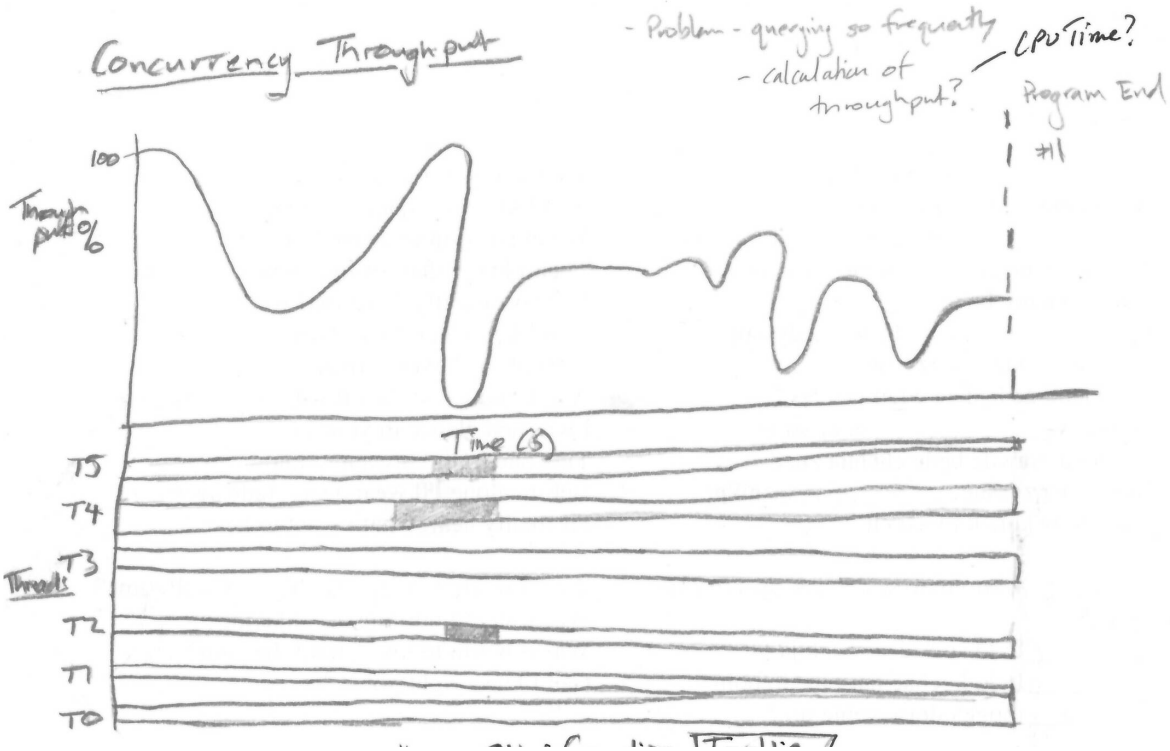


Figure 8.3: Initial visualisation design example.

This is essentially two graphs stacked on top of one another. The bars on the bottom part show thread states and information relating to those threads such as IDs. On the upper half, it shows the total throughput of all the threads taken as a whole. One example of calculating this value would be to look at the proportion of all threads that are doing useful work at that point in time. Each bar would then delineate the different states by using different colors. The next page shows two other versions of this same visualization that have superimposed one graph onto another. Throughput would be shown in the background while the main bar representing the threads would be in the foreground. Note that due to the fact that the throughput is 'behind' the threads, the bars would need to be opaque to some degree, see figure 8.4.

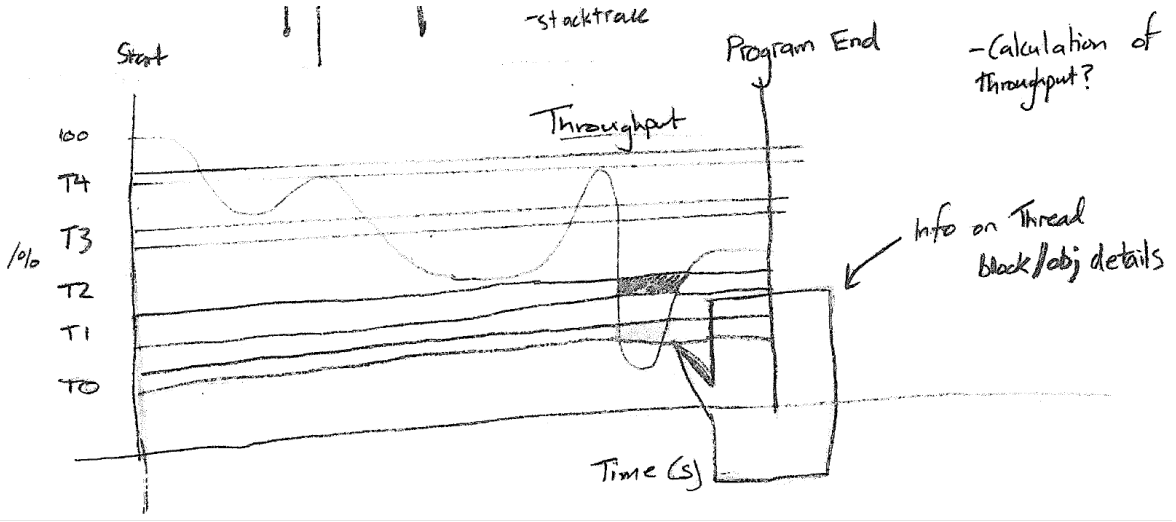


Figure 8.4: Visualization based on superimposed graphs.

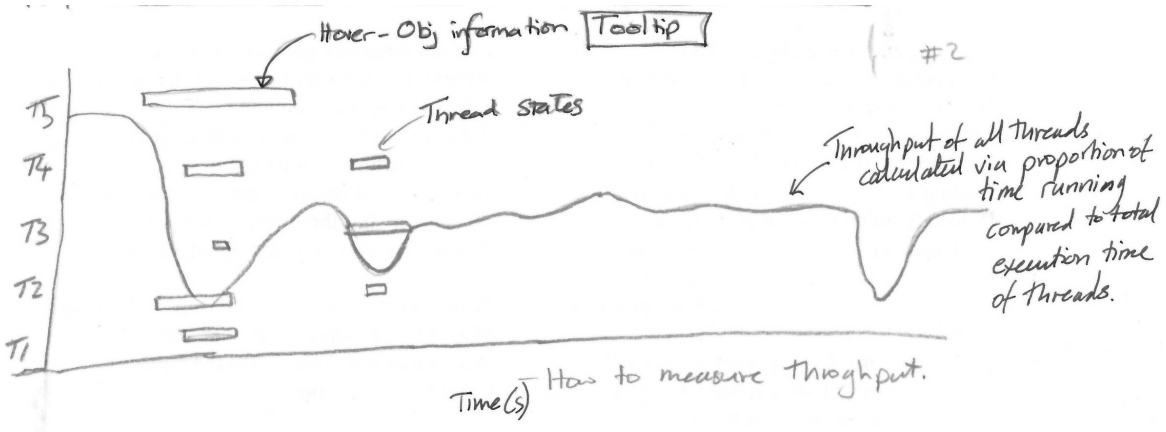


Figure 8.5: Alternative version of figure 8.4

In figure 8.5 the threads have been removed and only sections of interest are shown, for example, sections where threads are in a blocked state. In the previous visualisation, a large amount of information is shown about all the currently running threads and so it can be difficult to locate parts that are of interest. Filtering certain aspects such as threads in certain states can make it easier to locate the required information. An example usage of this would be if the user wanted to locate a possible deadlock scenario, then the bars will only be drawn when the threads are in a blocked or waiting state. Wherever two or more blocks are drawn at the same time, deadlock is a possibility.

Further designs centered around thread pools and the kinds of information that is considered to be useful when using a pool or a composite set of thread pools linked together. Details that could be relevant to thread pool interactions include:

- Queuing time - The amount of time a task spends waiting in a queue before being executed. Can be used to indicate a bottleneck in the queue or tasks are taking too long to execute within the pool. Both indicate some recalibration of the pool is required.

- Flow of tasks - A global view of the flow of information (as tasks) through a composite thread pool. This is useful to get an overall view of the pools and its particularly applicable to a composite pool setup.
- Pool efficiency - A display showing the proportion of the threads within the pool that are executing at any given time. This is applicable to a composite pool or to a single thread pool.

The drawings below give an idea of the sorts of visualisations available specifically for thread pools.

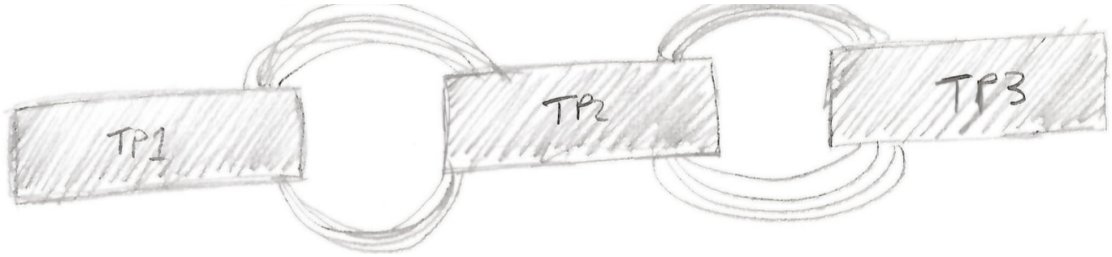


Figure 8.6: Flow of information between thread pools.

In figure 8.6 each pool is represented by a block of size that is determined by the number of initial threads within the pool. Once there has been some transfer of a task between the pools, a line is drawn between the source and destination pools. At the same time, if the receiver pool starts executing a task, one of the links is removed. By recording the number of completed tasks for the receiver pool, the total number of submitted tasks and the number of links which represents the task queue, the user is able to deduce the number of threads within the pool. The user may then wish to change the number of worker threads to reduce the task queue. It can also be combined with the general visualisation approaches to gain a greater understanding of the interactions of the working threads inside a particular pool.

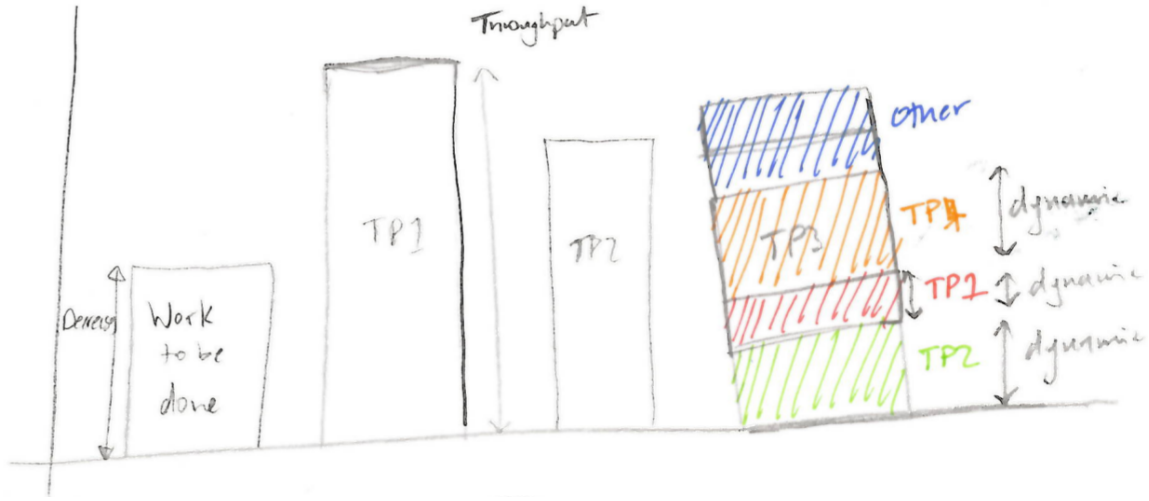


Figure 8.7: 'Work' view of the thread pools.

Figure 8.7 shows the amount of 'work' in the form of tasks that have been submitted to a queue for execution. The 'stacks' change dynamically in size showing the amount of queued work for a pool at

any one time. This is an alternative view of figure 8.6.

8.6 Gathering the data

So far only two proposals for collecting the data have been presented, one is using code instrumentation and the other being the JVMTI. A third way to do this is to create an API for users to insert calls into their own code, essentially creating a user library. While this last approach is easiest to implement it shifts the burden onto the user for inserting the calls. At the other end of the scale, byte code instrumentation requires no effort on part of the user but is difficult to implement effectively. Using the JVMTI is somewhere between the two, it allows the use of an API removing the need to do code instrumentation (for most information) but doesn't require the user to insert calls in to their own code. A related point to this is how to transfer the data from the back-end tool to the visualiser and there are a number of methods available including sockets, shared memory, writing and reading to a file and invoking a method directly on the visualiser, passing data in the parameters. For the sake of simplicity, the data will be transferred by writing and reading to a file, however, disadvantages to this method do exist. Without employing complex document layouts such JSON or XML the expressiveness of the file is limited to using punctuation or white space as a delimiter. Initially no complex data will be required and so there is no strong reason to use other transmission methods or markup documents such as XML.

Chapter 9

Initial Implementation

After the initial drawings for visualizations had been completed and a rough idea of the types of information had been established. Implementation began with a simple thread pool program that had been created while researching thread pools as a design pattern.

9.1 Java agent

The lack of thorough examples and good documentation about how to use the JVMTI required a lot of testing to see what things worked and what didn't. Therefore, it took quite a while to get an agent working with the information needed. Implementation of the agent was in C so direct access to the JVMTI is possible, there was no great advantage of using another language because it would still need to interface with a native API. It is compiled as a shared library and loaded either programmatically (shown later) or as a command-line parameter when invoking the JVM with the target program. Initially the target information was thread state information which is simple to implement assuming there was references to the threads..

The JVMTI has events for thread initialization and thread death and it is these capabilities that were first investigated. One of the advantages of using the call backs on these particular events is that they ignore JVM threads that are always present because the events are only generated after the JVM system threads have initialised. As it was later discovered, querying the JVM for all threads resulted in user threads as well as JVM threads being shown. Once a thread was created, a reference to the thread was saved via the event-triggered callback for thread creation. However, this resulted in problems due to the reference to thread going stale after the call back for the original event returned, crashing the JVM [Microsystems, 2002]. It became quickly apparent that this approach where references to all the threads are saved via the thread creation event, was not going to work.

An alternative was to call a function at the bottom of the entry function for the library that does the work required. To implement this, a function was created that loops and queries all threads' states, the update rate was controlled by suspending the thread initially for 2 seconds on each iteration. This approach worked and provided the information needed. Further investigation revealed the ability to create agent threads via the JVMTI, and for this to work, implementation of a helper function that constructs a Java thread object was required. This is important because the entry-points into the agent code should return once the appropriate settings have been set, as these are considered callbacks themselves. Another important point is that a large number of JVMTI functions are known as callback un-safe due to the way they traverse the heap inside the JVM. They cannot be used inside callback functions such as the one invoked on a thread initialization event. Creation of another agent

thread allowed these JVMTI functions to be used in a safe manner and does not require the use of the events.

There are two entry-points (functions) into the agent library and depending on how the library is loaded will dictate which entry-point to use. An important difference between these apart from the way the agent is loaded regards the amount of information that is available through the JVMTI. One of the purposes of the entry-points is to enable registration for events. If the agent is loaded at JVM start-up then all events are available for registration where as if the agent is loaded during runtime of the JVM, a subset is available. At this stage, it did not restrict the development but later on when events related to monitors were needed, it forced a change from attaching to a running process to loading the library when the target program is initially executed. This forces the user to start the agent when the target Java program is started. Difficulties resulted in the development of an agent taking a long time due to a lack of examples using the JVMTI API, and debugging was also difficult because it was a library and not an exectuable. Therefore, whenever it crashed the only information offered by the JVM would be its stack trace which included both Java and native functions making it difficult to pinpoint the problematic area.

9.2 Visualizer

Development of the visualizer was interleaved with development of the Java agent so once a basic agent was finished, focus shifted to the visualization part. Initially the plan was to leverage the Processing libraries to generate the required visualizations but after some time struggling to find solutions to draw a basic visualizer, and without resorting to manually drawing the shapes onto the screen, a new library was sought. This library came in the form of a utilities library called giCentreUtils that abstracted over Processing to allow the creation of various types of graphical data representations including pie charts and line graphs. This allowed the creation of a simple bar graph to show the thread state information. However it later turned out that the library was not flexible enough therefore this this option was abandoned in favour of a manual approach.

9.2.1 GUI Improvements

So far the implementation has forced the user to start the target program with the agent and not permitted them to attach to an already running process. This then influenced the decision to incorporate the ability for the agent to be attached to an already running Java program. Testing the agent then became more flexible because it could either be loaded at runtime or into any currently running Java process. The implementation for the loadtime entry-point was then copied into the runtime entry-point function, giving the agent the same capabilities regardless of how it was loaded. To realize the attachment mechanism on the Java front-end, the Java Attach API was required [Zukowski, 2007]. The purpose of the API is to allow for programmatic access via Java to other running Java virtual machines, in particular it gives functionality to list all executing JVMs as well as the ability to load dynamic libraries into them. This is what happens when the attach menu is selected from the Java GUI, it lists the current JVMs' names in terms of the command executed to start the JVM. It created problems however when the command was extremely long (as is the case when eclipse is running) so to deal with this, a limit of 50 characters was imposed after which the string was truncated.

This proved successful and it was later extended to allow for the Java front-end to initiate the whole process by launching the target Java program and injecting the Java agent into this JVM. The user

now has a choice whether to load during execution of the target program or before. Two important Java API's were used here, one that deals with creating processes called ProcessBuilder and the other that implements a simple way to search the file system. ProcessBuilder allows the execution of a command and arguments as if the user were typing it into a shell, example code below shows the method invocation.

```
ProcessBuilder pb = new ProcessBuilder("java" , "-agentpath:" + homeDir +
    AGENTPATH, fileWithoutEx);
//homeDir - Home directory of user
//AGENTPATH - Full path from user home directory
//fileWithoutEx - Target Java class for execution
```

Implementation of this is simple but could be improved because it relies on hard coded paths and the library being in a specific place.

An unpleasant side-effect of having the visualizer control the initiation of the the Java agent and target program is that is introduces implicit timing restrictions. In this case, the visualizer must not attempt to draw any visualizations hence request data, until the target program has started running and Java agent loaded. This was enforced by simply ordering the method invocations correctly so the sketch **cannot** start drawing until the methods that execute the Java program and load the agent, have returned. Although there is no guarantee that if something were to interrupt the process from starting but not notify the GUI then this could cause a fatal error. It is important to note that at this point, there was only one visualisation, and so ordering of method calls was enough to prevent the visualizer drawing before the agent was loaded. In later implementations a shared object is needed to facilitate signalling.

9.2.2 Data Visualisation

Implementation of the data retrieval is done by repeatedly calling functions that query for information. These function calls were wrapped in a while loop that also contained code that forced the thread to sleep for 2 seconds, slowing down the polling. Results of these calls are written to a file with a specific format, in this case it was simply the thread name followed by the state as an integer with a tab separating them. Using a tab between them made it easier to tokenize the string when reading line by line from the file. The visualizer is then tasked with reading this file and drawing a visualization to represent the thread states. Drawing the first visualisation manually involved drawing the shapes and lines onto the sketch by using a coordinate system. Within the visualiser code itself, an array of JVMThread objects that represent each thread are kept. Each object corresponds to one currently executing thread, storing information related to that thread and providing the link between the thread state and the position on the canvas where it is drawn.

The thread state graph is drawn by creating a rectangle for each thread and state, using colour to show the range of states. As long as a thread stays in a particular state the block drawn onto the graph is extended (literally) periodically after each update showing the whole history of the thread from a certain point. Within each JVMThread object, a stack keeps track of a Block objects that wrap coordinate information for each of the blocks, allowing them to be drawn. The reasons as to why a stack is used stems from the fact that the order that the blocks were created needs to be preserved and using a stack enforces this. An example would be a thread that has changed state twice over a 10 interval period going from n-10 up until time n. If it changes at interval 5, then between intervals 0 and 5 inclusive, a block will represent that time period with a certain colour. After this time period, a new block is created with a different colour to represent the different state that the thread is in. This is explained further in a diagram below:

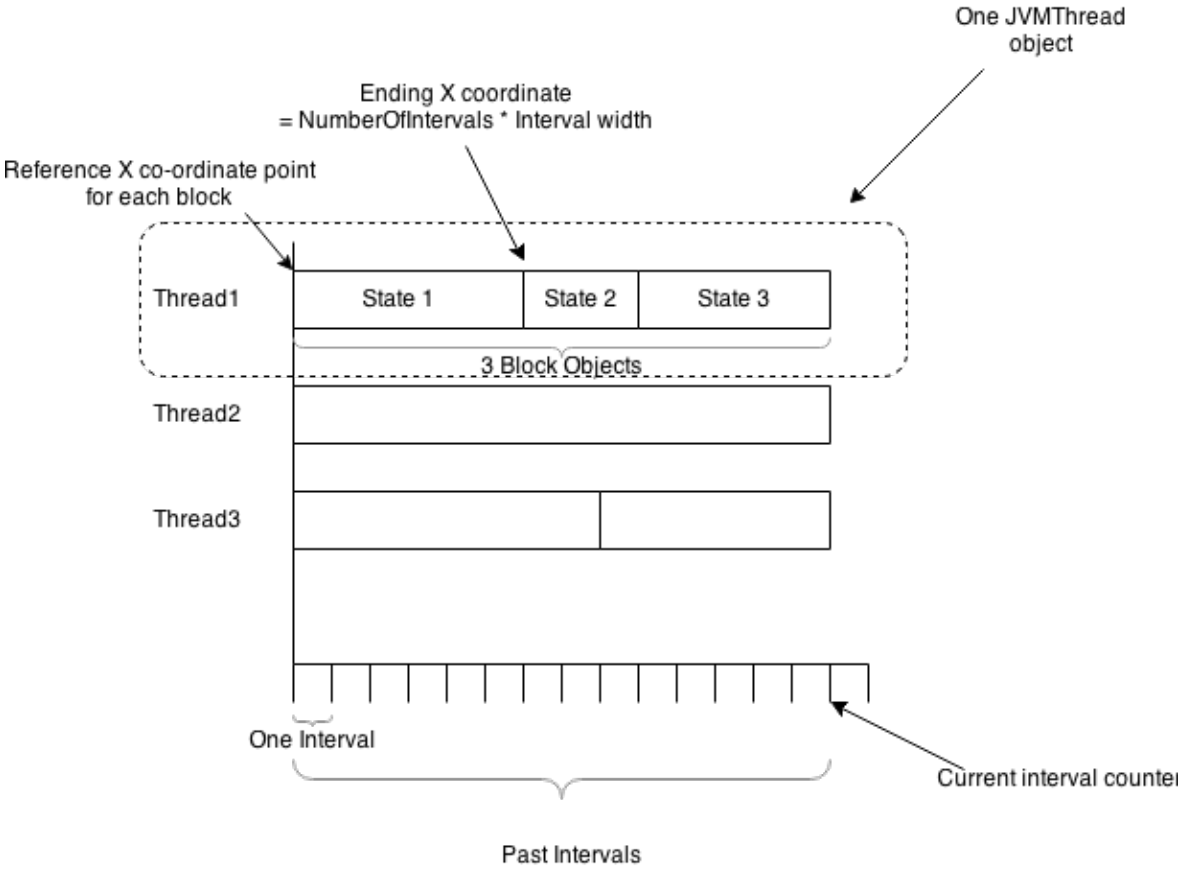


Figure 9.1: Representation of how the blocks are drawn.

Here the co-ordinates of interest are labeled and it indicates how the blocks are drawn for each thread. Each JVMThread object is looped over and their respective stack of Block objects is examined, permitting the blocks to be drawn for each thread sequentially. The update rate of the visualiser is controlled by the frame rate, which initially had an update rate of 2 times a second. This is much faster than the rate at which the agent updates so there is no possibility of the visualiser missing information from the agent. The sequence of events for the visualiser is as follows:

1. Update current states and create a new block if necessary (i.e. if the state has changed since the last interval.).
2. Draw axis.
3. Loop over each JVMThread object, inner loop over each Block object in stack to draw it.

A screen-shot of the visualization working is shown below.

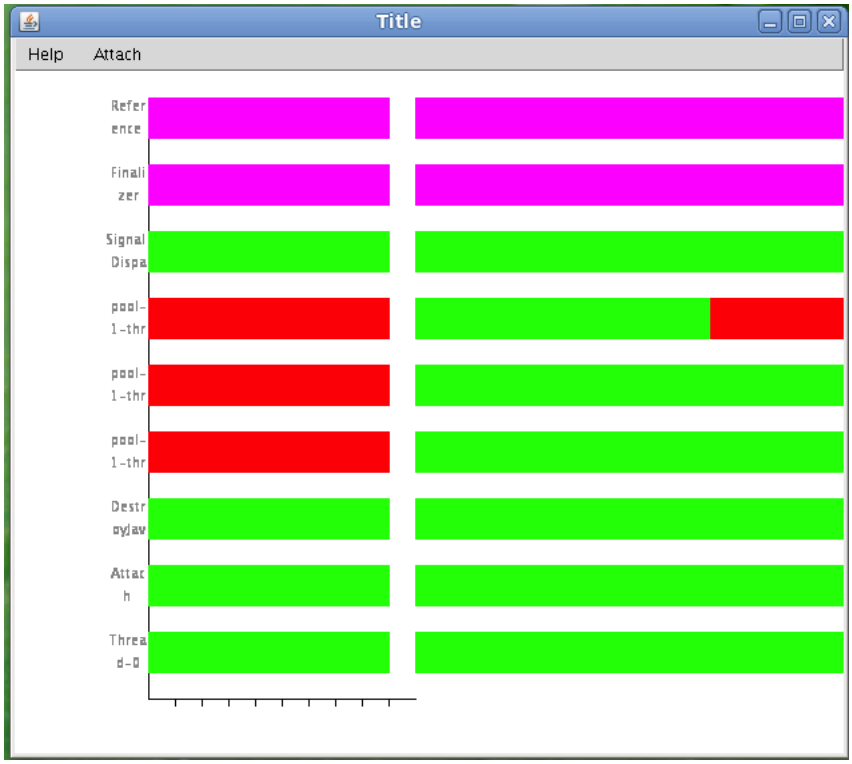


Figure 9.2: Initial thread state visualizer. All present threads are shown a long the y-axis and the history of the thread is shown as the coloured blocks, increasing in time.

9.2.3 Testing

To make sure the visualizer was functioning correctly, output from the agent regarding the states of threads was checked against what the visualizer was showing to confirm they had the same information.

Chapter 10

Second Design Iteration

A lot of experience had been gained from the first iteration of development and design. A much clearer view of what the JVMTI was capable of gave some ideas as to what could be displayed for the second visualisation. The focus for the second general thread visualisation was on efficiency. This can be useful when trying to get a general overview of the threading in the application and may help highlight areas where threads are making little progress. It could also be used as a verification tool by comparing results against expected behaviour. To implement this, a metric can be used to derive how 'useful' a thread is and the approach that will be used looks at how much CPU time the thread has had. Calculation of the proportion of time that a thread has had CPU time would be an example of one of these metrics. This is possible by dividing the amount of time a thread gets on the CPU by the duration of the process up to that point. The JVMTI has functions that allow the capture of time stamps so these can be used to calculate the total time. A different approach could look at the amount of time the thread has spent waiting or blocked.

The type of data that we get from this visualisation lends itself to something like a bar graph or line graph. Keeping in mind that this is a dynamic tool and the emphasis should be on the differences between the threads, some sort of chart would be most appropriate. This explicitly shows the direct value-to-value comparison between each of the threads.

Incorporating a second visualisation into the first one would be done by 'sharing' the sketch and calling a separate method to implement the drawing logic. Keyboard listeners could then be used to switch between the visualisations by changing an object reference, with each object representing one type of visualisation. It would make sense here to create an abstract base class that provides partial implementations of the *draw()* and *setup()* methods required by Processing. A class is created then for each view that extends the abstract class and the implementations would override the *draw()* and *setup()* methods. The reference switch would then cause the reference to point to an object that contains drawing logic for a different visualisation. This makes the code more modular so that further visualisations can also be easily added and reduces the burden on resources by only using one base Processing sketch. The disadvantage of this is that we cannot draw them at the same time, something like this would require each visualisation to draw into its own Processing sketch.

Chapter 11

Second Implementation Iteration

To realise the design for the second visualisation, the biggest modifications needed to be carried out on the visualiser so that we are able to switch through the different visualisations. The agent was first extended to get the required timing information via JVMTI functions. Execution followed exactly the same algorithm as the one in the first visualisation, show below:

- Get all current threads connected in the JVM (excluding native ones).
- Get CPU times for each of the threads.
- Calculate proportion of CPU time compared to total program running time and print out to file.

Because we are dealing with time here the data types used to represent the values need to be carefully checked so that they don't have a chance of over-flowing. The data types that the functions return are unsigned long integers that are able to record a time span of 584 years, in nanoseconds. To calculate the total time of program execution, a time stamp that is created when the agent is loaded is saved, in addition to a value of the current time stamp.

Each thread had an associated Java object representing it and storing the data from the Java agent. A group of simple loops and an array list was enough to implement this visualisation. This is mostly due to the fact that a library is used to draw the main component of this visualisation. The size of the data that is being received from the agent is not of sufficient magnitude where faster search algorithms would need to be employed.

With respect to drawing the graph, the giCentreUtils library that was abandoned in the first visualisation is used. This allows the creation of the bar graph by calling a few methods to setup data values, taking far less time to implement versus manually implementing the drawing logic.

This current version does not deal with all cases however, a program that contains threads which initiate some time after the initial time stamp being created will have a skewed result. The implementation is based on the assumption that all the threads within the program have been running since the start of the program. A fix for this is trivial but has not been implemented yet. In addition, the original implementation didn't deal with threads that died during execution. This has since been fixed by implementing a simple cleaning function that removes threads if they have not been 'seen' in the next update. All threads that are alive should have some sort of value from the agent, therefore any threads that do not appear must have died between the last update and the present.

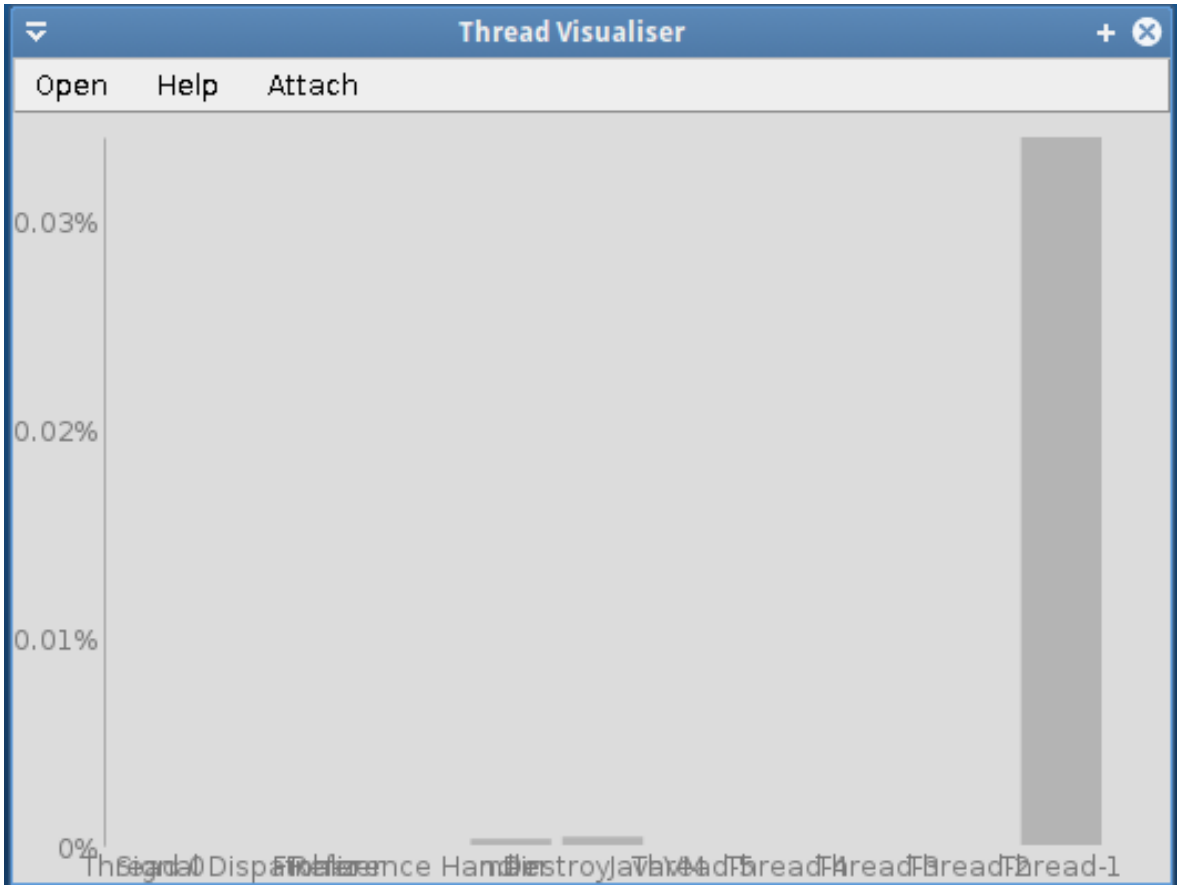


Figure 11.1: This is the result of this implementation cycle, with each bar representing a thread. The height of the bar represents how much execution time the thread has had proportional to the total program execution time.

Chapter 12

Third Design Iteration

The third visualisation will revolve around the synchronization primitives of Java. The interactions between the threads with synchronization primitives is an important part of debugging concurrent programs. So it is helpful for programmers to see at which times threads are obtaining monitors and show which threads are waiting on these monitor locks. From this the user can deduce areas of deadlock and explicitly see where there is lock contention. Information that is displayed in this visualisation can be seen in other tools but is not shown graphically.

The idea is to show each thread as a vertex in an N-sided polygon with the number of sides dependent on the number of threads present. Once a thread obtains a monitor lock, a ring will appear around the vertex after which any threads that are waiting on the same monitor lock will have a line drawn from their vertex to the vertex representing the thread holding the lock.

Already implemented functions inside the Java agent to fetch information regarding thread states can be extended to accommodate this visualisation. Similar to the previous visualization methods, the data will be stored inside the visualiser in a similar way to both previous implementation cycles, except this time there will be an emphasis on modular design. The thread information, polygon information and the monitor information can be separated into their respective sections but must all be related to each other to allow quick access between different sets of information.

Chapter 13

Third Implementation Iteration

After all existing threads in the JVM are requested a JVMTI function can query the thread to check for any monitor locks that the thread may own. If any are found, then this monitor object itself is queried via another JVMTI function that returns references to all the threads waiting on this monitor lock. Due to the fact that it was implemented inside the original looping code that gathered data regarding thread states and CPU time information, the resolution and accuracy of this visualisation is bound to the rate at which the state information is produced. But there is no real reason why either of these should be faster than the other, ideally they should both be as fast as possible. This information is output into a file where each line contains a list of thread names, the first is the name of thread that holds the monitor lock and the rest of the names are threads that are waiting to gain entry into the monitor.

Much of the visualiser was implemented by re-using some of the classes from the other visualisations, especially JVMThread which encapsulates information about a thread. Instead of using an array, this time a hash table was used with a hashed form of the thread's name as a key, the default string hash function is used here and it is assumed that all thread names are unique. A hash table is advantageous over an array because it allows instant lookup but with arbitrary keys, and this operation is quite frequent. Two important flags within the object were added to show when a thread is holding a monitor object and also to mark the thread as seen. Every time a new set of thread data is read in, all current threads have their states updated and are marked in this process. After the global update has occurred, any threads that are not marked as seen are removed, ensuring that the data is consistent with the data being output by the Java agent. Monitor information is dealt with inside the visualiser by its own object that uses the Singleton design pattern. Within this class is another hash table that stores a mapping between waiting threads and the thread owning the monitor lock. On every update, if a thread owns a monitor, a reference is set inside its object that points to a list containing the names of the waiting threads. However the implementation of the monitor object makes this action redundant because there is already a mapping inside the monitor that can facilitate access to this information. The reason it is present is because the current implementation cannot handle situations where a thread may own more than one monitor lock. Keeping a separate mapping for each monitor is a lot more cleaner than having the reference to a complete list of waiting threads.

Information relating to the drawing of the vertices on the screen was wrapped in a class called Polygon and contained a reference to its corresponding thread via a local variable that stores a key. This made it easy given a certain polygon, to query information related to the thread, such as whether it held a monitor lock or not. By abstracting the coordinate information out it reduces the amount of specialisation that the JVMThread class has, increasing is re-usability potential. To draw

the visualisation, only access to the polygon objects is needed.

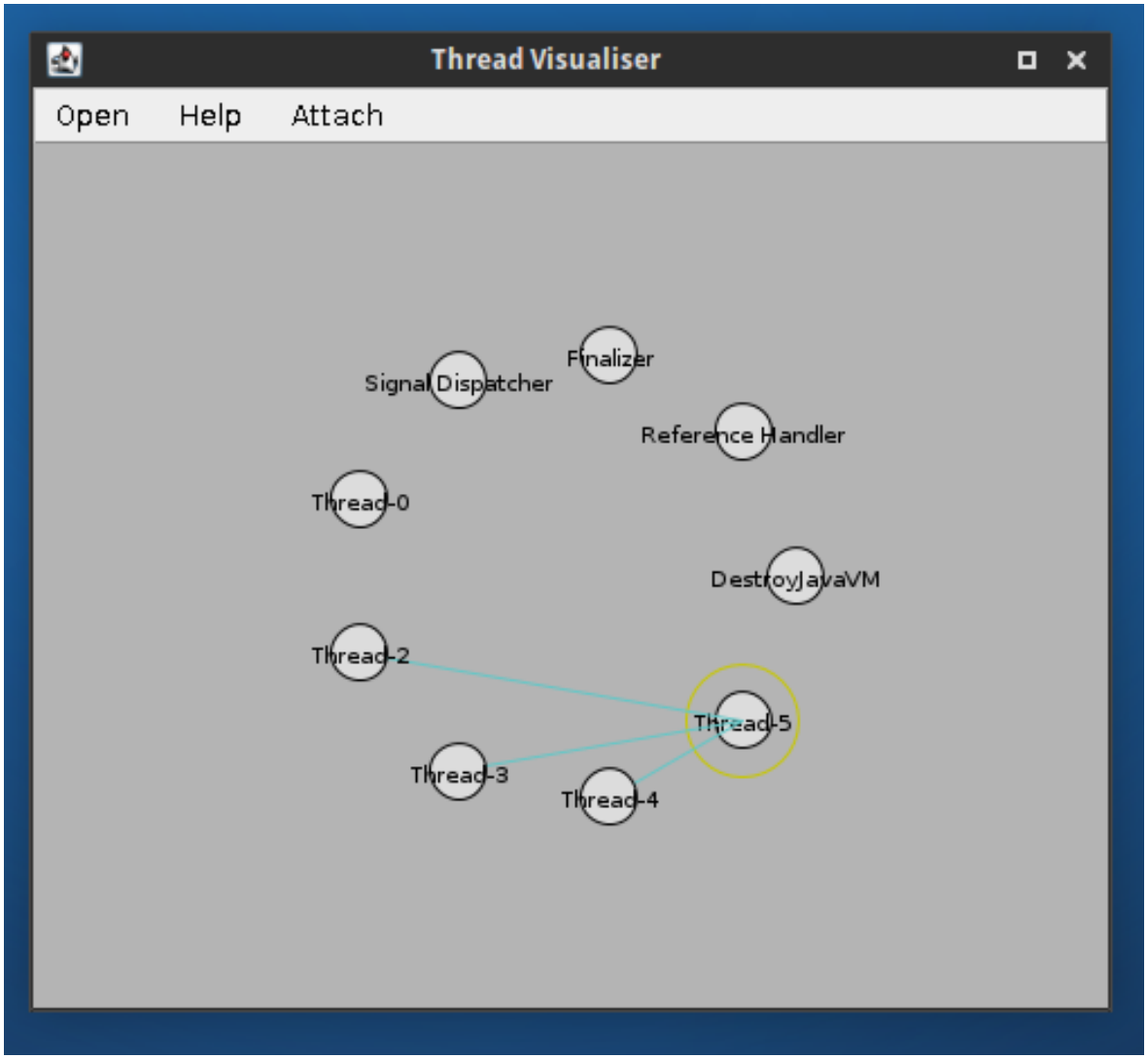


Figure 13.1: Third general visualisation showing thread locks and waiting threads. The thread owning the monitor is highlighted in yellow while the threads blocked on this monitor have blue lines linking them to the thread that owns the monitor.

Chapter 14

Thread Pool Visualisation

Now that a better understanding of what is possible has been established, the focus of the project will now shift towards thread pool specific visualisations. Using a composite thread pool as the example to show the kinds of visualisations possible, extra information can be leveraged to visualise in new ways.

14.1 First Design Iteration

The thread pool pattern itself adds an extra layer of complexity on top of already complicated thread interactions. These are exacerbated when we try to form composite thread pools and work out the complex interactions between each pool. When looking at thread pools, factors such as efficiency, throughput and information flow are important to understanding the interactions. And for this reason, the visualisations will focus on these areas. A user can then deduce how effective a thread pool is and tweak it so that the performance is maximal. Bottlenecks can also be easily detected via a visualisation and the general flow of tasks can be made explicit. Showing a deadlocking pool would also be simple but is not implemented here. To visualise a composite pool, there needs to be an implementation of a composite pool itself before there can be any analysis on it; the next section details this.

14.1.1 Design of a composite thread pool

Java's standard implementation comes in the form of a class from the Executor framework which will be extended to add some further functionality, this includes the ability to 'pass-on' a task once it has completed execution to the next pool in the chain. Each pool will execute the same task, although in a real implementation each pool may perform a different task, but for the purposes of the research here this is enough. A test harness will then be constructed in order to automate the construction of such a composite thread pool, making it easy to change certain parameters such as the number of pools in the chain and the number of threads given to each pool.

14.1.2 Data extraction from the composite pool

The Executor's framework provides a range of thread pool implementations but because it is implemented in Java, no specific information relating to them is available in the JVMTI. As it turns out the information available about Executor instances in Java offers a lot of information that could be used in visualisation. The agent was initially going to be re-targeted for gathering the required data but the JVMTI did not provision this. The starting idea was to try to 'find' the thread pool objects that had been instantiated by using certain JVMTI functions in the following ways:

- Method hooking can be used to fire events every time a method is invoked. The method names are known and so this can be an easy way to find the object. It incurs a heavy overhead because all methods would create an event.
- Intercepting object allocation permits analysis of objects which could then be inspected to see if it was the object we're looking for. Again this is a very invasive approach to finding a specific type of instance.

Further research lead to the conclusion that there was no feasible approach using the JVMTI. It was obvious that the JVMTI was going to offer little in this case and so it was decided that it was easiest to access the objects directly at the Java language level and get the information in this fashion. Methods on the executor objects can offer information including:

- Total number of submitted tasks for execution in that pool.
- Total number of tasks that have completed execution.
- Total number of threads in the pool.
- Total number of threads actively executing.
- The largest pool size (when an adaptive pool is used).

Using this approach the visualiser will be *moulded* around the composite thread pool.

14.1.3 First visualisation design

Visually seeing how the information travels through a chained thread pool gives the user a way to reason about global properties of the composite pool. This led to the idea of visualising the tasks as they went through the pools, showing which pool they're in. In this design each task is shown as a square on a grid, the grid represents all the tasks that have been submitted to the composite pool. Each pool is then labelled with a specific colour which is then used to colour code the square, demonstrating which pool the task is in. As the pool progresses executing the tasks, the colours of the squares will change indicating progression through the composite pool.

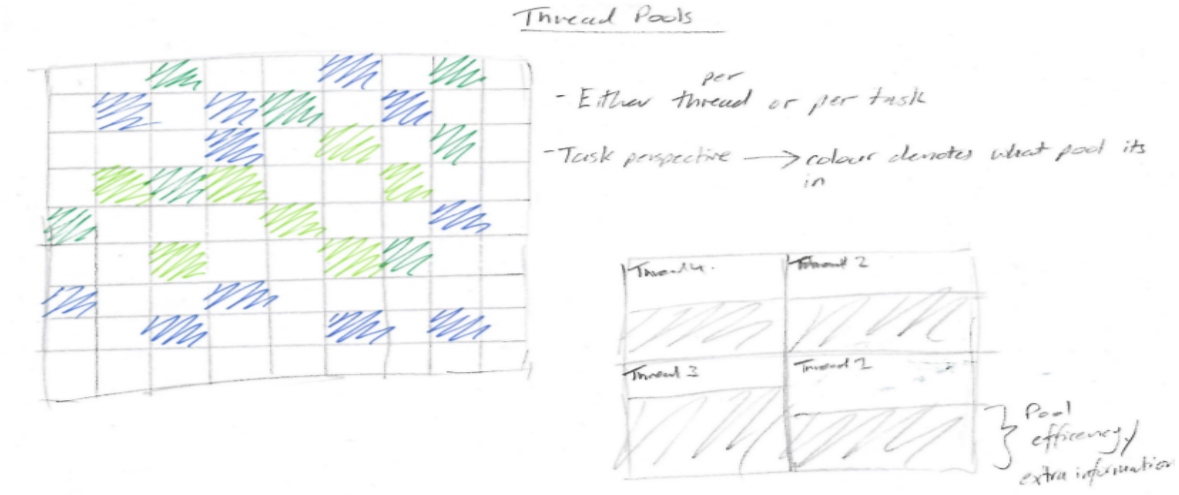


Figure 14.1: Sketch of visualisation idea.

By viewing the colour of the tiles as the visualisation is running an indication is given to the user where likely bottlenecks are occurring. Tiles that stay a particular colour for a prolonged period of time could be due to a lack of worker threads in the pool but it is not definitive because this situation also occurs if a task takes a particularly long time to execute. This allows the user to analyse the tasks involved in this pool and modify them accordingly. The only exception is for the first pool because all tasks are submitted at the same time in the example programs.

14.2 First Implementation Iteration

The details of the implementation of the first thread pool specific visualisation and the composite thread pool are shown here.

14.2.1 Composite thread pool

To be able to access each pool individually there needs to be some addressing scheme used so each pool will receive a unique ID. References to each pool are kept in an array with the index number being the pools unique ID, forming a mapping. To allow a task to be 'passed' from one pool to the next, the pool had to maintain a reference of the next pool, enabling it queue the task for execution. The standard implementation classes are extended and then wrapped in another class containing a reference to the next pool in the chain, the advantage of using a wrapper is that we're able to change the underlying thread pool implementation without having to change much of the way it interacts with the outside logic. Most of the extra information needed for the thread pool was actually added to the wrapper and not the extended class, access to particular methods was also required so extension of an Executor class is required.

Methods that enable interception of a task both before and after it has executed in a pool are utilized. Interception of the task after it has been executed facilitates the task being passed to the next pool. Method invocations for these interceptions are passed up to the wrapper class and the core implementations of these methods are there, instead of inside the extending class. The advantage of this is that specialisation is kept out of the underlying thread pool class making it easier if specialized thread pool is required, it only entails replacing the underlying class and passing method calls up. Orderly shut-down of the thread pool is required by the standard implementation otherwise the pool will wait for tasks indefinitely. To implement this, a check is performed on the number of tasks it has executed and if it matches or exceeds the number of tasks submitted overall then it calls the shutdown method on the thread pool.
Each thread pool is instantiated by creating its wrapper class and then calling a method which sets up an instance of the thread pool inside the wrapper, this is to prevent references to the wrapper object escaping before its constructor has completed. The various parameters for the executor such as the number of threads it uses, the timeout threshold for a thread and the type of queue it uses, are all passed to the underlying thread pool via its constructor, much of which is unchangeable from the wrapper interface. Changing these parameters are not of interest but are required in-case modification of the thread pool is desired. Only the number of threads in the underlying pool is of interest so this can be set via parameters to the wrapper constructor.

14.2.2 Visualiser

The information that needs to be known for this visualisation include, the number of tasks, the time at which a task is submitted for execution, the unique pool id, the unique id of the task and the co-ordinates of the square within the sketch space. Each task id is chosen by using the position at which

that task is created within the group of tasks, it is simple and provides a guarantee that no task will have duplicate ids. A task class is created to encapsulate all information relating to a task including its id. Task objects are then stored in a thread-safe map to avoid inconsistencies when updating the task's pool id. Thread-safety ensures that any any atomic operation on the map is thread-safe, in-case access synchronization is not enforced explicitly. The key for each task object in the map is its id value, granting instant access to update its pool id each time it transfers between pools. Colours are mapped to each pool using its pool id, set when the visualiser starts. Coordinates of where the square will by drawn is set when the task is first added to the synchronized map, and not changed.

Calculation of the tile size is done by taking the number of tasks and then finding the next biggest perfect square. By doing this, it keeps the proportions of the grid with respect to the number of rows and columns equal for better visual proportions. Each square is then scaled to its correct proportion by dividing the width and length of the sketch by the number of rows and columns respectively. This does however result in some space at the bottom of the sketch when the number of tasks is not a perfect square but this a minor aesthetic problem.

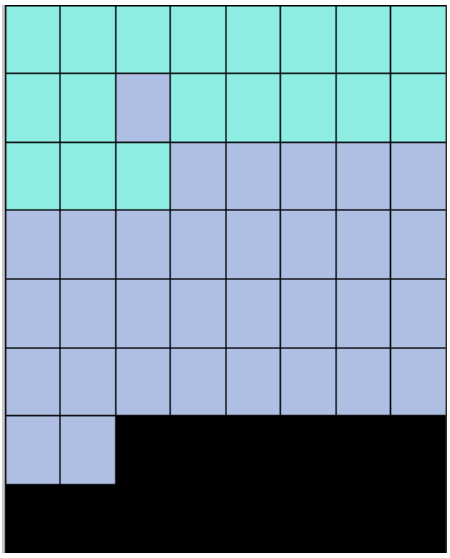


Figure 14.2: Grid of tasks from the first visualisation, each colour shows which pool the task is currently in.

Initially the the backing data structure was polled at a rate set by the frame rate of the sketch, by default this is 60 frames per second. This induced a rather high lock contention on the backing data structure so to resolve this the frame rate was lowered to 30 frames per second. The only disadvantage of the current implementation is that it uses a callback to send updates to the visualiser, holding up the task while this is done. This may affect the ordering of the tasks as they are submitted to the next pool but the delay is not significant enough to affect the overall behaviour of the composite pool. An important point to note here is that the update method that is called as part of the interception method in the pools must be ordered correctly with respect to passing the task on. The update to the visualiser must be issued before the task is passed on otherwise a case can arise where the task has completed the pool before the previous update to the visualiser has been written in to the backing data structure, resulting in inaccurate data.

14.3 Second Design Iteration

One of the deficiencies of the first visualisation is that the user cannot tell whether a task is actually being executed in that pool or waiting in a queue for a thread to become available. This problem will form the basis of the second visualisation which will explicitly show the number of tasks that have been submitted but are awaiting execution by the pool. By incorporating information regarding the active number of threads in the pool as well as the queue of jobs for the pool, the efficiency of the pool can be concluded. Combining these sets of data for each pool, the user can get an overall idea of how efficient the composite thread pool is.

To show this information, the user will be able to click on the sketch and select a position to a draw circle representing the pool. Edges will then form between circles once tasks have been submitted for execution to that pool. Each task submitted will create an arc to the next pool and from this it will be easy to see the number of tasks that are waiting to be executed. The data needed for this visualisation will be the same as the last visualisation in addition to information regarding the number of completed tasks and active worker thread counts. These values are found by querying the thread pool objects directly so the wrapper must support a method to return the underlying thread pool instance that is contains.

In addition, the visualisation will be embedded into a frame that contains the first view, making comparisons easy. The inheritance approach used in the general visualiser cannot be used here because both views require their own independent sketches, they cannot 'share' one. The next figure shows a design drawing of the visualisation.

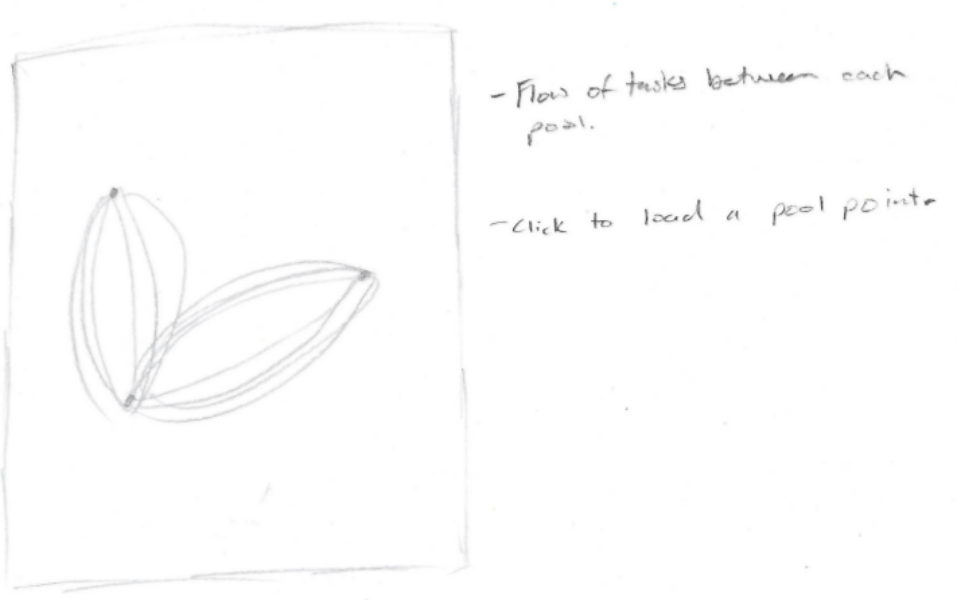


Figure 14.3: Sketch of second visualisation approach for thread pools.

14.4 Second Implementation Iteration

To determine the number of arcs the individual thread pools can be queried to get the total number of tasks submitted and the total number of tasks that have completed execution. By subtracting

the number of tasks completed as well as the number of active threads, the left over amount is the number of tasks that are still queued. Values obtained via querying are not exact and are only approximates, because of the high concurrency within one thread pool it is impossible to get exact values that are valid for a substantial period of time.

Each time the user clicks on the sketch, a circle will appear that represents one thread pool, this can be repeated until all thread pools have been accounted for. Processing provides useful abstractions for event handling in the form of callback functions, so it is easy to get coordinate information of where the user clicked on the sketch. Individual pools are represented by their own class and it contains information such as the coordinate of where its circle is on the screen, this class is different from the actual thread pool class.

Owing to the fact that the data is already contained within the underlying thread pool objects no auxiliary data structures are required to implement the main drawing logic. A simple loop over the PoolInfo objects exposes coordinate information for each pool which in combination with the thread pool object, is enough to draw the sketch.

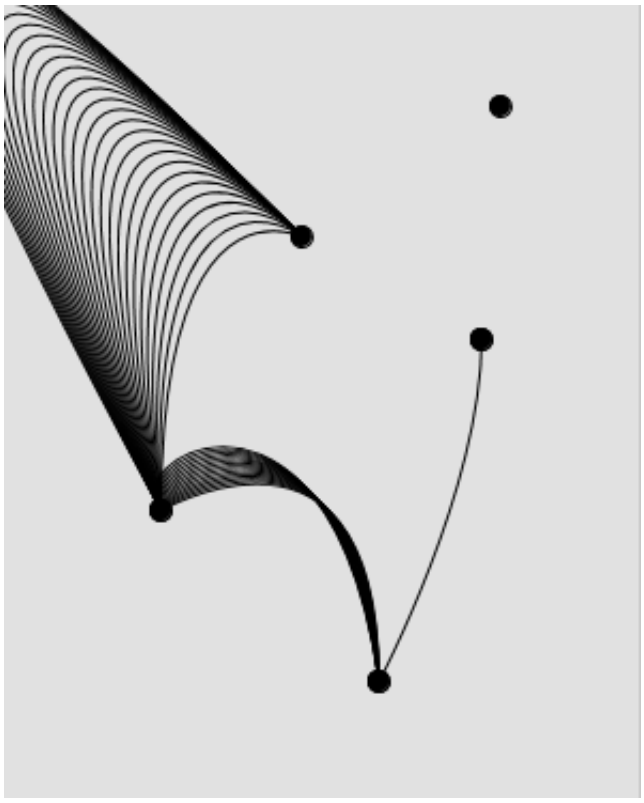


Figure 14.4: This is a screenshot from the tool showing the arcs between each thread pool. It is not immediately clear the direction of the edge, in later versions this is corrected.

Difficulty was encountered trying to implement the curves using the Processing API so due to time restraints an arrow with a number in the middle was used instead. It is less ‘visual‘ than using a curve per submitted task but the same information is available and the fact that arrows are used indicate the direction of flow through the composite pool is an additional advantage. The result of the change to arrows can be seen in the next figure.

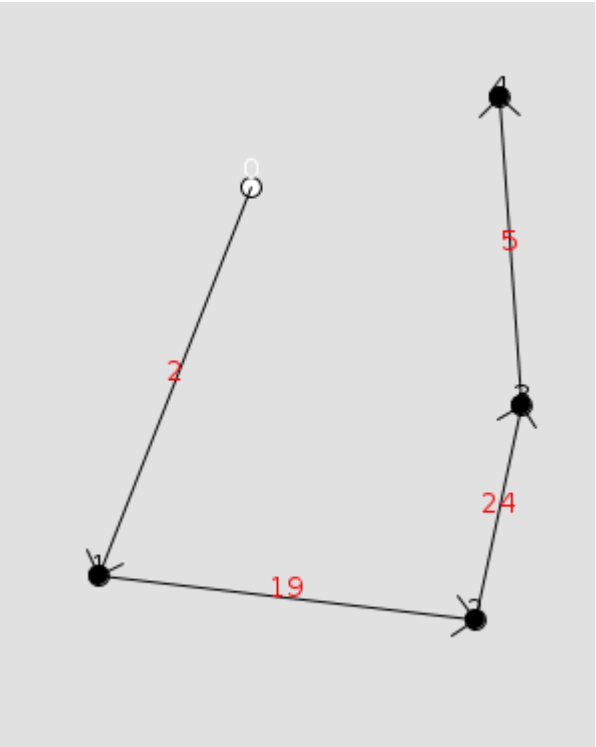


Figure 14.5: The total number of tasks submitted but not yet executed displayed as a number on top of the arrow.

Another advantage of using an arrow is that it scales well with many tasks, using curves imposes a limit before they start being drawn outside the frame or obscuring other curves. It is difficult to see on this screenshot because it is concealed by its colour and position, but each thread pool is also labelled with its ID. Both the sketches were then added to the same panel with a supplementary colour key for the first view displaying information about which colours are mapped to which thread pool.

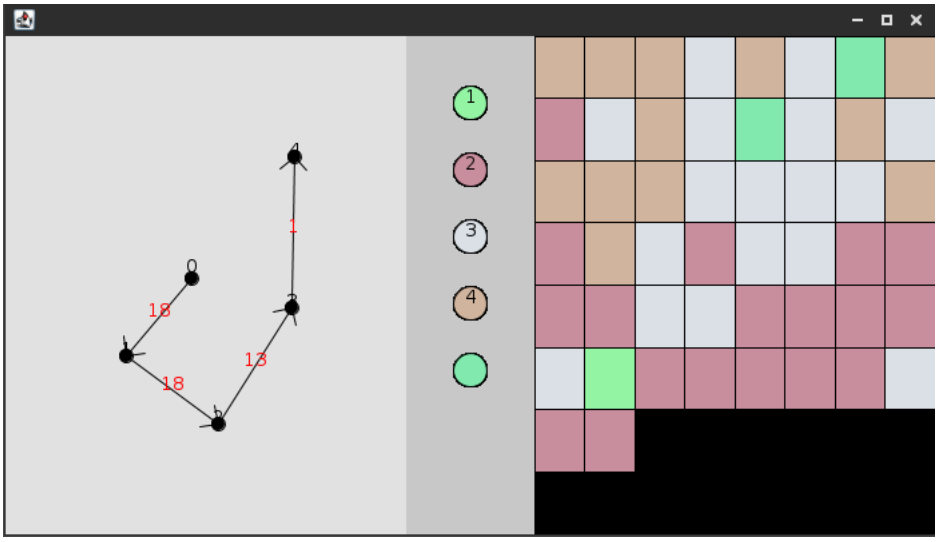


Figure 14.6: The final result of the thread pool visualisation, both views are seen here. Right is the 'task view' while the left is more of an individual thread pool view.

A strict ordering is imposed on the threads, without this the composite thread pool will start execution before the visualisers have initialised and are ready to receive information. To prevent this, each visualisations' monitor object is used as a lock to control the order instantiation. The main instantiating thread can then be controlled by waiting for notification from these visualiser objects. After the visualiser objects have reached the end of their initialisation methods, they can then inform the main thread that it is allowed to proceed. This stops the main thread issuing tasks before the visualiser objects are ready to start retrieving information.

Chapter 15

Enhancements

As a result of the development process certain sections of the program were often left unpolished. This section outlines development that took place at the end of the implementation with the aim of completing the user interfaces so.

15.1 General Visualiser

With respect to the first thread state visualiser, it is often the case that the chart will be drawn out of view of the window and the user would need to extend the window manually to see the current status of the threads. To make this more user friendly, an attempt was made to implement a scroll bar by using Swing components. The scroll bar component works by detecting when the component inside it gets bigger than the dimensions of the scroll bar's viewpoint. Once this happens the scroll bar fires an event and sets up a scroll bar with the correct parameters which then enables the user to drag a bar to scroll horizontally. However, it was not possible to get the scroll bar to detect this because drawing on the canvas does not increase the size of the component it sits in, and so the component itself does not change in size.

Modifications were also implemented to the second visualiser that shows CPU time of the threads. In the original implementation there was no code to remove dead threads and it would often display threads that had already died, sometimes leading to the y-axis becoming distorted. A simple marking algorithm is able to remove threads have not been seen on subsequent updates.

Functionality was also added in the third visualisation, this allowed the display of wait and timed wait thread states. In addition, a colour key was added to give meaning to the colours of the rings.

15.2 Agent

An error log was created for the agent so that debugging issues would be made easier. When the library is loaded, there is no way to differentiate between an error occurring inside the native agent or an exception being thrown inside the JVM due to some action the agent has taken. For this reason, a log file was created that records any errors from calls to JVMTI functions as well as errors inside the agent. This is done by re-directing stderr to a file.

15.2.1 General

After the final implementation iteration of the general visualiser, no code existed to cleanly close the program once the target program had died. This was then implemented using sockets to communicate between the visualiser process and the agent. Other methods of inter-process communication were looked at but due to the high-level nature of Java, many were out of reach, an example would be shared memory. Named and anonymous pipes could have been used but that increases the reliance of the tool on a particular platform. Sockets were chosen because I was familiar with them. However, as a result of using POSIX socket libraries it means that the agent has now become some what platform specific. The visualiser itself cannot force the agent to shut down as there is no platform-independent way of programmatically forcing the JVM to unload the agent, we can however implement a clean shut down when the target program dies. To implement this, the visualiser spawns a thread that listens on a TCP socket for any incoming connections. Once the target program dies, the JVMTI invokes a callback function that connects to the visualiser. The only disadvantage of this is that the visualiser is unable to get any information about the nature of the shut-down because the callback invoked in the agent is not able to receive information like return codes.

15.3 Thread Pool Visualiser

Additions were also added to the thread pool visualiser with more information being shown in the second view. New information such as the number of threads in each pool and the number of threads actively executing has also been added.

15.4 Summary

Here will be an overall summary of the tools and a look at what they can do. The first part will be about the first tool developed which is the general visualiser. It utilizes the agent to acquire data and consists of 3 views, a thread state view, CPU utilization view and a thread monitor/lock view.

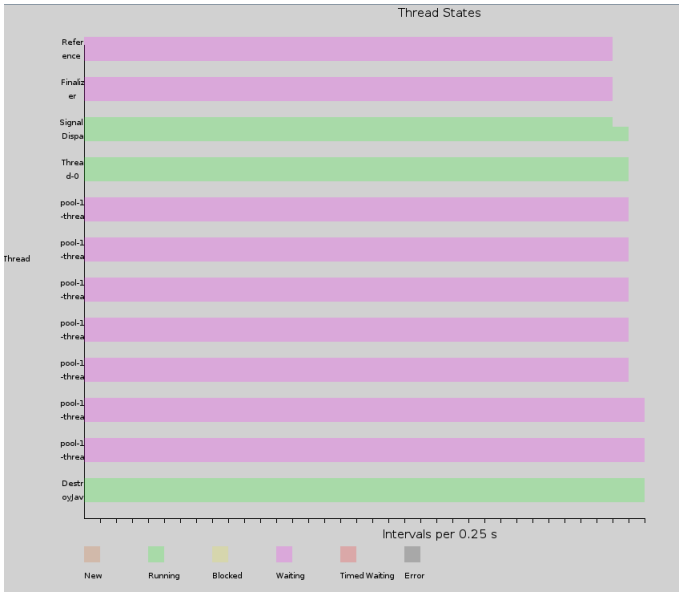


Figure 15.1: Thread state visualisation, the jagged edges are due to the slow rendering speed.

The view in figure 15.1 is very similar to the visualizations offered by tools seen in the context survey but the implemented tool here extends the information here by including all of the underlying Java thread states. The user is now able to to explicitly see when a thread is blocked on something, and it also gives the specific names for the thread states and doesn't assume a thread is sleeping when it enters the timed wait state which could be due to a range of reasons.

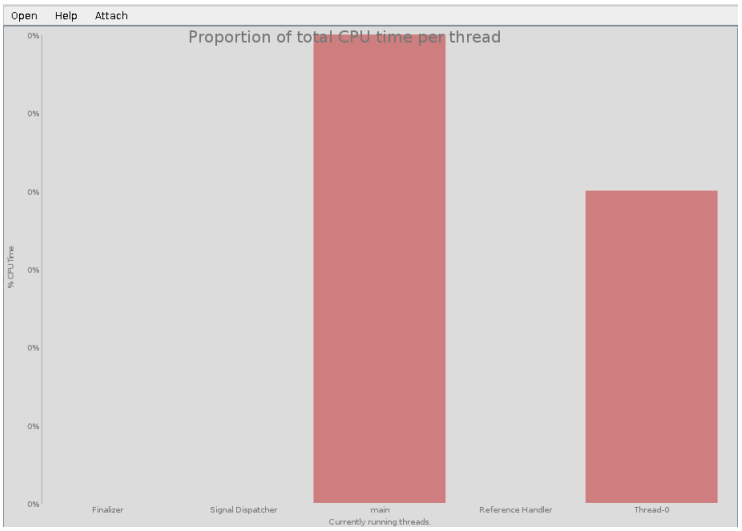


Figure 15.2: A version of CPU utilization proportional to the program under analysis only.

The second view seen in figure 15.2 is a form of showing the CPU utilization except its proportional to all the threads in the program. This offers greater insight into the threads than simply seeing the utilization of each CPU core as shown in some of the other tools. A user can then decide whether all threads are being used most efficiently or whether they need to restructure their program. This is not possible using the standard CPU utilization because this isn't specific to the process that is being analysed, it covers all processes.

In the final view seen in 15.4 there is a combination of data about current monitors, the current threads available and information relating to the threads blocked on other threads. This same information is available in some more complete suites in a textual format and the implementation extends on this by creating a full visualisation that makes this information even more obvious. From a usage perspective its a lot easier to see all the locks by using a visualisation because you can see all the locks and blocked threads in one graphic.

For thread pools two complimentary visualisations have been created that aid the user in understanding the concurrency interactions between them. All of the following visualisations are new and have not been seen in any of the analysed tools, mostly because it was specialized to the thread pool pattern. The first visualisation gives a 'task view' of the all the thread pools and the user can see how the tasks flow through the thread pools. Each task is given a square on a grid and coloured depending on which thread pool it is in, as they pass between pool the square changes colour. From this the user can gauge how efficient the current composite thread pool is, if the tiles happen to spend a lot of their time stuck on one colour compared to the rest of the colours there is a chance that there is a bottleneck. But there is still ambiguity because the task may just be computationally expensive, it cannot show when the task starts its execution. Dependencies and deadlock will show up as tiles that are stuck in a specific colour while all others have completed. Generally the tasks in

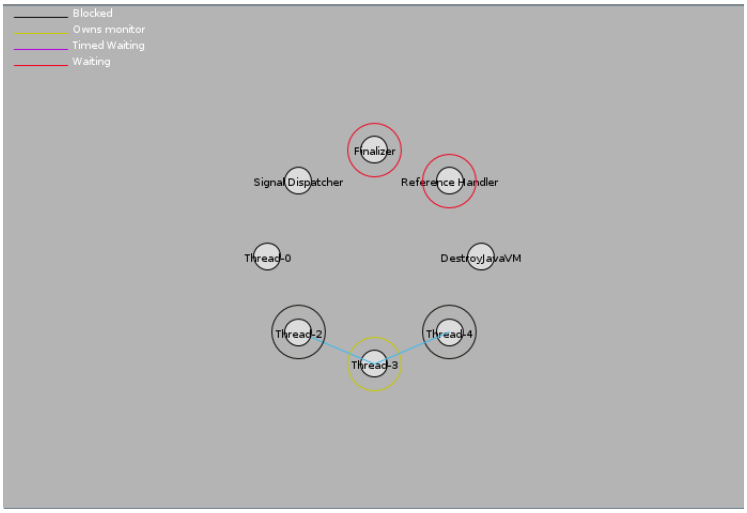


Figure 15.3: The third view available in the general visualiser, showing blocked threads and threads that own monitors.

one thread pool are all the same, so you could assume that two threads inside the same pool that have not changed colour while all the other tasks have completed are deadlocked.

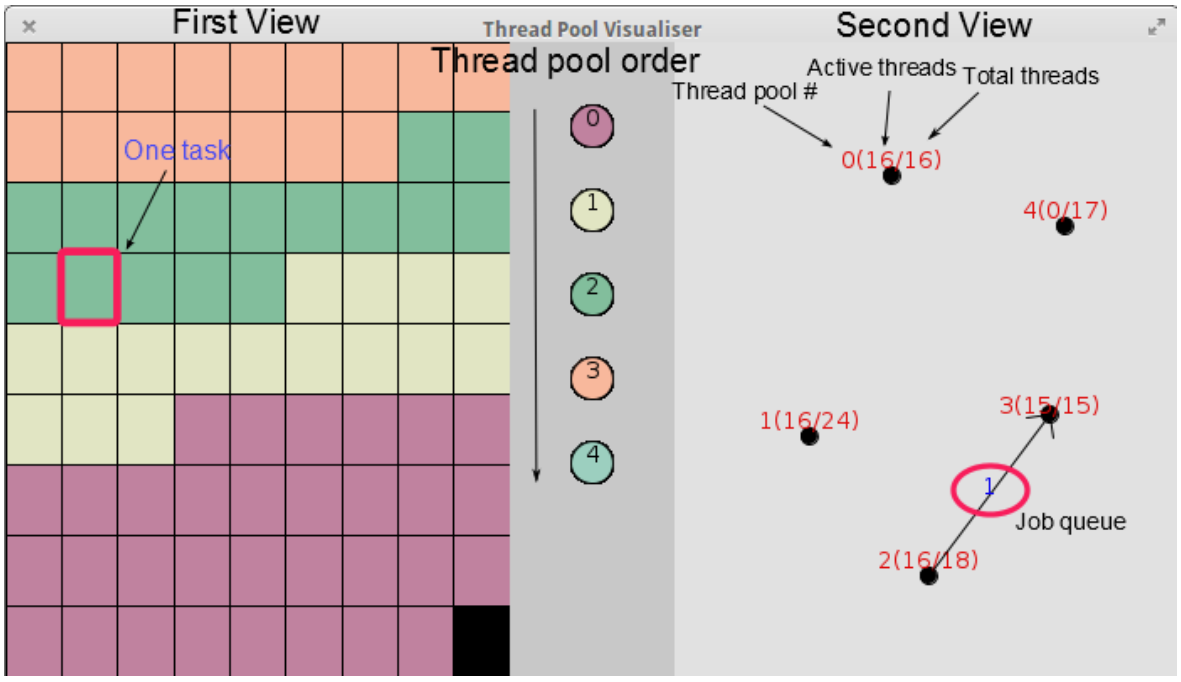


Figure 15.4: The final version of the thread pool visualiser. On the the left is the 'task' view and on the right is the second view. See labels for more information.

The second view of the thread pool visualizer aims to compliment the first by explicitly showing a job queue between thread pools and additional information on the activity of the worker threads. Using this view the user can inspect a specific thread pools' efficiency by monitoring the count

of active threads depicted as numbers above the circle representation of the thread pool seen in figure 15.4. If it never becomes saturated then there is room to reduce the number or there may be a bottleneck in an upstream pool. Bottlenecks are now made obvious by displaying the job queue for a pool, this is the number of tasks sent to the pool but awaiting execution. It is shown by an arrow between two circles representing the pools on the view and will contain a number denoting the number of waiting tasks. A composite pool that has only one arrow between two of the thread pools indicates to some extent, there is a bottleneck. Severity will depend on the length of time the arrow is around and the average number of tasks in the queue.

All the views in this visualisation have leveraged new information and created new visualisations that show information that could be shown in a textual form but it would require an efficient mechanism to show all the information at once.

Chapter 16

Refactoring

Because of the agile nature of the development, it often left the product unpolished and lacking some features. This section outlines what refactoring took place after the main development. A lot of the simple data structures used were not really that effective for their tasks and so many of these were changed to more appropriate data structure. An example of this occurred when refactoring the CPU time visualisation, an array list was originally used but as the implementation grew, appropriate integer keys began to become problematic. This was then replaced with a hashmap, which gives a lot more flexibility for keys.

16.1 Calculating Update Rate

Information gathered in the agent is controlled by a loop that repeatedly calls the required functions. To maximise the chances the visualiser detects a change, the exact time that the agent thread should sleep before calling the functions again needed to be investigated. Original implementation settings had the update at once per second, and while this is fast enough for most interactions, I wanted to see whether this could be optimized slightly. There is no point in making it microseconds faster but it would be advantageous to have it in the region of 0.25 - 1 second, this would increase the accuracy of the data shown in the visualiser. The goal here would be to increase the rate of updates while at the same time not exceeding the requested sleep period, with a lower bound of around 250 ms. Any faster and this would force the visualiser to render at too fast a rate. The scheduler enforces a lower bound on the times and even in the worst case, which is when we exceed our requested time period, there isn't any method to force the thread to be scheduled any earlier. This means that while some of the requested periods still exceed the amount, a majority of the requests will be on time. Within any fixed period, the proportion of on time requests will increase and hence the update rate increases overall. For this experiment the third test program, originally created to test the general visualiser was used and the settings for the agent were modified to output additional information to the console. By running the test program with a general load on the computer, we can see if the values stabilise around the requested value or whether they're generally higher. If they do stabilise it will indicate that the value can be decreased further because the scheduler has no problem scheduling the task and if not we're not losing anything by requesting a shorter sleep period.

Looking at figure 16.1, on the left are the results from using an update rate of 500 ms while the right shows results using an update rate of 250 ms. The blue boxes highlight the calls that exceeded the requested interval, while the red boxes highlight calls that did not exceed their requested sleep interval. There is clearly a higher proportion of sleep requests that do not exceed the requested time using the lower 250 ms rate which facilitates a faster update rate. Another consideration to look at is

Figure 16.1: Timing results using a normal thread priority.

the thread priority used and for the test above, a normal priority was used as categorized by the Java thread priorities. This offers an alternative method of potentially improving the chances that the agent thread is scheduled as soon as its ready to be run again by increasing the thread priority. To see if this is true, the test was repeated using the same time intervals but changing the thread priority to high.

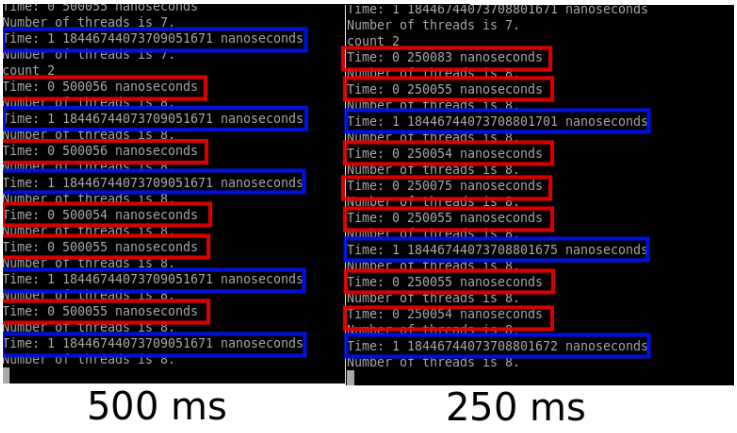


Figure 16.2: Timing results using the highest thread priority.

Analysis of figure 16.2 shows that changing the thread priority does not have a significant affect on the proportion of on-time sleep intervals, the results to a large extent, mirror the results in the last test. In this case it appears that thread priority does not have a difference but this does not rule it out having an affect at other time scales. An increase in the update rate is advantageous and so a new value of 250 ms was used as the time period between data requests within the agent and the visualiser.

Chapter 17

Evaluation

The project has proceeded well and proved to be difficult at times but I firmly believe that progress has been made towards a better understanding of threading interactions. Results from the proof-of-concept tool suggest that there are more intuitive ways to visualise information regarding concurrent interactions that help concurrent programmers debug problems within their applications and adjust to increase performance. In this sense the tool has been valuable in seeing whether such visualisations are possible which has shown to be true and although there is no third-party evaluation to back-up the usefulness of such visualisations, there is still much more potential to develop and pursue this idea further.

17.1 Context Survey

Many of the tools seen in the context survey are tools used in large software firms. Some of the tools have been research artefacts but its clear to see that there is demand for tools that allow programmers insight and better understanding of the interactions between the threads in their concurrent programs.

Many have very similar visualisation methods for the information, the key difference between them being the scope of their information. The fact that not many tools if it all, bother to use visualisations enforces the idea that the focus of these tools is simply to get as much information as possible. Where as, the focus of my implemented tool has been to visualise these data gathered and it offers something that is not present in any of the tools surveyed so far, complete visualisation of all the data gathered by the tool. Where there are visualisations in other tools, they are typically quite basic with a majority of the information compiled by the tool displayed in a textual format without any emphasis on important aspects. Not only does my implemented visualiser offer information, it has designed the visualisations in such a way as to emphasise such important aspects.

17.2 Similar Research

17.2.1 JACOT

JACOT or Java Object Concurrency Tool was a prototype research tool that was created to visualise concurrent interactions. It is particularly interesting because the research direction was very similar to the one here and they used some interesting visualisation methods. It is an events-based visualiser that uses the Java Virtual Machine Debugger Interface and UML sequence diagrams to model the concurrency interactions. The approach used for data gathering was exactly the same as the one in this project and one of the views showed exactly the same information. For example they had

a thread state view that groups threads into their state using a colour code but instead of using a moving graph, they represented the thread states as boxes on the screen and the threads as circles that moved between the boxes as they changed state. The other view focussed on representing the execution of the program by keeping track of the objects and the method invocations on these objects. This was done by giving each object a column with a box at the top and as time passed a line would extend from the bottom of the box towards the bottom of the view. It was scrollable and so the complete history is available. It was particularly effective at detailing the execution but required the user to stay alert and pay attention to the visualisation.

Many of the differences regarded the information it collected, in addition to the thread state information it collected events on object allocation and method invocation. Using events for these increases over-head significantly and does affect performance of the targeted application. There is no mention of taking performance into account when they were designing the tool and this could have a large effect on the result of the interleaving. The performance degradation could be to such an extent that running the program with the tool creates interactions that may never occur under normal conditions, giving a false impression. With the increase in the number of cores, the number of threads executing concurrently is at least 8 on a up-to-date computer. Looking at the sequence diagram view with this would result in a huge number of lines between the threads and consequently will result in the user having trouble following the execution [Leroux et al., 2003] .

17.2.2 DYVISE

This is an interesting real-time analysis tool that permits problem-specific visualisations to understand the execution flow of a system. The focus was on creating a tool that allowed the user to understand the execution of a system while minimising the burden on the user to instrument their code. It specialises for a problem by allowing the user to specify the problem in terms of threads, tasks and sub-tasks and utilizes the user definition of these to build a system-specific model [Reiss and Karumuri, 2010]. It does not explicitly show the information such as thread states but allows the user to effectively mark sections of code that they are interested in and specify threads that they want to watch. It uses an event-driven approach to get information from the target program. Once the program is executing it draws each thread as a horizontal section in the view and colour codes tasks and sub-tasks a specific colour, an example screen shot is below.

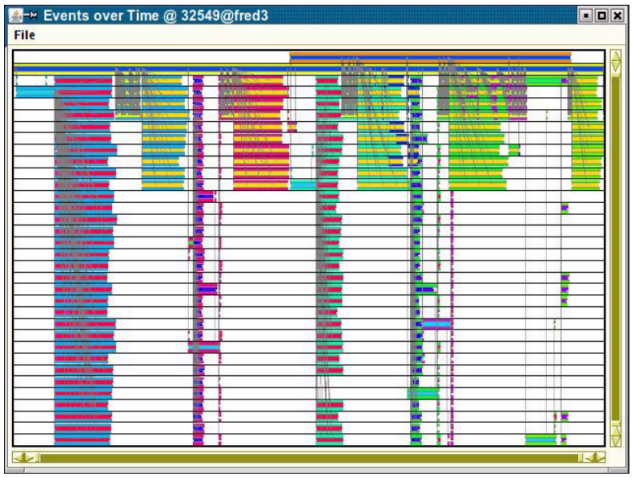


Figure 17.1: DYVISE tool screenshot, each horizontal coloured bar is a thread. The coloured sections represent whether its executing a task or sub-task.

Extra information about the thread and method invocations is available via dynamic tooltips, created when a user hovers their mouse over a section of interest. The flexibility of the tool is obvious but it does suffer from drawbacks including the assumption that the user understands the problem in the first place and can express it effectively using the primitives provided. It is not clear how much concurrency information is available apart from listings of the threads themselves. While it does to some extent visualise threads, the only similarity with my tool is its approach to data retrieval.

17.2.3 Alternative Visualisation Method

Another visualisation method that is based on UML sequence diagrams was the result of some research that focused on how best to evaluate results from trace tools or other verification tools. Targeted again at Java, they extended beyond the standard UML diagrams to incorporate extra concurrency information. It visualises in a very similar way to the first tool JACOT but in addition it integrates context-switching between the threads within the application, this explicitly exposes the happens-before dependencies within the threads. This allows reasoning about the re-ordering of events within a particular area. Other synchronization methods such as wait, notify and sleep were also recorded and displayed in the visualisation, the figure below shows an example visualisation.

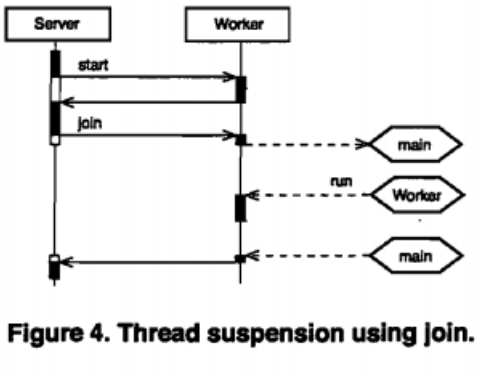


Figure 17.2: The visualisation shows multiple threads on the right and explicitly shows the interleaving between threads.

The dotted lines away from a hexagons indicate the creation of a thread while dotted lines towards the hexagons signify the scheduler context-switching off that thread. In this visualisation threads are modelled as both data structures and tasks, hexagons on the right are threads while the square boxes at the top are tasks that can be executed [Artho et al., 2007]. The solid black arrows indicate a task switch with the wide vertical sections indicating execution of that task. Its important to note that some of the task switches are heavily influenced by specific method invocations, for example wait would cause the current thread to cease execution while notify would allow suspended threads to become eligible for execution again.

The method here is quite comprehensive in that it maps exactly how the threads are interleaved and includes some concurrency information. It is similar to my implemented visualiser in that is shows when threads have been suspended or blocked, although it does not make this explicit. An interesting point is to note that the tool requires trace information from other sources and provides no abilities to gather information. This means that the accuracy of the visualisation depends on the granularity of the data that it is given. While the visualiser shows Java concurrency semantics, there is no requirement for the profiled target to be in Java which is slightly contradictory because it would require transforming the trace to Java specific concurrency semantics. An additional disadvantage

is that it will not work with real concurrent programs where multiple threads can be executed on multiple cores.

17.3 Objectives

The objectives that were set for this project initially were set without any real knowledge of what was possible and on hindsight it might have been a good idea to re-focus the objectives once more research had been done into areas such as the JVMTI.

Two tools have been created that are able to show all of the Java thread states for both thread pools and standard multi-threaded applications. This can be seen in the general visualiser's thread state view. Threads that own locks are also highlighted as well as the threads waiting on the lock, this can be seen in the thread lock view of the general visualiser. There is no functionality to support information about the section of code where the lock occurred, however, there is support to show the interactions between the thread pools. The thread pool visualiser is able to colour code the tasks depending on what pool they're in and can be seen in the thread pool visualiser's 'task view'. All but one of the requirements of the primary objectives have been met.

All of the requirements that do not require a thread pool have been met in the general visualiser. Thread state information, monitor information and information on blocked threads are all available to any program that does not use the thread pool pattern by using the general visualiser, therefore the first secondary objective has been met except for the fact that it does not highlight specific code sections. Although it is not explicit, deadlock can be viewed in the lock view of the general visualiser by looking for cycles within the blocked threads therefore, partially meeting the second objective.

Chapter 18

Conclusion

This project has been an investigation into visualisation the interactions between threads with a focus on the thread pool pattern as a prime example of where such analysis could be applied. It has involved independent research into the key factors that affect users when it comes to understanding concurrent interactions and potential methods for data retrieval using the JVMTI and the Java language. The other large part of this project has been to discover and implement ways of visualizing this information to the user in an intuitive and useful manner. It specifically targeted the Java platform due to its Executor framework and a large number of support libraries available for creating graphics. And although all the examples are of Java programs, the principles can be applied to any language because the concepts behind concurrency are the same throughout most imperative languages.

18.1 Key Achievements

The biggest achievement of this project is the discovery of new ways to visualize thread interactions within normal multi-thread applications as well as leveraging the extra information available when it comes to visualising thread pools. It has created new visualisation methods of existing data permitting a better understanding by the user but it has also added new information to improve this understanding. Many other visualisations not yet seen exist and their realization is bounded only by the programmer's creativity and the expressiveness of the graphics libraries utilized. Visualizations aid the user's understanding of concurrent interactions going on within an application and make it more likely that better concurrent code is produced. It has also been demonstrated that visualization as a method, is a good approach to showing such interactions, for example, the general visualiser it is able to show threads that are blocked as well what they're waiting on. A lot of the data available in the tools surveyed could be equally shown in a visual way similar to the example given above. Some things are more intuitively shown in a graphical manner, such as bottle-necks in the composite thread pool or heavy lock contention. A good visualization makes it much easier to interpret complicated behaviour much better than poring through textual data [De Pauw et al., 2002, p. 157]. The implemented visualizer offers an alternative method of imparting understanding to the user and is very different in the way that it does this compared to previously seen tools. It leverages the thread pool pattern to allow new insight into how a visualizations of thread pools and what they can potentially offer.

18.2 Drawbacks

The main drawbacks to the implementation in this project are that they're very specific and not very flexible, especially the visualizer that requires the use of Java's Executor framework. However, these can be generalized and made more flexible if time allowed. Further drawbacks to the tool include the simplicity of the visualizations, someone with more graphics programming experience could probably create something more appealing to the user but my lack of experience here prevented this. A lot of the data gathered in this scenario may not apply to other runtimes that forbid such intrusion into their inner workings. On the contrary, Java may have restricted the scope of visualizations here with the data available. Other languages may provide other concurrency primitives or more information which could facilitate better understanding by increasing the scope of the data.

With respect to the first general visualiser, the lack of synchronization between the agent and visualiser may result in missed information or no information at all. Every time the agent writes to the file, the old file is truncated at which point the visualiser may try to read at this point and sees an empty file. This manifests itself in the visualiser as 'gaps' in the visualisation. An alternative design could utilize an event queue, decoupling the agent from the visualiser, allowing the visualiser to have a dynamic resolution controlled by the user. A restriction is also imposed on the target Java program that forbids programs organised in packages because there is no way in the current implementation of the visualiser to know this. It is safer for the GUI not to assume anything about the directory structure and it makes no attempt to prepend parent directory names onto class names. A fix for this would be to ask the user to set the class path as the base directory and then the sub-directory names can be prepended onto the class name. The thread state visualiser also unable to keep the history of the thread states while other visualisations are shown, preventing the ability to show complete history since the program started execution. Currently, the history only goes as far back to when it began drawing and not of program execution. Implicit in the implementation of the second visualisation regarding CPU time is the assumption that the threads are created at the start of the program. It does not take into account the fact that some threads may be initiated after the visualiser has started and because the time is calculated from the program's initial execution, this may lead to skewed proportions. The effect of this is that threads may appear to have less proportional CPU time than is actually the case. To fix this, each thread will need to maintain a time stamp of when it was initiated and use this during the calculation of the proportional CPU value.

The tight coupling between the visualiser and agent can be reduced by abstracting out the current methods required to gather data, improving usability. In its current form, the implementation is also unable to explicitly show deadlock within a pool. A group of tasks taking a long time to execute could equally be interpreted as locked worker threads.

18.3 Future

As a result of this project, it has been shown that visualisation is a viable approach to tackling understanding in concurrent areas. The most obvious advances can be made in the visualizations themselves, only a few were shown here but many more are possible. Further more, the visualizations in this project could themselves be expanded to include other sources of data, for example leveraging the fact that a thread pool is used. Other potential directions involve investigating whether there is a general approach that could be taken to get the essential data regardless of programming language or runtime, perhaps the construction of some kind of framework that can be deployed. Alternatively, the approaches taken here for the composite pool visualiser could be generalized so that they no longer require explicit incorporation into the user's code.

Chapter 19

Appendix

19.1 Testing Summary

Due to the nature of the program, it was easy to detect any abnormalities in the program using a form a dynamic analysis by comparing the visualization on screen with the data that had been gathered. This caught the majority of simple bugs, there were some implicit logic bugs that were not detected this way and were only detected by doing static code analysis once a development cycle had finished. Other testing methods such as test-driven development would not have given much of an advantage and added increased the development time. The implementation does not use complex algorithms or have many many execution paths so it was decided that using any more than standard static code analysis would not be appropriate from a cost benefit point of view. Apart from this, standard good coding practices were employed to minimize the chance of bugs within the code and refactoring was also used to increase clarity and modularity. The section below shows the testing carried out on the final implementations and their results.

19.1.1 Testing - General Visualiser

It is difficult to test dynamic tools like this one because the whole process needs to be examined and not just the result at the end. Two approaches were used to verify correctness, firstly, test programs were run with the visualiser. By knowing the behaviour of the program we can see if the visualiser gives the same expected result. The second approach compares data given out by the agent with what is being visualised. This comparison approach is akin to sampling, the biggest disadvantage here is that there is no guarantee that the condition tested for at one point, in this case matching values, is true throughout the execution of the program. However, if it can be shown that at some points in the program the values match then this is enough because the execution logic is the same for each set of values. This is to say, that there are not many if any execution paths through the whole program. It should be noted that JVM threads as well as user threads show up on the visualiser so its important the can be distinguished. The JVM threads are DestroyJVM, SignalDispatcher, Finalizer and ReferenceHandler while user threads start with Thread or the name of the class that contains the main method for the Java program.

Each test program was designed to test different aspects of the visualiser with the first focusing on showing that the visualiser can correctly detect blocking. The second test program shows that it can handle a relatively large amount of threads while the third test program shows that the visualiser can deal wth threads spawning and dying in the middle of the program. Throughout the tests general aspects of the visualiser will also be tested including its ability to detect the correct number

of threads, show the threads in the correct states and ensure that timing data is accurate for the second visualisation.

19.1.1.1 Test 1 - Synchronization Locks

This test attempts to show that the visualiser can correctly detect blocked threads by using a test program of which we know the behaviour of. In this case, the test program simply spawns a small number of threads, all of which attempt to access a synchronised section. What should be seen is only one thread continuing execution while the rest are blocked waiting for the running thread to relinquish the monitor lock. After the thread has completed its work inside the synchronised section, it simply releases the lock and dies. So each thread will eventually get a turn at holding the lock for this monitor. Results should also show the correct number of JVM threads within the JVM a long with their states. The parameters for this test were 5 spawned threads plus the main thread. On the visualiser we should see all user threads blocked apart from one which is running, and over the execution time of the program, each of the blocked threads will eventually start to run and then die.

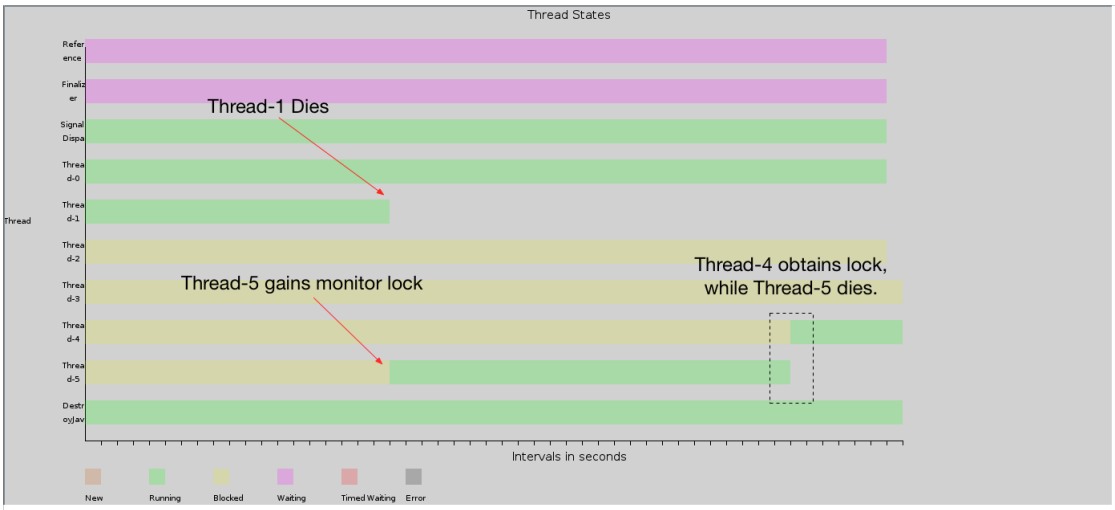


Figure 19.1: Thread visualiser showing sections of interest.

Labels in the above figure highlight the areas where the threads change state, either going from blocked to running or from running to nothing (i.e. dying). Behaviour shown in the thread state visualisation above is consistent with the expected behaviour and so it passes this test. The other visualisation methods also need to be confirmed as showing correct data. Testing the second visualisation requires the sampling approach, we do this by comparing values visualised on the bar graph against values coming from the agent. To this end, the agent will be modified to output extra data corresponding to values of interest which include total execution time of the program and the amount of CPU time a thread has. Whichever thread has the largest bar should have the largest ratio value when looking at the agent data. We can also say that the original time stamp used to calculate the total execution time of the program stays constant and that the thread with the highest CPU time will correspond to the same thread with the highest bar.

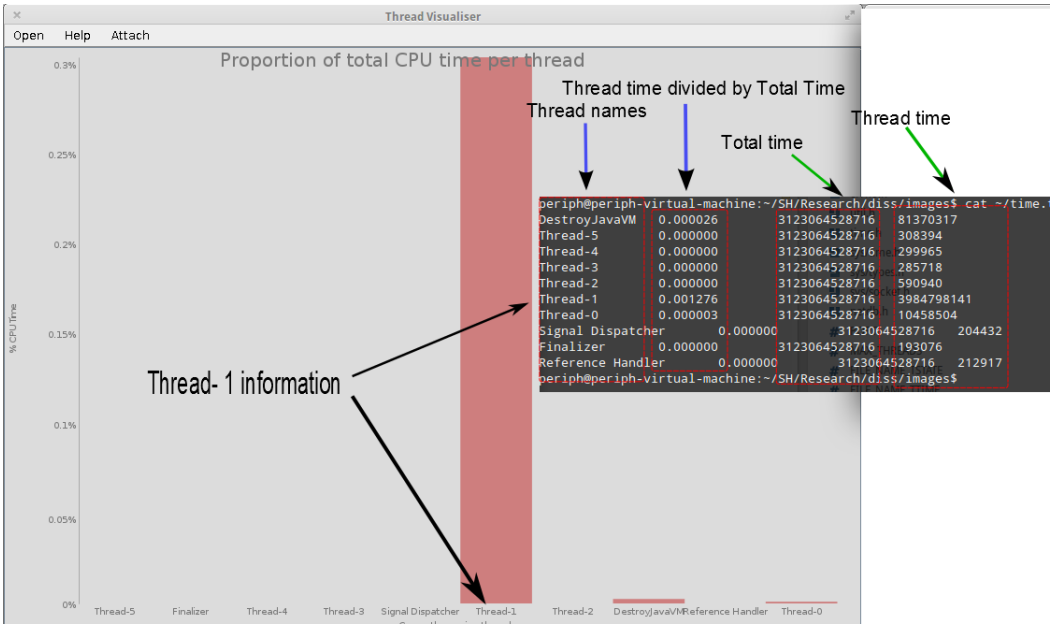


Figure 19.2: Output of the agent on the right with the CPU time visualisation behind it. It shows that data from the agent is correctly shown by the visualiser.

This screenshot shows output of the agent with the bar graph, it is clearly visible that the highest bar corresponding to Thread-1 has the highest value in the Thread time column (furthest from left) and the highest value in the proportional thread time colum (2nd from left). It can also be seen that the total time column shown 3rd from the left is also constant. These three statements satisfy the expected behaviour and so it passes this test as well. For the third visualisation, the approach used in the first thread state visualisation can also be used here. It shows the same data as the first visualisation except it includes extra information on monitor locks. From the description of the program we can expect to see one thread highlighted as having a lock while the rest of the spawned threads are in a blocked state waiting.

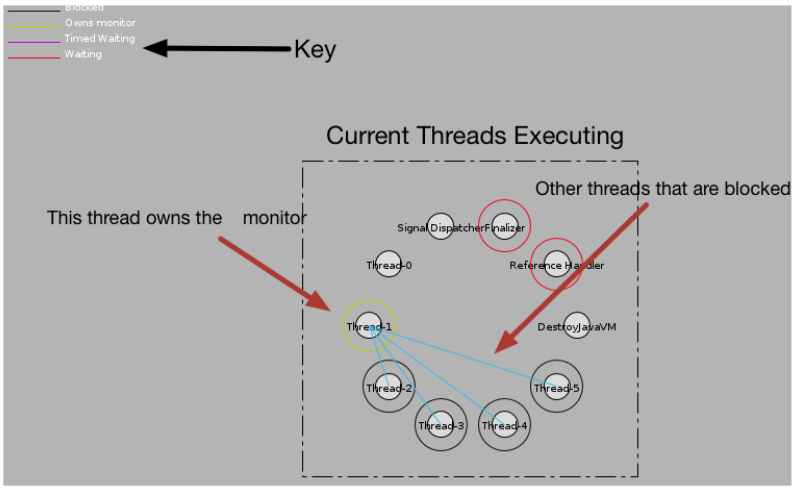


Figure 19.3: The third visualisation method showing all of the threads and their states.

The screenshot shows Thread-1 highlighted as owning a monitor lock and 4 other threads in a blocked state waiting for the same monitor lock. This is what was expected and so this passes as well.

19.1.1.2 Test 2 - Thread pools.

The test program here starts with a thread pool containing 7 worker threads. 100 tasks are then submitted for execution and the program terminates after all of these tasks have completed. This test will attempt to check that the visualiser can deal with a relatively large number of threads while still maintaining the other general aspects. It also allows some insight into the scalability of the tool. In this test the tool should show a number of worker threads all executing at the same time with no monitors. Throughput should be relatively high so the second visualiser should show a number of high bars, each representing a worker thread from the pool.

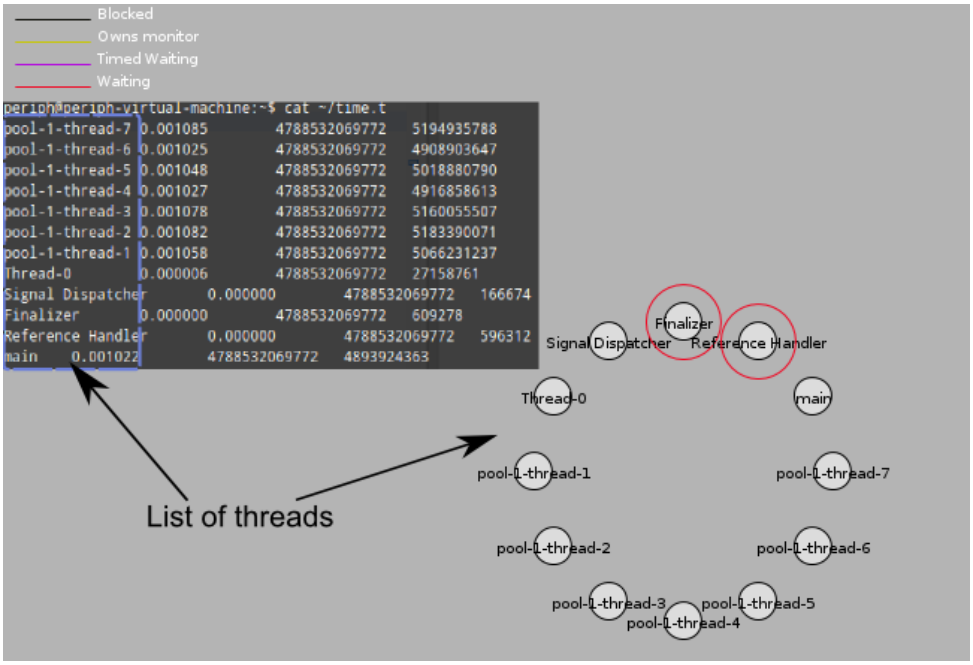


Figure 19.4: Third visualisation showing all the correct worker threads.

By examining the list of values in the console output, it shows the pool threads all having high proportions of CPU time relative to other threads. The list of values is just to the right of the highlighted blue box. This is exactly what is expected of this test program and if we were to look at the second visualisation, all of the worker threads would have had large bars. With respect to scaleability, it shows that there is a limit to the number of threads that can be displayed. The information is not yet obscured by the number of threads but any increase higher than the number currently shown may start to impact on the clarity.

19.1.1.3 Test 3 - File IO

In the final test program, an IO bound program is created to test how the visualiser would react. The theme is similar to the first test program except, instead of all new threads being spawned almost immediately, a delay is created. The visualiser can then be examined to see how it deals with spawning threads after the visualiser has started drawing and whether the correct state information

is displayed. Each newly started thread will write 100 MB worth of ‘A‘ characters to a file. What should be seen is threads starting in a staggered fashion and depending how long it takes for each thread to create the file, they should all die in the same order that they started. It is expected that the main thread will always be in the timed wait state because of the sleep method on that thread but should start running once the sleep timer has expired.

Figure 19.5: Thread state visualiser showing the correct behaviour of the third test program.

The screen shot shows the result of the test and it shows that writing the 100 MB file takes a short period of time and also more importantly the correct behaviour of the threads. The main thread is always shown in the timed wait state due to the thread sleeping and while there is no change of state shown when a new thread is spawned, it is still correct. This is down to the fact that spawning a new thread is many orders of magnitude smaller in time scale than the resolution of state change detection. While this might be seen as a fault in the visualiser, there is no practical way to get resolution on that time scale, primarily due to the type of scheduler. In addition, the interactions that the user will be interested in are on a much longer time scale.

19.1.2 Testing - Threadpool Visualiser

Because of the way this visualiser was implemented, there is not any way to verify it against test programs. Instead, the focus will primarily be on verifying that the data shown on the screen is what is actaully happening underneath. To do this, various parameters within the code that control aspects of the composite thread pool will be changed and their effect analysed on the visualiser.

19.1.2.1 Task Visualisation

The first test will confirm the correct number of jobs being displayed within the visualiser; this will be done by comparing the global variable which sets the number of jobs to the number of tiles seen on the visualiser. The second test will confirm the number of tasks within each pool. This will verify the colouring for each tile is working correctly. The number of tasks per pool is equal to the total number of tasks that have been submitted for execution in that pool minus the total number of

tasks that have completed execution in that pool. A method that repeatedly queries the underlying executor objects is used to extract the required information and print it to the console.

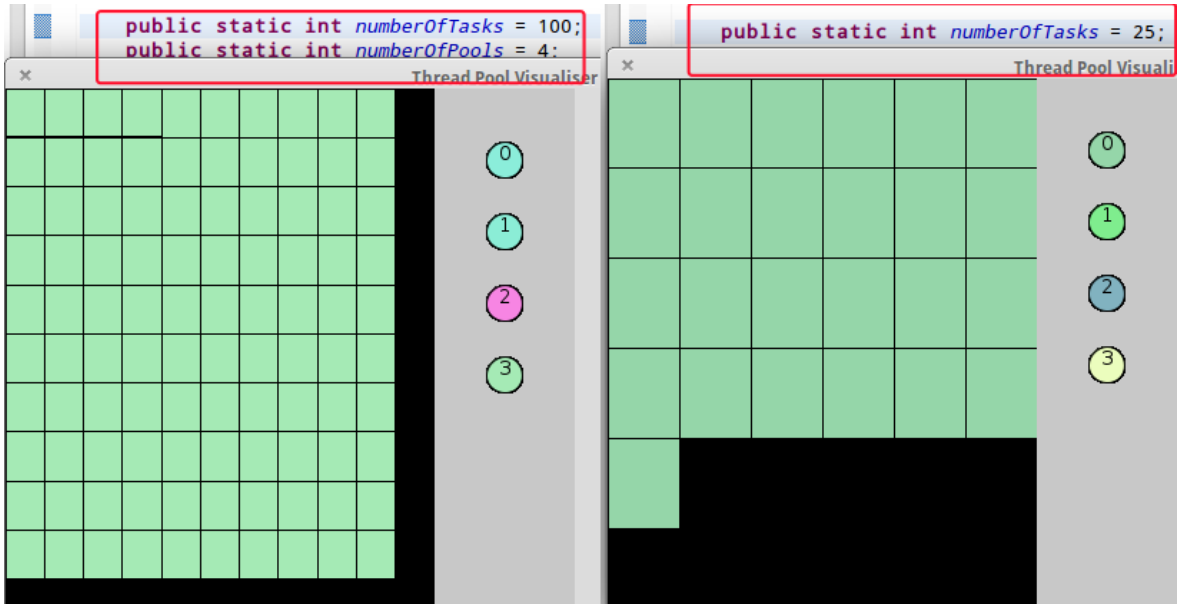


Figure 19.6: Comparing the number of tiles with the number of submitted tasks.

Examining the number tiles compared to the value of the submitted jobs within code shows a match and so it passes this test. The following screen shot shows the visualizer plus output from queries done on the underlying executor objects. By comparing the number for each pool from the console output with the number of coloured tiles that represent that pool, it confirms that the number of coloured tiles corresponds to the number of tasks in that pool.

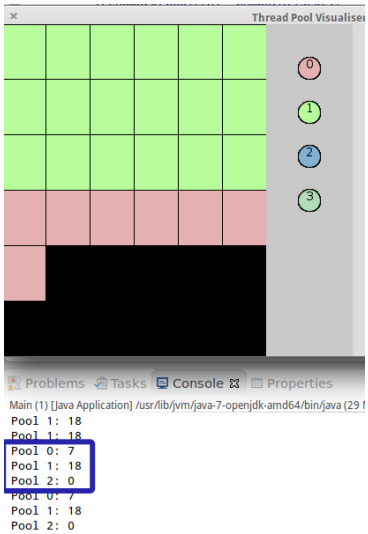


Figure 19.7: Comparing the number of tiles with the number of submitted tasks highlighted in the blue box.

Values highlighted by the blue box show the calculated number of tasks per pool. There are exactly

18 green tiles which corresponds to the second pool, while there are exactly 7 tiles corresponding to the first pool.

19.1.2.2 Job Queue Visualisation

A similar line of action was taken for this part of the visualizer. The main focus of this part of the visualizer is to add more information to the task visualization, in particular it allows the user to see the number of jobs waiting in a particular pool. So to confirm that these values are correct, the visualizer was modified slightly to output these calculated values. To calculate the number of waiting jobs for a particular pool, the underlying executor needs to be queried for the number of submitted tasks, the number of tasks that have completed execution and the total number of currently executing threads. Then to work out the number of jobs still waiting to be executing we simply subtract the number of executed tasks plus the number of currently executing tasks. The values are then output to the console and compared with the values shown in the visualizer; they're expected to match.

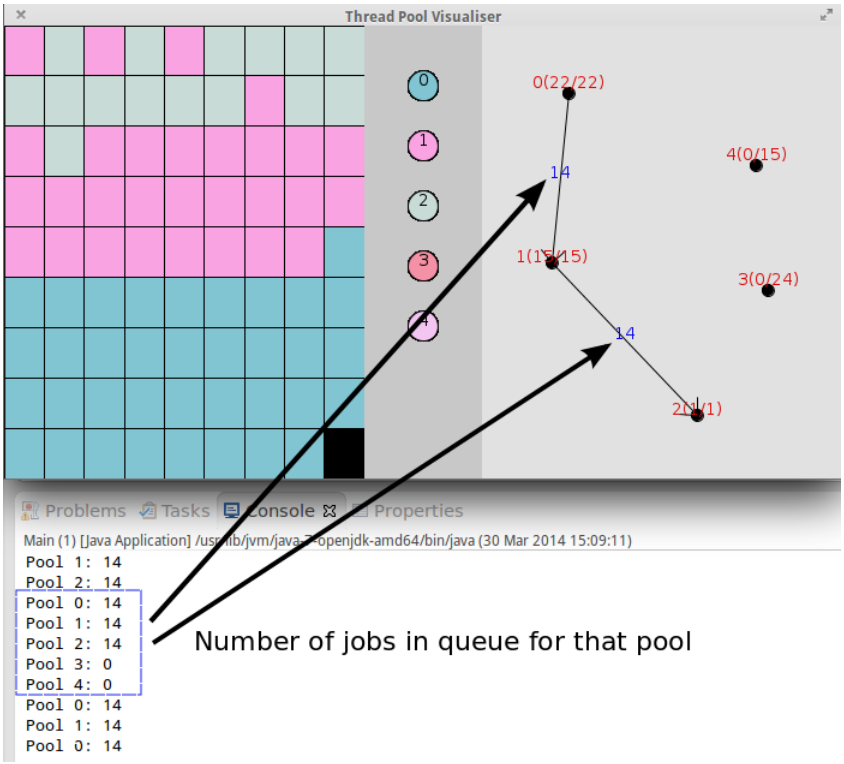


Figure 19.8: Comparison of waiting jobs with the calculated values.

19.1.3 Testing - Conclusion

An attempt was made to verify the important aspects of the visualizer and confirm that they are indeed working as expected. The testing has not been exhaustive and there are many other facets that could be looked at. Taking into account the visualizers simple logic behind the data gathering, the amount of testing shown here is deemed to be sufficient.

19.2 Status Report

The software artefact comes in two parts, the first part being the more general thread visualizer that includes a Java front-end GUI and a native agent library. And the second part which is a combination of a composite thread pool implementation including test harness and visualizer. Both of these are fully functioning tools although not feature complete yet.

Potential improvements to the visualizers have been highlighted in the conclusion section in additional to new directions for further research.

19.3 Appendices

19.3.1 Build & Execution

In order to build the visualizers, the following tools are required (ideally latest versions):

- make
- ant
- GCC toolchain (at least C compiler and C++ linker)
- Java (must be at least version 1.7)

Ant is used as the main method to drive builds, there are two separate ant build files for each visualization tool. For the general visualizer the make file requires modification in-order to correctly compile the shared library. Simply edit the makefile inside the agent directory and set the appropriate path to the directory that holds **jvmti.h** and **jni.h**, typically in the default install directory of the Java SDK, see makefile for more details.

19.3.2 General visualiser

To build the general visualiser, execute the following in the base directory:

```
ant -buildfile genvis.xml
```

19.3.3 Thread pool visualiser

To build the thread pool visualiser, execute the following in the base directory:

```
ant -buildfile tpvis.xml
```

19.3.4 Tested Builds

Tested on the Mac machines in the sub-honours lab, there were some issues with the thread pool visualisation being quite slow. You may want to run it on a more modern machine. The Mac machines ran OS X with:

- ant 1.9.3
- make 3.81
- Apple LLVM 5.0

- java 1.7

The agent is contained inside the agent directory while the general visualiser is inside the vis directory and finally the source files for the thread pool visualiser are inside the ThreadP directory.

19.3.5 Documentation

The main documentation of the tools comes in the form of the source code and the explanation of design decisions is in this document. Additional documentation is contained in Javadoc for the Java sections of the project while the native code contains source comments.

19.3.6 Usage Instructions

Using the tools should be quite straight forward, for the general visualiser select either open or attach to execute a Java class file that contains the main entry method. **Note:** attaching does not allow you to see monitor information.
Restrictions:

- Cannot execute a Java program in a package
- Directory structure must not be changed before a build, otherwise the scripts will not work.

Note: Closing the general visualiser does not shut the target program down. Restarting the visualiser without terminating the agent will lead to erroneous results because the agent will still be writing data to the file. If the agent shuts down gracefully, it will remove the temporary files in the user home directory.

Run the jar files from the command-line by running:

```
java -jar tpvis.jar# For thread pool visualiser.
java -jar genvis.jar <number of thread pool> <number of tasks> # For the general visu
```

In the case of the thread pool visualiser it also takes additional arguments, namely two numbers to set the number of thread pools and the number of tasks

Bibliography

[Artho et al., 2007] Artho, C., Havelund, K., and Honiden, S. (2007). Visualization of concurrent program executions.

[Beck et al., 2001] Beck, K., Grenning, J., Martin, R. C., Beedle, M., Highsmith, J., Mellor, S., Bennekum, A. v., Hunt, A., Schwaber, K., Cockburn, A., Jeffries, R., Sutherland, J., Cunningham, W., Kern, J., Thomas, D., Fowler, M., and Marick, B. (2001). The agile manifesto. http://agilemanifesto.org/. Last accessed March 22nd, 2014.

[Cantrill and Doeppner Jr, 1997] Cantrill, B. M. and Doeppner Jr, T. W. (1997). Threadmon: a tool for monitoring multithreaded program performance. In *System Sciences, 1997, Proceedings of the Thirtieth Hawaii International Conference on*, volume 1, pages 253–265. IEEE.

[cr, 2006] cr (2006). Lockness - faq. http://lockness.plugin.free.fr/faq.php. Last accessed March 31st, 2014.

[De Pauw et al., 2002] De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J., and Yang, J. (2002). Visualizing the execution of java programs. In *Software Visualization*, pages 151–162. Springer.

[Fry et al., 2014] Fry, B., Reas, C., and Volunteers (2014). Processing. http://www.processing.org/. Last accessed March 24th, 2014.

[Google, 2014] Google (2014).

[JetBrains, 2014] JetBrains (2014). Debugging tool window. threads. http://www.jetbrains.com/idea/webhelp/debug-tool-window-threads.html. Last accessed March 31st, 2014.

[Leroux et al., 2003] Leroux, H., Réquilé-Romanczuk, A., and Mingins, C. (2003). Jacot: a tool to dynamically visualise the execution of concurrent java programs. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, pages 201–206. Computer Science Press, Inc.

[Microsystems, 2002] Microsystems, S. (2002). Chapter 5 - local and global references. http://zavgren.com/JNI/refs.html. Last accessed March 20th, 2014.

[Morrison, 2005] Morrison, V. (2005). What every dev must know about multithreaded apps. *MSDN Magazine*.

[Prasad et al., 2004] Prasad, C. K., Ramachandani, R., Rao, G., and Levesque, K. (2004). Creating a debugging and profiling agent with jvmti.

[Reiss, 2003a] Reiss, S. P. (2003a). Jive: visualizing java in action demonstration description. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 820–821. IEEE.

[Reiss, 2003b] Reiss, S. P. (2003b). Visualizing java in action. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 57–ff. ACM.

[Reiss and Karumuri, 2010] Reiss, S. P. and Karumuri, S. (2010). Visualizing threads, transactions and tasks. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 9–16. ACM.

[Zukowski, 2007] Zukowski, J. (2007). The attach api. https://blogs.oracle.com/CoreJavaTechTips/entry/the_-attach_api. Last accessed March 20th, 2014.