5.1 INTRODUCTION

Due to proliferation in the software and system's market, it has become necessary to evaluate the things for various measures like performance, reliability, security etc. There are several more problems associated with the software's which include the underutilization of client resources, installation of additional hardware equipments and the congestion of computer systems either because of complete memory or CPU utilization. Therefore, the use of software and systems with enhanced performance are in high demand. There are various analytical tools available in the software engineering for examining the behaviour of applications like function-calls, time, CPU usage and memory usage etc. These tools have dramatically changed the way of analysis and become essential for optimizing an application's performance and profilers are one of them [248, 249]. Profilers can be used to find the overall appropriate level and bottlenecks in the application. These bottlenecks are the troublesome spots, which can be hidden and occur during the execution of the application. So, with the help of profilers one can reduce the time for detecting these bottlenecks in the applications. Most modern profilers support the graphical representation of the results obtained to facilitate quick analysis of the developed application. The role of profilers becomes important when the application is a real-time system where they check for whether the real-time task meet their deadlines and matches the estimated time derived by static analysis. In this chapter, we have evaluated the performance of our proposed energy sustainable framework of the power scheme under Swift and Exhaustive modes. This study investigates on more convenient and effective methods, which can minimize computer system's load and enhance the overall performance at the same time. The proposed framework requires only configuration and no installation is required, which in turn not only maximizes the conveniences for users but also keeps the system resources free for other works.
The performance measures for the proposed energy sustainable framework of the power scheme include the various system components like thread monitoring, analyzing memory, and CPU performance under specific workload. We used the profiler available in NetBeans-IDE [247] for evaluating the performance of proposed energy sustainable framework discussed in chapter 3, under different proposed modes with the help of the proposed algorithms. By using profiler, one can easily determine the performance of system’s memory and CPU under various performance measures like memory leakage [250], memory heap, and memory garbage collection [251], thread monitoring, CPU timestamps for each invoked methods etc.

5.2 THREAD MONITORING
During the thread monitoring session, the profiler monitors system and application threads activity and displays the information in the threads tab. This tab is responsible for displaying the timeline for each running threads with their states like running, sleeping, wait and monitor. We can also get the detail of time spent in each state. For the proposed energy sustainable framework, we have monitored various active threads in both the modes: swift and exhaustive. On the basis of their execution, these are typically divided into two categories known as system thread and user thread. Table 5.1 provides the overview for both types of threads with their uses class and descriptions.
<table>
<thead>
<tr>
<th>S. No</th>
<th>Thread Name</th>
<th>Uses Class</th>
<th>Type of Thread</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Reference Handler</td>
<td>java.lang.ref.ReferenceHandler</td>
<td>System</td>
<td>High priority thread that enqueues pending references.</td>
</tr>
<tr>
<td>2</td>
<td>Finalizer</td>
<td>java.lang.reflect.Finalizer$Finalizer$Finalizer$FinalizerThread</td>
<td>System</td>
<td>Performs finalization of objects before their garbage collection.</td>
</tr>
<tr>
<td>3</td>
<td>Attach Listener</td>
<td>java.lang.Thread</td>
<td>User</td>
<td>User Thread</td>
</tr>
<tr>
<td>4</td>
<td>Java 2D Disposer</td>
<td>java.lang.Thread</td>
<td>System</td>
<td>Handles disposal of native data associated with Java objects in Java 2D.</td>
</tr>
<tr>
<td>5</td>
<td>AWT Shutdown</td>
<td>java.lang.Thread</td>
<td>System</td>
<td>AWT system thread, handles shutdown of AWT (EventQueue) when no GUI is</td>
</tr>
<tr>
<td>6</td>
<td>AWT-EventQueue-0</td>
<td>java.awt.EventQueue$EventQueue$EventQueue$EventQueueThread</td>
<td>System</td>
<td>AWT thread, which is the main thread executing a GUI.</td>
</tr>
<tr>
<td>7</td>
<td>DestroyJava VM</td>
<td>java.lang.Thread</td>
<td>User</td>
<td>User Thread</td>
</tr>
<tr>
<td>8</td>
<td>Timer Queue</td>
<td>java.lang.Thread</td>
<td>System</td>
<td>Used to manage all javax.swing.Timer instances in one thread.</td>
</tr>
<tr>
<td>9</td>
<td>Thread-7</td>
<td>java.lang.Thread</td>
<td>User</td>
<td>User Thread</td>
</tr>
<tr>
<td>10</td>
<td>Thread-8</td>
<td>java.lang.Thread</td>
<td>User</td>
<td>User Thread</td>
</tr>
<tr>
<td>11</td>
<td>Thread-10</td>
<td>java.lang.Thread</td>
<td>User</td>
<td>User Thread</td>
</tr>
</tbody>
</table>
5.2.1 Thread Monitoring in Swift Mode

Here, we have executed the proposed energy sustainable framework under swift mode and performed thread monitoring. We have executed the framework for the discussed usage scenario in chapter 4 up to 20 minutes. The results obtained after thread monitoring are shown in the Figure 5.1.

![Figure 5.1: Various active threads during the framework monitoring in swift mode.](image)

From Fig. 5.1, one can easily observe that the user threads 7 and 8 gets started into sleep mode by providing the login-duration time to the framework and after a certain period of time inner-timer which corresponds to the thread 7 gets expired and the thread 8 continues till the end of the login-duration. Once the login-duration ends then the AWT-EventQueue-0 system thread, which is in wait state, gets activated and pop-up the message window to the user that “your time is finished” do you want to continue your working, if user reply for yes, then another message window gets pop-up to enter the extended time duration and the user thread 10 gets started into the sleep mode for the new login-duration.

5.2.2 Thread Monitoring in Exhaustive Mode

Here, we have executed proposed energy sustainable framework under the exhaustive mode and performed thread monitoring. We have executed the framework for discussed
usage scenario in chapter 4 up to 20 minutes and obtained results after thread monitoring are shown in the Fig. 5.2.

From Fig. 5.2, we can notice that all the system threads remain similar to the thread monitoring in swift mode. Whereas, various number of user threads could be observed in this exhaustive mode because proposed ESSA algorithm continuous checks the user activity on the machine and here we are observing the snapshots each with 1 minute duration and total login duration time is 20 minutes. Due to this reason, the user threads are created for each minute of login duration. In the Fig. 5.2, these threads are shown from thread-7, thread-10, and thread-11 to thread-28 each with one minute duration. As the supplied login duration time of 20 minutes gets expire proposed framework prompt a window to user for supplying new login duration time with one minute expiry and represented with user thread-30.
5.3 ANALYZE MEMORY PERFORMANCE

The analysis of memory performance of the proposed framework under different overheads includes the various results like memory heap, garbage collection and threads and loaded classes. Using VM telemetry, we analyzed these memory measures and obtained the various results as shown in further figures. Here, we have profiled the proposed framework for both the proposed modes: Swift mode and Exhaustive mode. In all our real time results of memory performance, we have profiled the proposed framework up to 20 minutes.

5.3.1 Heap Analysis

In Fig. 5.3(a) and Fig. 5.3(b) the memory heap size over the period of time for both the modes: Swift and Exhaustive have been analyzed, respectively. Here, we can easily find out the details of maximum available heap size versus used-heap of profiled framework for both the modes. For both scenarios, maximum available heap size is the same with variations in the maximum used-heap which is 9.5 MB to 13 MB, respectively for both the modes and the minimum used heap after garbage collection is 5.0 MB to 9.0 MB, respectively for both the modes. This is very clear as in exhaustive mode there is a continuous monitoring of the user activity for each minute is done by the framework. Here, Garbage collection is performed after a certain interval of time automatically, which minimizes the maximum used-heap sizes. These intervals are easily noticeable in Fig. 5.3(a) and Fig. 5.3(b). Throughout the login-duration framework continues its functioning smoothly, which can be noticed with the sharp edges of Fig. 5.3(a) and Fig. 5.3(b), but when the login-duration time comes to an end, there is always some deviations in the edge that refers to the activation and creation of new threads in the memory. To know more about these threads, one can refer the thread monitoring section as discussed above. We can also observe that for both of the scenarios maximum used heap never reaches up to maximum heap size and framework continues its functioning without any decrease in the performance.
5.3.2 Memory Leakage

The problem of memory leakage for the proposed framework has been analyzed by finding various surviving generations and relative time spent in the garbage collections. From Fig. 5.4(a) and Fig. 5.4(b), one can find that once the framework gets initialized total number of surviving generations becomes constant and remains at 6 for both the modes.
till the login-duration ends, which represent the framework behaves similarly for both of our proposed modes. So, there is no problem of memory leakage in proposed framework. Here, maximum relative time spent in garbage collection is 0.6% and 1.2% respectively for both of the modes as shown in Fig. 5.4(a) and Fig. 5.4(b).

**Figure 5.4:** Analysis of memory performance for Surviving generations vs. Relative time spent in GC. For each graph, x-axis denotes the time in (HH:MM:SS) and y-axis shows the surviving generations and y2-axis shows the relative time spent in GC (%). (a) Swift Mode and, (b) Exhaustive Mode.
5.3.3 Thread Analysis

In this subsection, we have analyzed the detailed view of the memory performance for various running threads versus loaded classes. It is very much similar to the scenario discussed in thread monitoring section. From Fig. 5.5(a), Fig. 5.5(b1) and Fig. 5.5(b2), one can find that the maximum number of threads remains at 10 and 11, respectively for both the modes, with a little variation in maximum loaded classes in exhaustive mode. In Fig. 5.5(a), this can be observed that during the supplied login duration of 20 minutes no new thread is created and total number of threads remains constant, which reveals that proposed framework is executed under swift mode and no popup window is activated during the supplied log-duration. The main reason behind this is in the swift mode, we are very much concern about the user and not about their activity for each minute on the computer system. Due to this, we are not concern about the percentage of total CPU usage in each snapshot, though we have recorded this CPU usage percentage in this mode too for future purpose. At the end of login-duration, some threads get activated from sleep mode and some are created newly, details of which can be found in the thread monitoring section. It is also because at this time popup window gets activated as the supplied login-duration time gets over to know the user status on the machine and threads are created.
Figure 5.5: Analysis of memory performance using ESSA algorithm for threads versus loaded classes. For each graph, x-axis denotes the time in (HH:MM) and y1-axis shows the running threads and y2-axis shows the loaded classes. (a) Swift mode, (b1) and (b2) Exhaustive mode. (b2 is an extension of b1).

In the exhaustive mode, as shown in Fig. 5.5(b1) and Fig. 5.5(b2) various peaks are shown after each minute of time interval, which represents that the ESSA algorithm is performing a check throughout the interval to find the percentage of total CPU usage and this utilization is found always below the threshold for each snapshots then an inner timer in the form of popup window to find the user’s activity on the machine gets
started otherwise this inner timer thread gets cancelled. The purpose of popup window is to know the user consensus on the machine. Here, to perform the performance evaluation till the end of login-duration we have pressed “YES” whenever the popup window was activated.

5.4 ANALYZE CPU PERFORMANCE
By using CPU performance measurement, the proposed framework have been analyzed and obtained data related to its performance including the time required to execute a code fragment within a method and the number of times that particular method was invoked. We have analyzed the CPU performance separately in both the modes: Swift mode and Exhaustive mode. For both the modes, we have profiled the CPU only for project related classes that includes the core java classes only.

5.4.1 CPU performance in Swift Mode
In this mode, the CPU performance analysis is performed for entire supplied login duration time up to 20 minutes. This analysis is shown in the following Fig. 5.6 (a1) and Fig. 5.6 (a7) that shows the call tree method for login-duration. CPUInfo and various threads created to monitor the proposed framework. Fig. 5.6 (a1) shows the call tree methods AWT-EventQueue-0, Thread-8 and main. The significant description about these methods is given in Table 5.1. We can observe that methods are invoked whenever they are required. Here, Thread-8 works till the end of login-duration.
Figure 5.6: Analysis of CPU performance in Swift mode (a1) call tree methods for AWT-EventQueue-0, Thread-8 and main (a2) call tree methods for Thread-7, Thread-10 and Thread-9.

In Fig. 5.6 (a2) call tree methods for user Thread-7, Thread-10 and Thread-9 are shown. Here, user Thread-7 gets stopped as soon as the framework start its functioning, whereas Thread-9 and Thread-10 created when the login-duration time gets over and the new login-duration time is asked to input for continue its functioning.

5.4.2 CPU performance in Exhaustive Mode

In this mode, we have analyzed the CPU performance by using ESSA algorithm that checks the user activity for each minute and record the snapshots of total CPU uses for each second in a file. The results obtained from this analysis are shown in Fig. 5.7 (a1), Fig. 5.7 (a2), Fig. 5.7 (a3) etc.
In the obtained results, we have analyzed the CPU performance up to 20 minutes using only all project-related classes. In these results, we can find the number of variations from the previously discussed scenario. In Fig. 5.7 (a1) shown various call tree methods are similar to methods shown in Fig. 5.6 (a1). Here User Thread-8 continues till the end of login-duration and User Thread-7 stops its working as the framework settles down. In Fig. 5.7 (a2) various user threads are shown and this scenario is very much similar to discussed scenario for Fig. 5.2 in the thread monitoring section. In Fig. 5.7 (a2), the user threads are created for monitoring the user activity on the computer system for each minute and whenever the percentage of total CPU usage is found below the threshold a popup window gets activated to know the user status on the machine. This pop-up window also created some user thread for a smaller duration of time as in this performance evaluation we have always given our consent in “YES” whenever the pop-up window was invoked and executed the process till the end of log-in-duration. In Fig. 5.7 (a3) some user threads are expanded to show their detailed functioning. In this figure, we have expanded the user Thread-10, 12, 18 and 14, all these threads are invoked 60 times and recorded the percentage of total CPU Usage for each minute in a file.
For all the created user threads in each minute, we found no thread which over utilizes the CPU. Here, all the methods are executed for their assigned time limit and proposed ESSA algorithm invoking the popup window whenever the percentage of total CPU usage found below the threshold for that interval because we have considered the snapshots timing for a minute.

5.5 Conclusion

This chapter presents different alternatives to the use of Java Technology for real time implementation and evaluation of the proposed energy sustainable framework. To evaluate the performance of Java code is really a difficult task. Performance-wise Java virtual machine (JVM) is a black box for any system analyst, which hides a lot of detail in comparison to the native systems. JVM hides the various performance measures like memory allocation, performance monitoring counters on the CPU, and thread monitoring by abstracting the underlying hardware and employing custom byte code execution mechanism. These JVMs also employ different just-in-time compilers and different garbage collection algorithms. Therefore, it is difficult to understand the flow of program execution. Moreover, the behaviour of the Java execution environment is not predictable and a number of events can lead to non-deterministic behaviour and consequently to various difficulties in performance evaluation. A variety of tools for JVM monitoring and application profiling are available in the market like Netbeans profiler, JProfiler and WebSphere console which can provide a good overview of application behaviour. We have used the Profiler available in NetBeans for the evaluation of proposed framework and monitored various threads, memory allocation, memory leakage, garbage collection and CPU performance. In the various proposed performance evaluation results, we have observed the constant behaviour of the application under swift mode and exhaustive mode. We have obtained no problem regarding memory leakage and garbage collection is performed regularly after a certain period of time. In the thread monitoring analysis, threads are created accordingly as the events taken place in both the modes. During the execution of the proposed framework no unnecessary user threads are created and the CPU performance analysis methods are executed equal to supplied login duration time and no method or process is noticed for CPU over utilization, which in turn is responsible for overall system slowdown.

Therefore, these performance evaluation results revealed that the proposed algorithm
achieved the proposed goals both theoretically and practically for designing a complete energy-sustainable framework and algorithms and suggested that changes can be incorporated into the power schemes of the operating systems.