Optimization of Java Graphical Applications in PDAs

Nuno Faria

Departamento de Informática, Universidade do Minho
4710 – 057 Braga, Portugal
cei7160@di.uminho.pt

Abstract. This work aims to show the performance evolution of a JAVA graphical application (a SVG file viewer). The critical parts of the application, mainly at the processor level, are analysed, through the use of a JAVA profiler. The impact of different optimization techniques on the application performance is shown. A performance comparison between the application of the most effective techniques and the initial code is presented.

1 Introduction

Current PDAs are limited at memory, processor, screen area and even with peripherals. The PDAs processors are most turned on to low power optimization and wireless communications than the speed performance (400MHz maximum speed). So, it is necessary (or at least wise) to build applications that not take lots of the available resources (that the concerning about memory and processor must be take in to account).

The application used to show the optimization techniques is a file viewer [1], based on a XML subset to generate 2D graphics (SVG, Scalable Vector Graphics) and which was developed in JAVA. This application, and like others graphical applications, needs to have a visualization area where the graphical elements are displayed. And because is a vectorial element based application, when applying transformations, all elements are calculated to new positions with no lost of image quality (this require some processing time).

The application contains the following modules:

- a SVG loader, which will parse the SVG files (the Crimson XML parser from Apache);
- a data structure, to save and access all the graphical elements parsed before;
- the visualization area, to display the graphical elements;
- the transformation module, where image operations like pan, rotation and zoom are available.

The test environment was based on a PC platform with a Pentium III – 1000MHz processor, 512 MB of RAM, WindowsXP Professional, JProfiler v2.4.1 and Java Development Kit v1.1.8.

Next chapter discusses issues related to performance behaviour, critical parts of the application and optimization techniques. Chapter 3 presents the most significant results of applying the techniques described in chapter 2 and how and what to measure in applications. Finally, the conclusions are presented.

2 Analysing the Performance Behaviour

Since the application is built in JAVA (also limited below to JAVA2 technology) and, consequently, cannot use assembly code, the optimizations that are suggested will be at a data structure level (faster and/or that takes less memory) and at iterative cycles.

2.1 Profiling an Application

In order to find out the critical parts of application performance, it was used the JProfiler tools, which allows analysing the life cycle of the application and seeing what are the methods/variables that take more resources. After the first result appears, the application was divided in two important stages to optimize: loading a file and applying visual transformations.

Next, the results of the most critical methods/variables used by the application are showed.

Results.
Memory instances/used (while loading a file):

- char[];
- String;
- Vector;
- int[];
- float[].

Processor occupation (while loading a file):

- elements().

Memory instances/used (while applying transformations):

- int[];
- draw shape methods.

Processor occupation (Wwhile applying transformations):

- draw shape methods;
- setColor();
- elementAt();
- ComputeXY().

2.2 Optimization Techniques

There are several techniques to apply in order to optimize the code of an application [2]. These can be split in two fundamental stages: machine independent, where the optimizations will be at programming language in general, and machine dependent, where the optimizations will concern about operating system, processor and memory.

2.2.1 Machine Independent

The first step, in general, to optimize the JAVA code of an application is to take a look in to the code to find out if there is any possibility to replace some variables, methods, cycles and data structures with others more efficient.

Let us take a look to the code of the method *elements* (used to transform a SVG node in a graphical element) of the SVG Viewer and make changes to exemplify some optimization techniques.

```java
public static void elements(Node nd, Scene scene) {
  if ((nd.getNodeName()).equals("svg")) {
    OsvSVG svg = Parser.svg(nd);
    if (svg.hasViewbox()) {
        scene.translate( -svg.getViewboxX(), -svg.getViewboxY());
        scene.scale((int)svg.getX(),(int)svg.getY(),
                    (svg.getWidth()/svg.getViewboxWidth()));
    }
    Node child = nd.getFirstChild();
    while(child!=null){
        Parser.elements(child,scene);
        child = child.getNextSibling();
    }
  }
  (…)
  if ((nd.getNodeName()).equals("polyline")) {
    try {
        Polyline pl = Parser.polyline(nd);
        scene.addNode(pl);
    } catch (OutOfMemoryError er){
         System.out.println("Erro de memoria ao criar objecto num:: ?");
         System.exit(0);
    }
  }
  if ((nd.getNodeName()).equals("desc")) {
    Node child = nd.getFirstChild();
    scene.setDesc(child.getNodeValue().trim());
  } else
    scene.setDesc("Description not available");
 (…)
}
```

Fig. 1. Java code of *elements* method.

Avoid Method Call Inside Loops.

Observe that method *nd.getNodeName()*, in Fig 1, is called several times as a test condition to create different elements and never changes during the method. When this occurs is more efficient declare a string with the *nd.getNodeName()* value and test that string because that method will be called only once (Fig. 2). The same can be applied to methods *att.getNodeName()* and *att.getNodeValue()* , as shown below in Fig. 4.

```java
    public static void elements(Node nd, Scene scene) {
       String t_elem = nd.getNodeName();
        if (t_elem.equals("svg")) {
                  (…)
               }
               (…)
        if (t_elem.equals("polyline")) {
             (…)
        }
        if (t_elem.equals("desc")) {
                    (…)
               }
               (…)
    }
```

Fig. 2. Java code of *elements* method with optimizations.

Eliminate Loop Inefficiencies.

First thing to say about loops in JAVA is that it is better to use *for* than *while*, because *for* allows to declare a variable that will be used just in the loop body and then destroyed when leaves the loop, while using *while* a test variable has to be declared before the loop and could be used, by mistake, later and could be in memory for a long time.

```
public static void elements(Node nd, Scene scene) {
  if ((nd.getNodeName()).equals("svg")) {
     OsvSVG svg = Parser.svg(nd);
     if (svg.hasViewbox()) {
       scene.translate( -svg.getViewboxX(), -svg.getViewboxY());
       scene.scale((int)svg.getX(),(int)svg.getY(),
                   (svg.getWidth()/svg.getViewboxWidth()));
     }
     for (Node child = nd.getFirstChild(); child != null;
          child = child.getNextSibling())
        Parser.elements(child,scene);
     }
     (…)
}
```

Fig. 3. Java code of *elements* method with optimizations.

Now let us look at next figure that is the code of method polyline (that is used in *elements* to create a polyline).

```
    public static Gpolyline polyline(Node node) {
       GPolyline obj = new Gpolyline();
       NamedNodeMap attributes = node.getAttributes();
       for (int i=0; i< attributes.getLength(); i++) {
           Node att = attributes.item(i);
                  String name = att.getNodeName();
           String value = att.getNodeValue();
           if (name.equals("points")) {
              String sep = new String(", ");
              Vector cords = new Vector();
              StringTokenizer st = new StringTokenizer(value, sep);
              while (st.hasMoreTokens()){
                  String cord = new String(st.nextToken());
                  cords.addElement(cord);
              }
              obj.np = cords.size()/2;
              obj.px = new float[cords.size()/2];
              obj.py = new float[cords.size()/2];
              for (int j=0; j<cords.size(); j++){
                  String cxy = new String((String)cords.elementAt(j));
                  obj.str2pts(cxy, j);
              }
           } else if (name.equals("stroke")) {
              String stroke = new String(value);
              obj.setStrokeColor(stroke);
           } else if (name.equals("fill")) {
              String fill = new String(value);
              obj.setFillColor(fill);
           } else if (name.equals("style")) {
              String style = new String(value);
              Parser.setStyle(obj, style);
           }
       }
       obj.compile();
       return obj;
    }
```

Fig. 4. Java code of *polyline* method.

Observe that methods *attributes.getLength()* and *cords.size()* are called as test conditions of the *for* loops. Because the lengths of string *attributes* and vector *cords* do not changes as the loop proceeds, we could compute that sizes only once.

```
public static Gpolyline polyline(Node node) {
            GPolyline obj = new Gpolyline();
            NamedNodeMap attributes = node.getAttributes();
             int atlen = attributes.getLength();
            for (int i=0; i<atlen; i++) {
                 (…)
                        obj.np = coords.size()/2;
                        obj.px = new float[obj.np];
                        obj.py = new float[obj.np];
                        for (int j=0; j<(obj.np*2); j++){
                           String cxy = new String((String)cords.elementAt(j));
                           obj.str2pts(cxy, j);
                        }
               (…)
}
```

Fig. 5. Java code of *polyline* method with optimizations.

Avoid New Object Creations.

As we can see in Fig. 4., every time the method *polyline* needs to send a string to other methods, a new instance of string is created. Sometimes it is not necessary to do that and, that way, avoid the process of creating objects.

```
   public static GPolyline polyline(Node node) {
            GPolyline obj = new GPolyline();
            NamedNodeMap attributes = node.getAttributes();
             int atlen = attributes.getLength();
            for (int i=0; i<atlen; i++) {
               Node att = attributes.item(i);
               String name = att.getNodeName();
               String value = att.getNodeValue();
               if (name.equals("points")) {
                        Vector coords = new Vector();
                        StringTokenizer st = new StringTokenizer(value, ", ");
                        while (st.hasMoreTokens())
                                   coords.addElement(st.nextToken());
                        obj.np = coords.size()/2;
                        obj.px = new float[obj.np];
                        obj.py = new float[obj.np];
                        for (int j=0; j<(obj.np*2); j++)
                           obj.str2pts((String)coords.elementAt(j), j);
               } else if (name.equals("stroke")) {
                        obj.setStrokeColor(value);
               } else if (name.equals("fill")) {
                        obj.setFillColor(value);
               } else if (name.equals("style")) {
                        Parser.setStyle(obj, value);
               }
            }
            obj.compile();
            return obj;
   }
```

Fig. 6. Java code of *polyline* method with optimizations.

Use Basic Data Structures.
Every time that the number of objects to save is known, we should use arrays instead Vector, HashTable and other dynamic data structures. Due to its dynamism, some data structures require processing time to increase or decrease its capacity.

2.2.2 Machine Dependent

Some programming languages, like C, allow the programmer to code at very low level using assembly. This feature allows programming and taking control of some operations and functions at processor and memory level, which can be faster than functions used by native languages. It is possible to write code to a specific platform to explore all its capacities.

But with JAVA this is not possible. JAVA is a high level platform independent language and is the JAVA Virtual Machine (JVM) that converts this bytecode to a native binary code.

3 Experimental Results

The experimental results of applying the above techniques were based on the time that SVG Viewer takes to load a file and transform an image. Those tests were made in three different machines, with different processors and memory using time-of-day measurements.

What to Measure?
To provide great precision of measuring, many processors contain a timer that operates at the clock cycle level, a clock cycle counter. A special register IS incremented every clock cycle and can be accessed by specific machine instructions. But not all processors (including PDAs processors) have such counters. As a result, there is no uniform, platform-independent interface by which programmers can make use of these counters. It is possible to create a small program interface for any specific machine using assembly to perform this task, but while using JAVA it would be impossible.

A simpler way to measure the execution time of a program is to query the system clock and register the time that the program execution takes from the beginning to the end; this is called *time-of-day measurement.* The JAVA method *System.currentTimeMillis()* returns the difference, measured in milliseconds, between the current time and midnight, January 1, 1970 UTC and lets programmers know the time taken by a given method to execute.

```
long start = System.currentTimeMillis();

//method to measure

long end = System.currentTimeMillis();
double tot = (end - start)/1000.0; //In seconds
System.out.println("\nTime past in seconds : " + tot);
```

Fig. 7. Java code to get the system time in milliseconds.

This is a very portable solution (just the system clock, common in all systems, is used) but its accuracy depends on how the clock is implemented and this varies from system to system. This variation occurs, because some operating systems (like Linux) use cycle counters to implement its time function, while others (like Windows) use interval counting to implement the same function. In theory, this kind of measurement under Windows produces low accuracy, especially for short duration (time intervals shorter than 200ms), but the available examples require longer execution times.

How to Present Results?

The measurements were made with several different size SVG files in three different systems: a Pentium III 1000MHz with 512MB of RAM under Windows XP Professional; an AMD K6-2 400MHz with 384MB of RAM under Windows XP Professional; and in a Pocket LOOX 600 Fujitsu/Siemens 400MHz processor under Windows PocketPC 2002. The application was compiled with jdk1.1.8.

Fig. 8. shows the difference between the time that SVG Viewer takes to load a file before and after the optimization techniques[1].

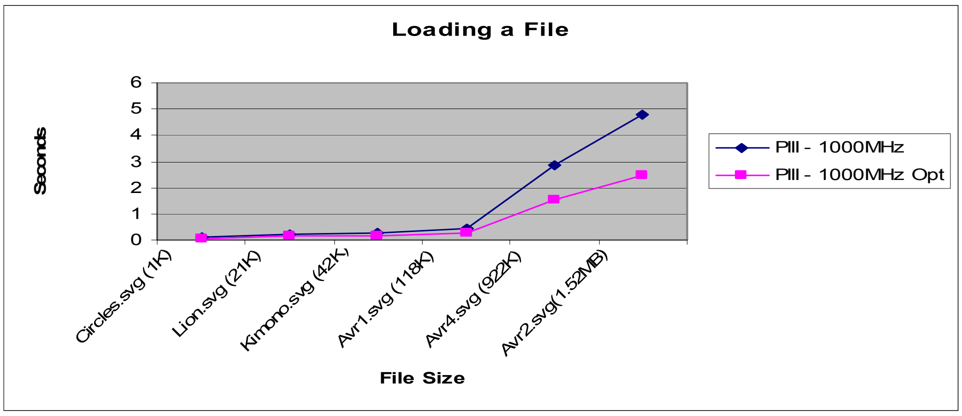

Fig. 8. Differences of performance in a PIII processor.

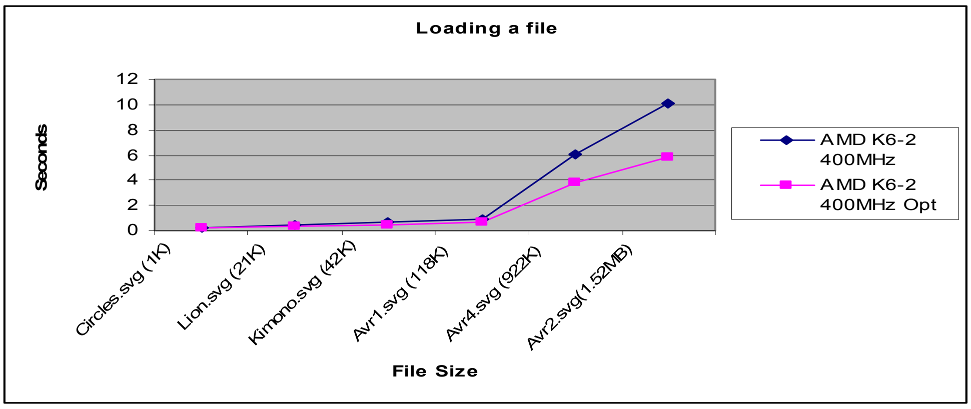

Fig. 9. Differences of performance in an AMD processor.

[1] For further detail insight this work, including code listings, see http://gec.di.uminho.pt/micei/ac0304/icca04/w3-pda.zip or see the entire project in http://opensvgviewer.sourceforge.net/home.php

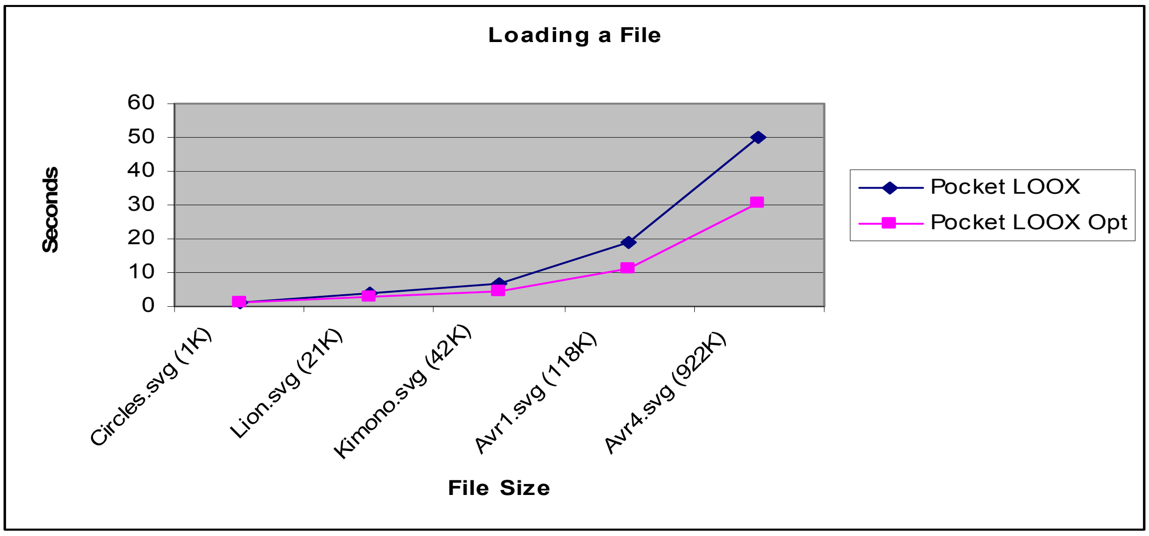

Fig. 10. Differences of performance in a PDA Intel 400MHz processor.

4 Conclusions

As a final conclusion, it is easy to notice (see the figures) that the optimization techniques applied to the initial code improves the application efficiently in different platforms. Using only machine independent techniques the application takes almost less 50% to load a file with time-of-day measurements, namely with larger files. Some of these techniques were applied to transformation functions, in the application, but the results were not faint. However, it is possible to conclude that parts of application may still need further optimizations, namely machine dependent ones.

Due to the limitations of JAVA at low level code, it was not possible to know at what point machine dependent techniques would affect the application performance. Which I guess it would be very high.

References

[1] Faria, Nuno André S.: Relatório de Estágio – Visualizador de Mapas para PDAs em SVG, Departamento de Informática - Universidade do Minho, 2002.

[2] Bryant, Randal E., O'Hallaron, David R.: Computer Systems A Programmer's Perspective, Prentice Hall, 2002.