Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications (Extended Abstract)

Jorge Navas,[1] Mario Méndez-Lojo,[1] Manuel V. Hermenegildo[1,2]

[1] Dept. of Computer Science, University of New Mexico (USA)
[2] Dept. of Computer Science, Tech. U. of Madrid (Spain) and IMDEA-Software

1 Introduction

Many space applications such as sensor networks, on-board satellite-based platforms, on-board vehicle monitoring systems, etc. handle large amounts of data and analysis of such data is often critical for the scientific mission. Transmitting such large amounts of data to the remote control station for analysis is usually too expensive for time-critical applications. Instead, modern space applications are increasingly relying on autonomous on-board data analysis.

All these applications face many resource constraints. A key requirement is to minimize energy consumption. Several approaches have been developed for estimating the energy consumption of such applications (e.g. [3, 1]) based on measuring actual consumption at run-time for large sets of random inputs. However, this approach has the limitation that it is in general not possible to cover all possible inputs. Using formal techniques offers the potential for inferring *safe* energy consumption bounds, thus being specially interesting for space exploration and safety-critical systems.

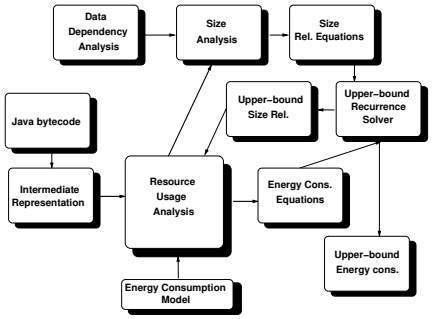

Fig. 1. Energy Consumption Framework

We have proposed and implemented a general framework for resource usage analysis of Java bytecode [2]. The user defines a set of resource(s) of interest to be tracked and some annotations that describe the cost of some elementary elements of the program for those resources. These values can be constants or, more generally, *functions of the input data sizes*. The analysis then statically derives an upper bound on the amount of those resources that the program as a whole will consume or provide, also as functions of the input data sizes. This article develops a novel application of the analysis of [2] to inferring safe upper bounds on the energy consumption of Java bytecode applications. We first use a resource model that describes the cost of each bytecode instruction in terms of the joules it consumes. With this resource model, we then generate energy consumption cost relations, which are then used to infer safe upper bounds. How energy consumption for each bytecode instruction is measured is beyond the scope of this paper. Instead, this paper is about how to infer safe energy consumption estimations assuming that those energy consumption costs are provided. For concreteness, we use a simplified version of an existing resource model [1] in which an energy consumption cost for individual Java opcodes is defined.

```
import java.lang.Stream;
public class SensorNet {
 public StringBuffer collectData(
                      Sensor sensors[]){
   int i;
   int n= sensors.length;
   StringBuffer buf = new StringBuffer();
   for (i=n; i > 0;i--){
     String data = sensors[i].read();
     buf.append(data);
   }
   return buf; }
 interface Sensor { String read();}
 class TempSensor implements Sensor {
   @Cost("10*size(ret)")
   public native String read(); }
 class SeismicSensor implements Sensor {
   @Cost("20*size(ret)")
   public native String read(); }
}
```

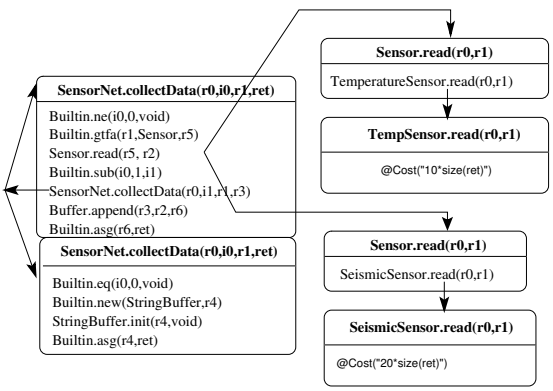

Fig. 2. Motivating example (Java source code and CFG)

2 Energy Consumption Analyzer

For space reasons, we will illustrate the overall energy consumption analyzer through a working example. The Java program in Fig. 2 emulates the process of collecting data from an array of sensors within a sensor network for further processing and sending to a remote control station. For simplicity, we only show the collecting process of the sensor network. The source code is provided here just for clarity, since the analyzer works directly on the corresponding bytecode. The sensor network is implemented by class `SensorNet` and defines the method `collectData` that receives an array of sensors (`Sensor`), reads from each one the data observed, and stores it in a buffer (`StringBuffer`) for further processing. There are two types of sensors: `TempSensor`, which takes simple temperature measurements, and `SeismicSensor` which records motions of the ground. The length of the buffer which the method ultimately produces depends on the size of data measured by the sensors.

Library methods including builtins (assignment `asg`, field dereference `gtf`, method invocations `invokevirtual`, etc.) have been annotated such that our analyzer can associate energy consumption costs with them using the cost model of [1]. The objective of the analysis is then to approximate the energy consumption of the whole program. Additionally, Java programmers can define *native* methods to represent methods with absence of any callee code to analyze. In the example, the energy consumption of reading data from `TempSensor` and `SeismicSensor` sensors is proportional (10 and 20 μJ/character, respectively) to the number of characters read. This domain knowledge is reflected by the programmer in the native methods that are ultimately responsible for reading (`TempSensor.read` and `SeismicSensor.read`), by adding the annotations `@Cost("10*size(ret)")` and `@Cost("20*size(ret)")`. The rest of this section describes the different steps applied by the analyzer to approximate the energy consumption of the program depicted in Fig. 2. The main components of the framework are shown in Fig. 1.

Step 1: Constructing the Control Flow Graph. The analysis translates the Java bytecode into an intermediate representation building a Control Flow Graph (CFG). Edges in the CFG connect *block methods* and describe the possible flows originated from conditional jumps, exception handling, virtual invocations, etc. A (simplified) version of the CFG corresponding to

our code example is also shown in Fig. 2. The `for` loop has been transformed into a recursion and the original `collectData` method has been compiled into two block methods that share the same signature: class where declared, name (`SensorNet.collectData`), and number and type of the formal parameters. The bottommost box represents the base case and the sibling corresponds to the recursive case. The virtual invocation of `read` has been transformed into a static call to a block method named `Sensor.read`. There are two block methods which are compatible in signature with that invocation, and which serve as proxies for the intermediate representations of the interface implementations in `TempSensor.read` and `SeismicSensor.read`. The annotations have been carried through the CFG and are thus available to the analysis.

Step 2: Inference of Data Dependencies and Size Relationships. The algorithm infers in this phase *size relationships* between the input and the output formal parameters of every block method. In this example, the size of (the contents of) a variable is its value. Note that for other type of variables we have also defined different ways of measuring its size. The following equations are inferred by the analysis for the two `SensorNet.collectData` block methods:

$$size_{ret}(s_{r_0}, s_{i_0}, s_{r_1}) \leq \begin{cases} 1 & \text{if } s_{i_0} = 0 \\ s_{r_2} + size_{ret}(s_{r_0}, s_{i_0} - 1, s_{r_1}) & \text{if } s_{i_0} > 0 \end{cases}$$

The size of the returned value *ret* is independent from the sizes of the input parameter *this* (s_{r_0}) but not from the length s_{i_0} of the array *sensors* (i_0 and r_1 respectively in the graph). Such size relationships are computed based on *dependency graphs*, which represent data dependencies between variables in a block, and user annotations if available. The equation system must be approximated by a recurrence solver in order to obtain a closed form solution. In this case, our analysis yields the solution $size_{ret}(s_{r_0}, s_{i_0}, s_{r1}) \leq s_{r_2} \times s_{i_0}$.

Step 3: Energy Consumption Analysis. In this phase, the analysis uses the CFG, the data dependencies, and the size relationships inferred in previous steps in order to infer energy consumption equations for each block method in the CFG and further simplify the resulting obtaining closed form solutions (in general, approximated –upper bounds). Therefore, the objective of the analysis is to statically derive safe upper bounds on the energy that each of the block methods in the CFG consumes. The result given by our analysis for the energy consumption of reading the array of sensors (`SensorNet.collectData`) is

$$cost_{collectData}(s_{r_0}, s_{i_0}, s_{r_1}) \leq \begin{cases} 241 & \text{if } s_{i_0} = 0 \\ 20 \times s_{r_2} + 487 + & \text{if } s_{i_0} > 0 \\ cost_{collectData}(s_{r_0}, s_{i_0} - 1, s_{r_1}) & \end{cases}$$

i.e., the energy consumption is proportional to the length of the array of sensors (`sensors` in the source, i_0 in the CFG), and the size s_{r_2} of observed data (r_2 in the CFG). Again, this equation system is solved by a recurrence solver, resulting in the closed formula $cost_{collectData}(s_{r_0}, s_{i_0}, s_{r_1})$-$\leq 20 \times s_{r_2} \times s_{i_0} + 487 \times s_{i_0} + 241$.

References

1. S. Lafond and J. Lilius. Energy Consumption Analysis for Two Embedded Java Virtual Machines. *J. Syst. Archit.'07*.
2. J. Navas, M. Méndez-Lojo, and M. Hermenegildo. Customizable Resource Usage Analysis for Java Bytecode. *TR-CS-2008-02*, UNM.
3. C. Seo, S. Malek, and N. Medvidovic. An Energy Consumption Framework for Distributed Java-based Systems. *ASE '07*.