Safe Upper-bounds Inference of Energy Consumption for Java Bytecode Applications

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

[1]University of New Mexico (USA)
[2]Technical University of Madrid (Spain) and IMDEA-Software

Newport News April 30, 2008

Motivation

- Many space applications handle a large amount of data and its analysis if often critical for the underlying scientific mission
- Transmitting data to the remote control station is usually too expensive
- Instead, modern space applications are increasingly relying on autonomous on-board data analysis
- Examples of these applications can be: sensor networks, on-board satellite-based platforms, on-board vehicle monitoring system, etc.
- In all these applications there are many resource constraints.

Motivation

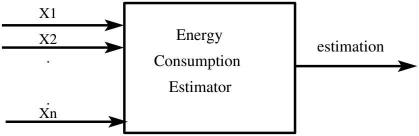
- Many space applications handle a large amount of data and its analysis if often critical for the underlying scientific mission
- Transmitting data to the remote control station is usually too expensive
- Instead, modern space applications are increasingly relying on autonomous on-board data analysis
- Examples of these applications can be: sensor networks, on-board satellite-based platforms, on-board vehicle monitoring system, etc.
- In all these applications there are many resource constraints.

A key requirement is to **minimize energy consumption**

Related Work

In current systems the estimation of energy consumption is inferred at *run-time* generating often large sets of random inputs $S = \{X_1, \ldots, X_n\}$ for calculating the energy consumption.

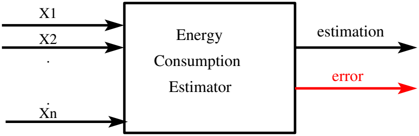
Φ : true energy consumption
$\hat{\Phi}(S)$: estimated (from S) energy consumption



Related Work

In current systems the estimation of energy consumption is inferred at *run-time* generating often large sets of random inputs $S = \{X_1, \ldots, X_n\}$ for calculating the energy consumption.

Φ : true energy consumption
$\hat{\Phi}(S)$: estimated (from S) energy consumption



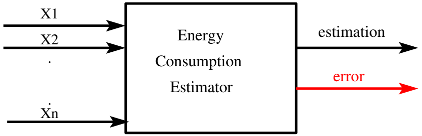
Disadvantages:

- S is finite
- There is always a potential error $(\hat{\Phi}(S) - \Phi)$

Related Work

In current systems the estimation of energy consumption is inferred at *run-time* generating often large sets of random inputs $S = \{X_1, \ldots, X_n\}$ for calculating the energy consumption.

Φ : true energy consumption
$\hat{\Phi}(S)$: estimated (from S) energy consumption



Disadvantages:

- S is finite
- There is always a potential error $(\hat{\Phi}(S) - \Phi)$

But safety critical systems require more **formal techniques** for inferring a safe energy consumption estimation.

Our Approach

We propose a fully automated analysis that infers **safe upper bounds** on the energy consumption in terms of input data sizes for Java (bytecode) applications **at compile time**

1. Define *energy consumption model* $\mathcal{M}$ [LL07] that describes the *upper bound cost* of each bytecode inst. in terms of joules it consumes:

Opcode	Inst. Cost in μJ	Mem. Cost in μJ	Total Cost in in μJ
`iadd`	.957860	2.273580	3.23144
`isub`	.957360	2.273580	3.230.94
...	...	...	...

2. With this resource model, we then generate energy consumption cost equations using resource usage analysis [NMLH08] which are then solved returning safe, upper bound *energy cost functions*.

Generate *cost equation system* by abstracting the iterative constructs (loops and recursion), and by inferring *size relations* between the arguments.

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
    }
}
```

Generate *cost equation system* by abstracting the iterative constructs (loops and recursion), and by inferring *size relations* between the arguments.

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
    }
}
```

$\mathcal{S}_{ret}(0) = 1$

Generate *cost equation system* by abstracting the iterative constructs (loops and recursion), and by inferring *size relations* between the arguments.

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

$\mathcal{S}_{ret}(0) = 1$

$\mathcal{S}_{ret}(s_n) = s_n \times \mathcal{S}_{ret}(s_n - 1)$

Generate *cost equation system* by abstracting the iterative constructs (loops and recursion), and by inferring *size relations* between the arguments.

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
    }
}
```

$\mathcal{S}_{ret}(0) = 1$

$\mathcal{S}_{ret}(s_n) = s_n \times \mathcal{S}_{ret}(s_n - 1)$

Equation systems can be often solved by usually using *difference equation solvers*, thus obtaining a *closed form* solution.

$$\left.\begin{array}{rcl} \mathcal{S}_{ret}(0) & = & 1 \\ \mathcal{S}_{ret}(s_n) & = & s_n \times \mathcal{S}_{ret}(s_n - 1) \end{array}\right\} \Rightarrow \mathcal{S}_{ret}(s_n) = s_n!$$

Generate *cost equation system* by abstracting the iterative constructs (loops and recursion), and by inferring *size relations* between the arguments.

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
    }
}
```

Generate *cost equation system* by abstracting the iterative constructs (loops and recursion), and by inferring *size relations* between the arguments.

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
    }
}
```

$$E_{fact}(0) = \mathcal{M}_{n=0}(\mu J)$$

Generate *cost equation system* by abstracting the iterative constructs (loops and recursion), and by inferring *size relations* between the arguments.

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

$E_{fact}(0) = \mathcal{M}_{n=0}(\mu J)$

$E_{fact}(s_n) = \mathcal{M}_{n>0}(\mu J) + E_{fact}(s_n - 1)$

Generate *cost equation system* by abstracting the iterative constructs (loops and recursion), and by inferring *size relations* between the arguments.

```
public int fact(int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n - 1);
  }
}
```

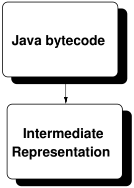
$$E_{fact}(0) = \underbrace{\mathcal{M}_{n=0}}_{a}(\mu J)$$

$$E_{fact}(s_n) = \underbrace{\mathcal{M}_{n>0}}_{b}(\mu J) + E_{fact}(s_n - 1)$$

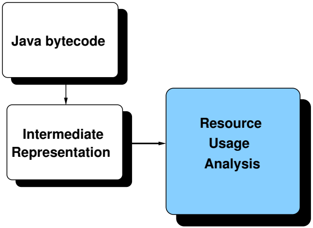
Equation systems can be often solved by usually using *difference equation solvers*, thus obtaining a *closed form* solution.

$$\left.\begin{array}{lcl} E_{fact}(0) & = & a \\ E_{fact}(s_n) & = & b + E_{fact}(s_n - 1) \end{array}\right. \Rightarrow \quad E_{fact}(s_n) = a + (b \times s_n)\ (\mu J)$$

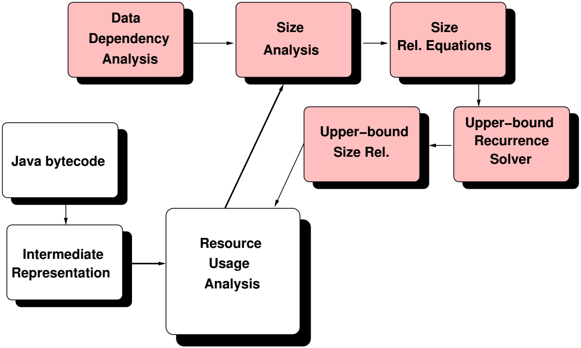
Energy Consumption Framework



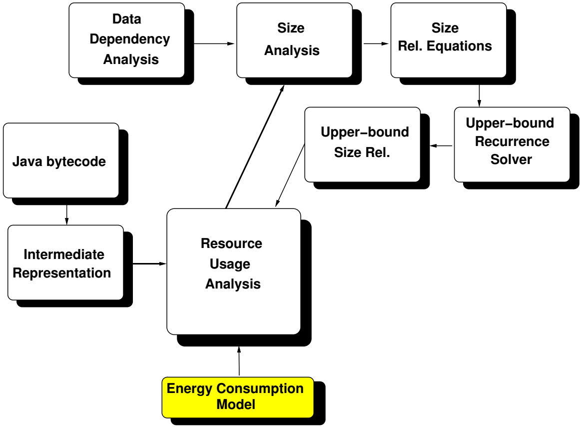
Energy Consumption Framework



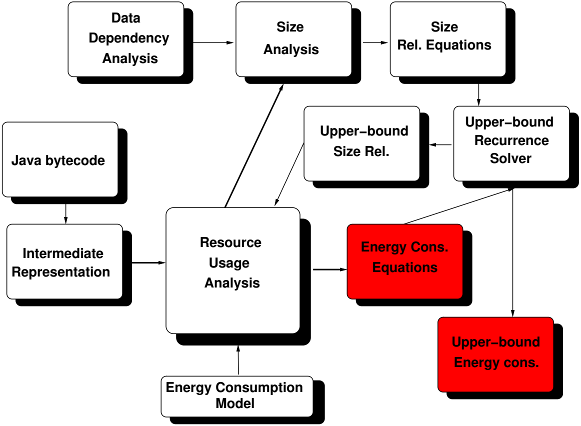
Energy Consumption Framework



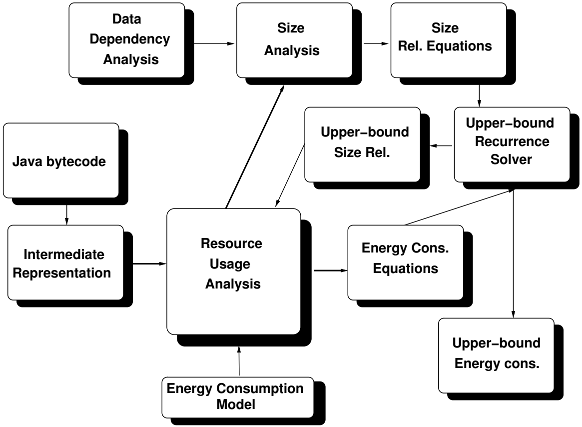
Energy Consumption Framework



Energy Consumption Framework



Energy Consumption Framework



Example: Java Program

```
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(); }
}
```

Example: Upper-bound Inference of Energy Consumption

1. Inference of size relationship equations:

$$\mathcal{S}_{ret}(s_{this}, s_i, s_{sensors}) \leq \begin{cases} 1 & \text{if } s_i = 0 \\ s_{data} + \mathcal{S}_{ret}(s_{this}, s_i - 1, s_{sensors}) & \text{if } s_i > 0 \end{cases}$$

2. Solving size relationship equations:

$$\mathcal{S}_{ret}(s_{this}, s_i, s_{sensors}) \leq s_{data} \times s_i$$

3. Inference of energy consumption equations:

$$E_{collectData}(s_{this}, s_i, s_{sensors}) \leq \begin{cases} 241 & \text{if } s_i = 0 \\ 20 \times s_{data} + 487 + & \text{if } s_i > 0 \\ E_{collectData}(s_{this}, s_i - 1, s_{sensors}) & \end{cases}$$

4. Solving energy consumption equations:

$$E_{collectData}(s_{this}, s_i, s_{sensors}) \leq (20 \times s_{data} \times s_i) + (487 \times s_i) + 241$$

Conclusions

- We have defined and implemented an energy consumption analysis that:
  - infers relatively accurate safe upper bounds
  - is independent from the Energy Consumption Model $\mathcal{M}$
  - supports a reasonable set of data-structures (trees, arrays, lists, etc.) and standard Java libraries used in real applications
  - covers a good range of complexity functions $(O(1), O(log(n)), O(n), O(n^2) \ldots, O(2^n), \ldots)$ and different types of structural recursion such as simple, indirect, and mutual.
- Many potential improvements (e.g., supporting more complex data-structures, more sophisticated data size metrics, etc.).

Questions ?

Bibliography

Sébastien Lafond and Johan Lilius.
Energy consumption analysis for two embedded java virtual machines.
J. Syst. Archit., 53(5-6):328–337, 2007.

J. Navas, M. Méndez-Lojo, and M. Hermenegildo.
Customizable Resource Usage Analysis for Java Bytecode.
Technical report, UNM, 2008.