Schedulability analysis for Java finalizers

Thomas Bøgholm, René R. Hansen, Anders P. Ravn,
Bent Thomsen, and Hans Søndergaard

CISS, Aalborg University
VIA University College, Horsens

JTRES, August 2010

Introduction

- Finalizers for Java
- Unpredictable
  - executed at garbage collection
    - in a separate thread
  - May not be executed at all

Introduction

- Finalizers for Java
- Unpredictable
  - executed at garbage collection
    - in a separate thread
  - May not be executed at all

Java finalizers in real-time systems

- Java finalizers are not suitable for real-time systems
  - which task will pay for execution time?
  - exactly *when* will clean-up code run?
  - how do we account for finalization in schedulability analysis?
- Finalizers are discouraged in RTSJ
  - and not allowed in SCJ

Java finalizers in real-time systems

- Java finalizers are not suitable for real-time systems
  - which task will pay for execution time?
  - exactly *when* will clean-up code run?
  - how do we account for finalization in schedulability analysis?
- Finalizers are discouraged in RTSJ
  - and not allowed in SCJ

Clean-up mechanism

- We do need a clean-up mechanism for real-time Java
  - must be executed even if exceptions are thrown
- Example situations:
  - Java wrapping up C-code using malloc/free
  - clean-up of temporary files/buffers
  - dealing with hardware

Strategies

- finalizers (Java)
  - unpredictable
- try/finally
  - *the alternative*
- destructors (C++)
  - predictable

try/finally

Manual clean-up

```
try{
  // init code
} finally{
  // clean−up code
}
```

- Why is this bad?
- unnatural programming style
- try/finally has to be written every time a class is used
  - also for derivatives
- An opportunity to make mistakes
  - clean-up without try/finally?
  - no clean up at all?
- Should it be the job of the programmer to remember to do clean-up?
  - then we depend on documentation and communication

try/finally

Manual clean-up

```
try{
  // init code
} finally{
  // clean−up code
}
```

- Why is this bad?
- unnatural programming style
- try/finally has to be written every time a class is used
  - also for derivatives
- An opportunity to make mistakes
  - clean-up without try/finally?
  - no clean up at all?
- Should it be the job of the programmer to remember to do clean-up?
  - then we depend on documentation and communication

Java Finalizers / C++ Destructors

- centralized clean-up
- natural programming style
- executed automatically
  - also in the case of exceptions
- is written once
  - is inherited
  - *may* be overridden by derivatives
- C++ destructors are predictable
  - executed when a stack-allocated object goes out of scope
  - executed at explicit deallocation

Making finalizers suitable

- Predictable clean-up mechanism for real-time Java
  - destructor-like finalizers for real-time Java
- Using task-private scoped memory makes finalizers behave like destructors
- Developers no longer need to do clean-up manually

Our profiles make finalizers predictable?

- Predictable Java
  - Similar to SCJ
- Disciplined use of scoped memory
  - a task is responsible for finalizing private objects
- We are able to include finalizers in schedulability analysis

Simple WCET analysis

- Each event handler has a private memory
- Object initializations are registered during handler execution
- Object finalization, will be performed after handler execution
  - in WCET analysis, simply add the cost of finalizers for created objects

Response time analysis

$$R_i = C_i + \sum_{j \in hp(i)} \lceil R_i / T_j \rceil C_j$$

$$C_i = WCET_i + \sum_{k \in WCObj(i)} WCETfinalizer_k$$

Response time analysis

$$R_i = C_i + \sum_{j \in hp(i)} \lceil R_i / T_j \rceil C_j$$

$$C_i = WCET_i + \sum_{k \in WCObj(i)} WCETfinalizer_k$$

Automated schedulability analysis

- SARTS: a tool for automated schedulability analysis
  - from byte-code to timed automata models
  - model-checking using UPPAAL
- Includes finalizers
  - for all program paths, only finalizers of objects actually created are considered
- Simple extension to existing tool

Conclusion

- Finalizers can be both useful and predictable

```
public void finalize(){
  //cleanup
}
```

```
try{
  a = new ...
  b = new ...
  //code
} finally{
  a.cleanup();
  b.cleanup();
}
```

- Natural style of programming
- Less boilerplate-code
- Less possibilities for program errors

Conclusion

- Can easily be included in response time analysis
- Finalizers could be allowed in SCJ
- This might be possible in RTSJ as well