

Supporting resource-awareness in managed runtime environments

Inti Gonzalez Herrera

PhD's thesis

Supervised by: Olivier Barais and Johann Bourcier

Jury: Isabelle Puaut, Vivien Quéma, Gilles Muller, Gaël Thomas, Laurent Réveillère

Diverse – IRISA/University of Rennes 1

December 14, 2015



Developers embrace managed runtime environments



Managed runtime environments (MRTes) boost productivity

- Portability of Applications
- Dynamic Code Loading (supporting the hypothesis of open world)
- Automatic Memory Management (usually with a Garbage Collector)
- Improved Error Handling

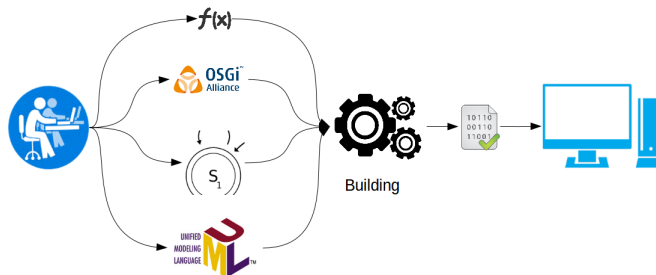
MRTes



High usability vs Limited Control

Focus on Ideas

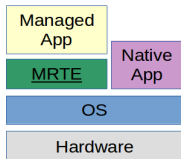
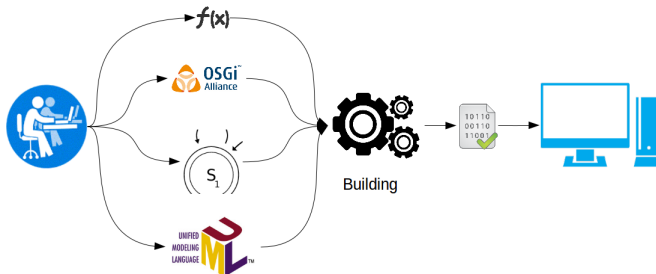
- Use high-level abstractions
- Forget about details such as resources



High usability vs Limited Control

Focus on Ideas

- Use high-level abstractions
- Forget about details such as resources



In the “software stack” we trust

- OSs abstract the hardware
- OSs multiplex resource
- Runtimes offer additional guarantees

It usually works!!!

But developers need control

Developers often need support for
finer-grain resource management

- 1 Developing middleware and Component Frameworks
- 2 Multi-tenant cloud systems
- 3 Smartphones and Embedded systems

A “traditional” **software stack** can be a
burden in these cases

- 1 Generic APIs might hinder performance
- 2 Safe APIs prevent collecting information and managing resources



CommitStrip.com

Consequences?

Suboptimal solutions

More control is needed!!!

The populous land of resource-aware applications

Resource-aware applications

- **Observe**
 - how different parts of applications consume resource?
 - how much resources are available?
- **Manage** the resource available
 - allocating and deallocating resource as needed
- **Modify their behavior**
 - to improve performance
 - to avoid critical failures

Resource-aware programming requires runtime support

We focus on supporting resource-aware programming by providing:

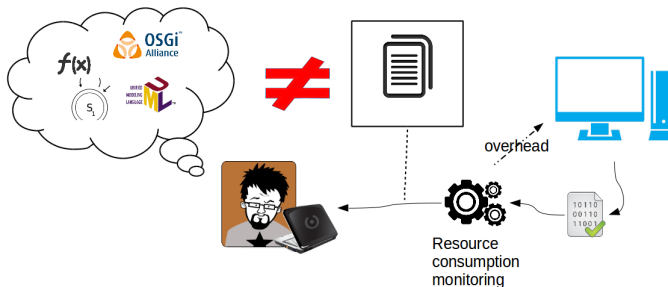
- ① Resource Consumption Monitoring
- ② Resource Reservation

Problems

Mismatch between the concepts used by developers and how tools present information

Developers define new abstractions to deal with specific domains, but development tools (e.g., profilers and debuggers) know nothing about such abstractions.

We need domain-specific development tools



We need efficient development tools

Resource Consumption Monitoring and Reservation in MRTes

OS Specific

low overhead

- Cgroups
- Overseer (HPC)
- Resource Containers
- Jails
- Jamus

Resource Consumption Monitoring and Reservation in MRTes

★ All resource types

OS Specific

low overhead

- Cgroups ★
- Overseer (HPC)
- Resource Containers ★
- Jails ★
- Jamus ★

Resource Consumption Monitoring and Reservation in MRTes

★ All resource types

OS Specific

low overhead

- Cgroups ★
- Overseer (HPC)
- Resource Containers ★
- Jails ★
- Jamus ★

M RTE Specific

low-medium overhead

- MVM
- KaffeOS
- Modified GC
- Dynamic Profiling ★
- OSGi Memory Profiling
- Tracing Objects

Resource Consumption Monitoring and Reservation in MRTes

★ Arbitrary structures

★ All resource types

OS Specific

low overhead

- Cgroups ★
- Overseer (HPC)
- Resource Containers ★
- Jails ★
- Jamus ★

M RTE Specific

low-medium overhead

- MVM
- KaffeOS
- Modified GC
- Dynamic Profiling ★★
- OSGi Memory Profiling
- Tracing Objects

Resource Consumption Monitoring and Reservation in MRTes

★ Arbitrary structures

★ All resource types

OS Specific

low overhead

- Cgroups ★
- Overseer (HPC)
- Resource Containers ★
- Jails ★
- Jamus ★

M RTE Specific

low-medium overhead

- MVM
- KaffeOS
- Modified GC
- Dynamic Profiling ★★
- OSGi Memory Profiling
- Tracing Objects

Application level (portable)

medium-high overhead

- JRes ★
- JRAF2
- Instrumentation-based monitoring ★★
- OSGi CPU Profiling
- Multi-tenant CPU isolation

Instrumentation-based monitoring is almost good
Can also be used as the foundation for reservation

Easing the construction of resource management tools

Relevance of approaches

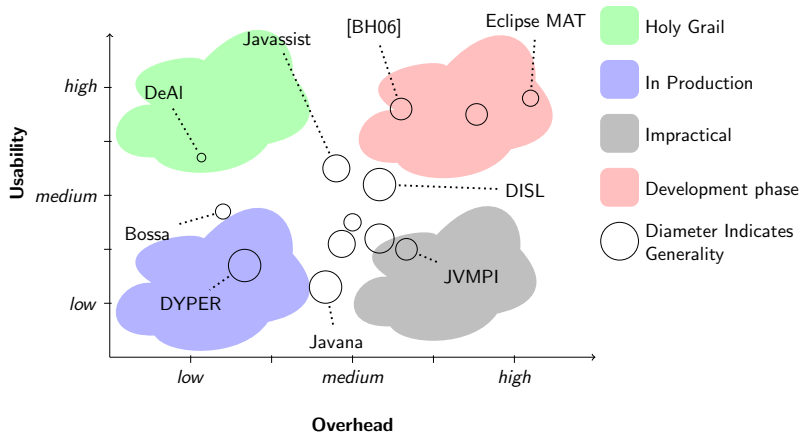


Fig.: Area of circumference indicates how general the approach is (the larger the better)

Synthesis

On resource consumption monitoring and reservation for MRTes

- There are many approaches with different overhead and accuracy
- Most approaches are not portable, but they offer low-medium overhead
- Portable approaches are not usable in production due to their high overhead

Easing the construction of resource management tools

- Most easy to use tools are useful during development, but they fail in production environments
- Most approaches are not usable enough nor they offer low performance overhead

Contributions

1 – Scapegoat

An adaptive resource consumption monitoring framework for component-based systems.

2 – Squirrel methodology

An architecture-driven approach to reduce the overhead of resource reservation by choosing the proper mapping to represent components in the runtime.

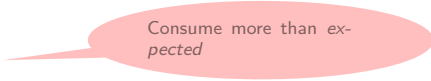
3 – A domain-specific language to define memory profilers

A generative approach to ease the construction of tools for supporting resource-aware programming. In particular, it support the construction of profilers that can be used at runtime.

Overview

What is the desired behavior?

- i Precise and lightweight
- ii Detect faulty components

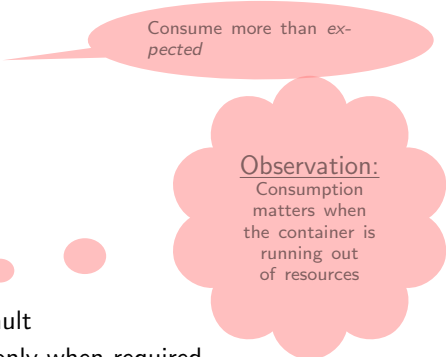


Consume more than expected

Overview

What is the desired behavior?

- i Precise and lightweight
- ii Detect faulty components



Consume more than expected

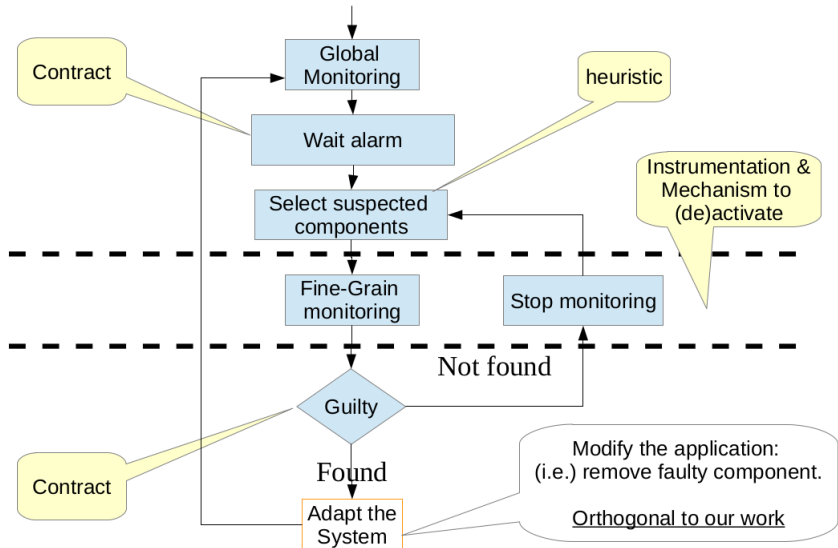
Observation:

Consumption matters when the container is running out of resources

How to achieve that goal?

- Global Monitoring by default
- Fine-Grained Monitoring only when required ...
 - i Assume some components are more likely to be faulty
 - ii Begin with suspected components

Main Loop



Contract-based approach

- ④ Contract for the whole system/JVM – when to trigger the alarm?
 - CPU Usage
 - Memory usage

Example (Contract specification for an application)

```
add node0 : JavaNode  
set node0.cpu_threshold = 80 %
```

Contract-based approach

- ❶ Contract for the whole system/JVM – when to trigger the alarm?
 - CPU Usage
 - Memory usage

Example (Contract specification for an application)

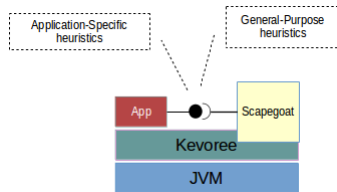
```
add node0 : JavaNode
set node0.cpu_threshold = 80 %
```

- ❷ Contract for each component – consumption allowed under certain operation conditions
 - Peak number of instruction per second
 - Maximum memory usage
 - Operation condition

Example (Contract specification for a component)

```
add node0.server : WsServer
set server.cpu_wall_time = 2580323 intr/sec
set server.memory_max_size = 15000 bytes
set server.throughput_all_ports = 10000 msg/sec # operation condition
```

Extending the framework with heuristics



Example

Number of previous failures:

Components are suspected if they have previously failed

Example

Models@runtime-based heuristic: Failures come from recent changes

- Recently added components
- Components that interact with recently added components

Mechanisms to perform monitoring

Global Monitoring

To perform global monitoring we use JMX

Memory Consumption Monitoring

- Instrumentation-based monitoring
- Using the JVMTI to explore the Java heap

Switching from global monitoring to fine-grain monitoring

The diagram illustrates the process of switching from global monitoring to fine-grain monitoring. It features two code snippets for a `method()` function, connected by two arrows. The left arrow is labeled 'activate' and points from the left code to the right code. The right arrow is labeled 'deactivate' and points from the right code back to the left code. The left code snippet is a standard Java method that creates a `List<Figure>`, adds a new `ArrayList<Figure>`, and iterates over it to create and add `Square` objects. The right code snippet is the same method but with instrumentation. It adds a call to `low-level-monitor.registerObject(l)` before the loop and a call to `low-level-monitor.registerObject(a)` inside the loop, where `a` is a `Square` object. Red dots highlight these new instrumentation calls in the right snippet.

```
void method() {  
    List<Figure> l =  
        new ArrayList<Figure>()  
    for (int i = 20 ; i < 40; i++) {  
        Square a = new Square(127.0)  
        l.add(a)  
    }  
}
```

activate

```
void method() {  
    List<Figure> l =  
        new ArrayList<Figure>()  
    low-level-monitor.registerObject(l)  
    for (int i = 20 ; i < 40; i++) {  
        Square a = new Square(127.0)  
        low-level-monitor.registerObject(a)  
        l.add(a)  
    }  
}
```

deactivate

Require Java Agent for:

- i Instrumenting bytecode
- ii Retransforming every class within a component

Research Questions

- RQ1 What is the impact of the various level of instrumentation?
- RQ2 What are the costs of using instrumentation-based memory monitoring and heap exploration?
- RQ3 Does adaptive monitoring outperform state-of-the-art instrumentation monitoring?
- RQ4 What is the impact of application size (i.e. number of components, size of components) and the quality of the heuristic?
 - a Execution time
 - b Time to discover faulty components

Overhead of the instrumentation solutions

RQ1:
What is the impact
of the various level of
instrumentation?

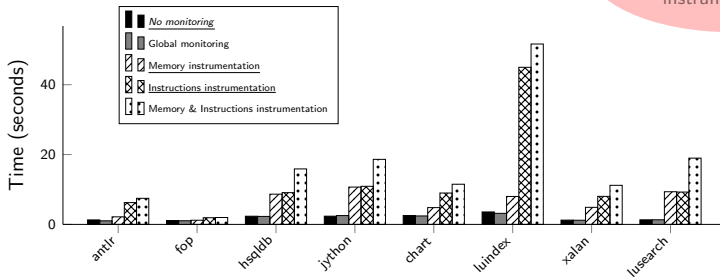


Fig.: Execution time for tests using the DaCapo Benchmark

Memory instrumentation, average overhead of 329%
Instruction instrumentation, average overhead of 562%

Instrumentation-based memory monitoring vs. heap exploration-based memory monitoring

RQ2:
What are the costs of
using
instrumentation-based
memory monitoring
and heap exploration?

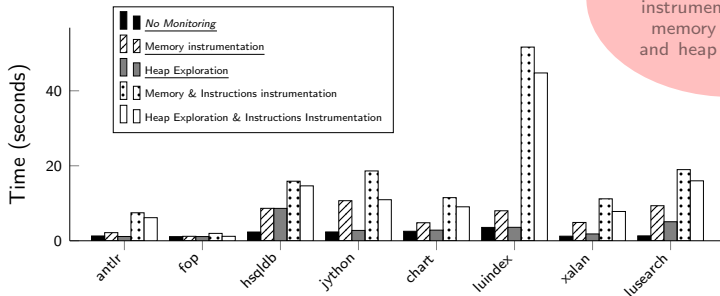


Fig.: Using different memory monitoring techniques

Memory Instrumentation => average overhead of 329%

Heap Exploration => average overhead of 179%

Overhead of Adaptive Monitoring vs Components Instrumented all the time

Experiment Setting

- Components of firefighter use case (13 components)
- One component that fails over and over
- Component with a job to complete – used to measure its **execution time**

Worst-case scenario:

Switching states

Global

Fine-Grain

Global

Fine-Grain

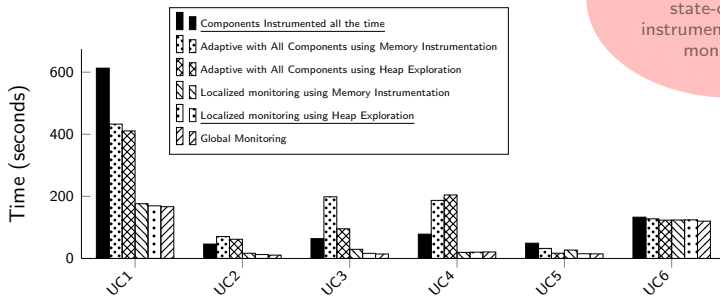
...

Table: Setting of use cases

Test Name	Monitored Resource	Faulty Resource	Heuristic	External Task
UC1	CPU, Memory	CPU	number of failures	Weka, train neural network
UC2	CPU, Memory	CPU	number of failures	dacapo, antlr
UC3	CPU, Memory	CPU	number of failures	dacapo, chart
UC4	CPU	CPU	number of failures	dacapo, xalan
UC5	CPU, Memory	CPU	less number of failures first	dacapo, chart
UC6	Memory	CPU	number of failures	Weka, train neural network

Overhead of Adaptive Monitoring vs Components Instrumented all the time

Experiment Results



RQ3:
Does adaptive monitoring outperform state-of-the-art instrumentation-based monitoring?

Components Instrumented all the time vs Adaptive Monitoring:
Overhead Reduced by 93% on average

Impact of application size and heuristic on execution time

RQ4a:

What is the impact of application size and the quality of the heuristic on execution time?

Experiment setup:

- Faulty component that fails over and over
- Component with a job to complete – to measure its execution time
- Variable number of components: 4, 8, 16, 32, 64, 128
- Perfect Heuristic: *Number of previous failures*

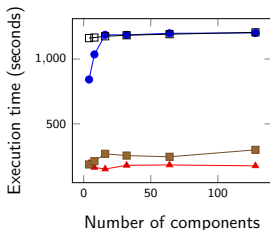


Fig.: 115 classes per component

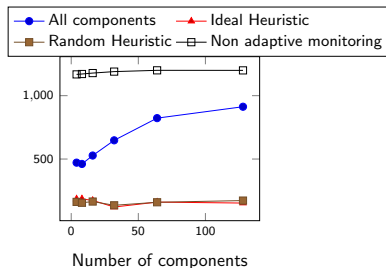


Fig.: Four classes per component

Heuristic has limited effect on execution time

Impact of application size and heuristic on delay time

RQ4b:

What is the impact of application size and the quality of the heuristic on time to discover failure?

Experiment setup:

- Same setting as before
- **Delay Time:** Time to discover faulty component.

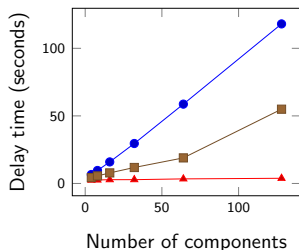


Fig.: 115 classes per component

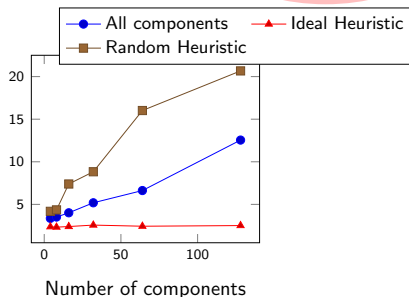


Fig.: Four classes per component

Delay Time highly affected by the heuristic

Summary

- ❶ Lightweight yet efficient monitoring system
 - Adaptive monitoring mode that only slowdowns parts of the application
 - Comparable delay time when a good heuristic is used
- ❷ Support the definition of application-specific heuristics. In this way developers can:
 - Leverages architectural information to drive the search for the faulty component
 - Define Models@runtime-based heuristics
- ❸ Monitoring overhead reduced by 93%

Contributions

1 – Scapegoat

An adaptive resource consumption monitoring framework for component-based systems.

2 – Squirrel methodology

An architecture-driven approach to reduce the overhead of resource reservation by choosing the proper mapping to represent components in the runtime.

3 – A domain-specific language to define memory profilers

A generative approach to ease the construction of tools for supporting resource-aware programming. In particular, it support the construction of profilers that can be used at runtime.

Squirrel – architecture-driven resource management

Several approaches to reserve resource

- i Support different resource types
- ii Impose different overhead

Squirrel – architecture-driven resource management

Several approaches to reserve resource

- i Support different resource types
- ii Impose different overhead

What approach to use?

- Delay the decision until deployment
- Use metadata on architecture to decide
- Use extensible component platform supporting:
 - i Many mappings from component to system-level abstractions
 - ii Each mapping knows how to manage some resource types

Assumption:
Components do not use resources in the same way.

Fact:
We only know components at deployment time

Squirrel – Extensible Component Platform

An implementation to assess the concept

During component framework design:

- ① Identify mappings from components to system-level abstraction
- ② For each mapping
 - i Build resource management mechanisms
 - ii Implement optimization for each mechanism
 - iii Evaluate mechanisms
 - iv Keep the set of most *efficient* mechanisms

At deployment time:

- ① What resources a component requires?
- ② Find the best mapping for each component using multi-objective optimization
- ③ Map components to system-level abstraction by wrapping components in resource-aware containers

An implementation to assess the concept

- 1 CPU reservation by mapping components to Linux's Cgroups
- 2 Memory Reservation using either Scapegoat or isolation components in different JVMs
- 3 Optimize communication and deployment time when components are isolated
 - Shared-memory channel for communication
 - Cloning processes to reduce start-up time

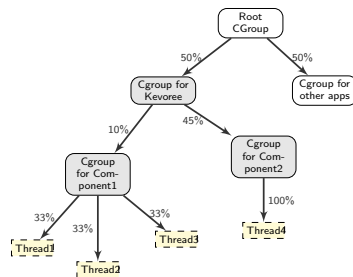


Fig.: Reserving CPU by mapping components to cgroups

Research Questions

- RQ1 What is the impact of the various resource management strategies?
- RQ2 When isolation of component is used as memory management strategy, is the deployment time reduced by using clones of the Kevoree runtime?
- RQ3 How is communication among components improved by using the proposed optimization?

Comparing resource management strategies

RQ1:
What is the impact
of the various
resource management
strategies?

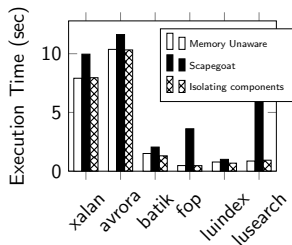


Fig.: CPU overhead during the execution of Dacapo benchmarks

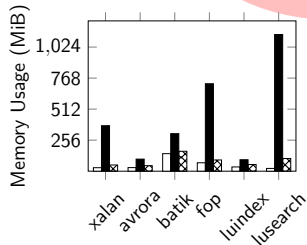


Fig.: Memory overhead during the execution of Dacapo benchmarks

CPU management produces no overhead
Memory management produces overhead: Scapegoat is worse

Importance of optimization for each mapping: starting time

RQ2:

When isolation of component is used as memory management strategy, is the deployment time reduced by using clones of the Kevoree runtime?

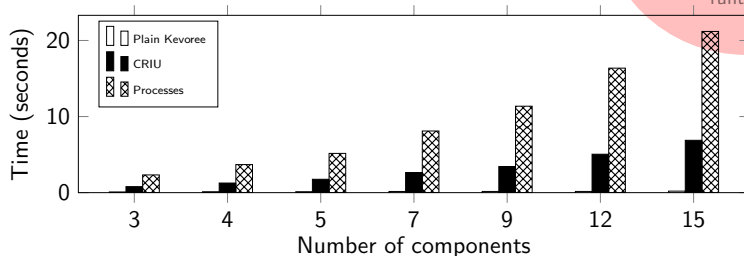


Fig.: Average deployment time per component using different strategies

cloning processes reduces starting time overhead from 4000% to 1900%

Importance of optimizations for each mapping: communication among isolates

RQ3:
How is
communication
among components
improved by using the
proposed
optimizations?

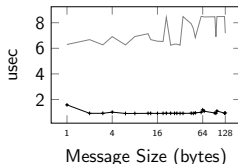


Fig.: Latency of IPC mechanisms

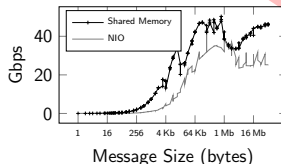


Fig.: Bandwidth of IPC mechanisms

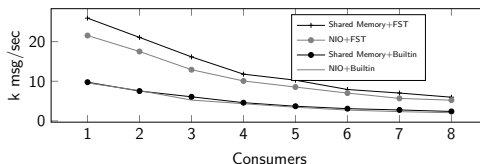


Fig.: Communication throughput for different channels

Proper channels reduce the cost of isolation in terms of communications

Summary

- A late binding approach to reduce the cost of resource management
- A prototype implemented using the Kevoree component framework
 - Different mapping for components
 - Different optimization strategies

Contributions

1 – Scapegoat

An adaptive resource consumption monitoring framework for component-based systems.

2 – Squirrel methodology

An architecture-driven approach to reduce the overhead of resource reservation by choosing the proper mapping to represent components in the runtime.

3 – A domain-specific language to define memory profilers

A generative approach to ease the construction of tools for supporting resource-aware programming. In particular, it support the construction of profilers that can be used at runtime.

Customized memory profilers

Software abstractions are at the core of development

- i Components, Domain-Specific Languages
- ii Development tools lack support for new abstractions
- iii Time consuming writing them from scratch


Customized memory profilers

Software abstractions are at the core of development

- i Components, Domain-Specific Languages
- ii Development tools lack support for new abstractions
- iii Time consuming writing them from scratch

Generative approach to the rescue

- Backend technology to perform the actual profiling
- What kind of information we want to compute?
- How we represent the great diversity of abstractions?



Observation:
In a MRTE all
abstractions are
represented by
objects in
memory

Number of objects reachable from a given object

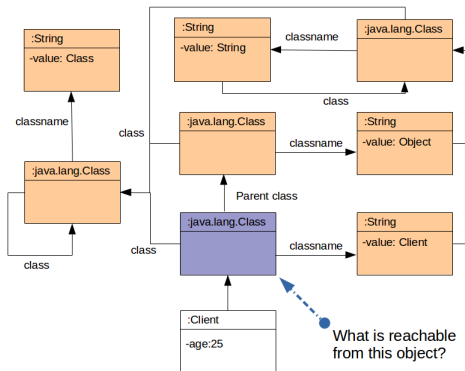


Fig.: Objects reachable from the Client class. Observe that only one object is not reachable.

This is a graph, and we are computing values in a subgraph

Length of singly linked lists

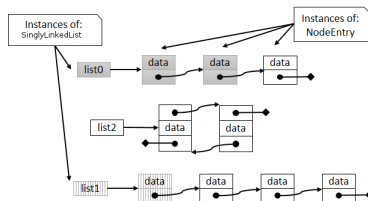


Fig.: Memory snapshot with three linked lists

$$f_{head}(O) = \begin{cases} head = O & O \text{ is SinglyLinkedList} \\ \exists x \in \text{Objects}, & x \text{ references } o \wedge f_{head}(x) \\ \text{false} & O \text{ is NodeEntry} \\ & \text{otherwise} \end{cases}$$

Other subgraphs (there are many lists)

A recursive equation determines whether an object is member of a subgraph

Idea for a domain-specific language to define memory profilers

- ① An instance of a software abstraction is represented as a collection of objects at runtime – a subgraph
- ② We call such subgraphs **structures**
- ③ The idea is to identify structures and to compute values on them.
 - Size
 - Number of Objects
 - How many objects in the structure met a first order predicate

The garbage collector knows how to traverse the whole graph – linear time

S1 For each object, checks if it is member of a **structure**

S2 If yes, perform a partial computation of the value associated to such a **structure**

Global Architecture

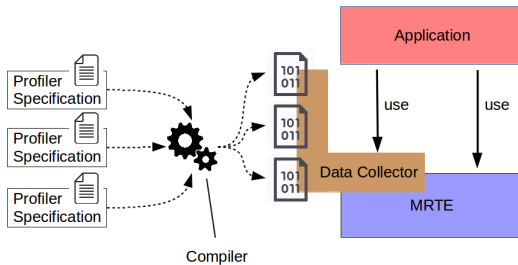


Fig.: In this case, three memory profilers are defined.

Declaring Types For Collecting Data

Concrete Syntax

```
Type ::= id ':' ('tableOf' id | 'struct' '{' (id ':' id)+ '}' )
```


Declaring Types For Collecting Data

Concrete Syntax

Type ::= **id** ':' ('tableOf' **id** | 'struct' '{' (**id** ':' **id**)⁺ '}')

Structures

```
// A simple structure
WantedObject : struct {
  class_name : string
  size : int
}
```

Declaring Types For Collecting Data

Concrete Syntax

Type ::= **id** ':' ('tableOf' **id** | 'struct' '{' (**id** ':' **id**)⁺ '}')

Structures

```
// A simple structure
WantedObject : struct {
  class_name : string
  size : int
}
```

Tables

```
// This new type is a table (i.e., list)
// of WantedObject
wanted : tableOf WantedObject
// In the language, values of this type
// have operations such as map and filter
```

Declaring Types For Collecting Data

Concrete Syntax

Type ::= **id** ':' ('tableOf' **id** | 'struct' '{' (**id** ':' **id**)⁺ '}')

Structures

```
// A simple structure
WantedObject : struct {
  class_name : string
  size : int
}
```

Tables

```
// This new type is a table (i.e., list)
// of WantedObject
wanted : tableOf WantedObject
// In the language, values of this type
// have operations such as map and filter
```

Built-In Types

- **int**
- **bool**
- **string**

Expressions

Basic

```
// simple arithmetic  
3 + i  
// boolean operators  
flag and (12 > a)  
// string to int  
"12".toInt() + 4  
12.toString() // int to string
```

Expressions

Basic

```
// simple arithmetic
3 + i
// boolean operators
flag and (12 > a)
// string to int
"12".toInt() + 4
12.toString() // int to string
```

Initializing structures and tables

```
// a table value that contains 1, 2, 3
#[ 1,2,3 ]
// a structure value WantedObject
struct WantedObject { "String", 12 + 3 }
// a list of WantedObject with one element
#[ struct WantedObject { "Integer", i*4 } ]
// an empty list of string
#string[] // need the type qualifier
```

Expressions

Basic

```
// simple arithmetic
3 + i
// boolean operators
flag and (12 > a)
// string to int
"12".toInt() + 4
12.toString() // int to string
```

Initializing structures and tables

```
// a table value that contains 1, 2, 3
#[ 1,2,3 ]
// a structure value WantedObject
struct WantedObject { "String", 12 + 3 }
// a list of WantedObject with one element
#[ struct WantedObject { "Integer", i*4 } ]
// an empty list of string
#string[] // need the type qualifier
```

Lambda Expressions

```
// keep three in the result
#[ 1,2,3 ].filter([it | it > 2])
// instances of class K3Object
objects.filter([it | it is K3Object]) // objects is a built-in value
// keep metadata of "big" objects
objects.filter([it | it > 1000]).map([it | struct WantedObject { it.classname, it.size }])
```

Defining structures in the heap

Concrete Syntax

```
structure ::= 'create structure foreach' id ':' exp 'using' body  
body ::= 'constructor' stmts 'membership' exp 'updates' stmts  
stmts ::= (id '=' exp) +
```

Number of instances of two classes

```
create structure foreach e:#[ "JFrame" ] using  
  constructor  
    initialObjects = #Object[] // built-in value  
    n = 0 // compute value n in this structure  
  membership this is JFrame // this is built-in value  
  updates n = n + 1
```

```
// defining another type of structure  
create structure foreach e:#[ "JPane" ] using  
  constructor  
    initialObjects = #Object[]  
    m = 0  
  membership this is JPane  
  updates m = m + 1
```

Developers View

Create software abstraction for other developers

- Component framework
- Library for a given domain
- Domain-specific language

Deliver software

Include domain-specific memory profilers along the *software abstraction* to ease its usage.

Native agent

JVMTI is used to traverse the graph of objects

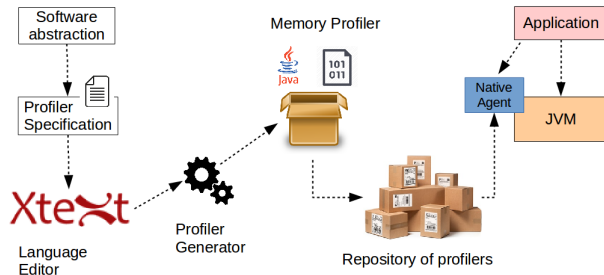
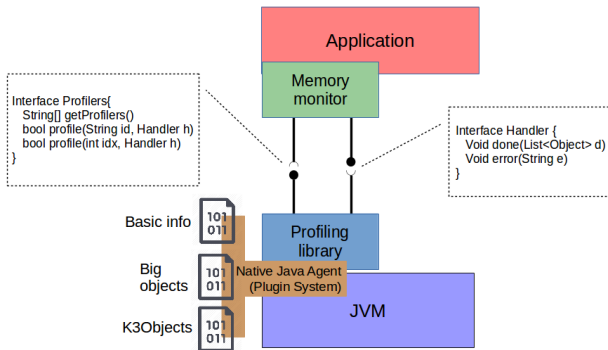


Fig.: Memory profilers are built from the description of software abstractions

Users View



Accessing profilers

- Memory profilers are black-boxes accessed through Java interfaces
- Data is collected in the form of plain Java objects

Research Questions

- RQ1 Is significant the difference between the time needed to execute a single analysis with our approach in comparison to previous solutions?
- RQ2 Does our approach produce profilers with lower overhead than state-of-the-art tools when used to perform many iterations of memory analysis at runtime?
- RQ3 Does the advantages of our approach remain for real applications?

Analysis Time

Example

```
create structure foreach e:#["jvm"], using
  constructor
    initialObjects = #[Object]
    exists = false
  membership true
  updates exists = exists or (this is UnusedClass)
```

RQ1:
Is significant the
difference between
the time needed to
execute a single
analysis with our
approach in
comparison to
previous solutions?

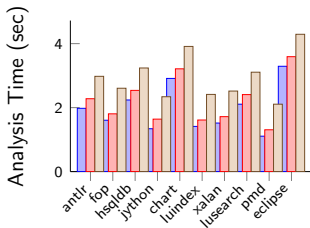


Fig.: Analysis time with default input size

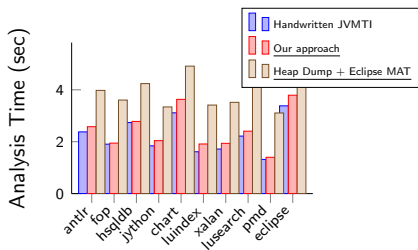


Fig.: Analysis time with large input size

Analysis time reduced between 25% and 39%

150+ sloc in handwritten solution
3+ sloc in Eclipse MAT

Does our approach produce profilers with lower overhead than state-of-the-art tools when used to perform many iterations of memory analysis at runtime?

Total Execution Time

Example

```
Info : struct { nbObjects : int, size : int }
create structure foreach e:#["whole-jvm"] using
  constructor
    initialObjects = threads
    data = struct Info { 0, 0 }
  membership ( referrer in this_structure )
  updates
    data = struct Info { data.nbObjects + 1 , data.size + this.size }
```

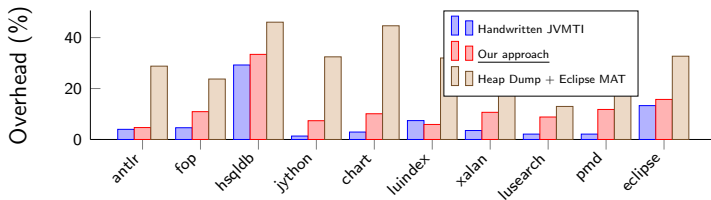


Fig.: Overhead on execution time compared to the execution without memory analysis

Average overhead is 12%

200+ sloc in handwritten solution
10+ sloc in Eclipse MAT

Memory consumption of OSGi bundles

RQ3:
Does the advantages
of our approach
remain for real
applications?

Example

```
create structure foreach e:classloaders using
  constructor
    initialObjects = #[e]
    size = 0
  membership ((ref_kind == root and this.class.classloader in this_structure)or
    (ref_kind != root and referrer in this_structure))
  updates
    size = size + this.size
```

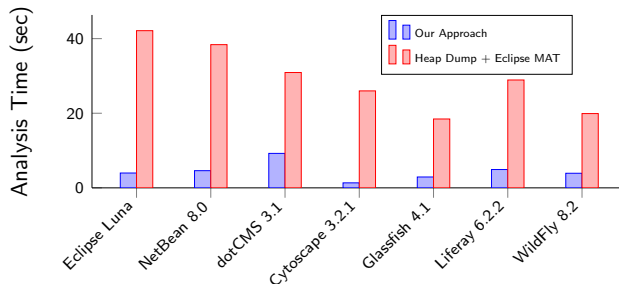


Fig.: Time needed to execute the analysis just once. The analysis finds the consumption of the top components

Summary

- DSLs should be delivered along the necessary tooling support in order to ease their use
- The mechanism we propose to describe memory profilers can be used by both the developers of DSLs and their users
- We present a DSL to define profilers and the tools to use it:
 - Generator of profilers that target JVM TI technology.
- The generated profilers impose less performance overhead than a previous approach

Conclusions

- Runtime support is required to implement resource-aware methods: resource accounting and reservation.
- Existing approaches impose high performance overhead and have limited capacity for dealing with arbitrary software abstractions.
- A framework, named Scapegoat, to efficiently compute per component resource utilization.
- A methodology to select a representation of each component in the runtime environment in such a way that resource reservations can be guaranteed with low performance overhead.
- A language to ease the definition of memory profilers that can be used in production environments.

Reducing overhead of instruction accounting

So far

Adaptive monitoring reduces the overhead of performing CPU consumption monitoring. Still, instrumenting components for monitoring imposes overhead.

Learn a predictive model for costly methods and avoid instrumenting them

- Create a training set by using the parameters of several invocations
- Build a simple predictive model
- If the model is good enough use it every time the method is invoked



- What kind of predictive model to use? It must be simple to train and evaluate
- What methods are worth considering when this technique is used?

A language to manipulate the graph of objects

Provided so far

A language to query the graph of objects in the memory heap

Manipulating the graph of objects seems useful

- A mechanism to remove stale references in OSGi applications is proposed in [ATM⁺15]. The approach is largely based on eliminating references between objects.



- Can we achieve the same without modifying the JVM by simply using an API such as JVMTI?
- What is the kind of interaction needed between the language and the rest of the MRTE?

Automatically generate memory profilers for DSLs

Provided so far

Developers of memory profilers must understand the DSL, and how its concepts are represented in memory. Then write the profiler by hand.

Can we automatically generate “*standard*” profilers?

- Identify a containment relationship as a *structure*
- Compute the size of structures
- Understand the relationships in memory and the classes used to represent concepts of the language?



- Can we do better than just providing a set of generic profilers?
- Can we abstract the target DSL?
- Is this feature useful at all?

Publications

- Inti Gonzalez-Herrera, Johann Bourcier, Erwan Daubert, Walter Rudametkin, Olivier Barais, François Fouquet, Jean-Marc Jézéquel: *Scapegoat: An Adaptive Monitoring Framework for Component-Based Systems*. WICSA 2014: 67-76
- Rima Al Ali, Ilias Gerostathopoulos, Inti Gonzalez-Herrera, Adrian Juan Verdejo, Michal Kit, Bholanathsingh Surajbali: *An Architecture-Based Approach for Compute-Intensive Pervasive Systems in Dynamic Environments*. HotTops@ICPE 2014: 3:1-3:6
- Inti Gonzalez-Herrera, Johann Bourcier, Walter Rudametkin, Olivier Barais, François Fouquet: *Squirrel: Architecture Driven Resource Management*. SAC 2016: [to appear]





Koutheir Attouchi, Gaël Thomas, Gilles Muller, Julia Lawall, and André Bottaro.

Incinerator - eliminating stale references in dynamic osgi applications.

In *Proceedings of the international conference on Dependable Systems and Networks, DSN '15*, page 11. IEEE Computer Society, 2015.



Walter Binder and Jarle Hulaas.

Flexible and efficient measurement of dynamic bytecode metrics.

In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE '06*, pages 171–180, New York, NY, USA, 2006. ACM.