Manipulating Managed Execution Runtimes to support Self-Healing Systems

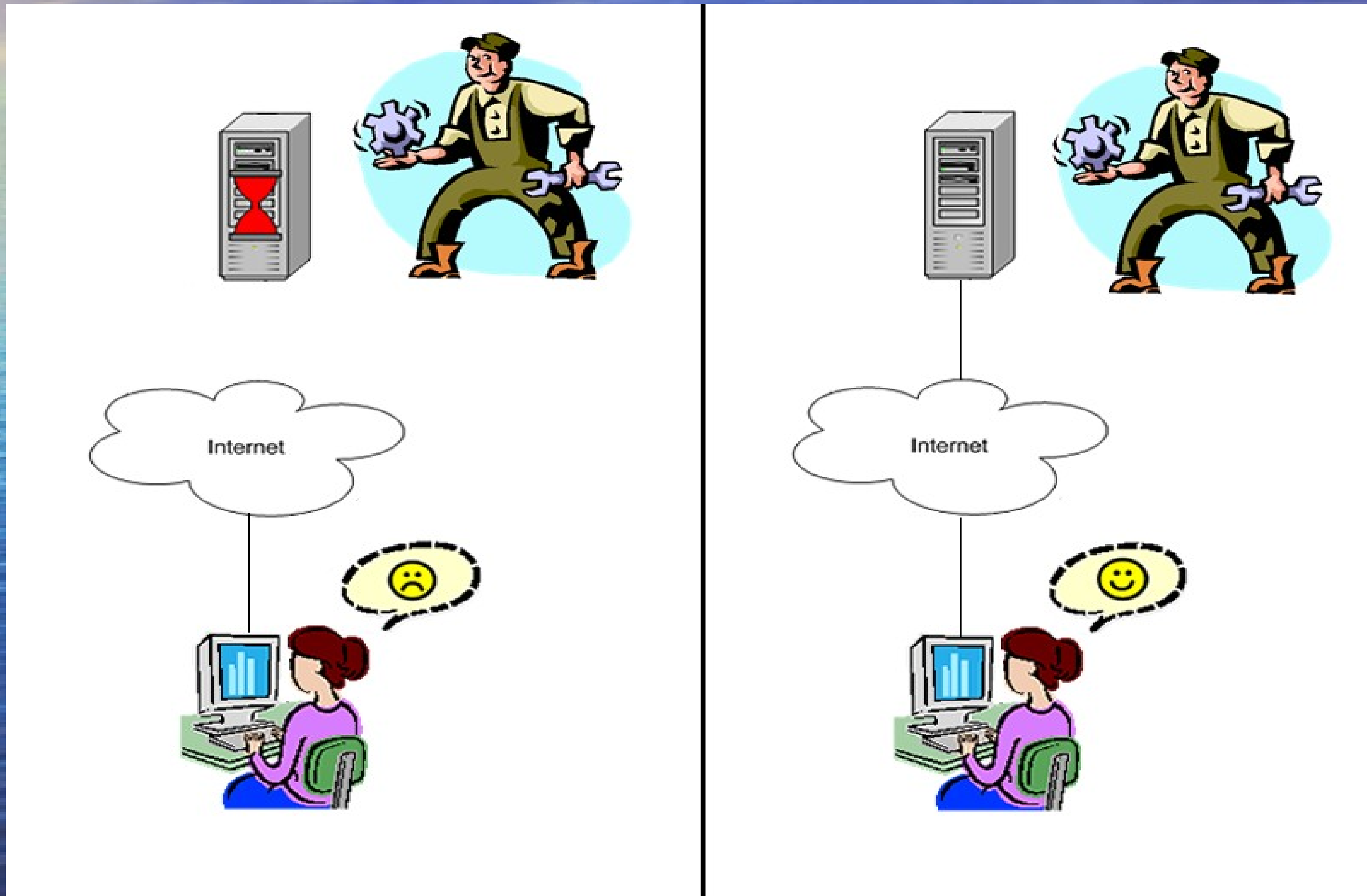
Rean Griffith‡, Gail Kaiser‡

Presented by Rean Griffith

rg2023@cs.columbia.edu

‡ - Programming Systems Lab (PSL) Columbia University

Introduction



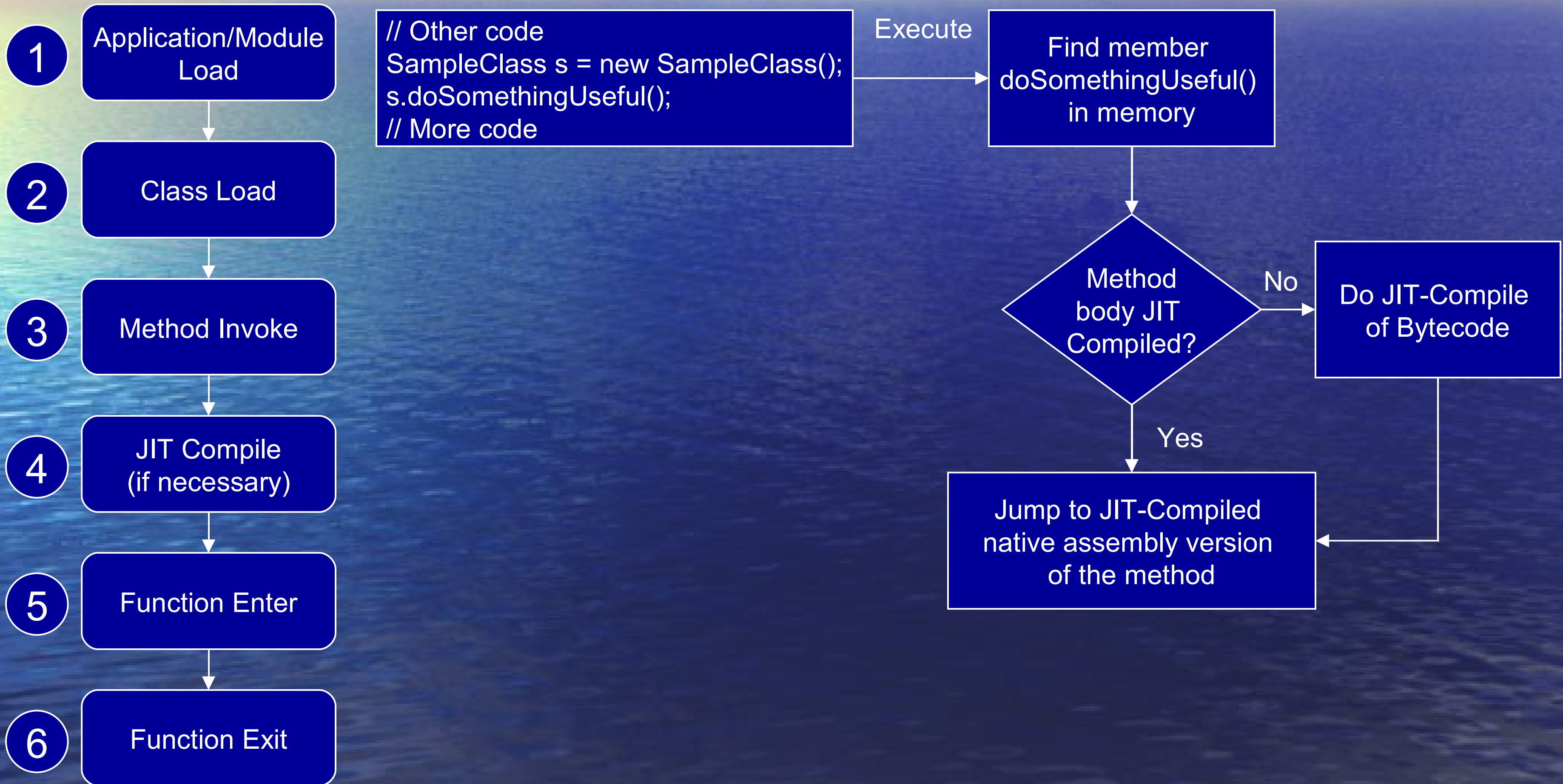
Overview

- Motivation
- Managed Execution Model
- System Architecture
- How it works
- Performing a repair
- Performance
- Conclusions & Future work

Motivation

- Managed execution environments e.g. JVM, CLR provide a number of application services that enhance the robustness of software systems, BUT…
- They do not currently provide services to allow applications to perform consistency checks or repairs of their components
- Managed execution environments intercept everything applications running on top of them attempt to do. Surely we can leverage this

Managed Execution Model



Runtime Support Required

Facility	CLR v1.1	JVM v5.0
The ability to receive notifications about current execution stage	Profiler API	JVMTI (no JIT)
The ability to obtain information (metadata) about the application, types, methods etc. from profiler	Yes	Yes
The ability to make controlled changes or extensions to metadata e.g. new function bodies, new type, type::method references	Fine-grained (full)	Coarse-grained (partial)
The ability to have some control over the JIT-Compilation process	Yes	No

System Architecture

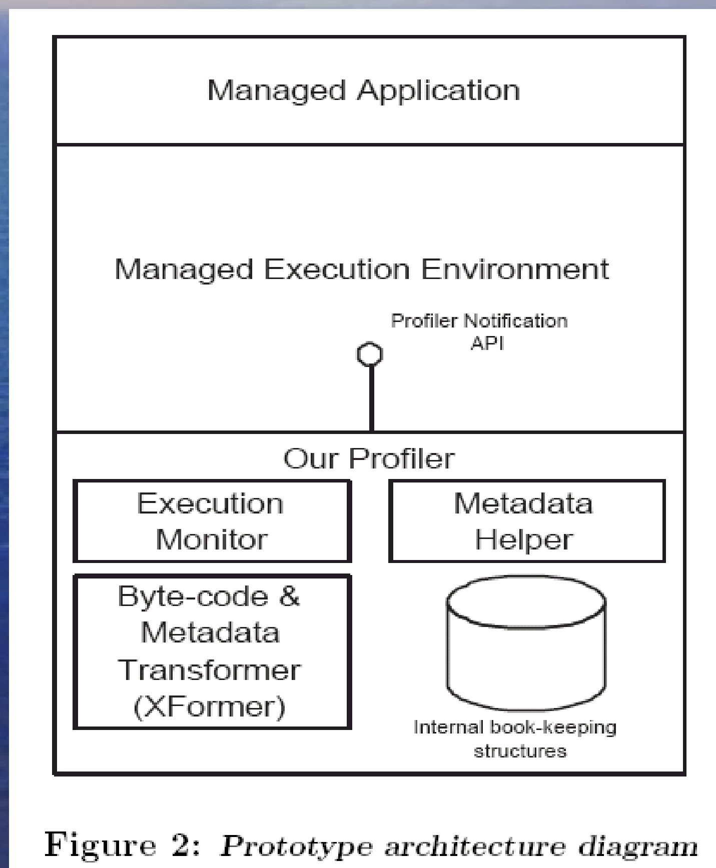
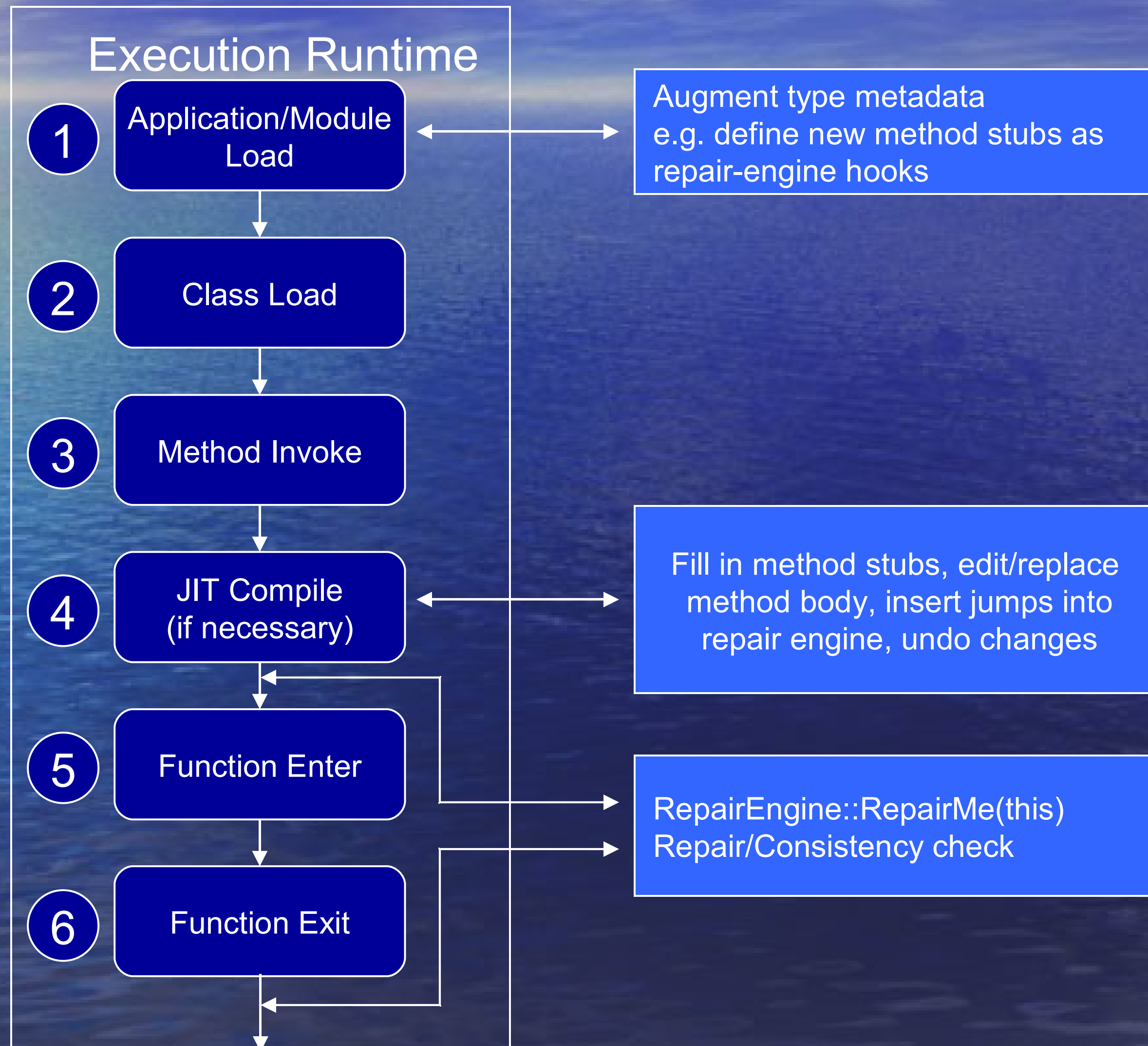


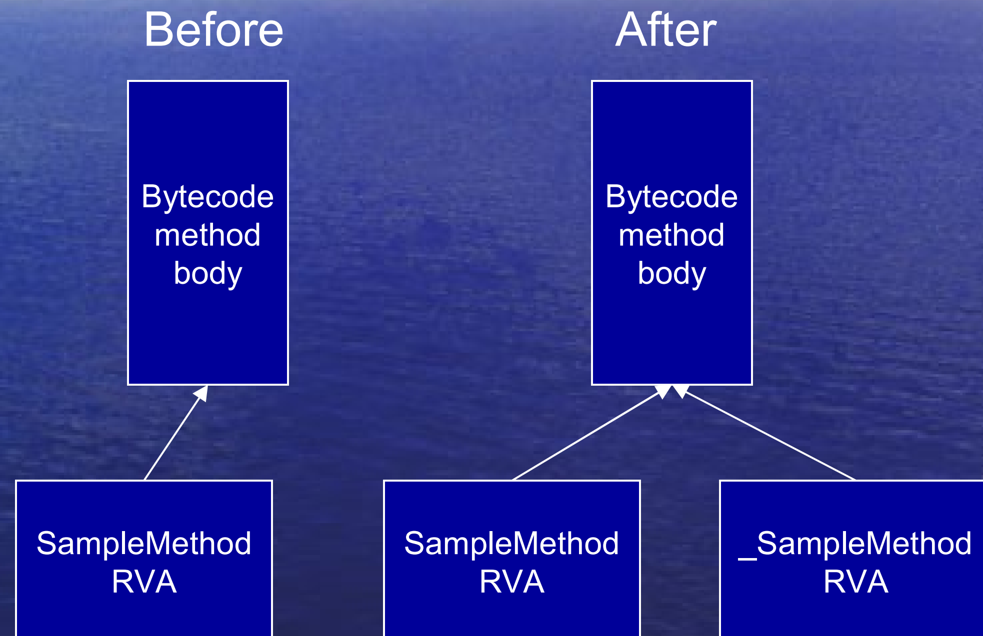
Figure 2: *Prototype architecture diagram*

Our Prototype's Model of Operation



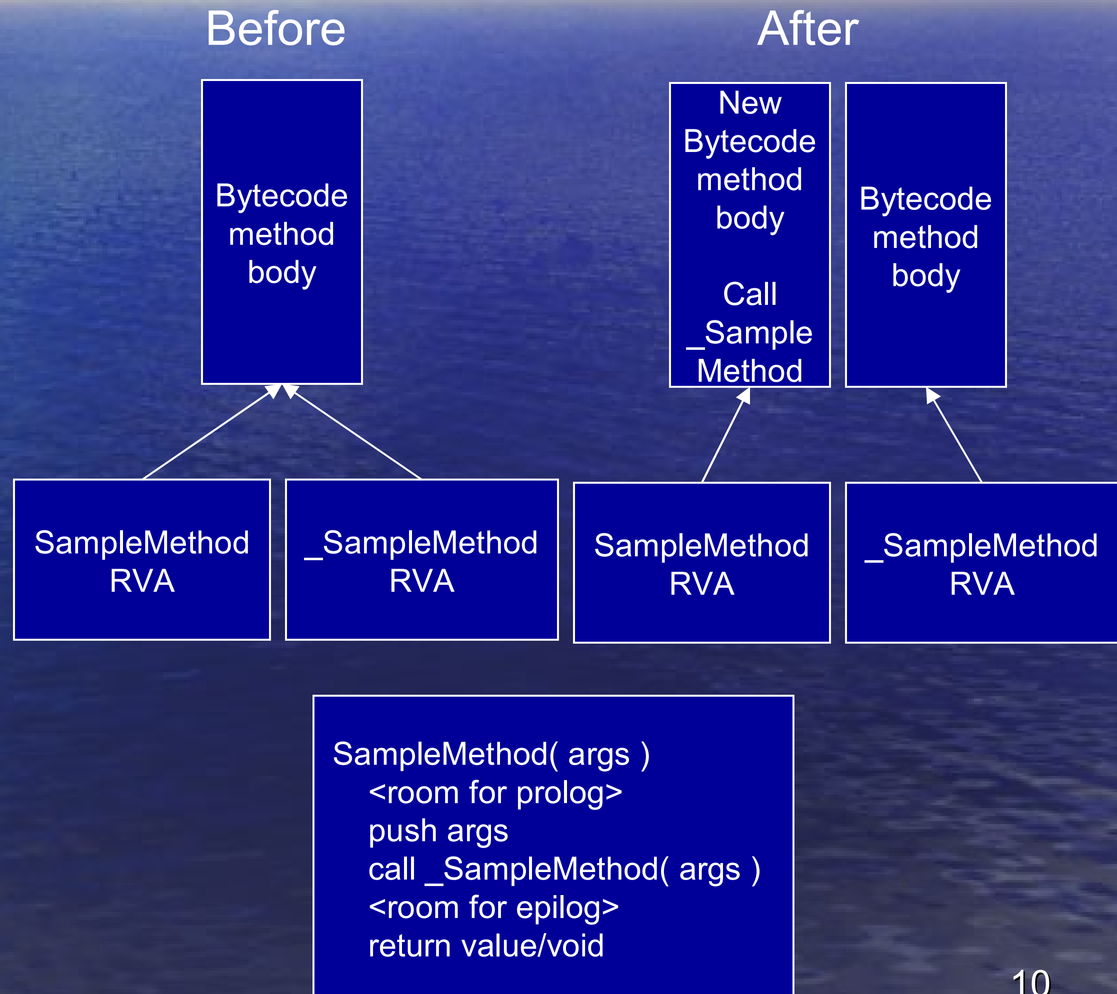
Phase I – Preparing Shadow Methods

- At module load time but before type definition installed
  - Extend type metadata by defining with new methods which will be used to allow a repair engine to interact with instances of this type



Phase II – Creating Shadow Methods

- At first JIT-Compilation
  - Define the body of the shadow method and re-wire some things under-the-hood



Performing a Repair

- Augment the wrapper to insert a jump into a repair engine at the *control point(s)* before and/or after a shadow method call

```
SampleMethod( args )
   RepairEngine::RepairMe(this)
   push args
   call _SampleMethod( args )
   RepairEngine::RepairMe(this)
   return value/void
```

Performance – No Repairs Active

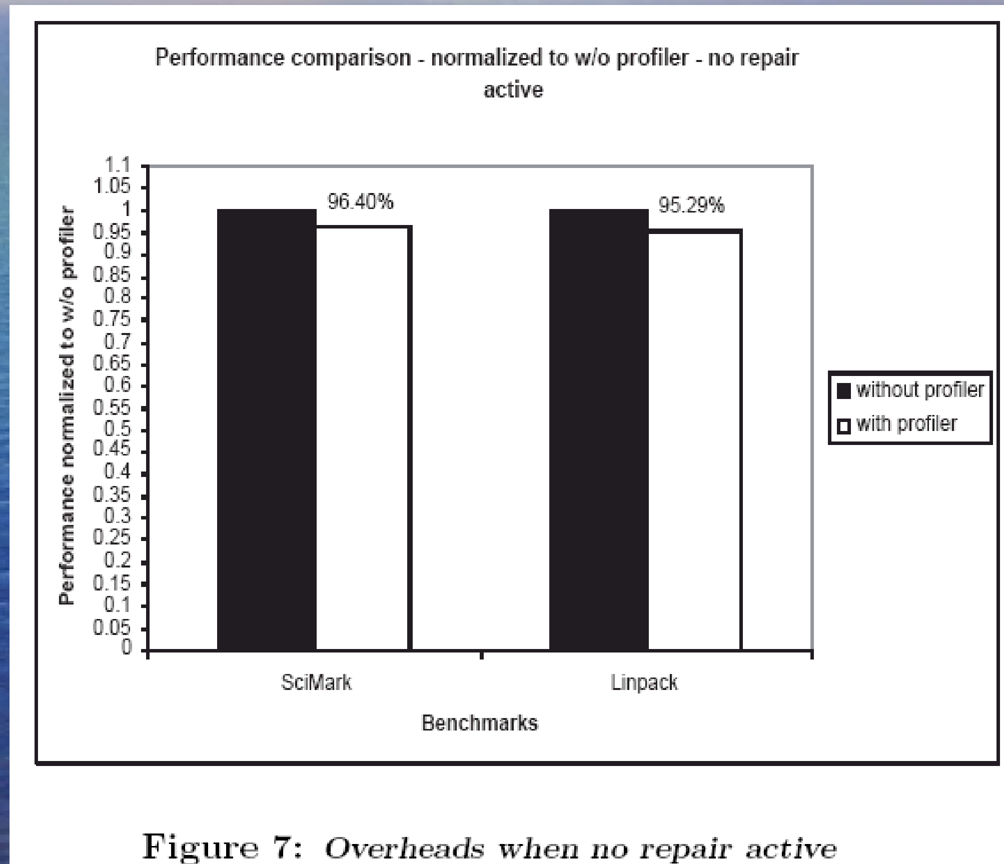


Figure 7: *Overheads when no repair active*

Overheads on the Managed Execution Cycle

Module Name	SciMark.exe
Module Load time (ms)	0.0230229
Module bind time (ms)	0.374817
# shadows prepared	2
Total shadow prepare time (ms)	0.196664
Average shadow prepare time (ms)	0.0983317
Bind time - shadow prepare time (ms)	**0.178153**

Table 1: *Overheads of preparing shadows*

Method name	SOR::execute
First JIT time (ms)	13.7202
# shadows created	1
Total shadow create time (ms)	13.3576
Average shadow create time (ms)	13.3576
First Jit time - shadow create time (ms)	**0.3626**

Table 2: *Overheads of creating shadows*

	Wrapper Method SOR::execute	Shadow Method SOR::_execute
Function ID	0x935ae8	0x935b18
Enter/Leave count	15	15
JIT Count	15	1
# shadows created	1	0
Create shadow (ms)	11.1834	n/a
Ttl Invoke time (ms)	6273.27	6272.31
Ttl JIT time (ms)	2.9622	0.90244
Ttl method time (ms)	6287.4156	6273.21244

Table 3: *Execution overheads on SciMark2.SOR::execute*

Contributions

- Framework for dynamically attaching/detaching a repair engine to/from a target system executing in a managed execution environment
- Prototype which targets the Common Language Runtime (CLR) and supports this dynamic attach/detach capability with low runtime overhead (~5%)

Limitations

- Repairs can be scheduled but they depend on the execution flow of the application to be effected
  - Deepak Gupta et al. prove that it is un-decidable to automatically determine that “now” is the right time for a repair
  - Programmer-knowledge is needed to identify “safe” control-points at which repairs could be performed
  - The “safe” control points may be difficult to identify and may impact the kind of repair action possible
- Primarily applicable to managed execution environments
  - Increased metadata availability/accessibility
  - Security sandboxes restrict the permissions of injected bytecode to the permissions granted to the original application

Conclusions & Future Work

- Despite being primarily applicable to managed execution environments, these techniques may help us "Watch the Watchers"
  - the management infrastructure we are building is likely to be written in managed code (Java, C#) running in the JVM, CLR mainly because these environments provide application services that enhance the robustness of managed applications
- On the to-do list:
  - Continue working on the prototype for the JVM so we can compare the performance and generalize the runtime support requirements listed earlier
  - Do a real case study to see what issues we run into with respect to identifying and leveraging "safe" control points, the implications of architectural style

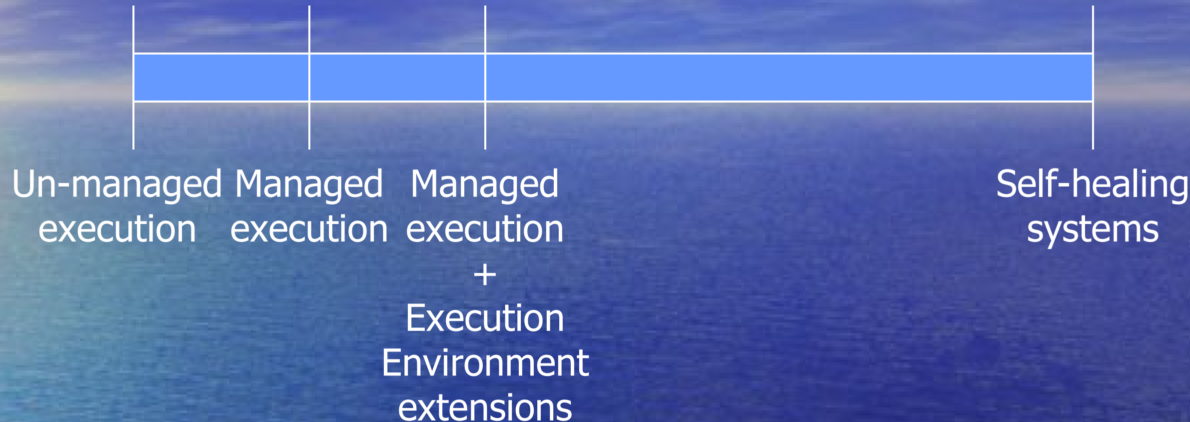
Comments, Questions, Queries

Thank You

Contact: rg2023@cs.columbia.edu

Extra slides

Motivation



- Managed execution environments e.g. JVM, CLR provide a number of application services that enhance the robustness of software systems BUT…
- They do not currently provide services to allow applications to perform consistency checks or repairs of their components
- Managed execution environments intercept everything applications running on top of them attempt to do. Surely we can leverage this

Our Prototype's Model of Operation