# Manipulating Managed Execution Runtimes to Support Self-Healing Systems

Rean Griffith
Columbia University
rg2023@cs.columbia.edu

Gail Kaiser
Columbia University
kaiser@cs.columbia.edu

## ABSTRACT

Self-healing systems require that repair mechanisms are available to resolve problems that arise while the system executes. Managed execution environments such as the Common Language Runtime (CLR) and Java Virtual Machine (JVM) provide a number of application services (application isolation, security sandboxing, garbage collection and structured exception handling) which are geared primarily at making managed applications more robust. However, none of these services directly enables applications to perform repairs or consistency checks of their components. From a design and implementation standpoint, the preferred way to enable repair in a self-healing system is to use an externalized repair/adaptation architecture rather than hardwiring adaptation logic inside the system where it is harder to analyze, reuse and extend. We present a framework that allows a repair engine to dynamically attach and detach to/from a managed application while it executes essentially adding repair mechanisms as another application service provided in the execution environment.

## Categories and Subject Descriptors

D.1 [**Software**]: Programming Techniques—*Self-healing systems*; H.4 [**Information Systems Applications**]: Miscellaneous

## General Terms

Design,Experimentation,Performance

## Keywords

self-healing systems, online reconfiguration, autonomic computing, Common Language Runtime

## 1. INTRODUCTION

A self-healing system "...automatically detects, diagnoses and repairs localized hardware and software problems" [9]. This definition promotes the non-functional requirement of having repair mechanisms as an important facility that must be available in the implementation of a self-healing system. The traditional approach to performing repairs on a system is to stop the system, make the necessary updates and restart the modified system. However, based on the conceptual architecture for a self-managing system introduced in [9] we expect a self-healing system to be able to perform a repair of its components as part of a proactive, preventative or reactive response to its operating environment while it executes.

Scheduled or unscheduled downtime to perform repairs incurs a cost, this cost cannot always be expressed strictly in terms of money [2, 23]. One potential approach self-healing systems can leverage to maintain high system availability is to perform repairs in a degraded mode of operation[24, 11].

Conceptually, a self-managing system is composed of four (4) key capabilities [13]; **Monitoring** to collect data about its execution and operating environment, performing **Analysis** over the data collected from monitoring, **Planning** an appropriate course of action and **Executing** the plan. Each of the four functions participating in the Monitor-Analyze-Plan-Execute (MAPE) loop consumes and produces knowledge which is integral to the correct functioning of the system. Over its execution lifetime the system builds and refines a knowledge-base of its behavior and environment. Information in the knowledge-base could include patterns of resource utilization and a "scorecard" tracking the success of applying specific repair actions to detected or predicted problems.

One software engineering challenge in implementing a self-healing system is managing the degree of coupling between the components that effect system repair (collectively referred to as *the repair engine*, and the components that realize the system's functional requirements) collectively referred to as *the target system*. For systems being built from scratch, designers can either hardwire repair logic into the target system or separate the concerns of repair and target system functionality as is done in external architectures like Kinesthetics eXtreme (KX) [5] or Rainbow [3]. For legacy systems – which we define as any system for which the source code is not available – designers are limited to using an external architecture and interacting with the target system using whatever effectors or system knobs provided by the original designers.

Externalized repair architectures are preferred for a number of software engineering reasons. Hardwiring the repair logic inside target system components limits its generalization and reuse [22]. The mixing of code that realizes functional requirements and code that meets non-functional requirements (*code tangling* [6]) makes it harder to analyze and reason about the correctness of the repairs being performed. Moreover, it is difficult to evolve (extend or update) the repair facilities without affecting the execution and deployment of the target system. Externalized architectures allow the repair engine and the target system to evolve separately rather than requiring that they are developed and deployed in tandem.

Related to the concern of coupling between the repair engine and the target system are issues of the interaction between the two and its impact on the target system. Examples of interaction issues include, but are not limited to:

- How does the repair engine effect the repair of the target system?
- What is the scope of the repair actions that can be performed, for example, can we perform repairs at the granularity of entire programs, subsystems, components, classes, methods or statements? Further, can we add, remove, update, replace or verify the consistency of elements at the same granularity?
- What is the impact of the repair engine on the performance of the target system when repairs are/are not being performed?
- How do we control and coordinate the interaction between the repair engine and the target application with respect to the timing of repair actions given that application consistency must be preserved?

## 1.1 Contribution

To address these issues we present a framework for dynamically attaching and detaching a repair engine to/from a target system executing in a managed execution environment. To demonstrate this capability we implemented a prototype which targets the Common Language Runtime (CLR). We chose the CLR because it includes facilities to make fine-grained changes to types and methods at runtime. There is infrastructure support for extending the existing metadata of types, generating new metadata, editing and replacing method bodies and performing multiple just-in-time (JIT) compilations to persist or undo changes made to method bodies. We leverage these facilities to add repair mechanisms as an application service provided in the managed execution environment.

The two popular Java Virtual Machine (JVM) implementations we considered do not yet provide the same level of flexibility as the CLR. Sun Microsystem's Java HotSpot Virtual Machine 5.0 and IBM's JVM implementation support coarse-grained "HotSwap" classfile replacement[16]. Both implementations support JIT compilation however we could not find any indication that APIs existed to allow us to programmatically control or influence the JIT compilation process at runtime[25]. Also, The current API reference for the Java Virtual Machine Tool Interface (JVMTI)[19] allows interested parties to receive notifications of JVM execution events such as class loads however, the API does not include callbacks that receive JIT compilation event notifications.

Using our framework a repair engine can attach to a running application and once attached, can perform highly specific consistency checks and repairs over individual components and sub-systems before detaching. Further, it allows for the replacement of individual method bodies and components. When no repairs are being performed, our prototype's impact on the target system is negligible, $\sim$5% runtime overhead. Finally, it allows repairs to be enacted at well understood times during target system execution.

Our framework is transparent to the application, it is not necessary to modify the target system's source code to facilitate attaching or detaching the repair engine or to initiate repair actions.

The rest of this paper is organized as follows. Section 2 covers some background on .NET and the CLR execution model. Section 3 presents the architecture of our framework. Section 4 evaluates the performance of our prototype. Section 5 covers related work, section 6 presents our conclusions and future work.

# 2. BACKGROUND

## 2.1 Common Language Runtime Basics

The CLR is the runtime environment in which .NET applications execute. It provides an operating layer between the .NET application and the underlying operating system [15]. The CLR manages the execution of .NET applications, taking on the responsibility of providing services such as application isolation, security sandboxing and garbage collection. Managed .NET applications are called *assemblies* and managed executables are called *modules*. Within the CLR, assemblies execute in *application domains* which are logical constructs used by the runtime to provide isolation from other managed applications.

.NET applications, as generated by the various compilers that target the CLR, are represented in an abstract intermediate form. This abstract intermediate representation is comprised of two main elements, *metadata* and *managed code*. Metadata is "...a system of descriptors of all structural items of the application – classes, their members and attributes, global items...and their relationships"[15]. *Tokens* are handles to metadata entries, they can refer to types, methods, members etc. Tokens are used instead of pointers so that the abstract intermediate representation is memory-model independent. Managed code "...represents the functionality of the application's methods...encoded in an abstract binary format known as Microsoft Intermediate Language (MSIL)" [15]. MSIL, also referred to as bytecode, is a set of abstract instructions targeted at the CLR.

.NET applications written in different languages can interoperate closely, calling each others functions and leveraging *cross-language inheritance*, since they share the same abstract intermediate representation.

## 2.2 Common Language Runtime Execution Model

During execution two major components of the CLR that interact with metadata and bytecode are the *loader* and the *just-in-time (JIT) compiler*. The loader reads the assembly metadata and creates an in-memory representation and layout of the various classes, members and methods on demand as each class is referenced. The JIT compiler uses the results of the loader and compiles the bytecode for each method into native assembly instructions for the target platform. JIT compilation only occurs the first time the method is called in the managed application. Compiled methods remain cached in memory, subsequent method calls jump directly into the native (compiled) version of the method skipping the JIT compilation step, as shown in Figure 1.

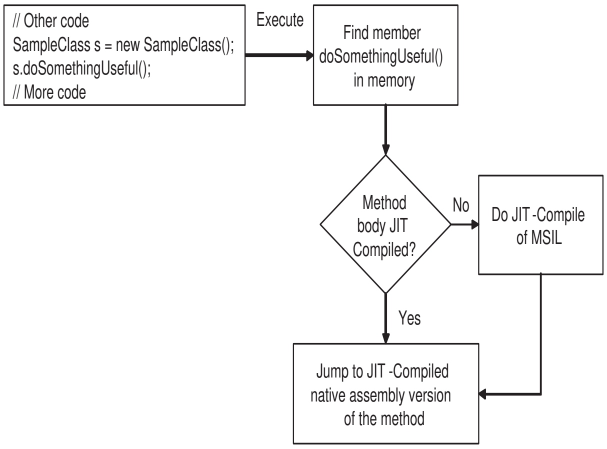


**Figure 1:** ***Overview of the CLR execution cycle***

## 2.3 The CLR Profiler and Unmanaged Metadata APIs

The CLR Profiler APIs allow an interested party (a profiler) to collect information on the execution and memory usage of a running application. There are two interfaces of interest, ICorProfilerCallback, which a profiler must implement and ICorProfilerInfo which is implemented by the CLR. Implementors of ICorProfilerCallback (also referred to as the *notifications API* [18]) can receive notifications about assembly loads and unloads, module loads and unloads, class loads and unloads, function entry and exit and just-in-time compilations of method bodies. The complete list of notifications can be found in [18].The ICorProfilerInfo interface is used by the profiler to obtain details about particular events e.g. when a module has finished loading, the CLR will call the ICorProfilerCallback::ModuleLoadFinished implementation of the profiler passing the *moduleID*. The profiler can then use ICorProfilerInfo::GetModuleInfo to get the module's name, path and base load address.

The unmanaged metadata APIs allow users to emit/import data for/from the CLR. These interfaces are considered low-level interfaces that provide fast access to metadata [17]. There are two interfaces of interest, IMetaDataEmit and IMetaDataImport. As the names suggest, the former is used to write metadata and the latter is used to read metadata. As mentioned earlier in Section 2.1, tokens are abstractions used as handles to the metadata of module, type, method, members etc. IMetaDataEmit generates new metadata tokens as metadata is written while IMetaDataImport resolves the details of a supplied metadata token.

## 2.4 Motivation behind our framework

Application services such as isolation, security sandboxing, garbage collection and structured exception handling which managed execution environments provide may make managed applications more robust but these facilities do not directly enable applications to effect the repair of their components or sub-systems.

Our framework demonstrates how self-healing capabilities can be added to a managed execution environment as another application service. We target the CLR with our prototype because it provides the infrastructure to make fine-grained changes to the metadata of types and methods at runtime and affords control over the JIT compilation process, currently offering more flexibility than the JVM.

Our prototype is enabled by the .NET Profiler Interface [18], specifically ICorProfilerCallback and ICorProfilerInfo, and the Metadata Unmanaged API [17], specifically IMetaDataImport and IMetaDataEmit.

# 3. ARCHITECTURE

Our framework prototype is implemented as a single dynamic linked library (DLL) which includes a profiler that implements ICorProfilerCallback. Figure 2 shows the four (4) main components in our prototype.

- The **Execution Monitor** receives module load, unload and module attached to assembly events, JIT compilation events and function entry and exit events from the CLR.
- The **Metadata Helper** wraps the IMetaDataImport interface and is used by the Execution Monitor to resolve metadata tokens such as method tokens to less cryptic method names and attributes.
- **Internal book-keeping structures** store the results of metadata resolutions as well as execution statistics such as method invocation and JIT compilation times.
- The **Byte-code and Metadata Transformer** wraps the IMetaDataEmit interface to write new metadata e.g. adding new methods to a type and adding references to external assemblies, types and methods. It also generates, inserts and replaces bytecode in existing methods as directed by the Execution Monitor. Bytecode changes are committed by forcing the CLR to JIT compile the modified methods again (**re-JIT**).

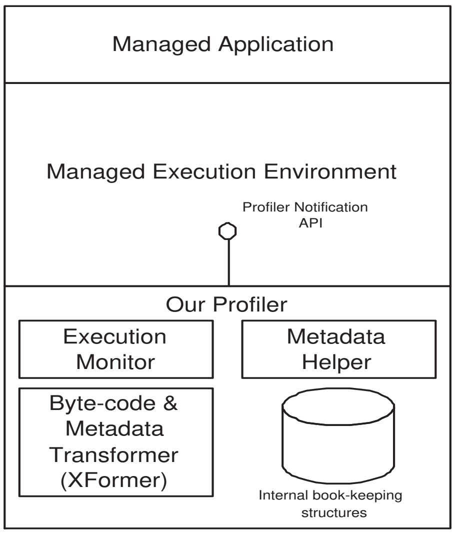


**Figure 2:** ***Prototype architecture diagram***

## 3.1 Model of operation

Our prototype performs operations on types and methods at various stages in the method invocation cycle shown in Figure 3 to make them capable of interacting with a repair engine.

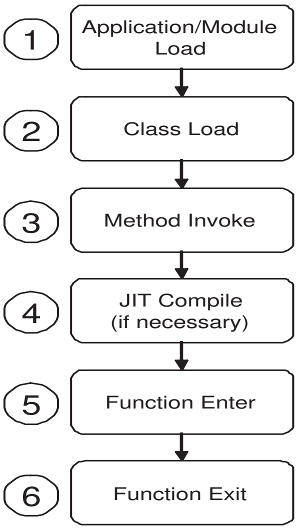


**Figure 3:** ***First method invocation in a Managed Application***

To allow a repair engine to interact with a class instance we augment the type definition such that the necessary "hooks" can be added. Augmenting the type definition is a two-phase operation. The first phase occurs at module load time, Stage 1 in Figure 3.

When the loader loads a module, the bytecode for the method bodies of the module's types is laid out in memory. The starting address of the first bytecode instruction in a method body is referred to as the *Relative Virtual Address (RVA)* of the method. At the end of the module load we add (prepare) *shadow methods*, using IMetaDataEmit::DefineMethod, for each of the original public and/or private methods of the type. A shadow method shares all the properties (attributes, signature, implementation flags and RVA) of the original method except the name. By sharing (borrowing) the RVA of the original method, the shadow method points at the method body of the original method. Figure 4 shows an example of adding a shadow method, _SampleMethod, for an original method, SampleMethod. Extending the metadata of a type by adding methods must be done before the type definition is installed in the CLR. Once the type definition is installed its list of methods and members becomes read only, further requests to define new methods or members are silently ignored even though the API call "succeeds".

The second phase of type augmentation occurs the first time an original method is JIT compiled, Stage 4 in Figure 3. This phase converts the original method into a thin *wrapper* which simply calls the shadow method as shown in Figure 5. The heart of phase 2 allocates space for a new method body, uses the Byte-code & Metadata Transformer to generate the sequence of bytecode instructions to call the shadow and sets the new RVA for the original method to point at the new method body.

Using shadows and wrappers has a number of advantages.

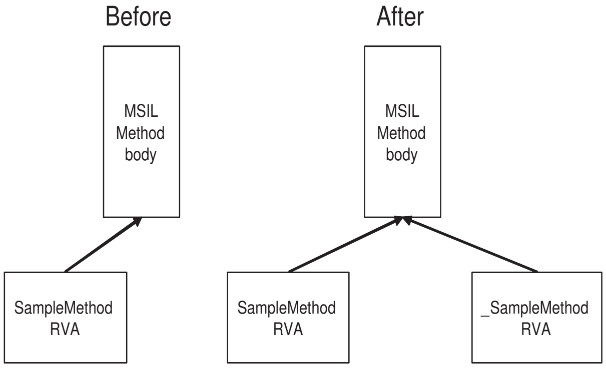


**Figure 4:** ***Preparing a shadow method***

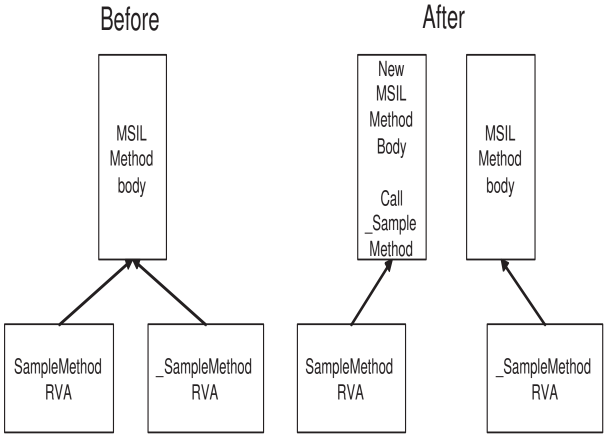


**Figure 5:** ***Creating a shadow method***

Given the structure of the wrapper method, see Figure 6, we can inject repair instructions as prologs and/or epilogs to shadow method calls.

```
SampleMethod ( args )
   <room for prolog>
   push args
   call _SampleMethod ( args )
   <room for epilog>
   return value/void
```

**Figure 6:** ***Conceptual diagram of a Wrapper***

Adding a prolog to the wrapper requires that new bytecode instructions prefix the existing bytecode instructions. The level of difficulty is the same whether we augment the wrapper or the original method. Adding epilogs, however, presents a few more challenges. Intuitively, to add an epilog, we wish to insert new instructions before control leaves a method. In the simple case, a method has a single return statement and the epilog can be inserted right before that point. For methods with multiple return statements and/or exception handling routines, finding every possible return point can be an arduous task [20]. Further, the layout and packing of the bytecode for methods that contain exception handling routines is considered a special case which may be challenging to augment correctly [20].

Using wrappers presents a cleaner approach since we can ignore all of the complexity in the shadow method. Further, the regular structure and single return statement of the wrapper method lends itself easily to adding an epilog.

## 3.2 Performing a repair

To perform a repair, we augment the wrapper to insert a jump into a repair engine at the *control point(s)* before and/or after a shadow method call. Effecting the jump into a repair engine is a four-step process.

- Step one extends the metadata of the assembly currently executing in the CLR such that a reference to the assembly containing the repair engine is added using IMetaDataEmit::DefineAssemblyRef.
- Step two uses IMetaDataEmit::DefineTypeRef to add references to the repair engine type (class).
- Step three adds references to the subset of the repair engine's methods that we wish to insert calls to, using IMetaDataEmit::DefineMemberRef.
- Step four augments the bytecode and metadata of the wrapper function to insert bytecode instructions to make calls into the repair engine before and/or after the existing bytecode that calls the shadow method.

To persist the bytecode changes made to the method bodies of the wrappers the Execution Monitor requests the CLR JIT compile the wrapper method again (referred to as a re-JIT). The actual re-JIT takes place the next time the wrapper method is called. In our protoype re-JIT requests are submitted in the Function Exit event, Stage 6 in Figure 3.

The infrastructure used to force a function re-JIT is provided by the ICorProfilerInfo::SetFunctionReJIT function. We use this function to persist bytecode changes but we can also use it to undo the changes we make. We can temporarily disable shadows, reverting back to shadow prepare phase, Figure 4 and we can remove prologs and/or epilogs by setting the wrapper method RVA to the RVA of a method body without those prologs and/or epilogs and requesting a re-JIT. This facility allows us to recover from any performance hit we take from making shadowed method calls and we can flexibly attach or detach the repair engine as desired.

The ability to perform multiple JIT compilations on demand is a quite powerful facility, however some additional tweaking is required to get function re-JITs to work as expected in our prototype, interested readers are directed to our technical report [7] for details.

# 4. PERFORMANCE EVALUATION

We have evaluated the performance of our prototype by quantifying the overheads on program execution using two separate benchmarks.

## 4.1 Experimental Setup

The experiments were run on a single Pentium III Mobile Processor, 1.2 GHz with 1 GB RAM. The platform was Windows XP SP2 running the .NET Framework v1.14322.

In our evaluation we used the C# benchmarks SciMark [1] and Linpack [2].

[1] http://rotor.cs.cornell.edu/SciMark/

[2] http://www.shudo.net/jit/perf/Linpack.cs

**SciMark** is a benchmark for scientific and numerical computing. It includes five (5) computation kernels: Fast Fourier Transform (FFT), Jacobi Successive Over-relaxation (SOR), Monte Carlo integration (Monte Carlo), Sparse matrix multiply (Sparse MatMult) and dense LU matrix factorization (LU).

**Linpack** is a benchmark that uses routines for solving common problems in numerical linear algebra including linear systems of equations, eigenvalues and eigenvectors, linear least squares and singular value decomposition. In our tests we used a problem size of 1000.

## 4.2 Overheads

Our framework consists of a profiler that uses the Profiler API [18] to intercept module load, unload and module attached to assembly events, JIT compilation events and function entry and exit events. As expected, running an application in the profiler imposes some overhead on the application. Figure 7 shows the runtime overhead for running the benchmarks with and without profiling enabled. We performed five (5) test runs for SciMark and Linpack each with and without profiling enabled. All executables under test and our profiler implementation were optimized release builds. For each benchmark, the bar on the left shows the performance normalized to one, of the benchmark running without profiling enabled. The bar on the right shows the normalized performance with our profiler enabled.

Our measurements show that our profiler contributes ∼5% runtime overhead when no repairs are active, which we consider negligible.

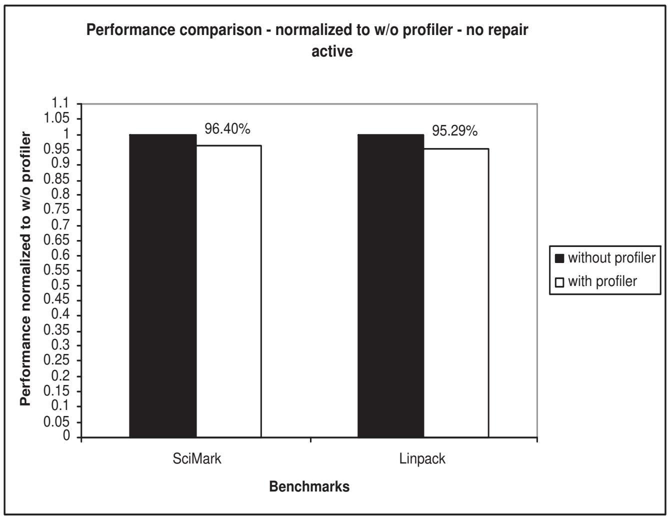


**Figure 7:** ***Overheads when no repair active***

Our prototype imposes additional overheads on the running application at different points in its execution. We prepare shadows at module load time, specifically when the module binds to an assembly which occurs before the application begins running. We create shadows the first time the method is JIT compiled, provided a shadow has been prepared for it and we force re-JITs when we add or remove the prologs and epilogs that jump into the repair engine.

To quantify these overheads, we use the SciMark2.SOR class which executes the Jacobi Successive Over-relaxation benchmark. Table 1 shows the impact on module bind time due to preparing shadows on the two public methods of SciMark2.SOR, SciMark2.SOR::execute and SciMark2.SOR::num_flops.

Preparing shadows at module load time causes the application to take slightly longer to load but does not affect its steady state execution since the module bind must occur before the application begins to execute. Moreover, the impact on module bind time in this case is relatively small, submillisecond, and is dominated by time spent making calls to IMetaDataEmit::DefineMethod which adds new method definitions to a type.

| 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***

Creating shadows imposes a one time overhead incurred the first time the method is JIT compiled. As shown in Table 2 the time for the first JIT compilation is dominated by the time spent creating the shadow[3].

| 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***

Forcing multiple JIT-compilations adds additional overhead to the steady-state execution times of the application. In our experiments we compute the method time as:

$$T_{totalmethodtime} = T_{shadowcreatetime} + T_{JITtime} + T_{invoketime}$$

Table 3 compares the total method time for the SciMark2.SOR::execute method, the wrapper method, with the total method time for its shadow method. In this case the disparity in method times is $\ll$1% and the overall impact on the performance of the benchmark is negligible.

For methods that are not as computationally intensive as SOR::execute, where $T_{shadowcreatetime} + T_{JITtime}$ is a significant fraction of $T_{invoketime}$, we expect the overheads of creating shadows and multiple re-JITs to be much worse.

# 5. RELATED WORK

Aspect Oriented Programming (AOP) is an approach to designing software that allows developers the modularize cross-cutting concerns [6] that manifest themselves as non-functional system requirements. Modularized cross-cutting concerns, *aspects*, allow developers to cleanly separate the logic that meets system requirements from the code that meets the non-functional system requirements.

[3]Shadow creation time is dominated by the calls to the IMethodMalloc::Alloc function which allocates the buffer for the new method body at the appropriate address in memory

| | 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***

In the context of self-healing systems AOP is an approach to designing the system such that the non-functional requirement of having repair mechanisms available is cleanly separated from the logic that meets the system's functional requirements. An AOP engine is still necessary to realize the final system.

AOP engines *weave* together the code that meets the functional requirements of the system with the aspects that encapsulate the non-functional system requirements. In an AOP engine implementation a *join point* is a well-defined point in the control-flow of a program e.g a method call and are the places where aspects can be woven in. A *pointcut* selects a particular join point e.g. method foo(). *Advice* defines the code to be executed at a join point. There are three (3) types of advice:

- Before advice executes when a join point is reached but before the computation begins e.g. before foo()
- After advice executes when a join point is reached by after the computation ends e.g. after foo()
- Around advice executes when a join point is reached instead of the original computation e.g. around foo()

Specifying that we wish to perform a repair before or after a particular method call (method call interposition) using AOP is simply a matter of indicating the methods and the kind of advice desired – usually AOP engine implementations have their own languages for doing this. However, the way different AOP engines effect the request vary depending on the kind of AOP engine it is.

There are three kinds of AOP engines, those that perform weaving at compile time (static weaving) e.g. AspectJ [21], Aspect C# [10], those that perform weaving after compile time but before load time e.g. Weave .NET [21] and Aspect.NET [21] which pre-process .NET assemblies, operating directly on bytecode and metadata and those that perform weaving at runtime (dynamic weaving) e.g. A dynamic AOP-Engine for .NET [4] and CLAW [14].

Compile time weavers require access to the source code, moreover once the weaving is done and the final system is compiled, there is no way to remove the aspects without re-compilation. Pre-load-time weavers operate on bytecode creating custom class files for java or custom assemblies for .NET, however, once weaving completes there is no way to remove the aspects. Runtime weavers offer the most flexibility, they do not require access to the source code and weaving and un-weaving can occur at runtime.

Our prototype exhibits dynamic weaving functionality and is most similar to A Dynamic AOP-Engine for .NET [4] and Cross-Language Load-Time Aspect Weaver (CLAW) [14].

A Dynamic AOP-Engine for .NET [4] exhibits the basic behavior necessary to enable method call interposition before, after and around a given method. Injection and removal of aspects is done at runtime using the CLR profiler API for method re-JITs and Unmanaged Metadata APIs, however, their system requires that applications run with the debugger enabled which incurs as much as a 3X performance slowdown.

The Cross-Language Load-Time Aspect Weaver (CLAW) [14] uses dynamically generated proxies to intercept method calls before passing them onto the "real" callee. CLAW uses the CLR profiler interface and the Unmanaged Metadata APIs to generate dynamic proxies and insert aspects. An implementation of CLAW was never released and development seems to have tapered off, as a result we were unable to investigate its capabilities and implementation details.

Despite the functional similarities to dynamic AOP engines our focus is to demonstrate how flexible dynamic AOP engine facilities can be used to add repair mechanisms as a service provided by managed execution environments.

AOP is not the only way to achieve method call interposition. Software Engineers are free to design in such facilities as is done in the open-source research operating system K42 [1]. K42 uses method-call interposition and designed in indirection mechanisms between operating system components to support hot component swaps.

Other approaches for building systems that exhibit self-healing capabilities construct the system from components that have specific properties and special interfaces. Candea et al. [2] propose the use of crash-only components and recursively restartable systems. With crash-only components the only way to stop them is to crash them and the only way to start them is to recover them. When components fail a recovery manager uses their recovery API to restart them. Recovery (micro-reboot) begins at a component and propogates "outwards" to sub-systems before restarting the application.

## 6. CONCLUSIONS

We have presented a framework prototype that uses dynamic AOP engine facilities to allow us to transparently attach/detach a repair engine to/from a target system executing in a managed execution environment with low-overhead. This allows us to add repair mechanisms as another application service provided by managed execution environments. Our framework relies on facilities for extending type and method metadata and the ability to control and direct the JIT compilation process.

For future work we seek to investigate the issue of repair timings. The timing of repairs is a major consideration for *change management* [12] in software systems. Gupta in [8] presents a proof of the undecidability of automatically finding all the control points in an application where a consistency-preserving repair/adaptation can be performed. However, we can use a software engineer's knowledge of the application to identify *some* safe control points and invoke highly specific repairs only at those well understood times.

## 7. ACKNOWLEDGMENTS

The Programming Systems Laboratory is funded in part by National Science Foundation grants CNS-0426623, CCR-0203876 and EIA-0202063, and in part by Microsoft Research.

## 8. REFERENCES

[1] C. Soules et. al. System support for online reconfiguration, 2003.

[2] G. Candea, J. Cutler, and A. Fox. Improving availability with recursive micro-reboots: A soft-state case study. In *Dependable systems and networks - performance and dependability symposium (DNS-PDS)*, 2002.

[3] S.-W. Cheng, A.-C. Huang, D. Garlan, B. R. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.

[4] A. Frei, P. Grawehr, and G. Alonso. A dynamic aop-engine for .net. Tech Rep. 445 Dept. of Comp Sci. ETH Zurich, 2004.

[5] Gail Kaiser et. al. Kinesthetics extreme: An enternal infrastructure for monitoring distributed legacy systems. In *The Autonomic Computing Workshop 5th Workshop on Active Middleware Services (AMS)*, June 2003.

[6] Gregor Kiczales et. al. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[7] R. Griffith and G. Kaiser. Adding self-healing capabilities to the common language runtime. Technical report, Columbia University, 2005.

[8] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *Software Engineering*, 22(2), 1996.

[9] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer magazine*, January 2003.

[10] H. Kim. Aspectc#: An aosd implementation for c#. Master's thesis, Department of Computer Science Trinity College Dublin, 2002.

[11] P. Koopman. Elements of the self-healing problem space. In *ICSE Workshop on Architecting Dependable Systems*, 2003.

[12] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11), 1990.

[13] L. Stojanovic et.al. The role of ontologies in autonomic computing systems. *IBM Systems Journal*, 43(3), 2004.

[14] J. Lam. Claw: Cross-language load-time aspect weaving on microsoft's clr demonstration at aosd 2002.

[15] S. Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft Press, 2002.

[16] M. Dmitriev. Application of the java hotswap technology to advanced profiling. In *Proceedings of the Workshop on Unanticipated Software Evolution*, June 2002.

[17] Microsoft. Common language runtime metadata unmanaged api, 2002.

[18] Microsoft. Common language runtime profiling, 2002.

[19] S. Microsystems. The jvm tool interface version 1.0. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html.

[20] A. Mikunov. Rewrite msil code on the fly with the .net framework profiling api.

[21] B. Rasmussen, C. Jensen, J. Nielsen, and L. Jensen. Aspect.net - a cross-language aspect weaver. Master's thesis, Department of Computer Science Trinity College Dublin, 2002.

[22] B. Schmerl and D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *Proceedings of the 14th International Conference of Software Engineering and Knowledge Engineering*, 2002.

[23] M. E. Segal and O. Frieder. On-the-fly program modification systems for dynamic updating. *IEEE Software*, 10(2), March 1993.

[24] C. Shelton and P. Koopman. Using architectural properties to model and measure system-wide graceful degradation. In *Workshop on Architecting Dependable Systems*, 2002.

[25] N. Yevik. Introduction to ibm jvm for linux jit diagnostics. www-128.ibm.com/developerworks/eserver/library/es-JITDiag.html.