
CRC Book

CRC PRESS
Boca Raton Ann Arbor London Tokyo

Contents

1 Effecting Runtime Reconfiguration in Managed Execution Environments

1

Rean Griffith, Giuseppe Valetto, Gail Kaiser Columbia University, IBM Thomas J.

Watson Research Center, Columbia University

1.1	Introduction	2
1.1.1	Contributions	3
1.2	Background	4
1.2.1	Common Language Runtime Basics	4
1.2.2	Common Language Runtime Execution Model	4
1.2.3	The CLR Profiler and Unmanaged Metadata APIs	4
1.3	Adaptation Framework Prototype Overview	5
1.3.1	Model of Operation	6
1.3.2	Performing an Adaptation	7
1.4	Dynamic Reconfiguration Experiment	8
1.4.1	Alchemi Architecture	8
1.4.2	Motivation behind Reconfiguring Alchemi	9
1.4.3	Reconfiguring Alchemi	10
1.4.4	The Reconfiguration Engine and Replacement Scheduler	11
1.5	Empirical Evaluation	12
1.5.1	Experimental Setup	12
1.5.2	Results	13
1.6	Related Work	13
1.7	Conclusions and Future Work	15
1.8	Acknowledgments	15

1

Effecting Runtime Reconfiguration in Managed Execution Environments

Rean Griffith, Giuseppe Valetto, Gail Kaiser

Columbia University, IBM Thomas J. Watson Research Center, Columbia University

CONTENTS

1.1	Introduction	1
1.2	Background	3
1.3	Adaptation Framework Prototype Overview	5
1.4	Dynamic Reconfiguration Experiment	8
1.5	Empirical Evaluation	12
1.6	Related Work	13
1.7	Conclusions and Future Work	15
1.8	Acknowledgments	15
	References	16

Managed execution environments such as Microsoft's Common Language Runtime (CLR) and Sun Microsystems' Java Virtual Machine (JVM) provide a number of services – including but not limited to application isolation, security sandboxing, garbage collection and structured exception handling – that are aimed primarily at enhancing the robustness of managed applications. However, none of these services directly enables performing reconfigurations, repairs or diagnostics on the managed applications and/or its constituent subsystems and components.

In this paper we examine how the facilities of a managed execution environment can be leveraged to support runtime system adaptations, such as reconfigurations and repairs. We describe an adaptation framework we have developed, which uses these facilities to dynamically attach/detach an engine capable of performing reconfigurations and repairs on a target system while it executes. Our adaptation framework is lightweight, and transparent to the application and the managed execution environment: it does not require recompilation of the application nor specially compiled versions of the managed execution runtime. Our prototype was implemented for the CLR. To evaluate our framework beyond toy examples, we searched on SourceForge for potential target systems already implemented on the CLR that might benefit from runtime adaptation. We report on our experience using our prototype to effect runtime reconfigurations in a system that was developed and is in use by others: the Alchemi Enterprise Grid Computing System developed at the University of Melbourne, Australia [1].

1.1 Introduction

A self-healing system “...automatically detects, diagnoses and repairs localized hardware and software problems” [2]. Thus we expect a self-healing system to perform runtime reconfigurations or repairs of its components as part of a proactive, preventative or reactive response to conditions arising within its operating environment. This runtime response contrasts with the traditional approach to performing system reconfigurations or repairs – stop the system, fix it, then restart – which requires scheduled or unscheduled downtime and incurs costs that cannot always be expressed strictly in terms of money [3, 4]. Keeping the system running while adaptations are being carried out (even if it means operating in a degraded mode [5, 6]) is in many cases more desirable since it maintains some degree of availability.

One software engineering challenge in implementing a self-healing system is managing the degree of coupling between the components that effect system adaptation (collectively referred to as *the adaptation 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 adaptation logic into the target system or separate the concerns of adaptation and target system functionality, by means of specialized middleware like IQ-Services [7] and ACT [8] or externalized architectures that include a reconfiguration/repair engine, as in Kineshetics eXtreme (KX) [9] or Rainbow [10]. For legacy systems – which we define as any system for which the source code is not available, or for which it is undesirable to engage in substantial re-design and development – one is limited to using an external adaptation engine that interacts with the target system using whatever effectors were exposed by the original designers.

Externalized adaptation architectures may be preferred for a number of software engineering reasons. Hardwiring the adaptation logic inside target system components limits its generalization and reuse [11]. The mixing of code that realizes functional requirements and code that meets non-functional requirements (*code tangling* [12]) makes it harder to analyze and reason about the correctness of the adaptations being performed. Moreover, it is difficult to evolve (extend or update) the adaptation facilities without affecting the execution and deployment of the target system. Externalized architectures allow the adaptation engine and the target system to evolve independently rather than requiring that they be developed and deployed in tandem.

We are concerned with identifying and addressing the interactions between the adaptation engine and the target system, while still seeking to minimize their coupling. Examples of interaction issues include, but are not limited to:

1. How does the adaptation engine effect (i.e., conduct) the reconfiguration or repair of the target system?
2. What is the scope of the adaptation actions that can be applied, e.g., can we perform reconfigurations at the granularity of entire programs, subsystems or

components? Can we repair whole programs, subsystems, individual components, classes, methods or statements? Further, can we add, remove, update, replace or verify the consistency of elements at the same granularity?

3. What is the impact of the adaptation engine on the performance of the target system when adaptations are/are not being performed?
4. How do we control and coordinate the adaptation engine and the target application with respect to the timing of adaptation actions given that application consistency must be preserved?

1.1.1 Contributions

In [13] we presented an adaptation framework to partially address 1, 2 and 3, in the context of target systems that run in a managed execution environment. Our main focus there was on evaluating performance overhead, using a set of computationally-intensive scientific applications written in C#. In this paper we present an experiment geared towards exploring some of the issues associated with effecting consistency-preserving reconfigurations or repairs (question 4) in a “real-life” system augmented with our adaptation framework. We chose Alchemi because it meets our technical criteria, and is publicly available and apparently in use.

Our prototype uses the profiler API of Microsoft’s managed execution environment – the Common Language Runtime (CLR) – to track the application’s execution, and effect changes via bytecode rewriting and creating/augmenting the metadata associated with modules, types and methods. Conceptually, our approach could be applied to other managed execution environments, e.g., Sun Microsystems’ Java Virtual Machine (JVM). We chose CLR due to certain technical limitations of most JVM implementations, which we elaborate in [14].

Using our adaptation framework, the reconfiguration/repair engine can attach to a running application and perform highly specific consistency checks, reconfigurations and/or repairs over individual components and sub-systems before detaching. The framework remains transparent to the application: it is not necessary to modify the target system’s source code to facilitate attaching/detaching the adaptation engine or to enable adaptation actions. Further, it is fine-grained, allowing for the replacement of individual method bodies as well as entire components. When no adaptations are being performed, our prototype’s impact on the target system is small, around ~5% or less runtime overhead (see [13] for details). Finally, it allows adaptations to be enacted at well-understood timing points during target system execution.

The remainder of this chapter is organized as follows: §1.2 covers some background on .NET and the CLR’s execution model. §1.3 describes how our adaptation framework prototype works. §1.4 describes the target system we selected for our experiment, the Alchemi Enterprise Grid Computing System, and outlines the steps involved in reconfiguring that system at runtime. §1.5 provides details about performance measurements and evaluates the impact of the prototype on the target system. §1.6 briefly discusses related work. Finally, §1.7 presents our conclusions and directions for future work.

1.2 Background

1.2.1 Common Language Runtime Basics

The CLR is the managed runtime environment in which .NET applications execute. It provides an operating layer between .NET applications and the underlying operating system [15]. The CLR takes 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 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, which 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 other's functions and leveraging *cross-language inheritance*, since they share the same abstract intermediate representation.

1.2.2 Common Language Runtime Execution Model

Two major components of the CLR interact with metadata and bytecode during execution, 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, and subsequent method calls jump directly into the native (compiled) version of the method skipping the JIT compilation step, see Figure 1.1.

1.2.3 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* [16]) 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 *ICor-*

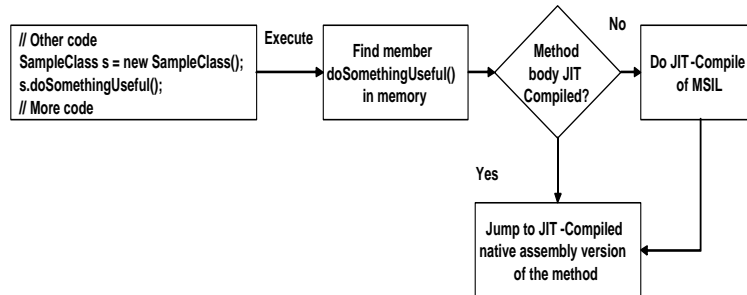


FIGURE 1.1

Overview of the CLR Execution Cycle

ProfilerInfo 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 are low-level interfaces that provide fast access to metadata, allowing users to emit/import data for/from the CLR [17]. There are two interfaces of interest, `IMetadataEmit` and `IMetadataImport`. `IMetadataEmit` generates new metadata tokens as metadata is written, while `IMetadataImport` resolves the details of a supplied metadata token.

1.3 Adaptation Framework Prototype Overview

Our adaptation framework prototype is implemented as a single dynamic linked library (DLL), which includes a profiler that implements `ICorProfilerCallback`. It is written in C++ and the implementation amounts to 3157 lines of code (LOC). There are four main components in our prototype:

- The **Execution Monitor** receives “module load”, “module 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 to less cryptic method names and attributes.
- **Internal book-keeping structures** store the results of metadata resolutions and method invocations, 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 causing the CLR to JIT-compile the modified methods again (referred to as **re-JIT**).

1.3.1 Model of Operation

Our prototype performs operations on types and methods at various stages in the method invocation cycle, shown in Figure 1.2, to make them capable of interacting with an adaptation engine.

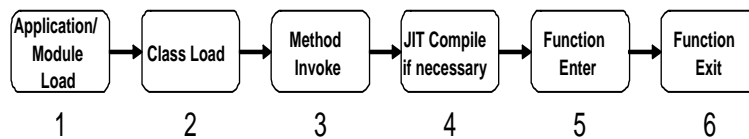


FIGURE 1.2

First Method Invocation in a Managed Application

To allow an adaptation 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. Phase one occurs at module load time, Stage 1 in Figure 1.2. 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 1.3, transition A to B, 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, after which point 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 1.2. This phase generates a thin *wrapper* that calls the shadow method as shown in Figure 1.3, transition B to C. 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 with wrappers has a number of advantages. Given the structure

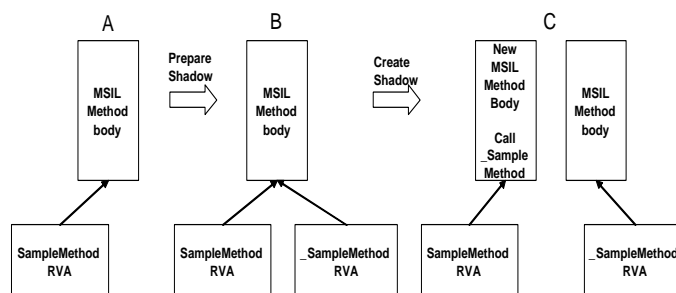


FIGURE 1.3

Preparing and Creating a Shadow Method

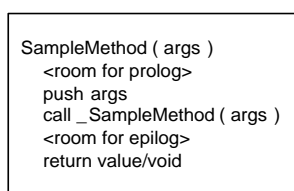


FIGURE 1.4

Conceptual Diagram of a Wrapper

of the wrapper method, see Figure 1.4, we can inject adaptation instructions as prologues and/or epilogues to shadow method calls. Adding a prologue 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 epilogues, however, presents a few more challenges. Intuitively, to add an epilogue, we wish to insert new instructions before control leaves a method. In the simple case, a method has a single return statement and the epilogue 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 [18]. Further, the layout and packing of the bytecode for methods that contain exception handling routines is considered a special case that may be challenging to augment correctly [18]. 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 lends itself easily to adding an epilogue.

1.3.2 Performing an Adaptation

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

1. Extend the metadata of the assembly currently executing in the CLR such that a reference to the assembly containing the adaptation engine is added using `IMetaDataEmit::DefineAssemblyRef`.

2. Use `IMetaDataEmit::DefineTypeRef` to add references to the adaptation engine type (class).
3. Add references to the subset of the adaptation engine's methods that we wish to insert calls to, using `IMetaDataEmit::DefineMemberRef`.
4. Augment the bytecode and metadata of the wrapper function to insert bytecode instructions to make calls into the adaptation 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 causes the CLR to re-JIT the wrapper method the next time the method is called (i.e., JIT-compile again). See [14] for details on CLR re-JITs.

By augmenting the bytecode of the wrappers, as explained above, we can leverage the control-points before and/or after calls to shadow methods to transfer control over to an adaptation engine where it can perform any number of operations including but not limited to: performing consistency checks over class instances or components, reconfigurations and diagnostics of components.

1.4 Dynamic Reconfiguration Experiment

For this experiment, we selected the Alchemi Enterprise Grid Computing System [19], developed at the University of Melbourne, Australia. Alchemi has several appealing characteristics. It is developed and maintained by others, hence we can regard it as a legacy system upon which runtime adaptations can be carried out only via an external engine. It is publicly available (on SourceForge [20]) and well-documented, which makes it easier to construct feasible scenarios where performing runtime reconfigurations on the system could result in real benefits. Alchemi is apparently being used in a number of scientific and commercial grid applications, including a Commonwealth Scientific and Industrial Research Organisation (CSIRO) Land and Water, Australia, application for natural resource modeling, and a micro-array data processing application for early detection of breast cancer developed by Satyam Computers Applied Research Laboratory in India.* Finally, Alchemi is implemented as a .NET application on top of the CLR, which is a prerequisite for our current prototype. Alchemi is written in C#, and leverages a number of technologies provided by the .NET Framework including .NET Remoting [21], multi-threading and asynchronous programming.

1.4.1 Alchemi Architecture

The Alchemi Grid follows a master-worker parallel programming paradigm, where a central component (the Manager) dispatches independent units of parallel execution

*A list of projects using Alchemi can be found at <http://www.alchemi.net/projects.html>.

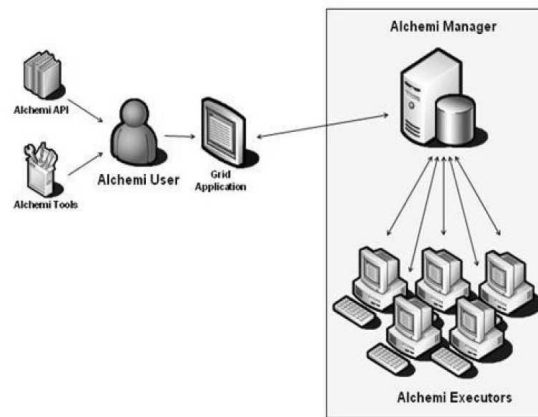


FIGURE 1.5

Alchemi Architecture – Source: User Guide for Alchemi 1.0 [22]

(grid threads) to be executed on grid nodes (Executors), see Figure 1.5. The Manager is responsible for providing the services associated with the execution of grid applications and their constituent grid threads. It monitors the status of the Executors registered with it, and schedules grid threads to run on them. Executors accept grid threads from the Manager, execute them and return the completed threads to the Manager. An Executor can be configured as either *dedicated*, i.e., managed centrally where the Manager “pushes” a computation to an idle, dedicated Executor whenever its scheduling requires it, or *non-dedicated*, where the Executor instead polls the Manager and hence “pulls” some computational work only during idle periods, e.g., when a screen saver is active.

1.4.2 Motivation behind Reconfiguring Alchemi

In Alchemi, the Manager is clearly one key system element and, within the Manager, the scheduler – which makes all of the work allocation decisions – is a key component. As in any resource allocation scenario, the scheduling strategy in a grid is critical to the overall efficacy of the system. Further, the efficacy of a particular scheduling algorithm may depend on factors that can vary quite dynamically within the grid, such as the characteristics, arrival times and rate of jobs submitted for execution, the computational weight of individual work units, the set of available Executors, and the overall workload placed on Executors at any point in time. The current version of Alchemi (v1.0 beta) provides a default scheduler, embodied in the `DefaultScheduler` class, that schedules grid threads on a Priority and First Come First Served (FCFS) basis, in that order. This scheduling algorithm is fixed at compile time and used throughout the execution lifetime. However, Alchemi also provides a scheduling API that allows custom schedulers to be written.

We do not address whether a one-size-fits-all scheduling algorithm could be implemented to take into account all operating conditions and all kinds of submitted

applications, but instead intend to enable the Alchemi Manager to switch among different scheduling algorithms, each tuned for specific conditions and workloads, as the state of the system changes. The same scheduler-swapping provisions could also be used to avert or alleviate situations in which (a subset of) Executors misbehave – for reasons varying from misconfiguration, to the occasional bug in the code of grid threads for some applications, to malicious interference of rogue Executor nodes – in ways that cannot be immediately detected by the monitoring capabilities of the Manager. (Only Executor liveness is currently considered.)

In the next section we describe a proof-of-concept experiment that demonstrates how we carry out such a runtime reconfiguration, without modifying the source code of the system. We show how our adaptation framework is able to transparently swap scheduler implementations on the fly, which would enable existing Alchemi installations to take advantage of multiple other scheduling algorithms without having to re-compile and re-install any system components. We also discuss how the re-configurations are carried out in a way that preserves the consistency of the grid applications, as well as the overall distributed grid system.

We should stress that our experiment focuses on the feasibility of effecting such *consistency-preserving* reconfigurations of a software system like Alchemi, running in a managed execution environment. We would not claim that our experiment – in itself – addresses repair and/or optimization issues. It rather facilitates the development of specific remedies and optimizations: for instance, our approach could enable an adaptive scheduler-swapping scheme that could ensure the grid’s performance across a vast range of applications and conditions, which remains an open and interesting research issue. We also do not address here other plausible applications of runtime adaptation, such as patching potential security vulnerabilities, although we anticipate that the same basic framework should work.

1.4.3 Reconfiguring Alchemi

To swap the grid scheduler in a running instance of the Alchemi grid, we need to implement a reconfiguration engine that interacts with the running instance of the Manager. Using our specialized CLR profiler, described in Section 1.3, we can dynamically attach/detach it to/from a running managed application in a fairly mechanical way. However, a first important step is to carefully plan the interactions between the running application, the reconfiguration engine and the CLR in a way such that they do not compromise the integrity of the managed application or the CLR.

Consequently, our next step is to gather some knowledge about the system. Specifically we need details about how the Manager component works, including the execution flow in the Manager from startup to shutdown – which allows us to identify potential “safe” control points where reconfiguration actions can take place. We also need to identify the set of important classes, i.e., those classes the reconfiguration engine must interact with to effect the scheduler swap. The final step is to implement the reconfiguration engine based on what we learn about the system.

In Alchemi, when the Manager is started (by running the `Alchemi.Manager.exe` assembly), an instance of the `ManagerContainer` class (found in the `Alchemi.Core.dll` assembly) is created. The instance of the `ManagerContainer` class represents the

Manager proper. On startup, the *ManagerContainer::Start()* routine performs a set of initialization tasks:

1. An object is registered with the .NET Remoting services, allowing Executors to interact with the Manager instance.
2. A singleton instance of the *InternalShared* class is created, holding a reference to the scheduler implementation being used (among other things). The concrete scheduler implementation is referenced as an implementation of the *Alchemi.Core.Manager.IScheduler* interface, which standardizes the scheduler API [19].
3. Two threads, the scheduler thread and the watchdog thread, are started. The scheduler thread runs the *ManagerContainer::ScheduleDedicated()* method, which loops “forever” on a flag member variable, *_stopScheduler*. It periodically retrieves the scheduler implementation from the *InternalShared* singleton instance and queries it for a *DedicatedSchedule*. A *DedicatedSchedule* is a <Grid Thread ID, Executor ID> tuple specifying where the selected grid thread should be scheduled to run. The watchdog thread runs the *ManagerContainer::Watchdog()* method, which loops “forever” on the *_stopWatchdog* flag member variable, periodically checking the status of dedicated Executors.

Based on this Manager startup sequence, we outline below the tasks involved in performing a scheduler swap:

1. Insert a prologue into the *ManagerContainer::Start()* method such that it jumps into the reconfiguration engine assembly where it can be cached.
2. Insert a prologue into the constructor for the *InternalShared* class such that it jumps into the reconfiguration engine assembly where it can be cached.
3. Once instances of the *ManagerContainer* and *InternalShared* classes have been cached, we can cause the scheduler thread to exit normally by setting the *_stopScheduler* flag to true, allowing the thread to exit when it next tests the while loop condition.
4. The *Alchemi.Core.Manager.IScheduler* reference stored in the *InternalShared* singleton can then be replaced by another *IScheduler* implementation.
5. The *_stopScheduler* flag is set to false and the scheduler thread is restarted.

1.4.4 The Reconfiguration Engine and Replacement Scheduler

Our implementation, *PSL.Alchemi.ReconfigEngine.dll* assembly, contains two C# classes, *PSLScheduler* and *ReconfigEngine*. The implementation was done without contacting the Alchemi developers and took about half a day to complete. The total implementation is 465 LOC – 95 LOC for *PSLScheduler.cs* and 370 LOC for *ReconfigEngine.cs*.

PSLScheduler implements the `Alchemi.Core.Manager.IScheduler` interface, and is functionally equivalent to the `DefaultScheduler` implementation that ships with Alchemi, modulo some extra debugging and logging facilities. We developed `PSLScheduler` solely to demonstrate a successful swap, not to actually improve scheduling.

`ReconfigEngine` is responsible for caching instances of the `Manager` classes of interest, `ManagerContainer` and `InternalShared`, as well as effecting the scheduler swap. It is implemented according to the singleton design pattern. To effect changes on the `ManagerContainer` and `InternalShared` instances, the `ReconfigEngine` relies heavily on the Reflection API since many of the key variables are private and in some cases readonly. The `ReconfigEngine` sets up a communication channel after it has attached to the `Manager`, which allows a `Reconfiguration Console` to send commands to the `ReconfigEngine` to trigger reconfigurations (this experiment did not include monitoring for those conditions under which a different scheduler would be warranted). Table 1.1 shows the method signatures of the `ReconfigEngine` API.

Method
<code>public static ReconfigEngine GetInstance()</code>
<code>public static void CacheManagerContainer(object o)</code>
<code>public static void CacheInternalShared(object o)</code>
<code>public void SwapScheduler()</code>

Table 1.1: Reconfiguration Engine API

1.5 Empirical Evaluation

1.5.1 Experimental Setup

The testbed for our experiment is an Alchemi cluster consisting of two Executors (Pentium-4 3GHz desktop machines with 1GB RAM running Windows XP SP2 and the .NET Framework v1.1.4322) and a single Manager (Pentium-III 1.2GHz laptop with 1GB RAM running Windows XP SP2 and the .NET Framework v1.1.4322).

We ran the `PiCalculator` sample grid application, which ships with Alchemi, multiple times while requesting that the scheduler implementation be changed during the application's execution. The `PiCalculator` application computes the value of Pi to `n` decimal digits. In our tests we used the default `n=100`.

In our tests we swapped between the `DefaultScheduler` and `PSLScheduler`. The two schedulers are functionally equivalent. However, `PSLScheduler` outputs extra logging information to the Alchemi Manager GUI so that we could confirm that a scheduler swap actually occurred.

1.5.2 Results

The first thing we measured was the time taken to swap the scheduler. We requested scheduler swaps between runs of the the PiCalculator application. In our experiment the time taken to replace the scheduler instance is about 500 ms on average, however that time is dominated by the time spent waiting for the scheduler thread to exit. In the worst case, a scheduler-swap request arrives while the scheduler thread is sleeping (as it is programmed to do for up to 1000 ms on every loop iteration), causing the request to wait until the thread resumes and exits before it is honored. As a result we consider the time taken to actually effect the scheduler swap (modulo the time spent waiting for the scheduler thread to exit) to be negligible.

Table 1.2 compares the job completion times when no scheduler swap requests are submitted during execution of the PiCalculator grid application with job completion times when one or more scheduler swap requests are submitted. As expected, the difference in job completion times is negligible, $\sim 1\%$, since the scheduler implementations are functionally equivalent. Further, swapping the scheduler has no impact on on-going execution of the Executors, as an Executor is not assigned an additional work unit (grid thread) until it is finished executing its current work unit.

run#	Job Completion time (ms) w/o swap	Job Completion time (ms) w/swap	#Swaps
1	18.3063232	17.2748400	2
2	18.3163376	18.4665536	1
3	18.3363664	17.3148976	4
4	18.3463808	17.3148976	2
5	18.3063232	17.4150416	2
6	17.4250560	18.2662656	2
7	18.3463808	18.3163376	4
8	17.5352144	18.5266400	1
9	17.5252000	18.4965968	2
10	18.3363664	18.3463808	2
Avg	18.07799488	17.97384512	2.2

Table 1.2: PiCalculator.exe Job Completion Times

Based on our experiment we are able to demonstrate that we can in fact perform a consistency-preserving reconfiguration of the Alchemi Grid Manager without compromising the integrity of the CLR, the Alchemi Grid Manager and by extension the Alchemi Grid and jobs actively executing in the grid.

1.6 Related Work

The techniques (bytecode rewriting, metadata augmentation and method call interposition) used by our adaptation framework to attach/detach a reconfiguration en-

gine to/from an application running in a managed execution environment are similar to techniques used by dynamic Aspect Oriented Programming (AOP) engines. In general, AOP is an approach to designing software that allows developers to modularize cross-cutting concerns [12] 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. In the context of adaptive systems, AOP is an approach to designing the system such that the non-functional requirement of having adaptation mechanisms available is cleanly separated from the system’s functional logic. 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 our case inserting hooks where reconfiguration and repair actions can be performed.

There are three kinds of AOP engines: those that perform weaving at compile time (static weaving), e.g., AspectJ [23] and Aspect C# [24]; those that perform weaving after compile time but before load time, e.g., Weave .NET [25] and Aspect.NET [26], which pre-process .NET assemblies, operating directly on type and assembly metadata; and those that perform weaving at runtime (dynamic weaving) at the bytecode level, e.g., A dynamic AOP-Engine for .NET [27] and CLAW [28]. Our adaptation framework prototype exhibits analogous dynamic weaving functionality.

A Dynamic AOP-Engine for .NET 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. CLAW 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, so we were unable to investigate its capabilities and implementation details.

Effecting runtime reconfigurations in software systems falls under the topic of *change management* [29]. Change management is a principled aspect of runtime system evolution that helps identify what must be changed, provides a context for reasoning about, specifying and implementing change, and controls change to preserve system integrity as well as meeting extra-functional requirements such as availability, reliability, etc.

A number of existing systems support runtime reconfiguration at various granularities. The Dynamically Alterable System (DAS) operating system [30] provides support for reconfiguring applications by letting a module be replaced by another module with the same interface. DAS’ replugging mechanism requires special memory addressing hardware and a complex virtual-memory architecture to work. The DMERT operating system [31] supports the reconfiguration of the C functions that make up the switching software running on AT&T’s 3B20D processor. Whole procedures can be interchanged provided that the function signature remains constant. DMERT uses a level of indirection between a function call and the actual target

of a function in memory. It is, however, very specific to the telecommunications application domain. K42 [32] is an example of an operating system that supports reconfiguration of its constituent components by virtue of its design. Explicit component boundaries, a notion of quiescent states, support for state transfers between functionally compatible components, and indirection mechanisms for accessing system components all play a role in supporting reconfigurations such as component swaps and object interposition.

Argus [33] supports coarse-grained reconfigurations in distributed systems. Argus is a language based on Clu [34] and an underlying operating system. Argus' unit of reconfiguration is a "guardian" – a server that implements a set of functions via a set of handlers. The approaches and techniques for reconfiguring a system are tightly tied to the Argus system and language. Conic [29, 35] provides a powerful environment for reconfiguring distributed systems following the change management model. However, it also restrains the language and runtime system.

1.7 Conclusions and Future Work

We describe an adaptation framework that uses facilities of a managed execution environment to allow us to transparently attach/detach a reconfiguration engine to/from a target system executing in that managed execution environment. We also present an example of using our adaptation framework prototype to effect consistency-preserving reconfigurations in the Alchemi Enterprise Grid Computing System. We leverage easily-obtained knowledge of the Alchemi system to identify "safe" control-points during program execution where reconfiguration actions can be performed. This approach to change management [29] is motivated by the results of Gupta et al. [36], who present a proof of the undecidability of automatically finding all the control-points in an application where a consistency-preserving adaptation can be performed.

Our proof-of-concept experiment shows the feasibility of using managed execution environment facilities to effect runtime reconfiguration on a legacy target system. For future work we seek to apply our approach to other managed execution environments, e.g., the Jikes Research Virtual Machine (RVM) [37]. Further, we are interested in investigating how our adaptation framework could be used to effect fine-grained reconfigurations or repairs co-ordinated by an existing externalized adaptation architecture such as Rainbow [10] or KX [9].

1.8 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.

References

- [1] The University of Melbourne, Alchemi – Plug & Play Grid Computing. Available at <http://www.alchemi.net/>.
- [2] Jeffrey O. Kephart and David M. Chess, The Vision of Autonomic Computing, In *IEEE Computer magazine*, pages 41–52, January 2003.
- [3] George Candea et al., Improving Availability with Recursive Micro-Reboots: A Soft-State Case Study, In *Dependable Systems and Networks – Performance and Dependability Symposium (DNS-PDS)*, pages 213–248, June 2002.
- [4] Mark E. Segal and Ophir Frieder, On-The-Fly Program Modification Systems for Dynamic Updating, In *IEEE Software magazine*, pages 53–65, March 1993.
- [5] Charles Shelton and Philip Koopman, Using Architectural Properties to Model and Measure System-wide Graceful Degradation, In *Proceedings of ICSE Workshop on Architecting Dependable Systems (WADS)*, pages 267–289, May 2002.
- [6] Philip Koopman, Elements of the Self-Healing System Problem Space, In *Proceedings of ICSE Workshop on Architecting Dependable Systems (WADS)*, pages 31–36, May 2003.
- [7] G. Eisenhauer and K. Schwan, An Object-Based Infrastructure for Program Monitoring and Steering, In *Proceedings of the 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT98)*, pages 10–20, August 1998.
- [8] S.M. Sadjadi and P.K. McKinley, Transparent Self-Optimization in Existing CORBA Applications, In *Proceedings of the 1st IEEE International Conference on Autonomic Computing*, pages 88–95, May 2004.
- [9] Gail Kaiser et al., Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems, In *Proceedings of The Autonomic Computing Workshop 5th Workshop on Active Middleware Services (AMS)*, pages 22–30, June 2003.
- [10] Shang-Wen Cheng et al., Rainbow: Architecture-based Self-Adaptation with Reusable Infrastructure, In *IEEE Computer magazine*, pages 46–54, October 2004.
- [11] Bradley Schmerl and David Garlan, Exploiting Architectural Design Knowledge to Support Self-Repairing Systems, In *Proceedings of the 14th International Conference of Software Engineering and Knowledge Engineering*, pages 241–248, July 2002.

- [12] Gregor Kiczales et al., Aspect-Oriented Programming, In *Proceedings of European Conference on Object-Oriented Programming*, pages 220–242, June 1997.
- [13] Rean Griffith and Gail Kaiser, Manipulating Managed Execution Runtimes to Support Self-Healing Systems, In *Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software* pages 1–7, May 2005.
- [14] Rean Griffith and Gail Kaiser, Adding Self-healing Capabilities to the Common Language Runtime, Technical Report CUCS-005-05, Department of Computer Science, Columbia University in the City of New York, February 2005. Available at <http://www.cs.columbia.edu/techreports/cucs-005-05.pdf>.
- [15] Serge Lidin, Inside Microsoft .NET IL Assembler, *Microsoft Press*, 2002.
- [16] Microsoft, Common Language Runtime Profiling, 2002.
- [17] Microsoft, Common Language Runtime Metadata Unmanaged API, 2002.
- [18] Aleksandr Mikunov, Rewrite MSIL Code on the Fly with the .NET Framework Profiling API, *MSDN Magazine*, September 2003. Available at <http://msdn.microsoft.com/msdnmag/issues/03/09/NETProfilingAPI/>.
- [19] Akshay Luther et al., Alchemi: A .NET-Based Enterprise Grid Computing System, In *Proceedings of the 6th International Conference on Internet Computing (ICOMP'05)*, June 2005.
- [20] Project: Alchemi [.NET Grid Computing Framework]: Summary. Available at <http://sourceforge.net/projects/alchemi/>.
- [21] Ingo Rammer, Advanced .NET Remoting (C# Edition) (Paperback), Apress, April 2002.
- [22] Akshay Luther et al., Alchemi: A .NET-based Enterprise Grid System and Framework, User Guide for Alchemi 1.0, July 2005. Available at <http://www.alchemi.net/files/1.0.beta/docs/AlchemiManualv.1.0.htm>.
- [23] G. Kiczales et al., An Overview of AspectJ, In *Proceedings of European Conference on Object-Object Programming*, pages 327–353, June 2001.
- [24] Howard Kim, AspectC#: An AOSD implementation for C#, *Masters Thesis*, Department of Computer Science, Trinity College, Dublin, September 2002. Available at <https://www.cs.tcd.ie/publications/tech-reports/reports.02/TCD-CS-2002-55.pdf>.
- [25] Donal Lafferty et al., Language Independent Aspect-Oriented Programming, In *Proceedings of the 18th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 1–12, October 2003.
- [26] Bjorn Rasmussen et al., Aspect.NET - A Cross-Language Aspect Weaver, Department of Computer Science, Trinity College, Dublin, 2002.

- [27] Andreas Frei et al., A Dynamic AOP-Engine for .NET, Technical Report 445, Department of Computer Science, ETH Zurich, May 2004. Available at <http://www.iks.inf.ethz.ch/publications/files/daopnet.pdf>.
- [28] John Lam, CLAW: Cross-Language Load-Time Aspect Weaving on Microsoft's Common Language Runtime Demonstration at the 1st International Conference on Aspect-Oriented Software Development, April 2002. Available at <http://trese.cs.utwente.nl/aosd2002/index.php?content=clawclr>.
- [29] J. Kramer and J. Magee, The Evolving Philosophers Problem: Dynamic Change Management, In *IEEE Transactions on Software Engineering*, pages 1293–1306, November 1990.
- [30] Hannes Goullon et al., Dynamic Restructuring in an Experimental Operating Systems, In *IEEE Transactions on Software Engineering*, pages 298–307, July 1978.
- [31] R. Yacobellis et al., The 3B20D Processor and DMERT Operating System: Field Administration Sub-system, Bell Systems Technical Journal, pages 323–339, January 1983.
- [32] C. Soules et al., System Support for Online Reconfiguration, In *Proceedings of USENIX Annual Technical Conference*, pages 141–154, June 2003.
- [33] Toby Bloom, Dynamic Module Replacement in a Distributed Programming System, *PhD Thesis*, Technical Report MIT/LCS/TR-303, MIT Laboratory for Computer Science, Cambridge MA, March 1983. Available at <http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-303.pdf>.
- [34] Barbara Liskov, A History of CLU, Technical Report MIT/LCS/TR-561, MIT Laboratory for Computer Science, Cambridge MA, April 1992. Available at <http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-561.pdf>.
- [35] Jeff Magee et al., Constructing Distributed Systems in Conic, In *IEEE Transactions on Software Engineering*, pages 663–675, June 1989.
- [36] Deepak Gupta et al., A Formal Framework for On-line Software Version Change, In *IEEE Transactions on Software Engineering*, pages 120–131, February 1996.
- [37] Jikes. Available at <http://jikes.sourceforge.net/>.