Evaluating Software Systems via Fault-Injection and Reliability, Availability and Serviceability (RAS) Metrics and Models

Thesis proposal

Rean Griffith
Department of Computer Science
Columbia University
1214 Amsterdam Avenue
Mailcode 0401
New York, NY 10027
+1.212.939.7184
rg2023@cs.columbia.edu

Advisor: Gail E. Kaiser

February 14, 2007

Abstract

The most common and well-understood way to evaluate and compare computing systems is via performance-oriented benchmarks. However, numerous other demands are placed on computing systems besides speed. Current generation and next generation computing systems are expected to be reliable, highly available, easy to manage and able to repair faults and recover from failures with minimal human intervention.

The extra-functional requirements concerned with reliability, high availability, and serviceability (manageability, repair and recovery) represent an additional set of high-level goals the system is expected to meet or exceed. These goals govern the system's operation and are codified using policies and service level agreements (SLAs).

To satisfy these extra-functional requirements, system-designers explore or employ a number of mechanisms geared towards improving the system's reliability, availability and serviceability (RAS) characteristics. However, to evaluate these mechanisms and their impact, we need something more than performance metrics.

Performance-measures are suitable for studying the feasibility of the mechanisms i.e. they can be used to conclude that the level of performance delivered by the system with these mechanisms active does not preclude its usage. However, performance numbers convey little about the efficacy of the systems RAS-enhancing mechanisms. Further, they do not allow us to analyze the (expected or actual) impact of individual mechanisms or make comparisons/discuss tradeoffs between mechanisms.

What is needed is an evaluation methodology that is able to analyze the details of the RAS-enhancing mechanisms – the micro-view as well as the high-level goals, expressed as policies, SLAs etc., governing the system's operation – the macro-view. Further, we must establish a link between the details of the mechanisms and their impact on the high-level goals. This thesis is concerned with developing the tools and applying analytical techniques to enable this kind of evaluation. We make three contributions.

First, we contribute to a suite of runtime fault-injection tools with Kheiron. Kheiron demonstrates a feasible, low-overhead, transparent approach to performing system-adaptations in a variety of execution environments at runtime. We use Kheiron's runtime-adaptation capability to inject faults into running programs. We present three implementations of Kheiron, each targeting a different execution environment. Kheiron/C manipulates compiled C-programs running in an unmanaged execution environment – comprised of the operating system and the underlying processor. Kheiron/CLR manipulates programs running in Microsoft's Common Language Runtime (CLR) and Kheiron/JVM manipulates programs running in Sun Microsystems' Java Virtual Machine (JVM). Kheiron's operation is transparent to both the application and the execution environment. Further, the overheads imposed by Kheiron on the application and the execution environment are negligible, $<5\%$, when no faults are being injected.

Second, we describe analytical techniques based on RAS-models, represented as Markov chains and Markov reward models, to demonstrate their power in evaluating RAS-mechanisms and their impact on the high-level goals governing system-operation. We demonstrate the flexibility of these models in evaluating reactive, proactive and preventative mechanisms as well as their ability to explore the feasibility of yet-to-be-implemented mechanisms. Our analytical techniques focus on remediations rather than observed mean time to failures (MTTF). Unlike hardware, where the laws of physics govern the failure rates of mechanical and electrical parts, there are no such guarantees for software failure rates. Software failure-rates can however be influenced using fault-injection, which we employ in our experiments. In our analysis we consider a number of facets of remediations, which include, but go beyond mean time to recovery (MTTR). For example we consider remediation success rates, the (expected) impact of preventative-maintenance

and the degradation-impact of remediations in our efforts to establish a framework for reasoning about the tradeoffs (the costs versus the benefits) of various remediation mechanisms.

Finally, we distill our experiences developing runtime fault-injection tools, performing fault-injection experiments and constructing and analyzing RAS-models into a 7-step process for evaluating computing systems – the 7U-evaluation methodology. Our evaluation method succeeds in establishing the link between the details of the low-level mechanisms and the high-level goals governing the system's operation. It also highlights the role of environmental constraints and policies in establishing meaningful criteria for scoring and comparing these systems and their RAS-enhancing mechanisms.

Contents

1 Introduction

Measuring a system's performance is the most well-understood approach to evaluating and comparing computing systems. Researchers routinely use traditional performance benchmarks produced by organizations including the National Institute of Science and Technology (NIST) [63], the Standard Performance Evaluation Corporation (SPEC®) [77] and the Transaction Processing and Performance Council (TPC) [84], to demonstrate the feasibility of some experimental system prototype. However, there are a number of other demands placed on computing systems besides being fast. We expect current and next generation computing systems to be reliable, highly available, easy to manage and able to repair faults and recover from errors with minimal human intervention.

Extra-functional requirements concerned with reliability, high availability, and serviceability (manageability, repair and recovery) represent additional high-level goals the system is expected to meet or exceed. These goals are codified in policies and service level agreements (SLAs), which govern the system's operation.

In an attempt to satisfy these extra-functional requirements, system designers explore or employ mechanisms geared towards improving the system's reliability, availability and serviceability (RAS) characteristics. However, to reason about tradeoffs between mechanisms or to evaluate these mechanisms and their impact we need something other than performance metrics.

Whereas performance metrics are suitable for studying the feasibility of having RAS-enhancing mechanisms activated, i.e. to demonstrate that the system provides "acceptable" performance with these mechanisms enabled, the performance numbers convey little about the efficacy of the mechanisms. Further, performance measures do not allow us to analyze the expected or actual impact of individual mechanisms on the system's RAS-profile, compare the efficacy of individual mechanisms or reason about tradeoffs between individual mechanisms.

What we need is an evaluation methodology, that allows us to analyze the details of RAS-enhancing mechanisms (the micro-view) as well as the high-level goals governing the system's operation (the macro-view). We must establish a link between the details of the mechanisms and their (expected or actual) impact on the high-level goals. This link serves to justify the addition of the mechanism and enables us to reason about whether the mechanism is useful to the system i.e. whether the mechanism helps the system to meet or exceed the high-level goals set for it.

To determine a system's RAS-characteristics and investigate the impact of its existing (or yet-to-be-added) RAS-enhancing mechanisms we require two things; runtime fault-injection tools and a rigorous analytical framework. Runtime fault-injection tools allow us to study the impact of faults on a system and the system's failure behavior in its deployed environment. These tools allow us to exercise any RAS-enhancing mechanisms they system may have. A rigorous analytical framework allows us to quantify the expected or actual impact of existing or yet-to-be-added RAS-mechanisms and helps to define objective criteria for comparing mechanisms.

This thesis is concerned with developing the fault-injection tools and applying analytical techniques to enable the evaluation of the RAS-characteristics of the popular N-tier web-application stack via fault-injection experiments that target commodity operating systems, application servers and relational database systems. We make three contributions.

First, we contribute to a suite of runtime fault-injection tools with Kheiron. Kheiron demonstrates

a feasible, low-overhead, transparent approach to performing system-adaptations in a variety of execution environments at runtime. We use Kheiron's runtime-adaptation capability to inject faults into running programs. We present three implementations of Kheiron, each targeting a different execution environment. Kheiron/C [27] manipulates compiled C-programs running in an unmanaged execution environment – comprised of the operating system and the underlying processor. Kheiron/CLR [25, 3] manipulates programs running in Microsoft's Common Language Runtime (CLR) and Kheiron/JVM [27] manipulates programs running in Sun Microsystems' Java Virtual Machine (JVM). Kheiron's operation is transparent to both the application and the execution environment. Further, the overheads imposed by Kheiron on the application and the execution environment are negligible, $<5\%$, when no faults are being injected.

Second, we describe analytical techniques based on RAS-models, represented as Markov chains and Markov reward models, to demonstrate their power in evaluating RAS-mechanisms and their impact on the high-level goals governing system-operation. We demonstrate the flexibility of these models in evaluating reactive, proactive and preventative mechanisms as well as their ability to explore the feasibility of yet-to-be-implemented mechanisms. Our analytical techniques focus on remediations rather than observed mean time to failures (MTTF). Unlike hardware, where the laws of physics govern the failure rates of mechanical and electrical parts, there are no such guarantees for software failure rates. Software failure-rates can however be influenced using fault-injection, which we employ in our experiments. In our analysis we consider a number of facets of remediations, which include, but go beyond mean time to recovery (MTTR). For example, we consider remediation success rates, the (expected) impact of preventative-maintenance and the degradation-impact of remediations in our efforts to establish a framework for reasoning about the tradeoffs (the costs versus the benefits) of various remediation mechanisms.

Finally, we distill our experiences developing runtime fault-injection tools, performing fault-injection experiments and constructing and analyzing RAS-models into a 7-step process for evaluating computing systems – the 7U-evaluation methodology. Our evaluation method succeeds in establishing the link between the details of the low-level mechanisms and the high-level goals governing the system's operation. It also highlights the role of environmental constraints and policies in establishing meaningful criteria for scoring and comparing these systems and their RAS-enhancing mechanisms.

2 Problem, Definitions and Requirements

2.1 Definitions

This section formalizes some of the terms used throughout this proposal.

- An **error** is the deviation of system external state from correct service state [43].
- A **fault** is the adjudged or hypothesized cause of an error [43].
- The **fault hypothesis/fault model** is the set of faults a system is expected to be able to respond to with a reactive, proactive or preventative action [40].
- **Remediation** is the process of correcting a fault. In this paper remediation spans the activities of detection, diagnosis and repair since the first step in responding to a fault is detection [40].

- A **failure** is an event that occurs when the delivered service violates an environmental/contextual constraint e.g. a policy or SLA. This definition emphasizes the client-side (end-user's) perspective over the server-side perspective [5].
- **Reliability** is the number (or frequency) of client-side interruptions.
- **Availability** is a function of the rate of failure/maintenance events and the speed of recovery [35].
- **Serviceability** is a function of the number of service-visits and their duration and costs.
- An **existing/legacy system** is any system for which the source code may not be available or for which it is undesirable to engage in substantial re-design and development.
- An **execution environment** is responsible for the preparation for distinguished entities – *executables* – such that they can be run. Preparation in this context involves the loading and laying out in memory of an executable. The level of sophistication, in terms of services provided by the execution environment beyond loading, depends largely on the *type* of executable.
- A **managed execution environment**, e.g. Sun Microsystems' Java Virtual Machine (JVM) or Microsoft's Common Language Runtime (CLR), is responsible not only for loading and running *managed executables*, but for providing additional application services, including but not limited to: garbage collection, application isolation, security sandboxing and structured exception handling. These application services are typically geared towards enhancing the robustness of applications. Managed execution environments are typically implementations of an abstract machine with its own "specialized" instruction set and rules about the content/packaging of managed executables [46, 55].
- A **managed executable/application** is represented in an abstract intermediate form expected by the managed execution environment. This abstract intermediate form consists of *metadata* and *managed code*. Metadata describes the structural aspects of the application, including classes, their members and attributes, and their relationships with other classes [45]. Managed code represents the functionality of the application's methods encoded in an abstract binary form, *bytecode*, conforming to the specialized instruction set expected by the managed execution environment.
- An **unmanaged execution environment** consists of the underlying processor (e.g. IA-32/x86) and the operating system (e.g. Linux).
- An **unmanaged/native executable** also contains metadata, albeit not as rich as their managed counterparts. Compiled C/C++ programs may contain symbol information, however there is neither a guarantee nor requirement that it be present. Further, unmanaged/native executables contain instructions that can be directly executed on the underlying processor (hence the use of the term native) whereas the bytecode found in managed executables must be interpreted or Just-In-Time (JIT) compiled into processor instructions by a component of the managed execution environment.

2.2 Problem Statement

Performance-metrics and performance-oriented benchmarks are not the most effective way to evaluate systems given the extra-functional demands concerning reliability, availability and serviceability placed on them. As a result, we require an evaluation methodology that allows for the analysis of the existing (or yet-to-be-added) RAS-enhancing mechanisms and their impact on the high-level goals (expressed as policies and SLAs) governing the system's operation.

To study the RAS-characteristics of computing systems, we require runtime fault-injection tools and a rigorous analytical framework to evaluate and compare RAS-enhancing mechanisms. Existing/legacy systems may not have (any or all of the) built-in fault-injection mechanisms needed to conduct an evaluation, as a result we may be required to retro-fit these mechanisms – which is preferable to rebuilding and redeploying specialized versions of target systems. The analytical techniques should be capable of evaluating reactive, proactive and preventative RAS-enhancing mechanisms.

2.3 Requirements

Based on the above definitions and initial problem statement, we establish a set of requirements needed to effectively solve the problem.

1. **Support fault-injection in a variety of target systems**. The approach to fault-injection should not restrict the choice of target system or execution environment. Further, we must be able to collect information on the target system's execution such that it enables a basic understanding of the target system's operation and informs planning the introduction, modification, replacement or removal of fault-injection mechanisms dynamically.
2. **Support the efficient and transparent introduction of fault-injection mechanisms into existing/legacy systems**. No changes to the application's source code and/or the execution environment where it runs should be necessary. Neither should specialized execution environments or runtimes be required. Where necessary, existing extension mechanisms in both the applications and the execution environments may be used. This transparency allows us to study the failure behavior in the released/deployed version of the system, rather than a specially built version of the system.
3. **Support the flexible modification/reversal of the dynamic adaptations that insert fault-injection mechanisms**. This flexibility allows us to dynamically improve or remove the fault-injection mechanisms present in the system. New fault-injection mechanisms may complement or replace existing fault-injection mechanisms.
4. **Support the analysis of existing and yet-to-be-implemented reactive, proactive and preventative RAS-enhancing mechanisms**. Objective and quantitative measures should be devised such that mechanisms can be compared and evaluated.
5. **Support evaluating the impact of individual and combined RAS-enhancing mechanisms on the system's high-level goals**. Establish a link between the mechanisms and the policies and SLAs that govern the system's operation and constrains the choice of mechanisms.

2.4 Hypotheses

This thesis investigates the following hypotheses:

1. Runtime adaptation is a reasonable technology for implementing efficient and flexible fault-injection tools.
2. Continuous Time Markov Chains (CTMCs) and Markov Reward Models are a reasonable framework for analyzing system failures, remediation mechanisms and their impact on system operation.
3. RAS-models and fault-injection experiments can be used together to model and measure the RAS-characteristics of computing systems. This combination links the details of the mechanisms to the high-level goals governing the systems operation, supporting comparisons of individual or combined mechanisms.

3 Dynamic Fault-Injection via Runtime Adaptations

3.1 Introduction

The need for software to evolve as its usage and operational goals change has added the non-functional requirement of adaptation to the list of facilities expected in systems. Example system-adaptations include, but are not limited to, the ability to support reconfigurations, repairs, self-diagnostics or user-directed evaluations driven by fault-injection.

However, not all systems have the built-in facilities to support many of the desired system-adaptations. System designers have two alternatives when it comes to realizing software systems capable of adaptation. Adaptation mechanisms can be *static* i.e. built into the system, as is done in the K42 operating system [8], or such functionality can be *dynamically added* i.e. retro-fitted onto them using externalized architectures like KX [23] or Rainbow [72].

While arguments can be made for either approach, the retrofit approach provides more flexibility. Static system-adaptations force the system to be taken offline, rebuilt and restarted/redeployed to add, modify or remove mechanisms whereas dynamic adaptations allow mechanisms to be added, modified or removed while the system executes. The ability to keep the system running while adaptations occur make dynamic adaptations preferable to their static counterparts [74, 40, 70]. Further, "baked-in" adaptation mechanisms restrict the analysis and reuse of said mechanisms.

With any system there is a spectrum of adaptations that can be performed. Frameworks like KX perform coarse-grained adaptations e.g. re-writing configuration files and restarting/terminating operating system processes. However, in this proposal, we focus on fine-grained adaptations, those interacting with individual components, sub-systems or methods e.g. augmenting these elements at runtime to support reconfigurations, repairs, self-diagnostics or user-directed evaluations driven by fault-injection.

In this section we describe the technologies underlying Kheiron, a framework for facilitating adaptations in running programs in a variety of execution environments with low-overhead, upon which we build the dynamic fault-injection tools used in Chapter 4. The fault-injection tools we build are one example of *software-implemented fault-injection tools* [33].

For software-implemented fault-injection tools there are a number of benefits realized by building them on top of a dynamic-adaptation framework like Kheiron.

1. Unlike FAUMachine [76], Ferrari [36] and Ftape [85], which are limited to injecting bit flips in CPU registers, memory addresses and emulating disk I/O errors, using Kheiron's capabilities we build fault-injection tools that can inject more fine-grained faults targeting individual components, subsystems, methods and data structures e.g. removing components, inserting delays or hangs, modifying specific fields of data structures/objects or inducing resource leaks.

2. Unlike Doctor [30], which uses compile-time program modifications to insert the fault-injection mechanisms, Kheiron's ability to dynamically add and remove mechanisms allows us the flexibility to manage the performance overhead of persistent fault-injection mechanisms by dynamically removing them. Further, new fault-injection mechanisms can be added on-the-fly.

3. Unlike Xception [48], which depends on the low-level facilities of the PowerPC processor, Kheiron's ability to support the insertion of fault-injection mechanisms does not rely on specific debugging or performance monitoring facilities of the x86 processor.

4. Unlike FIST (Fault Injection System for Study of Transient Fault Effect) [29] and MARS (Maintainable Real-Time System) [37], fault-injection tools built using Kheiron we do not require special hardware to induce faults. FIST and MARS use hardware that generates ion radiation and electromagnetic fields to induce faults in target systems.

5. Holodeck [71] interposes between the application and the operating system. As a result, it induces faults in the application indirectly. For example, it can corrupt files, corrupt network packets, intercept/redirect system calls etc. However, fault-injection tools built on top of Kheiron can inject faults directly into the application itself as well as its environment (i.e. the operating system or managed execution environment), thereby expanding the set of potential targets for fault-injection.

6. Jaca [50] is a fault-injection tool intended to validate Java applications. Jaca injects high-level faults affecting attributes, and methods of an object's public interface via load-time bytecode rewriting. The faults injected by Jaca include corrupting method attributes, parameters and return values. In addition to performing load-time bytecode changes like Jaca, fault-injection tools built using Kheiron are also able to perform runtime changes that add, augment or remove fault-injection mechanisms. Further, Kheiron supports the adaptations of applications written in a broader set of languages including C, Java and languages targeting Microsoft's CLR e.g. C#, VB .NET etc.

Kheiron supports a variety of application-types and execution environments. It manipulates compiled C-programs running in an unmanaged execution environment as well as programs running in Microsoft's Common Language Runtime and Sun Microsystems' Java Virtual Machine. We present case-studies and experiments that demonstrate the feasibility of using Kheiron to support fine-grained runtime system-adaptations. We also describe the concepts and techniques used to retro-fit adaptations onto existing systems in the various execution environments.

Managing the performance impact of the mechanisms used to effect fine-grained adaptations in the running system presents an additional challenge. Since we are interacting with individual methods or components we must be cognizant of the performance impact of effecting the adaptations e.g.

inserting instrumentation into individual methods may slow down the system; but being able to selectively add/remove instrumentation allows the performance impact to be tuned throughout the system's execution.

This section is primarily concerned with addressing the challenges of efficiently retro-fitting fine-grained adaptation mechanisms onto existing software systems and managing the performance impacts associated with retro-fitting these adaptation mechanisms. We posit that we can leverage the unmodified execution environment to transparently facilitate the adaptations of existing/legacy systems. We describe three systems we have developed for this purpose. **Kheiron/C** manipulates running compiled C programs on the Linux platform, **Kheiron/CLR** manipulates running .NET applications and finally **Kheiron/JVM** manipulates running Java applications.

Our contribution is the ability to transparently retro-fit new functionality onto existing software systems. The techniques used to facilitate the retro-fit exhibit negligible performance overheads on the running systems. Finally, our techniques address effecting adaptations in a variety of contemporary execution environments. New functionality, packaged in separate modules, collectively referred to as an *adaptation engine*, is loaded by Kheiron. At runtime, Kheiron can seamlessly transfer control over to the adaptation engine, which effects the desired adaptations in the running application.

3.2 Motivation

The ability to adapt is critical for systems [38]. However, not every system is designed or constructed with all the adaptation mechanisms it will ever need. As a result, there needs to some way to enable existing applications to introduce and employ new mechanisms.

There are a number of specific fine-grained adaptations that can be retro-fitted onto existing systems including, but not limited to, adding fault-injection, problem detection, diagnosis and in some cases remediation mechanisms.

In this section we describe how our Kheiron implementations can be used to facilitate a number of fine-grained adaptations in running systems via leveraging facilities and properties of the execution environments hosting these systems. These adaptations include (but are not limited to): **Inserting or removing system instrumentation** [62] to discover performance bottlenecks in the application or detect (and where possible repair) data-structure corruption. The ability to remove instrumentation can decrease the performance impact on the system associated with collecting information. **Periodic refreshing** of data-structures, components and subsystems done using micro-reboots, which could be performed at a fine granularity e.g., restarting individual components or sub-systems, or at a coarse granularity e.g., restarting entire processes periodically. **Replacing** failed, unavailable or suspect components and subsystems (where possible) [28]. **Input filtering/audit** to detect misused APIs. **Inserting faults or initiating ghost transactions** against select components or subsystems and collecting the results to obtain more details about a problem or investigate a system response. **Selective emulation of functions** – effectively running portions of computation in an emulator, rather than on the raw hardware to detect errors and prevent them from crashing the application.

3.3 Challenges of Runtime Adaptation via the Execution Environment

There are a number of properties of execution environments that make them attractive for effecting adaptations on running systems. They represent the lowest level (short of the hardware) at which changes could be made to a running program. Some may expose (reasonably standardized) facilities (e.g. profiling APIs [57, 60]) that allow the state of the program to be queried and manipulated. Further, other facilities (e.g. metadata APIs [56]) may support the discovery, inspection and manipulation of program elements e.g. type definitions and structures. Finally, there may be mechanisms which can be employed to alter to the execution of the running system.

However, the low-level nature of execution environments also makes effecting adaptations a risky (and potentially arduous) exercise. Injecting and effecting adaptations must not corrupt the execution environment nor the system being adapted. The execution environment's rules for what constitutes a "valid" program must be respected while guaranteeing consistency-preserving adaptations in the target software system. Causing a crash in the execution environment typically has the undesirable side-effect of crashing the target application and any other applications being hosted.

At the level of the execution environment the programming-model used to specify adaptations may be quite different from the one used to implement the original system. For example, to effect changes via an execution environment, those changes may have to be specified using assembly instructions (moves and jump statements), or bytecode instructions where applicable, rather than higher level language constructs. This disconnect may limit the kinds of adaptations which can be performed and/or impact the mechanism used to inject adaptations.

3.4 Hypothesis

The main hypotheses investigated in this section are:

1. **The execution environment is a feasible target for efficiently and transparently effecting adaptations in the applications they host**. All software systems run in an execution environment, as a result we can target the execution environment as the lowest common denominator for adapting live systems.

2. **Existing facilities in execution environments can be leveraged to effect runtime adaptations in software systems.** Built-in facilities for profiling, execution control and any available APIs for metadata querying or manipulation allow for a transparent and sufficiently low-overhead approach to adapting running programs. Two adaptations of interest for the purposes of this thesis are: the insertion of monitoring/instrumentation, and the insertion of faults/disturbances to measure their effects on systems with/without appropriate remediation mechanisms.

3. **Any guarantees on application integrity/consistency are a function of the execution environment, the execution environment's operation and the amount of knowledge we have about the application's operation**. The ability to perform adaptations on running systems allows for a great degree of flexibility. On-the-fly adaptations allow the system to remain available (even if it operates in a degraded mode) during these changes. However, the greatest challenge is preserving the integrity/consistency during and after adaptations. We posit that properties of the execution environment and working knowledge of the target system's operation can be combined to guarantee that the application's integrity is preserved during and after adaptations.

3.5 Execution Environments

At a bare minimum, an execution environment is responsible for the preparation of distinguished entities – *executables* – such that they can be run. Preparation, in this context involves the loading and laying out in memory of an executable. The level of sophistication, in terms of services provided by the execution environment beyond loading, depends largely on the *type* of executable.

We distinguish between two types of executables, *managed* and *unmanaged* executables, each of which require or make use of different services provided by the execution environment. A managed executable, e.g. a Java bytecode program, runs in a *managed execution environment* such as Sun Microsystems' JVM whereas an unmanaged executable, e.g. a compiled C program, runs in an *unmanaged execution environment* which consists of the operating system and the underlying processor. Both types of executables consist of metadata and code. However the main differences are the amount and specificity of the metadata present and the representation of the instructions to be executed.

Managed executables/applications are represented in an abstract intermediate form expected by the managed execution environment. This abstract intermediate form consists of two main elements, *metadata* and *managed code*. Metadata describes the structural aspects of the application including classes, their members and attributes, and their relationships with other classes [45]. Managed code represents the functionality of the application's methods encoded in an abstract binary format known as *bytecode*.

The metadata in unmanaged executables is not as rich as the metadata found in managed executables. Compiled C/C++ programs may contain symbol information, however there is neither a guarantee nor requirement that it be present. Finally, unmanaged executables contain instructions that can be directly executed on the underlying processor unlike the bytecode found in managed executables, which must be interpreted or Just-In-Time (JIT) compiled into native processor instructions.

Managed execution environments differ substantially from unmanaged execution environments[1]. The major differentiation points are the metadata available in each execution context and the facilities exposed by the execution environment for tracking program execution, receiving notifications about important execution events including; thread creation, type definition loading and garbage collection. In managed execution environments built-in facilities also exist for augmenting program entities such as type definitions, method bodies and inter-module references whereas in unmanaged execution environments such facilities are not as well-defined.

3.6 Model

The key observation behind our approach for effecting runtime adaptations (via Kheiron) in existing/legacy systems is that all software systems run in a software execution environment. This fact makes the software execution environment an appealing target, as the lowest common denominator, for adapting live systems. Further, since we cannot assume anything about the types of applications being hosted, we are forced to develop general techniques for effecting adaptations via the execution environment.

[1] The JVM and CLR also differ considerably even though they are both managed execution environments.

Our techniques for adapting live systems are based on four key facilities exposed by contemporary execution environments:

1. Ability to trace program execution
2. Ability to control program execution
3. Access to metadata in the units of execution
4. Ability to add/edit metadata at load time or runtime

Table 1 summarizes the facilities exposed by three contemporary execution environments – the unmanaged/native execution environment comprising of the (Linux) operating system and the raw cpu (Intel x86), Sun Microsystems' Java Virtual Machine (JVM) v5.x and Microsoft's Common Language Runtime (CLR) v1.1 – that we use to effect and manage runtime adaptations in systems.

	Unmanaged Execution Environment	Managed Execution Environment	
	ELF Binaries	JVM 5.x	CLR 1.1
Program tracing	ptrace, /proc	JVMTI callbacks + API	ICorProfiler ICorProfilerCallback
Program control	Trampolines + Dyninst	Bytecode rewriting	MSIL rewriting
Execution Unit Metadata	.symtab, .debug sections	Classfile constant-pool + bytecode	Assembly, type & method metadata + MSIL
Metadata augmentation	N/A for compiled C-programs	Custom classfile parsing & editing APIs + JVMTI RedefineClasses	IMetaDataImport IMetaDataEmit APIs

Table 1: *Execution Environment Facilities*

Kheiron/CLR and **Kheiron/JVM** perform runtime adaptations in the Common Language Runtime and Java Virtual machine managed execution environments respectively. Conceptually, their approach to facilitating adaptations is the same. Kheiron/CLR and Kheiron/JVM perform operations on type definitions, object instances and various stages of the execution cycle to make them capable of interacting with an adaptation engine. To enable an adaptation engine to interact with a class instance, these Kheiron prototypes augment type definitions to add the necessary "hooks". Augmenting the type definition is a two-step operation.

Step 1 occurs when managed executables (assemblies in .NET and classfiles in JAVA) are loaded. At load-time the execution environment has obtained the execution-unit data but has not yet constructed the in-memory representation of the class. Kheiron/CLR and Kheiron/JVM add what we call *shadow methods* for each of the original public and/or private methods. A shadow method shares most of the properties – including a subset of the attributes, signature, implementation flags and the method descriptor – of the original method. However, a shadow method gets a unique name. Figure 2, transition A to B, shows an example of adding a shadow method **_SampleMethod** for the original method **SampleMethod**.

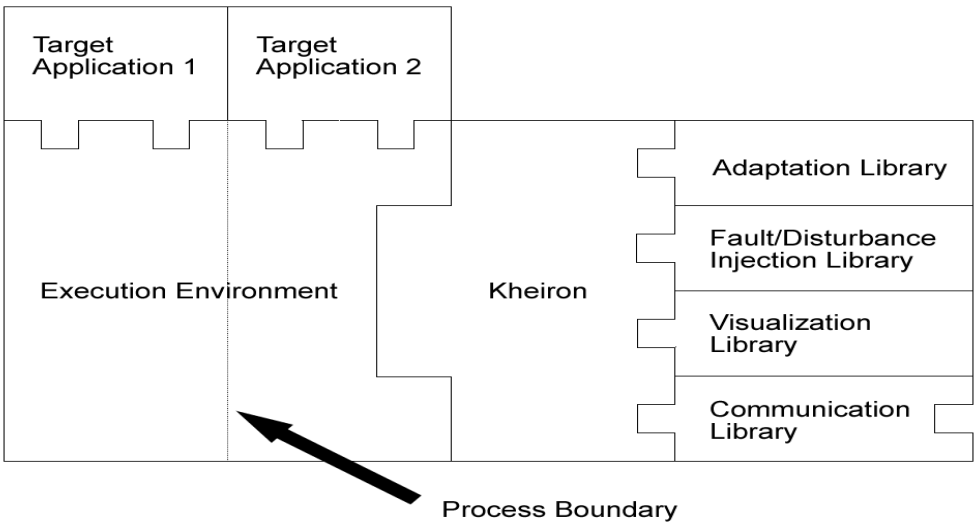

Figure 1: *Kheiron Conceptual Architecture*

Extending the metadata of a type by adding new methods must be done before the type definition is installed in the CLR or JVM. Once a type definition is installed, the execution environment will reject the addition or removal of methods or fields. Similarly, changing method signatures, method modifiers or inheritance relationships is also not allowed.

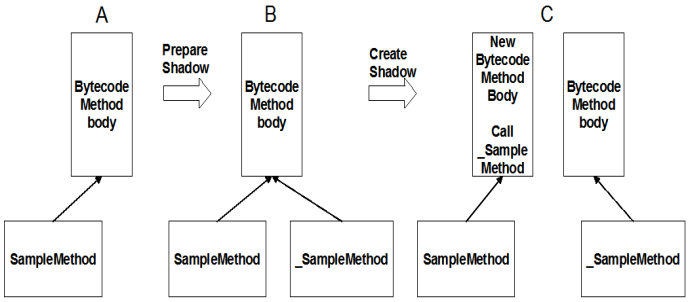

Figure 2: *Preparing and Creating a Shadow Method*

Step 2 of type augmentation occurs after the shadow method has been added. Kheiron/CLR and Kheiron/JVM use bytecode-rewriting techniques to convert the implementation of the original method into a thin *wrapper* that calls the shadow method, as shown in Figure 2, transition B to C.

Wrappers and shadow methods facilitate the adaptation of class instances. In particular, the regular structure and single return statement of the wrapper method, see Figure 3, enables Kheiron/CLR and Kheiron/JVM to easily inject adaptation instructions into the wrapper as prologues and/or epilogues to shadow method calls.

To add a prologue to a method new bytecode instructions must prefix the existing bytecode instructions. The level of difficulty is the same whether we perform the insertion in the wrapper or the original method. Adding epilogues, however, presents more challenges. Intuitively, we want to insert

```
SampleMethod( args ) [throws NullPointerException]
   <room for prolog>
   push args
   call _SampleMethod( args ) [throws NullPointerException]
   { try{…} catch (IOException ioe){…} } // Body of _SampleMethod
   <room for epilog>
   return value/void
```

Figure 3: *Conceptual Diagram of a Wrapper*

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. However, for methods with multiple return statements or exception handling routines, finding every possible return point can be an arduous task [61]. Using wrappers thus delivers a cleaner approach since we can ignore all of the complexity in the original method.

To initiate an adaptation, Kheiron/CLR and Kheiron/JVM augment wrappers to insert jumps into an adaptation engine at the *control point(s)* before and/or after a shadow method call. This allows an adaptation engine or suitable self-healing mechanism to be able to take control before and/or after a method executes.

Kheiron/C relies on the Dyninst API [7] (v4.2.1) to interact with target applications while they execute. Dyninst presents an API for inserting new code into a running program. The program being modified is able to continue execution and does not need to be recompiled or relinked. Uses for Dyninst include, but are not limited to, runtime code-patching and performance steering in large/long-running applications.

Dyninst employs a number of abstractions to shield clients from the details of the runtime assembly language insertion that takes place behind the scenes. The main abstractions are *points* and *snippets*. A point is a location in a program where instrumentation can be inserted, whereas a snippet is a representation of the executable code to be inserted. Examples of snippets include **BPatch_funcCallExpr**, which represents a function call, and **BPatch_variableExpr**, which represents a variable or area of memory in a thread's address space.

To use the Dyninst terminology, Kheiron/C is implemented as a *mutator* (Figure 4), which uses the Dyninst API to attach to, and modify a running program. On the Linux platform, where we conducted our experiments, Dyninst relies on ptrace and the /proc filesystem facilities of the operating system to interact with running programs.

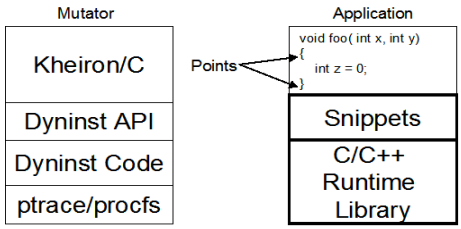

Figure 4: *Kheiron/C*

Kheiron/C uses the Dyninst API to search for global or local variables/data structures (in the scope of the insertion point) in the target program's address space, read and write values to existing variables, create new variables, load new shared libraries into the address space of the target program, and inject

function calls to routines in loaded shared libraries as prologues/epilogues (at the points shown in Figure 4) for existing function calls in the target application. As an example, Kheiron/C could search for globally visible data structures e.g. the head of a linked list of abstract data types, and insert periodic checks of the list's consistency by injecting new function calls passing the linked-list head variable as a parameter.

To initiate an adaptation Kheiron/C attaches to a running application (or spawns a new application given the command line to use). The process of attaching causes the thread of the target application to be suspended. It then uses the Dyninst API to find the existing functions to instrument (each function abstraction has an associated call-before instrumentation point and a call-after instrumentation point). The target application needs to be built with symbol information for locating functions and variables to work – with stripped binaries Dyninst reports ∼95% accuracy locating functions and an ∼87% success rate instrumenting functions. The disparity between the percentage of functions located and the percentage of functions instrumented is attributed to difficulties in instrumenting code rather than failures in the analysis of stripped binaries [31]. Kheiron/C uses the Dyninst API to locate any "interesting" global structures or local variables in the scope of the intended instrumentation points. It then loads any external library/libraries that contain the desired adaptation logic and uses the Dyninst API to find the functions in the adaptation libraries, for which calls will be injected into the target application. Next, Kheiron/C constructs function call expressions (including passing any variables) and inserts them at the instrumentation points. Finally, Kheiron/C allows the target application to continue its execution.

3.7 Feasibility

The techniques used by Kheiron to effect system-adaptations in running programs are elaborated further in [26] and [27]. Here we summarize the experiments and results evaluating the feasibility of using Kheiron for system adaptation.

Our first set of experiments were designed to evaluate the feasibility of using Kheiron/CLR, Kheiron/JVM and Kheiron/C in live systems. Kheiron/CLR and Kheiron/JVM impose a modest impact (∼5% and ∼2% respectively) on the performance of a target system when no adaptations, repairs or reconfigurations are active.

The Kheiron/CLR and Kheiron/JVM 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 and the Java HotspotVM v1.5 update 4. In our evaluation we used the C# and Java versions of the SciMark [2] [3] and Linpack [4] [5] computation-intensive benchmarks.

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).

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

[3] http://math.nist.gov/scimark2/

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

[5] http://www.shudo.net/jit/perf/Linpack.java

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.

Kheiron/CLR uses the CLR Profiler API [57] to intercept module load, unload and module attached to assembly events, Just-In-Time (JIT) compilation events and function entry and exit events. As expected, running an application in the profiler imposes some overhead on the application. Figure 5 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 Kheiron/CLR contributes ∼5% runtime overhead when no repairs are active, which we consider negligible.

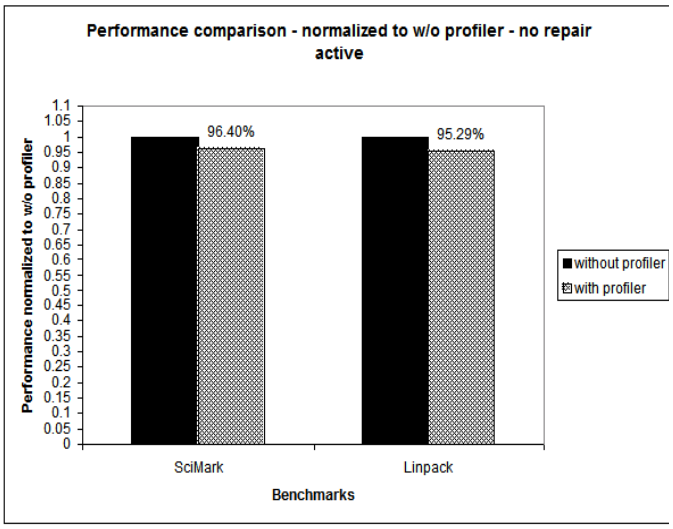

Figure 5: *Kheiron/CLR overheads when no repair active*

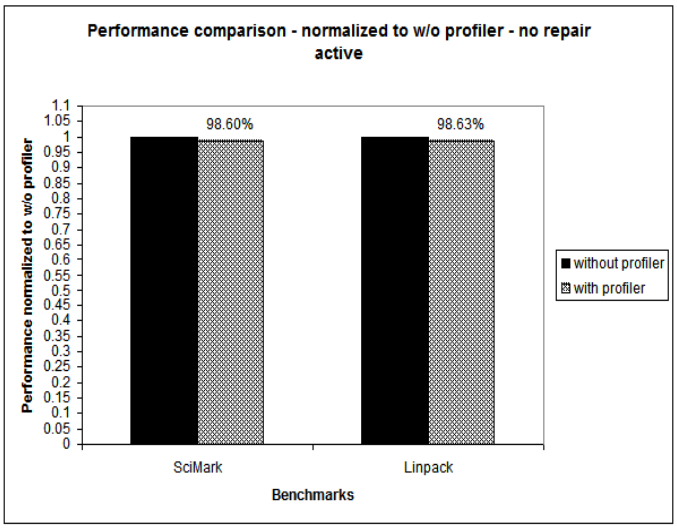

Figure 6: *Kheiron/JVM overheads when no repair active*

Kheiron/JVM uses the JVMTI interface to interact with running Java applications. Running an application under the JVMTI profiler imposes some overhead on the application. Also, the use of shadow methods and wrappers converts one method call into two. Figure 6 shows the runtime overhead for running the benchmarks with and without profiling enabled. We performed five test runs for SciMark and Linpack each with and without profiling enabled. Our Kheiron/JVM DLL profiler implementation was compiled as an optimized release build. 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 Kheiron/JVM contributes ∼2% runtime overhead when no adaptations are active, which we consider negligible. Note that we do not ask the Java HotspotVM to notify us on method entry/exit events since this can result in a slow down in some cases in excess of 5X. If adaptations were actually being performed then we expect the overheads measured to depend on the specifics of the adaptations.

By implementing Kheiron/CLR and Kheiron/JVM we are able to show that our conceptual approach of leveraging facilities exposed by the execution environment, specifically profiling and execution control services, and combining these facilities with metadata APIs that respect the verification rules for types, their metadata and their method implementations (bytecode) is a feasible, sufficiently low-overhead approach for adapting running programs in contemporary managed execution environments.

Effecting adaptations in unmanaged applications is markedly different from effecting adaptations in their managed counterparts, since they lack many of the characteristics and facilities that make runtime adaptation qualitatively easier, in comparison, in managed execution environments. Unmanaged execution environments store/have access to limited metadata, no built-in facilities for execution tracing, and less structured rules on well-formed programs.

Next we focus on using Kheiron/C to facilitate adaptations in running compiled C programs, built using standard compiler toolkits like *gcc* and *g++*, packaged as Executable and Linking Format (ELF) [83] object files, on the Linux platform.

We carry out a simple experiment to measure the performance impact of **Kheiron/C** on a target system. Using the C version of the SciMark v2.0 benchmark we compare the time taken to execute the un-instrumented program, to the time taken to execute the instrumented program – we instrumented the SOR_execute and SOR_num_flops functions such that a call to a function (AdaptMe) in a custom shared library is inserted. The AdaptMe function is passed an integer indicating the instrumented function that was called. Our experiment was run on a single Pentium 4 Processor, 2.4 GHz with 1 GB RAM. The platform was SUSE Linux 9.2 running a 2.6.8-24.18 kernel and using Dyninst v4.2.1. All source files used in the experiment (including the Dyninst v4.2.1 source tree) were compiled using gcc v3.3.4 and glibc v2.3.3.

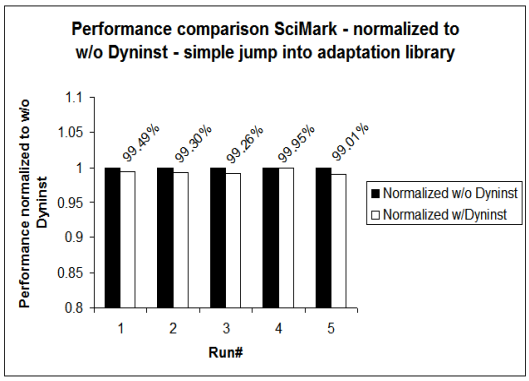

Figure 7: *Kheiron/C Simple Instrumentation Overheads*

As shown in Figure 7 the overhead of the inserted function call is negligible, $\sim$1%. This is expected since the x86 assembly generated behind the scenes effects a simple jump into the adaptation library followed by a return before executing the bodies of SOR_execute and SOR_num_flops. We expect that the overhead on overall program execution would depend largely on the operations performed while inside the adaptation library. Further, the time the SciMark process spends suspended while Kheiron/C performs the instrumentation is sub-second, $\sim$684 msecs $\pm$ 7.0686.

Our second set of experiments using Kheiron are concerned with practical uses of Kheiron. In [28] we explore consistency-preserving runtime adaptations in a non-trivial target system. Our case study uses

Kheiron to perform a dynamic reconfiguration in the Alchemi Enterprise Grid Computing System [3] while ensuring that the integrity of Alchemi and the execution environment are not compromised. We demonstrate the hot-swapping of the grid's job scheduler. Our experimental setup and results are described in **Appendix A**.

Finally, in [27] we describe using Kheiron/C to effect an example of a sophisticated system adaptation (see **Appendix B** for details). To enable applications to detect low-level faults and recover at the function level or, to enable portions of an application to be run in a computational sandbox, we use Kheiron/C to allow portions of an executable to be run under the STEM x86 emulator. Kheiron/C dynamically loads the emulator into the target process' address space and configures it to emulate individual functions. STEM (Selective Transactional EMulation) is an instruction-level emulator – developed by Locasto et al. [75] – that can be selectively invoked for arbitrary segments of code.

3.8 Related Work

Our Kheiron prototypes are concerned with facilitating very fine-grained adaptations in existing/legacy systems, whereas systems such as KX [23] and Rainbow [72] are concerned with coarser-grained adaptations. However, the Kheiron prototypes could be used as low-level mechanisms orchestrated/directed by these larger frameworks.

JOIE [13] is a toolkit for performing load-time transformations on Java classfiles. Unlike Kheiron/JVM, JOIE uses a modified classloader to apply transformations to each class brought into the local environment [12]. Further, since the goal of JOIE is to facilitate load-time modifications, any applied transformations remain fixed throughout the execution-lifetime of the class whereas Kheiron/JVM can undo/modify some of its load-time transformations at runtime e.g. removing instrumentation and modifying instrumentation and method implementations via bytecode rewriting. Finally, Kheiron/JVM can also perform certain runtime modifications to metadata, e.g. adding new references to external classes such that their methods can be used in injected instrumentation.

FIST [41] is a framework for the instrumentation of Java programs. The main difference between FIST and Kheiron/JVM is that FIST works with a modified version of the Jikes Research Virtual Machine (RVM) [4] whereas Kheiron/JVM works with unmodified Sun JVMs. FIST modifies the Jikes RVM Just-in-Time compiler to insert a breakpoint into the prologue of method to generate an event when a method is entered to allow a response on the method entry event. Control transfer to instrumentation code can then occur when the compiled version of the method is executed. The Jikes RVM can be configured to always JIT-compile methods, however the unmodified Sun JVMs, v1.4x and v1.5x, do not support this configuration. As a result, Kheiron/JVM relies on bytecode rewriting to transfer control to instrumentation code as a response to method entry and/or method exit – transfer of control will occur with both the interpreted and compiled versions of methods.

A popular approach to performing fine-grained adaptations in managed applications is to use Aspect Oriented Programming (AOP). AOP is an approach to designing software that allows developers to modularize cross-cutting concerns [24] that manifest themselves as non-functional system requirements. In the context of self-managing 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 logic that meets the system's functional requirements. An AOP engine is still necessary to realize the final system. Unlike Kheiron, which can facilitate adaptations in existing systems at the execution

environment-level, the AOP approach is a design-time approach, mainly relevant for new systems.

AOP engines *weave* together the code that meets the functional requirements of the system with the aspects that encapsulate the non-functional system requirements. There are three kinds of AOP engines: those that perform weaving at compile time (static weaving) e.g. AspectJ [22], Aspect C# [32], those that perform weaving after compile time but before load time, e.g. Weave .NET [17], which pre-processes managed executables, operating directly on bytecode and metadata and those that perform weaving at runtime (dynamic weaving) using facilities of the execution environment, e.g. A dynamic AOP-Engine for .NET [21] and CLAW [42]. Kheiron/JVM is similar to the dynamic weaving AOP engines only in its use of the facilities of execution environment to effect adaptations in managed applications while they run.

Adaptation concepts such as Micro-Reboots [9] and adaptive systems such as the K42 operating system [8] require upfront design-time effort to build in adaptation mechanisms. Our Kheiron implementations do not require special designed-in hooks, but they can take advantage of them if they exist. In the absence of designed-in hooks, our Kheiron implementations could refresh components/data structures or restart components and sub-systems, provided that the structure/architecture of the system is amenable to it, i.e., reasonably well-defined APIs exist.

Georgia Tech's 'service morphing' [64] involves compiler-based techniques and operating system kernel modifications for generating and deploying special code modules, both to perform adaptation and to be selected amongst during dynamic reconfigurations. A service that supports service morphing is actually comprised of multiple code modules, potentially spread across multiple machines. The assumption here is that the information flows and the services applied to them are well specified and known at runtime. Changes/adaptations take advantage of meta-information about typed information flows, information items, services and code modules. In contrast, Kheiron operates entirely at runtime rather than compile time. Further, Kheiron does not require a modified execution environment, it uses existing facilities and characteristics of the execution environment whereas service morphing makes changes to a component of the unmanaged execution environment – the operating system.

Trap/J [68], Trap.NET [67] produce adapt-ready programs (statically) via a two-step process. An existing program (compiled bytecode) is augmented with generic interceptors called "hooks" in its execution path, wrapper classes and meta-level classes. These are then used by a weaver to produce an adapt-ready set of bytecode modules. Kheiron/JVM, operates entirely at runtime and could use function call replacement (or delegation) to forward invocations to specially produced adapt-ready implementations via runtime bytecode re-writing.

For performing fine-grained adaptations on unmanaged applications, a number of toolkits are available, however many of them, including EEL [44] and ATOM [78], operate post-link time but before the application begins to run. As a result, they cannot interact with systems in execution and the changes they make cannot be modified without rebuilding/re-processing the object file on disk. Using Dyninst as the foundation under Kheiron/C we are able to interact with running programs – provided they have been built to include symbol information.

Our Kheiron implementations specifically focus on facilitating fine-grained adaptations in applications rather than in the operating system itself. KernInst [81] enables a user to dynamically instrument an already-running unmodified Solaris kernel in a fine-grained manner. KernInst can be seen as implementing some autonomic functionality, i.e., kernel performance measurement and consequent

runtime optimization, while applications continue to run. DTrace [10] dynamically inserts instrumentation code into a running Solaris kernel by implementing a simple virtual machine in kernel space that interprets bytecode generated by a compiler for the 'D' language, a variant of C specifically for writing instrumentation code. TOSKANA [20] takes an aspect-oriented approach to deploying before, after and around advice for in-kernel functions into the NetBSD kernel. They describe some examples of self-configuration (removal of physical devices while in use), self-healing (adding new swap files when virtual memory is exhausted), self-optimization (switching free block count to occur when the free block bitmap is updated rather than read), and self-protection (dynamically adding access control semantics associated with new authentication devices).

3.9 Conclusions and Future Work

In this section we describe the retro-fitting of fine-grained adaptation mechanisms onto existing/legacy systems by leveraging the facilities and characteristics of unmodified execution environments. We describe two classes of execution environments – managed and unmanaged – and compare the performance overheads of adaptations and the techniques used to effect adaptations in both contexts. We demonstrate the feasibility of performing adaptations using Kheiron/CLR, Kherion/JVM and Kheiron/C. We present a summary of a case study using Kheiron/CLR to perform a runtime adaptation in a non-trivial target system – the Alchemi Enterprise Grid Computing system. Finally, we describe a sophisticated adaptation, injecting the selective emulation of functions into compiled C-applications. Given that few legacy systems are written in managed languages (e.g. Java, C# etc.) whereas a substantial number of systems are written in C/C++, our techniques and approaches for effecting the adaptation of native systems may prove useful for retro-fitting new functionality onto these systems.

For future work, we are interested in conducting more sophisticated case studies where we can explore the effects of (and system response to) injecting faults into managed and unmanaged applications, which have/have not been dynamically modified with appropriate remediation (detection, diagnosis and repair/recovery) mechanisms. This last set of experiments is part of an effort to further the development of a methodology for evaluating the efficacy of these added RAS-enhancing mechanisms and benchmarking the capabilities [1, 6] of the resulting system.

4 Performing RAS-Evaluations for Computing Systems

4.1 Introduction

In this section we use a case study of an operating system and application server enhanced with recovery/repair mechanisms to show how the combination of dynamic fault-injection experiments and RAS-modeling provides us with tools we can use to assess the impact of the system's mechanisms on the environmental constraints governing the system's operation.

We make four contributions. First, we describe a set of fault-injection tools and experiments designed to exercise the recovery mechanisms of the system under test and obtain measurements for the rates of failure and time to recover.

Second, we use this experimental data to build RAS models of the system illustrating how they can be used to reason about the impact of the system's recovery mechanisms on high-level constraints such

as its reliability, availability and serviceability.

Third, we show how RAS models can be constructed to account for systems that employ fuzzy detection, diagnosis or repair (i.e. systems where detection, diagnosis or repair may not be 100% accurate or effective) and degraded modes of operation.

Finally, we distill our experiences and experimental results into a general recipe (the 7U-evaluation method) for evaluating and comparing self-healing systems. Our 7U-evaluation methodology highlights the role of the environmental/contextual constraints in establishing a meaningful scoring and comparison criteria.

4.2 Motivation

During its execution, a system's operation is typically governed by a number of environmental/contextual constraints, including but not limited to; policies set by administrators, service level agreements (SLAs) specifying the high-level reliability, availability and serviceability (RAS) goals the system is expected to meet or exceed. In addition to these high-level goals, the system is also expected to make decisions about which mechanisms to employ as a proactive, reactive or preventative response to changes in its environment. These decisions also impact the system's reliability, availability and serviceability. As a result, benchmarking the efficacy of the system requires evaluating both the mechanisms the system possesses (the micro-view) and their impact on the high-level goals governing the system's operation (the macro-view).

4.3 Hypothesis

The main hypotheses investigated in this section are:

1. **Continuous Time Markov Chains (CTMCs) and Markov Reward Models provide a reasonable framework for analyzing system failures, remediations and their impact on system operation**.
2. **RAS-models, represented as CTMCs, are flexible analysis tools**. They can be used to analyze reactive, proactive and preventative RAS-enhancing mechanisms. Further, they can be used to study both existing mechanisms and and those yet-to-be-implemented, making CTMCs applicable in both the post-deployment and design stages of the system and/or its RAS-enhancing mechanisms.
3. **RAS-models and fault-injection experiments can be used to evaluate the details of individual and combined mechanisms on real systems and quantify their impact on the high-level goals governing the system's operation.**

4.4 Model

Our first step in developing our evaluation techniques is to define the role of metrics using a thought-experiment based on five extreme types of systems to discuss the expected outcomes [6]:

[6]Due to the dependencies between detection and diagnosis and between diagnosis and repair we only consider four (Types 1 – 4) of the eight combinations of the detection, diagnosis and repair variables.

- **Type 1**: A system with no detection capabilities. It blunders on regardless of the carnage it fails to detect.
- **Type 2**: A system with perfect detection capabilities, but no diagnosis or repair mechanisms.
- **Type 3**: A system with perfect detection and diagnosis capabilities but no repair mechanisms.
- **Type 4**: A system that detects everything, diagnoses and repairs them perfectly.
- **Type 5**: A system that can detect everything, but has all of its detectors turned off.

We expect the metrics to punish a Type 1 system. Assuming the system never fails (or takes longer than the other systems to fail), it exhibits better availability, however, its goodput (the number of correct results returned per time interval) should decrease over time. Metrics should reward a Type 2 system for its detection capabilities (accuracy), but it should be penalized for failing. Type 3 systems, should be rewarded for its detection and diagnosis capabilities, under the assumption that it is easier for a human to repair the system using the results/log of the automated diagnosis, while the Type 2 system has to be manually diagnosed and repaired. Type 4 systems represent the ideal, we expect metrics to reward them for their detection, diagnosis and repair capabilities (accuracy). Finally, Type 5 systems should be credited for having detection mechanisms but penalized for turning them off. The main differentiator between Type 5 and Type 1 systems is the former's potential fault-model coverage – under the assumption that activating the mechanisms on a Type 5 system is easier than retro-fitting or acquiring them for a Type 1 system. Distinguishing between Type 1 and Type 5 systems is part of the qualitative analysis of the systems.

Based on these expectations, a ranking of these systems based on the self-healing benchmark results should be: Type 1 $<$ Type 5 $<$ Type 2 $<$ Type 3 $<$ Type 4. This ranking guides the choice of metrics used to differentiate among the four types of systems. Example metrics are shown in Table 2, categorized into macro-measurements and micro-measurements.

macro-view	goodput	reliability, availability and serviceability	fault-model coverage (expected vs actual)
micro-view	accuracy of detection, diagnosis and repair	speed of detection, diagnosis and repair	

Table 2: *Example Metrics*

Under the simplifying assumptions of exponentially distributed failure rates, a RAS-model, represented as a time-homogeneous continuous-time Markov chain (CTMC), can be used to relate the micro-measurements concerned with the accuracy and speed of detection, diagnosis and repair mechanisms to their impact on the macro-measurements concerned with system reliability, availability and serviceability, goodput and fault-model coverage.

Figure 8 shows a basic, 2-node, 2-parameter, RAS-model. The implicit assumptions of this simple model are; perfect problem detection and perfect (reactive) problem remediation strategies, however, more sophisticated RAS-models can be used to evaluate the impact of imperfect detection, diagnosis

and repair mechanisms. Further, they can also be used to evaluate proactive or preventative remediation mechanisms, under the assumption that the failure distribution is hypoexponential – divided into sequential stages where the time spent in each phase is independent and exponentially distributed [39].

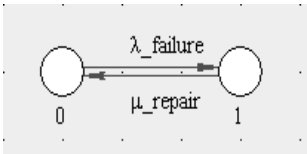

Figure 8: *Basic RAS Model (2-stage Markov Chain)*

Unlike hardware, where the laws of physics govern the failure rates of mechanical and electrical parts and materials, failure rates for software can be determined experimentally using fault-injection. Similarly, remediation times and accuracy of remediations can be determined experimentally and/or modeled mathematically to allow "what-if" scenarios to be considered.

Whereas fault-injection experiments may expose the system to rates of failure well above what the system may see in a given time period, these artificially high failure rates allow evaluators to explore the expected and unexpected system responses under stressful fault conditions, much like performance benchmarks subject the system under test to extreme workloads.

4.5 Feasibility

The goal of our experiments is to inject specific faults in the system under test and study the system's response. The faults we inject are intended to exercise the remediation mechanisms of the system. We use the experimental data to mathematically model the impact of the faults we inject on the system's reliability, availability and serviceability with and without the remediation mechanisms. Further, we consider the impact of imperfect repair on these macro-measurements. Our experiments and our models establish a link between the details of the remediation mechanisms and the high-level goals set for the system. This link is a key step in evaluating the efficacy of a system based on its remediation mechanisms. We first demonstrate how each fault and associated remediation mechanism(s) can be evaluated in isolation and then we evaluate the system taking into consideration all the faults and their associated remediation mechanisms.

To conduct our experiments we need: a test-platform, i.e. a hardware/software stack executing a reasonable workload, a fault model, fault-injection tools, a set of remediation mechanisms and a set of system configurations.

For our test platform we use VMWare GSX virtual machines configured with: 512 MB RAM, 1 GB of swap, an Intel x86 Core Solo processor and an 8 GB harddisk running Redhat 9 on 2.4.18 kernels. We use an instance of the TPC-W web-application (based on the implementation developed at the University of Madison-Wisconsin) running on MySQL 5.0.27, the Resin 3.0.22 application server and webserver, and Sun Microsystems' Hotspot Java Virtual Machine (JVM), v1.5. We simulate a load of 20 users using the Shopping Mix [52] as their web-interaction strategy. User-interactions are modeled using the Remote Browser Emulator (RBE) software also implemented at the University of Madison-Wisconsin. Our VMs are hosted on a machine configured with 2 GB RAM, 2 GB of swap, an Intel Core Solo T3100 Processor (1.66 GHz) and a 51 GB harddisk running Windows XP SP2.

Our fault model consists of device driver faults targeting the Operating System and memory leaks targeting the application server. We chose device driver faults because device drivers account for ∼70% of the Linux kernel code and have error rates seven times higher than the rest of the kernel [11]. While memory leaks and general state corruption (dangling pointers and damaged heaps) are highlighted as common bugs leading to system crashes in large-scale web deployments [9].

We identified the operating system and the application server as candidate targets for fault-injection. Given the operating system's role as resource manager [82] and part of the native execution environment for applications [27] its reliability is critical to the overall stability of the applications it hosts. Similarly, application servers act as containers for web-applications responsible for providing a number of services, including but not limited to, isolation, transaction management, instance management, resource management and synchronization. These responsibilities make application-servers another critical link in a web-application's reliability and another prime target for fault-injection.

We use a version of the SWIFI tools [53, 54] and a tool based on Kheiron/JVM [27] for device driver and application-server fault-injection respectively.

There are three remediation mechanisms we consider: system reboots, application server restarts and Nooks device driver protection and recovery [53] – Nooks isolates the kernel from device drivers using lightweight protection domains, as a result driver crashes are less likely to cause a kernel crash. Further, Nooks supports the transparent recovery of a failed device driver.

Finally, we use the following system-configurations: **Configuration A** – Fault-free system operation, **Configuration B** – System operation in the presence of memory leaks, **Configuration C** – System operation in the presence of device-driver failures (Nooks disabled), **Configuration D** – System operation in the presence of device-driver failures (Nooks enabled), and **Configuration E** – System operation in the presence of memory leaks and driver failures (Nooks enabled).

4.6 Results and Analysis

In our experiments we measure both client-side and server-side activity. On the client-side we use the number of web interactions and client-perceived rate of failure to determine client-side availability. On the server-side we use the number of resource (memory, process/thread creation, storage reads, writes, open and close) requests made and granted to determine the server-side availability. We collect server-side statistics via a kernel module that hooks the system call table entries underlying the malloc, fork, clone, read, write, open and close operations.

Each TPC-W run takes ∼24 minutes to complete. Table 3 shows the client-side goodput and server-side resource requests for **Configuration A**, a typical fault-free TPC-W baseline run.

Figure 9 shows the client-side goodput over the first forty runs (∼16 hours of continuous execution) in the presence of an accumulating memory leak – **Configuration B**. The average number of client-side interactions over this series of experiments is 3874.225 ± 94.760. In this figure there are two runs approximately 8 hours apart, runs 20 and 39 (circled), where the number of client interactions is 2 or more standard deviations below the mean (3626 and 3530 interactions respectively). Client-activity logs indicate a number of successive failed HTTP requests over an interval of ∼1 minute. Resin's logs indicate that the server encountered a low-memory condition, forces a number of JVM garbage collections before restarting the application server.

client-side	server-side	success rate
number of interactions: 3973	memory requests : 1848	memory : 100%
	memory requests granted : 1848	
	fork requests : 0	execution : n/a
	forks performed : 0	
	read requests : 3,498,678	reads : 99.5563%
	reads preformed : 3,483,154	
	write requests : 22,369	writes : 100%
	writes performed : 22,369	
	open requests : 18,476	opens : 100%
	opens performed : 18,476	
	close requests : 18,560	closes : 100%
	closes performed : 18,560	

Table 3: *Metrics for Configuration A, Fault-Free Run*

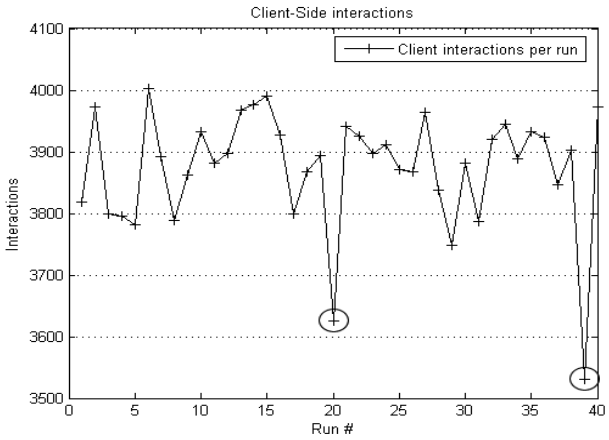

Figure 9: *Client interactions over first 40 runs (16 hrs) - Configuration B, Memory Leak Scenario*

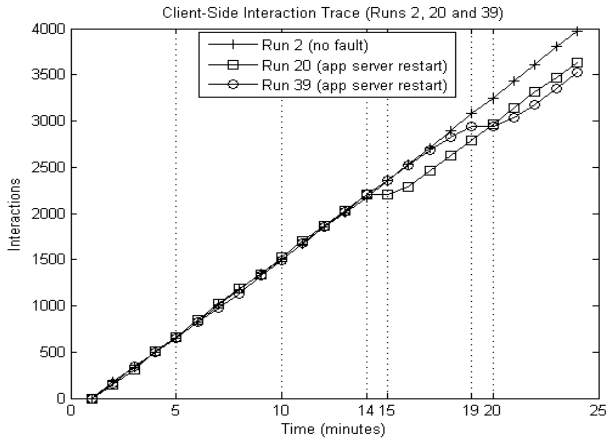

Figure 10: *Client-side Interaction Trace - Configuration B, Memory Leak Scenario*

Figure 10 shows a trace sampling the number of client interactions completed every 60 seconds for a typical run, (Run #2), compared to data from Runs 20 and 39 where low memory conditions cause Resin to restart. Restart times obtained from Resin's logs record startup times of: 3,092 msecs (initial startup), 47,034 msecs (restart 1) and 47,582 msecs (restart 2).

To evaluate the RAS-characteristics of the system in the presence of the memory leak, we use the SHARPE tool [69] to create the basic 2-node, 2-parameter RAS-model shown in Figure 11.

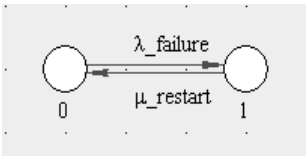

Figure 11: *Simple RAS Model*

State S_0 represents an UP state i.e. the system is servicing requests, and state S_1 represents the application server being restarted. Since no client requests are being serviced while the application server is being restarted (as shown in Figure 10), S_1 is a DOWN state. $\lambda_{failure}$ is the observed rate of

failures; 1 failure every 8 hours and $\mu_{restart}$ is the time to restart the application server, ~47 seconds. Whereas this model implicitly assumes that the detection of the low memory condition is perfect and the restart of the application server resolves the problem 100% of the time, in this instance our model assumptions are validated by the experiments.

Using the steady-state/limiting availability formula [39]: $A = \frac{\frac{1}{\lambda}}{\frac{1}{\lambda}+\frac{1}{\mu}}$ the steady state availability of the system is 99.838%. Further, the system has an expected downtime of 866 minutes per year – given by the formula $(1 - Availability) * T$ where $T = 525,600$ minutes in a year. At best, the system is capable of delivering two 9's of availability. Table 4 shows the expected penalties per year for each minute of downtime over the allowed limit. As an additional consideration, downtime may also incur costs in terms of time and money spent on service visits, parts and/or labor, which add to any assessed penalties.

Availability Guarantee	Max Downtime Per Year	Expected Penalties
99.999	~5 mins	(866 - 5)*$p
99.99	~53 mins	(866 - 53)*$p
99.9	~526 mins	(866 - 526)*$p
99	~5256 mins	$0

Table 4: *Expected SLA Penalties for Configuration B*

In **Configuration C** we inject faults into the pcnet32 device driver with Nooks driver protection disabled. Each injected fault leads to a kernel panic requiring a reboot to make the system operational again. Using the experimentally achieved fault rate, $\lambda_{failure}$, of 4 device driver faults every 8 hours and system reboot time, $\mu_{restart}$, of 1 minute 22 seconds and the fact that rebooting the system always resolves the device driver fault condition we can again use the simple RAS-model shown in Figure 11 to evaluate the RAS-characteristics of the system under test. Using SHARPE, we calculate the steady state availability of the system as 98.87%, with an expected downtime of 5924 minutes per year i.e. this system cannot deliver two nines of availability under these conditions.

Next we consider the case of the system under test enhanced with Nooks device driver protection enabled – **Configuration D**. Whereas we reuse the same fault-load and fault-rate, 4 device driver failures every 8 hours we need to revise the RAS-model used in our analysis to account for the possibility of imperfect repair i.e. to handle cases where Nooks is unable to recover the failed device driver and restore the system to an operational state. To achieve this we use the RAS-model shown in Figure 12.

Figure 12 is a 3-node, 4-parameter model. State S_0 is an UP state where the system services client requests. In state S_1, Nooks is recovering a failed device driver. Since it is possible for the system to continue to service requests during recovery, S_1 is also considered an UP state. State S_2 represents a failure that requires a hard reboot to restore the system to an operational state. Since the system is unable to service requests while it reboots, S_2 is a DOWN state.

In the model $\lambda_{driver\_failure}$ is the rate at which device driver failures occur, 4 failures every 8 hours. $\mu_{nooks\_recovery}$ is the time for Nooks to successfully recover a failed device driver, experimentally observed as 4093 microseconds in the worst case for our configuration. c is the *coverage factor*, it represents the probability that the driver fault can be successfully recovered by Nooks. The coverage factor can also be interpreted as the success rate of Nooks recovery – varying this parameter in our RAS model allows us to investigate the impact of imperfect recovery on the system's RAS-characteristics.

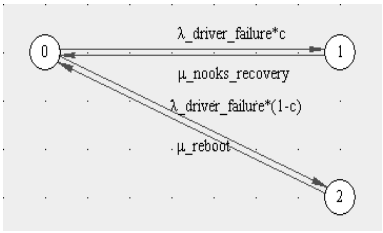

Figure 12: *RAS-Model of a system with imperfect repair*

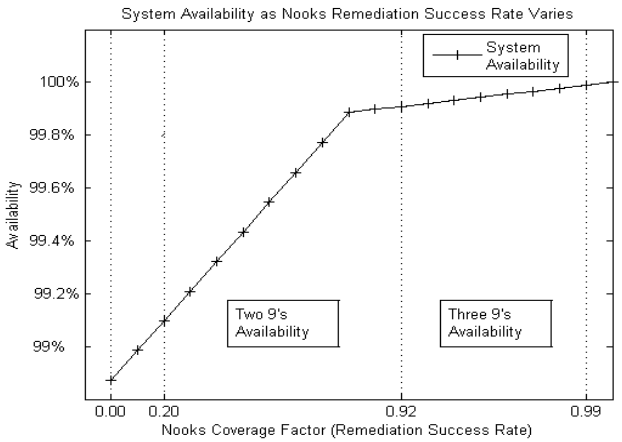

Figure 13: *Availability Versus Nooks Coverage – Configuration D*

Finally, μ_{reboot} is the time to reboot the machine, 1 minute 22 seconds. Figure 13 shows the expected impact Nooks recovery on the system's RAS-characteristics as its success rate varies.

Whereas Configuration C of the system under test is unable to deliver two 9's of availability in the presence of device driver faults, we can see from Figure 13 that a modest 20% success rate of Nooks in Configuration D is expected to promote the system into another availability bracket (reducing the expected downtime and SLA penalties by an order of magnitude) while a 92% success rate reduces the expected downtime and SLA penalties by two orders of magnitude [7].

Thus far we have analyzed the system under test and each fault in isolation i.e. each RAS-model we have developed so far considers one fault and its remediations. We now develop a RAS-model that considers all the faults in our fault-model and the remediations available, **Configuration E** – see Figure 14.

Figure 15 shows the expected availability of the complete system. The system's availability is limited to two 9's of availability even though the system could deliver better availability and downtime numbers – the minimum system downtime is calculated as 866 minutes per year, the same as for Configuration B, the memory leak scenario. Thus, even with perfect Nooks recovery, the system's availability is limited/determined by the reactive remediation for the memory leak. To improve the system's overall availability we need to improve the handling of the memory leak.

One option for improvement is to explore the possible benefits of preventative maintenance. For preventative maintenance to be an option we have to assume that the system's failure distribution is hypoexponential. We divide the system's lifetime into into two stages, where the time spent in each state is exponentially distributed.

We use the RAS-model shown in Figure 16 in our analysis. This model consists of six states; state S_0 is an UP state representing the first stage of the system's lifetime, state S_1 is an UP state representing the second stage of the system's lifetime. Stage S_2 is a DOWN state where the application server is

[7] In our experiments we were unable to encounter a scenario where Nooks was unable to successfully recover a failed device driver however the point of our exercise is to demonstrate how that eventually could be accounted for in an evaluation of a remediation mechanism.

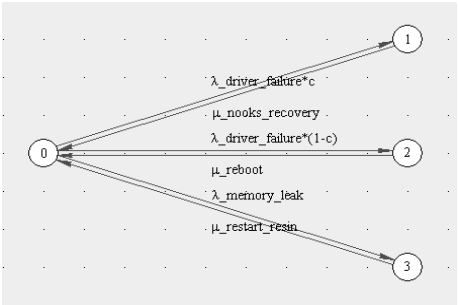

Figure 14: *Complete RAS-model – Configuration E*

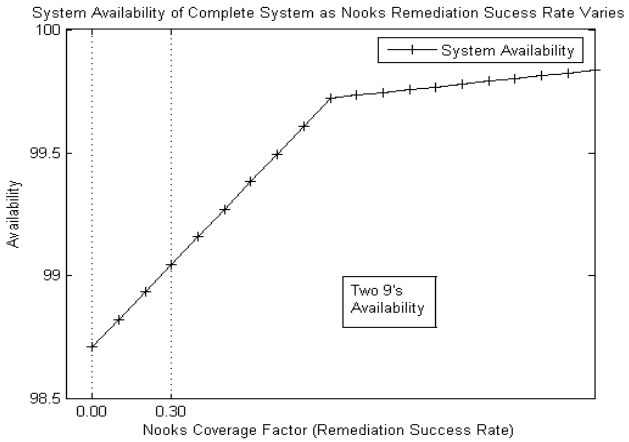

Figure 15: *Availability Versus Nooks Coverage For the Full System – Configuration E*

being restarted, state S_3 is an UP state representing an inspection of the memory conditions, since this check occurs from state S_0, the first stage of the system's lifetime, no preventative maintenance actions are carried out. State S_4 is an UP state representing an inspection of the memory conditions, since this check occurs from state S_1, the second stage of the system's lifetime, a preventative maintenance action is carried out. State S_5 represents the preventative maintenance action, which restores the system into the first stage of its lifetime.

There are six parameters in our preventative maintenance model. $\lambda_{2ndstage}$ is the rate of transition from the first stage of the system's lifetime into the second stage of the systems lifetime, from our observation this rate of transition is once every six hours. $\lambda_{failure}$ is the rate of transition into a state indicative of a low-memory condition, after 6 continuous hours in S_0 the system is expected to fail once in the next two hours. $\mu_{restart\_resin\_worst}$ is the time to restart Resin under low-memory conditions, $\sim$47 seconds. $\lambda_{inspect}$ is the rate at which we check for a decreasing trend in the amount of free memory available, $\mu_{inspect}$ is the time necessary for the check, 21,627 microseconds. $\mu_{restart\_resin\_pm}$ is the time to restart Resin when the system is not under severe memory pressure, 3,092 milliseconds.

Using these parameters we plot the graph shown in Figure 17, which shows the expected availability of the system as $\lambda_{inspect}$ varies. We see that performing a check 6 times every hour and conditionally performing a preventative maintenance is expected to allow the system to deliver better availability, however, actually implementing a preventative maintenance scheme and running more experiments is the only way to determine if the scheme has the desired effect.

4.7 7U-Evaluation

Based on our experiments, RAS-model construction and analysis we distill our steps into a 7-step process for evaluating a self-healing system and its self-healing mechanisms, the 7U-evaluation methodology. Our evaluation method establishes a link between the individual self-healing mechanisms and their impact on the high-level goals governing the system's operation. Further, it highlights the role of environmental constraints/policies governing the system's operation as the main criteria for comparing self-healing systems as well as determining their efficacy. The seven steps are shown in Figure

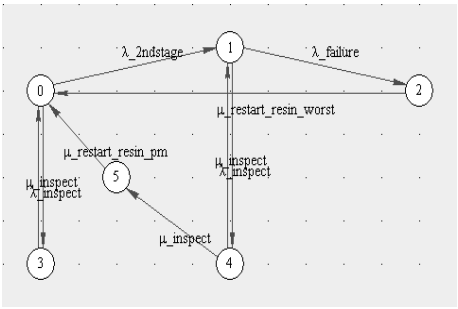

Figure 16: *Preventative Maintenance RAS-model*

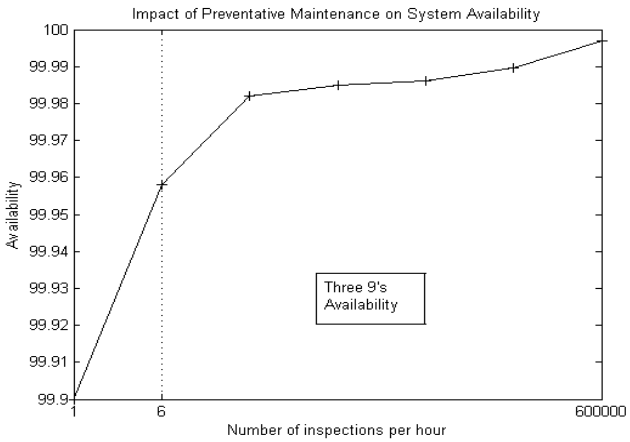

Figure 17: *Expected impact of preventative maintenance*

18.

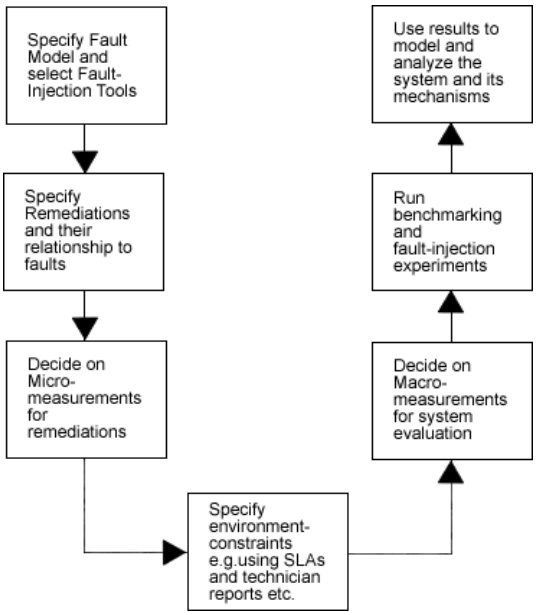

Figure 18: *7U-evaluation methodology*

The 7U-evaluation method derives its utility from the relationship between faults and the environment where the system will be deployed. The fault-environment relationship quantifies the effect of errors in a given context. It is influenced by data from policies, servicing/technician records, service-level agreements (SLAs), trouble tickets, system logs, customer-service call-center operation-logs, etc. These data sources can be used to establish a fault-model/hypothesis [40] of interest for conducting experiments, design fault-injection experiments to investigate existing or potential remediation mechanisms, guide the selection of micro-measurements (measurements concerned with the details of a remediation mechanism) and macro-measurements (measurements codifying system goals). This relationship can also be used to reason about symptoms, occurrence characteristics, error-severity, time and money spent on parts and labor for servicing. SLAs estimate the potential business impact of a compromised macro-measurement e.g. availability. Most of these measures have meaning only

within a given organization. As a result, the determination of whether one system is better than another must be decided in relation to the environmental/contextual constraints and policies that govern the system's operation.

Considering the environment also influences the scoring of each system and its mechanisms. For example, deciding whether to assess a system a bonus or penalty for using redundancy depends on the environmental constraints. In some contexts, having a system do more with less (i.e. no redundant parts) may be considered a good thing, especially when compared to the monetary and intangible costs of redundancy, however in other contexts a highly redundant system may be assessed a bonus. We posit that unlike performance benchmarks, which are universally comparable provided that the hardware configuration, software configuration and workload are kept constant, self-healing benchmark results are meaningful and comparable if the workload, fault-load and environmental constraints are kept constant. We expect there to be some flexibility in the hardware and software configurations insofar as class substitutions may be made e.g. substituting one relational database system for another. However, hardware configuration parameters, e.g. the amount of installed memory remain fixed.

4.8 Related Work

The work most similar to ours is [19]. In this paper the authors build a RAS model to explore the expected impact of Memory Page Retirement (MPR) on hardware faults associated with failing memory modules on systems running Solaris 10. MPR removes a physical page of memory from use by the system in response to error correction code (ECC) errors associated with that page. Using their models the authors investigate the expected impact of MPR on yearly downtime, the number of service interruptions and the number of servicing visits due to hardware permanent faults. Unlike our experiments, which focus on software and rely on fault injection experiments to collect data, the authors focus on hardware failures and use field data from deployed low-end and mid-range server systems to build models.

In [18] the authors study the availability of the Sun Java System Application Server, Enterprise Edition 7. The authors use hierarchical Markov reward models to model and obtain average system availability estimates. In a distributed load-balanced deployment, including two application server instances, 2 pairs of Highly Available Databases (HADBs) – used as http session state stores –, an Oracle database and a Sun Java System Directory Server, the authors induce faults concerned with whole-node removal to investigate the system's (session) fail-over and recovery mechanisms. Our experiments differ in the granularity of our fault-injection, rather than remove entire nodes, we focus on injecting faults in the individual components of a single node. Further, whereas we do not focus on evaluating remediation mechanisms that rely on whole-node redundancy or failover, RAS-modeling techniques can be adapted for this [39].

[34] describes the DBench-OLTP dependability benchmark. We differ from this work in our choice of metrics. The measures prescribed in the DBench-OLTP specification are analogous to our macro-measurements, some of which include but are not limited to: transactions per minute (tpmC), price per transaction ($/tpmC), availability from the system under test and remote terminal emulator points of view. We focus less on performance-related measures and present ways to analyze the impact of the system's remediation mechanisms on the macro-measurements of interest.

FAUMachine [76] (formerly UMLinux) is a virtualization platform supporting fault-injection. The

faults that can be injected include, but are not limited to; bit flips in memory and CPU registers, block device failures and network failures. For our experiments we required more fine-grained control over the faults injected. Further, the faults that could be injected using FAUMachine would not appropriately exercise the remediation mechanisms of our target system.

Our work is complementary to the work done on robustness benchmarking [16] and fault-tolerant benchmarking [85]. However, we focus less on the robustness of individual component interfaces for our fault-injection and more on system recovery in the presence of component-level faults i.e. resource leaks, delays or hangs in components and component-removals.

[5] conducts a study of availability and maintainability benchmarks using software RAID systems. In addition to studying availability from the end-user perspective as these authors do, we also include the use of mathematical models to assist in the analysis of existing and potential remediation-mechanisms.

[51] describes the System Recovery Benchmark. The authors propose measuring system recovery on a non-clustered standalone system. The focus of the work is on detailed measurements of system startup, restart and recovery events. Our work is complementary to this, relying on measuring startup, restart and recovery times at varying granularity. We consider these measurements at node-granularity as well as application/component granularity. Further, we relate these micro-measurements to the impact on the high-level objectives guiding the system's recovery decisions.

[6] describes work towards a self-healing benchmark. In our work we analyze the the individual mechanisms that impact the quality of service metrics of interest. Our focus on how the system accomplishes healing and its relation to the high-level system goals, dictated by SLAs and policies, ties together the micro and macro views of the system in the evaluation of the target system.

4.9 Conclusions and Future Work

In this section we use reliability, availability and serviceability (RAS) metrics and models, coupled with fault-injection experiments, to analyze the impact of self-healing mechanisms on these high-level (RAS) metrics of interest. We also demonstrate how we can account for imperfect remediations and preventative maintenance in our analysis. We identify RAS-metrics as reasonable gauges of the efficacy of a self-healing system since the intended goal of self-healing systems is to improve system reliability and availability and reduce the management burden on human administrators. We distill our experiences into a 7-step process for evaluating self-healing systems, the 7U-evaluation method. This methodology highlights the role of environmental constraints/policies governing the system's operation as the main criteria for comparing self-healing systems as well as determining their efficacy.

For future work, we are interested in conducting our experiments on other operating system platforms, most notably Solaris 10, which has been designed with a number of self-healing mechanisms [79, 73]. We will also continue our work developing practical fault-injection tools.

5 Research Plan and Schedule

The fault-injection tools and testbed described in section 4.5 have been completed. However, there are still enhancements to the fault-injection tools and additional fault-injection experiments to be

conducted. My objective is to conduct a study of the impact of faults and any existing or yet-to-be-added RAS-enhancing mechanisms on the various components of N-tier web applications.

The study is guided by the following goals.

The first goal is to move the testbed (TPC-W + Resin + MySQL) over to the Linux 2.6, Solaris 10 and Windows XP SP2 operating systems so that I can re-run our reliability, availability and serviceability experiments. My objective is to investigate the impact of faulty device drivers on these operating system kernels. Device drivers account for 70% of Linux kernel code, with error rates seven times higher than the rest of the kernel [11], whereas in Windows XP device driver faults account for 85% of recent failures [53]. Failure information for Solaris is not available but Solaris 10 is purported to include an IO Fault Services infrastructure to isolate the OS from some hardware and software defects, our intent is to evaluate the impact of this infrastructure.

To facilitate device driver fault-injection on Linux 2.6, I am working on porting the SWIFI tools over to Linux 2.6. To facilitate device driver fault-injection on Solaris 10, I have to develop a tool for this, based on my experience with the SWIFI tools and Unix-kernel development.

To facilitate device driver fault-injection on Windows XP SP2, I will be using Microsoft's Installable File Systems (IFS) kit [58] and my past experience developing Windows Device Drivers [15]. The IFS kit is Microsoft's kernel driver development toolkit. Details about the interactions between device drivers and core Windows kernel components can be found in [49].

The second goal is to explore database fault-injection on MySQL 5.0.27 using Kheiron/C's ability to dynamically adapt compiled C-programs and the MySQL Internals Manual [2], which describes the internal structure and operation of the database engine core. The objective is to focus on injecting faults in the database engine core that potentially affect the operation of the engine rather than injecting faults affecting the data being managed – the DBench OLTP benchmark [34] is concerned with faults that drop tables or delete schemas. My preliminary tests show that the latest version of Dyninst (v5.0) is able to interact with MySQL 5.0.27 at runtime.

The third goal is to extend the existing fault-model with more software faults. Via Kheiron/JVM we have the ability to inject delays, hangs and remove components, I now want to experiment injecting these faults into Resin and one other application server for comparison.

The fourth goal is to use our ability to inject fine-grained faults at runtime to study the behavior of the system in the presence of near-coincident faults. Near-coincident faults occur when a second fault occurs while the system is processing the previous fault [39]. Further, I will construct RAS-models for the systems under test to facilitate their analysis and comparison.

Table 5 shows my plan for completion of the research.

6 Expected Contributions

The contributions of this thesis are anticipated to include:

1. **Contributions towards a representative fault-model for computing systems that can be induced/injected using fault-injection tools**. One major requirement of fault-injection tools is that they be able to induce representative faults so we can reproduce the failure behavior of the system and accurately evaluate any mechanisms proposed to mitigate these failures.

Timeline	Work	Progress
	Develop initial Kheiron (CLR,JVM,C) prototypes	completed
Jan. 2006	Submitted Kheiron paper to ICAC 2006	accepted
Sep. 2006	Build GUI front-end for Kheiron/JVM	ongoing
Oct. 2006	Build self-healing benchmark simulator	completed
Nov. 2006	Build Linux-based testbed for RAS-benchmark experiments	completed
Dec. 2006	Run preliminary RAS-benchmarking experiments	completed
Jan. 2007	Submit paper on initial RAS-benchmark results to ICAC 2007	completed
Feb. 2007	Write Proposal Document	completed
Mar. 2007	Port Linux 2.4 device driver fault-injection tools (SWIFI tools) to Linux 2.6	ongoing
Mar. 2007	Write device driver fault-injection tools for Windows XP SP2 using IFS kit	ongoing
May. 2007	Write proof-of-concept database fault-injection tools using Kheiron/C	ongoing
Jun. 2007	Write (or acquire under NDA) fault-injection tools for Solaris 10	ongoing
Jul. 2007	Build machine for hardware and software fault-injection RAS-experiments	ongoing
Aug. 2007	Start next round of RAS-experiments using Linux 2.6, Solaris 10 and Windows XP SP2	ongoing
Jan. 2008	Thesis writing	
Aug. 2008	Thesis defense	

Table 5: Plan for completion of my research

2. **A suite of runtime fault-injection tools**. The suite of tools will support injecting a range of fine-grained faults into commodity operating systems, non-trivial compiled C-programs (e.g. RDBMS), Java and .NET applications (e.g. application servers). The tools we develop are complementary to existing software-based and hardware-based fault-injection tools.
3. **A survey** of RAS-enhancing mechanisms (or lack thereof) in contemporary operating systems and application servers.
4. **Analytical techniques** that can be used at design-time or post-deployment time.
5. **A RAS-benchmarking methodology, suite and report for computing systems**. A methodology for evaluating and analyzing the RAS-mechanisms of computing systems and their impact on the policies and SLAs the govern that system's operation. In the analysis I highlight the various facets of the mechanisms that can be modeled and analyzed. I also compare individual mechanisms and complete systems enhanced with these mechanisms.

In addition to the contributions listed above, at the time of this writing the following practical accomplishments have already been made:

- Published papers on dynamic system-adaptations using Kheiron; [26], [3] and [27].
- Submitted a paper on using RAS Models and Metrics to evaluate Self-Healing systems to the IEEE International Conference on Autonomic Computing (ICAC) 2007.

7 Future Work and Conclusion

This thesis has focused on developing a general approach to runtime fault-injection to enable the study the impact of faults and any remediation mechanisms. However, there are a number of interesting future work possibilities.

7.1 Immediate Future Work Possibilities

- Work towards a benchmark for self-healing systems. The 7U-evaluation approach emphasizes the link between the micro-view of the system (its mechanisms) and the macro-view of the

system (its goals). For self-healing systems, we expect the repair mechanisms of the system to impact the system's reliability, availability, serviceability and overall manageability. The rigorous analytical techniques and experimental studies of the impact of faults and remediations will help define a foundation for a self-healing benchmark.

- Further explore and compare fault-injection frameworks for software systems. As of this writing some operating systems are being developed with built-in fault-injection mechanisms. For example the Linux 2.6.19 (and later) kernels include facilities for injecting memory allocation faults and causing occasional disk I/O operations to fail [47]. Whereas, these facilities have been implemented to assist in exercising error paths during kernel and filesystem development, they are complementary to runtime fault-injection tools. Further, we expect their use to result in more robust remediation mechanisms.

7.2 Long Term Future Work Possibilities

- Explore the integration of RAS-modeling tools into integrated development environments (IDEs). Given the interest in system reliability, high-availability and manageability (serviceability, repair and recovery) and the ability to use RAS-models to explore the expected impact of RAS-enhancing mechanisms, incorporating them into the IDE e.g. via add-on toolkits and declarative language annotations [59, 80] may be a viable way to satisfy these extra-functional requirements early in the development cycle.

7.3 Conclusion

In this thesis I describe techniques for runtime fault-injection into commodity operating systems and applications as well as the analysis of computing systems based on their failure-behavior, (existing or lacking) remediation mechanisms and the policies, SLAs etc. that guide the system's operation. The end goal of this thesis is to improve the study, development and design of reliable, highly available and more manageable software systems through the use of rigorous analytical tools.

8 Appendix A – Alchemi Experiments

Our experimental testbed was an Alchemi cluster consisting of two Executors (Pentium-4 3GHz desktop machines each with 1GB RAM running Windows XP SP2 and the .NET Framework v1.1.4322), and a Manager (Pentium-III 1.2GHz laptop with 1GB RAM running Windows XP SP2 and the same .NET Framework version). 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.

One thing we measured was the time taken to swap the scheduler. We requested scheduler swaps between runs of the PiCalculator application. The time taken to replace the scheduler instance was about 500 ms, on average; however, that time was dominated by the time spent waiting for the scheduler thread to exit. In the worst case, a scheduler-swap request arrived while the scheduler thread was 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 6 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 had 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 6: *PiCalculator.exe Job Completion Times*

Thus we were able to demonstrate that Kheiron can be used to facilitate a consistency-preserving reconfiguration of the Alchemi Grid Manager without compromising the integrity of the CLR or the Alchemi Grid Manager, and by extension the Alchemi Grid and jobs actively executing in the grid. The combination of ensuring that the augmentations made by Kheiron to insert hooks for the adaptation engine respect the CLR's verification rules for type and method definitions (see [65] for details on how we guarantee this) and relying on human analysis to determine what transformations Kheiron should perform, and when they should be performed, can guarantee that the operation of the

target system is not compromised. Human analysis leverages the consistency-guarantees of Kheiron with respect to the CLR, allowing the designers of adaptations to focus on preserving the consistency of the target system (at the application level) based on knowledge of its operation.

9 Appendix B – Dynamic Selective Emulation or Compiled C-Applications

To enable applications to detect low-level faults and recover at the function level or, to enable portions of an application to be run in a computational sandbox, we describe an approach that allows portions of an executable to be run under the STEM x86 emulator. We use Kheiron/C to dynamically load the emulator into the target process' address space and emulate individual functions. STEM (Selective Transactional EMulation) is an instruction-level emulator – developed by Locasto et al. [75] – that can be selectively invoked for arbitrary segments of code. The emulator can be used to monitor applications for specific types of failure prior to executing an instruction, to undo any memory changes made by the function inside which the fault occurred (by having the emulator track memory modifications) and, simulate an error return from the function (error virtualization)[75].

```
void foo()
{
    int i = 0;
    // save cpu registers macro
    emulate_init();
    // begin emulation function call
    emulate_begin();
    i = i + 10;
    // end emulation function call
    emulate_end();
    // commit/restore cpu registers macro
    emulate_term();
}
```

Figure 19: *Inserting STEM via source code*

The original implementation of STEM works at the source-code level i.e. a programmer must insert the necessary STEM "statements" around the portions of the application's source code expected to run under the emulator (Figure 19). In addition, the STEM library is statically linked to the executable. To inject STEM into a running, compiled C application, we need to be able to: load STEM dynamically into a process' address-space, manage the CPU-to-STEM transition as well as the STEM-to-CPU transition.

To dynamically load STEM we change the way STEM is built. The original version of STEM is deployed as a GNU AR archive of the necessary object files; however, the final binary does not contain an ELF header – this header is required for executables and shared object (dynamically loadable) files. A cosmetic change to STEM's makefile suffices – using gcc with the -shared switch at the final link step. Once the STEM emulator is built as a true shared object, it can then be dynamically loaded into the address space of a target program using the Dyninst API.

Next, we focus on initializing STEM once it has been loaded into the target process' address space. The original version of STEM requires two things for correct initialization. First, the state of the machine before emulation begins must be saved – at the end of emulation STEM either commits its current state to the real CPU registers and applies the memory changes or STEM performs a rollback

of the state of the CPU, restoring the saved register state, and undoes the memory changes made during emulation. Second, STEM's instruction pipeline needs to be correctly setup, including the calculation of the address of the first instruction to be emulated.

To correctly initialize our dynamically-loadable version of STEM we need to be able to effect the same register saving and instruction pipeline initialization as in the source-scenario. In the original version of STEM register saving is effected via the emulate_init macro, shown in Figure 19. This macro expands into inline assembly, which moves the CPU (x86) registers (eax, ebx, ecx, edx, esi, edi, ebp, esp, eflags) and segment registers (cs, ds, es, fs, gs, ss) into STEM data structures.

Whereas Kheiron/C can use Dyninst to dynamically load the shared-object version of STEM into a target process' address-space and inject a call to the emulate_begin function, the same cannot be done for the emulate_init macro, which must precede a call to emulate_begin. Macros cannot be injected by Dyninst since they are intended to be expanded inline by the C/C++ preprocessor before compilation begins. This issue is resolved by modifying the trampoline – a small piece of code constructed on-the-fly on the stack – Dyninst sets up for inserting prologues, code (usually function calls) executed before a function is invoked.

Dyninst instrumentation via prologues works as follows: the first five bytes after the base address[8] of the function to be instrumented are replaced with a jump (`0xE9 [32-bit address]`) to the beginning of the trampoline. The assembly instructions in the trampoline save the CPU registers on the stack, execute the prologue instrumentation code, restore the CPU registers and branches to the instructions displaced by the jump instruction into the trampoline. Then another jump is made to the remainder of the function body before control is finally transferred to the instruction after the instrumented function call [7].

We modify this trampoline such that the contents of the CPU general purpose registers and segment registers are saved at a memory address (*register storage area*) accessible by the process being instrumented. This modification ensures that the saved register data can be passed into STEM and used in lieu of the emulate_init macro. In addition, we modify Dyninst such that the instructions affected by the insertion of the five-byte jump into the trampoline are saved at another memory address (*code storage area*) accessible by the process being instrumented. Since the x86 processor uses variable-length instructions, there is no direct correlation between number of instructions displaced and the number of bytes required to store them. However, Dyninst has an internal function **getRelocatedInstructionSz**, which it uses to perform such calculations. We use this internal function to determine the size of the code storage area where the affected instructions are copied.

The entire CPU-to-STEM transition using our dynamically-loadable version of STEM is as follows: Kheiron/C loads the STEM emulator shared library and a custom library (dynamically linked to the STEM shared library) that has functions (RegisterSave and EmulatorPrime). Next, Kheiron/C uses the Dyninst API to find the functions to be run under the emulator. Kheiron/C uses Dyninst functions which support its BPatch_thread::malloc API to allocate the areas of memory in the target process' address-space where register data and relocated instructions are saved. The addresses of these storage areas are set as fields added to the BPatch_point class – the concrete implementation of Dyninst's point abstraction. RegisterSave is passed the address of the storage area and copies data over from the storage area into STEM registers – so that a subsequent call to emulate_begin will work. Em-

[8]The location in memory of the first assembly instruction of the function.

ulatorPrime is passed the address of the code storage area, its size and the number of instructions it contains. Kheiron/C injects calls to the RegisterSave, EmulatorPrime and emulate_begin functions (in this order) as prologues for the functions to be emulated and allows the target program to continue. A modification to STEM's emulate_begin function causes STEM to begin its instruction fetch from the address of the code storage area.

At the end of this process, the instrumented function, when invoked, will cause the STEM emulator to be loaded and initialized with CPU and segment register values as well as enough information to cause our dynamically-loadable version of STEM to alter its instruction pointer after executing the relocated instructions and continue the emulation of the remaining instructions of the function. After the initialization, the injected call to emulate_begin will cause STEM to begin its instruction fetch-decode-execute loop thus running the function under the emulator.

The final modification to STEM addresses the STEM-to-CPU transition, which occurs when the emulator needs to unload and allow the real CPU to continue from the address after the function call run under the emulator. Rather than inject calls to emulate_end, we modify STEM's emulate_begin function such that it keeps track of its own *stack-depth*. Initially, this value is set to 0, if the function being emulated contains a *call* (0xE8) instruction, the stack-depth is incremented, when it returns the stack-depth is decremented. STEM marks the end of emulation by the detection of a *leave* (0xC9) or *return/ret* (0xC2/0xC3) at stack-depth 0. At this point, the emulator either commits or restores the CPU registers and, using the address stored in the saved stack pointer register (esp), causes the real CPU to continue its execution from the instruction immediately after the emulated function call.

As a comparison, performing STEM injection using Pin 2.0 [66] would call for less machinations with respect to initializing STEM (i.e. the CPU-to-STEM transition). Pin maintains two copies of the program text in memory, the original program text and the instrumented version of the program text (generated just-in-time by Pin) hence, there is no need for trampolines, nor any need to save instructions dislocated by jumps into the trampoline as in the Dyninst case. Once STEM is loaded, its instruction pointer can simply be set to the base address of the function which will mark the beginning of the original un-instrumented version of the function. Further, Pin 2.0 guarantees that analysis code – code executed at instrumentation points – will be inlined into the instrumented version of the function, as long as the analysis code contains no branches [14]. This inlining guarantee should allow the CPU state-capture assembly instructions needed to initialize STEM's registers to be emitted inline in the instrumented version of the function, as occurs at the source level with the original version of STEM. However, we need to verify that inlining actually occurs and devise an appropriate strategy for the STEM-to-CPU transition.

References

[1] Aaron Brown et al. Benchmarking Autonomic Capabilities: Promises and Pitfalls. In *1st International Conference on Autonomic Computing*, 2004.

[2] MySQL AB. MySQL Internals Manual. http://downloads.mysql.com/docs/internals-en.pdf, 2007.

[3] Akshay Luther et al. Alchemi: A .NET-Based Enterprise Grid Computing System. In *6th International Conference on Internet Computing*, June 2005.

[4] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeno JVM. In *Object Oriented Programming Systems, Languages and Applications*, 2000.

[5] Aaron Brown. Towards availability and maintainability benchmarks: A case study of software raid systems. Masters dissertation, University of California, Berkeley, 2001. UCB//CSD011132.

[6] Aaron Brown and Charlie Redlin. Measuring the Effectiveness of Self-Healing Autonomic Systems. In *2nd International Conference on Autonomic Computing*, 2005.

[7] Bryan Buck and Jeffrey K. Hollingsworth. An API for Runtime Code Patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[8] C. Soules et. al. System Support for Online Reconfiguration. In *USENIX Annual Technical Conference.*, 2003.

[9] George Candea, James Cutler, and Armando Fox. Improving Availability with Recursive Micro-Reboots: A Soft-State Case Study. In *Dependable systems and networks - performance and dependability symposium*, 2002.

[10] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *USENIX Annual Technical Conference*, pages 15–28, 2004.

[11] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In *Symposium on Operating Systems Principles*, pages 73–88, 2001.

[12] Geoff Cohen and Jeff Chase. An Architecture for Safe Bytecode Insertion. *Software–Practice and Experience*, 34(7):1–12, 2001.

[13] Geoff Cohen, Jeff Chase, and David Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Symposium*, pages 167–178, 1998.

[14] Robert Cohn and Robert Muth. Pin 2.0 User Guide. http://rogue.colorado.edu/pin/docs/3077/, 2004.

[15] Computer World. IBM researchers take AXE to computer security. http://www.infoworld.com/article/05/10/28/HNibmaxe_1.html, 2005.

[16] John DeVale. Measuring operating system robustness.

[17] Donal Lafferty et al. Language Independent Aspect-Oriented Programming. In *18th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications*, October 2003.

[18] Dong Tang et al. Availability Measurement and Modeling for an Application Server. In *International Conference on Dependable Systems and Networks*, 2004.

[19] Dong Tang et al. Assessment of the Effect of Memory Page Retirement on System RAS Against Hardware Faults. In *International Conference on Dependable Systems and Networks*, 2006.

[20] Michael Engel and Bernd Freisleben. Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects. In *4th International Conference on Aspect-Oriented Software Development*, pages 51–62, 2005.

[21] Andreas Frei, Patrick Grawehr, and Gustavo Alonso. A Dynamic AOP-Engine for .NET. Tech Rep. 445, Dept. of Comp Sci. ETH Zurich, 2004.

[22] G. Kiczales et al. An Overview of AspectJ. In *European Conference on Object-Object Programming*, June 2001.

[23] 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*, June 2003.

[24] Gregor Kiczales et. al. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, volume LNCS 1241. Springer-Verlag, 1997.

[25] Rean Griffith and Gail Kaiser. Manipulating managed execution runtimes to support self-healing systems. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, New York, NY, USA, 2005. ACM Press.

[26] Rean Griffith and Gail Kaiser. Manipulating Managed Execution Runtimes to Support Self-Healing Systems. In *Workshop on Design and Evolution of Autonomic Application Software*, May 2005.

[27] Rean Griffith and Gail Kaiser. A Runtime Adaptation Framework for Native C and Bytecode Applications. In *3rd International Conference on Autonomic Computing*, 2006.

[28] Rean Griffith, Giuseppe Valetto, and Gail Kaiser. Effecting Runtime Reconfiguration in Managed Execution Environments. In Manish Parishar and Salim Hariri, editors, *Autonomic Computing: Concepts, Infrastructure, and Applications,*. CRC, 2006.

[29] Ulf Gunneflo, Johan Karlsson, and Jan Torin. Evaluation of error detection schemes using fault injection byheavy-ion radiation. In *Nineteenth International Symposium on Fault-Tolerant Computing*, 1989.

[30] S. Han, K. Shin, and H. Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems, 1995.

[31] Laune Harris and Barton Miller. Practical Analysis of Stripped Binary Code. In *Workshop on Binary Instrumentation and Applications*, 2005.

[32] Howard Kim. AspectC#: An AOSD implementation for C#. Technical Report TCD-CS-2002-55, Department of Computer Science Trinity College, 2002.

[33] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[34] Information Society Technologies (IST). Dependability benchmarking project final report. http://www.laas.fr/DBench/Final/DBench-complete-report.pdf.

[35] Ji Zhu et al. R-Cubed: Rate, Robustness and Recovery An Availability Benchmark Framework. Technical Report SMLI TR-2002-109, Sun Microsystems, 2002.

[36] Ghani Kanawati, Nasser Kanawati, and Jacob Abraham. Ferrari: A tool for the validation of system dependability properties. In *Twenty-second International Symposium on Fault-Tolerant Computing*, 1992.

[37] Johan Karlsson, Jean Arlat, and Gunther Leber. Application of three physical fault injection techniques to the experimental assessment of the mars architecture. In *Fifth Annual IEEE Working Conference on Dependable Computing for Critical Applications*, 1995.

[38] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer magazine*, pages 41–50, January 2003.

[39] Kishor S. Trivedi. *Probability and Statistics with Reliability, Queuing and Computer Science Applications 2nd Edition*. Wiley, 2002.

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

[41] Naveen Kumar, Jonathan Misurda, Bruce Childers, and Mary Lou Soffa. Instrumentation in Software Dynamic Translators for Self-Managed Systems. In *Workshop on Self-Healing Systems*, 2004.

[42] John Lam. CLAW: Cross-Language Load-Time Aspect Weaving on Microsoft's CLR. Demonstration at AOSD 2002.

[43] Jean-Claude Laprie and Brian Randell. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33, 2004. Fellow-Algirdas Avizienis and Senior Member-Carl Landwehr.

[44] James R. Larus and Eric Schnarr. EEL: machine-independent executable editing. In *ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 291–300, 1995.

[45] Serge Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft Press, 2002.

[46] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification Second Edition. http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html, 1999.

[47] Linux Weekly News (LWN.net). Injecting faults into the kernel. http://lwn.net/Articles/209257/, 2006.

[48] H. Madeira, J. Carreira, and J. Silva. Injection of faults in complex computers, 1995.

[49] Mark E. Russinovich and David A. Solomon. *Microsoft Windows Internals 4th Edition*. Microsoft Press, 2005.

[50] Eliane Martins, Cecília M. F. Rubira, and Nelson G. M. Leme. Jaca: A reflective fault injection tool based on patterns. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 483–482, Washington, DC, USA, 2002. IEEE Computer Society.

[51] James Mauro, Ji Zhu, and Ira Pramanick. The system recovery benchmark. In *10th IEEE Pacific Rim International Symposium on Dependable Computing*, 2004.

[52] Daniel Menasce. TPC-W A Benchmark for E-Commerce. http://ieeexplore.ieee.org/iel5/4236/21649/01003136.pdf, 2002.

[53] Michael M. Swift et al. Improving the Reliability of Commodity Operating Systems. In *International Conference Symposium on Operating Systems Principles*, 2003.

[54] Michael M. Swift et al. Recovering Device Drivers. In *6th Symposium on Operating System Design and Implementation*, 2004.

[55] Microsoft. Common Language Infrastructure (CLI) Partition I: Concepts and Architecture, 2001.

[56] Microsoft. Common Language Runtime Metadata Unmanaged API, 2002.

[57] Microsoft. Common Language Runtime Profiling, 2002.

[58] Microsoft. IFS Kit – Installable File System Kit. http://www.microsoft.com/whdc/DevTools/IFSKit/default.mspx, 2005.

[59] Microsoft Corporation. Basic Instincts: Designing With Custom Attributes. http://msdn.microsoft.com/msdnmag/issues/05/05/BasicInstincts/default.aspx, 2005.

[60] Sun Microsystems. The JVM Tool Interface Version 1.0. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html, 2004.

[61] Aleksandr Mikunov. Rewrite MSIL Code on the Fly with the .NET Framework Profiling API. http://msdn.microsoft.com/msdnmag/ issues/03/09/NETProfilingAPI/, 2003.

[62] Alexander V. Mirgorodskiy and Barton P. Miller. Autonomous Analysis of Interactive Systems with Self-Propelled Instrumentation. In *12th Multimedia Computing and Networking*, January 2005.

[63] NIST. National institute of standards and technology (nist) website. http://www.nist.gov/.

[64] Christian Poellabauer, Karsten Schwan, Sandip Agarwala, Ada Gavrilovska, Greg Eisenhauer, Santosh Pande, Calton Pu, and Matthew Wolf. Service Morphing: Integrated System- and Application-Level Service Adaptation in Autonomic Systems. In *Autonomic Computing Workshop, Fifth Annual International Workshop on Active Middleware Services*, June 2003.

[65] Rean Griffith and Gail Kaiser. Adding Self-healing Capabilities to the Common Language Runtime. Technical Report CUCS-005-05, Columbia University, 2005.

[66] Vijay Janapa Reddi, Alex Settle, Daniel A. Connors, and Robert S. Cohn. PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education. In *Workshop on Computer Architecture Education*, 2004.

[67] S. M. Sadjadi and P. K. McKinley. Using Transparent Shaping and Web Services to Support Self-Management of Composite Systems. In *Second IEEE International Conference on Autonomic Computing*, June 2005.

[68] S. M. Sadjadi, P. K. McKinley, B. H. C. Cheng, and R. E. K. Stirewalt. TRAP/J: Transparent Generation of Adaptable Java Programs. In *International Symposium on Distributed Objects and Applications*, October 2004.

[69] A R Sahner and S K Trivedi. Reliability modeling using sharpe. Technical report, Durham, NC, USA, 1986.

[70] Bradley Schmerl and David Garlan. Exploiting Architectural Design Knowledge to Support Self-Repairing Systems. In *14th International Conference of Software Engineering and Knowledge Engineering*, 2002.

[71] Security Innovation. Holodeck Enterprise Edition Features and Benefits. http://www.securityinnovation.com/holodeck/features.shtml, 2007.

[72] Shang-Wen Cheng et al. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.

[73] Michael Shapiro. Self-healing in modern operating systems. http://www.acmqueue.com/modules.php?name=Content&pa=showpage&pid=242.

[74] Charles Shelton and Philip Koopman. Using Architectural Properties to Model and Measure System-wide Graceful Degradation. In *Workshop on Architecting Dependable Systems*, 2002.

[75] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a Reactive Immune System for Software Services. In *USENIX Annual Technical Conference*, pages 149–161, April 2005.

[76] Volkmar Sieh and Kerstin Buchacker. Umlinux – a versatile swifi tool. In *Proceedings of the Fourth European Dependable Computing Conference*, 2002.

[77] SPEC. Standard performance evaluation corporation (spec) website. http://www.spec.org/.

[78] Amitabh Srivastava and Alan Eustace. ATOM: a system for building customized program analysis tools. In *ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 196–205, 1994.

[79] Sun Microsystems. Predictive self-healing in the solaris 10 operating system. http://www.sun.com/software/whitepapers/solaris10/self_healing.pdf.

[80] Sun Microsystems. Annotations. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html, 2004.

[81] A. Tamches and B. P. Miller. Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels. In *3rd Symposium on Operating Systems Design and Implementation*, pages 117–130, 1999.

[82] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, 1992. TAN a 92:1 2.Ex.

[83] Tool Interface Standards (TIS) Committee. Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2. http://www.x86.org/ftp/manuals/tools/elf.pdf, 1995.

[84] TPC. Transaction processing performance council (tpc) website. http://www.tpc.org/.

[85] Timothy K. Tsai, Ravishankar K. Iyer, and Doug Jewitt. An approach towards benchmarking of fault-tolerant commercial systems. In *Symposium on Fault-Tolerant Computing*, pages 314–323, 1996.

Acknowledgments