The 7U Evaluation Method: Evaluating Software Systems via Runtime Fault-Injection and Reliability, Availability and Serviceability (RAS) Metrics and Models

Rean Griffith

Submitted in partial fulfillment of the
Requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2008

Abstract

The 7U Evaluation Method: Evaluating Software Systems via Runtime Fault-Injection and Reliability, Availability and Serviceability (RAS) Metrics and Models

Rean Griffith

Renewed interest in developing computing systems that meet additional non-functional requirements such as reliability, high availability and ease-of-management/self-management (serviceability) has fueled research into developing systems that exhibit enhanced reliability, availability and serviceability (RAS) capabilities. This research focus on enhancing the RAS capabilities of computing systems impacts not only the legacy/existing systems we have today, but also has implications for the design and development of next generation (self-managing/self-*) systems, which are expected to meet these non-functional requirements with minimal human intervention.

To reason about the RAS capabilities of the systems of today or the self-* systems of tomorrow, there are three evaluation-related challenges to address. First, developing (or identifying) practical fault-injection tools that can be used to study the failure behavior of computing systems and exercise any (remediation) mechanisms the system has available for mitigating or resolving problems. Second, identifying techniques that can be used to quantify RAS deficiencies in computing systems and reason about the efficacy of individual or combined RAS-enhancing mechanisms (at design-time or after system deployment). Third, developing an evaluation methodology that can be used to objectively compare systems based on the (expected or actual) benefits of RAS-enhancing mechanisms.

This thesis addresses these three challenges by introducing the 7U Evaluation Methodology, a complementary approach to traditional performance-centric evaluations that identifies criteria for comparing and analyzing existing (or yet-to-be-added) RAS-enhancing mechanisms, is able to evaluate and reason about combinations of mechanisms, exposes under-performing mechanisms and highlights the lack of mechanisms in a rigorous, objective and quantitative manner.

The development of the 7U Evaluation Methodology is based on the following three hypotheses. First, that runtime adaptation provides a platform for implementing efficient and flexible fault-injection tools capable of in-situ and in-vivo interactions with computing systems. Second, that mathematical models such as Markov chains, Markov reward networks and Control theory models can successfully be used to create simple, reusable templates for describing specific failure scenarios and scoring the system's responses, i.e., studying the failure-behavior of systems, and the various facets of its remediation mechanisms and their impact on system operation. Third, that combining practical fault-injection tools with mathematical modeling techniques based on Markov Chains, Markov Reward Networks and Control Theory can be used to develop a benchmarking methodology for evaluating and comparing the reliability, availability and serviceability (RAS) characteristics of computing systems.

This thesis demonstrates how the 7U Evaluation Method can be used to evaluate the RAS capabilities of real-world computing systems and in so doing makes three contributions. First, a suite of runtime fault-injection tools (Kheiron tools) able to work in a variety of execution environments is developed. Second, analytical tools that can be used to construct mathematical models (RAS models) to evaluate and quantify RAS capabilities using appropriate metrics are discussed. Finally, the results and insights gained from conducting fault-injection experiments on real-world systems and modeling the system responses (or lack thereof) using RAS models are presented. In conducting 7U Evaluations of

real-world systems, this thesis highlights the similarities and differences between traditional performance-oriented evaluations and RAS-oriented evaluations and outlines a general framework for conducting RAS evaluations.

Contents

List of Figures

List of Tables

Acknowledgments

Acknowledgments go here.

Chapter 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) [140], the Standard Performance Evaluation Corporation (SPEC®) [177] and the Transaction Processing and Performance Council (TPC) [190], to demonstrate the feasibility of some experimental system prototype. However, there are a number of other demands placed on computing systems besides being fast.

Recent renewed interest, [40, 79, 100, 102, 113], in realizing computing systems that meet additional non-functional requirements such as reliability, high availability and ease-of-management/self-management (also referred to as serviceability) has fueled research efforts into enhancing the reliability, availability and serviceability (RAS) capabilities of existing/legacy systems as well as next-generation self-managing, self-configuring, self-healing, self-optimizing and self-protecting systems (collectively referred to as self-* systems).

A common desired characteristic of these systems is that they collect, analyze and act on information about their own operation and changes to their environment while meeting their functional requirements. Whereas instrumenting systems, collecting, analyzing and

acting on behavioral and environmental data potentially impact the performance of a system by diverting processing cycles away from meeting functional requirements, these diverted cycles are used by mechanisms concerned with improving the RAS capabilities of the system by effecting a feedback/monitoring loop around it.

To reason about tradeoffs between RAS-enhancing 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 resulting performance numbers convey little about the efficacy of the mechanisms.

Performance measures do not allow us to analyze the expected or actual impact (beyond system overheads) of individual or combined mechanisms on the system's operation. They are inadequate for comparing the efficacy of individual or combined RAS-enhancing mechanisms, discussing tradeoffs between mechanisms, evaluating different styles of mechanisms (reactive vs. preventative vs. proactive) or reasoning about the composition of multiple mechanisms. In essence, performance metrics limit the scope and depth of analysis that can be performed on systems possessing (or considering the inclusion of) RAS-enhancing mechanisms.

Reasoning about the RAS capabilities of the systems of today or the self-* systems of tomorrow also involves addressing three evaluation-related challenges. First, developing (or identifying) practical fault-injection tools that can be used to study the failure behavior of computing systems and exercise any (remediation) mechanisms the system has available for mitigating or resolving problems. Second, identifying techniques that can be used to quantify RAS deficiencies in computing systems and reason about the efficacy of individual or combined RAS-enhancing mechanisms (at design-time or after system deployment). Third, developing an evaluation methodology that can be used to objectively compare

systems based on the (expected or actual) benefits of RAS-enhancing mechanisms.

This thesis addresses these three challenges by introducing the 7U Evaluation Methodology, a complementary approach to traditional performance-centric evaluations that identifies criteria for comparing and analyzing existing (or yet-to-be-added) RAS-enhancing mechanisms, is able to evaluate and reason about combinations of mechanisms, exposes under-performing mechanisms and highlights the lack of mechanisms in a rigorous, objective and quantitative manner.

Under the 7U approach, non-functional requirements concerned with reliability, high availability, and serviceability represent additional high-level goals the system is expected to meet. In this thesis, we demonstrate how these goals can be codified as augmentations or additions to the existing policies, service level agreements (SLAs) and service level objectives (SLOs) that govern the system's operation. In developing our methodology we demonstrate techniques that can be used to identify and quantify these goals as well as measure whether they are being met or exceeded.

1.1 Definitions

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

- An **error** is the deviation of system external state from correct service state [107]. Approaches to defining and detecting such deviations include, but are not limited to: monitoring violations of service level agreements (SLAs), quantifying system degradation, self-checking software approaches [157], the use of functional redundancy, e.g., Recovery Blocks [152] or computational redundancy, e.g., N-Version programming [7].

- A **fault** is the adjudged or hypothesized cause of an error [107].

- 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 [102]. This fault-model may include all plausible faults that can affect the system, regardless of whether an explicit remediation/system-response is available.
- **Remediation** is the process of trying to correct a fault. In this thesis, remediation spans the activities of detection, diagnosis and repair since the first step in responding to a fault is detection [102].
- A **failure** is an event that occurs when the delivered service violates an environmental/contextual constraint, e.g., a policy or SLA. This definition allows us to consider multiple perspectives when discussing failures including, but not limited to, that of the end-user [16] or system operator/administrator.
- **Reliability** is a function of the number (or frequency) of end-user interruptions[1].
- **Availability** is a function of the rate of failure/maintenance events and the speed of recovery [89].
- **Serviceability** is a function of the frequency and success of servicing and/or administrative activities addressing failures.

1.2 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. What is required is an evaluation methodology that allows us to go beyond drawing conclusions about the feasibility of using a system with its RAS-enhancing mechanisms enabled and instead directly addresses the issue of quantifying

[1] Reliability may be interpreted as the inverse of the frequency of end-user interruptions.

the expected or actual benefits of these RAS-enhancing mechanisms.

1.3 Requirements

There are a number of elements needed to effectively solve the problem:

1. **Fault-injection techniques** that facilitate "in-situ" and "in-vivo" interactions with computing systems. "In-situ" interactions (in principle[2]) allow us to study the failure-behavior of a computing system in its deployed environment while "in-vivo" interactions allow us to inject faults into running systems. Both techniques offer advantages for studying the failure-behavior of computing systems.

   Studying the failure-behavior of computing systems is a non-trivial task. Reproducing or replicating problems in computing systems may require interacting with the system in its production/deployed environment rather than in a replicated staging area/cleanroom since faults may manifest themselves due to unanticipated interactions between the system of interest and elements (e.g., other software systems) in its environment. Further, depending on the scale and/or complexity of the system, replicating the deployment environment can be a difficult task. Fault-injection tools that can be used "in-situ" would allow us to study the system directly in its current deployment, thereby removing the need for replicating the entire production environment.

   "In-vivo" interactions with computing systems allow us to perform operations on them while they execute. The ability to interact with computing systems in execution gives us a flexible tool that can be used to collect detailed information directly from the internals of a system and make fine-grained modifications to the system

[2] Prudence and/or organizational policies may limit or restrict conducting fault-injection experiments on systems being used by other members of the organization/business. Organizational restrictions on interacting with "live" production systems do not preclude us from using system mirroring and traffic/request replay techniques to create an environment suitable for in-situ studies.

from the inside [64]. A few examples of "in-vivo" interactions include, but are not limited to: dynamically connecting to/disconnecting from running systems, inserting/modifying/removing instrumentation from running systems and performing fine-grained adaptations in running systems [63], e.g., inducing failures.

An additional benefit of "in-situ" and "in-vivo" interactions is that neither requires that source code be available. The ability to interact with computing systems without requiring access to source code has implications for working with legacy and contemporary software systems where source code may not be readily accessible. Whereas we desire tools that can work without requiring access to the source code, access to the source code, however, may enhance our understanding of how the system operates and provide insights into how to perform "safe" adaptations of running systems, see §3.7.8 for an example and §3.10 for more discussion.

Fault-injection techniques that leverage these in-situ and in-vivo interaction capabilities can be used to build tools that can target specific components or subsystems in computing systems, inject faults into them while the system is running and collect data on the system's responses.

2. **Fault-injection tools** that exercise the RAS mechanisms available by inducing/injecting *reasonable* (or *representative*) faults for the system to be evaluated. Whereas there is no shortage of fault-injection tools – example tools include: [77, 112, 68, 99, 95, 70, 165, 116, 172, 123] – the utility of using a specific fault-injection tool in conducting a RAS evaluation depends on the fault-model under consideration and the granularity of the faults that can be injected using the tool. The granularity of the faults in the fault-model must match the granularity of the faults injected by the tool and the semantics of the target system's operation.

   Possible mismatches between the granularity of the faults in the fault-model and the faults that can be injected by the fault-injection tools available prevent us from

appropriately exercising (and studying) the existing RAS-enhancing mechanisms. Further, we may not be able to adequately identify all the RAS deficiencies under the fault-model being considered using these tools since these tools may not trigger/induce the failures that we wish to study. For example, whereas tools like FIST [68] and MARS [99] induce bit flips in chips (e.g. processor or memory) by exposing them to heavy-ion radiation it is not clear whether the bit-flips caused would have a specific (targeted) effect on a given workload such that perturbations to the workload could be detected and compensated for by some RAS mechanism.

When conducting a RAS evaluation of a computing system, the evaluation-process is guided by the fault-model/fault-hypothesis, which codifies the reasonable and/or representative faults of interest. Each fault in the fault-model is associated with an existing (or yet-to-be-added) RAS-enhancing mechanism; as a result, each fault injected by a fault-injection tool should either exercise an existing RAS-enhancing mechanisms or cause a system-response that highlights a RAS deficiency (under the current fault-model) that could be addressed by a yet-to-be-added mechanism.

3. **Analysis techniques** that can be used to quantify (at design-time and post-deployment time) the impact of the faults under consideration as well as the actual or expected impact/benefit of RAS-enhancing mechanisms. The analytical techniques we employ should allow us to identify RAS deficiencies or under-performing RAS mechanisms. Further, they should facilitate the study of individual or combined mechanisms as well as accommodate the analysis of different styles of mechanisms (e.g., reactive, proactive and preventative).

   To evaluate and compare the RAS capabilities of computing systems we need to be able to quantify the impact of faults in terms of reliability, availability and/or serviceability metrics. Quantifying fault-impacts in terms of RAS metrics involves identifying and measuring the facets of reliability, availability and serviceability that

vary when faults occur.

Fortunately, there are many facets of reliability, availability and serviceability that can be used, and have been used in the past, to quantify fault-impacts including, but not limited to: frequency of service interruptions or outages, yearly downtime and its associated "costs" (time and money spent on restoring complete or partial service, time and money lost due to system unavailability, end-user downtime, etc.), meantime to system breakdown, the number of servicing visits, the frequency of servicing visits, the ability to meet SLA targets, the ability to meet production targets, the ability to avoid or mitigate production slowdowns and system stability.

Further, there are a number of analytical tools/approaches, which have been used in other engineering disciplines, that we can use to inform our analyses. Probability Theory, Queuing Theory, Stochastic Petri Nets, Markov Chains and Markov Reward Networks have been used in Computer Engineering and Computer Science to study the Reliability and Availability properties of specific hardware and/or software systems [101, 69]. Techniques from Control Theory – used to study the behavior of dynamic systems – have found applications in Mechanical Engineering and more recently Computer Science [90] where the regulation of one or more system objectives is required.

4. An **Evaluation Methodology** that allows us to analyze the details of RAS-enhancing mechanisms (the micro-view) in the context of the high-level goals governing the system's operation (the macro-view).

   Establishing a link between the details of the mechanisms and their expected or actual impact on high-level goals allows us to reason about the benefits of existing RAS mechanisms or the necessity of additional mechanisms. Further, it informs discussions about the suitability of system objectives concerned with (or affected by) reliability, availability and serviceability issues.

1.4 Hypotheses

This thesis investigates three hypotheses for enabling the RAS evaluation of software systems:

1. Runtime adaptation provides a platform for implementing efficient and flexible fault-injection tools capable of "in-situ" and "in-vivo" interactions with computing systems.

2. Mathematical models such as Markov chains, Markov reward networks and Control theory models can successfully be used to create simple, reusable templates for describing specific failure scenarios and scoring the system's responses, i.e., studying the failure-behavior of systems, and the various facets of its remediation mechanisms and their (actual or expected) impact on system operation.

3. RAS models and experiments using flexible fault-injection tools can be used together to develop a RAS benchmarking methodology for computing systems. This combination provides practical advantages over existing purely model-based (e.g. [97, 182]), purely measurement-based (e.g. [37, 191, 17]) or simulation-based evaluation approaches (e.g. [50, 187]).

1.5 Thesis outline

The rest of this thesis is organized as follows:

- Chapter 2 describes the origins of this thesis, the motivations behind it and briefly summarizes its contributions.

- Chapter 3 presents techniques for enabling a range of runtime adaptations in software applications running in a variety of managed and unmanaged execution environments. The latter part of this chapter presents and evaluates Kheiron, a suite of runtime

adaptation tools for .NET, Java and compiled-C applications.

- Chapter 4 identifies analytical tools used to evaluate facets of reliability, availability and serviceability.
- Chapter 5 outlines the considerations of traditional and non-traditional benchmarks for software systems, develops the ideas leading to a discussion of the 7U-Evaluation Methodology and compares the 7U to other evaluation approaches. The latter part of the chapter describes experiments and presents results from conducting 7U-evaluations on a number of target systems.
- Chapter 6 summarizes the contributions of the thesis, presents its conclusions and discusses the possibilities for future work.

Chapter 2

Motivation

The research leading to the development of a Reliability, Availability and Serviceability (RAS) evaluation methodology had its origins in work on on-the-fly system reconfiguration and retro-fitting self-management (specifically monitoring, pattern/event analysis, repair and reconfiguration) capabilities onto existing systems, conducted under the DARPA Dynamic Assembly for Systems, Adaptability, Dependability and Assurance (DASADA) program [40].

2.1 DASADA Overview

The DASADA program was concerned with tackling the problem of system complexity and manageability through the identification of technologies that would allow systems to gauge their own health and rapidly integrate new heterogeneous, common-off-the shelf (COTS) components or reconfigure existing components while in operation – *Continual Validation and Co-ordination*. The focus of the program was on systems-of-systems and the technologies developed were expected to deal with heterogeneous systems/components and internet-scale systems [41]. A reference architecture for the realization of the DASADA

project is shown in Figure 2.1.

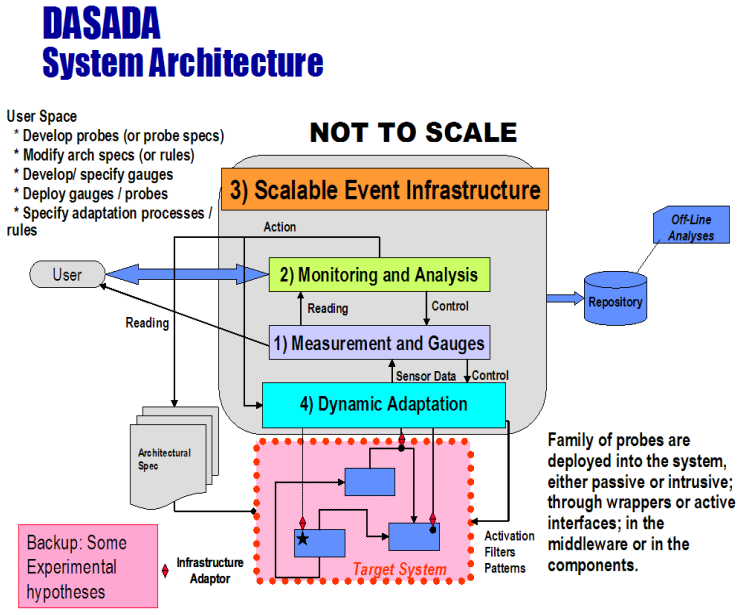

Figure 2.1: DASADA system architecture [41]

At a high-level, technologies devised to meet the requirements of DASADA revolved around the concepts of events, probes, gauges, models, controllers and effectors.

Events are the units of information communication and represent system activities of varying granularities, from low-level resource readings to high-level component or system interactions. **Probes** collect primitive data from the target system and send this information to gauges in the form of events. **Gauges** aggregate, filter and interpret probe data based on information contained in models of the system under consideration. **Models** codify properties of the target system. These properties may include, but are not limited to, structural, behavioral and domain considerations. **Controllers** use gauge-output and system models to decide which reconfigurations/adaptations need to be performed. **Effectors**/actuators are responsible for carrying out adaptations on the target system and its components.

One of the consortiums [1] of researchers collaborating under DASADA produced technologies used in Kinesthetics eXtreme (KX pronounced “kicks”) our implementation of the DASADA

[1]This consortium included: Teknowledge, BBN, CMU, WPI, OBJS, UMass and Columbia University.

reference architecture [93].

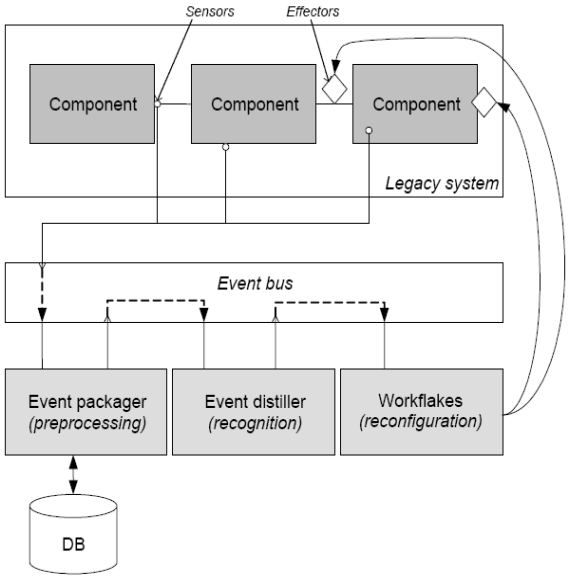

Figure 2.2: Kinesthetics eXtreme (KX) system architecture [88]

2.2 Kinesthetics eXtreme (KX)

KX is a platform for retro-fitting self-managing (i.e., monitoring, reconfiguration or repair) capabilities onto systems. It provides a framework for collecting and interpreting application-specific behavioral and performance data at runtime from a variety of systems and components. In its deployment, KX monitors, analyzes, reconfigures and/or repairs applications guided by models of application-level semantics, protocols and performance requirements [93]. These models express expected correct behaviors of the system and may be used to anticipate and address error situations. KX also includes a software feedback-control loop that plans, coordinates and automatically handles contingencies arising from reconfiguration or repair activities.

KX is an example of a *generic*, *externalized* adaptation platform (see Figure 2.2). It provides a general platform for monitoring and reconfiguring software systems. In order to

manage/interact with a variety of target systems, KX does not formally mandate specific probe, gauge, modeling, controller or effector technologies, rather it is able to loosely couple disparate implementations via semi-structured event formats and a content-based-routing, publish-subscribe communications substrate, such as Siena [26] [2], Elvin [167] [3] or Gryphon [12] [4], which facilitates the routing of probe data to interested gauges and controllers [5].

An important goal for KX is to be able to monitor and adapt existing/legacy systems [6]. However, there is a major practical issue that needs to be addressed. Whereas gauges, controllers and models can be generic, and possibly reusable across different target systems, probes and effectors may be more tightly coupled to the target system, its components/sub-systems and/or its environment. As a result, the degree to which KX can remain completely externalized (separate) from the system being managed (monitored and/or adapted at runtime) depends heavily on the *probe* and *effector* technologies it employs.

2.2.1 Probing Technologies used in KX

In the past, KX has employed a number of different probing solutions developed by others, each with their relative strengths and weaknesses:

- **AIDE**, the Active Interface Development Environment [74], developed by Worcester Polytechnic Institute (WPI), was created to enable the use of Active Interfaces [73] to adapt Java classes. Components built with active interfaces support two separate interfaces – one interface representing its functionality and the other representing adaptation facilities. Application builders can use the adaptation interface to associate

[2] Siena was developed at the University of Colorado at Boulder.
[3] Elvin was developed at the University of Queensland, Australia.
[4] Gryphon was developed at IBM Research.
[5] Early versions of KX have used both Siena and Elvin for communications.
[6] The term legacy here considers a) systems where the source code may not be available or easily accessible and b) systems that were not constructed with all/any of the self-management capabilities that could be beneficial

callbacks with the before and/or after invocation phases of a component's methods[7]. The AIDE compiler takes the source code of a Java class and inserts hooks before and after each phase of a method [58]. As implied from the description, AIDE is limited to the insertion of probes in Java applications. Further, probe-insertion requires access to the source-code of the target system.

- **ProbeMeister**, developed by Object Services and Consulting Inc. (OBJs), used an early implementation of the Java Debug Interface (JDI) – released as part of Sun Microsystems' JDK 1.4 [131] – to deploy probes into (local or) remotely running Java software [146]. The JDI is part of the Java Platform Debugger Architecture (JPDA) [130], and it (the JDI) defines a high-level Java Language interface which tool developers can use to write remote debugger applications. ProbeMeister instruments Java bytecode and uses the HotSwap Class File Replacement feature available in the v1.4.x Java Virtual Machine (JVM) [131] to dynamically replace an existing class with an instrumented one. Whereas ProbeMeister does not need access to the source code of the application being modified, the JVM where it is hosted needs to run with debugging services enabled. For example, to invoke methods on remote objects, ProbeMeister needs to cause a breakpoint in the remote application. However, running a Java application under a debugger can impose a non-negligible performance penalty.

- **Mediating Connectors**, developed by Teknowledge, is a technology for mediating all shared library calls [11] using wrappers and as a result operate in the environment surrounding a target application – the operating system (specifically the Windows operating system). Mediators can instrument interfaces, monitor interactions, integrate components together or sandbox potentially harmful or unreliable components. Whereas Teknowledge's approach is theoretically applicable to programs running on other operating systems that package functionality in shared libraries (modulo

[7] This is similar to the "before-method" and "after-method" advice concepts in Aspect Oriented Programming (AOP) [61].

idiosyncrasies of executable linking, program startup, library loading and process creation in Unix-based operating systems and in other members of the Windows family of operating systems), in practice, an implementation was only provided for programs running under Windows NT and porting to other operating systems was considered non-trivial.

2.2.2 Effector Technologies used in KX

KX also experimented with different effector technologies:

- **Worklets** [193], mobile agent technology, was used as the primary effector technology in KX. Worklets were originally developed as "...rehostable lightweight mobile agents for on-the-fly process construction, adaptation and evolution, system reconfiguration, and knowledge propagation" [92]. They can be transmitted from host to host along a pre-determined or dynamically determined route based on changes in a host and/or the host's environment. In KX, Worklets carry self-contained mobile code (JPython or Java) that can adapt (reconfigure) local target components based on the state of the component(s) and the capabilities of the worklet. Worklet interaction with the target system is mediated by service access modules (SAMs), which translate the internal configuration capabilities exposed by the host into terminology meaningful to the worklet. Whereas the movement of Worklets going from one host to another performing reconfigurations can be used to effect a flexible micro-workflow of co-ordinated reconfiguration activities, at each hop, the configuration actions that a Worklet can perform are limited by the configuration "knobs"/capabilities exposed by the target system or component.

- **JMX**, Java Management Extensions, provide a standard way of managing and monitoring local and/or remote resources, e.g., applications, devices, services and networks [133]. Each resource is instrumented with Java objects called ManagedBeans

(MBeans). One or more MBean codifies the monitoring and/or management interface exposed by a resource. [193] details a case study involving the combination of Worklets and JMX to effect the monitoring and reconfiguration of distributed software systems. Whereas MBeans can provide a uniform way to monitor and manage resources, they must either be embedded at the source level (for Java applications only), or they must interact with Java-based and non-Java-based resources via their existing/accessible configuration "knobs".

2.3 Short-term Research Objectives after KX

For a completely externalized approach to the dynamic adaptation of systems, KX relies on the judicious placement of probes in or around the target system and the exposure of appropriate configuration "knobs" by the target system for effecting reconfigurations and adaptations.

Experience with the probe and effector implementations used in the KX case studies [94, 88, 93, 192] during and after DASADA support this assessment of KX and highlight a number of limitations to retrofitting self-management capabilities (monitoring, reconfiguration and/or repair) onto existing/legacy systems using an externalized adaptation engine including, but not limited to:

- The extent to which monitoring, reconfiguration or repair activities could be carried out on an existing/legacy system was largely determined by the built-in instrumentation, reconfiguration or repair facilities exposed for external manipulation. For systems or components lacking these facilities, probes were limited to being placed in the environs of the target, e.g., monitoring the resource utilization of a process [192] or monitoring network activity. Similarly, effectors were limited to relatively coarse-grained reconfiguration or repair activities, e.g., editing a configuration file and/or

restarting a process [94].

- Limited ability to embed or modify monitoring, configuration or repair mechanisms in existing/legacy systems without recompiling and/or relinking the target application. ProbeMeister [146] supports the ability to embed new probes into Java applications only.
- Limited ability to remove instrumentation from existing/legacy systems without recompiling and/or relinking the target application.
- Limited ability to effect fine-grained repairs or reconfigurations in target systems, e.g., targeted interactions with individual components vs. interactions with the aggregating/composed application.

To address these limitations we identified four short-term research objectives for improving the probe and effector technologies used to retrofit self-management capabilities onto existing/legacy systems:

1. Develop techniques that support the dynamic insertion, modification and/or removal of monitoring, reconfiguration and/or repair facilities, without requiring recompilation or relinking of the target system/component - i.e., access to the source code should not be a requirement.
2. Develop tools and techniques that are able to interact with systems/components written in multiple programming languages and running on different operating systems.
3. Develop tools and techniques that are transparent to the target systems/components being adapted.
4. Develop tools and techniques that are able to perform fine-grained dynamic adaptations in existing/legacy systems.

2.4 Long-term Research Objectives

Our work towards retro-fitting self-management capabilities onto existing/legacy systems presented an interesting set of evaluation challenges. Specifically, to evaluate a self-managing system realized via retro-fitting we must consider:

1. The kinds of self-management capabilities that can be retro-fitted.
2. The approaches and technologies used to retro-fit self-management.
3. The impact of these technologies on specific functional characteristics of the system.
4. The efficacy of the self-management capabilities added to the system, i.e., the impact of these capabilities on specific non-functional characteristics of the system.

For practical reasons however, these challenges need to be refined. The challenges (as stated above) are *overly broad* – with respect to the self-management capabilities to be evaluated – and *overly restrictive* with respect to the class of systems considered – self-managing systems realized via retro-fitting.

2.4.1 Scoping the Self-Management Capabilities to be Evaluated

Under DASADA – DARPA's initiative to tackle system complexity and manageability issues – the term self-management was related to the identified principles of Continual Validation and Co-ordination: system monitoring, modeling, dynamic repair and dynamic reconfiguration. However, post-DASADA, the notion of self-management took on a broader context with the advent of *Autonomic Computing* in 2001 [79].

Autonomic Computing is IBM's proposal for addressing issues of system automation and system complexity. [79] identifies eight key elements (properties) of autonomic systems, which can be used to classify systems into one (or more) of four distinct classes – self-configuring systems, self-healing systems, self-optimizing systems and self-protecting

systems [8].

The following definitions for the four classes of self-* systems are adapted from [100]:

- Self-Configuring systems configure themselves automatically in accordance with high-level policies – representing business-level objectives. When a component is introduced, it will automatically learn about and take into consideration the composition and configuration of the system and incorporate itself seamlessly, while the rest of the system adapts to its presence.
- Self-Healing systems detect, diagnose, and repair localized hardware and software problems.
- Self-Optimizing systems continually seek ways to improve their operation, identifying and seizing opportunities to make themselves more efficient in performance or cost.
- Self-Protecting systems will defend the system as a whole against large-scale, correlated problems arising from malicious attacks.

In Autonomic Computing, the goal of self-management "...is to free system administrators from the details of system operation and maintenance..." [100]. As a result, this contemporary definition of self-management encompasses all four aspects of self-configuration, self-healing, self-optimization and self-protection.

Evaluating self-management capabilities of systems considering *all* four sub-areas (self-configuration, self-healing, self-optimization and self-protection) is a non-trivial task. As a result, the first step in refining the evaluation challenges outlined at the beginning of Section 2.4 is to focus on one of the four sub-areas of self-management.

Our past experience with effecting dynamic reconfigurations and repairs in systems (via KX) provided a suitable foundation for exploring the area of **Self-Healing** systems. Further, our short term research goals (Section 2.3) of developing more flexible, dynamic probe and

[8] Each system-class maps to a distinct research sub-area in Autonomic Computing.

effector technologies align nicely with the proposed core sub-areas of self-healing systems research – problem detection, diagnosis and repair.

2.4.2 Expanding the Classes of Systems to be Evaluated

Whereas focusing on the evaluation of self-managing (later refined to self-healing) systems realized via retrofit is specific to evaluating systems enhanced by frameworks like KX, this focus is unnecessarily restrictive for a number of reasons.

First, whether self-healing systems are realized via retrofit or via design, the approaches and techniques used to evaluate the efficacy of their self-healing capabilities are expected to be similar (if not identical) while evaluation approaches and tools concerned with the enabling technologies (runtime retro-fitting tools and technologies vs. design-time tools and technologies) are expected to differ, resulting in a multi-part evaluation process. Therefore, the first step in expanding the classes of systems to be evaluated is to consider self-healing systems regardless of whether they are realized by retrofit or by design.

Second, challenges associated with finding systems, which exhibit all the desired characteristics of self-healing systems, to evaluate or compare against. With a nascent research area such as autonomic computing, it will take some time for a) fully self-healing systems to appear, and b) researchers to determine whether any properties of existing systems can be mapped to the desiderata of self-healing systems [102]. The second step in expanding the classes of systems to be evaluated is to consider partially self-healing systems and/or existing systems re-classified as self-healing systems.

Further, an additional implication of a dearth of self-healing systems, is that the classes of systems to be evaluated may also be expanded to consider non-self-healing systems in order to facilitate comparisons between a non-self-healing/"vanilla" version of a system, $S_{vanilla}$, with its self-healing counterpart, $S_{self-healing}$.

Whereas expanding the classes of systems to be evaluated to include non-self-healing systems may at first seem overly permissive, additional motivation for this decision can be obtained by an examination of the expected benefits of self-healing systems.

Based on desired capabilities of self-healing systems provided in [79] and [100], we can identify and summarize a number of expected benefits including, but not limited to:

- Improved **reliability** resulting from the system's ability to automatically detect, diagnose and repair problems.
- High **availability** from the system's ability to orchestrate and effect repair activities online/dynamically – perhaps degrading its operation if necessary.
- Improved **manageability/serviceability** by shifting responsibility for some of the management/administration activities (e.g., problem detection, problem determination and problem resolution) onto the system, thereby reducing the management burden placed on system administrators.

These expected benefits, however, are not exclusive to self-healing systems alone, rather they are desirable characteristics for software systems in general. As a result, a Reliability, Availability and Serviceability evaluation is equally applicable/relevant to self-healing and non-self-healing systems.

2.5 Revised Research Agenda

The general theme of conducting reliability, availability and serviceability (RAS) evaluations of systems allows us to align the short term research objectives, concerned with developing flexible, dynamic probe and effector technologies, and the long-term research objectives concerned with assessing the impact and efficacy of any self-healing mechanisms a system may possess or be retro-fitted with.

Probes and effectors allow us to obtain information on the system's execution and initiate changes to the system's components and/or configuration respectively. Embedding probes dynamically allows us to provide additional (or modify existing) detection and diagnostic services, which we expect to impact the system's reliability and serviceability. Similarly, the ability to introduce or modify effectors allows us to repair or reconfigure the system as well as exercise the system's built-in or retro-fitted self-healing mechanisms via targeted fault-injection. In the former scenario, we expect dynamic repair or reconfiguration to impact the system's reliability, availability and serviceability, whereas in the latter scenario targeted fault-injection allows us to study the failure behavior of the system and evaluate the efficacy of any mechanism(s) the system has in place to deal with the faults injected.

To reason about the RAS-properties of systems we need to be able to evaluate target systems from three broad categories:

1. **Category A** – Systems without any self-healing/RAS-enhancing mechanisms. Systems in this category either have no RAS-enhancing mechanisms or have their mechanisms turned off. Evaluations of these systems will primarily focus on the failure behavior of systems and ways to quantify the impact of failures on the system.

2. **Category B** – Systems with some retro-fitted self-healing/RAS-enhancing mechanisms. Systems in this category include those retro-fitted using externalized adaptation platforms like KX. Evaluations of these systems must consider the feasibility of the retro-fitting techniques/technologies as well as the efficacy of the retro-fitted feedback loop.

3. **Category C** – Systems with some built-in self-healing/RAS-enhancing mechanisms. Evaluations of these systems will focus primarily on the efficacy of the built-in mechanisms; however, discussions of approaches and techniques used to design and develop these systems may warrant some consideration.

Whereas the development of dynamic probe and effector technologies are necessary to

realize Category B systems, these technologies can also be used to develop fault-injection tools for studying the failure behavior of Category A systems and exercising the RAS-enhancing mechanisms in Category B and C systems. Building fault-injection tools on top of a dynamic adaptation foundation allows us to use these tools *in-vivo* (while the system executes) and *in-situ* (in the system's current deployment rather than a staging environment or "clean-room") on Category A, B and C systems.

Interest in the RAS-evaluations of Category A, B and C systems guide our revised research agenda:

- Develop techniques, tools and identify principles for enabling in-vivo and in-situ retro-fitting/adaptations in systems. This research direction continues the work started in KX with a focus on supporting fine-grained adaptations of systems/components written in multiple languages, running on different platforms.

- Develop tools capable of in-vivo and in-situ fault-injection. This research direction is concerned with designing fault-injection tools and fault-models for target systems/components, studying the failure behavior of systems and exercising (where possible) any RAS-enhancing mechanisms available.

- Develop techniques for quantitatively reasoning about the the impact of faults and the benefits of RAS-enhancements. This research direction is concerned with identifying design-time and/or post-deployment time analytical techniques and metrics for quantifying RAS-deficiencies in systems and studying individual or combinations of existing or proposed RAS-enhancing mechanisms. To increase their applicability, the analytical techniques should be able to account for and compare mechanisms that employ different styles of operation (reactive, proactive or preventative) and different degrees of automation (e.g., to cater for mechanisms that may require some human intervention).

- Develop an evaluation methodology that allows us to reason about the details of

RAS-mechanisms or the lack of RAS-mechanisms in the context of the high-level goals/constraints governing the system's operation. This research direction is concerned with establishing a framework for comparing systems with and without RAS-enhancements (i.e., systems in categories A, B and C). The key focus is to identify ways to relate the details of the RAS-mechanisms a system may have or consider (accounting for composition, style, automation, etc.) to the high-level constraints governing the system's operation including, but not limited to: service level agreements (SLAs), administrator time/servicing activities, mean time to repair (MTTR) and cost considerations. In essence, we seek to investigate how RAS-mechanisms could affect/influence the choice of systems; whether quantitative data on RAS-capabilities and environmental constraints can be used to determine that one system is "better" than another; and finally to investigate the relationship between the process of evaluating and comparing RAS-capabilities to the more well understood and established process of evaluating and comparing performance.

2.6 Summary of Contributions

This thesis presents four contributions:

1. Kheiron, a suite of tools developed to perform runtime adaptations, transparently and with low overheads, on programs written in different languages running on different platforms. Three versions of Kheiron exist, each one targeting a specific platform. Kheiron/CLR manipulates .NET programs running in Microsoft's Common Language Runtime (CLR). Kheiron/JVM manipulates Java programs running in Sun Microsystems' Java Virtual Machine (JVM). Kheiron/C manipulates compiled-C programs (ELF binaries) running on Linux. Despite targeting three very different execution environments all three implementations of Kheiron are built around four

shared principles that allow us to use the unmodified execution environment as a common vehicle for manipulating programs in execution. (Chapter 3)

2. Runtime fault-injection tools for applications and operating systems. We build on Kheiron's dynamic adaptation capabilities and techniques to develop tools for injecting targeted faults into components of the popular N-tier web application stack (including application servers, the web-applications they host and the operating systems they run on). We use these fault-injection capabilities to develop a fault-model for N-tier web-application stacks and study the failure-behavior of the targeted systems/components. (Chapter 3)

3. RAS-models. These are analytical models that can be used at design-time and/or post-deployment to quantitatively reason about facets of reliability, availability and serviceability (potentially or actually) affected by failures and/or mitigated by the existence of reactive, proactive or preventative mechanisms for detection, diagnosis or repair. RAS-models depend on well known mathematical formalisms for studying system-behavior and system-failures including continuous time Markov chains (CTMCs), Markov Reward Networks and Control Theory as part of the process of quantitatively comparing the expected or actual RAS-properties of systems. (Chapter 4)

4. The 7U-Evaluation Methodology. A complementary approach to traditional performance-centric evaluations of systems that focuses on comparing the RAS-capabilities of systems. The 7U evaluates the details of RAS-enhancing mechanisms (micro-view) in the context of the high-level goals governing the system's operation (macro-view), e.g., SLAs, administrator time/servicing activities, Mean time to repair (MTTR), etc. via the combination of fault-injection tools, RAS-models and fault-injection experiments and model-driven-simulations. The 7U emphasizes the link between the mechanism-details and their impact on the policies governing the system as a way to

reason about the overall benefits derived from RAS-mechanisms. Using this link, we develop a RAS-benchmarking framework that allows us to discuss a number of issues quantitatively including: whether there are deficiencies that need to be addressed, the efficacy of current or proposed mechanisms, and the effects on high-level goals when mechanisms are added, replaced, removed or modified.

Part I

Runtime Adaptation and Fault-Injection

This part describes Kheiron, a suite of tools for effecting adaptations and injecting faults in programs running in contemporary managed and unmanaged execution environments based on a shared model of operation.

Chapter 3

Runtime Modification of Systems

Runtime adaptation allows us to effect controlled changes in software systems without having to take them offline. It is a form of **system evolution** – "...modification of a function already provided by the system or extension by the introduction of new functions" [103]. Examples of changes include, but are not limited to: introducing new functionality, modifying existing functionality [166], conducting system upgrades/updates [49] and performing reconfigurations [144, 104], repairs or fine-grained manipulations of program elements (data structures, modules, routines, type definitions, classes, objects,components etc.) [197, 49]. In this thesis we also include runtime fault-injection in this list as a form of runtime adaptation concerned with adding or evaluating (self-)diagnostic capabilities to systems.

Runtime adaptation provides two major benefits: support for in-vivo interactions with systems and support for in-situ interactions with systems. In-vivo interactions allow us to perform operations on systems **while** they execute, whereas in-situ interactions allow us to perform operations on systems **where** they execute, i.e., in their current deployment environment, obviating the need for a separate staging area or clean-room environment. Avoiding the re-creation of the deployment environment has three advantages; 1) system

engineers and operators can interact with systems that are deployed in environments that may be infeasible to duplicate due to cost, complexity, etc., 2) engineers and operators interested in studying the failure behavior of systems may find it more difficult to reproduce failures in a system re-deployed in a clean room, e.g., if factors in the deployed environment contribute to the failure events of interest and 3) allows the deployment organization, rather than the vendors, to manage the assessment process.

We identify four shared requirements for runtime adaptation and runtime fault-injection tools and techniques:

1. Support for in-vivo and in-situ interactions with systems. The tools and techniques developed must be able to interact with systems while they execute. Further, they should be amenable to interacting with systems in their current deployment.

2. The tools and techniques developed should be transparent to the target system. The target system should not require recompilation or relinking.

3. Support for fine-grained interactions with program elements (e.g., data types, type definitions, modules, methods).

4. Support for interacting with applications written in multiple languages, running on different operating system platforms.

In this chapter we develop a generic model for effecting runtime adaptations in systems and present a suite of runtime adaptation tools (Kheiron) that satisfy the above requirements. Our model for effecting runtime adaptations identifies the execution environment as the main enabler of transparent adaptations in existing/legacy software systems and is based on four key facilities exposed by (or transparently added to) contemporary execution environments:

- Profiling/tracing facilities to understand the operation of the target system

- Program steering facilities to modify or augment the control flow of the target system

- Meta-data querying facilities to discover structural properties of the system
- Metadata editing facilities to define/modify structural properties of the system in order to modify the functionality of the target system

We use these four facilities to build and evaluate a suite of runtime adaptation tools for .NET, Java and compiled C applications: Kheiron/CLR, Kheiron/JVM and Kheiron/C respectively. We demonstrate Kheiron's ability to effect a variety of sophisticated fine-grained adaptations in running applications via three case studies concerned with dynamic reconfiguration (Kheiron/CLR) §3.7.8, runtime fault-injection (Kheiron/JVM) §3.8.6, and selective emulation of applications (Kheiron/C) §3.9.4.

The remainder of this chapter is organized as follows: §3.1 introduces common terms used throughout this chapter. §3.2 provides an overview of runtime adaptation. §3.3 presents the motivations behind runtime adaptation. §3.4 provides some background on execution environments including their role and general operation. §3.5 describes some of the challenges involved in facilitating runtime adaptation via the execution environment. §3.6 outlines the hypotheses investigated in this chapter. §3.7 - §3.9 present the implementations of Kheiron and their evaluation. §3.11 covers related work and §3.12 summarizes the contributions made in this chapter.

3.1 Definitions

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

- 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 [111, 124].

- 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 [110]. 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 its 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.

3.2 Overview

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 [104, 144, 143, 75, 166]. 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 [19], or such functionality can be *dynamically added*, i.e., retro-fitted onto them using externalized architectures like KX [94] or Rainbow [169].

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 [170, 102, 162]. 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 thesis, 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 chapter 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 §3.8.6 and Chapter 5. The fault-injection tools we build are examples of *software-implemented fault-injection tools* [77].

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 chapter 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 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/CLR manipulates running .NET applications. **Kheiron/JVM** manipulates running Java applications. Finally, **Kheiron/C** manipulates running compiled C programs on the Linux platform.

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.

3.3 Motivation

The ability to adapt is critical for systems [100]. 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: adding fault-injection, problem detection, diagnosis and in some cases remediation mechanisms.

In this chapter 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: **Inserting or removing system instrumentation** [138] 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) [64]. **Input filtering/audit** to detect misused APIs. **Inserting faults or initiating ghost transactions**[157] 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.

3.4 Background on 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 .NET program or Java bytecode program, runs in a *managed execution environment* such as Microsoft's Common Language Runtime (CLR) or Sun Microsystems' Java Virtual Machine (JVM), respectively, 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 [110]. 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.

3.5 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

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

hardware)[2] at which changes could be made to a running program. Some may expose (reasonably standardized) facilities (e.g., profiling APIs [126, 134]) that allow the state of the program to be queried and manipulated. Further, other facilities (e.g., metadata APIs [125]) may support the discovery, inspection and manipulation of program elements, e.g., type definitions and structures. Finally, there may be mechanisms that 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 that can be performed and/or impact the mechanisms used to inject adaptations.

3.6 Hypotheses

The main hypothesis in this chapter is that: **Runtime adaptation provides a platform for implementing efficient and flexible fault-injection tools capable of "in-situ" and "in-**

[2]The un-managed execution environment includes the operating system.

vivo" interactions with computing systems. In validating this hypothesis we investigate the following supporting propositions:

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 demonstrate how 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.

3.7 Kheiron/CLR: Runtime Adaptation in the Common Language Runtime

The Common Language Runtime (CLR) is the runtime environment in which .NET applications execute. It provides an operating layer between the .NET application and the underlying operating system [110]. The CLR manages the execution of .NET applications, taking on the responsibility of providing services such as application isolation, security sandboxing and garbage collection. Managed .NET applications are called *assemblies* and managed executables are called *modules*. Within the CLR, assemblies execute in *application domains*, which are logical constructs used by the runtime to provide isolation from other managed applications.

.NET applications, as generated by the various compilers that target the CLR, are represented in an abstract intermediate form. This abstract intermediate representation is comprised of two main elements, *metadata* and *managed code*. Metadata is "...a system of descriptors of all structural items of the application – classes, their members and attributes, global items...and their relationships"[110]. *Tokens* are handles to metadata entries; they can refer to types, methods, members, etc. Tokens are used instead of pointers so that the abstract intermediate representation is memory-model independent. Managed code "...represents the functionality of the application's methods...encoded in an abstract binary format known as Microsoft Intermediate Language (MSIL)" [110]. MSIL, also referred to as bytecode, is a set of abstract instructions targeted at the CLR.

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

It should be noted that the ability to interoperate relies on programs' adherence to certain rules on naming conventions, data types, function types and certain other elements, forming

a common denominator for different languages [110]. These detailed rules can be found in the Common Language Specification (CLS) [124].

3.7.1 Common Language Runtime Execution Model

During execution, two major components of the CLR that interact with metadata and bytecode are the *loader* and the *just-in-time (JIT) compiler*. The loader reads the assembly metadata and creates an in-memory representation and layout of the various classes, members and methods on demand as each class is referenced. The JIT compiler uses the results of the loader and compiles the bytecode for each method into native assembly instructions for the target platform. JIT compilation only occurs the first time the method is called in the managed application. Compiled methods remain cached in memory; subsequent method calls jump directly into the native (compiled) version of the method skipping the JIT compilation step, as shown in Figure 3.1.

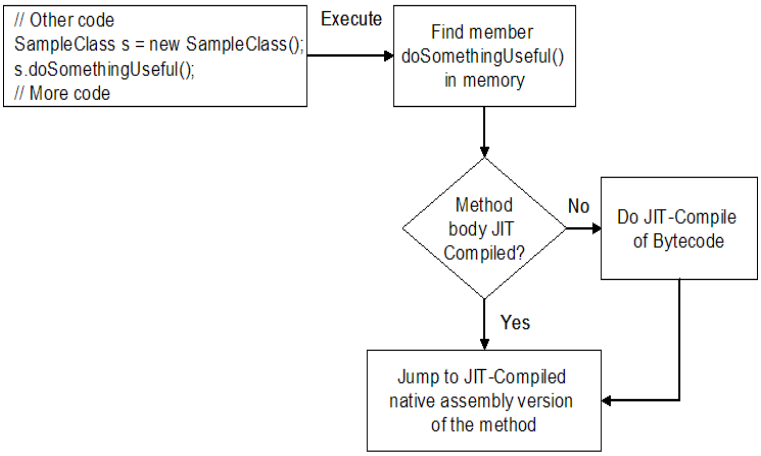

Figure 3.1: Overview of the CLR execution cycle

3.7.2 The CLR Profiler and Unmanaged Metadata APIs

The CLR Profiler APIs allow an interested party (a Kheiron/CLR) to collect information on the execution and memory usage of a running application. There are two interfaces of

interest, ICorProfilerCallback, which Kheiron/CLR must implement, and ICorProfilerInfo, which is implemented by the CLR. Implementors of ICorProfilerCallback (also referred to as the *notifications API* [126]) can receive notifications about assembly loads and unloads, module loads and unloads, class loads and unloads, function entry and exit, and just-in-time compilations of method bodies. The complete list of notifications can be found in [126]. The ICorProfilerInfo interface is used by Kheiron/CLR to obtain details about particular events, e.g., when a module has finished loading, the CLR will call the ICorProfilerCallback::ModuleLoadFinished implementation of Kheiron/CLR passing the *moduleID*. Kheiron/CLR can then use ICorProfilerInfo::GetModuleInfo to get the module's name, path and base load address.

The unmanaged metadata APIs allow users (e.g. Kheiron/CLR) to emit/import data for/from the CLR. These interfaces are considered low-level interfaces that provide fast access to metadata [125]. There are two interfaces of interest, IMetaDataEmit and IMetaDataImport. As the names suggest, the former is used to write metadata and the latter is used to read metadata. As mentioned earlier in Section 3.7, tokens are abstractions used as handles to the metadata of module, type, method, members etc. IMetaDataEmit generates new metadata tokens as metadata is written while IMetaDataImport resolves the details of a supplied metadata token.

3.7.3 Kheiron/CLR Architecture

Our Kheiron/CLR prototype is implemented as a single dynamic linked library (DLL) that includes an implementation of ICorProfilerCallback. Figure 3.2 shows the four (4) main components in our prototype.

- The **Execution Monitor** receives module load, unload and module attached to assembly events, JIT compilation, events and function entry and exit events from the CLR.

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

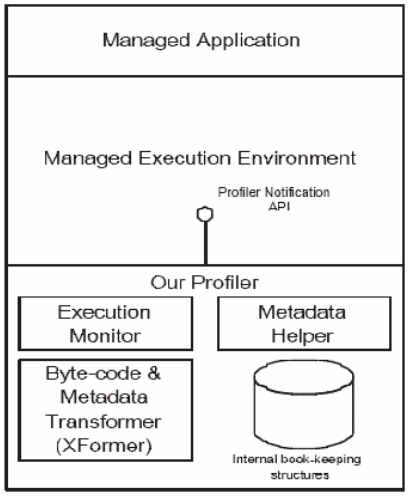

Figure 3.2: Kheiron/CLR prototype architecture diagram

3.7.4 Model of Operation

Kheiron/CLR performs operations on types and methods at various stages in the method invocation cycle shown in Figure 3.3 to make them capable of interacting with modules concerned with performing instrumentation, fault-injection, etc..

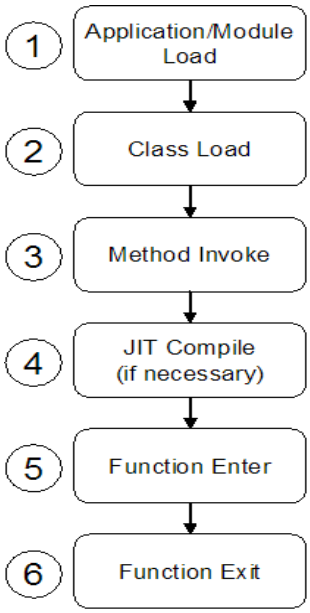

Figure 3.3: First method invocation in a managed application

To allow an adaptation engine to interact with a class instance we augment the type definition such that the necessary "hooks" can be added. Augmenting the type definition is a two-phase operation. The first phase occurs at module load time, Stage 1 in Figure 3.3.

When the loader loads a module, the bytecode for the method bodies of the module's types is laid out in memory. The starting address of the first bytecode instruction in a method body is referred to as the *Relative Virtual Address (RVA)* of the method. At the end of the module load Kheiron/CLR automatically adds (prepares) *shadow methods*, using IMetaDataEmit::DefineMethod, for each of the original public and/or private methods of the type. A shadow method shares all the properties (attributes, signature, implementation flags and RVA) of the original method except the name. By sharing (borrowing) the RVA of the original method, the shadow method points at the method body of the original method. Figure 3.4 shows an example of adding a shadow method, _SampleMethod, for an original method, SampleMethod. Extending the metadata of a type by adding methods must be done before the type definition is installed in the CLR. Once the type definition is installed its list of methods and members becomes read only; further requests to define new methods

or members are silently ignored even though the return value from the API call indicates success.

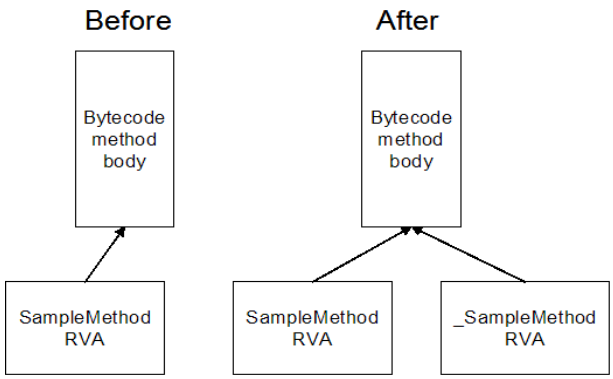

Figure 3.4: Preparing a shadow method

The second phase of type augmentation occurs the first time an original method is JIT compiled, Stage 4 in Figure 3.3. This phase converts the original method into a thin *wrapper* that simply calls the shadow method as shown in Figure 3.5. The heart of phase 2 allocates space for a new method body, uses the Byte-code & Metadata Transformer to generate the sequence of bytecode instructions to call the shadow, and sets the new RVA for the original method to point at the new method body.

There are a number of special considerations when creating shadows, especially in the case of non-void methods. The main issues revolve around ensuring the MaxStack and LocalVarSigTok properties in the method header of the wrapper are kept consistent with the newly defined method body with respect to the number of local variables and the maximum stack space needed to execute the instructions in the method body. Additionally, the new method body must contain any applicable instructions to push the arguments expected by the shadow method. Failure to get these details right results in a failed program verification and a subsequent crash of the CLR. The interested reader is directed to [110] for more details.

Using shadows and wrappers has a number of advantages. Given the structure of the wrapper method, see Figure 3.6, we can inject repair instructions as prologues and/or epilogues to shadow method calls.

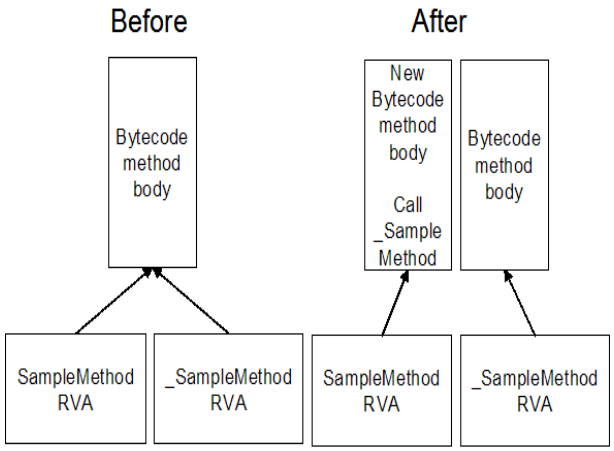

Figure 3.5: Creating a shadow method

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

Figure 3.6: Kheiron/CLR conceptual diagram of a wrapper

Adding a prologue to the wrapper requires that new bytecode instructions prefix the existing bytecode instructions. The level of difficulty is the same whether we augment the wrapper or the original method. Adding epilogues, however, presents a few more challenges. Intuitively, to add an epilogue, we wish to insert new instructions before control leaves a method. In the simple case, a method has a single return statement and the epilogue can be inserted right before that point. For methods with multiple return statements and/or exception handling routines, finding every possible return point can be an arduous task [137]. Further, the layout and packing of the bytecode for methods that contain exception handling routines is considered a special case which may be challenging to augment correctly [137].

Using wrappers presents a cleaner approach since we can ignore all of the complexity in the shadow method. Further, the regular structure and single return statement of the wrapper

method lends itself easily to adding an epilogue. Exceptions thrown, but not caught inside the shadow method, will cause the stack to be unwound to the wrapper where the exception can be caught – if an exception handler was included in the wrapper generation process – or passed up to the caller as would be the case in the original unmodified application.

3.7.5 Performing an Adaptation

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

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

Of the above four steps, steps 1 – 3 are relatively easy compared to step 4. The main concern when performing steps 1 through 3 is to ensure the assembly properties (name, version, path, culture info, etc.), type properties (type name and assembly reference) and member properties (method name, type reference, and method signature) are valid. The unmanaged APIs were designed to be fast and as a result sacrifice extensive semantic error checking [125]. Further, these APIs are intended to be used by tool developers and compiler writers

and as a consequence, it is the responsibility of the API user to get the details right as it relates to generating or editing metadata. Errors in these details can result in failed metadata verifications, failed assembly resolutions and halting of the CLR.

In step 4, adding a jump into the adaptation engine as a prologue is done by inserting as few as two (2) MSIL instructions[3], see Figure 3.7, before the existing MSIL instructions that comprise the current method body.

```
1: ldarg.0 //pass this pointer to the adaptation engine method
2: call <Metadata token of adaptation engine method>
```

Figure 3.7: Jump into adaptation engine

Adding a jump as an epilogue is slightly more complicated, despite the regular structure of the wrapper method. Class methods that have a return type other than void look like Figure 3.8 after we create a shadow for them.

```
1: ldarg.0 //push *this* before calling member method
2: call <Metadata token of shadow method>
3: stdloc.0 //store return value in first local slot
4: ldloc.0 //push the return value on the stack
5: ret //return
```

Figure 3.8: Before epilogue insertion

To add the epilogue, we need to keep track of where we inserted the last *call* instruction and whether it returns a value or not. If it returns a value we insert the instructions shown in Figure 3.7 between instructions 3 and 4 in Figure 3.8 and re-emit instructions 4 to 6. The final result is shown in Figure 3.9.

[3]This assumes that the method being called on the adaptation engine is a static method that takes an object as its sole argument, e.g., public static void RepairEngine::Repair(Object o). In the case of invoking non-static methods additional bytecode instructions to load the *this* pointer of an instance of the class containing the method to be invoked also need to be inserted

```
1: ldarg.0 //push *this* before calling member
method
2: call <Metadata token of shadow method>
3: stdloc.0 //store return value in first local slot
4: ldarg.0 //pass this pointer to the adaptation
engine method
5: call <Metadata token of adaptation engine
method>
6: ldloc.0 //push the return value on the stack
7: ret //return
```

Figure 3.9: After epilogue insertion

To persist the bytecode changes made to the method bodies of the wrappers, the Execution Monitor requests the CLR JIT compile the wrapper method again (referred to as a re-JIT). The actual re-JIT takes place the next time the wrapper method is called. In our Kheiron/CLR prototype re-JIT requests are submitted in the Function Exit event, Stage 6 in Figure 3.3.

Kheiron/CLR uses the ICorProfilerInfo::SetFunctionReJIT function to persist bytecode changes but it can also use it to undo the changes we make. We can temporarily disable shadows, reverting back to shadow prepare phase, Figure 3.4, and we can remove prologues and/or epilogues by setting the wrapper method RVA to the RVA of a method body without those prologues and/or epilogues and requesting a re-JIT. This facility allows us to manage the performance hit we take from making shadowed method calls and we can flexibly attach or detach the adaptation engine as desired by rewriting the bytecode in the wrapper method, removing the instrumentation or jumps into the repair engine and requesting a re-JIT of the modified wrapper.

The ability to perform multiple JIT compilations on demand is a powerful facility, allowing us to undo or redo any changes we make; however, some additional tweaking (see §3.7.6) is required to get function re-JITs to work as expected in our prototype.

3.7.6 Forcing Multiple JIT Compilations (re-JITs)

The CLR includes some infrastructure support for function re-JITs. To enable re-JITs the CLR the predefined constant COR_PRF_ENABLE_REJIT, found in corprof.h, must be used when informing the CLR of the kinds of notifications Kheiron/CLR wishes to receive. As shown in Figure 3.1, the CLR needs a way to determine whether a method body has already been JIT-compiled. To do this the CLR relies on a tripwire in the form of an indirect method call to a helper function known as *the prestub helper*. When a type is loaded, a structure known as a *MethodTable* is created for it. The method table will eventually contain pointers to the native assembly versions of the method bodies. However, initially each slot in the method table is loaded with a pointer to the prestub helper [181].

```
Function ID 0x00975338
Calculating the address of the
prestub by hand:

Before JIT Compilation
0x0097532A 00 00 f8 be de 02 04
0x00975331 00 fe e8 18 dc f7 ff
0x00975338 05 00 00 00 98 20 00
0x0097533F c0 05 00 fc e8 08 dc
0x00975346 f7 ff 06 00 00 00 b0

Function ID + Word(Function ID-4)

0x00975338 + 0xfff7dc18 = 008F2F50

Once we know where the prestub is
We can restore:

Byte Function ID-5
Word Function ID-4

by hand to force a re-JIT.
```

Figure 3.10: Locating the prestub and forcing a re-JIT by hand

The prestub helper does the work of compilation. After compilation the relevant slot in the MethodTable is updated with a pointer to the compiled version of the method body. Figure

3.11 illustrates what happens before and after a JIT compilation. Before the JIT compilation, execution jumps into the prestub helper, instruction *e8 near address* on X86. After JIT compilation this is replaced with an absolute jump, instruction *e9 address* on X86, where the jump target is the memory location of the compiled method body. The process used to force reJITs in our framework, is based on refinements and extensions to the process used in [56]. We calculate the address of the prestub helper in memory, as shown in Figure 3.10. The prestub address is used to calculate the offset for the near address jump for any function ID. Restoring the appropriate memory location causes the CLR to jump into the prestub helper the next time the function is called.

In the CLR v1.0 and v1.1 the changes we make by hand to force a re-JIT can be achieved using the ICorProfilerInfo::SetFunctionReJIT; however, this API function was inadvertently included in the CLR v1.x releases and has been subsequently removed from CLR v2.0 [127]. As a result Kheiron/CLR's ability to make changes and then undo them (via causing a re-JIT) is currently limited to CLR v1.x. Our by hand approach for causing a re-JIT (Figure 3.10) has not been tested in the CLR v2.0.

3.7.7 Evaluation Part 1: Kheiron/CLR Performance Impact

The first part of the evaluation of our Kheiron/CLR prototype focuses on quantifying the overheads on program execution using two separate benchmarks.

The experiments were run on a single Pentium III Mobile Processor, 1.2 GHz with 1 GB RAM. The platform was Windows XP SP2 running the .NET Framework v1.14322. Enabling Kheiron/CLR is done by setting four environment variables before starting the application (Figure 3.12) [4]. In our evaluation we used the C# benchmarks SciMark [5] and

[4] Information about the CLR Profiler must also be entered into the Windows registry via regsvr32 /s <fully qualified path to Kheiron/CLR dll>

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

```
Function ID 0x00975338

Before JIT Compilation
0x0097532A 00 00 f8 be de 02 04
0x00975331 00 fe e8 18 dc f7 ff
0x00975338 05 00 00 00 98 20 00
0x0097533F c0 05 00 fc e8 08 dc
0x00975346 f7 ff 06 00 00 00 b0
0x0097534D 20 00 c0 00 00 08 00
0x00975354 0c 00 00 00 08 34 e2
0x0097535B 02 00 00 00 00 00 00

After JIT Compilation
0x0097532A 00 00 f8 be de 02 04
0x00975331 00 fe e9 a0 70 47 02
0x00975338 05 00 00 00 d8 c3 de
0x0097533F 02 05 00 fc e9 e0 88
0x00975346 47 02 06 00 00 00 28
0x0097534D dc de 02 00 00 08 00
0x00975354 0c 00 00 00 08 34 e2
0x0097535B 02 00 00 00 00 00 00

Distinguished memory addresses:
Byte Function ID-5
Word Function ID-4
Word Function ID+4 restored by
SetFunctionReJIT
Byte Function ID+11
Word Function ID+12
```

Figure 3.11: JIT compilation overview

Linpack [6].

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

Linpack is a benchmark that uses routines for solving common problems in numerical linear algebra including linear systems of equations, eigenvalues and eigenvectors, linear

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

```
set DBG_PRF_LOG=0x1
set Cor_Enable_Profiling=0x1
set COR_PROFILER_DLL=<path to Kheiron/CLR dll>
set COR_PROFILER=<Identifier string Kheiron/CLR>
```

Figure 3.12: Enabling Kheiron/CLR

least squares and singular value decomposition. In our tests we used a problem size of 1000.

Overheads. Kheiron/CLR consists of a profiler that uses the Profiler API [126] to intercept module load, unload and module attached to assembly events, JIT compilation events and function entry and exit events. As expected, running an application in the profiler imposes some overhead on the application. Figure 3.13 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 of the benchmark running without profiling enabled. The bar on the right shows the normalized performance with our profiler (Kheiron/CLR) enabled.

Our measurements show that Kheiron/CLR contributes ~5% runtime overhead when no repairs are active, which we consider negligible.

SCIMark	Composite Score					**Average**	**Stdev**
Without Kheiron/CLR	187.71	189.15	189.28	189.08	189.56	**188.956**	**0.720**
With Kheiron/CLR	181.52	181.95	182.19	182.43	182.64	**182.146**	**0.435**
% Slowdown	3.30%	3.81%	3.75%	3.52%	3.65%	**3.60%**	**0.20%**

Table 3.1: Kheiron/CLR overheads on SCIMark when no repair active

Linpack	Composite Score					**Average**	**Stdev**
Without Kheiron/CLR	60.1	60.045	61.54	61.089	61.37	**60.829**	**0.709**
With Kheiron/CLR	57.91	57.511	58.112	58.366	57.91	**57.962**	**0.314**
% Slowdown	3.64%	4.22%	5.57%	4.46%	5.64%	**4.71%**	**0.87%**

Table 3.2: Kheiron/CLR overheads on Linpack when no repair active

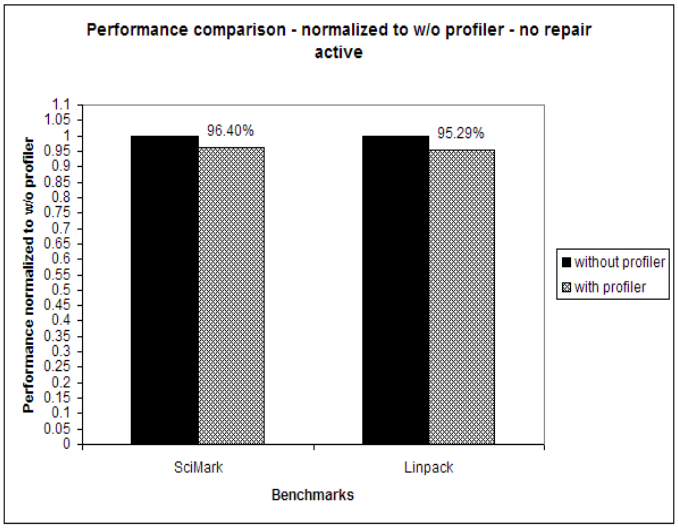

Figure 3.13: Kheiron/CLR overheads when no repair active

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

To quantify these overheads, we use the SciMark2.SOR class, which executes the Jacobi Successive Over-relaxation benchmark. Table 3.3 shows the impact on module bind time due to preparing shadows on the two public methods of SciMark2.SOR, SciMark2.SOR::execute and SciMark2.SOR::num_flops.

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

Creating shadows imposes a one time overhead incurred the first time the method is JIT

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

Table 3.3: Kheiron/CLR overheads of preparing shadows

compiled. As shown in Table 3.4 the time for the first JIT compilation is dominated by the time spent creating the shadow[7].

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

Table 3.4: Kheiron/CLR overheads of creating shadows

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

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

Table 3.5 compares the total method time for the SciMark2.SOR::execute wrapper method, with the total method time for its shadow method. In this case the disparity in method times is $\ll 1\%$ and the overall impact on the performance of the benchmark is negligible.

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

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

	Wrapper Method SOR::execute	Shadow Method SOR::_execute
Function ID	0x935ae8	0x935b18
Enter/Leave count	15	15
JIT Count	15	1
# shadows created	1	0
Create shadow (ms)	11.1834	n/a
Total Invoke time (ms)	**6273.27**	**6272.31**
Total JIT time (ms)	**2.9621**	**0.90244**
Total method time (ms)	**6287.4156**	**6273.21244**

Table 3.5: Execution overheads on SciMark2.SOR::execute

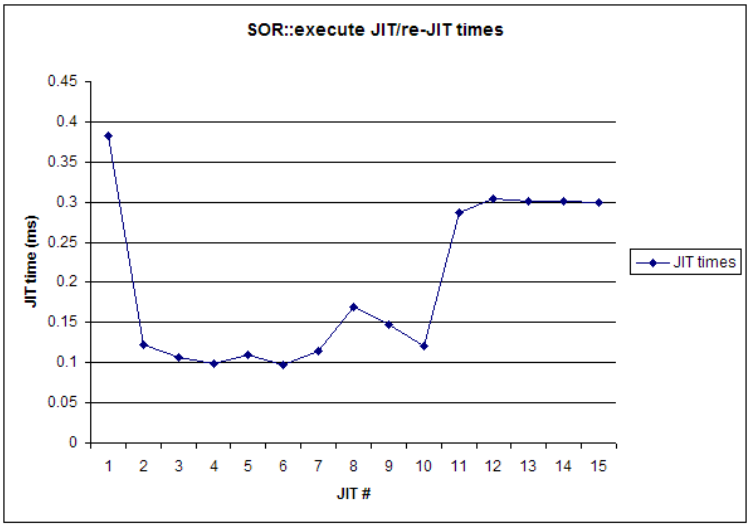

Figure 3.14: CLR re-JIT measurements for SciMark2.SOR::execute wrapper

Based on our experiments we are able to identify and measure three sources of overhead Kheiron/CLR imposes on a target system's operation – load-time overhead, JIT-compilation time overhead and runtime overhead. In the case of the load-time and JIT-compilation overheads, the impact is small (sub-second and in some cases sub-millisecond) and for runtime overhead, Kheiron/CLR's impact on the target system is negligible ~5% when no repairs or reconfigurations are active. We are also able to demonstrate how Kheiron/CLR can interact with the Common Language Runtime and the applications it hosts in a manner that is transparent to both the runtime and the target application.

JIT #	JIT Time (ms)
1	0.3829
2	0.121878
3	0.106419
4	0.098367
5	0.109359
6	0.097914
7	0.114577
8	0.169962
9	0.147907
10	0.120658
11	0.286591
12	0.304285
13	0.300659
14	0.300795
15	0.299871
Total JIT time (ms)	**2.962142**
Average JIT time (ms)	**0.197476**
Stdev (ms)	**0.101077**

Table 3.6: CLR re-JIT measurements for SciMark2.SOR::execute wrapper

3.7.8 Evaluation Part 2: Kheiron/CLR Dynamic Reconfiguration Case Study

The second part of the evaluation of Kheiron/CLR looks at its ability to effect repairs and reconfigurations in a non-trivial system while preserving the integrity of the target system and the CLR.

To evaluate our Kheiron/CLR prototype beyond small/toy examples, we searched on SourceForge.NET [174] for potential target systems already implemented on the CLR that might benefit from runtime adaptation. We report on our experience using Kheiron/CLR to facilitate runtime reconfigurations in a system that was developed (and is in use) by others: the Alchemi Enterprise Grid Computing System developed at the University of Melbourne, Australia [188].

We selected the Alchemi Enterprise Grid Computing System [6], from the University of Melbourne, Australia. Alchemi has several appealing characteristics relevant for our case

study purposes: It was developed and is currently maintained by others, whom we do not know and have not contacted, hence we regard it as a legacy system upon which runtime adaptations can be carried out only via an externalized engine. It is publicly available on SourceForge [175], which makes it possible for other autonomic computing researchers to "repeat" our experiment employing their own technology for comparison purposes. Alchemi is also well-documented, which makes it feasible to construct plausible scenarios, where performing runtime reconfigurations and/or repairs on the system could result in real benefits for its real-world users.

Alchemi is apparently being used in a number of scientific and commercial grid applications, including an application for distributed, parallel environmental simulations at Commonwealth Scientific and Industrial Research Organisation (CSIRO) Land and Water, Australia, and a micro-array data processing application for early detection of breast cancer developed by Satyam Computers Applied Research Laboratory in India.[8] Finally, Alchemi is implemented as a .NET application on top of the CLR, which is a prerequisite for Kheiron/CLR. Alchemi is written in C#, and leverages a number of technologies provided by the .NET Framework, including .NET Remoting [86], multi-threading and asynchronous programming.

Alchemi Architecture. The Alchemi Grid follows a master-worker parallel programming paradigm, where a central component (the Manager) dispatches independent units of parallel execution (grid threads) to be executed on grid nodes (Executors), see Figure 3.15. The Manager is responsible for providing the services associated with the execution of grid applications and their constituent grid threads. It monitors the status of the Executors registered with it, and schedules grid threads to run on them. Executors accept grid threads from the Manager, execute them, and return the completed threads to the Manager. An Executor can be configured as either *dedicated*, i.e., managed centrally where the Manager

[8] A list of projects using Alchemi can be found at http://www.alchemi.net/projects.html.

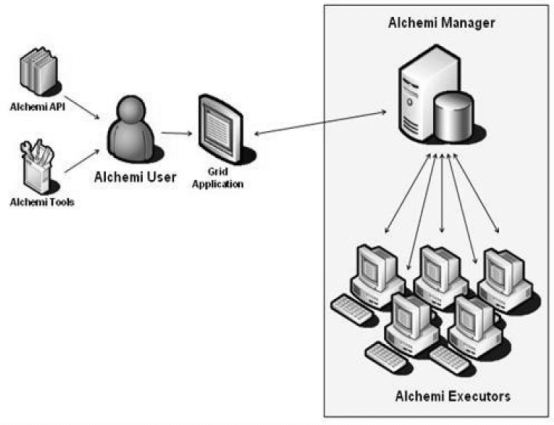

Figure 3.15: Alchemi architecture – source: User Guide for Alchemi 1.0 [5]

"pushes" a computation to an idle, dedicated Executor whenever its scheduling requires, or *non-dedicated*, where the Executor instead polls the Manager and hence "pulls" some computational work only during idle periods, e.g., when a screen saver is active.

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

We do not address whether a one-size-fits-all scheduling algorithm could be implemented

to take into account all operating conditions and all kinds of submitted application mixes, but instead intend to enable the Alchemi Manager to switch dynamically among different scheduling algorithms, each potentially tuned for specific conditions and workloads, as the state of the system changes. The same scheduler-swapping provisions could also be used to avert or alleviate situations in which (a subset of) Executors misbehave – for reasons varying from misconfiguration, to the occasional bug in the code of grid threads for some applications, to malicious interference by rogue Executor nodes – in ways that cannot be immediately detected by the monitoring capabilities of the Manager. By default, the Manager only tracks whether an executor node is "alive" using periodic heatbeats.

In the next section we describe a proof-of-concept experimental case study that demonstrates how Kheiron/CLR can be used to facilitate runtime reconfiguration, specifically replacement of the Alchemi scheduler, without any modifications to the source code of the target system or the underlying CLR managed execution environment. We show how our adaptation engine attached via Kheiron/CLR is able to transparently swap scheduler implementations on the fly, which would enable existing Alchemi installations to take advantage of multiple alternative scheduling algorithms without having to re-compile and re-install any system components. We also discuss how the reconfigurations are carried out in a way that preserves the consistency of the running grid application, as well as the overall distributed grid system.

We should stress that our case study focuses on the feasibility of effecting such *consistency-preserving* reconfigurations of a legacy software system like Alchemi running in a managed execution environment. We do not at all address the optimization issues implied by the concept of dynamic scheduler replacement. We claim only that Kheiron/CLR facilitates the development of specific remedies such as optimization: for instance, our approach could enable an adaptive scheduler-swapping scheme that could ensure the grid's performance across a vast range of applications and conditions, which remains an open and interesting research issue.

Reconfiguring Alchemi. To swap the grid scheduler in a running instance of the Alchemi grid, we need to implement the reconfiguration engine that interacts with Alchemi's Manager component. Using Kheiron/CLR, our CLR profiler described in Section 3.7.3, we can dynamically attach/detach such an adaptation engine implemented as a separate assembly to/from a running managed application in a fairly mechanical way. However, a first important step is to carefully plan the interactions between the running application, the reconfiguration engine and the CLR, in such a way that they do not compromise the integrity of either the managed application or the CLR.

Consequently, we – as the developers of the adaptation engine to be attached by Kheiron/CLR – must gather some knowledge about the system. Specifically, we need details about how the Alchemi Manager component works, particularly the execution flow in the Manager from startup to shutdown. That enables us to identify potential "safe" control points where reconfiguration actions can take place. We also need to identify those classes the adaptation engine must interact with to effect the scheduler swap. The final step is to implement the special-purpose reconfiguration engine based on what we learn about the system.

In particular, we learned that when the Alchemi Manager is started (by running the **Alchemi.Manager.exe** assembly) an instance of the *ManagerContainer* class, from the **Alchemi.Core.dll** assembly), is created. The instance of the ManagerContainer class represents the Manager proper. On startup, the **ManagerContainer::Start()** routine performs a set of initialization tasks:

1. An object is registered with the .NET Remoting services, allowing Executors to interact with the Manager instance.

2. A singleton instance of the *InternalShared* class is created, holding a reference to the scheduler implementation being used (among other things). The concrete scheduler implementation is referenced as an implementation of the **Alchemi.Core.Manager.IScheduler** interface, which standardizes the scheduler API [6].

3. Two threads, the scheduler thread and the watchdog thread, are started. The scheduler thread runs the **ManagerContainer::ScheduleDedicated()** method, which loops "forever" on a flag member variable, **_stopScheduler**. It periodically retrieves the scheduler implementation from the InternalShared singleton instance and queries it for a *DedicatedSchedule*. A DedicatedSchedule is a <Grid Thread ID, Executor ID> tuple specifying where the selected grid thread should be scheduled to run. The watchdog thread runs the **ManagerContainer::Watchdog()** method, which loops "forever" on the **_stopWatchdog** flag member variable, periodically checking the status of dedicated Executors.

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

1. Use Kheiron/CLR to insert a prologue into the **ManagerContainer::Start()** method such that it jumps into the reconfiguration engine assembly where the instance of the ManagerContainer can be cached so we can interact with it later to effect the scheduler swap.

2. Use Kheiron/CLR to insert a prologue into the constructor for the InternalShared class such that it jumps into the reconfiguration engine assembly where the instance can be cached.

3. Once instances of the ManagerContainer and InternalShared classes have been cached, the reconfiguration engine can cause the scheduler thread to exit normally by setting the **_stopScheduler** flag to true, allowing the thread to exit when it next tests the while loop condition.

4. The **Alchemi.Core.Manager.IScheduler** reference stored in the InternalShared singleton can then be replaced by another IScheduler implementation.

5. The **_stopScheduler** flag is set to false and the scheduler thread is restarted.

The Reconfiguration Engine and Replacement Scheduler. Our adaptation engine implementation, found in the **PSL.Alchemi.ReconfigEngine.dll** assembly, consists of two C# classes, *PSLScheduler* and *ReconfigEngine*. The implementation was done without contacting the Alchemi developers and took about half a day to complete. The total implementation is 465 LOC – 95 LOC for *PSLScheduler.cs* and 370 LOC for *ReconfigEngine.cs*.

PSLScheduler implements the **Alchemi.Core.Manager.IScheduler** interface, and is functionally equivalent to the DefaultScheduler implementation that ships with Alchemi, except for some extra debugging and logging facilities. As noted previously, the goal of PSLScheduler is solely to demonstrate a successful reconfiguration – the scheduler swap – and to exemplify how Kheiron facilitates the development of such a reconfiguration, not to actually improve scheduling.

ReconfigEngine is responsible for caching instances of the Manager classes of interest, ManagerContainer and InternalShared, as well as effecting the scheduler swap. It is implemented according to the singleton design pattern. To effect changes on the ManagerContainer and InternalShared instances, the ReconfigEngine relies on the *Reflection API*, since many of the key variables are private and in some cases read-only. The ReconfigEngine sets up a communication channel after it has attached to the Manager, which allows a Reconfiguration Console to send commands to the ReconfigEngine to trigger reconfigurations (our case study did NOT include sensor monitoring for those conditions under which a different scheduler would be warranted). Table 3.7 shows the method signatures of the ReconfigEngine API.

Method
public static ReconfigEngine GetInstance()
public static void CacheManagerContainer(object o)
public static void CacheInternalShared(object o)
public void SwapScheduler()

Table 3.7: Reconfiguration engine API

Experimental Setup. 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.

We swapped between the DefaultScheduler and the PSLScheduler. The two schedulers are algorithmically equivalent, except that the PSLScheduler outputs extra logging information to the Alchemi Manager GUI so that we could confirm that a scheduler swap actually occurred.

Results. One thing we measured was the time taken to swap the scheduler. We requested scheduler swaps between runs of the 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 3.8 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, ~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 3.8: PiCalculator.exe job completion times

Thus we were able to demonstrate that Kheiron/CLR 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/CLR to insert hooks for the adaptation engine respect the CLR's verification rules for type and method definitions and the inclusion of human analysis to determine what transformations Kheiron/CLR should perform on the target system, and when they should be performed, can help guarantee that the operation of the target system is not compromised. Human analysis of the target system's operation leverages the consistency-guarantees of Kheiron/CLR 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.

3.8 Kheiron/JVM: Runtime Adaptation in the Java Virtual Machine

The Java Virtual Machine (JVM) is the technology component responsible for the hardware and operating system independence of Java applications [111]. It is an abstract computing machine, with its own instruction set and binary format for executables (the *class* file format). These two elements – the abstract computing machine and the class file binary format for executables – allow Java applications to be written and compiled once (into class files) and run on multiple operating system and hardware platforms provided that an implementation of the Java Virtual Machine exists for that operating system/hardware platform [9].

Java applications are compiled into executables (*classfiles*), which contain JVM instructions (*bytecodes*), a symbol table (*constant pool*) and other ancillary information.

Despite the JVM's primary association with the Java programming language, it is not tightly tied to the Java. In fact the JVM "...knows nothing of the Java programming language, only of a particular binary format, the *class* file format" [111]. As a result any language with functionality that can be expressed in a valid class file can be hosted in the JVM.

The JVM, like the CLR, is an example of a managed execution environment In addition to providing host and operating system independence, it also provides a number of services to the applications it hosts including: application isolation, garbage collection of memory, security sandboxing of applications and structured exception handling.

Despite the conceptual similarities between the CLR and JVM with respect to the abstractions and services they provide to the applications they host, there are many differences between them. In this thesis we only highlight the differences related to effecting adaptations in running Java applications.

[9] This is/was one theoretical goal for Java and the JVM; however, multiple JVM implementations running on top of different operating systems e.g. Windows, Linux, Solaris, etc. led to subtle differences in how a program executes across platforms [194].

3.8.1 Java Virtual Machine Execution Model (Java HotspotVM)

The unit of execution (sometimes referred to as a module) in the JVM is the *classfile*. Classfiles contain both the metadata and bytecode of a Java application. Two major components of the JVM interact with the metadata and bytecode contained in the classfile during execution, the *classloader* and the *global native-code optimizer*.

The classloader reads the classfile metadata and creates an in-memory representation and layout of the various classes, members and methods on demand as each class is referenced. The global native-code optimizer uses the results of the classloader and compiles the bytecode for a method into native assembly for the target platform.

The JVM first runs the program using an interpreter, while analyzing the code to detect the critical hot spots in the program. Based on the statistics it gathers, it then focuses the attention of the global native-code optimizer on the hotspots to perform optimizations including JIT-compilation and method inlining [131]. This model of execution of the JVM was introduced in v1.4, and is the reason why these and later VM versions/implementations are referred to as Hotspot VMs. Compiled methods remain cached in memory, and subsequent method calls jump directly into the native (compiled) version of the method.

3.8.2 JVM Profiler and Metadata APIs

The v1.5 implementation of the Java HotspotVM introduces a new API for inspecting and controlling the execution of Java applications – the Java Virtual Machine Tool Interface (JVMTI) [134]. JVMTI replaces both the Java Virtual Machine Profiler Interface (JVMPI) and the Java Virtual Machine Debug Interface (JVMDI) available in older releases. The JVMTI is a two-way interface: clients of the JVMTI, often called *agents*, can receive notifications of execution events in addition to being able to query and control the application via functions either in response to events or independent of events. JVMTI notification

events include (but are not limited to): classfile loading, class loading, method entry/exit.

The Java HotspotVM does not have a built in API for manipulating type definitions. As a result, to perform operations such as reading class and method attributes, parsing method descriptors, defining new methods for types, emitting/rewriting the bytecode for method implementations and creating new type references the first version of Kheiron/JVM relied on APIs we developed based on information provided in the Java Virtual Machine Specification (Chapter 4) [111]. Later versions of Kheiron/JVM use the Byte Code Engineering Library (BCEL) [149], which provides a set of abstractions for manipulating metadata in classfiles and facilities for verifying the validity of the modified classfile(s).

3.8.3 Kheiron/JVM Architecture

The implementation of Kheiron/JVM consists of a JVMTI agent (1890 lines of C++ code) and a set of supporting Java classes that modify classfiles, collect execution statistics and communicate with the JVMTI agent (779 lines of Java code) using the Java Native Interface (JNI). To deploy Kheiron/JVM, the C++ code is packaged in an application extension library, (.dll on Windows and .so on Linux), while the Java code is packaged in a jar file. Figure 3.16 shows the five main components of Kheiron/JVM.

- The **Kheiron JVMTI Agent** receives execution-related events from the JVM. Specifically, our agent subscribes to class hook events (e.g., classfile load events for every class), garbage collection events (e.g., GC start and end events), compiled method load events (method compiled, loaded and unloaded events) and exception events (exception thrown and caught events). Our JVMTI agent does not subscribe to method entry/exit events from the JVM since these can severely degrade application performance [134]. Instead, we rely on the Bytecode Transformer to add method entry and exit profiling hooks to the methods of the classes we are interested in.

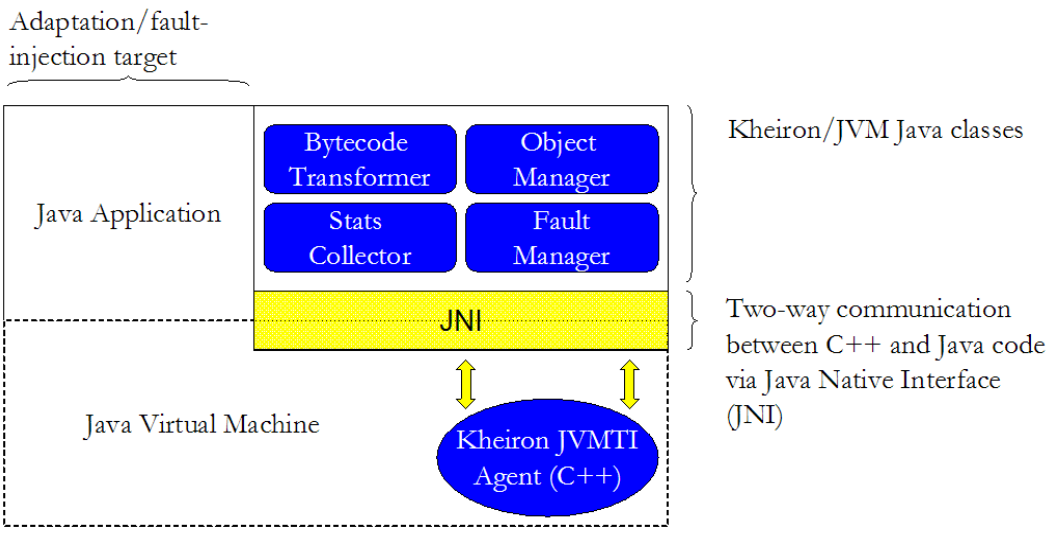

Figure 3.16: Kheiron/JVM architecture diagram

- The **Bytecode Transformer** parses and modifies classfiles. It is able to add new methods or variables to types and add references to other classes or methods. It is also responsible for generating, inserting or replacing the bytecode in existing methods. Bytecode changes can be committed at loadtime (via returning a modified classfile in response to the ClassfileLoadHook event generated by the JVM when a classfile is read from storage) or at runtime via the RedefineClasses function exposed by the JVMTI. In the case of runtime modifications of methods, active method invocations continue to use the old implementation of a method while new invocations use the latest version [134][10]. In our Kheiron/JVM implementation the Bytecode Transformer is primarily responsible for injecting instrumentation hooks into classes such that method invocations and object creations can be tracked. The hooks inserted interact with methods exposed by the Stats Collector and/or the Fault Manager depending on the desired adaptation.

- The **Stats Collector** captures object creation and method entry and exit events via the hooks inserted by the Bytecode Transformer. The Stats Collector uses a number of different book-keeping data structures to manage and collate the events received while

[10] [134] also outlines the restrictions on the modifications permitted using the RedefineClasses API.

an application is executing.

- The **Object Manager** manages an object pool of book-keeping data structures. It is responsible for creating, distributing and recycling these book-keeping data-structure instances in an effort to limit the amount of memory consumed by Kheiron related objects.

- The **Fault Manager** is responsible for injecting faults or inducing failures in a Java application. It relies on hooks inserted by the Bytecode Transformer to capture and/or interact with elements (object instances, data structures, methods implementations, etc.) in the target application.

Communication between the Kheiron JVMTI agent written in C++ and the Kheiron/JVM Java classes is achieved using the Java Native Interface (JNI). JNI is a two-way interface that allows Java code running in a JVM to call and by called by code written in other languages, e.g., C and C++ [132]. Whereas there are other approaches to facilitating communication between Java and non-Java applications, e.g., TCP/IP sockets, interprocess communication mechanisms (IPC), etc., JNI allows communication between Java and non-Java elements that share the same process space as is the case with our JVMTI C++ agent and Kheiron/JVM Java classes, which are hosted in a single JVM process.

The ability of the profiler/JVMTI agent to call or interact with managed (Java) code in a structured way via the JNI APIs is unique to the JVM. This allows JVMTI agents to leverage functionality available in the Java system libraries and/or other Java based libraries. In the CLR, profilers are intended to be purely unmanaged code, i.e., written in C/C++. Profiler developers are warned that "...attempts to combine managed and unmanaged code from a CLR profiler can cause crashes, hangs and deadlocks" [126].

3.8.4 Model of Operation

Kheiron/JVM performs operations on type definitions, object instances and methods at various stages in the execution cycle (see Figure 3.17) to make them capable of interacting with an adaptation engine. In particular, to enable an adaptation engine to interact with a class instance, Kheiron/JVM augments the type definition to add the necessary "hooks". Augmenting the type definition is a two-step operation.

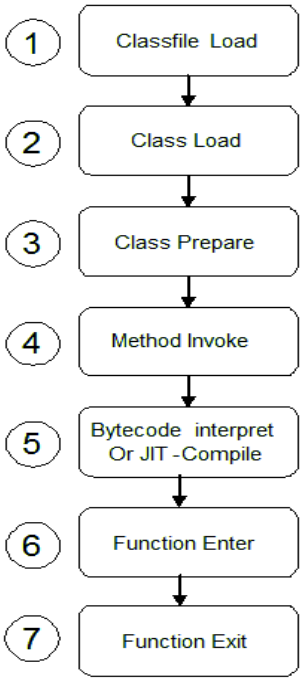

Figure 3.17: First method invocation in the Java HotspotVM

Step 1 occurs at classfile load time (Stage 1 in Figure 3.17), signaled by the **ClassFileLoad-Hook** JVMTI callback that precedes it. At this point the VM has obtained the classfile data from storage but has not yet constructed the in-memory representation of the class. Kheiron/JVM adds 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 attributes, e.g., exception specifications and the method descriptor – of the corresponding original method. However, a shadow method gets a unique name. Figure 3.18, transition A to B, shows an example of adding a shadow method **_SampleMethod** for the original

method **SampleMethod**.

Extending the metadata of a type by adding new methods must be done before the type definition is installed in the JVM. Once a type definition is installed, the JVM will reject the addition or removal of methods. Attempts to call RedefineClasses will fail if new methods or fields are added. Similarly, changing method signatures, method modifiers or inheritance relationships is also not allowed.

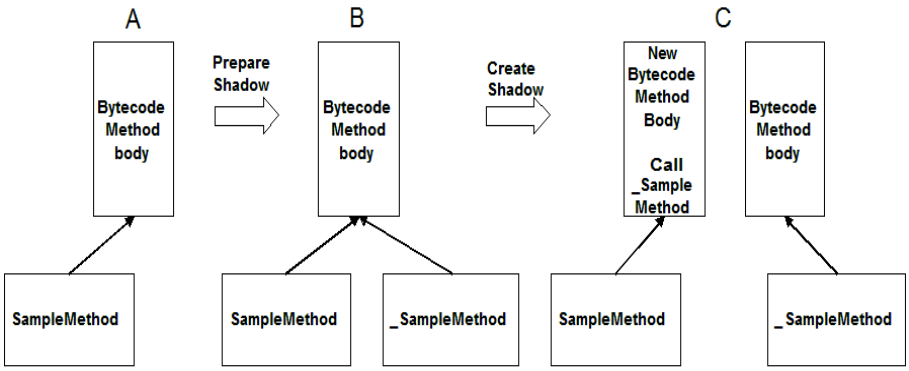

Figure 3.18: Preparing and creating a shadow method

Step 2 of type augmentation occurs immediately after the shadow method has been added, while still in the ClassFileLoadHook JVMTI callback. Kheiron/JVM uses bytecode-rewriting techniques to convert the implementation of the original method into a thin *wrapper* that calls the shadow method, as shown in Figure 3.18, transition B to C.

Kheiron/JVM's 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 3.19, enables 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 this insertion in the wrapper or in the original method. Adding epilogues, however, is more challenging (as highlighted in 3.7.4). To address these challenges, we employ the same wrapper-based

```
SampleMethod( args ) [throws NullPointerException]
  <room for prolog>
  push args
  call _SampleMethod( args ) [throws NullPointerException]
  { try{...} catch (IOException ioe){...} } // Source view of _SampleMethod's body
  <room for epilog>
  return value/void
```

Figure 3.19: Kheiron/JVM conceptual diagram of a wrapper

approach, which allows us to create a regular method structure with a single entry point, and a single *known* exit point. The simplified structure of the wrapper makes it easy to add/edit prologues and epilogues as necessary.

To initiate an adaptation, Kheiron/JVM augments the wrapper to insert a jump into an adaptation engine at the *control point(s)* before and/or after a shadow method call. This allows an adaptation engine to be able to take control before and/or after a method executes. Effecting the jump into the adaptation engine is a two-step process.

- **Step 1**: Extend the metadata of the classfile currently executing in the JVM such that a reference to the classfile containing the adaptation engine is added to the *constant pool*[11] as well as references to the subset of the adaptation engine's methods that we wish to insert calls to.

- **Step 2**: Augment the bytecode and metadata of the wrapper function to insert bytecode instructions to transfer control to the adaptation engine before and/or after the existing bytecode that calls the shadow method. The adaptation engine can then perform any number of operations, such as inserting and removing instrumentation, caching class instances, performing consistency checks over class instances and components, injecting faults, or performing reconfigurations and diagnostics of components.

[11]The constant pool stores symbolic information about fields, methods, interfaces, constants, data types, etc. that is referenced by bytecode instructions.

3.8.5 Evaluation Part 1: Kheiron/JVM Performance Impact

We are able to show, that like our other framework for facilitating adaptations in a managed execution environment, Kheiron/CLR, Kheiron/JVM imposes only a modest performance impact on a target system when no adaptations, repairs or reconfigurations are active. We have evaluated the performance of our prototype by quantifying the overheads on program execution using two separate benchmarks.

The experiments were run on a single Pentium III Mobile Processor, 1.2 GHz with 1 GB RAM. The platform was Windows XP SP2 running the Java HotspotVM v1.5 update 4. Enabling Kheiron/JVM is done by adding a switch to the commandline that starts the JVM (Figure 3.20). In our evaluation we used the Java benchmarks SciMark v2.0[12] and Linpack[13].

```
java -cp .;kheiron.jar -agentpath:<path to
Kheiron/JVM dll/.so> <main class>
```

Figure 3.20: Enabling Kheiron/JVM

SciMark is a benchmark for scientific and numerical computing. It includes five computation kernels: Fast Fourier Transform (FFT), Jacobi Successive Over-relaxation (SOR), Monte Carlo integration (Monte Carlo), Sparse matrix multiply (Sparse MatMult) and dense LU matrix factorization (LU). **Linpack** is a benchmark that uses routines for solving common problems in numerical linear algebra including linear systems of equations, eigenvalues and eigenvectors, linear least squares and singular value decomposition. In our tests we used a problem size of 1000.

SCIMark	Composite Score					**Average**	**Stdev**
Without Kheiron/JVM	115.15	115.83	116.10	116.01	116.57	**115.934**	**0.515**
With Kheiron/JVM	113.61	111.50	114.89	116.00	115.54	**114.308**	**1.807**
% Slowdown	1.34%	3.74%	1.04%	0.02%	0.89%	**1.40%**	**1.39%**

Table 3.9: Kheiron/JVM overheads on SCIMark when no repair active

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

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

Linpack	Composite Score					**Average**	**Stdev**
Without Kheiron/JVM	55.00	58.06	57.81	58.42	58.36	**57.531**	**1.434**
With Kheiron/JVM	54.33	57.47	56.30	57.66	57.96	**56.744**	**1.488**
% Slowdown	1.22%	1.02%	2.62%	1.30%	0.69%	**1.369%**	**0.738%**

Table 3.10: Kheiron/JVM overheads on Linpack when no repair active

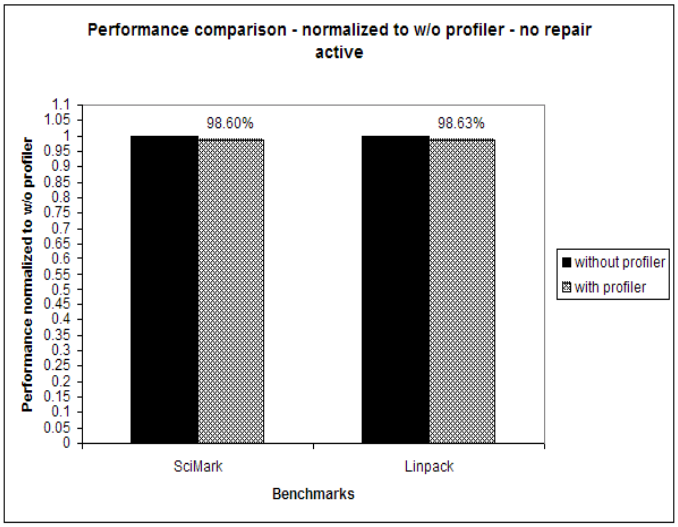

Figure 3.21: Kheiron/JVM overheads when no repair active

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 3.21 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 our profiler contributes ~2% runtime overhead when no adaptations are active, which we consider negligible. Further, Kheiron-related objects occupy ~155K of memory on the JVM heap, 120K of which is pre-allocated for the ObjectManager's pool of book-keeping data structures. Finally, we demonstrate how Kheiron/JVM can interact

with the Java Virtual Machine and effect adaptations in the applications it hosts in a manner that is transparent to both the JVM and the target application requiring only a change to the commandline used to start the application.

By implementing 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 edit and emit APIs that respect the verification rules for types, their metadata and their method implementations (bytecode) is a sufficiently low-overhead approach for adapting running programs in contemporary managed execution environments.

3.8.6 Evaluation Part 2: Kheiron/JVM Web-Application Fault-Injection

The second part of the evaluation of Kheiron/JVM looks at its ability to dynamically instrument, inject faults and induce failures in a Java-based application server and its hosted web-application classes.

For this case study we use an n-tier web application stack consisting of the TPC-W web-application [119][14], the (Java-based) Resin web and application server [186] and a MySQL database server as our system under test (SUT). We target the Java-based components of the stack, i.e. the web/application server and the TPC-W servlet classes for fault-injection.

We add to Kheiron/JVM the ability to inject 18 different faults into the web-application stack components. Wrapper methods generated by Kheiron/JVM include a call into the Fault Manager, which looks up the name of the method being invoked and performs a specific fault-injection action, e.g., allocating a block of memory or throwing a specific exception as necessary. Methodname-fault mappings are stored in a fault-model specification file that is read by Kheiron/JVM upon initialization.

[14] TPC-W is a transactional web e-Commerce benchmark, modeled after an online bookstore.

These 18 faults make up our fault-model and are are grouped into four failure categories – resource depletion failures, processing failures, configuration failures and JVM failures – (see Table 3.11. Our fault-model for N-tier web-applications is motivated primarily by the discussion in [21] and [142], which identify resource leaks/state corruption, intermittent/transient processing faults, hangs, configuration errors and crashes as major causes of failures in internet services.

Failure category	Fault
Resource depletion failures	Memory Leak Out of Memory Error
Processing failures	Delays Hangs Servlet Exception IO Exceptions NullPointer Exception Index Out of Bounds Exception Arithmetic Exception
Configuration failures	Class Cast Exception Illegal Argument Exception Missing Resource Exception Security Exception No Class Definition Found Exception Unsatisfied Link Error (JNI) Type Not Present Exception
JVM failures	Internal Error Stack Overflow Error

Table 3.11: Kheiron/JVM web-application stack fault-model

In our evaluation experiments we use an implementation of the TPC-W benchmark (web-application and client-load generator) developed by the Predictive High-Performance Architecture Research Mavens (PHARM) group at the University of Wisconsin-Madison. Resin 3.0.22 was chosen as the web/application server and MySQL 5.0.27 was used for the database servers. Resin and MySQL were installed on a single Windows XP Media Center Edition Version 2002 SP2 machine (2 GB RAM, 228 GB HD and an Intel®Core™2 Duo CPU E6750 @ 2.66 GHz Processor) running a v1.5.0 update 7 Java Virtual Machine. These evaluations were conducted using an optimized release build of the Kheiron/JVM Windows

DLL.

We use the injection of resource depletion faults, processing faults and configuration faults to demonstrate Kheiron/JVM's ability to induce failures in the TPC-W web-application classes. Fault-injection using Kheiron/JVM is guided by a fault-specification configuration file which contains the name of the fault to inject, the fully-qualified name of the method to target for fault-injection and the frequency of injection (e.g. once every N method invocations or once every N minutes).

Servlet Instrumentation. The TPC-W distribution consists of 14 servlet classes, which together expose 15 servlet methods for remote clients to interact with. To identify potential targets for fault-injection we first collect profiles of servlet execution over five 25-minute intervals using the TPC-W load generator configured to generate traffic for 20 clients. Enabling application-server instrumentation was done using the commandline shown in Figure 3.22. The configuration file *watch.config* contains the names of the servlets to instrument as well as a pointer to a fault-specification file, which uses a *NullFault* (not to be confused with the NullPointer Fault mentioned in Table 3.11) to collect invocation statistics on the 15 servlet methods.

```
httpd.exe -verbose -classpath .;kheiron.jar
-J-agentpath:kheiron_jvm.1.5.dll=watch.config
```

Figure 3.22: Enabling application-server instrumentation with Kheiron/JVM

The execution profile for the TPC-W servlets is shown in Figure 3.23. From this graph we can identify five potential fault-injection targets: TPCW_search_request_servlet.doGet, TPCW_home_interaction.doGet, TPCW_execute_search.doGet, TPCW_product_detail_servlet and TPCW_shopping_cart_interaction.doGet.

Resource Depletion Failures. For our resource depletion tests we target the TPCW_execute_search servlet with an aggressive memory leak, 500K every 5 invocations,

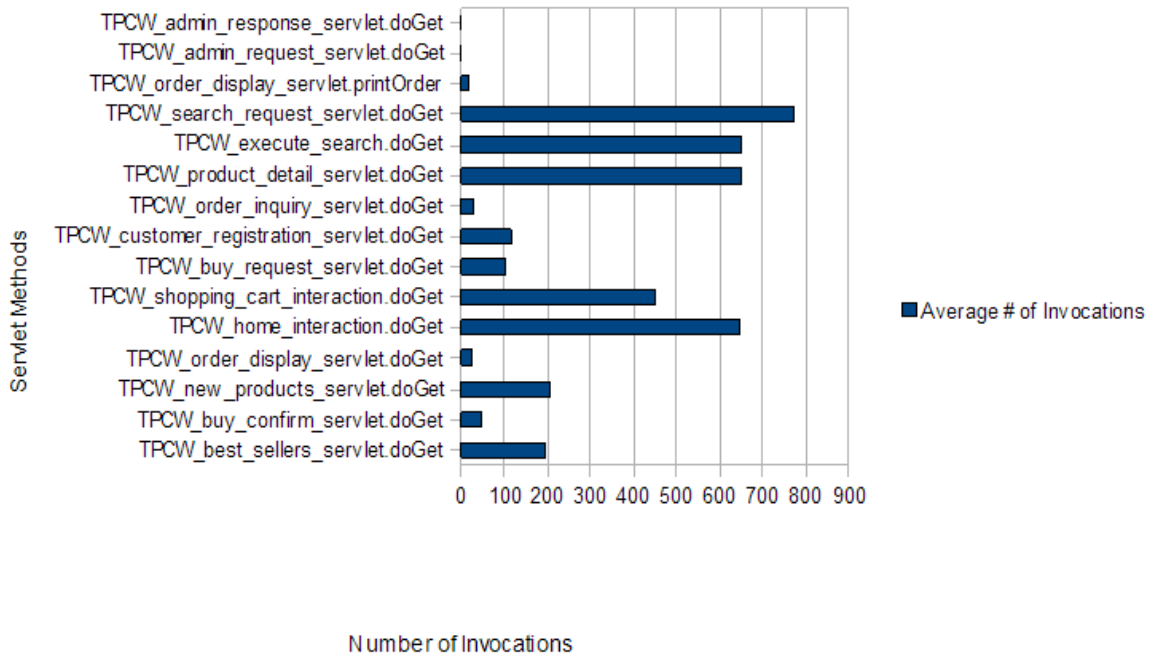

Figure 3.23: TPC-W servlet method invocation profile

in a JVM configured to use a maximum heap size of 64 MB. The TPCW_execute_search servlet is invoked approximately 651 times during a 25 minute TPC-W run with a 20 client load. Figures 3.24 and 3.25 show the effects of the memory leak injected into the TPC-W servlet on 1) the length of time the underlying JVM operates with a given heap-size and 2) the rate at which the JVM requests additional memory from the Operating System to accommodate the expanding heap. In addition to the increased memory-request rate, we also observe a surge in Garbage Collection activity as the JVM's Garbage Collector works to recover memory in order to compensate for the heap's growth (see Figure 3.26).

Processing Failures. To demonstrate a processing failure, we slow down the entire TPC-W web-application by injecting a 100 msec delay in every servlet method invoked by a remote client – the name of each servlet method is associated with a delay in the fault-model specification file used by Kheiron/JVM. Figure 3.27 shows the average execution time of

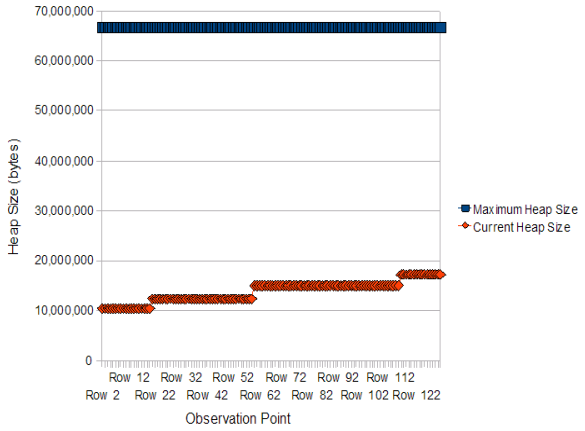

Figure 3.24: JVM memory request profile w/o Kheiron/JVM-injected memory leak

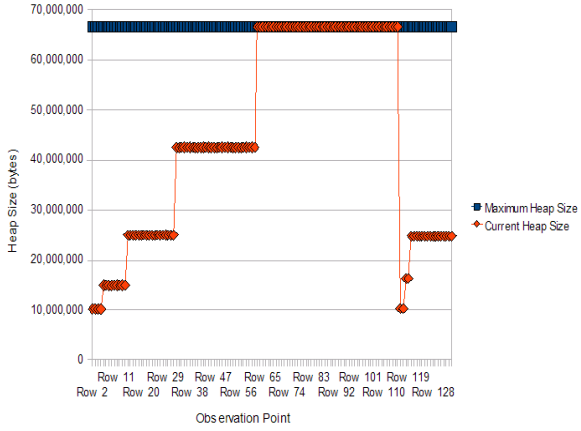

Figure 3.25: JVM memory request profile w/Kheiron/JVM-injected memory leak

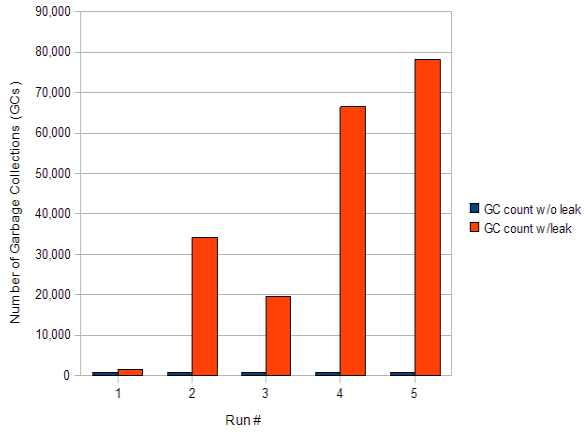

Figure 3.26: JVM garbage collection events with and without Kheiron/JVM-injected memory leak

the servlets with and without the Kheiron/JVM injected delays.

Configuration Failures. To demonstrate a targeted configuration failure, we cause the TPCW_execute_search servlet to fail 30% of the time by injecting a MissingResourceFault (by causing a Missing Resource Exception to be thrown). The invocation results are shown in Figure 3.28. Further stacktraces dumped to the application server logs (see Figure 3.29) confirm Kheiron/JVM as the cause of the failure.

In this section we demonstrated the ability of Kheiron/JVM to transparently instrument

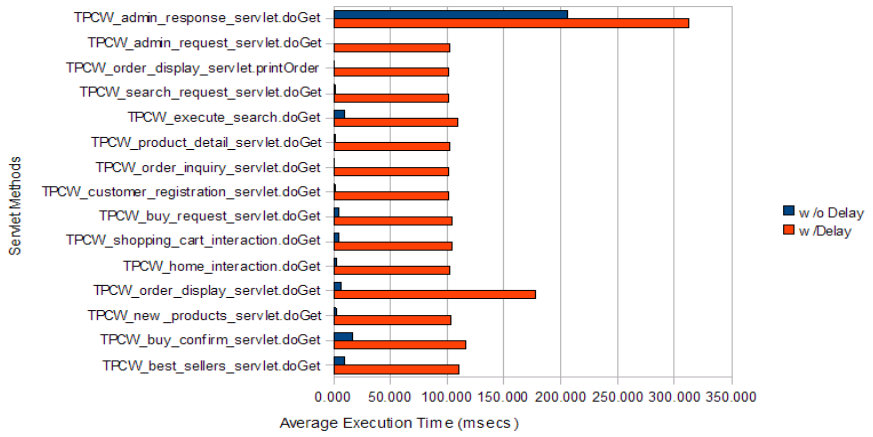

Figure 3.27: Average servlet method execution times with and without Kheiron/JVM-injected delays

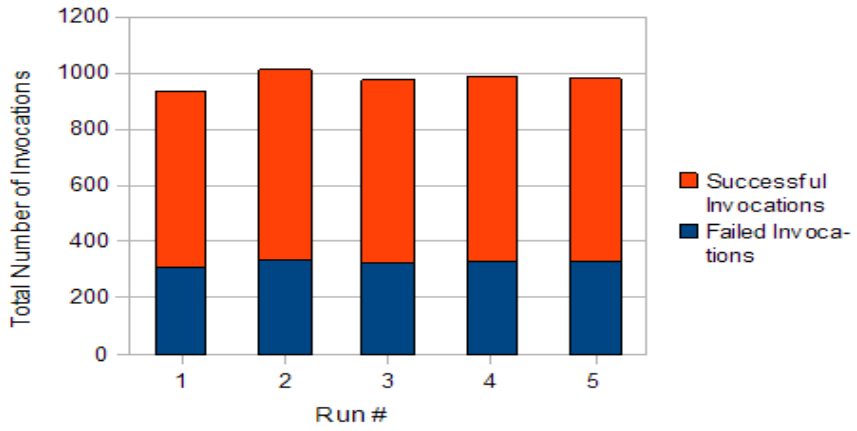

Figure 3.28: TPCW_execute_search invocation failures

and inject targeted faults into web-application classes hosted in an application server. We presented a simple fault-model for web-applications consisting of four failure categories, injected faults from three of the four categories and measured the effects on the web-application classes, remote clients accessing these classes and the the underlying Java Virtual Machine.

```
java.util.MissingResourceException:  Kheiron/JVM
MissingResourceException
at psl.kheiron.faults.MissingResourceFault.doInject
at psl.kheiron.faults.Fault.injectFault
at psl.kheiron.FaultManager.injectFault
at TPCW_execute_search.doGet
[Stacktrace edited for brevity]
at com.caucho.server.port.TcpConnection.run
at com.caucho.util.ThreadPool.runTasks
at com.caucho.util.ThreadPool.run
at java.lang.Thread.run
```

Figure 3.29: Injecting configuration faults with Kheiron/JVM

3.9 Kheiron/C: Runtime Adaptation of Compiled-C Programs

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 about program elements, limited or no built-in facilities for execution tracing, and less structured rules on well-formed programs.

In this section 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) [189] object files, on the Linux platform.

3.9.1 Native Execution Model

One unit of execution in the Linux operating system is the ELF executable. ELF is the specification of an *object file format*. Object files are binary representations of programs intended to execute directly on a processor as opposed to being run in an implementation of

an abstract machine such as the JVM or CLR. The ELF format provides parallel views of a file's contents that reflects the differing needs of program linking, loading and program execution.

Program loading is the procedure by which the operating system creates or augments a process image. A process image has segments that hold its text (instructions for the processor), data and stack. On the Linux platform the loader/linker maps ELF sections into memory as segments, resolves symbolic references, runs some initialization code (found in the *.init* section) and then transfers control to the *main* routine in the *.text* segment.

One approach to execution monitoring in an unmanaged execution environment is to build binaries in such a way that they emit profiler data. Special flags, e.g., -pg, are passed to the gcc compiler used to generate the binary. The executable, when run, will also write out a file containing the times spent in each function executed. Since a compile-time/link-time flag is used to create an executable that has logic built in to write out profiling information, it is not possible to augment the data collected without rebuilding the application. Further, selectively profiling portions of the binary is not supported.

To gain control of a running unmanaged application on the Linux operating system, tools use built-in facilities such as *ptrace* and the */proc* file system. ptrace is a system call that allows one process to attach to a running program to monitor or control its execution and examine and modify its address space. Several monitored events can be associated with a traced program including: the end of execution of a single assembly language instruction, entering/exiting a system call, and receiving a signal. ptrace is primarily used to implement breakpoint debuggers. Traced processes behave normally until a signal is caught – at which point the traced process is suspended and the tracing process notified [39]. The /proc filesystem is a virtual filesystem created by the kernel in memory that contains information about the system and the current processes in their various stages of execution.

With respect to metadata, ELF binaries support various processors with 8-bit bytes and

32-bit architectures. Complex structures, etc. are represented as compositions of 32-bit, 16-bit and 8-bit "types". The binary format also uses special sections to hold descriptive information about the program. Two important sections are the *.debug* and *.symtab* sections, where information used for symbolic debugging and the symbol table, respectively, are kept.

The symbol table contains the information needed to locate and relocate symbolic references and definitions. The fields of interest in a symbol table entry (Figure 3.30) are *st_name*, which holds an index into the object file's symbol string table where the symbol name is stored, *st_size*, which contains the data object's size in bytes and *st_info*, which specifies the symbol's type and binding attributes.

```
typedef struct {
          Elf32_Word    st_name;
          Elf32_Addr    st_value;
          Elf32_Word    st_size;
          unsigned char st_info;
          unsigned char st_other;
          Elf32_Half    st_shndx;
} Elf32_Sym;
```

Figure 3.30: ELF symbol table entry [189]

Type information for symbols can be one of: STT_NOTYPE, when the symbol's type is not defined, STT_OBJECT, when the symbol's type is associated with a data object such as variable or array, STT_FUNC, for a function or other executable code, and STT_SECTION, for symbols associated with a section. As we can see, the metadata available in ELF object files is not as detailed or as expressive as the metadata found in managed executables. For example, we lack richer information on abstract data types and their relationships, functions and their signatures – number of expected parameters, parameter types and function return types – i.e., limited support for sophisticated reflection and metadata APIs. Further, since unmanaged applications run on the underlying processor, there is no intermediary exposing an execution tracing and control API; instead we have to rely on platform-specific operating system support, e.g., ptrace and strace on Unix.

3.9.2 Kheiron/C Model of Operation

Our current implementation of Kheiron/C relies on the Dyninst API [18] (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. Behind the abstractions, Dyninst relies on trampolines – a small piece of code constructed on-the-fly on the stack – to alter the original flow of execution to include the inserted instrumentation (see Figure 3.31).

To use the Dyninst terminology, Kheiron/C is implemented as a *mutator* (Figure 3.32), which uses the Dyninst API to attach to and modify running programs. 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.

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 3.32) 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

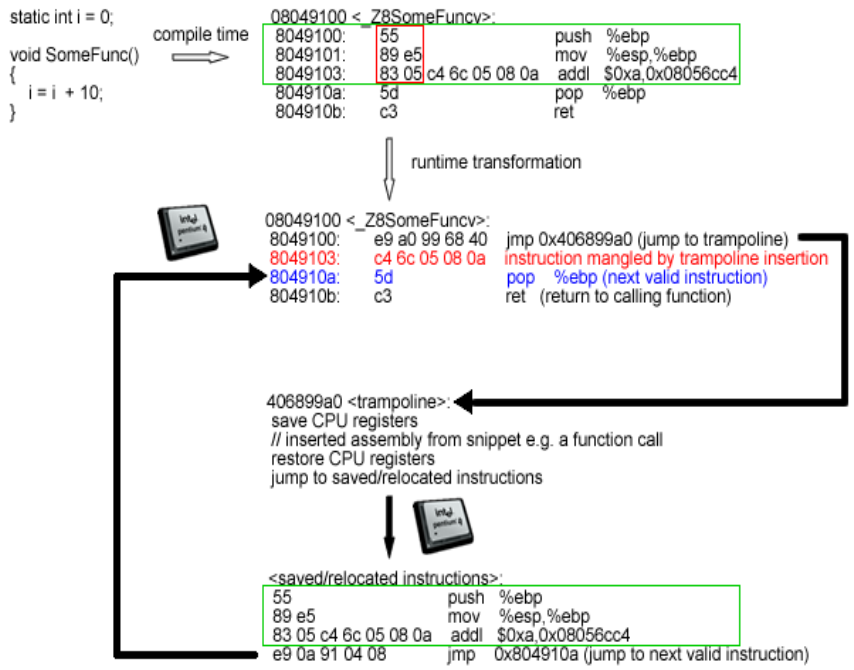

Figure 3.31: Dyninst model of operation

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 [71]. Kheiron/C uses the Dyninst API to locate global structures or local variables in the scope of the intended instrumentation points. It then loads any

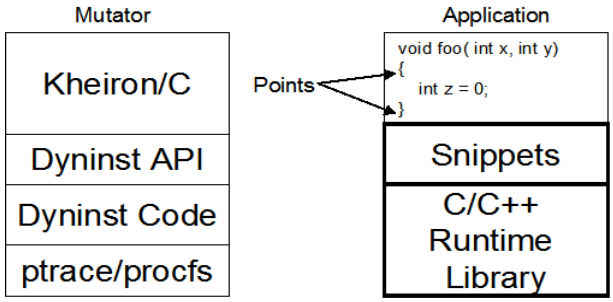

Figure 3.32: Kheiron/C

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, which are converted into assembly instruction sequences by Dyninst, and inserts them at the instrumentation points. Finally, Kheiron/C allows the target application to continue its execution.

3.9.3 Evaluation Part 1: Kheiron/C Performance Impact

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.

As shown in Figure 3.33, the overhead of the inserted function call is negligible, ~1%. This is expected since the x86 assembly generated behind the scenes effects a simple jump into

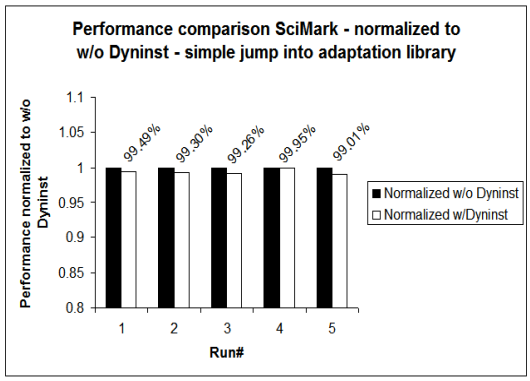

Figure 3.33: Kheiron/C overheads of simple instrumentation

the adaptation library followed XSby 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 by the inserted "snippets". Further, the time the SciMark process spends suspended while Kheiron/C performs the instrumentation is sub-second, ~684 msecs ± 7.0686.

3.9.4 Evaluation Part 2: Kheiron/C Injecting Selective Emulation

In this section, we explore the flexibility of Kheiron/C by using it enable a sophisticated runtime adaptation of a compiled-C application.

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 (see Figure 3.36). 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. [171] – 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)[171].

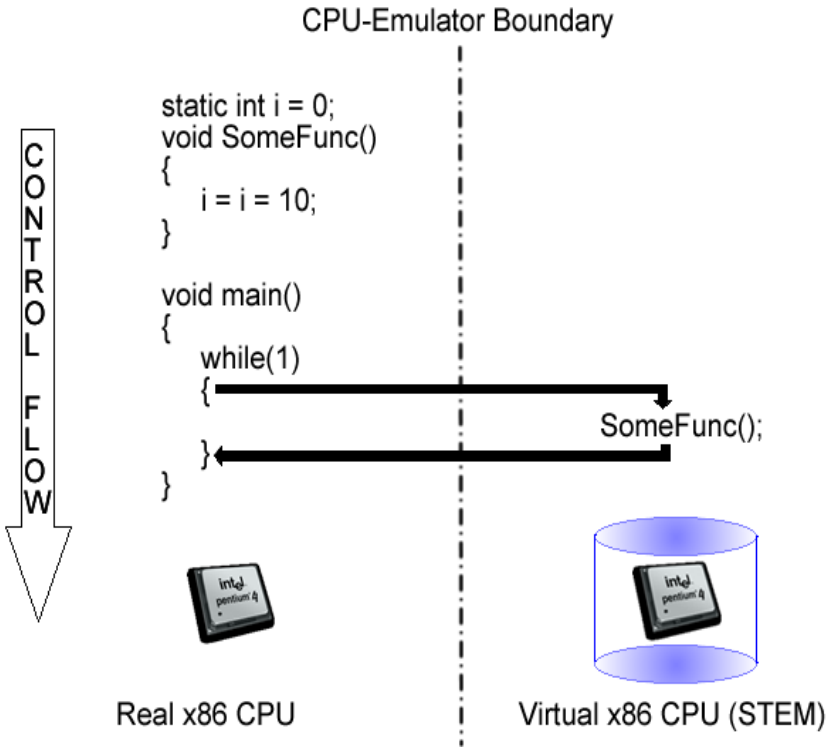

Figure 3.34: Selective emulation in action

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 3.35). 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

```
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 3.35: Inserting STEM via source code

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 3.35. 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[15] 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 [18].

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

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

under the emulator. Kheiron/C uses Dyninst functions that 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. EmulatorPrime 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, loads the STEM emulator and initializes it with the 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

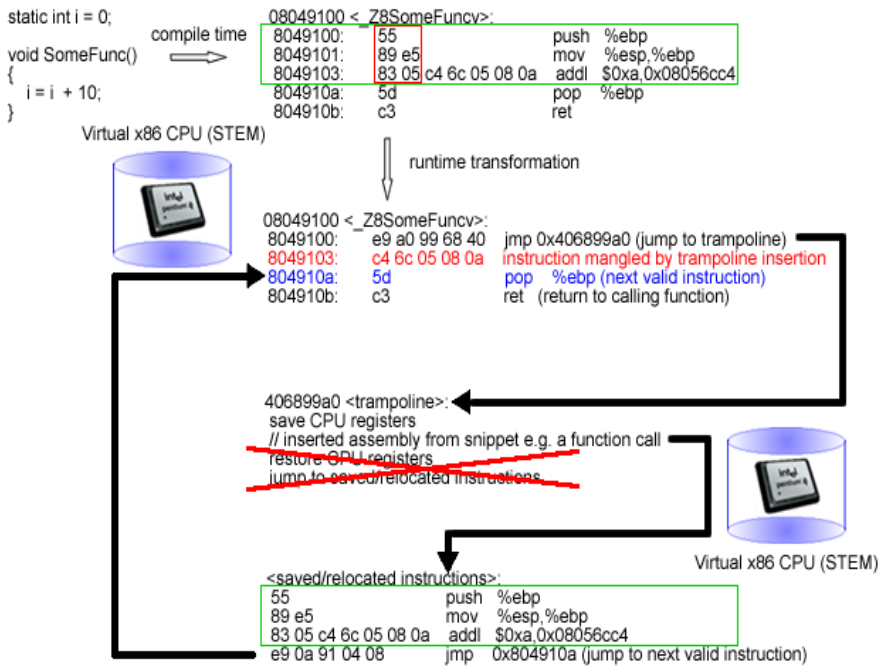

Figure 3.36: Selective emulation via Kheiron/C + Dyninst

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.

3.10 Integrity/Consistency-preserving Adaptations

To provide any guarantees that the runtime adaptations we effect preserve the integrity and consistency of the application and execution environment we need to have an understanding of how the execution environment works and how the application works.

In the preceding three sections (§3.7, §3.8 and §3.9) we discuss and demonstrate techniques for effecting a variety of runtime adaptations in applications running in managed and unmanaged execution environments; however, in the presentation of our runtime adaptation techniques we have primarily focused on the execution environment's role in ensuring that adaptations preserve the integrity/consistency of the application and the execution

environment.

For example, managed execution environments like the CLR and JVM have rules on what constitutes a valid program, these rules act as guidelines for our metadata and bytecode insertions and modifications. Similarly, the instruction set of the underlying processor, function call setup/cleanup conventions and linkage specifications provide guidelines for runtime adaptations in unmanaged execution environments.

However, understanding how the execution environment works is only one aspect of effecting integrity/consistency-preserving adaptations. Knowledge of how the target application operates is also important. Our case study of reconfiguring the Alchemi Enterprise Grid Computing system using Kheiron/CLR briefly touches on this (see §3.7.8).

Whereas Kheiron/CLR can effect adaptations in Alchemi without requiring access to source code for re-compilation, our ability to safely effect a scheduler swap in the Alchemi Manager was enhanced by having access to its source code. This access allowed us identify the main activities and actors (object instances, threads, variables, etc.) involved in its startup and shutdown processes and develop an adaptation strategy that effected an orderly shutdown of the Alchemi Manager and correctly re-initialized it with a with a new scheduler instance.

This example of adapting Alchemi safely underscores the need to understand the operation of the target system being adapted. Without access to knowledge of how the system works – either through source code, developer/engineer knowledge etc. – the only guarantees that can be given are ones concerning the execution environment's rules for valid programs.

Whereas analysis of executable units can identify structural dependencies, e.g., compile-time program-element dependencies, symbols and function/method implementations (bytecode or assembly instructions), while program tracing can provide insights into the sequences of operations that occur, they are not guaranteed to reveal the semantics of the system's operation – symbol names may be misleading or obfuscated/mangled – making it difficult to understand how the system works and harder to provide guarantees that the adaptations

being effected will not compromise its integrity/consistency.

3.11 Related Work

3.11.1 Runtime Adaptation

Our Kheiron prototypes are concerned with facilitating very fine-grained adaptations in existing/legacy systems, whereas systems such as KX [94] and Rainbow [169] are concerned with coarser-grained adaptations. However, the Kheiron prototypes could be used as low-level mechanisms orchestrated/directed by these larger frameworks.

JOIE [34] 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 [33]. 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 [105] 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) [9] 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 a method to generate an event when the 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 [16].

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 [61] 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 [57], Aspect C# [76]; those that perform weaving after compile time but before load time, e.g., Weave .NET [46], 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 [56] and CLAW [106]. 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 [21] and adaptive systems such as the K42 operating system [19] require upfront design-time effort to build in adaptation mechanisms.

[16] Kheiron/JVM uses the JVMTI, which was introduced in the v1.5 JVM and as a result only works with v1.5x or later JVM implementations.

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' [148] 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 [159] and Trap.NET [158] 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 [108] and ATOM [178], 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 [184] 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 [24] 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 [48] 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).

3.11.2 Software Implemented Fault-Injection Tools

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 [172], Ferrari [95] and Ftape [191], 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 specific 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 [70], 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 [112], 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) [68] and MARS (Maintainable Real-Time System) [99], fault-injection tools built using Kheiron 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 [165] 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.

6. Jaca [116] 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. through the use of a common model for interacting with programs dynamically that is built on top of existing execution environment facilities.

3.12 Summary

This chapter introduced Kheiron, a suite of tools for effecting fine-grained in-vivo and in-situ adaptations in software systems written in different languages (.NET, Java and C), running in different execution environments. We identified two major classes of execution environments, managed and unmanaged, and presented a generic model, which is used by Kheiron, for facilitating runtime adaptation in these execution environments.

	Unmanaged Execution Environment	Managed Execution Environment	
	ELF Binaries	JVM 1.5.x	CLR 1.1
Program tracing	ptrace, /proc	JVMTI callbacks	ICorProfiler ICorProfilerCallback
Program steering	Trampolines + Dyninst	Bytecode rewriting	MSIL rewriting
Execution-unit metadata available to query	.symtab, .debug sections	Classfile constant-pool + bytecode	Assembly, type & method metadata + MSIL
Metadata augmentation/ editing	N/A for compiled C-programs	Custom classfile parsing & editing APIs using BCEL + JVMTI RedefineClasses	IMetaDataImport IMetaDataEmit APIs

Table 3.12: Execution environment facilities

Our generic model of adaptation is based on four key facilities existing, or easily added to, contemporary execution environments: program tracing, program steering, metadata

querying and metadata editing. Table 3.12 summarizes techniques used to effect adaptations in the three execution environments studied – Microsoft's Common Language Runtime, Sun Microsystems' Java Virtual Machine and the unmanaged execution environment consisting of the Linux operating system and the raw x86 processor.

In elaborating on the implementation details of Kheiron, we comprehensively cover and compare the techniques that are used to effect runtime adaptations in the contemporary managed and unmanaged execution environments studied.

Finally, we demonstrate Kheiron's ability to effect fine-grained adaptations in multiple systems using three case studies: runtime reconfiguration of .NET applications using Kheiron/CLR (§3.7.8), runtime fault-injection in Java-based applications using Kheiron/JVM ((§3.8.6)) and selective emulation of C programs using Kheiron/C (§3.9.4). The next chapter develops an evaluation methodology and benchmark for assessing the Reliability, Availability and Serviceability (RAS) properties of software systems, which uses the runtime adaptation capabilities of Kheiron (specifically its in-vivo and in-situ fault-injection capabilities) to construct failure scenarios that are used in RAS-evaluations.

Part II

RAS Evaluations via Runtime Adaptation and RAS Modeling

This part describes the runtime fault-injection tools and analytical techniques that we combine to construct failure scenarios, which allow us to evaluate and compare the RAS capabilities of software systems.

Chapter 4

Evaluating RAS Capabilities

Evaluating and comparing the Reliability, Availability and Serviceability (RAS) capabilities of systems requires reasoning about aspects of the system's operation that may be difficult to capture or quantify using performance metrics alone.

Whereas performance metrics provide insights into the feasibility of using a system with its RAS-enhancing remediation mechanisms enabled, there are more in-depth analyses that we wish to perform. For example, we want to be able to evaluate the efficacy of any RAS mechanisms the system may have, reason about the expected benefits of yet-to-be-added RAS-enhancing mechanisms, reason about RAS deficiencies, evaluate different combinations of mechanisms, evaluate and compare mechanisms that may employ different remediation-strategies (reactive, preventative, proactive), reason about tradeoffs between mechanisms and identify under-performing or sub-optimal mechanisms. Measures concerned with overall system performance do not adequately capture the details that distinguish one remediation mechanism from another, e.g., remediation accuracy/success rates, fault/failure coverage, the impact of remediation failures, the consequences of remediation strategy/style and accounting for partially automated remediations. These deficiencies of performance metrics and benchmarks limit our ability to use them as a primary means of comparing or ranking

systems based on their RAS capabilities.

An additional consideration for evaluating the RAS capabilities of systems is that the notions of "good" and "better" are dependent on the *environmental constraints* governing the system's operation. For example, service level agreements (SLAs), policies, and internally/externally visible service level objectives including but not limited to: uptime guarantees, meeting production targets, reducing production delays, improving problem-resolution and service-restoration activities, etc. Whereas there are aspects of the environmental constraints that can be evaluated using performance metrics, such as response time guarantees in SLAs, these metrics are insufficient for evaluating other constraints.

As a result, evaluating and comparing RAS capabilities requires something beyond performance metrics and benchmarks. Specifically, tools and techniques that support more in-depth analyses of the details of RAS mechanisms (*the micro-view*), while considering the role and effects of the environmental constraints (*the macro-view*).

The importance of the environmental constraints in evaluating the RAS capabilities of systems cannot be understated since these constraints serve four major purposes. First, they help identify the failures and faults that impact these environmental constraints. Second, they enable reasoning about these impacts from the different perspectives of those affected (end-users, system operators/engineers/administrators and management). Third, they provide a source of possible metrics that can be used to quantify the impacts of RAS deficiencies, remediation failures and partially automated remediations. And finally, they establish the (scoring-)boundaries within which a system and its collection/composition of mechanisms can be considered to be better than another.

In this thesis we develop a model-based and measurement-based approach to evaluating the RAS capabilities of systems. Our evaluation approach is based on *failure scenarios*, which can be combined and extended to develop a RAS benchmark for a specific system or class of systems.

A failure scenario consists of three elements:

1. A set of faults that induce the failure of interest
2. A set of fault-injection tools capable of a) injecting one or more of the faults or b) otherwise inducing the failure of interest
3. A set of reusable analytical model templates used for scoring i.e. to quantify the impact(s) of a failure and/or the efficacy of any remediation mechanism(s) available and to capture the different perspectives of interest (end-user, operator/engineer and management)

4.1 Hypotheses

In Chapter 3 we demonstrated techniques and a suite of tools for effecting fine-grained adaptations that could be used to inject faults and induce failures in a variety of systems written in multiple languages running on different platforms. In the context of RAS evaluations and the construction of failure-scenarios, similarly flexible adaptation tools allow failure scenario support to be grafted onto existing/legacy systems allowing for the study of the failure behavior of systems and an evaluation of their RAS capabilities directly in their deployment environments. Such dynamic tools play a major role in our measurement-based evaluations of RAS capabilities.

The main hypothesis in this chapter is that **mathematical tools such as Markov chains, Markov reward networks and Control Theory models can be successfully used to describe failure scenarios designed to quantitatively evaluate/score the RAS capabilities of systems.** In validating this hypothesis we demonstrate how these tools can be used to create simple, reusable model templates for scoring and studying RAS properties. Further, we show that these model templates produced for scoring can be simpler than a detailed model of the implementation of the mechanisms, sub-system or system being studied while

still providing insights into the failure behavior of systems and the efficacy of its remediation mechanisms.

4.2 Analytical Tools

4.2.1 Continuous Time Markov Chains (CTMCs)

The first analytical tool we discuss is the Continuous Time Markov Chain (CTMC). We use CTMCs to model the failure behavior of systems, model the activities associated with remediations and quantitatively assess the impacts of these failures and/or remediations on facets of system reliability, availability and serviceability.

We choose CTMCs as one of our evaluation tools because of their flexibility, their ability to be combined and/or arranged hierarchically, their wide use and the existence of numerous solution techniques for their analysis [69, 101]. CTMCs have been well studied and have been used to analyze different classes of failures, e.g., independent [2], near-coincident [42] and cascading failures [91] in degradable, repairable and fault-tolerant systems.

CTMCs can also be used to reason about different remediation strategies – reactive, preventative, proactive [101]. Further, they provide the foundations for other modeling formalisms used to study the behavior of computer systems including, but not limited to: Stochastic Petri Nets (SPNs) and Stochastic Activity Networks (SANs) [161].

Finally, CTMCs can lead to the development of fluid models of system behavior allowing for powerful control theoretic analysis of system operation [139].

We now provide some background on CTMCs, highlighting the properties that make them suitable for modeling and evaluating RAS capabilities.

A Markov chain is defined as a Markov process with a finite (or countably infinite) state

space. A Markov process is a stochastic process whose dynamic behavior is such that the probability distributions for its future development depend only on the present state and not on how the process arrived in that state [101] – the *memoryless* property. The memoryless property of Markov processes greatly simplifies their analysis [69] and provides for tractable solution techniques.

Graphically, Markov chains can be represented by a directed graph. The vertices in the graph represent the states of system operation and the edges represent state transitions (see Figure 4.1).

A Markov chain is *irreducible* if all states in the chain can be reached pairwise from each other. A state in the chain is said to be *absorbing* if and only if no other state in the chain can be reached from it [69]. These two distinctions allow us to use Markov chains to model a software system (or aspects of its operation) as an infinitely running process or as a terminating/on-demand process [59], depending on the kind of system being studied and/or the kinds of analysis to be conducted e.g. studying the steady-state behavior of a system using irreducible Markov chains vs. reasoning about the expected time to completion for an operation using Markov chains with absorbing states.

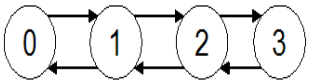

Figure 4.1: Markov chain

States in a Markov chain may be labeled/grouped to identify some interesting behavioral property of the system or process being modeled. For example a state may be marked as 'UP' to indicate that the system being modeled is doing useful work (this includes operating in a degraded mode); otherwise it may be labeled as 'DOWN' to indicate that the system is offline or not doing useful work.

A Markov chain may be (time)-homogeneous or non-homogeneous. In a homogeneous Markov chain, the transition probabilities are independent of the time epoch, i.e., the

probability of a transition from state s_i to s_j during an interval $[v, t]$ – usually written $p_{ij}(v, t)$ – depends only on the time difference $(t - v)$ rather than on the specific time/epoch (global clock value) when the transition occurs. However, in a non-homogeneous Markov chain the transition probabilities $p_{ij}(v, t)$ can change as a result of the current epoch (global clock). Using homogeneous Markov chains, allows us to model the failure behavior of a system as *time-independent*, whereas using non-homogeneous Markov chains allows us to analyze time-dependent failures, e.g., aging-related failures as presented in [13].

There are two classes of Markov chains, Continuous Time Markov Chains (CTMCs) and Discrete Time Markov Chains (DTMCs). Whereas DTMCs and CTMCs have a number of similarities, the major distinction between them comes with respect to when transitions between states can occur. This distinction is integral to our use of CTMCs to model failures and remediation activities rather than DTMCs.

In a DTMC, state transitions occur at fixed discrete time points. From its start state s_0, a DTMC evolves step by step according to one-step transition probabilities. For any time point the probabilities of a transition from s_i to s_j can be computed using the transition probabilities of the initial state, s_0. CTMCs, on the other hand, are more flexible. In a CTMC, state transitions occur at arbitrary points in time leading to a fluid-like interpretation of its behavior using rates of transition between states to describe/characterize the behavior of a CTMC over time.

State transition rates (i.e., state transitions and state sojourn times) of a homogeneous CTMC can be described by an exponential distribution [69] [1]. In the context of studying system failures the memoryless property of the exponential distribution implies that failures appear at random points during an interval. This can be restated as: the time we must wait for the next failure event is statistically independent of how long we have spent waiting for it to happen [72]. Modeling failures as randomly-appearing using CTMCs provides more

[1] The exponential distribution is the only continuous-time distribution that provides the memoryless property.

flexibility than DTMCs and may be more natural, for example, in situations where predicting failure events down to a specific time step, as can be done with a DTMC using its one-step transition probabilities, is difficult in the general case or unnecessary for the analysis at hand. However, if such specificity is required, *transient analysis* of the CTMC, using a technique called Uniformization can derive the one-step transition probability matrix for an equivalent DTMC [69].

The exponentially distributed rates of transition between the states of a CTMC and its structure – the arrangement of subsets of states in series, parallel or combinations thereof – can be used to model processes that may be characterized by a variety of probability distributions. In the context of RAS evaluations, the probability distributions are used to estimate the hazard rates (rates of failure), describe the failure behaviors and/or remediation activities and to construct the CTMCs used in their analysis.

There are a number of probability distributions that have been used to model failures and remediations including, but are not limited to: the Erlang-k distribution, the Hypo-exponential distribution, the Hyper-exponential distribution, the Weibull distribution and the Log-Logistic distribution. Details on estimating the rate of failure (hazard rate) for each of these probability distributions using exponentially distributed transition rates can be found in [101].

- The **Erlang-k distribution** is used to model processes with k sequential stages each having *identical* exponential rates of transition. The Exponential distribution is a special case of the Erlang distribution with k = 1. In the context of characterizing failure behaviors, the Erlang-k distribution can be used to model constant failure rate (CFR) distributions in systems without redundancy ($k = 1$) or in systems with redundancy ($k > 1$). Similarly, in the context of characterizing remediation activities the Erlang-k distribution can be used to characterize systems with constant repair times and a single shared repair station or multiple repair stations.

- The **Hypo-exponential distribution** is used to model processes with multiple sequential stages as well; however, it provides for a variation on the Erlang-k distribution, and allows each stage to have *different* exponential rates of transition. In the context of characterizing failure behaviors, the Hypo-exponential distribution is used to model systems with increasing failure rate (IFR) distributions. IFR distributions are one way to model cascading failures, where the rate of failure monotonically increases at each stage. In the context of characterizing remediation activities, IFR distributions are a prerequisite for preventative maintenance. If a system's failure behavior cannot be described by an IFR distribution then preventative maintenance will not result in any improvements [101]. The goal of preventative maintenance is to avoid the system entering a stage where its failure rate increases by performing actions that revert the system to an earlier point in its lifetime. If failure rates normally decrease over the lifetime of a system, preventative maintenance actions that revert the system to an early point in its lifetime would be counter-productive. The Hypo-exponential distribution can also be used to model remediations with multiple sequential steps where the remediation times are strictly increasing, e.g., resorting to one or more manual remediations after one or more automated remediations have been unsuccessful.

- The **Hyper-exponential distribution** is used to model processes with k alternate or parallel stages where the process can only occupy one stage at any time. In the context of characterizing failure behaviors, the Hyper-exponential distribution is used to model decreasing failure rate (DFR) distributions [101]. In the context of characterizing remediation activities, the Hyper-exponential distribution can be used for analyzing systems with multiple alternative remediations for a single failure. This includes considerations for imperfect remediations where $c\%$ of failures are successfully handled by remediation R_1 and $(1 - c)\%$ of failures are handled by remediation R_2.

- The **Weibull distribution** is a parametric distribution, which can be used to model DFR, CFR or IFR distributions. Whereas its modeling capabilities are equivalent to the Erlang-k, Hypo-exponential and Hyper-exponential distributions, it provides for more flexibility via the choice of its parameters.

- The **Log-Logistic distribution** is a less rigid probability distribution able to model more complex failure rate distributions than DFR (strictly decreasing), CFR (constant) and IFR (strictly increasing). The Log-Logistic distribution can be used to model processes where the rate of failure initially increases then decreases – UBT (upside-down bathtub) distributions [101]. The Log-Logistic distribution has been used in reliability growth models, which track reliability improvements over the lifetime of a system as enhancements are made to its design, subsystems and/or components.

For modeling more complex processes, CTMCs may be constructed to represent combinations of one or more of the probability distributions listed above. The Generalized Erlang distribution is one example of a distribution realized by combining the Hyper-exponential and Erlang-k distributions [69]. In addition to combining CTMCs to model different probability distributions, multiple CTMCs can be composed and/or arranged hierarchically to analyze a system at different levels of detail. Such compositions allow us to manage the size (state space) of the Markov chain being analyzed and is a standard largeness-avoidance technique for enabling the tractable analysis of complex Markov Chains [98]. In a hierarchical arrangement, sub-models can be evaluated independently and their results later combined using another sub-model.

In addition to being tractable to analyze, composable, and powerful enough to model complex failure behaviors and/or remediation activities that can be characterized by different probability distributions, techniques also exist for creating Markov chains to model processes that may have non-exponential probability distributions. In the “simple” case, approximations based on combinations of exponential distributions may be used. For more complex

cases techniques such as the inclusion of *supplementary and indicator variables* [121, 69] and *embedding techniques* [69] may be used to turn an initially non-Markov process into a Markov process, which can then be analyzed using existing solution techniques.

4.2.2 Markov Reward Networks

Markov Reward Networks are a simple extension of Markov Chains that allow us to assign cost or reward structures (values) to states and/or transitions of a Markov process. As a result, Markov Reward Networks provide a unifying framework for an integrated specification of model structure and system requirements [69].

Markov Reward Networks have been used extensively in optimization problems in Markov decision theory [69] and performability analysis [120] (an integrated approach to evaluating performance and dependability characteristics of computing systems).

In our construction of RAS models we use Markov Reward Networks based on Continuous Time Markov Chains (CTMCs) discussed earlier (4.2.1) to quantify the impacts of failure and/or remediations. Using Markov Reward Networks does not preclude considering the performance implications of failure and/or remediations, e.g., degradation since performance related measures such as throughput per unit time can be assigned to one or more states in the CTMC and used to quantify the performance impacts. Assigning rewards or costs to CTMC states combined with the appropriate labeling/grouping of states allow us to capture the impacts of failures and/or remediations from the three different perspectives of interest – end user/client, administrator/operator/engineer and business/management (examples of this are provided in 4.3).

4.2.3 Feedback Control Models

In this thesis we use principles of the branch of Control Theory concerned with Feedback control systems to reason quantitatively about a system's ability to meet the operational goals and environmental constraints (policies), which govern its operation.

We use feedback control models as one of our evaluation tools because of the framework it provides for realizing *predictable* systems – systems where the expected response of the system to changes (in the system and/or in its environment) can be characterized and/or evaluated quantitatively. Further, feedback control can also be used to realize robust *adaptive* systems – systems that automatically adjust to reject disturbances and accommodate noise while continuing to meet their operational goals.

We posit that predictable systems are easier to manage than unpredictable systems, and as a result predictability affects the Serviceability characteristics of a system concerned with meeting objectives in the presence of failures and/or remediations. In this section we identify the properties of feedback control systems that can be used to quantify facets of system Serviceability in the development of our RAS models.

Control Theory and feedback control has been widely studied and employed in other engineering disciplines including, but not limited to: mechanical engineering and electrical engineering. Further, despite the stochastic nature of computing systems, feedback control has been applied to their analysis and design with encouraging results [90, 145, 45, 196] [2].

We now provide some background on Feedback control, highlighting the properties that make it suitable for use in constructing RAS models.

Control Theory is concerned with the study of dynamical systems and is commonly used to achieve one or more of the following objectives: *regulatory control*, *disturbance rejection* and *optimization*.

[2] One approach for dealing with system stochastics involves building on results from Queuing Theory[90] when developing feedback control models of system behavior.

Regulatory control ensures that some measurable characteristic (*measured output*) of the (target) system is equal to or near a desired/specified reference value (*reference input*). Disturbance rejection ensures that disturbances acting on the system do not significantly affect its measured output. And finally, optimization is concerned with obtaining the best value of the measured output of the system. In our development of RAS models we are interested primarily in regulatory control.

There are two main classes of control systems, open-loop (*feedforward*) control systems (Figure 4.2) and closed-loop (*feedback*) control systems (Figure 4.3). Before discussing the differences between feedforward and feedback control systems we first describe the elements typically found in control systems:

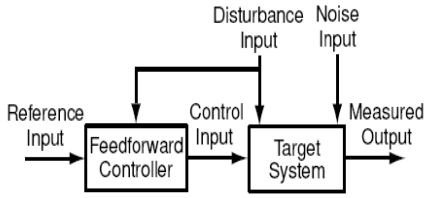

Figure 4.2: Block diagram of feedforward control [90]

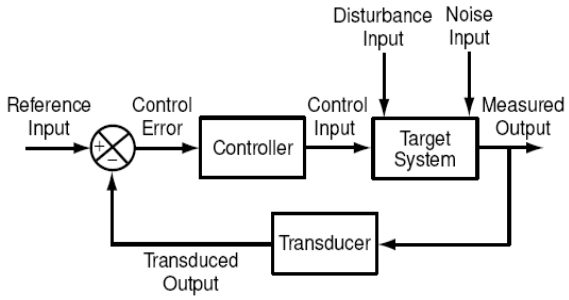

Figure 4.3: Block diagram of a feedback control system [90]

- **The Target System** – the system to be controlled.
- **Measured output** – one or more measurable characteristics of the target system.
- **Reference input** – desired value(s) of the measured output.

- **Control input** – one or more (*dynamically adjustable*) parameters that affect the behavior of the target system.
- **The Controller** – manipulates/determines the setting of the control input to achieve the reference input.
- **The Transducer** – transforms the measured output such that it can be compared with the reference input and/or used by the Controller. Examples include moving-average filters and unit conversions.
- **Control error** – difference between the measured output and the reference input.
- **Disturbance input** – changes that affect the way the control input influences the measured output.
- **Noise input** – any effect that changes the measured output of the target system.

The main difference between feedforward control and feedback control is the role of the measured output in each of these control systems and its implications for controller design.

Feedforward controllers use the reference input (and sometimes the disturbance input) to determine the setting of the control input needed to achieve the desired measured output. Unlike feedback controllers, they do not use the measured output to adjust the control input. As a result, feedforward control is more suitable for systems where the control input is a deterministic function of the reference and/or disturbance input and an accurate model of the system that is robust to changes in the system and its operating environment is available or can be constructed [90]. These properties of feedforward control systems, while making them less complex to design than feedback control systems, also make them less flexible/adaptive.

In addition to being more flexible than feedforward control systems, feedback control systems and the design principles used to realize them can be used to develop systems that exhibit four desirable properties – referred to as SASO properties – and analyze whether

systems exhibit any or all of these properties:

1. **Stability** – a stable system produces bounded output for any bounded input (these control systems are sometimes referred to as being Bounded Input Bounded Output/BIBO stable).

2. **Accuracy** – the measured output of an accurate control system converges or becomes sufficiently close to the reference input (small steady-state control error).

3. Short **settling times** – a control system with short settling times quickly converges to its steady state value.

4. Avoids **overshoot** – a control system that avoids overshoot allows changes to the control input to made while maintaining its measured output.

The flexible/adaptive nature of feedback control systems and the ability to analyze the SASO properties of such systems provides a framework for codifying operational policies (internally and externally visible service level objectives, environmental constraints, SLAs etc.) for a computing system and reasoning about the ability of the system to meet these goals in the presence of failures and/or remediation activities.

4.3 Analysis Techniques

In §4.2 we described the three analytical tools/ frameworks: Continuous Time Markov Chains (CTMCs) §4.2.1, Markov Reward Networks §4.2.2, and Feedback Control §4.2.3 – and their associated properties that motivated their use in the creation of (RAS) models used to analyze the failure behavior and/or remediation activities of systems. In this section we discuss the specific facets (metrics) of reliability, availability and serviceability quantified using our RAS models and describe the analysis techniques used to calculate them.

To aid this discussion we demonstrate the construction and use of RAS models via an exam-

ple analytical evaluation of a recursively restartable (microrebootable) J2EE[3] application server prototype developed by the Berkeley/Stanford Recovery Oriented Computing (ROC) group on top of a modified version of the open-source JBoss application server. Our analysis complements the measurement-based evaluation done in [20][4] and uses the results reported therein to derive estimates for RAS model parameters.

Whereas the evaluation done in [20] focuses primarily on comparing fine-grained microreboots to coarser-grained full-system reboots, the goal of our analysis is to create an RAS model that can be used to describe failure scenarios for a system using recursive microreboots and to score/evaluate the system's responses.

4.3.1 Microreboot RAS Model

Recursive microreboots are a technique for improving overall system availability by reactively restarting failed components and rejuvenating functioning components to prevent degradation [21]. It is specifically targeted at recovering from failures such as crashes, deadlocks, infinite loops, livelocks and state corruption (memory leaks, dangling pointers, damaged heaps, etc.).

A microreboot (μRB) can be applied at different levels of a system: component-level, subsystem-level or whole-system level[5]. As a remediation technique, recursive microreboots target the minimal set of a system's components for a restart and progressively restart larger subsets of components up to and including restarting the entire system. Microreboots, like whole-system reboots, have a number of properties in common that make them attractive as a remediation mechanism. They return the target of recovery (component, subsystem,

[3] Java 2 Platform, Enterprise Edition (J2EE) defines the standard for developing multi-tier enterprise applications [128].

[4] Additional measurement-based evaluations can be found in [23] and [22].

[5] The ability to precisely target and restart system elements at these various levels depend on a number of structural properties and design considerations of the system under consideration. See [21] for more details on design considerations for recursively restartable systems.

system) to a well-understood state – its start state. Further, they provide a high confidence way of reclaiming stale or leaked resources [21].

We chose recursive microreboot for our analysis example because it is an instance of a sophisticated remediation mechanism that exhibits a number of characteristics that make it interesting to study:

1. Layered recovery strategy – one layer for each level at which recovery can occur in the system.

2. Imperfect recovery between layers – failures can escalate to higher layers e.g. if component-level reboots are unsuccessful then the failure "bubbles" up to the next higher layer to be handled – subsystem-level reboots – and so on.

3. Problem mitigation rather than elimination – microreboots do not eliminate the underlying root cause of the problem, rather they attempt mitigate its effects. Over time, the same failures can resurface.

In [20], the authors evaluate the efficacy of microrebooting, comparing fine-grained microreboots to coarse-grained system reboots using their microrebootable J2EE application server, custom fault-injection tools and eBid, a version of the Rice University Bidding System (RUBiS) N-tier web-application, modified to be amenable to microreboots. RUBiS is a J2EE/Web-based auction system modeled after eBay.com.

The test system deployment in [20] consists of the following elements, which also correspond to the units of recovery. These recovery units are listed in order of fine-grained restarts to coarse-grained restarts:

- Enterprise Java Beans (EJBs) – these encapsulate the business logic of the eBid web-application. They may interact with other EJBs and/or backend databases in the processing of a client request.

- Web Archive (WAR) – this is the unit of deployment for the web application. It

contains the presentation tier of the web application: Java Server Pages (JSPs) and servlets. These invoke EJB methods and format the returned results for presentation to the client.

- eBid web-application – the collection of EJBs, JSPs and servlets.
- JVM/JBoss – the execution/hosting environment for the eBid web-application.

A Recovery Manager component added to the JBoss application server performs failure diagnosis and recovery guided by the simple recursive policy of "cheapest recovery first". In response to the faults injected into eBid, the recovery manager progressively reboots larger sets of components: first EJBs, then eBid's WAR, then the eBid web application, followed by the JVM/JBoss, and if necessary finally reboots the operating system. To fully resolve some failures microreboots may be followed up by additional automated or manual actions, e.g., recovering persistent data may be done automatically (via transaction rollback) or may require manual reconstruction of the data in the database.

Based on the description of the microrebootable application server in [20], we use the SHARPE [160] RAS modeling and analysis tool to generate a model (shown in Figure 4.4) that can be used to evaluate the efficacy of the application server and its recovery manager. The RAS model is an irreducible CTMC that consists of 6 states and 17 parameters, see Table 4.1.

Our RAS model captures a number of key elements of the operation of the application server's recovery manager including: a) multiple layers of recovery and b) the possible escalation of failures to higher levels of recovery. Further, the use of an irreducible CTMC allows us to model the operation of the Recovery Manager as an infinitely running process where failures can re-occur.

This RAS model, plus fault-injection tools like Kheiron (Chapter 3) or the ones used in the experiments in [20], can be used to design, initiate and score fault-injection experiments

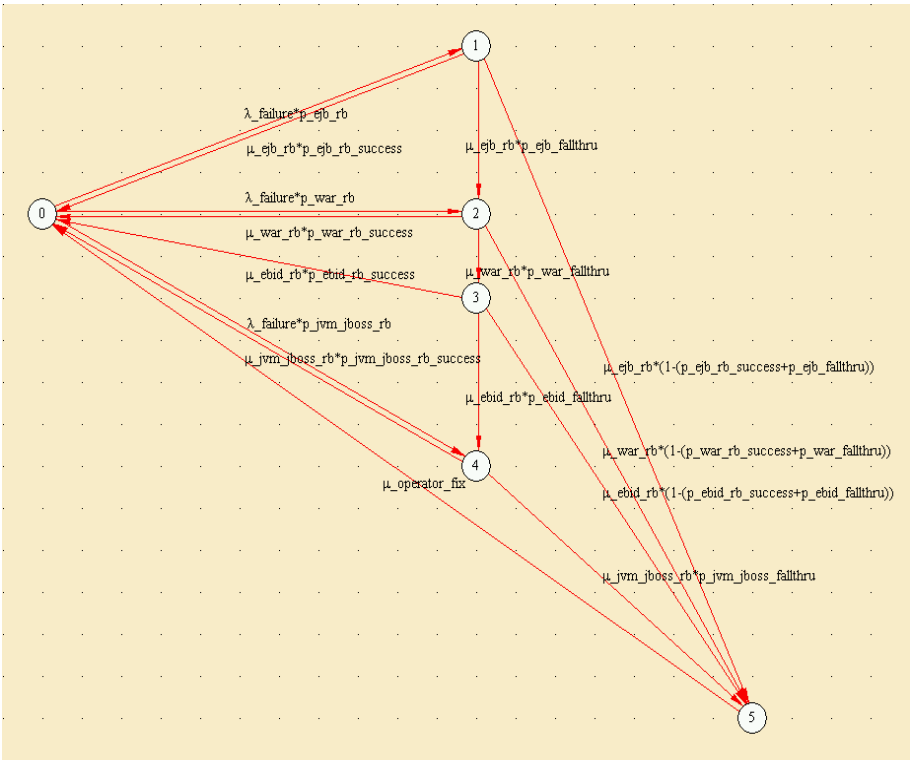

Figure 4.4: RAS model for a microrebootable application server

that represent different failure scenarios for evaluating the efficacy of microreboots.

Fault-injection tools can be used to control the rate of failure ($\lambda_{failure}$) and/or the proportions of failures that initially target a specific level of recovery (p_{ejb_rb}, p_{war_rb}, and $p_{jvm\_jboss\_rb}$). Varying these parameters allows us to study the behavior of the system under different fault-loads/failure mixes.

Parameters concerned with the success or failure of recovery at a specific level ($p_{ejb\_rb\_success}$, $p_{ejb\_fallthru}$, $p_{war\_rb\_success}$, $p_{war\_fallthru}$, $p_{ebid\_rb\_success}$, $p_{ebid\_fallthru}$, $p_{jvm\_jboss\_rb\_success}$, and $p_{jvm\_jboss\_fallthru}$) can be observed experimentally or varied in the model to reason about their expected impacts on system operation.

Parameters concerned with recovery times at a specific level (μ_{ejb_rb}, μ_{war_rb}, μ_{ebid_rb}, $\mu_{jvm\_jboss\_rb}$, and $\mu_{operator\_fix}$) can be observed experimentally or varied in the model based on simple rules of thumb, e.g., an order of magnitude increase in recovery time as the recovery

S_0	The initial state of the system
S_1	State where one or more EJBs is being restarted
S_2	State where the eBid WAR file is being restarted
S_3	State where the entire eBid application is being restarted
S_4	State where the JVM/JBoss application server is being restarted
S_5	State where an operator performs some action(s) to resolve an issue
$\lambda_{failure}$	Rate at which faults are injected/failures induced
p_{ejb_rb}	Proportion of failures that are initially handled by an EJB restart
p_{war_rb}	Proportion of failures that are initially handled by a WAR restart
$p_{jvm\_jboss\_rb}$	Proportion of failures that are initially handled by a JVM/JBoss restart
$p_{ejb\_rb\_success}$	Proportion of failures successfully resolved by an EJBs restart
$p_{ejb\_fallthru}$	Proportion of failures that fall through to WAR restart level
$p_{war\_rb\_success}$	Proportion of failures successfully resolved by a WAR restart
$p_{war\_fallthru}$	Proportion of failures that fall through to eBid restart level
$p_{ebid\_rb\_success}$	Proportion of failures successfully resolved by restarting eBid
$p_{ebid\_fallthru}$	Proportion of failures that fall through to JVM/JBoss restart level
$p_{jvm\_jboss\_rb\_success}$	Proportion of failures successfully resolved by a JVM/JBoss restart
$p_{jvm\_jboss\_fallthru}$	Proportion of failures that fall through to operator fix level
μ_{ejb_rb}	EJB restart time
μ_{war_rb}	WAR restart time
μ_{ebid_rb}	eBid web-application restart
$\mu_{jvm\_jboss\_rb}$	JVM/JBoss restart
$\mu_{operator\_fix}$	Time for an operator resolution

Table 4.1: RAS model parameters for a microrebootable application server

level increases.

Finally, labeling states associated with normal request processing or degraded request processing as UP states and states where no requests are processed as DOWN states allow us to capture different perspectives on what it means for the microrebootable application server to be considered “working”. By adjusting state-labels and varying the parameters of the RAS model, we can quantify various facets of reliability, availability and serviceability for the microrebootable application server.

4.3.2 Model Analysis – RAS Measures and Metrics

In our example analysis we use the numerical parameter values shown in Table 4.2 to describe a specific failure scenario used to evaluate the efficacy of microreboots:

$\lambda_{failure}$	3 failures every 10 minutes (3/600,000 msecs) [20]
p_{ejb_rb}	100%
p_{war_rb}	0%
$p_{jvm\_jboss\_rb}$	0%
$p_{ejb\_rb\_success}$	95%
$p_{ejb\_fallthru}$	5%
$p_{war\_rb\_success}$	95%
$p_{war\_fallthru}$	5%
$p_{ebid\_rb\_success}$	95%
$p_{ebid\_fallthru}$	5%
$p_{jvm\_jboss\_rb\_success}$	95%
$p_{jvm\_jboss\_fallthru}$	5%
μ_{ejb_rb}	EJB restart time – 1/501.27 msecs[6]
μ_{war_rb}	WAR restart time – 1/1,028 msecs (Table 3 [20])
μ_{ebid_rb}	eBid web-application restart – 1/7,699 msecs (Table 3 [20])
$\mu_{jvm\_jboss\_rb}$	JVM/JBoss restart – 1/19,083 msecs (Table 3 [20])
$\mu_{operator\_fix}$	Time for an operator resolution – 1/5 minutes (1/300,000 msecs)

Table 4.2: Microrebootable application server RAS model failure scenario parameters

In this failure scenario, recovery is always initiated at the EJB restart level ($p_{ejb\_rb} = 100\%$), recovery at each level is assumed to be 95% successful and human operators are only involved if recovery of a failure escalates beyond the JVM/JBoss level.

Note that whereas some of the numerical parameter values used in our analysis are based on experimental results reported in [20], e.g., $\lambda_{failure}$, μ_{ejb_rb}, μ_{war_rb}, μ_{ebid_rb}, and $\mu_{jvm\_jboss\_rb}$, the remaining parameter values are hypothetical and are used solely to discuss the different RAS measures and metrics that can be calculated.

The first step in our analysis is to use SHARPE to compute the steady-state probability vector, π, for the CTMC in Figure 4.4 using the numerical parameters in Table 4.2 – see Table 4.3 for the results. The steady-state probabilities are the time-independent probabilities of being

[6] Average restart time of the 22 EJBs in eBid, calculated using Table 3 in [20].

in a particular state of the CTMC as time, $t \rightarrow \infty$. In the sections below we demonstrate how the steady-state probability vector is used in the calculation of various reliability, availability and serviceability measures.

π_0	0.997127
π_1	0.002499
π_2	0.000256
π_3	0.000096
π_4	0.000012
π_5	0.000009

Table 4.3: Microreboot RAS model steady-state probabilities

4.3.3 Reliability Measures

Reliability measures emphasize the occurrence of undesirable events in the system [72]. There are a number of forms and metrics that can be used to express the reliability of a system including:

1. Reliability Functions – the probability that an incident of sufficient severity has not yet occurred since the beginning of a time interval of interest.

2. Mean Time to Failure (MTTF) – the average length of time that elapses until an incident occurs.

3. Frequency of Incidents – the average number of incidents that occur per unit time.

In our analytical evaluation of the microrebootable application-server we discuss its reliability in terms of the Frequency of Incidents, where the frequency of an incident is a function of the probability of being in a particular state, π_i.

For the microrebootable application server we can identify four kinds of incidents that may affect its reliability:

1. Frequency of failure escalations to higher levels of recovery ($F_{a \rightarrow b}$), i.e., frequency of

S_1 to S_2 transitions, S_2 to S_3 transitions, S_3 to S_4 transitions or S_4 to S_5 transitions. Frequent failure escalations delays system recovery, may signal instabilities in the system or its environment, or may result in other disruptions.

2. Frequency of recovery activities (F_2), i.e., time spent in S_1, S_2, S_3, S_4, and S_5.

3. Frequency of outages resulting from more expensive recovery actions (F_3) e.g. eBid or JVM/JBoss restart vs. EJB or WAR restarts.

4. Frequency of recovery actions exceeding a given duration tolerance (F_4).

Frequency of failure escalations ($F_{a\to b}$). The frequency of failure escalations to higher levels of recovery ($F_{a\to b}$) during an interval T is given by:

$$F_{a\to b} = T * (\gamma_{a\to b} * \pi_a) \tag{4.1}$$

Where $\gamma_{a\to b}$ is the rate of transition out of recovery state S_a to higher level recovery state S_b.

During an interval of 1 day (T = 1,440 minutes = 86,400,000 msecs) we expect to inject a total of $1440 * \frac{3\ failures}{10\ minutes} = 432$ failures of which, 21.537952 (Equation 4.2) are escalated from EJB recovery level to WAR recovery level, 1.076898 (Equation 4.3) are escalated from WAR recovery level to eBid recovery level, 0.053845 (Equation 4.4) are escalated from eBid to JVM/JBoss recovery level and 0.002692 (Equation 4.5) are escalated from JVM/JBoss recovery level to Operator recovery level, resulting in a total of of 22.671386 failure escalation events per day (see Table 4.4).

$F_{ejb\to war}$	21.537952
$F_{war\to eBid}$	1.076898
$F_{eBid\to JVM/JBoss}$	0.053845
$F_{JVM/JBoss\to Operator}$	0.002692
Total	22.671386

Table 4.4: Failure escalation incidents per day

$$\begin{aligned} \gamma_{ejb \rightarrow war} &= p_{ejb_{f}allthru} * \mu_{ejb_{r}b} = \left(0.05 * \frac{1}{501.27}\right) \\ F_{ejb \rightarrow war} &= 86,400,000 * \gamma_{ejb \rightarrow war} * \pi_1 \\ \rightarrow F_{ejb \rightarrow war} &= 86,400,000 * \gamma_{ejb \rightarrow war} * 0.002499 = 21.537952 \end{aligned} \tag{4.2}$$

$$\begin{aligned} \gamma_{war \rightarrow eBid} &= p_{war_{f}allthru} * \mu_{war_{r}b} = \left(0.05 * \frac{1}{1028}\right) \\ F_{war \rightarrow eBid} &= 86,400,000 * \gamma_{war \rightarrow eBid} * \pi_2 \\ \rightarrow F_{war \rightarrow eBid} &= 86,400,000 * \gamma_{war \rightarrow eBid} * 0.000256 = 1.076898 \end{aligned} \tag{4.3}$$

$$\begin{aligned} \gamma_{eBid \rightarrow JVM/JBoss} &= p_{ebid_{f}allthru} * \mu_{ebid_{r}b} = \left(0.05 * \frac{1}{7699}\right) \\ F_{eBid \rightarrow JVM/JBoss} &= 86,400,000 * \gamma_{eBid \rightarrow JVM/JBoss} * \pi_3 \\ \rightarrow F_{eBid \rightarrow JVM/JBoss} &= 86,400,000 * \gamma_{eBid \rightarrow JVM/JBoss} * 0.000096 = 0.053845 \end{aligned} \tag{4.4}$$

$$\begin{aligned} \gamma_{JVM/JBoss \rightarrow Operator} &= p_{ebid_{j}vm_{j}boss_{f}allthru} * \mu_{jvm_{j}boss_{r}b} = \left(0.05 * \frac{1}{19083}\right) \\ F_{JVM/JBoss \rightarrow Operator} &= 86,400,000 * \gamma_{JVM/JBoss \rightarrow Operator} * \pi_3 \\ \rightarrow F_{JVM/JBoss \rightarrow Operator} &= 86,400,000 * \gamma_{JVM/JBoss \rightarrow Operator} * 0.000012 = 0.002692 \end{aligned} \tag{4.5}$$

Frequency of recovery activities (F_2). The frequency of recovery activities (F_2) during an interval T is given by:

$$F_2 = T * (\pi_2 + \pi_3 + \pi_4 + \pi_5) \tag{4.6}$$

During an interval of 1 day (T = 86,400,000 msecs) we expect the recovery manager of the microrebootable application server to spend 4.14 minutes per day performing failure recovery activities (Equation 4.7).

$$F_2 = 86,400,000 * (\pi_2 + \pi_3 + \pi_4 + \pi_5) = 248,193.82\ msecs\ (4.14\ mins) \tag{4.7}$$

Frequency of outages (F_3). The frequency of outages during 1 day resulting from more expensive recovery actions (F_3) is given by:

$$F_3 = F_{eBid \rightarrow JVM/JBoss} + F_{JVM/JBoss \rightarrow Operator} \tag{4.8}$$

We therefore expect the microrebootable application server to experience 0.056537 outages per day (20.64 per year) due to expensive recovery actions (Equation 4.9).

$$F_3 = 0.053845 + 0.002692 = 0.056537 \tag{4.9}$$

Frequency of recovery actions exceeding a given duration tolerance (F_4). The frequency of recovery actions exceeding a given duration tolerance (F_4) in an interval T is given by:

$$F_4(\tau) = T * \left[\sum_{i \in S_{Recovery}} \gamma_{i \rightarrow 0} * \pi_i * e^{(-\gamma_{i \rightarrow 0} * \tau)} \right] \tag{4.10}$$

Where:

- τ is the duration tolerance in time units.
- $S_{Recovery} = \{ S_1, S_2, S_3, S_4, S_5 \}$.
- $\gamma_{i \to 0}$ is the rate of transition from a recovery state $S_i \in S_{Recovery}$ to the failure free state, S_0.
- $e^{(-\gamma_{i \to 0} * \tau)}$ is the probability that a recovery action takes longer than τ time units.

Of the 432 failure recovery actions initiated per day (T = 86,400,000 msecs) , the number of recovery actions expected to exceed 1000 msecs is 70.578179 (Equation 4.11).

$$
\begin{aligned}
F_4(1000\ msecs) = T \quad * \quad & (0.95 * \frac{1}{501.27} * \pi_1 * e^{-0.95 * \frac{1}{501.27} * 1000} \\
+ \quad & 0.95 * \frac{1}{1028} * \pi_2 * e^{-0.95 * \frac{1}{1028} * 1000} \\
+ \quad & 0.95 * \frac{1}{7699} * \pi_3 * e^{-0.95 * \frac{1}{7699} * 1000} \\
+ \quad & 0.95 * \frac{1}{19083} * \pi_4 * e^{-0.95 * \frac{1}{19083} * 1000} \\
+ \quad & \frac{1}{300000} * \pi_5 * e^{-\frac{1}{300000} * 1000}) \\
= \quad & 70.578179
\end{aligned}
\tag{4.11}
$$

4.3.4 Availability Measures

Availability measures capture the proportion of total time in which a system is in an operational condition [72]. To discuss system availability using the RAS model shown in Figure 4.4, we need to identify which states of the model represent an operational state (UP state) or an outage state (DOWN state). Availability measures can then be expressed as a function of the UP states.

There are three forms and metrics that can be used to express the availability of a system [72]:

1. Instantaneous (or point) basic availability – the probability that a system is up at time t.

2. Steady state basic availability – the probability that the system is up assuming that the system has reached a steady state (i.e., time $t \rightarrow \infty$).

3. Interval basic availability – the proportion within a given interval of time that the system is up, which is calculated by carrying out a time average value of instantaneous availability over the time interval of interest.

In our analytical evaluation of the microrebootable application server we discuss its availability in terms of its steady-state basic availability (SS_{avail}), where the steady-state basic availability is a function of the probability of being in a particular state, π_i.

For the microrebootable application server we can identify three perspectives on its steady-state availability that may be of interest:

1. Basic steady-state availability where all recovery activities are considered outages (zero tolerance for recovery actions). This perspective captures the proportion of time the system is in normal operating mode, S_0, and may be of special interest to system administrators – $SS_{avail}(admin)$.

2. Tolerance availability, here a subset of the recovery actions result in outages that are above a specific tolerance threshold, e.g, for the microrebootable application server, recovery actions associated with eBid restarts, JVM/JBoss restarts and Operator restarts may be considered above the tolerance threshold, while restarts to the EJB or WAR levels may be considered tolerable. This perspective captures the proportion of time that end-users can receive service, $\{ S_0, S_1, S_2 \}$, and may be of special interest to end-users/clients of the system – $SS_{avail}(client)$.

3. Capacity-oriented availability, captures how much service the system is delivering. Whereas in [20] microreboots are presented as a means of improving the availablility of web application servers, they are not perfect. During a microreboot (or more expensive system reboot) some client requests are lost – 78 requests per fine-grained restart and 3917 requests per coarse-grained restart[7]. Using these numbers we can estimate the percentage of requests lost during an interval as compared to the total number of requests that could be serviced during that same interval – $SS_{avail}(capacity)$.

Basic Steady-state Availability ($SS_{avail}(admin)$). The basic steady-state availability of the system from the administrator's perspective (i.e., zero tolerance for restarts) during an interval T is given by:

$$UP_{admin} = \{S_0\}, DOWN_{admin} = \{S_1, S_2, S_3, S_4, S_5\} \quad (4.12)$$

$$SS_{avail}(admin) = T * \pi_0$$

$$SS_{downtime}(admin) = T * (\pi_1 + \pi_2 + \pi_3 + \pi_4 + \pi_5) \quad (4.13)$$

During an interval of 1 day (1440 minutes), we expect the microrebootable application server to be UP for 1435.86 minutes and DOWN for 4.14 minutes from the administrator's perspective (Equation 4.14).

$$SS_{avail}(admin) = T * \pi_0 \quad (4.14)$$

$$\rightarrow SS_{avail}(admin) = 1440 * 0.997127 = 1435.86\ minutes$$

$$SS_{downtime}(admin) = 1440 * (1 - 0.997127) = 4.14\ minutes \quad (4.15)$$

[7] See [20] Figure 1.

Tolerance Availability ($SS_{avail}(client)$). The tolerance availability of the system (basic steady state availability from the client perspective) during an interval T is given by:

$$\begin{aligned} UP_{client} &= \{S_0, S_1, S_2\}, DOWN_{client} = \{S_3, S_4, S_5\} && (4.16) \\ SS_{avail}(client) &= T * (\pi_0 + \pi_1 + \pi_2) \\ SS_{downtime}(client) &= T * (\pi_3 + \pi_4 + \pi_5) && (4.17) \end{aligned}$$

During an interval of 1 day (1440 minutes), we expect the microrebootable application server to be UP for 1439.83 minutes and DOWN for 0.17 minutes from the client's perspective (Equation 4.18).

$$\begin{aligned} SS_{avail}(client) &= T * (\pi_0 + \pi_1 + \pi_2) && (4.18) \\ \rightarrow SS_{avail}(client) &= 1440 * 0.999883 = 1439.83\ minutes \\ SS_{downtime}(client) &= 1440 * (1 - 0.999883) = 0.17\ minutes && (4.19) \end{aligned}$$

Capacity-oriented Availability. The capacity-oriented availability of the microrebootable application server, $SS_{avail}(capacity)$ during an interval T is calculated using:

- The number of failures during the interval, F_T.
- The number of requests serviced during the interval, r_T; this is a function of the throughput of the application server.
- The percentage of failures handled via fine-grained recovery actions, p_{fgr}.
- The number of requests lost as a result of fine-grained recovery actions, r_{fgr}.
- The percentage of failures handled via coarse-grained recovery actions, $p_{cgr} = (1 -$

p_{fgr}).

- The number of requests lost as a result of coarse-grained recovery actions, r_{cgr}.

And is given by:

$$SS_{avail}(capacity) = 1 - \frac{F_T * \Big((p_{fgr} * r_{fgr}) + (p_{cgr} * r_{cgr})\Big)}{r_T} \tag{4.20}$$

Letting S_1 and S_2, EJB restarts and WAR restarts respectively, represent fine-grained recovery actions and S_3, S_4 and S_5, eBid restarts, JVM/JBoss restarts and Operator fixes, respectively, represent coarse-grained recovery actions, during the interval of 1 day (T = 86,400,000 msecs):

- $F_T = \frac{3}{600000} * 86,400,000 = 432$ failures per day
- $r_T = 70\ requests\ per\ sec$[20] $* 86,400\ secs = 6048000\ requests\ per\ day$
- $p_{fgr} = 99.74\%$ (see failure/fault coverage equation, Equation 4.22)
- $r_{fgr} = 78$ requests
- $r_{cgr} = 3917$ requests

Therefore:

$$SS_{avail}(capacity) = 1 - \frac{432*\Big((0.9974*78)+((1-0.9974)*3917)\Big)}{6048000} = 0.993716 \tag{4.21}$$

4.3.5 Serviceability Measures

Serviceability measures capture the impacts of system failures and/or remediation activities. Whereas reliability and availability have more rigorous mathematical definitions [72],

serviceability is less well defined. For our evaluation purposes, when we discuss the serviceability of a system we are specifically interested in quantifying the impacts of failures (which may be expressed using a variety of metrics – e.g., monetary or time penalties) as well as the efficacy of remediation mechanisms – overall success and coverage of remediation mechanisms.

For the microrebootable application server, we identify four metrics that can be used to evaluate its serviceability properties:

1. The fault/failure coverage of the system. This is the percentage of failures for which the system has an acceptable response.

2. The mean time to system restoration. This is the expected number of time units needed to restore the system to its normal/original operating condition, S_0.

3. The expected penalties associated with outages (downtime). These quantify the negative consequences of outages and may be expressed in terms of money spent/lost due to system un-availability (e.g., paying for SLA violations, or lost revenue due to the system being down) or some other suitable metric.

4. The SASO (stability, accuracy, settling time and overshoot) properties of the Recovery Manager's operation. This describes the operation of the Recovery Manager using a simple feedback control loop, which we can analyze using traditional control theory tools.

Fault/Failure Coverage. The fault/failure coverage for the microrebootable application server during an interval T is the percentage of failures for which there is an acceptable response. In our analytical evaluation, failures handled in S_1 or S_2 are considered acceptable since these result in minimal disruptions to the end-users of the system [20].

The fault/failure coverage of the system during an interval T is a function of the number of

failure escalations. In 1 day (T = 86,400,000 msecs), 432 failures are injected, of which we expect 99.74% to be handled in S_1 or S_2 (Equation 4.22).

$$\frac{432 - (F_{war \rightarrow eBid} + F_{eBid \rightarrow JVM/JBoss} + F_{JVM/JBoss \rightarrow Operator})}{432} * 100\% = 99.74\% \quad (4.22)$$

Mean Time to System Restoration (MTTSR). The MTTSR is the average number of time units required to return the system to its original/normal operating mode. Using SHARPE we calculate this to be 576.177875 msecs.

Expected Downtime Penalties. The total expected downtime penalty is a function of the total number and duration of outages experienced by the system during an interval T. These penalties can be calculated based on any availability guarantees that govern the system's operation. Table 4.5 gives the downtime allowance per day (in minutes) based on the number of 9's availability guaranteed. Using this table and $SS_{avail}(client)$ calculated earlier we can express the expected downtime penalties.

Availability guarantee	Max downtime per day	Expected penalties
99.999	~ 0.0144 mins	(0.17 - 0.0144)*$p
99.99	~ 0.1440 mins	(0.17 - 0.1440)*$p
99.9	~ 1.4400 mins	$0
99	~14.4000 mins	$0

Table 4.5: Expected downtime penalties using Microreboots

Stability of the Microreboot Recovery Manager. To reason about the stability of the Recovery Manager's operation we propose the feedback control diagram shown in Figure 4.5. In our diagram a controller takes a recovery time deadline as input (e.g., the MTTSR calculated earlier – ~ 576 msecs) and monitors how well the Recovery Manager is able to meet that deadline (i.e., measured output equals the actual MTTSR). Based on the deviations between the desired MTTSR and the actual MTTSR the controller can vary the control

input. Possible choices for control inputs include, but are not limited to: the rate at which failure events are routed to (or admitted by) the Recovery Manager for resolution via a microreboot or the percentage of failure events dispatched to the Recovery Manager for resolution via microreboot. Spikes in failure event arrival rates *and* accompanying increases in the measured MTTSR may signal that these failures may need to be resolved by other means and/or should be brought to the attention of a human operator.

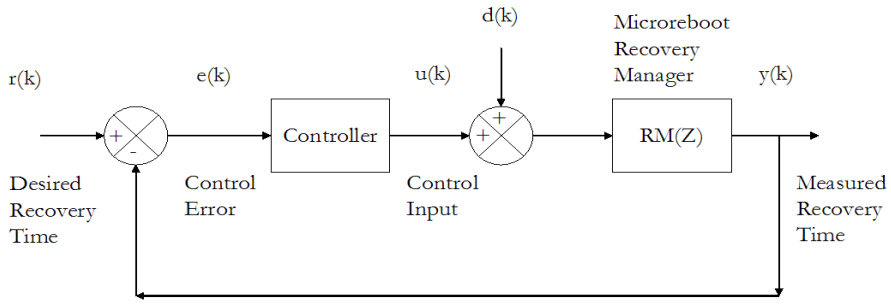

Figure 4.5: Microreboot Recovery Manager feedback control diagram

Other Types of Analyses. In addition to evaluating (Markov and Control Theory) models of the system to quantify various reliability, availability and serviceability properties, there are three other types of analyses that can be performed on these models: sensitivity analysis, tradeoff analysis and specification determination.

- Sensitivity analysis – looks at how the analysis results change if one or more of the input parameters change [72]
- Tradeoff analysis – investigates how trading off a change in one input parameter for another affects the analysis results [72].
- Specification determination – determines the values of given input parameters required to meet a specific reliability, availability or serviceability goal.

4.3.6 Analysis Results

Table 4.6 summarizes the analysis results for the microrebootable application server RAS model shown in Figure 4.4 based on the model parameters in Table 4.2.

Measure	Metrics	Results
Reliability	Failure escalations per day ($F_{a\rightarrow b}$)	22.671386
	Frequency of recovery activities per day (F_2)	4.14 mins
	Frequency of outages per day (F_3)	0.056537
	Frequency of recovery actions per day > 1 sec (F_4)	70.578179
Availability	Basic steady-state availability ($SS_{avail}(admin)$)	0.997127
	Tolerance availability ($SS_{avail}(client)$)	0.999883
	Capacity-oriented availability ($SS_{avail}(capacity)$)	0.993716
Serviceability	Fault/failure coverage	99.74%
	Mean-time to system restoration (MTTSR)	576 msecs
	Expected downtime penalties per day (4 9's)	(0.17 - 0.1440)*$p

Table 4.6: Summary of Microreboot RAS model analysis results

4.4 Related Work

The analytical tools – Continuous Time Markov Chains (CTMCs), Markov Reward Networks and Feedback Control models – and techniques for their analysis have been well studied and used by others to study many aspects of computing system behavior.

[69], [101] and [121] provide a rigorous discussion of the mathematical principles (probability theory and queuing theory) underlying Markov chains and Markov reward networks as well as techniques for their analysis and solution. [69] and [101] specifically provide numerous examples of applying Markov chains to study the performance, reliability and availability characteristics of computing systems.

[98] and [2] discuss techniques for the computationally tractable analysis and solution of Markov models. These techniques are available in the SHARPE [160] RAS modeling tool, which we use in the construction and analysis of RAS models.

Markov chains have been used in the study and analysis of dependable and fault-tolerant systems and the techniques used to realize them. Examples include analyses of RAID (Redundant Arrays of Inexpensive Disks) [114] and telecommunication systems [101]. They have also been used in the study of software aging [13] and in evaluating the efficacy of preventative maintenance (software rejuvenation). Dependability is concerned with assessing the ability of a system to deliver its intended level of service to its users especially in the presence of failures which impinge on its level of service [107]. There are three of the dependability measures of interest: reliability measures, availability measures and task completion measures [72] – task-completion is the likelihood that a task will be completed satisfactorily. [72] also discusses four types of analyses that can be performed – model evaluation, sensitivity analysis, tradeoff analysis and specification determination. In our construction and analysis of RAS models we employ select reliability and availability measures. Further, whereas sensitivity analysis, tradeoff analysis and specification determination are discussed in [72] with regard to Markov chains, these types of analyses can also be applied to models constructed using other modeling formalisms. As a result, we can employ these analyses as part of the RAS evaluation process.

Performability [120] provides unified measures for considering the performance and reliability of systems together. Markov reward networks [69] have been used as a formalism for establishing this link between the performance of a system and its reliability. Other formalisms used in performability analysis include Stochastic Petri Nets (SPNs) [101] and Stochastic Activity Networks (SANs) [161], which are both built on top of Markov chains. SPNs and SANs allow for more detailed and sophisticated modeling of a system's operation, e.g., modeling concurrent activities in a system. In our construction of RAS models, we use Markov reward networks to quantify the impacts of failures and/or remediation activities. Further, our goal is to develop simple, reusable models templates that can be used for describing failure scenarios and scoring system responses, rather developing a detailed model of the operation of underlying system being evaluated.

Different classes of failures have been studied using Markov chains including independent failures [101], near-coincident failures [42] and cascading failures [91]. Leveraging these analytical tools in our construction of RAS models allows us to describe failure scenarios that represent these different classes of failures.

Feedback control has been used in the development of adaptive computing systems providing mathematical tools for constructing predictable systems [90]. For example, [139] uses control theory to develop a fluid model for network traffic management. [145] presents a database server that adaptively throttles administrative utilities when necessary to maintain a given level of query performance (an example of disturbance rejection in a control system) while work in [45] describes the use of feedback control to automatically adjust the size of memory pools to balance the resource demands in a database management system (an example of regulatory control/regulation). Finally, [44] presents the principles of feedback control and discusses the implications for realizing self-managing systems which exhibit the desirable properties of stability, accuracy, short settling times and avoiding overshoot (SASO) with respect to the policy-based objectives that govern their operation. In constructing RAS models of systems we are interested primarily in applications of regulatory control and assessing the SASO properties of systems and their failure handling mechanisms.

4.5 Summary

This chapter introduced the analytical tools and techniques used to construct RAS models – Continuous Time Markov Chains, Markov Reward Networks and Feedback Control Models. In §4.2 we provide background information on these analytical tools. And §4.3 discusses the measures and metrics of reliability, availability and serviceability while providing an example RAS model and analysis of the microrebootable application server described in [20].

Using our analytical example we demonstrate the construction of a basic RAS model that can be used to a) describe failure scenarios used to evaluate a microrebootable application server and b) score/evaluate the application server's responses to injected faults. In conducting our analysis we identify a number of reliability, availability and serviceability metrics that may be used in system evaluations and illustrate how they are derived and calculated.

Our analysis is complementary to the measurement-based evaluation done in [20], considering other aspects of reliability, availability and serviceability not covered in the original work, e.g., reasoning about the frequency of failure escalations, recovery activities and outages; presenting three perspectives on availability for comparison – basic steady-state availability, tolerance availability and capacity-oriented availability; and finally discussing fault/failure coverage, mean-time to system restoration and estimated downtime penalties for the microrebootable application server.

In the next chapter we combine runtime fault-injection tools, including Kheiron, which was described in Chapter 3, with the RAS modeling tools described in this chapter to develop the 7U-Evaluation Benchmark – a model-based and measurement-based reliability, availability and serviceability benchmark for web-application stacks and their components.

Chapter 5

The 7U-Evaluation Benchmark

In this chapter we present a methodology for evaluating the RAS characteristics of N-tier web application stacks and their components – the 7U-Evaluation method – and demonstrate its effectiveness via three case studies measuring the RAS properties of different deployments of the TPC-W web-application [119].

The 7U-Evaluation Benchmark is a model-based and measurement-based evaluation approach that combines runtime fault-injection tools, Chapter 3, with analytical RAS models, that describe and score specific failure scenarios (Chapter 4).

In our experiments we subject different TPC-W deployments to the same failure conditions, develop RAS models to describe and score the failure scenarios, and conduct fault-injection experiments to obtain values for the RAS model parameters. Based on the data collected from the fault-injection experiments, we compute, compare and discuss the RAS metrics for each deployment.

In the sections that follow we discuss the challenges of RAS benchmarking, the design considerations for the 7U-Evaluation Benchmark that address those challenges, present our experimental results, and compare our approach to traditional performance benchmarking approaches as well as similar efforts to benchmark aspects of reliability, availability and ser-

viceability in the fault-tolerant computing, dependable computing and autonomic computing communities.

5.1 Introduction

The importance placed on realizing reliable, highly available and serviceable (easy-to-manage/self-managing) software systems necessitates approaches for evaluating the reliability, availability and serviceability (RAS) characteristics of systems [113, 118, 102, 3]

Benchmarks provide a structured way to evaluate systems by allowing interested parties "*...to measure well-defined features of a system or component according to an agreed ... set of methods and procedures* [113]. In assessing the RAS characteristics of systems, we wish to identify or develop methods and procedures that quantitatively capture: a) the impacts of faults or failures on a system's reliability, availability and serviceability and b) the efficacy of any remediation mechanisms.

In order to conduct an RAS benchmark, it is necessary to have an environment and tools that allow the system under test to be exposed to failure-provoking stimuli [16]. Direct fault-injection into components of the system under test is the primary technique that enables such an environment [8]. An important part of the RAS evaluation process is to inject faults that exercise any remediation mechanisms that the system under test has or that highlight RAS deficiencies. The determination of which faults meet this criteria depend on a) the system or class of system being evaluated and b) problems that have been observed and/or are currently being studied.

Another important element of an RAS benchmark is the generation of realistic workloads for the system under test. This allows us to study the impact of failures and other stressful conditions (the fault-load) on the typical operation of the system. Generating realistic workloads for systems is a difficult problem; however, RAS evaluations can build upon

existing performance benchmarks, which provide excellent sources of workloads, e.g., benchmarks produced by the Standard Performance Evaluation Corporation (SPEC) [177], the Transaction Processing Performance Council (TPC) [190] and the National Institutes of Standards and Technology (NIST) [140].

The use of performance benchmarks to provide realistic workloads during an RAS evaluation influences the metrics that are collected as well as the way these metrics are used in scoring the system under test. The performance metrics collected can be used to reason about complete outages or degraded modes of operation, which result from injecting faults or inducing failures in the test system. In §4.3.6 we illustrate how variations in performance metrics can be used/incorporated to reason about different facets of reliability, availability and serviceability, e.g., basic availability vs. tolerance availability vs. capacity-oriented availability, which take different operating modes of the system into consideration. The metrics used to express these facets of reliability, availability and serviceability and the RAS models used to compute them specify the scoring criteria for an RAS evaluation and describe the failure scenarios that the system under test is subjected to during an evaluation.

5.2 The 7U RAS Benchmarking Methodology

In the previous section we describe the conceptual elements needed to build RAS benchmarks: a system under test, a testing environment and tools that support fault-injection, realistic workload generators and a set of scoring criteria/failure scenario descriptions. In this section we combine these elements into a methodology for performing RAS evaluations. Our methodology is an extension of measurement-based dependability benchmarks [96]. It consists of seven steps:

1. **Specify fault-model**: first a fault-model is developed that mimics faults, failures and stressful conditions previously seen or likely to be seen in practical deployments of the

system under test. The fault-model codifies the set of problems the system is expected to detect, diagnose and/or repair. During this stage of the methodology fault-injection tools that can reproduce the faults and failures of interest are identified or developed.

2. **Specify the fault-remediation relationship**: injecting faults in the system should elicit a response from the system, e.g., triggering an existing remediation or compromising a measure of interest. In the case of the latter, variations in the compromised measure of interest can be used to build or improve detection or repair mechanisms in the system.

3. **Decide on micro-measurements for remediations**: these are metrics collected from the remediation mechanisms, e.g., remediation success, remediation times, etc.

4. **Decide on macro-measurements and create scoring models for system evaluation**: macro-measurements are the measures of interest, e.g., different facets of reliability, availability and serviceability. During this step RAS models used for scoring are developed. These scoring models describe different failure scenarios, establishing a link between the micro-measurements of the remediations – success, coverage, recovery times, etc. – and the reliability, availability and serviceability measures.

5. **Develop workload and metric collectors**: during a benchmark run the system under test is subjected to a workload to simulate actual use of the system and allow for the collection of metrics from the system related to its performance, failure behavior and/or remediation activities.

6. **Run benchmarking and fault-injection experiments**: the overall benchmark study consists of multiple experiments, each involving one or more failure scenarios where the system under test is subjected to the workload and a faultload.

7. **Analyze results and revise scoring models**: results from the benchmark runs are

analyzed and scored, after which the existing scoring models may be refined or new models that describe new failure scenarios added.

The RAS benchmarking methodology outlined above presents a number of practical challenges: selecting reasonable or representative faults, representative workloads, reproducibility of results and portability to different systems, identifying metrics used for scoring the responses of the system under test, and collecting data from benchmark runs used as parameters in scoring models. In the next section we discuss how we addressed these challenges as we develop an RAS benchmark for N-tier web-applications and their components.

5.3 RAS Benchmarking Challenges

5.3.1 Selecting reasonable or representative faults

The first practical challenge in developing an RAS benchmark is to specify the fault-model under consideration. The fault-model follows from the system under test (or class of system under test). Based on the system under test and/or the class of system under test, published studies on problems/failures experienced in the field may be used to guide the creation of the fault-model. To conduct an RAS evaluation the fault-model may have to be appropriately scoped/restricted to the set of problems/failures that can be reproduced using accessible fault-injection tools.

In our first application of this RAS benchmark we use N-tier web-applications as the class of system under test and the TPC-W web-application as a specific test subject. We chose N-tier web applications as our class of systems under test due to their ubiquity. They have standardized components – web server, application server, database server and operating system and a well-understood client-server workload model. Further, we were able to find published studies about problems/failures experienced by N-tier applications

and their components [21, 31, 142, 179], which guided the identification and development of fault-injection tools and provided some ideas for metrics (see §5.3.4).

The TPC-W web-application specification was developed by the Transaction Processing Performance Council (TPC) [190] as part of a benchmark for e-commerce sites. It mimics the activities of an online bookstore in a controlled environment and reference implementations of the benchmark specifications can be found online, e.g., the Java-based implementation from the PHARM research team at the University of Wisconsin-Madison [147], which we use in our experiments (see §5.4, §5.5 and §5.6). Alternative web-applications, e.g., RUBiS [141], which mimics an online auction website like eBay, or the SPEC jAppServer [176], which emulates information flow among an automotive dealership, manufacturing, supply chain management and an order/inventory system could have also been used.

Our fault-model for the TPC-W web application stack 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 [31]; similarly Microsoft's analysis of crash dumps submitted by Windows XP users to their Online Crash Analysis website attribute 70% of crashes to faults or failures in third-party device drivers [115] – faulty device drivers easily compromise the integrity and reliability of the kernel, 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 [21].

We identified the operating system and the application server as candidate targets for fault-injection. Given the operating system's role as resource manager [185] and part of the native execution environment for applications [63], its reliability is critical to the overall stability of the applications it hosts. Similarly, application servers act as containers for web-applications and are 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. In our experiments we target Java-based application servers due to the availability of a number of open-source/free alternatives including Caucho Technology's Resin [186] application server and The Apache Foundation's Tomcat [53] application server.

With respect to remediations for these faults, there are a number of possibilities. For device driver failures operating system kernels may crash, which is a structured way of recording a problem and halting the system to prevent further damage [115], or employ some form of device driver recovery, e.g., Nooks [122], hardened device drivers on OpenSolaris [136][1].

To address resource leaks/memory leaks, application server restarts have been used to react to low memory conditions resulting from memory leaks or as preventative maintenance to avoid low memory conditions (rejuvenation) [21]. Other approaches combine rejuvenation with redundancy/load-balancing, e.g., VM-Rejuv [173], to mitigate the effects of memory leaks.

In evaluating the RAS properties of N-tier web-application deployments we investigate the efficacy of these remediation mechanisms.

5.3.2 Representative Workloads

The workload used to exercise the system during an RAS evaluation must be representative of realistic uses of the system [113] to provide insights into the impacts of faults and failures on its operation.

The reference implementation of the TPC-W provided by the PHARM research team at the University of Wisconsin-Madison includes a workload generation tool that emulates browser clients for the TPC-W web-application. Remote Browser Emulators (RBEs) act

[1] Programming-language extensions e.g. SafeDrive [198] are also used to develop recoverable device drivers.

like users sending requests to the TPC-W web-application, interacting with it according to one of three strategies (mixes) outlined in the TPC-W benchmark specification – Browsing mix, Shopping mix or Ordering mix. Different mixes control the proportion of activities that involve a specific page of the TPC-W web-application (see Table 1 in [119] for further details on the interaction strategies).

5.3.3 Reproducibility and Portability

Conducting RAS evaluations involves introducing changes into systems, e.g., injecting faults or inducing failures to evaluate the system's response (or lack thereof). However, individual changes must be introduced in a manner that is reproducible on a specific system and possibly reproducible across different systems if cross-system comparisons are necessary [3]. In the case of cross-system comparisons an RAS benchmark must be portable if it is to be used across different platforms. Reproducibility allows evaluations to be repeated, which can improve the confidence in the measurements [113].

For reproducibility and portability of our benchmark we use target systems and fault-injection tools that can be deployed on multiple platforms.

Our Kheiron/JVM (§3.8) fault-injection tool can be used on any Java-based application, which allows us to target a variety of Java-based application servers and/or Java-based web-applications.

For device driver fault-injection we target the network device drivers on Linux 2.4.18, 2.6.20 and OpenSolaris operating systems. We use a version of the Nooks SWIFI device driver fault-injection tools [122, 123] developed at the University of Washington for device driver fault-injection on Linux 2.4.18. We developed a port for these tools that injects faults into device drivers on Linux 2.6.20

The Linux (2.4 and 2.6) device driver fault-injection tools inject a variety of faults including:

text faults, stack faults and null pointer faults. Faults injected by the Linux device driver fault-injection tools can lead to kernel panics if no device driver recovery services are available or device driver recovery fails.

To evaluate the efficacy of hardened device drivers on OpenSolaris we use the device driver hardening test harness provided by Sun Microsystems [136] to inject faults into network device drivers. The test harness operates at the level of data accesses, intercepting data accesses of the device driver and injecting faults into the device driver, e.g., corrupting data and interfering with interrupts, simulating faults that occur in the hardware managed by the device driver. The corruptions performed by the test harness can lead to device driver crashes, hangs and/or kernel panics if no device driver recovery services are available or if device driver recovery is unsuccessful.

Device driver fault-injection is inherently not very portable across different operating system kernels, e.g., Windows, Linux and OpenSolaris, since these kernels differ significantly in the data structures, components/kernel-objects and component-interactions used to implement resource management functions, I/O, etc. [115, 39, 155]. Further, porting between major versions of a specific operating system kernel, e.g., Linux 2.4.x and Linux 2.6.x may require changes to some of the underlying mechanisms used to support driver fault-injection tools. For example, accommodating changes to kernel data structures used to represent modules and processes between Linux 2.4.x and Linux 2.6.x.

5.3.4 Metrics and Scoring

Metrics and scores for an RAS evaluation may be based on *direct measurement* or *calculation* using modeling [113]. The ability to obtain direct measurements is dependent on the availability of system observation points. Facilities may exist in the system to collect direct measurements, e.g., logs, or the system may need to be instrumented to facilitate data collection. Calculating reliability, availability and serviceability metrics require modeling.

These models in turn require input from field experience, e.g., fault probabilities and fault distributions, which may be difficult to estimate. Overall, the scores obtained should account for the impact of failures and/or responses on different users (administrators and clients) and the efficacy of the response.

In our 7U Evaluations, rather than estimate fault properties and distributions, we use fault-injection tools to control these parameters – estimates of fault properties and distributions based on field-data from failures can, however, inform and improve the fault-mixes used in fault-injection experiments. We induce failure conditions in the system under test and collect data on a number of aspects of system operation, including the occurrence of outages and degradation events, downtime, and the speed, coverage and success of remediations. We use this data in our scoring models to estimate RAS properties for a specific scenario.

It is accepted that whereas fault-injection is a powerful tool for validating and evaluating remediation mechanisms in systems [191, 32], it cannot predict actual availability or mean time between failure (MTBF) [195, 77]. However, the goal of our 7U benchmark is not to predict MTBFs and MTTFs in absolute terms [2], but rather to provide a framework for 1) reproducing and studying specific failures in systems leading to better fault-injection tools; 2) validating the remediation mechanisms available in a system or reasoning about yet-to-be-added mechanisms and; 3) providing a consistent method of scoring, using simple, reusable RAS models as templates, which capture failure impacts and/or remediation activities to be evaluated from the different perspectives of interest.

[179, 142, 14] present some specific metrics that may be used for N-tier web applications and internet services, including frequency of outages, frequency of degradation events, downtime from the perspective of the client, downtime from the perspective of the IT operators and lost revenue, while §4.3.2 discusses other RAS metrics that may also be of

[2] Predicting MTTFs for software is inherently difficult due to the lack of physical laws governing the operation software components/systems[21], unlike for hardware/physical systems, whose operation is governed by the laws of physics. However, MTTF predictions for hardware have recently been called into question [163].

interest, e.g., frequency of failed remediations, basic steady-state availability, tolerance availability, capacity-oriented availability, fault-coverage, mean time to system restoration and expected downtime penalties.

5.4 Evaluation Part 1

In our first evaluation case-study we model, evaluate and compare two deployments of the TPC-W web-application subjected to memory leaks and device driver faults, which target the application server and operating system, respectively.

5.4.1 7U Process

System under test. For the system under test we use the TPC-W web-application stack. The TPC-W stack consists of a (Java-based) TPC-W web-application implementation, the Resin web/application server [186] and the MySQL [4] database server co-located on a single operating system instance[3].

The two deployments used and compared in our evaluation are shown below:

1. Resin 3.0.22, MySQL 5.0.27, Linux 2.4.18
2. Resin 3.0.22, MySQL 5.0.27, Linux 2.6.20

Fault model. The Resin application server uses automatic restarts to react to memory leaks, while the device driver recovery framework Nooks protects the Linux 2.4.18 kernel from the effects of device driver failures. In our evaluations we exercise these remediation mechanisms and compare against a stack deployed on Linux 2.6.20, which does not have device driver recovery.

[3] We use virtual machines for each deployment.

Memory-leak failures in the TPC-W web-application classes hosted in Resin are induced using Kheiron/JVM [63] and the approach described in §3.8.6. Device-driver faults are injected into the network device driver (the pcnet32 ethernet device driver) using the Nooks SWIFI (SoftWare Implemented Fault-Injection) tools [122], which were originally developed for Linux 2.4.18 by Michael Swift et al. [122] and ported, by us, to Linux 2.6.20.

Fault-remediation relationship. Resin initiates an application server restart under memory pressure, while the operating system employs device driver recovery (where possible) or crashes to protect the system.

Micro-measurements. For micro-measurements we collect metrics on application-server restart times, device driver recovery times, device driver recovery success rates, operating system restart times and client-side goodput.

Macro-measurements. For macro-measurements we use the four node, five parameter model shown in Figure 5.1, with parameter descriptions shown below, to describe and score the failure scenario used in our evaluation. We use this scoring model to quantify the following facets of reliability, availability and serviceability:

- Reliability – frequency of outages
- Availability – tolerance availability
- Serviceability – expected downtime penalties

The construction of this scoring model is described in §5.4.2 and it is a generalization of the RAS model shown in Figure 5.7.

The model consists of four states and six parameters:

- S_0 - System working normally.

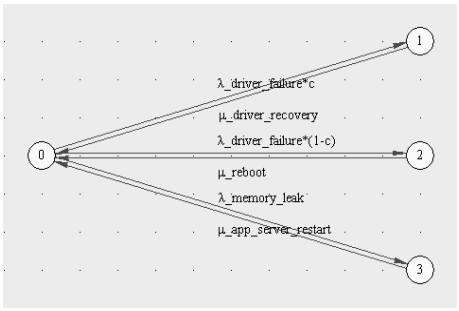

Figure 5.1: Failure scenario scoring RAS model

- S_1 - System recovering a failed device driver.
- S_2 - Device driver recovery failed and system needs to be rebooted.
- S_3 - Application server restart due to memory exhaustion.
- $\lambda_{driver\_failure}$ – forced rate of device driver failures.
- $\mu_{driver\_recovery}$ – mean time for device driver recovery.
- c – the *coverage factor*, success rate of device driver recovery, this allows us to consider imperfect recovery scenarios.
- μ_{reboot} – mean time to reboot the system if device driver recovery is unsuccessful.
- λ_{memory_leak} – observed rate of memory-leak related failures.
- $\mu_{app\_server\_restart}$ – worst-case restart application-server under low-memory conditions.

The goal of our experiments is to inject faults into specific components of the system under test and study its 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.

In our experiments we force/set the rate of memory-leak failures to 1 every 8 hours and the rate of device driver faults is set to 4 every 8 hours. These rates were chosen arbitrarily,

and are used to illustrate the modeling and evaluation process; however, additional data on failure mixes can improve similar evaluation efforts.

Whereas our 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 us to explore the expected and unexpected system responses under stressful fault conditions, much like performance benchmarks subject the system under test to extreme workloads.

5.4.2 Deployment 1: Resin, MySQL, Linux 2.4.18

In Deployment 1 our test platform uses VMWare GSX virtual machines configured with: 512 MB RAM, 1 GB of swap, a single x86 processor and an 8 GB harddisk running Redhat 9 on Linux 2.4.18. 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 [119] as their web-interaction strategy. User-interactions are simulated 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.

There are three remediation mechanisms we consider: (manual) system reboots, (automatic) application server restarts, and Nooks device driver protection and recovery [122] – 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).

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.

A typical fault-free run of the TPC-W (**Configuration A**), takes ~24 minutes to complete and records 3973 successful client-side interactions (166 client-side interactions per minute).

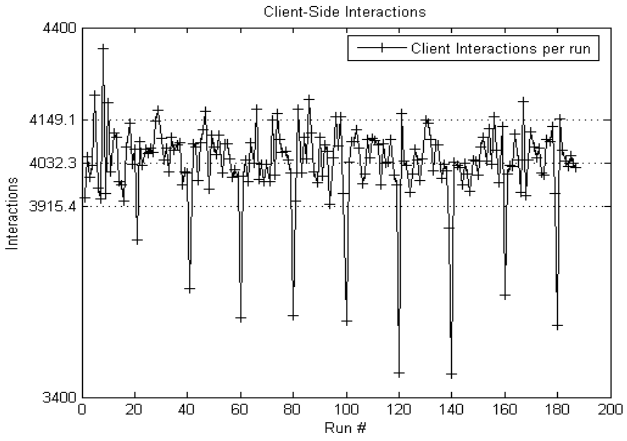

Figure 5.2: Client interactions – Configuration B

Figure 5.2 shows the client-side goodput over ~76 hours of continuous execution (187 runs) in the presence of an accumulating memory leak – **Configuration B**. The average number of client-side interactions over this series of experiments is 4032.3 ± 116.8473. In this figure there are nine runs where the number of client interactions is 2 or more standard deviations below the mean. Client-activity logs indicate a number of successive failed HTTP requests over an interval of ~1 minute during these runs. Resin's logs indicate that the server encounters a low-memory condition, forcing a number of JVM garbage collections before

restarting the application server. During the restart, requests sent by RBE-clients fail to complete. A Poisson fit of the time-intervals between these nine runs at the 95% confidence interval yields a hazard rate of 1 memory-leak related failure (Resin restart) every 8.1593 hours.

Figure 5.3 shows a trace sampling the number of client interactions completed every 60 seconds for a typical run, (Run #2), compared to data from some runs where low memory conditions cause Resin to restart. Data obtained from Resin's logs record startup times of 3,092 msecs (initial startup) and restart times of approximately 47,582 msecs.

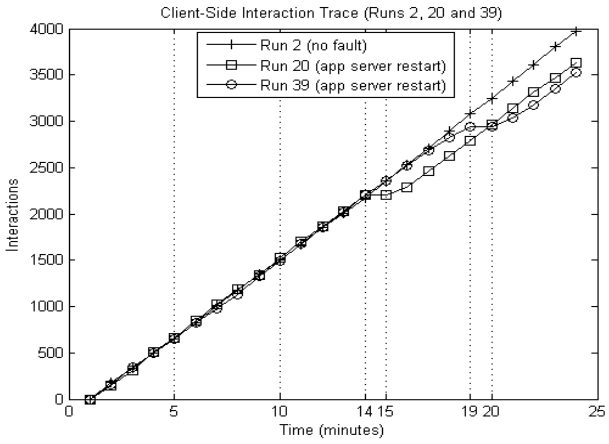

Figure 5.3: Client-side interaction trace - Configuration B

To evaluate the RAS-characteristics of the system in the presence of the memory leak, we use the SHARPE RAS-modeling and analysis tool [160] to create the basic 2-node, 2-parameter RAS-model shown in Figure 5.4. Table 5.1 lists the model's parameters.

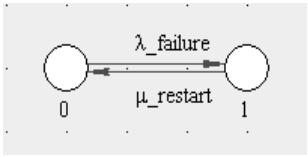

Figure 5.4: Simple RAS model

Whereas the model shown in Figure 5.4 implicitly assumes that the detection of the low memory condition is perfect and the restart of the application server resolves the problem

S_0	an UP state where the system services requests
S_1	a DOWN state, no client requests are serviced while the application server is being restarted
$\lambda_{failure}$	observed rate of failure, 1 failure every 8 hours
$\mu_{restart}$	time to restart the application server, ~47 seconds

Table 5.1: RAS-Model Parameters – Configuration B

100% of the time, in this instance these assumptions are validated by the experiments.

Using the steady-state/limiting availability formula [101]: $A = \frac{\frac{1}{\lambda}}{\frac{1}{\lambda}+\frac{1}{\mu}}$ 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 5.2 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 5.2: 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. For this set of experiments we arbitrarily choose a fault rate of 4 device failures every 8 hours and use the SWIFI tools to achieve this rate of failures in our system under test. The fact that that the remediation mechanism (the reboot) always restores the system to an operational state allows us to reuse the basic 2-parameter RAS model shown in Figure 5.4 to evaluate the RAS-characteristics of the system in the presence of device driver faults. Table 5.3 shows the parameters of the model.

Using SHARPE, we calculate the steady state availability of the system as 98.873%, with an expected downtime of 5,924 minutes per year, i.e., under this fault-load the system cannot

S_0	an UP state where the system services requests
S_1	a DOWN state, no client requests are serviced while the application server is being restarted
$\lambda_{failure}$	achieved rate of failure, 4 failures every 8 hours
$\mu_{restart}$	time to reboot the system, 1 minute 22 seconds

Table 5.3: RAS model parameters – Configuration C

deliver two nines of availability.

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 5.5; its parameters are listed in Table 5.4.

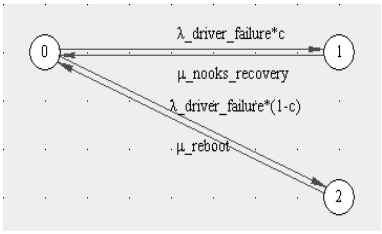

Figure 5.5: RAS model of a system with imperfect repair

S_0	an UP state where the system services requests
S_1	an UP state, where Nooks is recovering a failed driver
S_2	a DOWN state, where Nooks' recovery attempt fails and the system needs to be rebooted
$\lambda_{driver\_failure}$	achieved rate of failure, 4 failures every 8 hours
$\mu_{nooks\_recovery}$	time for Nooks to successfully recover a failed device driver, 4,093 microseconds worst case
c	the *coverage factor*, represents the success rate of Nooks, varying this parameter lets us study the impact of imperfect recovery
μ_{reboot}	time to reboot the system, 1 minute 22 seconds

Table 5.4: RAS model Parameters – Configuration D

Figure 5.6 shows the expected impact of Nooks recovery on the system's RAS-characteristics as its success rate varies.

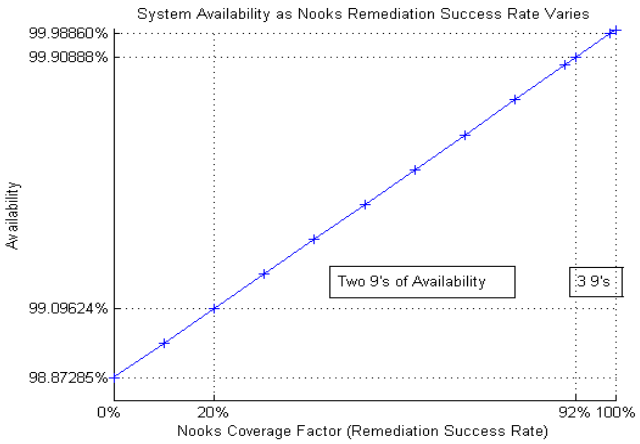

Figure 5.6: Availability – Configuration D

Whereas Configuration C of the system under test is unable to deliver two 9's of availability in the presence of device driver faults, a modest 20% success rate from Nooks is expected to promote the system into another availability bracket while a 92% success rate reduces the expected downtime and SLA penalties by two orders of magnitude (see Figure 5.6) [4].

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 an RAS-model that considers all the faults in our fault-model and the remediations available, **Configuration E** – see Figure 5.7.

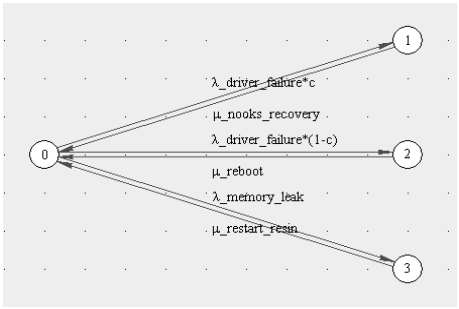

Figure 5.7: Complete RAS-model – Configuration E

Figure 5.8 shows the expected availability of the complete system. The system's availability

[4] 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.

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 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.

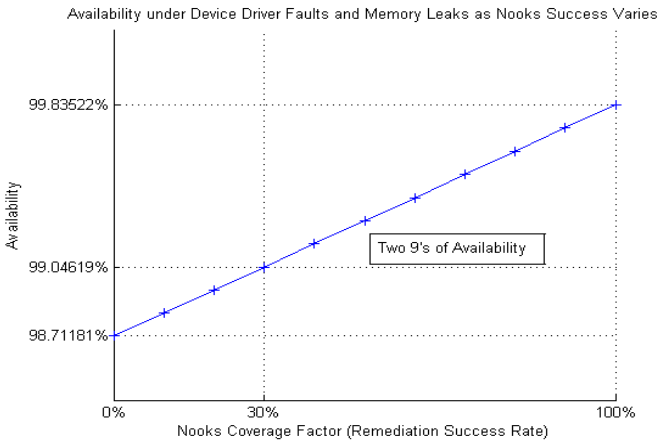

Figure 5.8: Availability – Configuration E

5.4.3 Deployment 2: Resin, MySQL, Linux 2.6.20

In Deployment 2 our test platform uses VMWare GSX virtual machines configured with: 512 MB RAM, 1 GB of swap, a single x86 processor and an 8GB harddisk running OpenSuse 9.2 on Linux 2.6.20. 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 [119] as their web-interaction strategy (the same conditions used in §5.4.2). User-interactions are simulated using the Remote Browser Emulator (RBE) software also implemented at the University of Madison-Wisconsin. Our VMs are hosted on a machine

configured with 2GB RAM, 2 GB swap, an Intel Core Duo E6750 Processor (2.67GHz) and a 228 GB harddisk running Windows XP Media Center Edition SP2.

Unlike Linux 2.4.18 where there is an available device driver recovery framework (Nooks), there is no equivalent device driver protection framework for Linux 2.6.20[5]. As a result there are two remediation mechanisms we consider: (manual) system reboots in the case of device driver crashes and (automatic) application server restarts in the case of memory exhaustion.

Subjecting Deployment 2 to the same failure conditions as Deployment 1 we collect measurements for: 1) the fault-free system operation, 2) the application server restart times in the presence of memory leaks and, 3) the system operation in the presence of device driver failures, and enter these into our scoring model to compare against Deployment 1.

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, and **Configuration D** – System operation in the presence of memory leaks and driver failures.

A typical fault-free run of TPC-W (**Configuration A**) takes 20 minutes to complete and records 3268 successful client-side interactions (163 client-interactions per minute). During our experiments we record an average of 3398 ± 67.7209 successful client-side interactions.

In **Configuration B**, the normal restart time for Resin is 1,499 ms while the restart time under low-memory conditions is 16,117 ms. Using the simple RAS model (Figure 5.4) with parameters $\lambda_{failure} = 1$ every 8 hours and $\mu_{restart} = 16,117\ msecs$ we calculate the steady-state availability for this configuration as 99.944%, with expected yearly downtime of 294 minutes, i.e., this system is able to deliver 3 9's of availability under these conditions.

In **Configuration C** we inject faults into the pcnet32 device driver at the same rate as

[5]We only ported the Nooks device driver fault-injection tools from Linux 2.4.18 to Linux 2.6.20, **not** the Nooks device driver recovery framework.

we did in Deployment 1 – 4 faults every 8 hours. Without device driver recovery, each injected fault leads to a kernel panic requiring a system reboot to restore system operation (1 minute 28 seconds). For this configuration we calculate the steady-state availability for this configuration as 98.781% with expected yearly downtime of 6,407 minutes.

In **Configuration D** we consider the combination of memory-leak related failures and device driver failures. Using the scoring model in Figure 5.1 we calculate the steady-state availability of the full system as 98.725% availability, with expected yearly downtime of 6,700 minutes, i.e., overall we expect this configuration to deliver less than 2 9's of availability under these conditions.

5.4.4 Deployment Comparisons

Table 5.5 compares the performance and RAS characteristics of Deployments 1 and 2. For the failure scenario involving memory leak failures observed once every eight hours and device driver failures injected four times every eight hours (Figure 5.1), Deployment 1 with device driver recovery enabled is better than Deployment 2. Further, modest success rates (30%) for the device driver recovery framework on Deployment 1 are expected to move the configuration into a higher availability bracket (Figure 5.8).

		Deployment 1	Deployment 2
Performance Measures	Interactions/min	166	163
	Normal Resin restart (msec)	3,092	1,499
	Low memory Resin restart (msec)	47,582	16,117
	OS restart (min:secs)	1:22	1:28
	Device driver recovery	4,093 μ *secs*	n/a
RAS Measures	UP states	S_{UP}={S_0, S_1}	S_{UP}={S_0}
	DOWN states	S_{DOWN}={S_2, S_3}	S_{DOWN}={S_2, S_3}
	SS_{avail}(*memory leak only*)	99.838%	99.944%
	SS_{avail}(*driver failure only*) *no recovery*	98.873%	98.781%
	SS_{avail}(*memory leak, driver failure*)	98.712%→99.835%	98.725%
	Downtime(memory leak only)/yr	866 mins	294 mins
	Downtime(driver failure only)/yr *no recovery*	5,924 mins	6,407 mins
	Downtime(memory leak,driver failure)/yr	6,771→866 mins	6,700 mins

Table 5.5: TPC-W Deployment 1 and Deployment 2 Results

Using our scoring model and the experimental results we can identify a weakness in the deployments evaluated – reactive handling of memory leaks – and consider possible mitigations.

Whereas the application server used in the experiments (Resin) automatically restarts itself under low-memory conditions, the reactive strategy built into the application server may be complemented (or superseded) by a preventative maintenance scheme that performs a periodic early restart of the application server.

Preventative maintenance actions may be carried out on a fixed schedule or an adaptive schedule. For preventative maintenance to be an option the system's failure distribution must be hypoexponential (Increasing Failure Rate/IFR §4.2.1), which allows us to divide the system's lifetime into two stages. Further, it must be possible to create detection mechanisms that accurately identify/predict the transition from the first stage of the system's lifetime to the second stage.

We use the RAS-model shown in Figure 5.9 in our analysis. Its parameters are listed in Table 5.6 and are based on the application server restart times of Deployment 1.

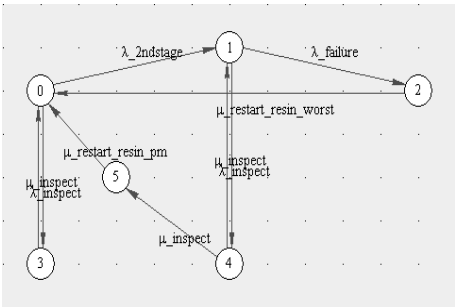

Figure 5.9: Preventative maintenance RAS-model

Using these parameters we plot the graph shown in Figure 5.10, which shows the expected availability of the system as $\lambda_{inspect}$ varies. For the failure conditions/model-parameters supplied performing a free-memory check a modest number of times per hour and performing a preventative maintenance action is expected to improve the system's availability.

S_0	an UP state, 1st stage of system lifetime
S_1	an UP state, 2nd stage of system lifetime
S_2	a DOWN state, application server is restarted
S_3	an UP state, free-memory inspection occurs during the 1st stage of the system's lifetime
S_4	an UP state, free-memory inspection occurs during the 2nd stage of the system's lifetime. A preventative restart is carried out returning the system to the first stage of its lifetime
S_5	a DOWN state, preventative restart occurs
$\lambda_{2ndstage}$	rate of transition into 2nd stage of its lifetime, once every six hours
$\lambda_{failure}$	rate of transition into low-memory condition state, once in either the 7th or 8th hour
$\mu_{restart\_resin\_worst}$	time to restart Resin under low-memory conditions, ~47 seconds
$\lambda_{inspect}$	rate of free-memory trend-checks
$\mu_{inspect}$	time to conduct free-memory check, 21,627 microseconds
$\mu_{restart\_resin\_pm}$	best-case time to restart application, server 3,092 milliseconds

Table 5.6: Preventative maintenance model parameters

5.5 Evaluation Part 2

In our second case-study we model and experimentally evaluate the efficacy of VM-Rejuv – a prototype implementation of a virtual machine (VM) based software rejuvenation scheme for application servers and internet sites [173] developed at the Universitat Politècnica de Catalunya (UPC) in Barcelona.

Software rejuvenation is the concept of gracefully terminating an application and immediately restarting it in a clean internal state [78]. This technique has been implemented as a form of preventative/proactive maintenance in a number of systems, e.g., AT&T billing applications [78][6], telecommunications switching software [10], online transaction processing (OLTP) servers [27], middleware applications [15] and web/application-servers [109], as an approach to mitigate the effects of software aging – the degradation of the state of a software system, which may eventually lead to system performance degradation and/or crash/hang failure [1].

[6]The original proposal of the software rejuvenation technique by Huang et al.

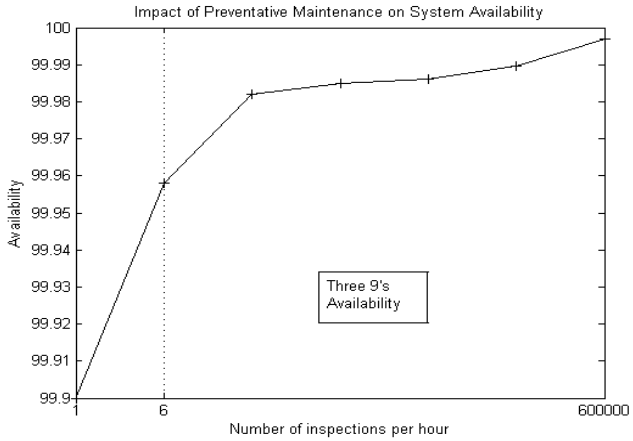

Figure 5.10: Expected impact of preventative maintenance

Strategies for rejuvenation can be divided into two classes: *time-based* rejuvenation and *prediction-based* rejuvenation [1]. With time-based rejuvenation state-restoration activities are preformed at regular deterministic intervals, whereas with prediction-based rejuvenation the time to rejuvenate is based on the collection and analysis of system data, e.g., resource metrics.

State-restoration activities may include one or more of: garbage collection, preemptive rollback, memory defragmentation, therapeutic reboots, flushing and/or reinitializing data structures [27].

VM-Rejuv employs a **prediction-based** rejuvenation strategy for mitigating the effects of software aging and transient failures on web/application-servers. Software aging and transient failures are detected through continuous monitoring of system data and performance metrics of the application-server; if some anomalous behavior is identified the system triggers an automatic rejuvenation action [173]. Rejuvenation actions in VM-Rejuv take the form of **preventative** application-server restarts.

To minimize the disruption to clients due to an application-server restart, VM-Rejuv employs redundancy and load-balancing.

Web-application servers are deployed under VM-Rejuv in multiple virtual machines logically

organized in a cluster. Hosting multiple virtual machines on a single physical machine allows it to be treated like a cluster as shown in Figure 5.11.

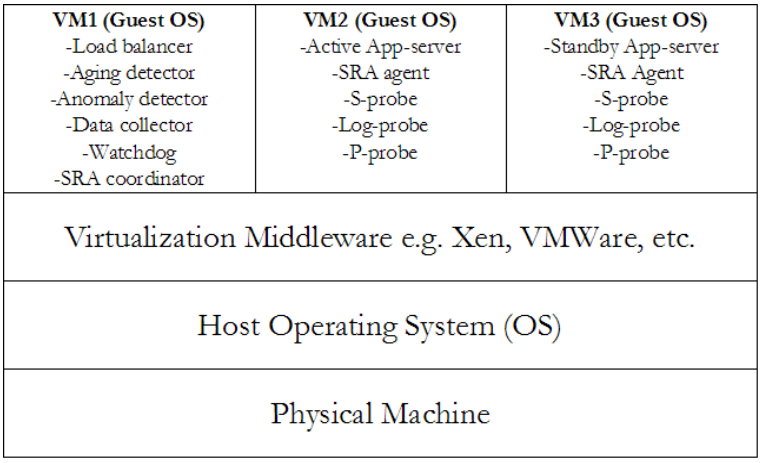

Figure 5.11: VM-Rejuv framework

VM-Rejuv uses three virtual machines for a hosted web-application: one VM to run a software load-balancer (VM1), one VM to be the main/"active" application server and one VM to be a hot-standby replica of the main application server (VMs 2 and 3).

The first virtual machine, VM1, runs:

- A load-balancer – the VM-Rejuv prototype uses Linux Virtual Server (LVS) as its load-balancer [151]. LVS is a layer-4 load-balancer, which provides IP-failover and a number of load-balancing policies (round-robin, weighted round-robin, etc.).
- An Aging detector – module for forecasting aging-related failures. In the current VM-Rejuv prototype the Aging detector uses simple threshold techniques concerned with memory utilization [173].
- An Anomaly detector – module that detects anomalies in VM2 and VM3 using threshold violations as indicators of anomalies, e.g., throughput falling below a preset threshold or response time exceeding a preset threshold (SLA violations).
- A Data collector – module that collects statistics from VMs 2 and 3 for analysis.

- A Watchdog – module that detects server outages. VM-Rejuv uses the ldirectord tool, which is used to monitor and administer real servers in an LVS cluster [156].
- Software Rejuvenation Agent (SRA) coordinator – module that directs SRAs on VMs 2 and 3 to initiate an application-server restart.

While virtual machines 2 and 3 run:

- The web-application server – the resource being load-balanced and periodically rejuvenated.
- Software rejuvenation agents – modules that initiate rejuvenation actions.
- A set of probes – modules that collect statistics from various sources including log files, (guest) operating system kernel (e.g., CPU utilization, memory usage, swap space, etc.) and application-server proxies (e.g., the P-probe module sits in front of the application-server collecting statistics on throughput and latency).

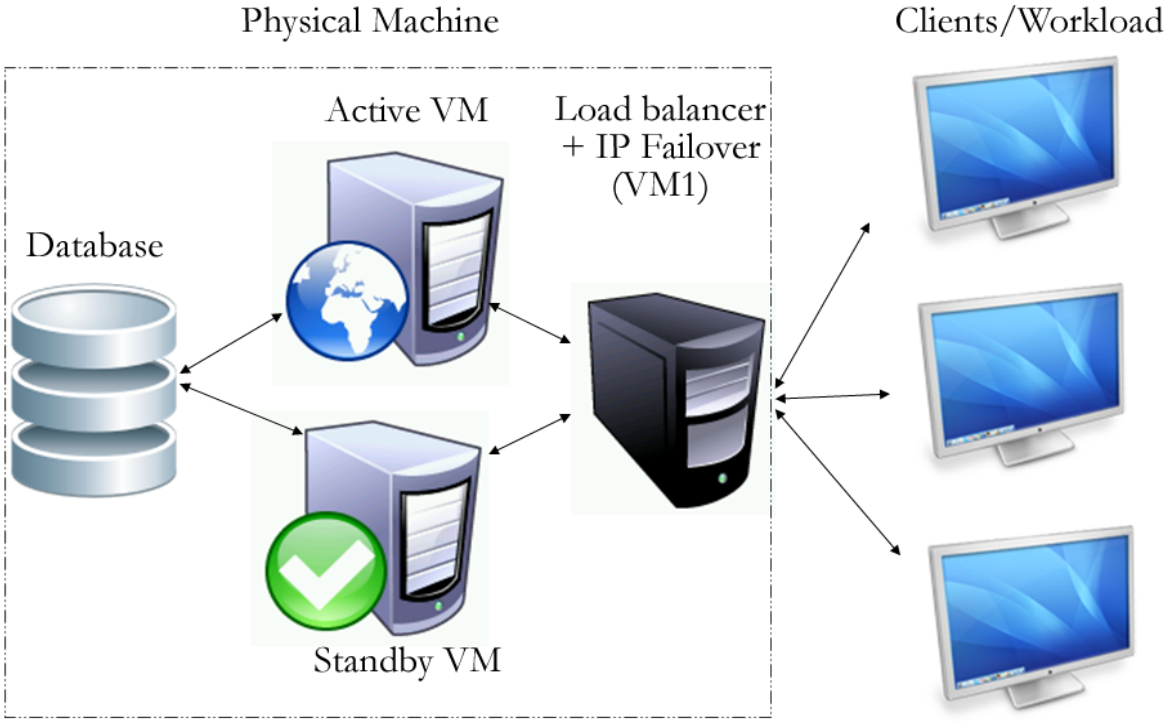

Figure 5.12: VM-Rejuv deployment[7]

An example deployment of a web-application using VM-Rejuv is shown in Figure 5.12. During its operation, client requests to the web-application are routed by the LVS load-balancer on VM1 to the application server on the active VM, while the standby VM (and its application-server) remains ready but inactive as a hot replica until a rejuvenation is signaled by the SRA coordinator.

When a rejuvenation action is signaled, the active VM and standby VM switch roles. New client requests are routed to the application server on the standby VM (old standby VM marked as the "new" active VM); the application-server on the old active VM finishes processing any outstanding requests before the local SRA agent restarts the application server. The interval of time the old active VM spends processing client requests that are in-flight/outstanding when a rejuvenation is signaled is referred to as the *pre-rejuvenation delay-window*.

The use of redundancy in VM-Rejuv and coordinated switch-overs between the active VM and the standby VM support application-server restarts that minimize the loss of in-flight client-requests during rejuvenation. These elements combined with application-specific technologies like session migration/replication (e.g., as found in the Apache Tomcat web/application server [173]) allow rejuvenations to be performed without disrupting clients, which potentially improves the client-perceived availability of the web-application.

Deploying a web-application under a prediction-based rejuvenation scheme like VM-Rejuv has a number of implications for its reliability, availability and serviceability.

Rejuvenation activities can be used as preventative maintenance to avoid certain kinds of failures, e.g., memory-leaks as shown in [173]. The use of redundancy and IP failover allow clients to be shielded from the failure of the active VM and minimizes disruptions due to preventative restarts. These aspects of VM-Rejuv's operation potentially improve the

[7]Server icons by Fast Icon Studio (http://www.fasticon.com) designed by Dirceu Veiga. Client/workstation icons by Layered System Icons designed by BogdanGC (http://bogdangc.deviantart.com/). Database icon by DryIcons (http://dryicons.com).

web-application reliability, availability and serviceability. However, the efficacy of problem detection/prediction mechanisms, the frequency of rejuvenation actions, the success rate of rejuvenation actions, and the size of the pre-rejuvenation delay-window are all elements that can negatively affect the RAS properties of an application deployed under VM-Rejuv.

Problem detection/prediction mechanisms influence the rate at which rejuvenation actions are initiated. Imperfect detection/predictions can result in too many or too few rejuvenation actions. Whereas too many rejuvenations may not disrupt clients (due to the redundancy and fail-over) time spent waiting to rejuvenate (the pre-rejuvenation delay-window) represents a period of vulnerability during which a failure of the active VM can affect clients. Further, frequent rejuvenations may put the system in a state where the active and standby VMs are constantly switching roles, indicating that the thresholds used to trigger rejuvenations may be inappropriate or may make the system unstable. Finally, rejuvenation actions may also fail, e.g., application servers could fail to restart or node-failover may be unsuccessful, in which case some other mechanism would need to be in place to rectify the situation.

On the other hand, too few rejuvenations may result in failures/unplanned downtime, which could have been avoided and may indicate inadequate fault/failure coverage for the system.

In our evaluation of VM-Rejuv we wish to quantify the effects of:

- the rejuvenation frequency
- the success rate of rejuvenation actions (node-failover and application-server-restart)
- the size of the pre-rejuvenation delay-window

on its reliability, availability and serviceability.

5.5.1 7U Process

System under test. For the system under test we use the TPC-W web-application hosted on two Apache Tomcat web/application servers [53] under VM-Rejuv. Tomcat is a Java-based web/application server developed by The Apache Foundation. Apache Tomcat is used as the web-application server in the VM-Rejuv experiments since a P-probe designed specifically for communicating performance statistics from Tomcat to the SRA coordinator is included in the VM-Rejuv prototype [8].

Fault model. VM-Rejuv's main detection mechanisms use the violation of response time and/or throughput thresholds to indicate that a rejuvenation action is required. We identify faults that can be used to trigger these detection mechanisms. Severe memory leaks affect both throughput and response time, degrading these performance metrics [173] in application servers. We use Kheiron/JVM to inject memory leaks into the web-application servers deployed under VM-Rejuv.

Fault-remediation relationship. VM-Rejuv initiates a node-failover and signals a rejuvenation (application-server restart) action in response to throughput or response time violations or application server crashes.

Micro-measurements. For micro-measurements we collect metrics on: the time for node-failover, the frequency of rejuvenation actions, the success of a rejuvenation, the size of the pre-rejuvenation delay-window, and application-server restart, server-side estimates of request throughput, and response time client-side goodput via instrumenting parts of VM-Rejuv (specifically the SRA agent coordinator and the SRA agents), and parsing application-server logs and parsing TPC-W client logs (client-side goodput is reported as

[8] The Tomcat P-probe is a Java class that is installed as a filter [129] in the pipeline that processes requests received by the application-server.

the number of web-interactions performed by TPC-W clients).

Macro-measurements. For macro-measurements we use the seven node, six parameter scoring model shown in Figure 5.13, with parameter descriptions in Table 5.7, to quantify the following the following facets of reliability, availability and serviceability:

- Reliability – frequency of rejuvenations, frequency of active VM failures during rejuvenation.
- Availability – basic steady state availability and tolerance availability.
- Serviceability – mean time to system restoration.

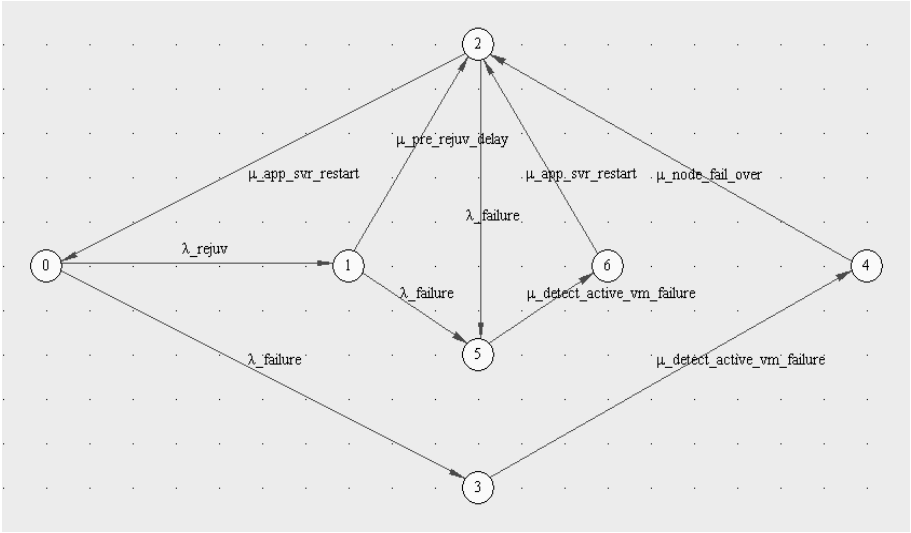

Figure 5.13: VM-Rejuv RAS model

Workload and metric collectors. Scripts that parse TPC-W client logs, Tomcat logs, SRA coordinator logs and SRA agent logs are used to gather micro-measurement data.

S_0	state where active VM services requests and standby VM ready
S_1	state where VM-Rejuv prepares to rejuvenate the active VM and the standby VM becomes the new active VM servicing new client requests
S_2	state where old active VM is ready to rejuvenate
S_3	state where the active VM has failed during normal operation
S_4	state where the failure of the active VM has been detected
S_5	state where the new active VM (the old standby VM) has failed while the old active VM is rejuvenating
S_6	state where the failure of the active VM during rejuvenation has been detected
λ_{rejuv}	rate of rejuvenation
$\lambda_{failure}$	forced/induced rate of failure of the active VM
$\mu_{pre\_rejuv\_delay}$	size of pre-rejuvenation delay-window
$\mu_{app\_svr\_restart}$	mean time to restart/rejuvenate the application server on the active VM
$\mu_{detect\_active\_vm\_failure}$	mean time to detect that the active VM has failed/crashed
$\mu_{node\_fail\_over}$	mean time to failover to the standby VM

Table 5.7: VM-Rejuv RAS model

5.5.2 VM-Rejuv Evaluation

We create a test deployment of VM-Rejuv consisting of three virtual machines co-located on a single physical machine. VM1 is configured with 640 MB RAM, 1GB swap, 2 virtual CPUs and an 8GB harddisk. VM2 and VM3 are each configured with 384 MB RAM, 512 MB swap, 2 virtual CPUs and 8GB harddisks. All three VMs run Centos 5.0 with a Linux 2.6.18-8.el5 SMP kernel.

To enable LVS load-balancing, the network interface on VM1 is configured with two IP addresses, one public IP address and one private IP address (192.168.1.xxx). Our LVS configuration is based on LVS-NAT [150]. VM2 and VM3 are configured with private IP addresses only (192.168.1.xxx). VM2 and VM3 can route to VM1 only, whereas VM1 can route to VMs 2 and 3 and the internet.

The physical machine hosting the VMs is configured with 2 GB RAM, 2 GB swap, an Intel Core Duo E6750 Processor (2.67 GHz) and a 228 GB harddisk running Windows XP Media Center Edition SP2.

Figure 5.14 shows our VM-Rejuv configuration. We install Apache Tomcat v5.5.20 and Sun

Microsystems' Hotspot Java Virtual Machine v1.5 on VMs 2 and 3 as well as instances of the TPC-W web-application. We use the MySQL 5.0.27 database server to store the TPC-W web-application data, and this is installed on VM1. The TPC-W web-application instances on VMs 2 and 3 are configured to access the database server on VM1. The LVS tools (IPVS v1.2.1 and ipvsadm v1.24) are installed on VM1 [150]. The following VM-Rejuv components are installed on the three VMs: the SRA coordinator, ldirectord watchdog, response time and throughput monitors are installed on VM1 while the SRA agents are installed on VM2 and VM3.

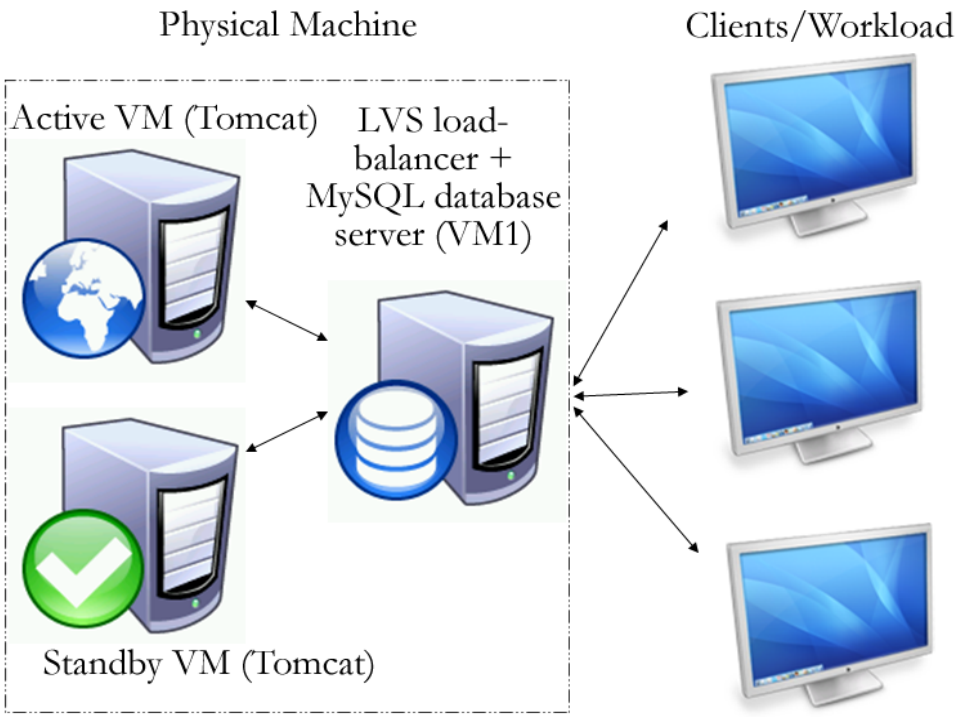

Figure 5.14: VM-Rejuv configuration[9]

The VM-Rejuv prototype works with the Apache Tomcat web/application server [173]. Whereas the components of VM-Rejuv are written in Java, operations such as rejuvenating application servers and updating LVS tables for failover are facilitated by shell scripts called from Java using the java.lang.Runtime::exec() API. To restart/rejuvenate Tomcat,

[9]Server icons by Fast Icon Studio (http://www.fasticon.com) designed by Dirceu Veiga. Client/workstation icons by Layered System Icons designed by BogdanGC (http://bogdangc.deviantart.com/).

VM-Rejuv's SRA agents invoke the shutdown.sh and startup.sh scripts in the bin directory under the Tomcat installation directory, while updates to the LVS table to designate the new active VM are performed via calls to the Linux Virtual Server Administration utility, ipvsadm.

We simulate a client load of 50 TPC-W clients using the Shopping Mix as their web-interaction strategy.

During 15 failure-free runs, each lasting 22 minutes, the average number of client-side interactions recorded is 7745.2 ± 748.9. Figures 5.15 and 5.16 show a 10 minute sample of the throughput and response time data reported by VM probes during one of our failure-free runs. From our failure-free runs the average throughput is ~13 requests per second and the average response time is ~11 ms. We use the server-side throughput and response time numbers reported to set the SLA violation thresholds for VM-Rejuv and inject faults that result in the violation of these thresholds, triggering rejuvenation actions so we can estimate the parameters for our scoring model.

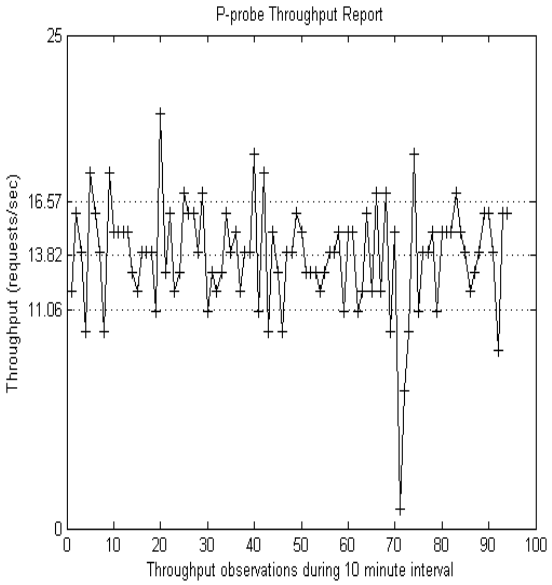

Figure 5.15: VM-Rejuv baseline throughput sample

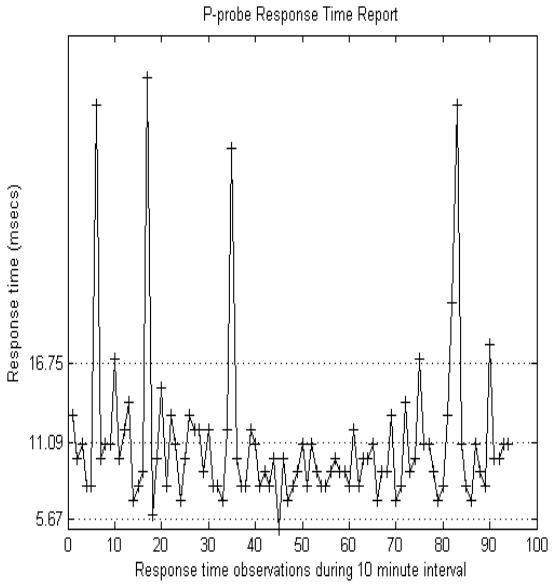

Figure 5.16: VM-Rejuv baseline response time sample

To estimate the size of the rejuvenation window, we set VM-Rejuv's response time violation

threshold at mean response time (11 ms) and re-run the workload of 50 clients. VM-Rejuv triggers rejuvenations after four consecutive SLA violations. During three 22 minute runs we observe an average of 4 rejuvenation actions per run. During rejuvenation actions, the mean failover time is 25.62 msecs ± 3.46 msecs (see Figure 5.17) with a mean pre-rejuvenation delay window size of 14,769 msecs ± 5,420 msecs (see Figure 5.18).

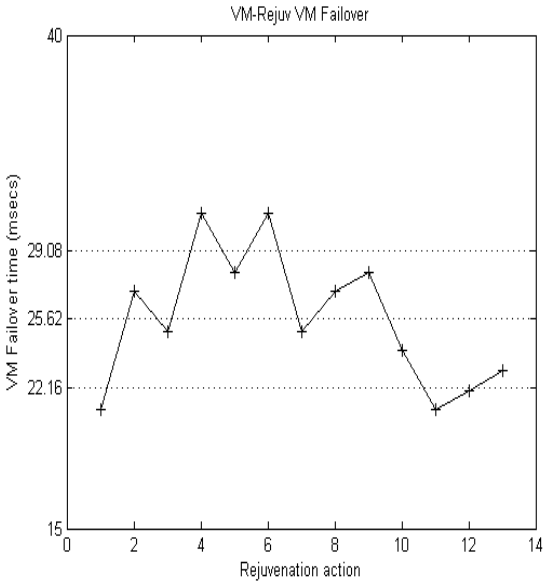

Figure 5.17: VM-Rejuv VM failover time

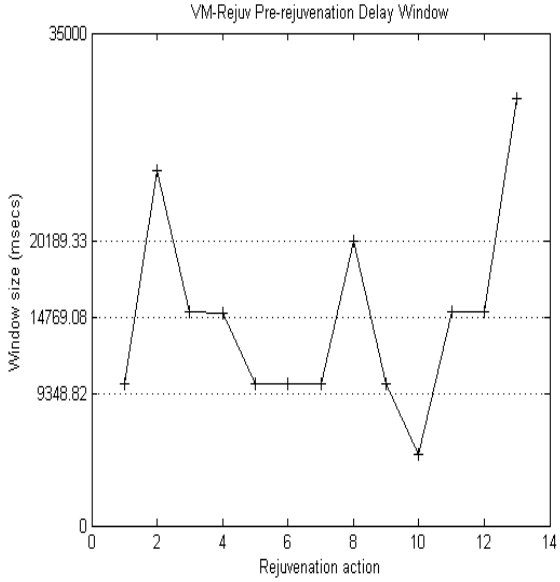

Figure 5.18: VM-Rejuv rejuvenation window size (50 clients)

In our fault-injection experiments we subject both Tomcat application servers deployed under VM-Rejuv to memory leaks that result in resource exhaustion within 5.53 minutes (332.017 seconds) of running the 50 client TPC-W workload (see Figure 5.19 for example resource exhaustion traces). We set VM-Rejuv's response time violation threshold to the mean response time of the failure-free runs (11 ms) and measure the frequency of rejuvenations, and the size of the pre-rejuvenation delay window. Introducing memory leaks in the Tomcat application servers increases the response time and delays the rejuvenation of the old active VM after the standby server is brought online, since the old active VM must service outstanding requests before it rejuvenates. Table 5.8 summarizes the results from five 22 minute memory-leak experiments.

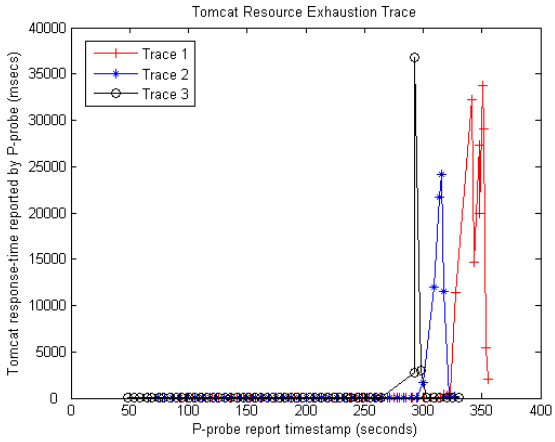

Figure 5.19: Tomcat resource exhaustion trace

Run #	Rejuvenation actions	Rejuvenation interval (secs)	Failover time (msecs)	Pre-rejuvenation delay window (msecs)
1	8	155.47	33.88	34,657.63
2	9	142.13	31.63	18,321.38
3	6	155.76	27.20	16,175.60
4	7	149.08	24.71	37,538.57
5	8	167.86	27.29	30,314.43
Avg	**7.6**	**154.06**	**28.94**	**27,401.52**

Table 5.8: VM-Rejuv subjected to memory leaks

Using a mean rejuvenation interval of 154.06 seconds, mean rejuvenation window size of 27,401.52 msecs and mean failover time of 28.94 msecs, we score the VM-Rejuv deployment using the RAS model in Figure 5.13. The mean time to restart Tomcat during the memory leak experiments is 3 seconds and the mean time to detect a server outage (via the ldirectord watchdog) is 5 seconds.

π_0	0.824673
π_1	0.135495
π_2	0.023510
π_3	0.012419
π_4	0.000072
π_5	0.002395
π_6	0.001437

Table 5.9: VM-Rejuv steady state probabilities – memleak scenario

The steady-state probabilities of the VM-Rejuv model are shown in Table 5.9 and model analysis results are shown in Table 5.10.

Using the scoring model we can estimate the number of active VM failures expected during rejuvenation actions per day, i.e., the frequency of transitions from S_1 to S_5 ($F_{S_1 \to S_5}$) plus the frequency of transitions from S_2 to S_5 ($F_{S_2 \to S_5}$). This we estimate at 41 per day under the failure conditions used in our experiments (1 memory-leak failure every 5.53 minutes).

From the steady-state probabilities of the model we estimate that the deployment spends ~82% of the time in its normal operating mode/configuration, π_0, and ~16% of its time rejuvenating ($\pi_1 + \pi_2$). While rejuvenations are taking place clients-requests are serviced by the standby VM; as a result the system would be considered UP from the client's perspective in states $\{S_0, S_1, S_2\}$ – UP 1416.5 minutes per day (98.37%) and DOWN 23.5 minutes per day (1.63%). Administrators on the other hand may consider the system to be UP if it is in state S_0 since states S_1 and S_2 represent a window of vulnerability. From the administrator's perspective the system is UP 1187.5 minutes per day (82.47%) and DOWN 252.5 minutes per day (17.53%), of which 229 minutes are spent performing rejuvenation actions.

In state S_1 clients still connected to the old active VM may experience some performance degradation and even lose requests if the degree of resource depletion on the old active VM is so severe that it cannot clear its backlog before the other VM needs rejuvenating. Further, increasing the size of the pre-rejuvenation delay window (either through missing rejuvenation opportunities or imperfect prediction) increases the time spent in S_1 where the overall system is vulnerable to failures of the current active VM.

5.6 Evaluation Part 3

In our final case-study we model and experimentally evaluate the efficacy of hardened device drivers in OpenSolaris.

Measure	Metrics	Results
Reliability	Frequency of active VM failures during rejuvenation per day $F_{S_1 \to S_5} + F_{S_2 \to S_5}$	41.377455
Availability	Basic steady-state availability ($UP_{admin} = \{S_0\}$)	0.824673
	Tolerance availability ($UP_{client} = \{S_0, S_1, S_2\}$)	0.983678
Serviceability	Mean-time to system restoration ($UP_{admin} = \{S_0\}$)	22,373 msecs
	Mean-time to system restoration($UP_{client} = \{S_0, S_1, S_2\}$)	5,509 msecs

Table 5.10: Summary of VM-Rejuv RAS model analysis results

OpenSolaris is a fully functional Solaris operating system release built from open source [155]. Solaris is a UNIX operating system developed by Sun Microsystems. Under the OpenSolaris initiative the Solaris kernel source was made available under an open license (circa June 2005). The most recent release of OpenSolaris is based on the Solaris 10 operating system. In the remainder of this section all references to OpenSolaris pertain to the release based on Solaris 10.

OpenSolaris includes a number of technologies designed to improve the reliability, availability and serviceability of the operating system, one of which is the Solaris Fault Manager.

The Solaris Fault Manager is a software architecture for fault management that incorporates several software components: an **event protocol** for sending and recording error and fault information, a **fault-diagnosis engine** and **a set of programming interfaces** that improve diagnosis, isolation, recovery and dynamic deactivation of faulty hardware [155]. This collection of software components is referred to as the Fault Management Architecture (FMA) [136]. A fault-centric software model correlates error reports into a binary telemetry flow and dispatches the telemetry stream to an appropriate diagnosis engine. Diagnosis engines can generate specific information about the fault for use by maintenance personnel and cause corrective actions to be automatically taken if possible, e.g., taking a faulty hardware component offline.

The FMA I/O Fault Services enable device driver developers to integrate fault management capabilities into I/O device drivers [136]. The Solaris I/O fault services framework defines a

set of interfaces that enable device drivers to coordinate and perform basic error handling tasks and activities. *Hardened* device drivers make use of the I/O fault services framework for error handling and diagnosis.

5.6.1 7U Process

System under test. In our evaluation of hardened device drivers we use the Broadcom Gigabit Ethernet (bge) device driver as a test subject. TPC-W web-application stack components are deployed on OpenSolaris and the web-application and database components are bound to a network interface managed by the bge device driver.

Fault model. We use the bus_ops fault injection tool (bofi) [135] to inject faults into the bge device driver. bofi is part of the device driver hardening test harness provided by Sun Microsystems [136]. bofi facilitates controlled corruption of programmed I/O (PIO) and DMA requests and interference with interrupts, thus simulating faults that occur in the hardware managed by the driver [136]. These faults, when injected, can lead to service loss due to corrupted PIO/DMA operations, service loss due to stuck interrupts, service degradations and unresponsive drivers.

In our fault-injection experiments we script bofi's fault-injection operations using utilities in the driver hardening test harness. These utilities are used to run a specific workload, log the accesses made by the device driver while the workload is run, generate specifications on how to corrupt the driver's accesses to its hardware and generate test scripts that re-run the workload while injecting faults that corrupt specific device driver accesses.

Fault-remediation relationship. Hardened device drivers are required to respond immediately to detected errors by attempting recovery, retrying an I/O transaction, attempting fail-over, reporting the error to the calling application/stack or panicking if the error cannot

be constrained in any other way. Detected errors are communicated to the Fault Manager as an *ereport* – a structured event defined by the FMA event protocol specification [136]. The event protocol specifies a common set of data fields that must be used to describe error and fault events, and a list of suspected faults. ereports may be used to indicate a number of events including, but not limited to, reporting that a device has: entered an invalid state, self-corrected an internal error, encountered an uncorrectable internal error, detected a stalled data transfer, detected an unresponsive device or detected that a device has raised too many consecutive invalid interrupts.

In addition to detecting and reporting errors, hardened device drivers must indicate whether or not an error has impacted the services provided by a device. Service impacts are reported as one of:

1. Service lost – service provided by the device is unavailable.
2. Service degraded – driver can provide a partial or degraded level of service.
3. Service unaffected – an error was detected but the services provided by the device are unaffected.
4. Service restored – all the device's services have been restored.

Micro-measurements. For micro-measurements we collect metrics on: driver recovery times, driver recovery success and the frequency of service losses or degradation.

Macro-measurements. For macro-measurements and scoring we use the model shown in Figure 5.20 to quantify the following facets of reliability, availability and serviceability:

- Reliability – frequency of service losses that escalate to the driver being marked as unresponsive, frequency of partial service restorations (degradation of service).
- Availability – basic steady-state availability, tolerance availability.

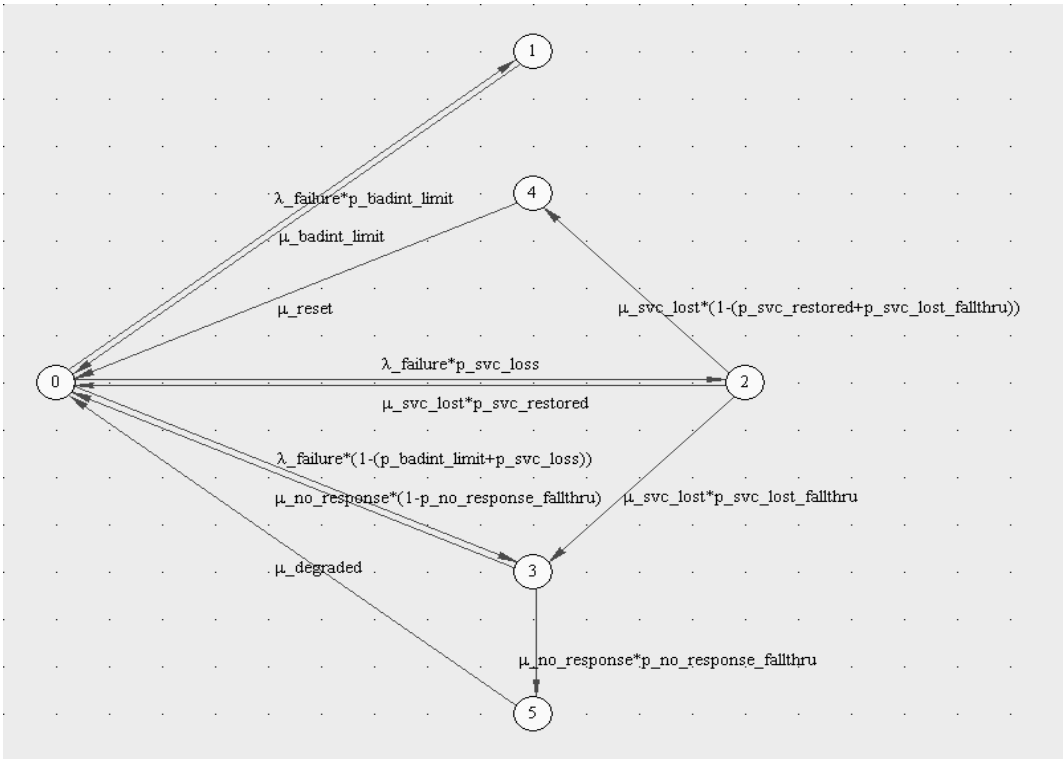

Figure 5.20: Hardened device driver RAS model

- Serviceability – mean time to system restoration.

The model consists of six states and ten parameters:

- S_0 – driver/device working normally.
- S_1 – service loss due to stuck interrupts.
- S_2 – service loss due to corrupted PIO operations.
- S_3 – driver/device status reported as unresponsive.
- S_4 – service recovery not reported after service loss due to corrupted PIO operations.
- S_5 – service reported as degraded.
- $\lambda_{failure}$ – forced rate of device driver failures.
- p_{badint_limit} – proportion of failures that result in stuck interrupts.

- p_{svc_loss} – proportion of corrupted PIO operations that lead to service loss.
- μ_{badint_limit} – mean time for recovery from stuck interrupts.
- μ_{svc_lost} – mean time to recover from service losses due to corrupted PIO operations.
- μ_{reset} – mean time until a response is received from the driver/device in lieu of a service recovery report.
- $\mu_{no\_response}$ – mean time to report restoration of services after driver/device marked as unresponsive.
- $\mu_{degraded}$ – mean time to report return to normal operation after service degradation.
- $p_{svc\_lost\_fallthru}$ – proportion of service losses that lead to the driver/device being marked as unresponsive.
- $p_{no\_response\_fallthru}$ – proportion of unresponsive driver events that are partially restored to degraded level of service.

Workload and metric collectors. Fault-injection activities, device driver diagnosis, device driver service impact reports, and driver recovery actions are timestamped and stored in the fault-management logs – accessible via the *fmdump* utility. We use data recorded in this log to estimate parameters used in our scoring model.

5.6.2 Evaluating Hardened Network Device Drivers on OpenSolaris

Our test platform uses a Sun Ultra40 Workstation configured with 4 GB RAM, 6 GB swap, 1 AMD Opteron Dual Core Processor and a 500 GB harddisk running OpenSolaris. The Ultra 40 Workstation is equipped with three network interface cards (NICs), two on board and one on an PCI Express (PCI-E) expansion card. The two onboard NICs are managed by the unhardened nVidia 1Gb Ethernet v1.15 device driver (nge), while the PCI-E NIC

uses the hardened Broadcom Gigabit Ethernet v0.57 device driver (bge). One nge interface and the bge interface are assigned static IP addresses while the remaining nge interface is un-used.

We configure bofi to target the FMA-aware/hardened bge device driver. For the workload we use an instance of the TPC-W web-application, running on MySQL 5.0.27, Resin 3.0.22 and Sun Microsystems v1.5 Hotspot Java Virtual Machine. The database server and web/application server components of the TPC-W web-application stack receive requests from 20 Remote Browser Emulator (RBE) clients using the Shopping Mix as their web-interaction strategy. The MySQL database server and Resin web/application server are bound to the bge interface, while RBE clients submit requests using the available nge and bge interfaces.

Accesses to/from the bge interface are logged during the execution of a 23 minute TPC-W run and used to create fault-injection test-scripts for the bge device driver. The fault-injection test-scripts re-run the TPC-W workload multiple times injecting one or more faults during each run. Between runs the database server and web/application server are restarted.

Over the course of 39 fault-injection runs (15 hours, 23 minutes) the following fault/failure data was retrieved from the fault-management logs: a total of 100 faults are injected, 83 of which result in corrupted PIO operations, 1 stuck interrupt and 16 unresponsive driver events [10].

67 of the 83 corrupted PIOs led to service loss, the stuck interrupt failure led to service loss and the 16 unresponsive driver/device events result in periodic, but short lived, service interruptions – a total of 84 faults/failures during 923 minutes ($\lambda_{failure}$) leading to service loss or interruptions with 79.76% attributed to corrupted PIO operations (p_{svc_loss}), 1.19% attributed to stuck interrupts (p_{badint_limit}) and 19.05% attributed to unresponsive driver/device events. Of the 67 service lost events due to corrupted PIO operations, recovery was re-

[10]The failure mix was a consequence of the workload being run and the accesses that occur during the workload.

ported for 55 of them ($p_{svc\_restored}$ = 82.09%), 1 resulted in an unresponsive driver/device ($p_{svc\_lost\_fallthru}$ = 1.49%) and there were 11 unreported recoveries[11]. μ_{reset} was set to 556.9 msecs for the unreported recoveries based on the inter-fault injection times during runs where this behavior was observed.

Of the 16 unresponsive driver/device events, 15 were reported as restored while 1 lead to degradation before complete service restoration was reported ($p_{no\_response\_fallthru}$ = 6.25%).

Recovery from service loss due to stuck interrupts (μ_{badint_limit}) was reported as 1085.2 msecs, recovery from service loss due to corrupted PIO operations (μ_{svc_lost}) was 576.9 msecs and recovery from degraded operations ($\mu_{degraded}$) was 173.5 msecs.

Using these parameter values in our scoring model (Figure 5.20 we obtain the steady-state probabilities shown in Table 5.11, which we use to calculate the reliability, availability and serviceability metrics shown in Table 5.12. From the steady-sate probabilities of the model we estimate that the bge device driver experiences 1.6 service losses that escalate to the driver being marked as non-responsive ($F_{S_2 \to S_3}$) per day, 17.1 un-reported device driver recovery events ($F_{S_2 \to S_4}$) per day, and 1.7 service losses that lead to partial/degraded service restorations ($F_{S_3 \to S_5}$) per day. We also estimate that the driver spends 99.92% of its time servicing requests, UP ~1438 minutes per day and DOWN ~2 minutes per day considering both steady-state availability (π_0 = 99.9169%) and tolerance availability ($\pi_0 + \pi_5$ = 99.9173%). Mean time to service restoration for the bge device driver is estimated at ~550 msecs.

[11] Situations where service impacts were reported but no log entry for service restoration was reported even though the workload continued to run. The bge device driver subsequently responded to new fault-injections and report service impacts with no further follow-up.

π_0	0.99916919
π_1	0.00001957
π_2	0.00069735
π_3	0.00000003
π_4	0.00011054
π_5	0.00000333

Table 5.11: Hardened bge device driver steady-state probabilities

Measure	Metrics	Results
Reliability	Service loss to unresponsive driver fall-throughs ($F_{S_2 \to S_3}$)	1.6 a day
	Service loss to unreported recovery fall-throughs ($F_{S_2 \to S_4}$)	17.1 a day
	No response to degraded-service fall-throughs ($F_{S_3 \to S_5}$)	1.7 a day
Availability	Basic steady-state availability ($UP_{admin} = \{S_0\}$)	0.999169
	Tolerance availability ($UP_{client} = \{S_0, S_5\}$)	0.999173
Serviceability	Mean-time to system restoration ($UP_{admin} = \{S_0\}$)	548.2 ms
	Mean-time to system restoration($UP_{client} = \{S_0, S_5\}$)	546.0 ms

Table 5.12: Summary of hardened bge driver RAS model analysis results

5.7 Related Work

Our approach to benchmarking reliability, availability and serviceability combines runtime fault-injection tools with the models of failure scenarios used to describe and score fault-injection experiments. The models used for scoring can be used to capture different perspectives on the failure and recovery behavior of systems.

In [77], [25] and [32] the authors discuss the importance of fault-injection tools in evaluating the reliability of systems and compare the tradeoffs between different approaches for hardware fault-injection and software-implemented fault-injection (SWIFI). Two classes of hardware fault-injection (injection with contact, e.g., pin/chip-level fault injection and injection without contact, e.g., exposure to heavy ion radiation) and two classes of software fault-injection (compile-time fault-injection and runtime fault-injection) are presented. Compared to hardware fault-injection, software-implemented fault-injection has a number of benefits including: the ability to emulate a variety of faults/failures, lower cost since dedicated hardware is not needed, convenience, portability to other platforms and extensibility to

include new classes of faults. In our 7U evaluations we use SWIFI tools capable of runtime fault-injection to allow us the flexibility to interact with different target systems without the need for re-compilation and/or re-linking.

[96] presents a framework for benchmarking the reliability, availability and serviceability characteristics of systems and discusses three classes of benchmarks: measurement-based, model-based and hybrid – combinations of measurements and modeling where models guide experiments and/or are validated/refined by experiments. Our 7U evaluation method is an example of the hybrid benchmark approach. In our 7U benchmark RAS models are used to guide fault-injection experiments; however, we expect these models to evolve over time as different failure scenarios are considered and/or more insights about the failure and recovery behavior of the system under test are obtained from fault-injection experiments.

[89], [16], [87], [117] and [17] are examples of measurement-based evaluations of reliability and availability.

[89] proposes the R-Cubed (R^3) – Rate, Robustness, and Recovery – framework for availability benchmarking that evaluates availability as a function of three attributes: the rate of failures and maintenance events, robustness and recovery. Whereas our 7U benchmark considers the failure-rates, failure handling and recovery it does not consider maintenance-induced failures/faults. However, analytical models can be constructed to reason about maintenance-induced failures/faults, their impacts and their resolutions. [16] conducts a measurement-based study of availability and maintainability benchmarks using software RAID systems. In evaluating availability, the authors emphasize 1) capturing the perspective of the end-user and 2) the need for availability metrics that capture the spectrum of availability, i.e., taking into consideration degraded modes of operation as well as the normal mode of operation. In our 7U benchmark we demonstrate how the models used for scoring can quantify different facets of availability, e.g., basic steady-state availability, tolerance availability and capacity-oriented availability of a system.

[87] describes the DBench-OLTP dependability benchmark. DBench-OLTP is a measurement-based dependability benchmark for online transaction processing systems (database systems). The fault-model used in DBench emulates operator faults, e.g., deleting a database table, deleting a user schema, abrupt transaction system shutdown, etc. The DBench benchmark is composed of three sets of measures – baseline performance measures, performance measures in the presence of the faultload and dependability measures. Baseline performance measures are reported in terms of transactions per minute (tpmC) and price per tpmC ($/tpmC). Performance measures in the presence of the faultload are reported in terms of number of transactions executed per minute in the presence of faults and the price per transaction in the presence of faults. Finally, the dependability measures reported are: the number of data errors detected by consistency tests, availability from the point of view of the system under test (SUT) and the availability from the client's point of view. We differ from this work in our choice faults, choice of metrics and our use of models for describing failures and scoring recovery activities by computing multiple facets of reliability, availability and serviceability.

[117] 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.

[17] describes work towards a self-healing benchmark. The authors identify a number of challenges to benchmarking self-healing capabilities including: quantifying healing effectiveness (identifying different metrics to quantify the impact of disturbances), accounting for incomplete healing and accounting for healing specific resources (spare disks, hot standbys, etc.). In our 7U benchmark RAS models based on Markov Chains and Markov reward

networks can be used to capture different facets of reliability, availability and serviceability, model imperfect recovery/repair scenarios and consider spare/redundant resources. Further, our use of models allows us to make a connection between the mechanisms used by a system to accomplish healing and their relation to the high-level system goals, dictated by SLAs and policies, governing the system's operation. Finally, we can also use models to analyze the effects of individual or combined remediation mechanisms on the overall efficacy of the system's healing capabilities.

[168] makes a case for application-specific benchmarks; the application of our 7U approach to web-application stacks and their components is an example of an application-specific benchmark. Further, the use of tailored models for scoring allow us to focus on specific aspects of the system under test being evaluated.

Our work is complementary to the work done on robustness benchmarking [43] and fault-tolerant benchmarking [191]. 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.

[47] is an example of a model-based approach to RAS evaluation. 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 and evaluate their models.

5.8 Summary

In this chapter we discussed and presented a model-based and measurement-based approach to evaluating the reliability, availability and serviceability properties of web-application stacks and their components. We use runtime fault-injection tools to insert faults/induce failures into three target systems (§5.4, §5.5 and §5.6), developed analytical models for describing the failure scenarios and scoring system responses, and demonstrated the measures that can be computed to evaluate or compare systems based on their responses (or lack thereof) to different failure scenarios.

Using RAS models we identify different facets of reliability, availability and serviceability, which can be quantified via fault-injection experiments, and link the details of remediation mechanisms (recovery time, recovery success rates, etc.) to high-level RAS-metrics that govern the system's operation.

Chapter 6

Contributions, Future Work and Conclusion

6.1 Thesis Contributions

The contributions of this thesis include the following:

1. A generalized approach to effecting runtime adaptations in applications hosted in managed and unmanaged execution environments. In developing our runtime adaptation techniques, we identify facilities in contemporary execution environments (e.g., Microsoft's Common Language Runtime, Sun Microsystems' Java Virtual Machine and the Linux operating system on the Intel x86 processor) that can be used to effect dynamic modifications to the applications they host. The runtime adaptation techniques we develop facilitate in-situ and in-vivo interactions with systems and are transparent to both the application being modified and the execution environment.

2. A suite of runtime fault-injection tools, Kheiron, that targets multiple execution environments and applications written in multiple languages. Kheiron uses our

runtime adaptation techniques to interact with application elements (data-structures, functions/methods, data-types, classes, type-instances/object-instances, etc.).

3. Identification of analytical tools – Markov Chains, Markov Reward Networks, Feedback Control – and techniques that can be used to reason quantitatively about different facets of the reliability, availability and serviceability properties of systems. In our work towards an RAS benchmark we identified analytical tools (models) that can describe and score the failure scenarios used to evaluate systems. We discuss the RAS measures and metrics that can be computed using these models and present examples based on real systems to illustrate their use.

4. A model-based and measurement-based approach to evaluating the RAS characteristics of systems, which combines runtime fault-injection with the analytical models. We describe and demonstrate how these evaluations are conducted and scored, identify the data sources used to estimate parameters of analytical/scoring models, identify runtime fault-injection tools (some developed by us, and some developed by third-parties) that can be used in the system-evaluations, and discuss the results.

6.2 Research Accomplishments

In addition to the contributions listed above, the following practical accomplishments have already been completed to date:

- Published papers, including [44], [196], [62], [64], [63], [65], [66] and an invited talk [67].

- Equipment donations from Sun Microsystems and StackSafe Inc. supporting our RAS-benchmarking work.

6.3 Practical Concerns

There are a number of practical concerns that need to be addressed when conducting a 7U evaluation:

Sourcing or creating RAS models. RAS models are used to describe the failure scenarios used in the evaluations and to score the system's responses. However, the issue of who creates these models is an important one. System vendors have an important role to play in this process. Whereas vendors may have detailed knowledge about (parts of) the system and are well-placed to discuss failure modes and scoring responses, end users rely on these systems for their business' day-to-day and/or mission-critical activities and also have opinions on whether the system has failed. Discrepancies between what vendors consider failures and what clients/end-users consider failures can, and have, occurred [163]. As a result, considering both perspectives in the evaluation process is key to increasing confidence in the systems being developed and deployed. We expect that more than one RAS model will be used in the evaluation of a system and over time, models contributed by both vendors and end-users (customers, researchers, etc.) will result in a set of standardized failure scenarios and scoring criteria as has occurred with performance benchmarks (e.g, SPEC, NIST, TPC) [177, 140, 190]. Vendors may use RAS models to conduct their own internal evaluations, compare against other systems and/or demonstrate compliance with agreed upon standards, while end-users can use these RAS models to verify/validate vendor claims.

Incremental evaluations. During a system's lifetime the fault-model used in its RAS evaluations will be modified and/or expanded. New faults may be added to augment the existing faults in the model and/or RAS-enhancing mechanisms may be added or improved in the system. These modifications to the fault-model and/or system may require new or refined RAS models to describe and score the failure scenarios used in the RAS evaluations.

RAS evaluations may employ one or more failure scenarios; depending on the changes made to the fault-model or the system some (or in the worst case all) of the failure scenarios may have to be re-run to obtain updated information on the system's RAS capabilities under the selected failure scenarios. The number of failure-scenarios re-run may be influenced by the importance placed on each failure scenario by the evaluator.

System accessibility to collect model parameters. Collecting the data used to estimate parameters in the RAS models may necessitate observation points/hooks in systems. Example observation points include: log files, console output, and compiled-in or dynamically added instrumentation points. Logging APIs/toolkits, e.g., log4j, log4cxx and log4net [52] may be used by the original system developers to produce data about the system's operation, data may be collected from the execution environments where the system runs or dynamic instrumentation tools like DTrace [24], Dyninst [18] and Kheiron [63, 62] may be employed to collect data from the system and/or execution environment. These data-collection strategies are applicable to both closed-source and open-source software systems. In the case of closed-source systems, third-party evaluators may rely on existing instrumentation points or employ dynamic instrumentation tools, whereas for open-source systems third-party evaluators can augment the system with compiled-in instrumentation and/or use dynamic instrumentation tools. Whereas it is unlikely that vendors and end-users will agree on every observation point, access to the source and/or runtime instrumentation tools allow parties the flexibility to obtain the data they are interested in from the system being studied.

Managing the costs of running the benchmark. Running a 7U evaluation may incur a number of costs, which may be expressed in terms of time, money and/or effort concerned with: setting up or configuring the infrastructure used in the evaluations, e.g., obtaining physical and/or virtual machines to create evaluation testbeds, installing and configuring target systems and their dependencies, identifying (obtaining) or developing fault-injection

tools and workload generators. Virtualization technologies, e.g., VMWare [83], Xen [183], Kernel-based Virtual Machines [80], Solaris Zones [155], etc.) can be used to create evaluation testbeds, specialized testing/staging environments, e.g., StackSafe's Test Center [81][1], VMWare ESX [85], etc., can be used to clone existing physical or virtual machines and import them into the testbed. Fault-injection tools and workload generators may need to be developed; some may be provided by vendors, e.g., bofi [135] while others may be open-source, e.g., TPC-W [119].

Limitations. With respect to evaluating the RAS capabilities of a system, difficulties in coercing the system into specific failure modes and reproducing specific failure scenarios or classes of failures represent the major limitation of our RAS-benchmarking approach. Our evaluation approach is based on a combination of modeling and measurement, where both elements rely on re-creating specific failures in systems. If the failures under consideration are reproducible, e.g., by using specific fault-injection tools, generating specific workloads or capturing and replaying requests/events, then they can more readily be packaged into scenarios and distributed with an RAS benchmarking suite. However, failures that are difficult to reproduce or induce are hard to include in the set of failure-scenarios distributed in such a suite. The distinction between reproducible failures and hard-to-reproduce failures is analogous to the distinction between repeatable bugs (Bohrbugs) and non-repeatable bugs (Heisenbugs) [60][2].

6.4 Future Work

The work in this thesis has been focused on developing an approach to benchmarking reliability, availability and serviceability. As a result there are a number of interesting

[1] See §A for details on our experience using the Test Center.

[2] We do not claim that the failures to be studied as part of an RAS benchmark are necessarily manifestations of software defects. This perspective on failures is illustrated in our definition of failures presented in §3.1.

possibilities for future work.

6.4.1 Immediate Future Applications

Near-term research directions based off the work done in this thesis include:

Selecting fault-injection targets. Improving the selection of fault-injection targets in internet applications and cost-estimates of failure impacts using **path-based request-tracing** [30]. In our current work injecting failures into web-application stacks and their components, we considered each failure/failed request to have equal consequences. However, in an E-commerce web-application, like the TPC-W online book store, some failed requests are more costly than others. For example, failed operations on shopping carts or failed payment processing activities can be more costly than failed item-search operations, and as a result evaluators may want to focus on failures that affect "high-value" requests and re-produce these failures in benchmark runs. To classify/identify high-value requests we need to trace client-interactions from the initial contact with the web-application through to a specific target operation, e.g., client-interactions that lead to a payment submission.

Employing a path-based tracing toolkit like X-Trace [51] is one possible option. X-Trace is a network diagnostic tool designed to provide users and network operators with better visibility into Internet applications. It annotates network requests with metadata that can be used to reconstruct requests (including requests that make use of multiple network layers). Request-reconstruction is facilitated by X-Trace identifiers used to record the path requests take through a network. Currently components are X-Trace-enabled via source-code augmentations (Java and C++ applications are supported [154]); however, we envision using the runtime adaptation techniques developed in this thesis to dynamically inject X-Trace support into applications/components.

Evaluating other web-application deployments. Applying our RAS evaluation approach to more sophisticated web-applications, which mirror enterprise web-application deployments, e.g., J2EE web-applications or .NET web-applications using component services. Component services include: transaction management (Java Transaction API/JTA, COM+ transactions), messaging (Java Message Service/JMS, COM+ queued components), object pooling, remoting services (Java Remote Method Invocation/RMI, .NET Remoting) and directory services (Java Naming and Directory Interface/JNDI, COM+ catalog). These services represent key elements that are intended to improve the web-application's performance and reliability.

The TPC-W web-application used in our evaluation experiments does not use any component services, however, applying our evaluation approach to the SPECjAppServer2004 [176] would allow us to interact with a J2EE web-application. SPECjAppServer is a multi-tier benchmark for measuring the performance of J2EE application servers that exercises all major J2EE technologies implemented by compliant application servers including: transaction management, messaging services and object pooling.

Workload generator tools and strategies. Using different workload generator tools and strategies to study system behavior. Workload generators fall into two major classes: those that use a closed system model and those that use an open system model. In a closed system model, new job arrivals are only triggered by job completions (followed by think time), whereas in an open model new jobs arrive independently of job-completions [164]. The TPC-W workload generator (RBE client emulator) used in our evaluation experiments follows a closed system model. [164] shows that closed and open system models yield significantly different results when both models are run with the same load and service demands. Further, they posit that many applications exhibit behavior that is "in-between" the extremes of closed and open system models – described as *partly open system models*.

An important part of RAS evaluations involves quantifying the impacts of failures and

the impacts of remediations under typical operating conditions. As a result using flexible workload generators will allow evaluators to create environments that are closer to normal operating conditions, which will facilitate a better understanding of typical system behavior and the behavior of the system when subjected to faults/failures. Further, whereas workloads may be viewed as activities conducted while faults are injected, they may also be crafted to induce failures in systems. Advances in problem diagnoses, e.g., use of statistical machine learning [35, 28, 29] may identify a vector of attributes that reasonably predict the failure of a system (e.g., specific fluctuations of system resources). Designing fault-injection tools that are capable of re-creating all of necessary conditions (reproducing the failure vector) may be challenging; however, specific workloads or workload variations, e.g., targeted surges, may be used to reproduce the necessary conditions for system failure.

Evaluating classes of systems other than web-applications. Modeling and injecting failures in other classes of systems besides web-applications, e.g., multimedia streaming/delivery platforms and studying their responses (or lack thereof). Recent work [38] looks at studying the effects of failures and repairs in a peer-to-peer video delivery network (GolP2P) using Markov chains. Failures are described as the loss of peer-nodes transmitting video (and the subsequent depletion of play buffers), while repairs/reconfigurations occur when receiving nodes identify suitable replacements for lost transmitter-nodes before user experience suffers. In [38] the authors devise quality of experience metrics (QoE) for the video and audio streams received, which are a function of the loss rates, delays, reliability, availability, etc. of transmitting peers.

Flexible work generation strategies and tools, models of failure scenarios, fault-injection/failure-inducing tools and RAS metrics can be used to evaluate the RAS properties of streaming/delivery platforms.

6.4.2 Future Directions

Longer-term research directions based off the work done in this thesis include:

Programming language and execution environment support for runtime modification of applications. As discussed in this thesis a number of contemporary execution environments provide facilities that can be used to dynamically modify running systems, however, there are few guarantees of the safety of runtime changes. The ability to manipulate programs in execution is a powerful yet risky facility. However, stakeholders will be wary of using runtime adaptation facilities in production systems without stronger guarantees on their safety. The form and the degree to which we can express and codify such guarantees is still an open question, but the increasing sophistication of system-construction tools (high-level languages, modeling tools, integrated development environments/IDEs[3], etc.) and application execution environments (managed execution environments, e.g., the JVM and CLR and unmanaged execution environments, e.g., operating systems, processors, Xen, VMWare, hypervisors etc.) may provide insights into additional support for realizing adaptive systems and the development of runtime adaptation toolkits.

Developing runtime fault-injection/failure-inducing tools for systems. Injecting faults and inducing failures in systems are important activities in evaluating system reliability, availability and serviceability, and the development of fault-injection tools and fault-load generators is currently an open area of research. Tools that inject faults and induce failures in systems can be used to study how systems fail and benchmark system responses. System administrators/operators can use these tools to develop pre-canned failure-scenarios (workloads and fault-loads) to benchmark their systems. Further, they can also be used to train system operators, familiarizing them with system failure-modes and/or the (manual or

[3]Contemporary development environments already make use of runtime code updates during debugging, e.g., Hot Code Replacement (HCR) in Eclipse [54] and Edit-and-Continue in Visual Studio .Net 2005 [36].

automated) mechanisms available to mitigate or address these failures.

Familiarizing operators with system failure-modes and the mechanisms available to address failures is an important tool in combating *automation irony* [153] and has implications for the development of self-managing/autonomous systems. As systems become more autonomous, they assume more responsibility over their management activities – configuration, healing, optimization and protection. Whereas this allows operators to focus on other tasks, reduced contact with the (autonomous) system limits the amount of hands-on control experience they get and inhibits their ability to construct mental models and rules of system operation used for resolving problems. In essence, system automation may potentially make system administration harder, e.g., resulting in cases where automation takes care of the majority of management tasks, leaving administrators to deal only with the exceptional states that occur when automation fails and/or complex management tasks. Restoring system operation from these exceptional states may require detailed knowledge of the system's operation *and* the operation of the automated mechanisms the system employed unsuccessfully.

As a result, the development of these runtime fault-injection tools can be used to increase operator-confidence in the system's failure handling mechanisms while allowing them an opportunity to get hands on control experience with the system in different failure modes. Facilities for familiarizing operators with the system's failure-modes and failure handling capabilities can be enhanced by mechanisms that provide a degree of transparency into the activities of the system, e.g., providing descriptions and/or justifications for system (failure and recovery) actions [180].

Creating specialized testing environments for conducting RAS evaluations. Whereas we can create tools that are able to interact with systems "in-situ" and "in-vivo" to make modifications and/or inject faults, enterprises are likely to be wary of allowing runtime fault-injection experiments on production systems (stronger safety guarantees for runtime modifications and high confidence in remediation mechanisms may reduce, but not eliminate

their concerns). As a result, RAS evaluations may occur in testing/staging environments.

Testing/staging environments require infrastructure and management, e.g., sourcing, installing, configuring and maintaining multiple machines (including keeping production systems and staging systems in sync). The use of virtualization technologies may reduce the physical hardware resources needed; however, they do not eliminate the management overhead concerned with installing, configuring and updating these copies of production systems. Tools that support the cloning/import of production systems into virtualized containers, where they can be organized into application stacks, may reduce some of the management overheads concerned with setting up the infrastructure needed to perform RAS evaluations. The inclusion of RAS benchmarking tools in these virtualized staging environments may be a reasonable compromise between the need to evaluate production systems and the desire to evaluate the reliability, availability and serviceability capabilities of systems. Examples of such virtualized staging environments include the StackSafe Test Center [81] and VMWare's ESX [85][4].

Inducing failures via security vulnerabilities, e.g., system corruptions, crashes, denial of service (DoS) attacks, worm outbreaks/ propagation, etc. Tools that exploit security vulnerabilities in controlled ways, e.g., those provided by the Open Web Application Security Project (OWASP) [55], may be used in RAS evaluations of systems where the fault-model of interest is focused on specific attack vectors. The objectives of the RAS evaluations may include: threat-modeling, penetration testing, understanding how applications and systems are affected by exploiting specific security vulnerabilities, designing or validating threat/attack responses and/or hardening systems against specific attack vectors.

[4] Using VMWare Converter [84] to convert physical machines into virtual machines.

6.5 Conclusion

In this thesis, we develop a measurement-based and model-based evaluation methodology for evaluating the reliability, availability and serviceability properties of systems. In developing our RAS evaluation methodology we:

- Develop a generalized approach to effecting runtime adaptations in applications hosted in managed and unmanaged execution environments.
- Implement runtime fault-injection tools capable of in-situ and in-vivo interactions with systems.
- Identify analytical tools that can be used to quantify multiple facets of reliability, availability and serviceability. These analytical tools are used to construct RAS models, which describe failure scenarios and score system responses to these failure scenarios.
- Combine runtime fault-injection experiments with RAS models to demonstrate the evaluation process.

As the future work above demonstrates, this thesis enables the beginning of new research areas, especially in the areas of realizing systems capable of runtime adaptations and improving fault-injection tools and environments used for RAS evaluations. Further, this thesis presents a framework for developing RAS benchmarks for systems that combines practical tools with rigorous analytical techniques. Ultimately, we hope the work done here bridges the gap between practical and analytical approaches for studying and understanding the failure behavior of systems and reasoning about mechanisms that improve the reliability, availability and serviceability of current and next-generation (self-managing) systems.

Chapter 7

Bibliography

[1] A comprehensive model for software rejuvenation. *IEEE Trans. Dependable Secur. Comput.*, 2(2):124–137, 2005. Member-Kalyanaraman Vaidyanathan and Fellow-Kishor S. Trivedi.

[2] A. Goyal and S.S. Lavenberg and K.S. Trivedi. Probabilistic modeling of computer system availability. In *Annals of Operation Research*, pages 285–306, 1987.

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

[4] MySQL AB. Mysql open source database. http://www.mysql.com.

[5] Akshay Luther et al. Alchemi: A .net-based enterprise grid system and framework, user guide for alchemi 1.0, july 2005. http://www.alchemi.net/files/1.0.beta/docs/AlchemiManualv.1.0.htm.

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

[7] Algirdas Avizienis Fellow IEEE. The N-Version Approach to Fault-Tolerant Software. In *Proceedings of IEEE Transactions on Software Engineering Vol. SE-11 No. 12*, December 1985.

[8] Jean Arlat, Alain Costes, Yves Crouzet, Jean-Claude Laprie, and David Powell. Fault injection and dependability evaluation of fault-tolerant systems. *IEEE Transactions on Computers*, 42(8):913–923, 1993.

[9] 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.

[10] Alberto Avritzer and Elaine J. Weyuker. Monitoring smoothly degrading systems for increased dependability. *Empirical Softw. Engg.*, 2(1):59–77, 1997.

[11] Robert Balzer and Neil M. Goldman. Mediating connectors: A non-bypassable process wrapping technology. In *DARPA Information Survivability Conference and Exposition Volume 2*, 2002.

[12] Guruduth Banavar, Marc Kaplan, Kelly Shaw, Robert E. Strom, Daniel C. Sturman, and Wei Tao. Information flow based event distribution middleware. In *Middleware Workshop at the International Conference on Distributed Computing Systems (ICDCS)*, 1999.

[13] Yujuan Bao, Xiaobai Sun, and Kishor S. Trivedi. A workload-based analysis of software aging, and rejuvenation. *IEEE Transactions on Reliability*, 54(3):541–548, 2005.

[14] J.B. Bowles and J.G. Dobbins. High-availability transaction processing: practical experience in availability modeling and analysis. *Reliability and Maintainability Symposium, 1998. Proceedings., Annual*, pages 268–273, Jan 1998.

[15] T. Boyd and P. Dasgupta. Preemptive module replacement using the virtualizaing operating system realizing multi-dimensional software adaptation. citeseer.ist.psu.edu/boyd02preemptive.html, 2002.

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

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

[18] 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.

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

[20] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot–a technique for cheap recovery. citeseer.ist.psu.edu/candea04microreboot.html, 2004.

[21] 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.

[22] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. A microrebootable system – design, implementation, and evaluation. *CoRR*, cs.OS/0406005, 2004.

[23] George Candea, Emre Kiciman, Shinichi Kawamoto, and Armando Fox. Autonomous recovery in componentized internet applications. *Cluster Computing*, 9(2):175–190, 2006.

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

[25] J.V. Carreira, D. Costa, and J.G. Silva. Fault injection spot-checks computer system dependability. *Spectrum, IEEE*, 36(8):50–55, Aug 1999.

[26] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[27] K.J. Cassidy, K.C. Gross, and A. Malekpour. Advanced pattern recognition for detection of complex software aging phenomena in online transaction processing servers. *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 478–482, 2002.

[28] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic, internet services. citeseer.ist.psu.edu/chen02pinpoint.html, 2002.

[29] Mike Chen, Alice X. Zheng, Jim Lloyd, Michael I. Jordan, and Eric A. Brewer. Failure diagnosis using decision trees. In *Proceedings of the 1st International Conference on Autonomic Computing 17-19 May 2004 New York NY USA*, 2004.

[30] Mike Y. Chen, Anthony Accardi, Emre K?c?man, Jim Lloyd, Dave Patterson, O Fox, and Eric Brewer. Path-based failure and evolution management. In *In NSDI*, pages 309–322. USENIX Association, 2004.

[31] 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.

[32] J.A. Clark and D.K. Pradhan. Fault injection: a method for validating computer-system dependability. *Computer*, 28(6):47–56, Jun 1995.

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

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

[35] Ira Cohen, Moises Goldszmidt, Terence Kelly, Julie Symons, and Jeffrey S. Chase. Correlating instrumentation data to system states: a building block for automated diagnosis and control. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[36] Micorsoft Corporation. Using the Edit and Continue Feature in C# 2.0. http://msdn.microsoft.com/en-us/library/ms379578(VS.80).aspx, 2004.

[37] Diamantino Costa, Tiago Rilho, and Henrique Madeira. Joint evaluation of performance and robustness of a cots dbms through fault-injection. In *DSN '00: Proceedings of the 2000 International Conference on Dependable Systems and Networks (formerly FTCS-30 and DCCA-8)*, pages 251–260, Washington, DC, USA, 2000. IEEE Computer Society.

[38] Ana Paula Couto da Silva, Pablo Rodriguez-Bocca, and Gerardo Rubino. Coupling qoe with dependability through models with failures. *Proceedings of the 8th International Workshop on Performability Modeling of Computer and Communication Systems*, 2007.

[39] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel 2nd Edition*. O'Reilly, 2002.

[40] DARPA. Dasada (dynamic assembly for system adaptability, dependability, and assurance) project. http://www.schafercorp-ballston.com/dasada/index2.html, 2000.

[41] DARPA. Dynamic assembly for system adaptability, dependability, and assurance semi-annual review november 2000. http://www.schafercorp-ballston.com/dasada/DASADANov00PADw_obudget.ppt, 2000.

[42] Edmundo de Souza e Silva and H. Richard Gail. Calculating availability and performability measures of repairable computer systems using randomization. *J. ACM*, 36(1):171–193, 1989.

[43] John DeVale. Measuring operating system robustness. Master's thesis, Carnegie Mellon University.

[44] Yixin Diao, Joseph L. Hellerstein, Sujay Parekh, Rean Griffith, Gail Kaiser, and Dan Phung. Self-managing systems: A control theory foundation. In *Proceedings of 2nd IEEE International Workshop on Engineering of Autonomic Systems*, 2005.

[45] Yixin Diao, Joseph L. Hellerstein, Adam J. Storm, Maheswaran Surendra, Sam Lightstone, Sujay Parekh, and Christian Garcia-Arellano. Using mimo linear control for load balancing in computing systems. In *Proceedings of the American Control Conference*, 2004.

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

[47] 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.

[48] 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.

[49] R.S. Fabry. How to design a system in which modules can be changed on the fly. In *Proceedings of International Conference on Software Engineering*, 1976.

[50] P. Folkesson, S. Svensson, and J. Karlsson. A comparison of simulation based and scan chain implemented fault injection. In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, page 284, Washington, DC, USA, 1998. IEEE Computer Society.

[51] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *NSDI*. USENIX, 2007.

[52] The Apache Foundation. Welcome to apache logging services. http://logging.apache.org/, 2003.

[53] The Apache Software Foundation. Apache tomcat. http://tomcat.apache.org/.

[54] The Eclipse Foundation. Faq what is hot code replace? http://wiki.eclipse.org/FAQ_What_is_hot_code_replace

[55] The OWASP Foundation. The open web application security project (owasp). http://www.owasp.org/index.php/About_OWASP, 2007.

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

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

[58] Peter W. Gill. Probing for a continual validation prototype. Masters dissertation, Worcester Polytechnic Institute, 2001. http://www.wpi.edu/Pubs/ETD/Available/etd-0826101-235008/.

[59] Swapna S. Gokhale and Kishor S. Trivedi. Analytical models for architecture-based software reliability prediction: A unification framework. *IEEE Transactions on Reliability*, 55(4):578–590, 2006.

[60] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

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

[62] 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.

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

[64] 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.

[65] Rean Griffith, Ritika Virmani, and Gail Kaiser. RAS-Models: A Building Block for Self-Healing Benchmarks. In *8th International Workshop on Performability Modeling of Computer and Communication Systems (PMCCS-8)*, 2007.

[66] Rean Griffith, Ritika Virmani, and Gail Kaiser. The Role of Reliability, Availability and Serviceability (RAS) Models in the Design and Evaluation of Self-Healing Systems. In *International Conference on Self-Organization and Autonomous Systems (SOAS) in Computing and Communications*, 2007.

[67] Rean Griffith, Ritika Virmani, and Gail Kaiser. Tools and techniques for designing and evaluating self-healing systems. http://www1.cs.columbia.edu/ rg2023/pubs/Tools%20and%20Techniques%20for%20Designing%20and%20Evaluating%20Self-Healing%20Systems.pdf, 2007.

[68] 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.

[69] Gunter Bolch and Stefan Greiner and Herman de Meer and Kishor S. Trivedi. *Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications 2nd Edition*. Wiley, 2006.

[70] S. Han, K. Shin, and H. Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems. citeseer.ist.psu.edu/han95doctor.html, 1995.

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

[72] David I. Heimann, Nitin Mittal, and Kishor S. Trivedi. Availability and reliability modeling for computer systems. pages 175–233, 1990.

[73] George T. Heineman. A model for designing adaptable software components. In *Proceedings of the 22nd Annual International Computer Software and Applications Conference*, 1998.

[74] George T. Heineman, Paul Calnan, and Ben Kurtz. Active interface development environment (aide). http://web.cs.wpi.edu/ heineman/dasada/, 2001.

[75] Michael W. Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation Snowbird Utah USA June 20-22 2001. SIGPLAN Notices 36(5) (May 2001) ACM 2001 ISBN 1581134142*, pages 13 – 23, 2001.

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

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

[78] Yennun Huang, Chandra Kintala, Nick Kolettis, and N. Dudley Fulton. Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing Pasadena CA June 1995*, pages 381 – 390, 1995.

[79] IBM. Autonomic Computing: IBM's Perspective on the State of Information Technology, October 2001.

[80] Qumranet Inc. Kernel based virtual machines. http://kvm.qumranet.com/kvmwiki, 2007.

[81] StackSafe Inc. Improve Business Uptime and Resiliency through a New Model for Software Infrastructure Testing by IT Operations. http://www.stacksafe.com/uploads/PDFs/StackSafe_White_Paper.pdf, 2007.

[82] StackSafe Inc. Improving Uptime and Resiliency Through Software Infrastructure Testing for IT Operations: StackSafe® Test Center. http://www.stacksafe.com/uploads/PDFs/StackSafe_Product_Brief.pdf, 2007.

[83] VMWare Inc. http://www.vmware.com/.

[84] VMWare Inc. Convert physical machines to virtual machines – free. http://www.vmware.com/products/converter/get.html, 2007.

[85] VMWare Inc. Vmware esx server. http://www.vmware.com/products/esxi/, 2008.

[86] Ingo Rammer. *Advanced .NET Remoting (C# Edition) (Paperback)*. Apress, 2002.

[87] Information Society Technologies (IST). Dependability benchmarking project final report. http://www.laas.fr/DBench/Final/DBench-complete-report.pdf, 2004.

[88] Philip Gross Janak Parekh, Gail Kaiser and Giuseppe Valetto. Retrofitting autonomic capabilities onto legacy systems. *Journal of Cluster Computing*, April 2006.

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

[90] Joseph L. Hellerstein et al. *Feedback Control of Computing Systems*. Wiley-Interscience, 2004.

[91] E.G. Coffman Jr., Z. Ge, Vishal Misra, and Don Towsley. Network resilience: Exploring cascading failures within bgp. In *Allerton Conference on Communication, Control and Computing*, October 2002.

[92] G. KAISER and A. DOSSICK. A mobile agent approach to lightweight process workflow. citeseer.ist.psu.edu/kaiser99mobile.html, 1999.

[93] G. Kaiser, P. Gross, G. Kc, J. Parekh, and G. Valetto. An approach to autonomizing legacy systems. citeseer.ist.psu.edu/kaiser02approach.html, 2002.

[94] Gail Kaiser, Janak Parekh, Phil Gross, and Giuseppe Valetto. 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.

[95] 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.

[96] K. Kanoun and H. Madeira. A framework for dependability benchmarking. citeseer.ist.psu.edu/kanoun02framework.html, 2002.

[97] Karama Kanoun, Mohamed Kaâniche, and Jean-Claude Laprie. Qualitative and quantitative reliability assessment. *IEEE Softw.*, 14(2):77–87, 1997.

[98] Heinz Kantz and Kishor S. Trivedi. Reliability modeling of the mars system: A case study in the use of different tools and techniques. In *PNPM*, pages 268–277, 1991.

[99] 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.

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

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

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

[103] J. Kramer and J. Magee. A model for change management. *Distributed Computing Systems in the 1990s, 1988. Proceedings., Workshop on the Future Trends of*, pages 286–295, 14-16 Sep 1988.

[104] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. In *Proceedings of IEEE Transactions on Software Engineering November 1990 Vol. 16 No. 11*, pages 1293 – 1306, 1990.

[105] 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.

[106] John Lam. CLAW: Cross-Language Load-Time Aspect Weaving on Microsoft's CLR. http://www.iunknown.com/000092.html, 2002.

[107] 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.

[108] 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.

[109] Lei Li, K. Vaidyanathan, and K.S. Trivedi. An approach for estimation of software aging in a web server. *Empirical Software Engineering, 2002. Proceedings. 2002 International Symposium n*, pages 91–100, 2002.

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

[111] 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.

[112] H. Madeira, J. Carreira, and J. Silva. Injection of faults in complex computers. citeseer.ist.psu.edu/madeira95injection.html, 1995.

[113] H. Madeira and P. Koopman. Dependability benchmarking: making choices in an n-dimensional problem space. citeseer.csail.mit.edu/madeira01dependability.html, 2001.

[114] Manish Malhotra and Kishor S. Trivedi. Reliability analysis of redundant arrays of inexpensive disks. *J. Parallel Distrib. Comput.*, 17(1-2):146–151, 1993.

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

[116] Eliane Martins, Cecilia 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.

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

[118] Julie McCann and Marcus Huebscher. Evaluation issues in autonomic computing. In *International Workshop on Agents and Autonomic Computing and Grid Enabled Virtual Organizations*, 2004.

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

[120] J.F. Meyer. On evaluating the performability of degradable computing systems. *IEEE Transactions on Computers*, 29(8):720–731, 1980.

[121] M.H.A. Davis. *Markov Models and Optimization*. Chapman & Hall, 1993.

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

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

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

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

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

[127] microsoft.public.dotnet.framework.clr. Icorprofilerinfo::setfunctionrejit causes deadlock. http://www.dotnet247.com/247reference/msgs/58/290727.aspx.

[128] Sun Microsystems. Java 2 platform, enterprise edition (j2ee) overview. http://java.sun.com/j2ee/overview.html, 1999.

[129] Sun Microsystems. The essentials of filters. http://java.sun.com/products/servlet/Filters.html, 2001.

[130] Sun Microsystems. Java platform debugger architecture - architecture overview. http://java.sun.com/j2se/1.4.2/docs/guide/jpda/architecture.html, 2001.

[131] Sun Microsystems. The java hotspot virtual machine v1.4.1, d2. http://java.sun.com/products/hotspot/docs/ whitepaper/Java_Hotspot_v1.4.1/JHS_141_WP_d2a.pdf, 2002.

[132] Sun Microsystems. The Java Native Interface Programmer's Guide and Specification. http://java.sun.com/docs/books/jni/html/titlepage.html, 2002.

[133] Sun Microsystems. Introduction to jmx technology. http://java.sun.com/j2se/1.5.0/docs/guide/jmx/overview/intro.html#wp5529, 2004.

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

[135] Sun Microsystems. Driver hardening test harness. http://docs.sun.com/app/docs/doc/819-3196/gemgi, 2008.

[136] Sun Microsystems. Writing device drivers. http://dlc.sun.com/pdf/819-3196/819-3196.pdf, 2008.

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

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

[139] Vishal Misra. *Stochastic Models for Network Traffic*. Ph.D. dissertation, University of Massachusetts Amherst, 2000.

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

[141] ObjectWeb. Rubis: Rice university bidding system project web page. http://rubis.objectweb.org/, 2002.

[142] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do internet services fail, and what can be done about it. citeseer.ist.psu.edu/oppenheimer03why.html, 2003.

[143] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. In *Proceedings of IEEE Intelligent Systems archive Volume 14, Issue 3 (May 1999)*, pages 54 – 62, 1999.

[144] Peyman Oreizy, Nenad Medovic, and Richard N. Taylor. Architecturebased runtime software evolution. In *Proceedings of the International Conference on Software Engineering 1998*, pages 177 – 186, 1998.

[145] Sujay Parekh, Kevin Rosey, Yixin Diao, Victor Changy, Joseph Hellerstein, Sam Lightstoney, and Matthew Huras. Throttling utilities in the ibm db2 universal database server. http://www.research.ibm.com/PM/rc23163.pdf, 2004.

[146] Paul Pazandak and David Wells. Probemeister: Distributed runtime software instrumentation. In *First International Workshop on Unanticipated Software Evolution*, 2002.

[147] PHARM research team University of Wisconsin-Madison. Java tpc-w implementation distribution. http://www.ece.wisc.edu/ pharm/tpcw.shtml, 2003.

[148] 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.

[149] The Apache Jakarta Project. Bcel - byte code engineering library (bcel). http://jakarta.apache.org/bcel/manual.html, 2006.

[150] The Linux Virtual Server Project. The linux virtual server project. http://www.austintek.com/LVS/LVS-HOWTO/mini-HOWTO/LVS-mini-HOWTO.html, 2001.

[151] The Linux Virtual Server Project. The linux virtual server project. http://www.linuxvirtualserver.org/, 2002.

[152] B. Randell. System structure for software fault tolerance. In *Proceedings of the International Conference on Reliable Software*, pages 437 – 449, 1975.

[153] James Reason. *Human Error*. Cambridge University Press, 1990.

[154] UC Berkeley Reliable Adaptive Distribued (RAD) Systems Lab. X-trace wiki. http://www.x-trace.net/wiki/doku.php, 2008.

[155] Richard McDougall and Jim Mauro. *Solaris Internals - Solaris 10 and OpenSolaris Kernel Architecture, 2nd Edition*. Prentice Hall, 2006.

[156] Jacob Rief and Simon Horman. ldirectord. http://www.vergenet.net/linux/ldirectord/, 1999.

[157] R. C. Cheung S. S. Yau. Design of self-checking software. In *Proceedings of the International Conference on Reliable Software*, pages 450 – 455, 1975.

[158] 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.

[159] 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.

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

[161] Willaim H. Sanders and John F. Meyer. Stochastic activity networks: formal definitions and concepts. pages 315–343, 2002.

[162] 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.

[163] Bianca Schroeder and Garth A. Gibson. Disk failures in the real world: what does an mttf of 1,000,000 hours mean to you? In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, page 1, Berkeley, CA, USA, 2007. USENIX Association.

[164] Bianca Schroeder and Adam Wierman. Open versus closed: A cautionary tale. citeseer.ist.psu.edu/751284.html, 2006.

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

[166] Mark E. Segal and Ophir Frieder. On-The-Fly Program Modification Systems for Dynamic Updating. *IEEE Software*, 10(2), March 1993.

[167] B. SEGALL, D. ARNOLD, J. BOOT, M. HENDERSON, and T. PHELPS. Content based routing with elvin. citeseer.ist.psu.edu/segall00content.html, 2000.

[168] Margo I. Seltzer, David Krinsky, Keith A. Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In *Workshop on Hot Topics in Operating Systems*, pages 102–, 1999.

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

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

[171] 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.

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

[173] Luis Silva, Javier Alonso, Paulo Silva, Jordi Torres, and Artur Andrzejak. Using virtualization to improve software rejuvenation. 2007.

[174] SourceForge.Net. The world's largest Open Source software development web site. http://www.sourceforge.net.

[175] SourceForge.NET. Alchemi [.NET Grid Computing Framework]: Summary. http://sourceforge.net/projects/alchemi/, 2004.

[176] SPEC. Specjappserver2004 (java application server). http://www.spec.org/jAppServer2004/.

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

[178] 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.

[179] Stacksafe. It ops research report: Downtime and other top concerns. www.stacksafe.com, 2007.

[180] L. Stojanovic, J. Schneider, A. Maedche, S. Libischer, R. Studer, Th. Lumpp, A. Abecker, G. Breiter, and J. Dinger. The role of ontologies in autonomic computing systems. *IBM Systems Journal – Unstructured Information Management*, 43(3), 2004.

[181] David Stutz, Ted Neward, and Geoff Shilling. *Shared Source CLI*. O'Reilly & Associates Inc., 2003.

[182] Neeraj Suri and Purnendu Sinha. On the use of formal techniques for validation. In *Symposium on Fault-Tolerant Computing*, pages 390–399, 1998.

[183] Citrix Systems. What is xen? http://www.xen.org/, 2005.

[184] 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.

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

[186] Caucho Technology. Resin high performance, open source application server. http://www.caucho.com/.

[187] Toby J. Teorey and Wee Teck Ng. Dependability and performance measures for the database practitioner. *IEEE Trans. on Knowl. and Data Eng.*, 10(3):499–503, 1998.

[188] The University of Melbourne. Alchemi – plug & play grid computing. http://www.alchemi.net/, 2004.

[189] 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.

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

[191] 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.

[192] G. Valetto, G. Kaiser, and D. Phung. A uniform programming abstraction for effecting autonomic adaptations onto software systems. *Autonomic Computing, 2005. ICAC 2005. Proceedings. Second International Conference on*, pages 286–297, 13-16 June 2005.

[193] Giuseppe Valetto, Gail E. Kaiser, and Gaurav S. Kc. A mobile agent approach to process-based dynamic adaptation of complex software systems. In *EWSPT '01: Proceedings of the 8th European Workshop on Software Process Technology*, pages 102–116, London, UK, 2001. Springer-Verlag.

[194] Wikipedia. Write once, run anywhere. http://en.wikipedia.org/wiki/Write_once,_run_anywhere, 1996.

[195] Don Wilson, Brendan Murphy, and Lisa Spainhower. Progress on defining standardized classes for comparing the dependability of computer systems. citeseer.ist.psu.edu/wilson02progress.html, 2002.

[196] Yixin Diao et al. A control theory foundation for self-managingcomputing systems. *IEEE Journal on Selected Areas in Communications AUTONOMIC COMMUNICATION SYSTEMS*, December 2005.

[197] Stanley B. Zdonik. Maintaining consistency in a database with changing types. In *Proceedings of the 1986 SIGPLAN workshop on Object-oriented programming Yorktown Heights, New York, United States*, pages 120 – 127, 1986.

[198] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. Safedrive: safe and recoverable extensions using language-based techniques. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 45–60, Berkeley, CA, USA, 2006. USENIX Association.

Appendix A

Experience with StackSafe's Test Center

The StackSafe Test Center is a pre-production staging, testing and analysis platform targeted at IT Operations teams [81]. The Test Center establishes a virtualized sandbox (see Figure A.1) in which end users (IT Operations staff) can test and analyze systems in a representative production environment. Included in the Test Center are tools that allow end-users to create copies of the production servers that make up a software infrastructure stack. System images can be imported from physical and/or virtual machines; further, the Test Center's use of virtualization allows imported images to be networked into a working software infrastructure stack, which mirrors a production configuration.

The Test Center is intended to support testing activities that cover a number of areas including, but not limited to: application assembly and validation, performance tuning, security/risk assessments, patch testing, continuity and disaster recovery (playing what-if scenarios with production), diagnostics and root cause analysis, and reliability testing [81]. In this chapter we familiarize ourselves with the Test Center, using it as a testbed environment for conducting RAS evaluations. We re-create the VM-Rejuv deployment used in §5.5 (the load-balanced TPC-W web-application stack), discuss the Test Center configuration used, measure the failure-free performance of the system and re-run our

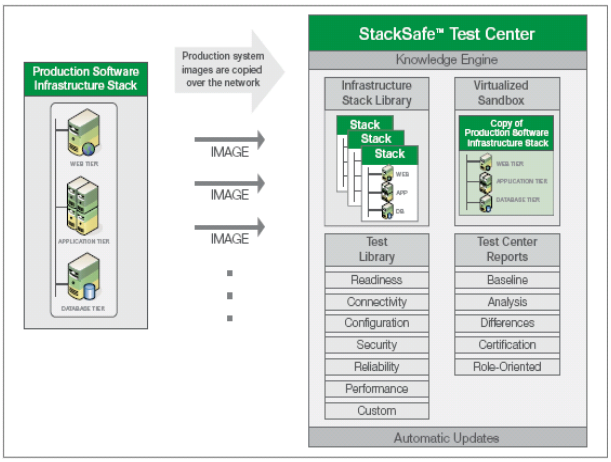

Figure A.1: StackSafe Test Center – source Improve Business Uptime and Resiliency through a New Model for Software Infrastructure Testing by IT Operations [81]

fault-injection experiments inside the Test Center.

A.1 Experimental Setup

Our experimental platform uses a Test Center configured with 8 GB RAM, 512 MB swap, 4 Intel Xeon E5345 Dual Core 64-bit 2.33 GHz CPUs and a 1.5 TB disk array running StackSafe™ Test Center release 5 on a Linux 2.6.18 SMP kernel. Test Center release 5 uses Xen v3.1.0 [183] to provide virtualization services. Outbound network access is enabled in the Test Center to allow imported machines (guest hosts) to connect to resources on the same network as the Test Center, e.g., database servers, Lightweight Directory Access Protocol (LDAP) servers, etc. Enabling outbound network access is also a prerequisite for allowing inbound connections to individual guest hosts, i.e., assigning an externally visible static IP address to a guest host.

Our original VM-Rejuv deployment (§5.5) consists of three VMWare GSX virtual machines: VM1, the Linux Virtual Server (LVS) load-balancer (IPVS v1.2.1 and ipvsadm v1.24) and database server (MySQL 5.0.27), and VMs 2 and 3, the Apache Tomcat web/application servers (Apache Tomcat v5.5.20) hosting the TPC-W web-application classes. All three

VMs run Centos 5.0 on Linux 2.6.18-8.el5 (see [82] for a list of operating systems that can be imported into the Test Center as guest hosts).

Using the Test Center's Import CD we boot these three VMs from the import CD and clone them into the Test Center [1]. We enable outbound connections on each VM and we assign a static IP address to the imported VM1 guest host and enable inbound connections to it so that the TPC-W workload generator (remote browser emulators/RBEs) can access the TPC-W web-application from IP addresses external to the Test Center. We assemble VMs 1, 2 and 3 into an Infrastructure Stack inside the Test Center, which allows us to treat these three VMs as a single logical unit, e.g, issuing start/stop commands or running tests and reports against all components.

Since the Test Center is configured with 8GB of RAM, each imported VM is allocated 1 GB of RAM, instead of the 512 MB, 384 MB and 384 MB allocations used in §5.5 for VMs 1, 2 and 3 respectively [2]. The LVS load-balancer in the VM1 guest host is configured to direct web-requests to the VM2 and VM3 guest hosts using LVS-NAT[3] as before and we test the failure-free operation of the TPC-W web-application deployed under VM-Rejuv inside the Test Center.

We simulate a load of 50 TPC-W clients using the Shopping Mix as their web-interaction strategy.

During ten failure free runs each lasting 22 minutes the average number of client-side interactions recorded is 9315.9 ± 120.6. Figures A.2 and A.3 show a 16 minute sample of the throughput and response time data reported by VM probes during one of our failure-free runs. The average throughput is ~6 requests per second and the average response time is ~27 ms.

[1] The import of each 8 GB VM harddisk takes ~20 mins to complete on our network.

[2] The current release of the Test Center limits the RAM allocated to each imported guest host to 2GB.

[3] The Test Center uses the 10.216.71.x and 172.30.8.x networks internally to provide IP addresses for guest hosts. In our experiments we also assigned 192.168.1.x IP addresses to the guest host network interfaces using OS-level configuration files.

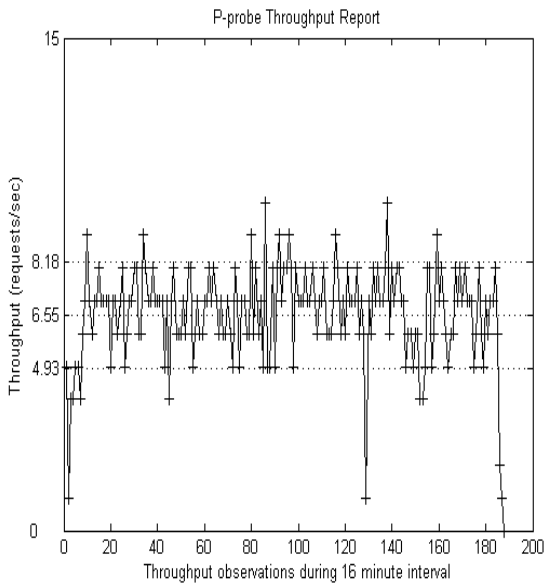

Figure A.2: Test Center: VM-Rejuv baseline throughput sample

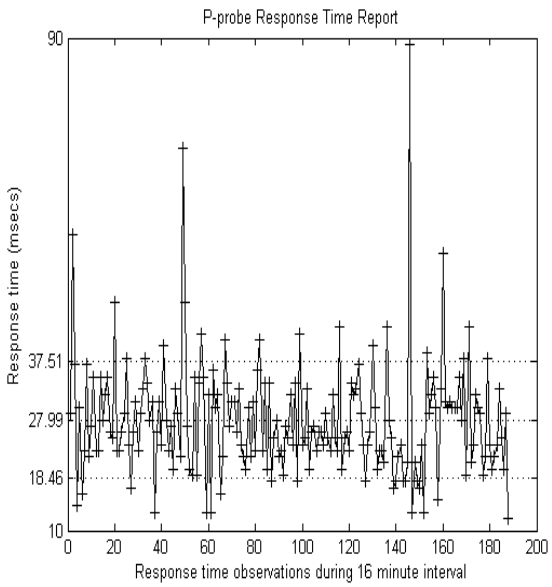

Figure A.3: Test Center: VM-Rejuv baseline response time sample

Setting VM-Rejuv's response time violation threshold at the mean response time (27 ms) and re-running the workload of 50 clients we observe an average of 2 rejuvenation actions per run over 10 runs. The mean failover time is 86 ms and the average pre-rejuvenation delay window size is 19,778 msecs.

In our fault-injection experiments we subject both Tomcat application servers deployed under VM-Rejuv to memory leaks that result in resource exhaustion within 11.1 minutes (666.271 seconds) of running the 50 client TPC-W workload. We set VM-Rejuv's response time violation threshold to the mean response time of the failure free runs (27 ms) and measure the frequency of rejuvenations, the VM failover time and the size of the pre-rejuvenation delay window. Over five fault-injection runs, each lasting 22 minutes, we record an average of 5 rejuvenations per run (mean rejuvenation interval of 256.69 seconds) with mean switchover time of 68 ms and mean pre-rejuvenation delay window size of 30,202 ms (Table A.1).

Using a mean rejuvenation interval of 256.69 seconds, mean rejuvenation window size of 30,202.37 msecs and a mean failover time of 68.10 msecs we score this VM-Rejuv deployment using the RAS model in Figure 5.13 (see Table 5.7 for parameter descriptions).

Run #	Rejuvenation actions	Rejuvenation interval (secs)	Failover time (msecs)	Pre-rejuvenation delay window (msecs)
1	5	257.85	64.60	36,090.60
2	5	263.11	64.40	24,775.60
3	4	242.92	140.50	38,625.25
4	5	238.78	27.40	21,859.20
5	5	280.78	43.60	29,661.20
Avg	**4.8**	**256.69**	**68.10**	**30,202.37**

Table A.1: Test Center: VM-Rejuv subjected to memory leaks

The mean time to restart Tomcat during the memory leak experiments is 2 seconds and the mean time to detect a server outage (via the ldirectord watchdog) is 5 seconds.

The steady-state probabilities of the VM-Rejuv model are shown in Table A.2 and model analysis results are shown in Table A.3.

π_0	0.883192
π_1	0.099412
π_2	0.009533
π_3	0.006628
π_4	0.000090
π_5	0.000818
π_6	0.000327

Table A.2: Test Center: VM-Rejuv steady state probabilities – memleak scenario

Using the scoring model we can estimate the number of active VM failures expected during rejuvenation actions per day, i.e., the frequency of transitions from S_1 to S_5 ($F_{S_1 \to S_5}$) plus the frequency of transitions from S_2 to S_5 ($F_{S_2 \to S_5}$). This we estimate at 14 per day under the failure conditions used in our experiments (1 memory-leak failure every 11.1 minutes).

From the steady-state probabilities of the model we estimate that the deployment spends ~88% of the time in its normal operating mode/configuration, π_0, and ~11% of its time rejuvenating ($\pi_1 + \pi_2$). While rejuvenations are taking place client-requests are serviced by the standby VM; as a result the system would be considered UP from the client's perspective in states $\{S_0, S_1, S_2\}$ – UP 1428.7 minutes per day (99.21%) and DOWN 11.3 minutes per

day (0.79%). Administrators on the other hand may consider the system to be UP if it is in state S_0 since states S_1 and S_2 represent a window of vulnerability. From the administrator's perspective the system is UP 1271.8 minutes per day (88.32%) and DOWN 168.2 minutes per day (11.68%), of which 157 minutes are spent performing rejuvenation actions.

Measure	Metrics	Results
Reliability	Frequency of active VM failures during rejuvenation per day $F_{S_1 \to S_5} + F_{S_2 \to S_5}$	14.127668
Availability	Basic steady-state availability ($UP_{admin} = \{S_0\}$)	0.883192
	Tolerance availability ($UP_{client} = \{S_0, S_1, S_2\}$)	0.992137
Serviceability	Mean-time to system restoration ($UP_{admin} = \{S_0\}$)	24,507 msecs
	Mean-time to system restoration($UP_{client} = \{S_0, S_1, S_2\}$)	5,280 msecs

Table A.3: Test Center: Summary of VM-Rejuv RAS model analysis results

A.2 Summary

In this chapter we use StackSafe's Test Center as a platform/environment for conducting RAS evaluations. We import and configure a load-balanced TPC-W web-application (deployed under VM-Rejuv[173]) in the Test Center to use as our target system. The guest hosts used for the load-balancer, and the two application server components are imported from the VMWare GSX virtual machines created for the experiments in §5.5.

In preparing for our fault-injection experiments we rely on the Test Center's ability to clone/import existing production systems and assemble them into infrastructure stacks. Further, we use Test Center's support for incoming and outgoing network connections (between guest hosts and resources on the same network as the Test Center, e.g., load generators, database servers or directory services) to create a testing environment that closely mirrors the production environment. Our cloned environment differs from the environment in §5.5 only in the amount of resources (RAM) assigned to the virtual machines – we were able to allocate more memory to the VMs imported into the Test Center since it is

configured with 8 GB of RAM in contrast to the 2 GB of RAM installed on the physical machine hosting the three VM-Rejuv virtual machines in §5.5.

We combine our runtime fault-injection tools and RAS models with the testing environment/infrastructure provided by the Test Center (physical/virtual machine cloning, the virtualization of systems and networks, support for network connections between imported guest hosts and external resources) to demonstrate a practical approach to performing RAS evaluations, where the requirement that RAS evaluations be carried out on production systems may be satisfied via sophisticated testing/staging tools capable of replicating portions of production environments. Further, runtime instrumentation tools, runtime fault-injection tools and RAS modeling tools/environments (probe placement, failure-scenario design interfaces, etc.) can be included in the set of tools/services provided by virtualized staging environments for integrated reliability, availability and serviceability testing and evaluation. Finally, the ability to integrate supporting tools/services for RAS evaluations (e.g., runtime fault-injection tools, RAS modeling environments, etc.) directly into a virtualized testing/staging environment like the Test Center represent an opportunity to reduce the number of disparate stand-alone tools/interfaces that evaluators need to content with when preparing to conduct a set of RAS evaluations.