A Runtime Adaptation Framework for Native C and Bytecode Applications

Rean Griffith
Department of Computer Science
Columbia University
Email: rg2023@cs.columbia.edu

Gail Kaiser
Department of Computer Science
Columbia University
Email: kaiser@cs.columbia.edu

***Abstract*— The need for self-healing software to respond with a reactive, proactive or preventative action as a result of changes in its environment has added the non-functional requirement of adaptation to the list of facilities expected in self-managing systems. The adaptations we are concerned with assist with problem detection, diagnosis and remediation. Many existing computing systems do not include such adaptation mechanisms, as a result these systems either need to be re-designed to include them or there needs to be a mechanism for retro-fitting these mechanisms. The purpose of the adaptation mechanisms is to ease the job of the system administrator with respect to managing software systems. This paper introduces Kheiron, a framework for facilitating adaptations in running programs in a variety of execution environments without requiring the re-design of the application. Kheiron 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 self-healing systems. We also describe the concepts and techniques used to retro-fit adaptations onto existing systems in the various execution environments.**

I. Introduction

System adaptation has been highlighted as a necessary feature of autonomic software systems [1]. In the realm of self-healing software we are concerned primarily with adaptations that effect problem diagnosis – via consistency checks or *ghost transactions*[1] – and remediation – in the form of reconfiguration or repair. In many situations adaptations must occur while the system executes so as to maintain some degree of availability. Having a critical software system operate in a degraded mode is preferable to taking the system offline to perform scheduled (or unscheduled) reconfiguration or repair activities [2], [3].

System designers have two alternatives when it comes to realizing software systems capable of adaptation. Adaptation mechanisms can be built into the system – as done in the K42 operating system [4] – or such functionality can be retro-fitted onto them using externalized architectures like KX [5] or Rainbow [6]. While arguments can be made for either approach, the retrofit approach provides more flexibility. "Baked-in" adaptation mechanisms restrict the analysis and reuse of said mechanisms. Further, it is difficult to evolve (via updates and extensions) the adaptation mechanisms without affecting the execution and deployment of the target system [7].

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. In this paper, we focus on fine-grained adaptations, those interacting with individual components, sub-systems or methods e.g. restarting/refreshing individual components or sub-systems, or augmenting methods.

Whereas the retro-fit approach is attractive because it does not require a re-design of the system and it is possible to separately evolve the target system and the adaptation mechanisms, it is not always easy to achieve. A major challenge is that of actually **retro-fitting fine-grained adaptation mechanisms onto existing/legacy systems**[2].

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

Our contribution is the ability to transparently retro-fit new functionality (for the purpose of diagnosing problems and resolving problems where possible) onto existing software

[1]A ghost transaction is a special form of a self-test/diagnosis targeting a specific subset of subsystems or components.

[2]For purposes of discussion we define a legacy system as 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.

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 transfer control over to the adaptation engine, which effects the desired adaptations in the running application.

The remainder of the paper is organized as follows; §II motivates retro-fitting fine-grained adaptation mechanisms onto existing systems and presents a number of specific adaptations and their potential benefits. §III-A gives a working definition of an execution environment and describes two classes of execution environments – *managed* and *unmanaged*. §III-B outlines some challenges associated with performing adaptations at the execution environment level. §IV describes the mechanisms and concepts used to adapt running bytecode-based applications, using our Kheiron/JVM implementation and its performance overhead. Kheiron/CLR, our first adaptation framework, targets Microsoft Intermediate Language (MSIL) bytecode applications and is discussed in [8], [9], [10]. §V compares and contrasts runtime adaptation in an unmanaged execution environment with runtime adaptation in a managed execution environment. We also present Kheiron/C and discuss some experimental results of the performance impact imposed on target systems and §V-E describes a special case of adaptation – dynamically adding fault detection and recovery to running compiled C programs via selectively emulating individual functions.§VI covers related work and finally §VII presents our conclusions and future work.

II. Motivation

The ability to adapt is critical for self-healing systems [1]. 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 self-healing mechanisms. As a hypothetical example, consider an existing computer system which periodically raises questions about its stability – evidenced by user complaints concerning lost work and system unavailability.

A system administrator may decide to adapt the system by injecting instrumentation into specific sub-systems to further investigate the problem. Information collected via the inserted instrumentation is then routed to separate analysis components and/or visualization consoles. Analysis and visualization leads to the diagnosis hypothesis that identifies (with high probability) specific conditions of increasing resource demands as the primary contributing factors. A possible post-analysis adaptation would be the insertion of specific monitoring code to detect similar conditions coupled with the insertion of mechanisms, which when triggered by the monitoring code, intercept and queue requests to specific sub-systems while simultaneously restarting these sub-systems to force the release of key resources – analogous to a *micro-reboot* [11] except there is no presumption that the sub-systems were designed as crash-only components supporting APIs for recovery and shutdown as proposed in [11].

There are a number of specific fine-grained adaptations that can be retro-fitted onto existing systems to aid problem detection, diagnosis and in some cases remediation via performing reconfigurations or (temporary) repairs. In this paper we describe how our Kheiron implementations can be used to facilitate a number of fine-grained adaptations in running systems via leveraging facilities and properties of the execution environments hosting these systems.

These adaptations include (but are not limited to): **Inserting or removing system instrumentation** [12] 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) [10]. **Input filtering/audit** to detect misused APIs. **Initiating ghost transactions** against select components or subsystems and collecting the results to obtain more details about a problem. **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.

III. Background

A. Execution Environments

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

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

Managed executables/applications are represented in an abstract intermediate form expected by the managed execution environment. This abstract intermediate form consists of two main elements, *metadata* and *managed code*. Metadata describes the structural aspects of the application including classes, their members and attributes, and their relationships

with other classes [13]. 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[3]. 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.

B. Challenges of Runtime Adaptation via the Execution Environment

There are a number of properties of execution environments that make them attractive for effecting adaptations on running systems. They represent the lowest level (short of the hardware) at which changes could be made to a running program. Some may expose (reasonably standardized) facilities (e.g. profiling APIs [14], [15]) that allow the state of the program to be queried and manipulated. Further, other facilities (e.g. metadata APIs [16]) may support the discovery, inspection and manipulation of program elements e.g. type definitions and structures. Finally, there may be mechanisms which can be employed to alter to the execution of the running system.

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

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

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

IV. Adapting Managed Applications

Kheiron/JVM leverages facilities exposed by Sun Microsystems' v1.5 implementation of the JVM, the Java HotspotVM, to dynamically attach/detach an engine capable of performing adaptations. Examples of adaptations include: adding instrumentation and performing consistency checks to improve problem detection and diagnosis, performing reconfigurations such as component replacements or component swaps, and performing repairs (where possible) to a target Java application while it executes.

A. Java HotspotVM Execution Model

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 Java HotspotVM 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 Java HotspotVM 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 [17]. Compiled methods remain cached in memory, and subsequent method calls jump directly into the native (compiled) version of the method.

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) [15]. 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, creating new type references and defining new strings we were required to roll our own APIs based on information provided in the Java Virtual Machine Specification [18].

B. Kheiron/JVM Operation

Kheiron/JVM is implemented as a single dynamic linked library (DLL), which includes a JVMTI agent. It consists of 2658 lines of C++ code and is divided into four main components. The **Execution Monitor** receives classfile load, class load and class prepare events from the JVM. The **Metadata Helper** wraps our metadata import interface, which is used to parse and read the classfile format. **Internal book-keeping structures** store the results of metadata resolutions. The **Bytecode and Metadata Transformer** wraps our metadata emit interface to write new metadata, e.g., adding new methods to a type, adding references to other classes and methods. It also generates, inserts and replaces bytecode in existing methods as directed by the Execution Monitor. Bytecode changes are committed using the RedefineClasses function exposed by the JVMTI. Active method invocations continue to use the old implementation of their method body while new invocations use the latest version.

Kheiron/JVM performs operations on type definitions, object instances and methods at various stages in the execution cycle 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.

Step 1 occurs at classfile load time, signaled by the **ClassFileLoadHook** JVMTI callback that precedes it. At this point the VM has obtained the classfile data 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 1, 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.

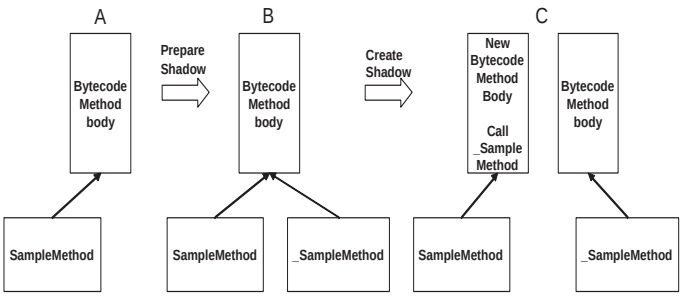

Fig. 1. *Preparing and Creating a Shadow Method*

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 1, 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 2, enables Kheiron/JVM to easily inject adaptation instructions into the wrapper as prologues and/or epilogues to shadow method calls.

```
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
```

Fig. 2. *Conceptual Diagram of a Wrapper*

To add a prologue to a method new bytecode instructions must prefix the existing bytecode instructions. The level of difficulty is the same whether we perform the insertion in the wrapper or the original method. Adding epilogues, however, presents more challenges. Intuitively, we want to insert instructions before control leaves a method. In the simple case, a method has a single return statement and the epilogue can be inserted right before that point. However, for methods with multiple return statements or exception handling routines, finding every possible return point can be an arduous task [19]. Using wrappers thus delivers a cleaner approach since we can ignore all of the complexity in the original method.

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 three-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 using our **IMetaDataEmit::DefineTypeRef** and **IMetaDataEmit::DefineNameAndTypeRef** methods. **Step 2**: Add references to the subset of the adaptation engine's methods that we wish to insert calls to, using **IMetaDataEmit::DefineMethodRef**. **Step 3**: 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, or reconfigurations and diagnostics of components. To persist the bytecode changes made to the method bodies of the wrappers, the Execution Monitor uses the RedefineClasses method of the JVMTI.

C. High-level Kheiron/JVM interfaces

To interact with Kheiron/JVM the most important APIs are: **AddModuleToObserve**, which informs Kheiron/JVM to intercept the the JVMTI's ClassFileLoadHook event and perform processing on the classfile for a named class. When the class is loaded, any necessary shadow methods are created via the **PrepareShadow** method. This API call also allows a user to specify whether shadow methods should be created for public, private and/or inherited methods. **AddPrologue** is used to specify the (wrapper) method of a named class to be augmented. The typical augmentation is to insert a call to a method in another named class before the wrapper method calls the shadow method. **AddEpilogue** is similar to AddPrologue except that the augmentation inserts a call to a method in another named class after the wrapper method calls the shadow method. AddPrologue and AddEpilogue use the RedefineClasses method of the JVMTI and can be called at anytime. If the method being augmented is currently on the stack, existing clients continue to use the original version, whereas subsequent method invocations use the new version [15].

D. Preliminary Results

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. In our evaluation we used the Java benchmarks SciMark v2.0[4] and Linpack[5].

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.

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 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. Note that we do not ask the Java HotspotVM to notify us on method entry/exit events since this can result in a slow down in some cases in excess of 5X. If adaptations were actually being performed then we expect the overheads measured to depend on the specifics of the adaptations.

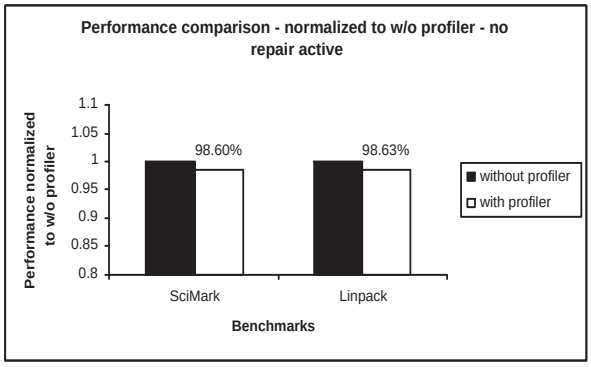

Fig. 3. *Overheads when no repair active*

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 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.

V. Adapting Unmanaged Applications

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

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) [20] object files, on the Linux platform.

A. 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

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

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

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 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 [21]. 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 4) 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.

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

```
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;
```

Fig. 4. *ELF Symbol Table Entry [20]*

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.

B. Kheiron/C Operation

Our current implementation of Kheiron/C relies on the Dyninst API [22] (v4.2.1) to interact with target applications while they execute. Dyninst presents an API for inserting new code into a running program. The program being modified is able to continue execution and does not need to be recompiled or relinked. Uses for Dyninst include, but are not limited to, runtime code-patching and performance steering in large/long-running applications.

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

To use the Dyninst terminology, Kheiron/C is implemented as a *mutator* (Figure 5), which uses the Dyninst API to attach to, and modify a running program. On the Linux platform, where we conducted our experiments, Dyninst relies on ptrace and the /proc filesystem facilities of the operating system to interact with running programs.

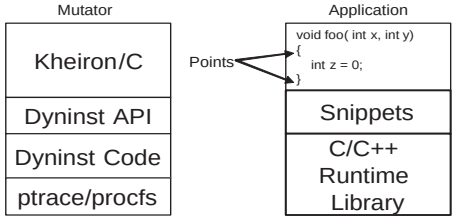

Fig. 5. *Kheiron/C*

Kheiron/C uses the Dyninst API to search for global or

local variables/data structures (in the scope of the insertion point) in the target program's address space, read and write values to existing variables, create new variables, load new shared libraries into the address space of the target program, and inject function calls to routines in loaded shared libraries as prologues/epilogues (at the points shown in Figure 5) for existing function calls in the target application. As an example, Kheiron/C could search for globally visible data structures e.g. the head of a linked list of abstract data types, and insert periodic checks of the list's consistency by injecting new function calls passing the linked-list head variable as a parameter.

To initiate an adaptation Kheiron/C attaches to a running application (or spawns a new application given the command line to use). The process of attaching causes the thread of the target application to be suspended. It then uses the Dyninst API to find the existing functions to instrument (each function abstraction has an associated call-before instrumentation point and a call-after instrumentation point). The target application needs to be built with symbol information for locating functions and variables to work – with stripped binaries Dyninst reports ∼95% accuracy locating functions and an ∼87% success rate instrumenting functions. The disparity between the percentage of functions located and the percentage of functions instrumented is attributed to difficulties in instrumenting code rather than failures in the analysis of stripped binaries [23]. Kheiron/C uses the Dyninst API to locate any "interesting" global structures or local variables in the scope of the intended instrumentation points. It then loads any external library/libraries that contain the desired adaptation logic and uses the Dyninst API to find the functions in the adaptation libraries, for which calls will be injected into the target application. Next, Kheiron/C constructs function call expressions (including passing any variables) and inserts them at the instrumentation points. Finally, Kheiron/C allows the target application to continue its execution.

C. High-level Kheiron/C interfaces

To interact with Kheiron/C the most important interfaces are: **LaunchProcess** and **AttachToProcess**, which creates a new process to interact with or interacts with an existing process, respectively. **LoadLibrary**, introduces a shared library into the address space of the target process. **AddPrologue** allows a user to specify a call to a named function to be inserted before a call to an existing function. Similarly, **AddEpilogue** inserts a call to a named function before the invocation of an existing function returns.

D. Preliminary Results

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.

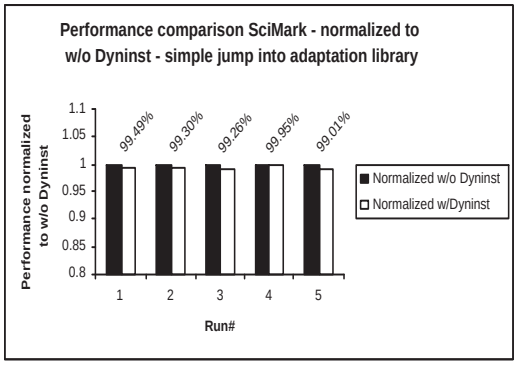

Fig. 6. *Overheads Simple Instrumentation*

As shown in Figure 6 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 the adaptation library followed by a return before executing the bodies of SOR_execute and SOR_num_flops. We expect that the overhead on overall program execution would depend largely on the operations performed while inside the adaptation library. Further, the time the SciMark process spends suspended while Kheiron/C performs the instrumentation is sub-second, ∼684 msecs ± 7.0686.

E. Injecting Selective Emulation

To enable applications to detect low-level faults and recover at the function level or, to enable portions of an application to be run in a computational sandbox, we describe an approach that allows portions of an executable to be run under the STEM x86 emulator. We use Kheiron/C to dynamically load the emulator into the target process' address space and emulate individual functions. STEM (Selective Transactional EMulation) is an instruction-level emulator – developed by Locasto et al. [24] – 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)[24].

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 7). 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'

```
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();
}
```

Fig. 7. *Inserting STEM via source code*

address-space, manage the CPU-to-STEM transition as well as the STEM-to-CPU transition.

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

Next, we focus on initializing STEM once it has been loaded into the target process' address space. The original version of STEM requires two things for correct initialization. First, the state of the machine before emulation begins must be saved – at the end of emulation STEM either commits its current state to the real CPU registers and applies the memory changes or STEM performs a rollback of the state of the CPU, restoring the saved register state, and undoes the memory changes made during emulation. Second, STEM's instruction pipeline needs to be correctly setup, including the calculation of the address of the first instruction to be emulated.

To correctly initialize our dynamically-loadable version of STEM we need to be able to effect the same register saving and instruction pipeline initialization as in the source-scenario. In the original version of STEM register saving is effected via the emulate_init macro, shown in Figure 7. 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[6] 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 [22].

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

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

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

the relocated instructions and continue the emulation of the remaining instructions of the function. After the initialization, the injected call to emulate_begin will cause STEM to begin its instruction fetch-decode-execute loop thus running the function under the emulator.

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

As a comparison, performing STEM injection using Pin 2.0 [25] would call for less machinations with respect to initializing STEM (i.e. the CPU-to-STEM transition). Pin maintains two copies of the program text in memory, the original program text and the instrumented version of the program text (generated just-in-time by Pin) hence, there is no need for trampolines, nor any need to save instructions dislocated by jumps into the trampoline as in the Dyninst case. Once STEM is loaded, its instruction pointer can simply be set to the base address of the function which will mark the beginning of the original un-instrumented version of the function. Further, Pin 2.0 guarantees that analysis code – code executed at instrumentation points – will be inlined into the instrumented version of the function, as long as the analysis code contains no branches [26]. This inlining guarantee should allow the CPU state-capture assembly instructions needed to initialize STEM's registers to be emitted inline in the instrumented version of the function, as occurs at the source level with the original version of STEM. However, we need to verify that inlining actually occurs and devise an appropriate strategy for the STEM-to-CPU transition.

VI. Related Work

Our Kheiron prototypes are concerned with facilitating very fine-grained adaptations in existing/legacy systems, whereas systems such as KX [5] and Rainbow [6] are concerned with coarser-grained adaptations. However, the Kheiron prototypes could be used as low-level mechanisms orchestrated/directed by these larger frameworks.

JOIE [27] 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 [28]. 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 [29] 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) [30] whereas Kheiron/JVM works with unmodified Sun JVMs. FIST modifies the Jikes RVM Just-in-Time compiler to insert a breakpoint into the prologue of method to generate an event when a method is entered to allow a response on the method entry event. Control transfer to instrumentation code can then occur when the compiled version of the method is executed. The Jikes RVM can be configured to always JIT-compile methods, however the unmodified Sun JVMs, v1.4x and v1.5x, do not support this configuration. As a result, Kheiron/JVM relies on bytecode rewriting to transfer control to instrumentation code as a response to method entry and/or method exit – transfer of control will occur with both the interpreted and compiled versions of methods.

A popular approach to performing fine-grained adaptations in managed applications is to use Aspect Oriented Programming (AOP). AOP is an approach to designing software that allows developers to modularize cross-cutting concerns [31] 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 [32], Aspect C# [33], those that perform weaving after compile time but before load time, e.g. Weave .NET [34], 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 [35] and CLAW [36]. 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 [11] and adaptive systems such as the K42 operating system [4] require upfront design-time effort to build in adaptation mechanisms. Our Kheiron implementations do not require special designed-in hooks, but they can take advantage of them if they exist. In the absence of designed-in hooks, our Khe-

iron 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' [37] 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 [38], Trap.NET [39] 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 [40] and ATOM [41], 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 [42] 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 [43] 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 [44] 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).

VII. Conclusions and Future Work

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

For future work, we are interested in conducting more sophisticated case studies where we can explore: the runtime patching of managed and unmanaged applications and the management and coordination of various fine-grained adaptations. Finally, we are also interested in measuring the effects of (and system response to) injecting faults into managed and unmanaged applications, which have/have not been dynamically modified with appropriate self-healing (detection, diagnosis and remediation) mechanisms. This last set of experiments is part of an effort to further the development of a methodology for evaluating the efficacy of these added self-healing mechanisms and benchmarking the self-healing capabilities [45], [46] of the resulting system.

Acknowledgment

The Programming Systems Laboratory is funded in part by National Science Foundation grants CNS-0426623, CCR-0203876 and EIA-0202063. We would also like to thank Matthew Legendre and Drew Bernat of the Dyninst team for their assistance as we used and modified Dyninst. We thank Michael Locasto and Stelios Sidiroglou-Douskos for their assistance as we used STEM.

References

[1] J. O. Kephart and D. M. Chess, " The Vision of Autonomic Computing ," *Computer magazine*, January 2003.

[2] C. Shelton and P. Koopman, "Using Architectural Properties to Model and Measure System-wide Graceful Degradation," in *Workshop on Architecting Dependable Systems*, 2002.

[3] P. Koopman, "Elements of the Self-Healing Problem Space," in *ICSE Workshop on Architecting Dependable Systems*, 2003.

[4] C. Soules et. al, "System Support for Online Reconfiguration," in *USENIX Annual Technical Conference.*, 2003.

[5] Gail Kaiser et. al, "Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems," in *The Autonomic Computing Workshop 5th Workshop on Active Middleware Services (AMS)*, June 2003.

[6] S.-W. Cheng, A.-C. Huang, D. Garlan, B. R. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based Self-Adaptation with Reusable Infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, October 2004.

[7] B. Schmerl and D. Garlan, "Exploiting Architectural Design Knowledge to Support Self-Repairing Systems," in *14th International Conference of Software Engineering and Knowledge Engineering*, 2002.

[8] R. Griffith and G. Kaiser, "Manipulating Managed Execution Runtimes to Support Self-Healing Systems," in *Workshop on Design and Evolution of Autonomic Application Software*, May 2005.

[9] Rean Griffith and Gail Kaiser, "Adding Self-healing Capabilities to the Common Language Runtime," Columbia University, Tech. Rep. CUCS-005-05, 2005.

[10] R. Griffith, G. Valetto, and G. Kaiser, "Effecting Runtime Reconfiguration in Managed Execution Environments," in *Autonomic Computing: Concepts, Infrastructure, and Applications*,, M. Parishar and S. Hariri, Eds. CRC, 2006.

[11] G. Candea, J. Cutler, and A. Fox, "Improving Availability with Recursive Micro-Reboots: A Soft-State Case Study," in *Dependable systems and networks - performance and dependability symposium (DNS-PDS)*, 2002.

[12] A. V. Mirgorodskiy and B. P. Miller, "Autonomous Analysis of Interactive Systems with Self-Propelled Instrumentation," in *12th Multimedia Computing and Networking*, January 2005.

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

[14] Microsoft, "Common Language Runtime Profiling," 2002.

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

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

[17] S. Microsystems, "The Java Hotspot Virtual Machine v1.4.1," http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_4.html, 2002.

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

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

[20] 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.

[21] Daniel P. Bovet and Marco Cesati, *Understanding the Linux Kernel 2nd Edition*. O'Reilly, 2002.

[22] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching," *The International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, Winter 2000. [Online]. Available: citeseer.ist.psu.edu/buck00api.html

[23] L. Harris and B. Miller, "Practical Analysis of Stripped Binary Code," in *Workshop on Binary Instrumentation and Applications (WBIA)*, 2005.

[24] S. Sidiroglou, M. E. Locasto, S. W. Boyd, and A. D. Keromytis, "Building a Reactive Immune System for Software Services," in *USENIX Annual Technical Conference*, April 2005, pp. 149–161.

[25] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn, "PIN: A Binary Instrumentation Tool for Computer Architecture Research and Education," in *Workshop on Computer Architecture Education (WCAE)*, 2004.

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

[27] G. Cohen, J. Chase, and D. Kaminsky, "Automatic program transformation with JOIE," in *1998 USENIX Annual Technical Symposium*, 1998, pp. 167–178. [Online]. Available: citeseer.ist.psu.edu/cohen98automatic.html

[28] G. Cohen and J. Chase, "An Architecture for Safe Bytecode Insertion," *Software–Practice and Experience*, vol. 34, no. 7, pp. 1–12, 2001.

[29] N. Kumar, J. Misurda, B. Childers, and M. L. Soffa, "Instrumentation in Software Dynamic Translators for Self-Managed Systems," in *Workshop on Self-Healing Systems (WOSS)*, 2004.

[30] 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.

[31] Gregor Kiczales et. al, "Aspect-Oriented Programming," in *Proceedings European Conference on Object-Oriented Programming*. Springer-Verlag, 1997, vol. LNCS 1241. [Online]. Available: citeseer.ist.psu.edu/kiczales97aspectoriented.html

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

[33] Howard Kim, "AspectC#: An AOSD implementation for C#," Department of Computer Science Trinity College, Tech. Rep. TCD-CS-2002-55, 2002.

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

[35] A. Frei, P. Grawehr, and G. Alonso, "A Dynamic AOP-Engine for .NET," Tech Rep. 445, Dept. of Comp Sci. ETH Zurich, 2004.

[36] J. Lam, "CLAW: Cross-Language Load-Time Aspect Weaving on Microsoft's CLR," Demonstration at AOSD 2002. [Online]. Available: http://www.iunknown.com/000092.html

[37] C. Poellabauer, K. Schwan, S. Agarwala, A. Gavrilovska, G. Eisenhauer, S. Pande, C. Pu, and M. 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.

[38] 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. [Online]. Available: ftp://ftp.cse.msu.edu/pub/crg/PAPERS/trap-doa04.pdf

[39] 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 (ICAC)*, June 2005. [Online]. Available: ftp://ftp.cse.msu.edu/pub/crg/PAPERS/icac-2005.pdf

[40] J. R. Larus and E. Schnarr, "EEL: machine-independent executable editing," in *ACM SIGPLAN 1995 conference on Programming language design and implementation*, 1995, pp. 291–300.

[41] A. Srivastava and A. Eustace, "ATOM: a system for building customized program analysis tools," in *ACM SIGPLAN 1994 conference on Programming language design and implementation*, 1994, pp. 196–205.

[42] A. Tamches and B. P. Miller, "Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels," in *3rd Symposium on Operating Systems Design and Implementation (OSDI)*, 1999, pp. 117–130.

[43] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic Instrumentation of Production Systems," in *USENIX Annual Technical Conference*, 2004, pp. 15–28.

[44] M. Engel and B. Freisleben, "Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects," in *4th International Conference on Aspect-Oriented Software Development*, 2005, pp. 51–62.

[45] A. Brown, J. Hellerstein, M. Hogstrom, T. Lau, S. Lightstone, P. Shum, and M. P. Yost, "Benchmarking Autonomic Capabilities: Promises and Pitfalls," in *1st International Conference on Autonomic Computing*, 2004.

[46] A. Brown and C. Redlin, "Measuring the Effectiveness of Self-Healing Autonomic Systems," in *2nd International Conference on Autonomic Computing*, 2005.