

# Kernel Plugins: When A VM Is Too Much

Ivan Ganey, Greg Eisenhauer, Karsten Schwan

*College of Computing*

*Georgia Institute of Technology*

*Atlanta, GA 30332*

{ganev,eisen,schwan}@cc.gatech.edu

## Abstract

This paper presents *kernel plugins*, a framework for dynamic kernel specialization inspired by ideas borrowed from virtualization research. Plugins can be created and updated inexpensively on-the-fly and they can execute arbitrary user-supplied functions such that neither safety nor performance are compromised. Three key techniques are used to implement kernel plugins: (1) hardware fault isolation, (2) dynamic code generation, and (3) dynamic linking. Hardware fault isolation protects kernel-level services from plugin misbehavior, dynamic code generation enables rapid online creation of arbitrary plugins, and dynamic linking governs the kernel/plugin interface.

We discuss the design and implementation of the kernel plugin facility, as well as its advantages and shortcomings. Its use is demonstrated by a range of micro- and macro-benchmarks and a real-life application featuring plugins that dynamically transcode images served by a high-performance kernel web server. Benefits realized from plugins can be both qualitative (adapting services to clients' needs), and quantitative (improving performance through co-location of application plugin code with kernel services). Plugins are implemented in GNU/Linux on the Intel x86 platform. Reported performance results include plugin upcalls in 0.45-0.62  $\mu S$ , dynamic code generation in 4  $mS$ , and linking/unlinking in 3.1/1.6  $\mu S$  for an image grayscale plugin – a dynamically code generated 66-line function written in a subset of C. All results are measured on an 866 MHz Pentium III.

## 1 Introduction

Recent years have seen explosive growth in ubiquitously available computing power and network bandwidth, and we have witnessed the advent of novel products like smart mobile phones, wireless PDAs, and tablet PCs. These advances have spurred a wide range of applications, including Internet radio stations, peer-to-peer networks, and cellphone-based photography. Common to all such ubiquitous devices and applications is the need to guarantee high quality of service despite unpre-

dictable availability of platform resources and dynamically varying user needs. Two methods of addressing these issues have traditionally been resource reservation and system adaptation. Because of its ability to provide firm guarantees, the former has enjoyed strong popularity in real-time and mission-critical applications. Such firm guarantees, however, come at the cost of markedly lower resource utilization and that fact has made adaptation the method of choice for non-critical and consumer applications [2, 20].

The need for adaptation has spurred an extensive body of research into dynamically extensible systems at all levels of the computing and networking infrastructure, from library-based middleware techniques [6, 10], to extensible operating systems [4, 12, 15, 22, 27], to programmable network processors [13], and even active networks [24].

Techniques for implementing runtime extensions must balance *performance* with *safety* concerns. Efforts to achieve higher performance can degrade the safety and security of services, while efforts to bolster security may negatively impact performance by requiring time- and resource-consuming runtime checks. Consequently, a wide variety of solutions for extending kernel-level services have been proposed, ranging from approaches based on 'little languages' [16], to entirely new operating system kernels [4, 12]. One solution is to place extensions inside a virtual machine (VM) [25], completely isolating them from the rest of the system and thereby avoiding the need to trust them. The simplicity and safety of this approach is accompanied by some drawbacks, however, including: (1) the performance of virtual machines is inferior to that of native hardware [22], and (2) multiple OSes running in multiple VMs can complicate resource sharing and result in inefficient resource usage.

Our research seeks a middle ground between the complete isolation offered by virtual machines and the unsafe practice of system extension by adding new kernel modules. Our approach combines the use of virtualization techniques with dynamic binary code generation and dynamic linking, resulting in the *kernel plugin*

framework for runtime kernel extension.

A kernel plugin is made up of one or more application-supplied program functions that extend some kernel-level service. It is installed upon a client application's request and runs on its behalf. Plugins are designed to cooperate with, rather than replace, kernel-level services. Their interactions are controlled, so that a plugin only has privileges explicitly granted to it by the kernel. A well-defined plugin/kernel interface governs all such interactions. The efficient plugin mechanism permits rapid creation, update, and removal of plugins, thereby encouraging applications to frequently avail themselves of the mechanism's advantages.

Plugins are realized for the standard Linux kernel and the popular x86 hardware platform, offering a safe, efficient service extension mechanism to a broad set of developers. Our implementation achieves both high performance and safety by integrating three key techniques: (1) hardware fault isolation, (2) dynamic code generation, and (3) dynamic linking. Hardware fault isolation protects kernel services from misbehaving plugins. Dynamic code generation enables rapid runtime creation of custom plugins. Dynamic linking governs the kernel/plugin interface.

A key result of our research is the high performance of plugins, made possible by using isolation techniques borrowed from virtualization research [8, 9, 25], and by promoting frequent system adaptation through efficient plugin creation and deletion. For instance, plugin invocation costs are 0.45-0.62  $\mu S$  on an 866 MHz Pentium III depending upon the number of plugin parameters. In addition, plugin creation and setup costs are low, thereby encouraging their use in ways that are not easily implemented with coarser-grain mechanisms. Code generating a sample 66-line C code plugin on the same platform takes 4  $mS$ , while linking and unlinking take 3.1  $\mu S$  and 1.6  $\mu S$ , respectively.

In the remainder of this paper, we describe the design and implementation of kernel plugins on Intel x86 platforms running the GNU/Linux operating system. Kernel plugins are evaluated with micro- and macro-benchmarks, as well as with a realistic application – an accelerated web server augmented by a plugin specializing the data it delivers to clients. This example, evaluated in detail, is on-the-fly transcoding of image data, streamed from the server's disk to its communication link.

## 2 Related Work

While safe runtime kernel extension has previously been addressed in the literature, unfortunately such functionality is not generally available in commonly used operating systems. Several classes of solution techniques have

been proposed:

### Programming Language Techniques

In the SPIN operating system, the safety of kernel extensions is based on the properties of the Modula-3 type-safe programming language and a trusted compiler [4]. Furthermore, because SPIN's kernel extensions use relatively heavyweight external compile/link/execute facilities, creation costs must be amortized over extended and frequent use. As a result, SPIN extensions are best suited to long-lived functionality.

The Open Kernel Environment (OKE) [5] employs a variation of the same idea, substituting the type-safe Modula-3 with Cyclone, an 'elastic' customizable version of C, and trust management integrated with the compiler.

In contrast to these schemes, kernel plugins are designed to be lightweight, agile, and easy to adapt on-the-fly. Plugin creation, invocation, and removal overheads are very low and do not involve execution of external compilers or linkers. Furthermore, our facility implements both preemption and isolation and thus does not need to trust any binaries outside the kernel.

### Proof-Carrying Code

Proof-carrying code [18] is a mechanism for safety verification of code that requires that a 'safety proof' is attached to each piece of code, certifying its adherence to a pre-defined 'safety policy'. The proof is such that quick validation is possible without cryptography or external references. Despite those desirable properties there are three drawbacks to proof-carrying code.

The first and foremost one is that generating a comprehensive safety policy for non-trivial code is very hard. The difficulty results from the fact that the policy needs to cover all obvious and implied rules and invariants of the execution environment. Furthermore, there is no way to guarantee the completeness of the policy itself. Second, the method has scaling issues because the safety proof's size grows large rather quickly. As an example, a trivial function summing two numbers under a basic safety policy is quoted to have 60 bytes of code and 430 bytes of safety proof [18]. Finally, no automatic proof generators exist.

Kernel plugins provide an alternative – an engineering solution that achieves native code performance and safety without the burden of a proof or type-safe language restriction.

### Software Fault Isolation

SFI approaches [26] rely on rewriting the machine code of extensions so that memory accesses and jump targets are checked and instrumented, thereby restricting them to the scope of the extension's protection domain. Only after such *sandboxing* is an extension allowed to execute. Program interpretation is a related approach in

which extensions are executed by a trusted interpreter that enforces safety.

Typical examples of such extensible kernels are VINO [21], which relies on SFI, and packet filters like the Berkeley Packet Filter [16], which implements an interpreted ‘little language’ for custom, in-kernel, packet filtering rules. The primary problem with these approaches is that the price of safety is non-trivial performance degradation, which makes them less appealing for high-performance applications. The performance of type-safe language extensions is quoted to be 10% to 150% worse than regular C code, and SFI can be as much as 220% slower [8]. In comparison, kernel plugins do not incur per-instruction execution overheads. Plugin code generation is a one-time cost, significantly smaller than compilation alternatives and amortized over the lifetime of the plugin.

### Hardware Fault Isolation

HFI relies on hardware-provided memory management features to enforce the isolation between the kernel and extensions. This is the same method that traditional operating systems use to isolate their kernels from user-space applications. It also forms the basis for most ‘virtualization’ and ‘isolation’ systems, which can be viewed as very coarse-grain extension mechanisms. Notable examples include the VMware [25] and Virtual PC [9] virtual machines, as well as the library operating systems supported by Exokernel [12], the Denali isolation kernel [28], and Xen [3] – a new VM monitor that defines an abstract VM to which kernels are then ported, reportedly achieving close to native performance.

Palladium [8] also uses hardware features to achieve extension isolation, but on a somewhat finer grain and without striving to provide a complete virtualization environment. It limits its scope only to untrusted kernel modules, and uses segmentation and privilege-checking hardware to ensure that they cannot interfere with the kernel proper. While Palladium’s strategy results in better performance compared to virtual machines, it still restricts system adaptation to relatively coarse-grain kernel modules, and limits the dynamic use of such extensions because it requires off-line module compilation.

### Kernel Plugins

Like some of the above approaches, we choose to employ a hardware-based scheme, exploiting the x86 architecture’s segmentation hardware and unused privilege rings to provide isolation. Specifically, the x86 hardware provides 4 ‘privilege ring levels’. Typical operating systems use ring-0 (most privileged) and ring-3 (least privileged) for kernel and user modes, respectively. Kernel plugins utilize one of the unused privilege rings. Thus, memory protection and control-flow restrictions are enforced entirely in hardware, causing no discernible performance degradation. This is a popular isolation ap-

proach employed by all x86 virtual machine projects of which we are aware, as well as the implementation of intra-address space protection in Palladium.

Unlike VMware and VirtualPC style VMs, however, we do not strive to provide the illusion of a dedicated machine. Instead, we define a streamlined, lightweight execution environment in a manner which is more meaningful and fitting to a plugin’s purpose of customizing existing services rather than deploying new ones. Unlike Exokernel, Denali, and Xen, we do not modify host architectural assumptions and require no porting or reimplementation of host-kernel subsystems that do not need to be extensible. Finally, unlike Palladium we strive to achieve finer granularity and enable runtime online adaptation while keeping setup overheads low. Experimental results presented in this paper demonstrate that kernel plugins experience no additional runtime costs per instruction. We also show that the overhead of protected control transfers to and from plugins are both small and predictable.

## 3 Motivation

Previous work [4, 8, 12, 20, 21, 22] has already demonstrated that application-specific extension of operating system kernels can be a key contributor to attaining high end-to-end performance. A wide range of specializations exist that can easily be realized using plugins – the spectrum of opportunities spans virtually all subsystems of a modern OS kernel. Plugins could augment a file system with custom caching or prefetching algorithms, or modify a TCP stack’s back-off strategy to reflect loss properties of a particular client’s link. They could enhance core system services like scheduling, by providing scheduling hints in the guise of payoff functions, or extend memory management by specializing the behavior of page replacement algorithms. Finally, kernel plugins can even be useful in high-performance kernel servers like the Linux accelerated web servers TUX and kHTTPd. Some examples of the rich set of application-specific plugins that can be deployed are (1) dynamic compression and decompression of data to effect trade-offs in server vs. client CPU needs and/or required transmission bandwidth, (2) runtime downsampling techniques reflecting a clients’ preferences for fidelity vs. timeliness, (3) region-of-interest type transformations, removing unnecessary data from a communication stream, etc.

The following example kernel plugin usage scenarios have guided our research:

### Smart Filtering

One usage of plugins is to permit end users to directly affect data production, transmission, and reception at the kernel level. For instance, if certain data is not of current

interest to the recipient, it can be eliminated early in the receiving OS kernel, rather than being transferred to user level only to be discarded. Similarly, if only subsets of data are of interest to specific recipients, then source-based and client-specific data filtering may be implemented with plugins [10]. Alternatively, plugins can be used for ‘valuation’ of information being captured, processed, transmitted, or received, by applying payoff or utility functions to it. Research has shown that such utility functions can be a very useful adaptation tool.

### Intelligent Introspection

Another possible domain of use for kernel plugins is system monitoring and instrumentation [23]. The idea is to deploy code that is tailor-made for its specific purpose and to allow it to evolve dynamically with the needs of the client, instead of having to measure and export a large and generic set of metrics. For instance, an NFS client experiencing degradation of service can dynamically instrument its server’s disk and network subsystems to discover where the bottleneck is and adapt or possibly work around it.

### Runtime Adaptation

A final example of a kernel plugin usage scenario is to enable low-overhead dynamic self-adaptation of a system’s behavior, perhaps as a response to changes in monitored conditions. For instance, the NFS client from our previous example determines that there is a disk head scheduling bottleneck and adapts by pushing into the NFS server an aggressive prefetch algorithm customized to its current access patterns.

## 4 Design

### 4.1 Approach

The success of any OS facility is strongly linked to its performance characteristics and ease of use. Thus, a principal goal of our framework is to provide an effective, efficient, and easy to use extension mechanism. The following properties guided our design:

- *Generality*: The API should be generic and avoid targeting a specific kernel service.
- *Functionality*: Unnecessary restrictions should be avoided on what constitutes valid plugin code. Plugin creation, use, and deletion should be possible in runtime, using both statically pre-compiled and dynamically generated code.
- *Safety*: The core kernel should be protected from direct or unintended manipulation by plugin code.
- *Efficiency*: Implementation overheads should be less than or comparable to alternatives.

Kernel plugins attain these properties by combining three key technologies: (1) hardware fault isolation, (2) dynamic code generation, and (3) lightweight dynamic

linking.

Hardware fault isolation protects the core kernel from the untrusted plugins and helps to avoid costly per-instruction runtime overheads. It provides an engineering solution to the isolation problem without the complexity and overheads inherent in programming-language techniques, proof-carrying code, or software-fault isolation.

While a library of pre-compiled adaptation strategies that clients can choose from can go a long way, sometimes applications need tailor-made solutions. Adapting file system prefetching to irregular access patterns, or filtering out or digesting parts of complex objects to transfer are but a few such examples.

Dynamic code generation, thus, serves a two-fold purpose. First, it provides a common language for arbitrary and cross-platform runtime adaptation in a heterogeneous environment, and second, it promotes performance by translating extensions into native machine code able to run at full speed on bare hardware.

It is important to realize that we do not mean to discount the usefulness of libraries of pre-compiled plugins. Such libraries are certainly instrumental for complex, static codes like fast Fourier transforms, JPEG encoding/decoding, etc. Rather, we propose to augment such libraries with a complementary mechanism that is able to adapt to variable runtime conditions.

Dynamic linking controls the kernel/plugin interface. It enhances the plugins’ expressive power by permitting collaborative compositions of plugin functions to perform complex tasks.

### 4.2 Plugin Runtime

The base plugin mechanism is a simple abstraction of an ‘execution environment’. This environment, termed the *plugin runtime*, registers, handles, and manipulates the kernel plugins of a single extensible entity. It can be thought of as defining a streamlined abstraction of a tiny virtual machine. As our aim is not to emulate a particular systems platform but to create a clean and efficient extension environment, we are able to design for simplicity and reap the benefits of efficiency.

Each runtime has a restricted, but well-defined API providing the means to add new plugin functions, as well as to execute and delete existing ones. Multiple runtimes may exist simultaneously at any given time, each managing the extension functionality of a single client or client instance. Each instance of an extensible entity creates its own runtime and dynamically populates it with client-supplied plugins. The resulting multiplicity of runtimes serves a threefold purpose. It allows extensibility on a per-instance basis, prevents plugin namespace pollution, and isolates related or cooperating plugin functions

within a single runtime. Actual coordination and cooperation of plugin functions of a single client is left to the client itself, with the runtime only providing the glue primitives to enable it.

As an illustration, consider two separate kernel services: a kernel http daemon (as in our sample application), and a kernel NFS server. Each server instance creates a runtime for its plugins and populates it upon its clients' request. The http daemon's threads might install any number of image manipulation plugins, whereas the NFS daemon's threads might install various data compression algorithms. The separate runtimes ensure that the namespaces of unrelated plugins belonging to different clients are disjoint and that unrelated data and symbols cannot be named or invoked.

### Built-in Plugins

Sometimes application-specific plugin code will need to call on certain kernel functions to achieve its goals, e.g. enqueueing a packet or reading/writing a block from/to disk. To accommodate such *callbacks* seamlessly within our framework they are represented as a kind of plugin. These 'built-in' plugins are explicitly added to the runtime by the kernel service it extends. Even though they act as kernel callbacks, within the restricted plugin environment they are indistinguishable from a regular 'dynamic' or user-supplied plugin. That is to say that they are invoked and used in exactly the same fashion. Immediately after its creation, a runtime's namespace contains only a default set of available built-ins listed in Figure 1. They perform basic namespace maintenance expected from the dynamic linker: `create()`, `lookup()`, and `delete()`.

The availability of callbacks poses the question of how to handle kernel resources acquired through them in the event that a plugin needs to be terminated. Because of the rich variety of kernel resources, we considered building a system that tracks all of them to be impractical. We believe that the runtime's owner service is able to handle the cleanup of the limited number of kernel resources it makes available to its plugins in a much more efficient way, if at all needed, e.g. through callback wrappers tracing resource usage, etc.

## 4.3 Memory Model

The memory model of the plugins' execution environment is influenced by the choice of hardware isolation mechanism. The scheme exploits features of the Intel x86 architecture's segmentation and protection hardware by placing all plugins into an unused privilege ring. While such hardware dependence may seem restrictive, the 'privilege rings' concept on which it relies is available on all modern CPU architectures. The most popular ones, Intel's IA32 and IA64, provide 4 privilege rings,

```
long create(runtime_t * rt, char * code, char * name);
long delete(runtime_t * rt, char * name);
long lookup(runtime_t * rt, char * name);
```

Figure 1: Built-in plugins' prototypes

```
long call_plugin(int id, runtime_t * rt, ...);
```

Figure 2: Gate function for invoking plugins

whereas others like the SPARC and the PowerPC provide only 2 privilege rings for supervisor and user mode, respectively. Kernel plugins can still be implemented on the latter in at least two different ways. One is to place plugins in pinned, unpagged memory in the user-level privilege ring. Isolation is enforced by the hardware and many overheads associated with using a process are avoided. Another option is to place plugins within the kernel privilege ring but to restrict them to dynamically generated code, thereby guaranteeing that they cannot interfere with paging and segmentation hardware. The former approach allows the use of arbitrary code in plugins at the expense of requiring somewhat complicated transfer of control between privilege rings. The latter approach invokes plugins just like ordinary kernel functions, but restricts them to dynamically generated code.

On x86 hardware, the OS kernel runs in ring-0 (highest-privilege). We allocate memory to hold all plugins' code, data, and stacks in ring-1, thereby guaranteeing the kernel memory's safety. In contrast, callback built-ins are invoked through a hardware trap, not unlike system calls, and run in ring-0, that is, they run in the OS kernel. Control and data flows between privilege rings are governed by the host kernel through hardware traps.

Plugins have full access to their parameters and local variables allocated on the plugin stack. They also have full access to a pool of ring-1 memory, effectively acting as a heap. The contents of the heap persist between plugin invocations, so it is also used for static variables. The heap is allocated on a per-runtime basis, which means that all plugins within a runtime share it and can use it for global variables, communication, and cooperation. Additionally, it is possible to provide select plugins with read-only access to parts of the kernel proper's memory. While such a feature could simplify the implementation of system monitoring plugins or the sharing of data between the kernel and plugins, it can also have security implications so it should be employed judiciously.

## 4.4 E-code Language Specification

In our design, plugins can be specified either as pre-compiled machine code, or in E-code – a language akin to ‘C’ [19] and developed as part of the ECho high-performance event-delivery middleware [10]. E-code is a fairly complete subset of the C language that compiles to native machine code at runtime using a dynamic code generator that processes one function at a time. A more detailed list of E-code capabilities follows:

- *Datatypes:* E-code supports the following basic types: `char`, `int`, `float`, `double`, and `boolean`. It also supports structures, pointers (including pointers to structures), and pointer arithmetic.
- *Variables:* Global variables are allocated on the heap, which is a per-runtime pool of ring-1 memory persistent across plugin invocations. Local variables are allocated on the plugin stack.
- *Function calls:* Plugins are allowed to perform function calls only to other functions or callbacks registered within their runtime. Appropriate trap or trampoline code for the invocation is generated automatically and transparently.
- *Function prototypes:* Plugin functions must conform to a prototype convention – their first argument must be a `‘runtime_t *’` to provide linkage back to their runtime. Furthermore, their result type is restricted to `long`, however, that is not a severe restriction since most basic datatypes are easily cast to a `long` value, with the notable exception of the class of floating point numbers, which must be passed back by reference.
- *Language:* E-code supports the C operators, `for` loops, `if`, and `return` statements.

Currently, E-code does *not* support `while` loops, `switch` statements, unions, and function pointers, though they do not pose conceptual difficulties and can be implemented if needed in the future.

## 4.5 Interface

The kernel/plugin interface consists of the runtime namespace manipulation routines, any additional kernel callbacks that an extensible subsystem instance exports to its plugins, the plugin invocation mechanism, and the pool of plugin static memory.

The runtime namespace manipulation routines displayed in Figure 1 are implemented as kernel proper functions. Thus, they are directly available to the kernel proper and are isolated from plugins, yet available to them in the form of ‘built-in’ plugins. Besides that mandatory minimum, each extensible service can augment the interface by exporting more kernel callbacks of its choosing, e.g. `sendmsg()` and `recvmsg()`, in the form of

additional ‘built-in’ plugins. We continue with a more detailed description of the namespace manipulation interface.

### Creation

Each plugin function is specified by a tuple that describes it completely. The tuple consists of the following elements:

- *Runtime pointer:* It refers to the runtime this function is to be created in. All functions’ prototypes have a `‘runtime_t *’` first argument serving as a link to their runtime environment and allowing them to interface with other functions. It provides closure (in the mathematical sense) of the namespace with respect to the operations of its functions.
- *Code:* This is either an ASCIIZ string specifying a single E-code function or a pre-compiled relocatable machine code dump. In the former case the runtime translates the E-code function into efficient, native machine code at creation time. The translation is a one-time cost and is amortized over all subsequent executions of that function. Translation costs are relatively small, thanks to the efficiency of E-code’s dynamic code generator [10]. For example, the image grayscaling plugin used in our experimental evaluation consists of a 66-line E-code function which translates in only 4 *mS*, compared to the 700 *mS* it takes to spawn an external compiler (with compiler binary already present in the OS buffer cache).
- *Name:* A string constant providing the name this function is to assume in the runtime’s symbol table. After its creation, a function can be looked up and called upon using that name.

### Deletion

Deleting a function is a straightforward operation that deallocates the code and static data resources associated with it and then unlinks it from the symbol table. The `delete()` built-in plugin’s prototype is self-explanatory and also appears in Figure 1.

### Invocation

A function is available for execution immediately after its creation. The actual invocation, however, is not as trivial as a simple function call because of the privilege ring-based isolation scheme.

**From Kernel Space:** Normally, hardware does not allow higher-privileged code to call untrusted, lower-privileged code. To circumvent the problem our framework provides a ‘gate’ function `call_plugin()` that encapsulates the implementation complexity and hides hardware details. This makes invoking any plugin as simple as calling the gate function whose prototype is shown in Figure 2.

The gate function looks up the target plugin’s entry in its

runtime’s symbol table and copies the declared number of parameters from the kernel’s to the plugin’s stack. It then invokes the plugin by branching to its address and sidestepping the hardware restriction. The mechanics of the latter are described in more detail in the implementation section.

**From Plugin Space:** To encourage function composition we provide a similar gate function in the isolated address space, permitting plugin functions to invoke each other. It is syntactically and semantically identical to its counterpart employed from the host kernel despite significant implementation differences.

Invoking a plugin function from another one has overhead akin to that of a simple function call. The reason for this being that control flows within the isolated address space and no protection boundary needs to be crossed. The benefit is that this enables plugin functions to cooperate easily and cheaply, thereby increasing the utility of the model for complex extensions.

Invoking a kernel callback (built-in plugin) from a dynamic, user-defined one, however, does require crossing the protection boundary from ring-1 back into ring-0. This is achieved by means of a hardware trap, the details of which are hidden in the gate function’s implementation and explained further in the next section.

Finally, irrespective of whether the call originates in ring-0 or ring-1, invoking a plugin requires naming it unambiguously, i.e., by its name and runtime context. Unfortunately, matching name strings in the symbol table repeatedly is needlessly expensive. To avoid that overhead, we map the string name to an integer id unique within each runtime, thereby speeding-up lookup and simultaneously making id caching much easier. All built-ins are also assigned fixed well-known integer ids. The mapping between dynamic plugins’ names and integer ids is performed by the `lookup()` plugin.

## 5 Implementation

A prototype of the kernel plugin facility has been implemented in recent stable-tree Linux kernels (versions 2.4.18 and 2.4.19). The prototype implements hardware isolation, dynamic code generation, and dynamic linking fully, though details of the design are still evolving. This section describes some relevant implementation details and their implications.

### 5.1 Hardware Fault Isolation

Our plugin isolation scheme is a clean re-implementation of a popular concept employed in a number of systems, both virtual machines and others, e.g. VMware [25], Palladium [8], etc. It exploits features of the segmentation and privilege checking hardware of the Intel x86 architecture to achieve

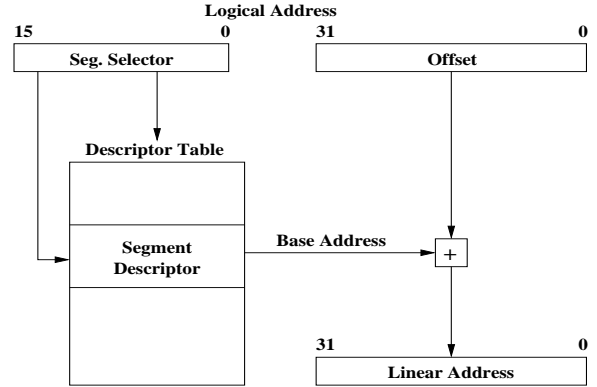


Figure 3: Intel x86: logical to linear address translation

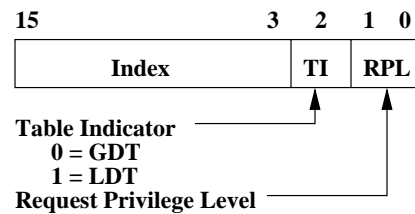


Figure 4: Intel x86: segment selector layout

address space isolation within the Linux kernel. We briefly describe the method next; for a full discussion, the reader is referred to the original papers and Intel documentation.

The fundamental idea is to allow application-specific code to run in the core kernel by placing it in a separate protection domain and relying on hardware to enforce it. The new domain is an address space – a proper subset of the Linux kernel’s virtual address space. While the kernel itself can access the plugins’ address space freely, plugins cannot, in general, access the larger kernel memory.

### Segmentation

In practice, domains are implemented as protected memory segments directly supported by the hardware MMU. Segments are ranges of consecutive addresses described by base address and length. The operating system maintains linear tables of ‘segment descriptors’ for all segments. In addition to base and length, each segment descriptor stores privilege, access, and type information for its segment. Two tables of descriptors are active at any given point in time: the Global Descriptor Table (GDT) and a Local Descriptor Table (LDT). While the former is static and immutable, the latter is a per-process structure. Figure 3 depicts how segmentation addressing works on the x86. Logical addresses are composed of a 16-bit segment selector and a 32-bit offset. Segment selector values are used to index into segment descriptor tables. The layout of a segment selector is shown in Figure 4.

It contains a linear table index, a one-bit Table Indicator (TI), and a two-bit Requested Privilege Level (RPL). The TI determines which of the two currently active tables the selector is referring to, whereas the RPL is used in privilege checks. The chosen descriptor entry provides a base address, a limit to check the offset against, read/write/execute permissions, and a descriptor privilege level (DPL).

The CPU maintains a Current Privilege Level (CPL) for the currently executing instruction. The DPL is compared to the CPL and RPL for each memory access. In general, CPL and RPL need to be numerically less than or equal to the target segment's DPL (i.e. at the same or higher privilege) in order for access to be granted. For a complete explanation of access permission checks and motivation for the existence of the RPL the reader is referred to Intel documentation [1, pp. 105–140]. All access violations, such as pointer dereferences to memory outside of the ring-1 segment (including NULL pointer dereference), attempts to execute an illegal or protected instruction, etc., result in an exception being thrown and a trap to the kernel proper to handle it. Additionally, the owner service allows each plugin a quantum of time in which to complete. Overrunning that quantum is detected by a periodic timer interrupt and also results in forceful preemption and a trap to the kernel.

Our current method for dealing with offending plugins is immediate termination. An interesting future direction we are considering is to implement a recovery mechanism, allowing extensible services the choice to terminate or to continue a plugin based on custom per-service policies and the type of the failure or misbehavior.

### Initialization

At boot time, the plugin facility allocates a region of memory to be used exclusively for plugin code and data. A simple first-fit private memory allocator is initialized with the parameters of the pool and is used to allocate memory for structures to be placed in the isolated area.

Two new segment descriptors are computed and installed in the GDT, each covering the whole isolated memory pool. Both descriptors are assigned the same ring-1 privileges but different types. The type of the first one is set to 'code' and it is used to address executable plugin code. The type of the second one is set to 'data', and it is used for data manipulations involving plugin stacks and heaps. This simple overlay scheme was chosen to ease the initial implementation effort by avoiding the need to parameterize the memory allocator for disjoint code and data memory pools. It could be replaced with a split code and data design in the future, to prevent the possibility for self-modifying plugin code and to limit the amount of damage a misbehaving plugin can wreak upon other plugins.

A third segment descriptor can be defined overlaying part or all of the kernel's ring-0 memory but accessible in read-only mode from ring-1. Although such a segment has the potential of simplifying or optimizing kernel/plugin data interactions it has to be used with great care because of possible security implications. It can be thought of as an optional feature for cases when performance benefits outweigh potential security concerns.

Runtimes are created at the request of kernel services. Each runtime's control structure contains pointers to its stack and heap as well as a symbol table of registered plugins. The control structure is allocated in ring-0 kernel memory to protect it from being tampered with by plugins. The built-in plugins are implemented as trusted kernel subroutines enabling them to modify the control structures during operations like creation or deletion of dynamic plugins. In contrast, the heap, stack, and code of plugins are all allocated within ring-1 memory.

### Control Transfers

Passing control from a plugin in ring-1 to the kernel in ring-0 is straightforward, by use of a trap gate similar to the one implementing system calls from user-space (ring-3). The hardware handles the trap and passes control to the kernel in a protected fashion. Any state needed for returning to ring-1 is saved on the kernel stack. If the control transfer is to be one way (e.g. return from a plugin) rather than two way (e.g. kernel callback), then that state is simply cleaned up by the kernel trap handler.

Passing control from the kernel to a plugin, unfortunately, is more difficult. There is an inherent asymmetry in control transfers between privilege levels because the hardware is designed to prohibit high-privileged code from invoking lower-privileged code. To sidestep the problem, a stack frame is carefully forged (on the ring-0 stack) that emulates the state the stack would have had if it had been called from ring-1 and then executes a `ret` instruction to the forged return address. This causes the CPU to switch into ring-1 and start execution of the targeted plugin function.

### Interrupts

x86 interrupt handlers run in ring-0. In case an interrupt occurs while the CPU is already executing in kernel mode, the interrupt simply grows the current stack. If, however, the CPU is executing in a privilege level different from ring-0, then it switches to ring-0 immediately, performing the necessary stack swap to the *bottom* of the kernel stack. This behavior is predicated on the premise that an interrupt occurring while in user-space has no kernel state to preserve, so the new frame can start from the base of the kernel stack.

It is not hard to see how this otherwise normal behavior can cause trouble when interacting with ring-1 plugins, however. The aforementioned premise is negated



because plugins *are*, in effect, a part of the kernel, yet they execute outside of ring-0. If an interrupt fires while a plugin is running, the CPU switches immediately to a ring-0 handler and uses the kernel stack. Unfortunately, starting from the *base* of the stack it overwrites the state already there, accumulated prior to invoking the plugin. This effect is due to the unconventional use of privilege rings to implement what amounts to protected upcalls of which the hardware is unaware.

There are a few possible solutions to this problem: (1) disable interrupts while plugins are running, (2) save and restore the kernel stack before and after plugin invocations, and (3) trick the hardware to *grow* the stack upon an interrupt in ring-1.

While the first solution does not add any overhead to plugin execution, it has the undesirable effect of blocking interrupts for potentially non-trivial lengths of time. In the kernel, blocking of *all* interrupts is allowed only for the shortest times, since it could lead to loss of important device interrupts and disrupt the operation of peripherals. Moreover, such a solution would also prevent the implementation of plugin preemption, which relies on a periodic hardware timer interrupt. Clearly, this approach is unsuitable.

The second solution, saving the kernel stack's state before plugin invocation and restoring it immediately after that, is workable. It was our first implementation, but it increased plugin invocation overheads and introduced significant irregularities in their cost, due to the unpredictable amount of state (up to a page frame in the worst case) that needs to be saved and restored each time.

These disadvantages led us to come up with our final solution. It is based on an architectural programming trick that fools the interrupt handling hardware into *growing* the kernel stack rather than overwriting its bottom, despite the fact that the interrupt occurs outside of ring-0. The trick involves careful manipulation of the stack base pointer in the task state segment structure (TSS) of the CPU [1]. This allows us to continue servicing interrupts while plugins are running, yet, at the same time avoid the unpredictability of kernel stack saving and restoring. As an added bonus, the overhead of this method is extremely small, its implementation consisting of only a few assembly instructions.

### A Remaining Issue

A discussion of kernel plugins would not be complete without mention of any remaining issues with their current implementation. One such issue is the lack of protection across multiple plugins. Thanks to the compartmentalization of each client's plugins into a separate runtime, plugins have no means of naming symbols in other runtimes. This, however, does not provide firm isolation guarantees, even though it raises the bar for how

difficult it would be for a plugin to interfere with plugins in other runtimes.

To address this issue, we are considering developing our scheme further to include two GDT descriptors *per runtime*, to describe each runtime's code and data separately from other runtimes. In this way, we can exploit the segmentation hardware further and achieve isolation not only between the kernel proper and plugins but also among runtimes. Such an enhancement would not add any runtime overhead and is under active development.

## 5.2 Dynamic Linking

As part of the provided trusted runtime environment, a dynamic linker operates on a runtime's symbol table and implements symbol creation, lookup, execution, and deletion. The symbol table is an array of symbol structures, and looking a symbol up in it has linear complexity. This choice was made to simplify the initial implementation and will not result in problems unless a very large number of plugins are registered within a single runtime. Re-coding the symbol table as a hash table may be used to address this issue should it become necessary.

## 5.3 Machine Code Generation

The E-code dynamic code generator operates by parsing the source language and emitting the appropriate instructions into a memory buffer from which they can be executed directly. Currently, E-code supports dynamic code generation for Intel x86, MIPS, StrongARM, and Sun SPARC (32- and 64-bit) processors. Support for Intel's 64-bit EPIC architecture is under development.

The code generator is subroutine-based and does not require invocation of external binaries. Code is emitted during parsing in the form of virtual instructions for an idealized RISC architecture. Simple, low-hanging fruit optimizations are applied (constant propagation, register renaming, and limited common subexpression elimination), and then the virtual instructions are mapped to their physical counterparts for the target architecture.

E-code's early versions were based upon Icode, an internal interface developed at MIT as part of the 'C project [19]. Icode is itself based on Vcode [11], also developed at MIT by Dawson Engler. E-code's recent versions, however, are based on DRISC, a low-level DCG package developed at Georgia Tech. The performance of the two versions are similar. More information about characteristics of the E-code language, such as examples of the generated code, further details on its performance, and generation times can be found elsewhere [10].

## 6 Experimental Evaluation

This section demonstrates the base performance of plugins using two micro-benchmarks, two macro-

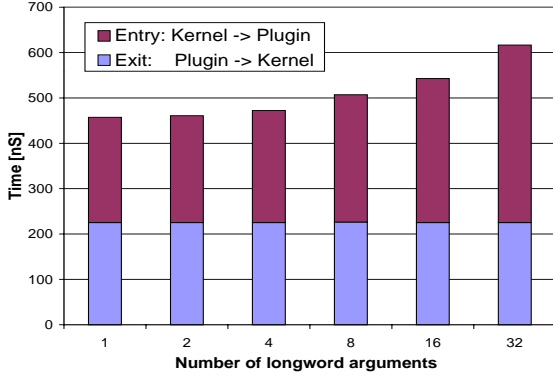


Figure 5: Control transfer cost vs. number of arguments

benchmarks, and finally, an evaluation of the utility and practicality of plugins with a realistic image-transcoding plugin. The latter color-downsamples images using an application-specific integer-only method as per some client’s needs.

Experimental results reported in this section are obtained on an 866 MHz Pentium III processor, with 16 KB L1 I&D caches, 256 KB unified L2 cache, 512 MB of PC133 RAM, and a 20 GB Western Digital WD205AA hard drive used in UDMA66 mode. The operating system is Fedora Core 1 running Linux kernel version 2.4.19 augmented by our kernel plugin facility.

Timing is performed using the Pentium processor’s internal time-stamp counter (TSC), except for the *httperf* [17] macro-benchmark which uses much coarser granularity system timers. The TSC is a 64-bit register zeroed at power-up and incremented by exactly 1 with each clock tick of the CPU core. Its 2 ns error is insignificant in comparison to statistical variations in experimental data.

## 6.1 Micro-benchmarks

### Plugin Execution

Two metrics important to any server application are latency and throughput. We measure the impact that placing code in a kernel plugin has on these metrics. We define *latency overhead* as the amount of time passing between the first instruction of a kernel plugin invocation and the execution of the plugin’s first instruction. Latency overhead thus defines the latency cost of utilizing the kernel plugin facility. Similarly, we define *throughput overhead* to be the execution time of a null plugin. This represents the pure cost of the plugin abstraction. Since plugins are executed directly on the underlying hardware, these metrics are the only runtime costs incurred by kernel plugins.

Micro-benchmark data displayed in Figure 5 depicts the execution time of a null kernel plugin (in nanoseconds),

versus the number of long word arguments passed to it. Execution time is comprised of two parts: entry into the plugin and exit from the plugin. The first part characterizes the latency overhead experienced due to plugin use, whereas the second part represents the remaining cleanup overhead at exit time. It is easily observed from the graph that the entry latency is weakly linearly dependent on the number of plugin parameters, whereas the exit overhead is constant. It is important to realize that the measurements in Figure 5 are for plugin invocation from the kernel. Function invocation from within ring-1 is almost identical to a user-space function call, meaning that it is essentially ‘free’.

Results are obtained by timing 2001 runs, dropping the first one to avoid cold CPU cache effects and averaging the rest. Furthermore, interrupts are disabled during each individual benchmark run to shield measurements from the high timing variability that interrupts induce under Linux v2.4 [14]. The observed standard deviation for this plot is less than 1% from the mean, implying very high confidence in the data and predictability of the mechanism’s performance.

The main result is that for a reasonable number of parameters, the baseline cost of kernel plugins is between 0.45  $\mu S$  and 0.62  $\mu S$  for this hardware. Thus, our implementation’s performance is on par or better than similar schemes [8] (after adjusting for our faster hardware) despite major differences in kernel architecture (2.0.34 vs. 2.4.19) and implementation methodology.

### Plugin Creation/Deletion

Since kernel plugins are intended to be easily and frequently updated, it is important to characterize their creation and deletion costs.

Our dynamic code generator uses subroutine-based techniques that do not require invocation of an external compiler. It is fast and of time complexity roughly linearly proportional to the source code size. For the sample image-transcoding plugin used in our experiments the costs for code generation, linking, and unlinking of the plugin are 4 ms, 3.1  $\mu S$ , and 1.6  $\mu S$  respectively.

## 6.2 Macro-benchmarks

To better understand application-level effects of the baseline costs associated with different isolation techniques and with kernel plugins in particular, we next turn our attention to macro-benchmarks. We compare and contrast the overheads and service effects of two proposed isolation techniques for user-specific kernel extensions: placing extensible services in virtual machines, and implementing extensions as kernel plugins.

Modern services need to provide customizability while maintaining high levels of performance. Generally these two imperatives are in conflict. For our particular exper-

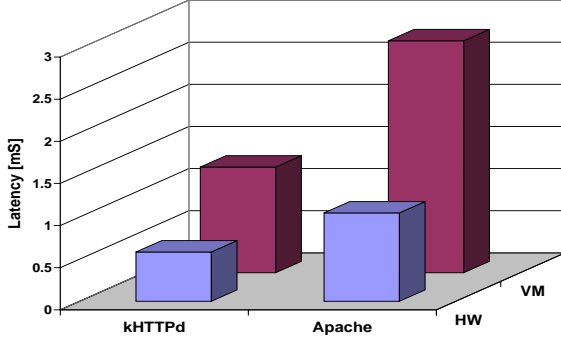


Figure 6: Latency effects of commercial VM technology (VMware 3.2.0)

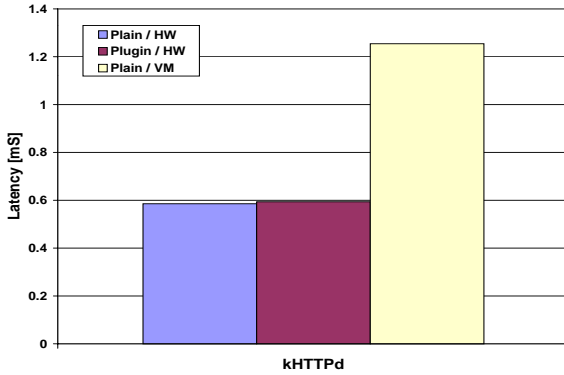


Figure 7: Comparative latency effects of kernel plugins vs. commercial VM technology

iments, we chose to look at a web server, as it is a relatively simple, typical, and popular service with many available implementations. A well-known approach to building high-performance web servers is to run the service daemon within the OS kernel. While this eliminates many inefficiencies inherent to user-space and increases the performance of the server substantially, it has the unfortunate effect of discouraging extensibility due to safety concerns when running within the kernel. We propose to use kernel plugins to rectify this problem.

To quantify our assertions and to provide a better gauge for the expected performance of typical kernel- vs. user-space web servers, as well as different isolation techniques, we measured the server reply latency and reply throughput of popular web server implementations and report our findings in figures 6, 7, and 8.

The web servers ran on the machine described previously, while the test load was provided by a more powerful Dell Workstation 340 (2.2 GHz Pentium 4, 512 KB L2 cache, 512 MB of RDRAM-400) over an otherwise quiescent 100 Mbps Ethernet network. Results for these three figures were measured in user-space at the client

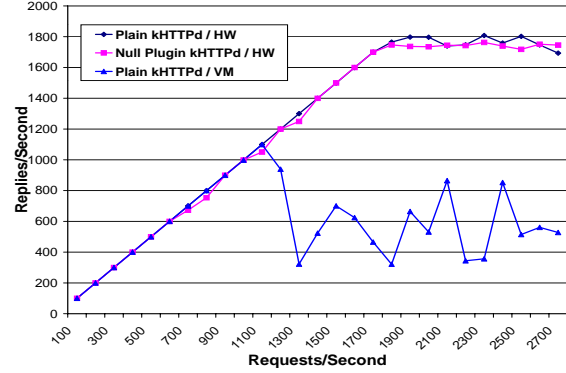


Figure 8: Isolation technology impact on throughput

workstation and include jitter induced by interrupt processing on both ends. Therefore, these are a good indicator of subjective performance for a client application in this environment.

Figure 6 provides a server reply latency comparison between kernel- vs. user-space based servers, as well as a measure of the cost of full virtualization (as measured on an industry standard product VMware 3.2.0). The figure shows that a typical kernel-space web server's latency (kHTTpd) is roughly half of a user-space server's latency (Apache). It also shows that full virtualization increases service latency 2-2.5 times. Thus, the addition of safe extensibility through virtual machine techniques likely cancels the performance benefits of employing kernel-based web servers.

Our proposed alternative, kernel plugins, compare favorably latency-wise to the baseline case and the VM solution, as shown in Figure 7. We measure an unmodified kernel web server (baseline), the same server extended with a null kernel plugin (invoked once per request from within kHTTpd), and another copy of the server isolated in a VMware virtual machine. Kernel plugins add minute latency overhead to the service being extended. Figure 7 shows an average plugin overhead of 8  $\mu S$ , though that value is inflated due to the well-known significant variability that interrupt processing induces in Linux [14]. A comparison between the averages of the top 10% timing samples provides a less variable and more accurate estimate of the real plugin overhead (less than 1  $\mu S$ ), consistent with the jitter-free results of Figure 5. The small cost is striking compared to the larger latency overhead imposed by the VM approach. The benefit obtained by using plugins over virtual machines comes from achieving extensibility without complete virtualization. Instead, only the isolation properties of virtualization are really needed. Since kernel plugins are designed to provide exactly that, they are able to avoid unnecessary overhead.

The last macro-benchmark we consider explores the

server's throughput degradation as a function of the isolation technique. We define throughput as the relation between the clients' request rate and a server's sustained reply rate. We measure throughput utilizing the well-known *httperf* benchmark [17].

Figure 8 shows a family of graphs describing the throughput performance of an unmodified kernel-based web server (baseline), a null-plugin modified server, and an unmodified server running in a virtual machine. While kernel plugins preserve the server's ability to handle high throughput almost untouched, the virtual machine-based server is saturated at a little more than half the throughput. Similarly, consistency and predictability inside a virtual machine are severely degraded. In contrast, the kernel plugin approach is remarkably stable and consistent. Again, the disparity is attributed to the many unnecessary non-isolation-related aspects of full virtualization.

We note that the inefficiencies inherent in virtualization schemes like VMware's are known and that there are promising alternatives like Xen [3]. However, Xen requires a host OS kernel to be ported to its specially defined abstract VM model, whereas plugins can be used with existing operating system kernels. The Xen approach, therefore, is complementary to our research.

### 6.3 A Practical Example

To demonstrate the utility and actual real-life performance of plugins we provide a practical example of their use. Specifically, we compare and contrast the performance of a user-space and a kernel-space web server, both with and without extensions. The user-space web server is the popular Apache (version 2.0.48). We extend Apache with an image-transcoding function that reduces image color-depth from 24-bit true-color to 8-bit monochrome. It is an example of a useful extension that a PDA with modest display, CPU, and power resources, could use to adapt images to its capabilities and/or to shift workload to the server. For a kernel-space web server, we use kHTTPd which comes standard with stock Linux v2.4 kernels. We extend kHTTPd with the same color-depth reduction code, placing it into the kernel both as an unprotected kernel function and as a dynamically deployable, isolated kernel plugin. The transcoding function consists of 66 lines of E-code including whitespace and comments and compiles to 371 instructions totaling 1078 bytes of machine-code. For comparison, gcc 3.2.2 without optimizations compiles the same code into 245 instructions totaling 623 bytes, and -O2 optimizations shrink that further to 151 instructions and 338 bytes. Despite its larger size E-code machine-code has roughly similar code path length as unoptimized gcc code, determined by hand comparison of the resulting machine-code. The absolute size dif-

ference is a consequence of E-code's simpler but faster code generation strategy.

Experiments consist of repeatedly requesting images from the web servers and recording request service times (measured at the server side). Timing instrumentation is implemented using the Pentium time-stamp counter, and again 2001 samples are taken, disregarding the first to control against cold OS buffer-cache effects. The images we used were in the Portable Pixmap (PPM) format with sizes 9 KB, 99 KB, 270 KB, and 3.3 MB. The sizes were chosen to approximate both extremes, as well as average typical online image sizes. Most image data on the Internet today is encoded in the JPEG format, which is highly compressed and harder to transcode than the relatively simpler PPM format. To avoid the graphics complexity, yet account for the format differences, we emulate typical JPEG file sizes with the thumb, small, and medium PPM data sets. To emulate the pixel dimension of JPEG files we use the large PPM data [7]. Moreover, note that during color-depth reduction PPMs are reduced by 66%, because each pixel's RGB components are replaced with a single monochrome value. Therefore, the processed images have sizes of 3 KB, 33 KB, 90 KB, and 1.1 MB, respectively.

Figures 9, 10, 11, and 12 present our experimental results. Each figure plots service times for the servers with and without transcoding. The first item to note is the oscillation in the performance of Apache from Figure 9 to Figure 10, as opposed to the performance of kHTTPd. The reason for this oscillation is that the transcoding plugin touches the contents of the entire file during conversion and the time spent transcoding exceeds the time saved by bandwidth reduction for small files. In contrast, kHTTPd does not exhibit such oscillation, despite identical transcoding size reductions. We believe that this is due to a combination of factors related to efficiency gained from co-location in the kernel: (1) avoiding multiple user/kernel protection boundary crossings, (2) related reduction in data copying, (3) benefits from kernel code non-preemptability in Linux, (4) related improvement in CPU cache and TLB performance. In essence, the overhead of reading data from disk should dominate this benchmark, but once the data is in memory (after the OS buffer cache has warmed up), the co-located in-kernel transcoding and the asynchronous network send cost relatively little when compared to their counterparts in user-space, which are further subject to scheduling.

The minimal difference between transcoding costs incurred by the unprotected kernel function and the dynamically deployed kernel plugin suggest that plugins are on the same order of latency. In practice today, function invocations are considered to be essentially 'cost-free'. We view the fact that kernel plugins' costs are comparable as a validation for our design's achievement

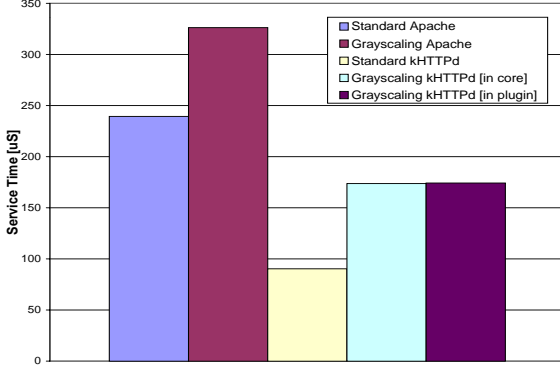


Figure 9: Service times for a thumb image (9KB)

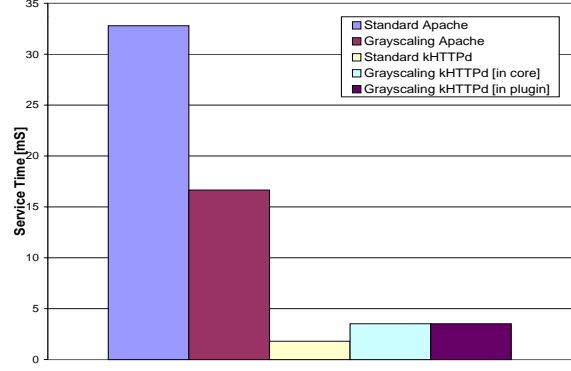


Figure 11: Service times for a medium image (270KB)

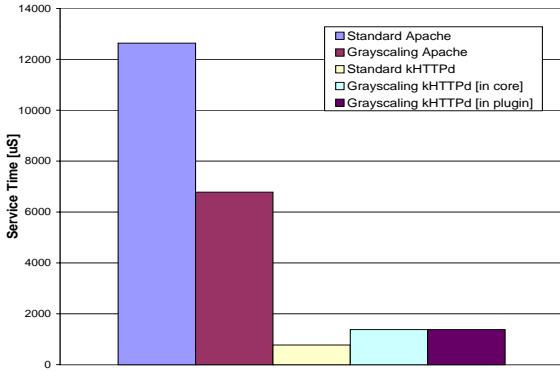


Figure 10: Service times for a small image (99KB)

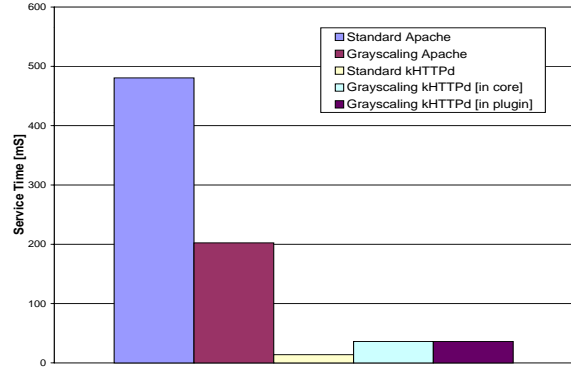


Figure 12: Service times for a large image (3.3MB)

of its efficiency and performance goals.

To summarize, experimental evaluation shows that kernel plugins enable applications to adapt kernel services and extract significant flexibility advantages, while being sufficiently lightweight to not compromise the gains from co-location in the kernel.

## 7 Conclusions and Future Work

We have presented the design, implementation, and evaluation of a novel framework for safe deployment of application-specific code into an OS kernel. The mechanism is based on three key technologies: hardware fault isolation, dynamic code generation, and dynamic linking. HFI relies on commonly available hardware features, and offers low-overhead isolation. Our dynamic code generation is based on E-code, a DCG package developed at Georgia Tech. Using DCG, plugins may be comprised of user-defined code, thereby enabling arbitrary application-specific specializations of the kernel services with which they are associated. Dynamic linking enforces a narrow kernel/plugin interface, provides logical isolation between extensible system-level entities, and eliminates kernel namespace pollution.

Micro-benchmarks evaluating kernel plugins show the

base cost of plugin invocation to be between  $0.45 \mu S$  and  $0.62 \mu S$ . Plugin code generation, linking, and un-linking costs are  $4 mS$ ,  $3.1 \mu S$  and  $1.6 \mu S$ , respectively, for the sample image-transcoding plugin used in this paper. In general, code generation cost depends on code size, and both linking and un-linking costs can be improved further by optimization of the symbol tables currently used in the plugin facility. More importantly, macro-benchmarks and experimental results from a realistic sample application showcase performance advantages offered to end-user applications using kernel plugins in lieu of specializations implemented at user level.

In its current state, the plugin facility fully implements hardware fault isolation, dynamic code generation, dynamic linking, and plugin preemption based on hardware system timers. Planned future work and improvements include tighter integration of code generation and isolation, further performance characterization, exploration of inter-plugin memory protection, implementation of a fault recovery and continuation mechanism, porting the system to Intel's 64-bit Itanium 2 architecture, and optimization of the implementation bottlenecks.

## References

- [1] *Intel Pentium Processor Family Developer's Manual*. Volume 3: Architecture and Programming Manual. Intel Corporation, Santa Clara, CA, 1995.
- [2] A. Banerji and D. L. Cohn. An infrastructure for application-specific customization. In *Proceedings of the 6th Workshop on ACM SIGOPS European workshop*, pages 154–159. ACM Press, 1994.
- [3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th Symposium on Operating Systems Principles*. ACM Press, October 2003.
- [4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 267–283. ACM Press, 1995.
- [5] H. Bos and B. Samwel. Safe kernel programming in the OKE. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming*, pages 141–152. IEEE, 2002.
- [6] F. E. Bustamante, G. Eisenhauer, P. Widener, K. Schwan, and C. Pu. Active streams: An approach to adaptive distributed systems. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, 2001.
- [7] S. Chandra, C. S. Ellis, and A. Vahdat. Differentiated multimedia web services using quality aware transcoding. In *INFOCOM 2000 - Nineteenth Annual Joint Conference of the IEEE Computer And Communications Societies*, March 2000.
- [8] T. Chiueh, G. Venkitachalam, and P. Pradhan. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 140–153. ACM Press, 1999.
- [9] Connectix, Corp. The technology of Virtual PC, 2000.
- [10] G. Eisenhauer, F. E. Bustamante, and K. Schwan. A middleware toolkit for client-initiated service specialization. *ACM SIGOPS Operating Systems Review*, 35(2):7–20, July 2001.
- [11] D. R. Engler. VCODE: a retargetable, extensible, very fast dynamic code generation system. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 160–170. ACM Press, 1996.
- [12] D. R. Engler, M. F. Kaashoek, and J. J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266. ACM Press, 1995.
- [13] A. Gavrilovska, K. Mackenzie, K. Schwan, and A. McDonald. Stream handlers: Application-specific message services on attached network processors. In *Proceedings of the 10th Symposium on High Performance Interconnects*, August 2002.
- [14] A. C. Heursch, A. Horstkotte, and H. Rzehak. Preemption concepts, Rhealstone benchmark and scheduler analysis of linux 2.4. In *Proceedings of the Real-Time & Embedded Computing Conference*, November 2001.
- [15] J. Liedtke. On  $\mu$ -kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250. ACM Press, 1995.
- [16] S. McCanne and V. Jacobson. The BSD packet filter: a new architecture for user-level packet capture. In *Proceedings of the Winter 1993 USENIX Conference*, pages 259–269, 1993.
- [17] D. Mosberger and T. Jin. httpperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59–67. ACM, June 1998.
- [18] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.
- [19] M. Poletto, D. R. Engler, and M. F. Kaashoek. tcc: A template-based compiler for 'C. In *Proceedings of the First Workshop on Compiler Support for System Software (WCSSS)*, February 1996.
- [20] M. Satyanarayanan and C. S. Ellis. Adaptation: the key to mobile I/O. *ACM Computing Surveys (CSUR)*, 28(4es):211, 1996.
- [21] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–227. ACM Press, 1996.
- [22] C. Small and M. I. Seltzer. A comparison of OS extension technologies. In *USENIX Annual Technical Conference*, pages 41–54, 1996.
- [23] A. Tamches and B. P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 117–130. USENIX Association, 1999.
- [24] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.
- [25] VMware, Inc. VMware virtual platform, technical white paper, 1999.
- [26] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216. ACM Press, 1993.
- [27] D. A. Wallach, D. R. Engler, and M. F. Kaashoek. ASHs: application-specific handlers for high-performance messaging. *IEEE/ACM Transactions on Networking (TON)*, 5(4):460–474, 1997.
- [28] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: A scalable isolation kernel. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, September 2002.