Solving the Starting Problem: Device Drivers as Self-Describing Artifacts

Michael F. Spear
Dept. of Computer Science
University of Rochester
Rochester, NY 14627
spear@cs.rochester.edu

Tom Roeder
Dept. of Computer Science
Cornell University
Ithaca, NY 14853
tmroeder@cs.cornell.edu

Orion Hodson
Microsoft Research
One Microsoft Way
Redmond, WA 98052
ohodson@microsoft.com

Galen C. Hunt
Microsoft Research
One Microsoft Way
Redmond, WA 98052
galen.hunt@microsoft.com

Steven Levi
Microsoft Research
One Microsoft Way
Redmond, WA 98052
levi@microsoft.com

ABSTRACT

Run-time conflicts can affect even the most rigorously tested software systems. A reliance on execution-based testing makes it prohibitively costly to test every possible interaction among potentially thousands of programs with complex configurations. In order to reduce configuration problems, detect developer errors, and reduce developer effort, we have created a new first class operating system abstraction, the application abstraction, which enables both online and offline reasoning about programs and their configuration requirements.

We have implemented a subset of the application abstraction for device drivers in the Singularity operating system. Programmers use the application abstraction by placing declarative statements about hardware and communication requirements within their code. Our design enables Singularity to learn the input/output and interprocess communication requirements of drivers without executing driver code. By reasoning about this information within the domain of Singularity's strong software isolation architecture, the installer can execute a subset the system's resource management algorithm at install time to verify that a new driver will not conflict with existing software. This abstract representation also allows the system to run the full algorithm at driver start time to ensure that there are never resource conflicts between executing drivers, and that drivers never use undeclared resources.

Keywords

operating systems, programming language support, dependable computing, experience with existing systems, declarative configuration

EuroSys'06, April 18–21, 2006, Leuven, Belgium.

1. INTRODUCTION

The complexity of device drivers has grown considerably in the last decade as users have come to expect rich features such as hot-swapping and power management. Popular operating systems such as Windows, Linux, and FreeBSD have responded in a variety of ways, but at their core these systems possess the same driver model, with the same inherent weaknesses, as they possessed a decade ago.

The manner in which these systems load device drivers is consistent and dangerous; in all of these systems, the OS loads executable code into the same protection domain as the kernel [8, 41, 40]. Once the driver is installed into this address space, the kernel cannot prevent it from accessing any (or all) hardware in the system. Furthermore, as these drivers are typically written with low-level primitives to access hardware directly, the kernel rarely verifies that drivers use only appropriate hardware resources. Instead, the kernel trusts that the driver will only access hardware for the device it claims to serve. Furthermore, often the kernel cannot guarantee that a driver will respect the memory allocated to processes, or even the memory allocated to other components within the kernel.

Consequently, drivers are among the most unreliable components in the OS. Swift *et al.* [39] report that 85% of diagnosed Windows crashes are caused by drivers. Chou *et al.* [9] found that Linux drivers are seven times more likely to contain bugs than other kernel code. We offer the following four deficiencies as a partial explanation for the unreliability of these systems.

- Device drivers are loaded into the kernel's address space and hardware protection domain with no mechanisms to isolate driver code from kernel code.
- The precise resource needs of device drivers are not declared, and the driver model provides no mechanism for an operating system to verify such declarations even if they existed.
- Every driver is permitted to acquire its hardware resources without monitoring by the kernel, even though this acquisition is rarely device-specific.
- The kernel lacks the information necessary to monitor the behavior of a driver from the instant it begins, and as such cannot perform complete resource management.

We used the Microsoft Research Singularity Operating System [27] to experiment with solutions to the problems of driver reliability and sandboxing. Singularity uses type-safe languages and system-wide metadata to replace hardware protection with verification of memory safety.

In this paper, we extend Singularity's metadata to allow both online and offline verification of driver resource utilization and to ensure three invariants: the OS never installs a device driver that cannot start successfully, the OS never starts a device driver that cannot run successfully, and device drivers never use hardware or IPC resources they haven't declared.

1.1 Singularity Overview

Singularity runs each driver in a separate software isolated process (SIP)[1]. Unlike processes in traditional operating systems, which rely on hardware protection mechanisms to provide memory isolation, Singularity relies on language safety to verify that no SIP is capable of writing onto another SIP's pages. Encapsulated in SIPs, individual drivers can be stopped and restarted as needed without bringing down the entire operating system.

With the exception of a small hardware abstraction layer (HAL), the operating system itself is written in C# and type-safe Sing#, an extension of C# with additional static analysis and embedded support for Singularity's type-safe IPC primitives [28].

All programs in Singularity are statically linked at install time to a trusted runtime. While all programs are statically verified for type safety, each trusted runtime is a component of they system's trusted computing base (TCB). Each trusted runtime encapsulates unsafe code, such as a garbage collector, that cannot be represented in our type system. Trusted runtime code maintains process isolation, allowing processes to run in the privileged/supervisor mode of the host processor without being able to affect the memory and hardware resources of other processes. Dynamic reflection or other mechanisms that can circumvent type safety are not permitted in user code.

The trusted runtime for device drivers provides a managed environment that abstracts communication with hardware. The assembly-level instructions for handling interrupt requests, fixed memory, ports, and direct memory access channels (DMA) are all protected through abstractions exposed by the driver runtime.

All IPC in Singularity is through strongly typed bidirectional channels [26]. These channels have exactly two endpoints, labeled `Exp` (exported) and `Imp` (imported), corresponding to the "server" and "client", respectively. Messages on a channel are restricted to value types, and the format of these messages is defined by a contract. The contract also serves as a channel protocol that specifies valid sequences of messages sent across the channel, and includes a handshaking step to initiate communication. An application's conformance to a contract can be statically verified.

Some endpoints must have a public name in order to allow easy connection by clients. This is achieved through a singly rooted, globally accessible namespace. A global namespace server manages the namespace, and allows the mapping from names to channel endpoints, directories, and symbolic links. The namespace is not attached to a persistent backing store. Instead, system policy permits some applications (such as the filesystem) to create virtual subtrees within the namespace and map content into these trees. This allows the equivalent of a traditional file system, with the distinction that file access is through the channel abstraction.

[1]While all processes run in the same address space, they cannot share language objects with each other. Communication between processes involves only value types, and transfers exclusive ownership of affected memory from the sender to the receiver.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
  <application ...>
  <signatures ...>
  <assemblies ...>
  <driverCategory ...>
</manifest>
```

Listing 1: Key Tags in the Driver Manifest

In Singularity, application installation is a privileged operation. During installation, the OS compiles an application from type-safe MSIL[2] to native code with the Bartok compiler [19]. While conducting this compilation, the installer can also use static analysis to enforce system policy, as well as language safety rules. In addition, since applications are presented to the installer as MSIL assemblies, they carry their programmer-created metadata internally, enabling easy inspection and verification by the installer.

1.2 Contributions

Our research exploits the attributes of Singularity to improve the management of device drivers. We offer three contributions, which comprise the three main sections of this paper.

First, we have created an abstraction for treating applications as first-class entities, which enables the operating system to reason about applications and provide guarantees. We make device drivers a subclass of this abstraction, and make installation a first-class operation performed by the OS on applications.

Secondly, we present a simple language extension for declaring the I/O and IPC resource requirements of a device driver. In contemporary systems, these requirements can only be inferred from the state of executing code. In Singularity, we have created a specification for this information that is verifiable at compile time, install time, boot time, and execution time. In effect, the specification turns the device driver into a self-describing artifact. Given the set of MSIL assemblies for the device driver, the OS can reason completely about the hardware and software preconditions that must be met in order for the device driver to function correctly. Our language extension has the added benefit of simplifying the development process for driver writers.

Third, we provide extensions to Singularity that use the application abstraction and driver resource declarations to provide guarantees about the I/O and IPC resources used by a device driver. Our extensions allow Singularity to detect resource conflicts before drivers execute, infer a valid total boot order from strictly declarative syntax, and automatically generate significant driver initialization code. These capabilities increase the reliability and maintainability of the system with no significant cost in run-time performance.

The combination of these three features gives Singularity the ability to reason about all driver resource requirements, preconfigure driver resources, and vet driver access to hardware in detail from system startup to shut-down.

2. THE APPLICATION ABSTRACTION

Existing operating systems lack an aggregate concept of an application. While processes effectively encapsulate the resources possessed by a running code segment, and threads represent schedulable, executable segments of that code, neither is an appropriate level at which to represent the relationship between the

[2]Microsoft Intermediate Language (MSIL) is a superset of the Common Intermediate Language (CIL) [29]. MSIL is the CPU-independent byte code format used by Microsoft's .NET compilers and tools.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>

 <application identity="Sb16" />

 <assemblies>
  <assembly filename="Sb16.exe" />
  <assembly filename="Namespace.Contracts.dll" />
  <assembly filename="Io.Contracts.dll" />
  <assembly filename="Singularity.V1.ill" />
  <assembly filename="Corlib.dll" />
  <assembly filename="Singularity.DriverRuntime.dll" />
 </assemblies>

 <driverCategory>
   <device signature="/pnp/PNPB003" />

   <ioPortRange   index="0"        baseAddress="0x220" rangeLength="0x10" />
   <ioPortRange   index="1"        baseAddress="0x380" rangeLength="0x10" />
   <ioIrqRange    index="2"        baseAddress="5"     rangeLength="1" />
   <ioDmaRange    index="3"        baseAddress="1"     rangeLength="1" />
   <ioDmaRange    index="4"        baseAddress="5"     rangeLength="1" />
   <ioMemoryRange addressBits="24" alignment="0x20000" rangeLength="0x4000" fixed="True" />

   <extension contractName="Microsoft.Singularity.Extending.ExtensionContract"
              startStateId="3"  endpointEnd="Exp" assembly="Namespace.Contracts"
              version="0.0.0.0" culture="neutral" publicKeyToken="null">
     <imp>
       <inherit name="Microsoft.Singularity.Channels.Endpoint" />
       <inherit name="Microsoft.Singularity.Extending.ExtensionContract.Imp" />
     </imp>
     <exp>
       <inherit name="Microsoft.Singularity.Channels.Endpoint" />
       <inherit name="Microsoft.Singularity.Naming.ServiceContract.Exp" />
       <inherit name="Microsoft.Singularity.Extending.ExtensionContract.Exp" />
     </exp>
   </extension>

   <serviceProvider contractName="Microsoft.Singularity.Io.SoundDeviceContract"
                    startStateId="3"  endpointEnd="Exp" assembly="Io.Contracts"
                    version="0.0.0.0" culture="neutral" publicKeyToken="null">
     <imp>
       <inherit name="Microsoft.Singularity.Channels.Endpoint" />
       <inherit name="Microsoft.Singularity.Naming.ServiceProviderContract.Imp" />
     </imp>
     <exp>
       <inherit name="Microsoft.Singularity.Channels.Endpoint" />
       <inherit name="Microsoft.Singularity.Naming.ServiceProviderContract.Exp" />
     </exp>
   </serviceProvider>

 </driverCategory>
</manifest>
```

Listing 2: Pre-Installation Manifest for a Sound Blaster

underlying application and the OS. In particular, the files that comprise the application, the set of legal security contexts for an application, and the IPC and I/O resources that are potentially accessed by an application are not properties of an instance of the program, but of the program as it has been configured in a particular system. Abadi et al. present illustrative examples of advanced access control lists [1], none of which can be directly applied to the basic units of management of applications in traditional operating systems, such as files, processes, and threads.

The problem is that the traditional application takes different forms at various levels of the system, and each level imposes its own constraints on the nature of the application. At the filesystem level, the application is a collection of files, often structurally linked through a folder hierarchy. At the system configuration level, the application is a tree in a registry, or a set of files in the `/etc` directory. At run time, the application is a process hierarchy. The network sees the application as a set of addresses and ports.

Unlike traditional operating systems, in Singularity these multiple views of an application are unified with a consistent OS application abstraction. We define this abstraction as a tuple $App = \{R, P\}$, where R is a set of resources (content mapped into the namespace, channels, hardware) and P is a set of declarative policies on those resources. Since there is a global namespace, we represent this tuple as a directory containing a distinguished entry, containing the manifest, and a protected folder tree. The manifest lists all resources, with the constraint that all content associated uniquely with the application (such as executable code) must be in the protected folder tree. The manifest also specifies all policy relating to the application, and is the only logical unit by which the OS can reason about applications. Through this manifest, applica-

tions can have a lifecycle that extends from compilation through installation and execution. Furthermore, in the case of drivers, changes to the manifest are permitted only during installation.

This application abstraction is the enabling idea for the remainder of this paper. In particular, since the application is a first-class entity, we can define a special subclass of applications that specifically applies to device drivers, and then we can set system policies that apply exclusively to this class. Furthermore, the specific nature of installation as we define it allows for semantic analysis and verification of system policy on every application at install time. Since installation is rare, and since the cost of a faulty installation is tremendous, we believe that the impact of a longer installation procedure is more than justified by its benefits.

2.1 The Manifest

Listing 1 shows the key tags of a device driver manifest. The installer is only permitted to make changes to content in the `application` tree, and to add signatures to the installed manifest in the `signature` tree. The `assembly` and `driverCategory` trees provide the entire declaration of all software components of the driver and all I/O and IPC resources the driver needs in order to run.

Other classes of applications might have trees for command-line parameters, run-time metadata (such as the last n files opened), and user configuration options. However, the existence of these tags in a device driver's manifest are illegal and will prevent installation. Each device driver is compiled to a single binary image whose name is stored in the `application` tree by the installer.

In Listing 2 we present the compile-time manifest for a Sound Blaster 16 driver. This manifest is most typically generated automatically as part of the build step, and it is machine and installation-agnostic. The entire `assemblies` and `driverCategory` trees are fixed at compile time. We can statically verify at installation time that these trees have not been modified.

Through these two fixed trees, the compile-time manifest lists all assemblies that comprise the application, as well as sufficient metadata to enable the kernel to completely construct all I/O and IPC resources required by the driver at run time. The manifest is completely declarative, contains no system state, and is general to any use of this driver in any Singularity system.

However, this purely declarative representation does not yet describe a runnable application. In order to trust the accuracy of the manifest, it must be verified and installed by the system. In our design, the manifest cannot be modified except through trusted operations. Thus we can ensure that only the installer modifies those elements of the manifest that apply to installation.

Furthermore, the manifest is the focal point for all system policies that deal with applications. The manifest is expressed in XML and can be easily analyzed off-line, as we will discuss later in Section 4.

2.2 Installation

To install the driver in Listing 2, we perform a set of analytical steps and transformations. The actual changes to the manifest apply only to the `application` and `signature` sections, and appear in Listing 3.

The installation process has three main operations: it guarantees that the driver is appropriate for the system, it optimizes the driver for the system by compiling it to native code, and it records sufficient signature information to ensure that the installed application is not modified except by the installer. The signature algorithm is specified in the system policy.

Installation begins by statically analyzing the entire set of assemblies. The Singularity bytecode verifier tests that every channel is implemented according to its contract, and determines if the application is a driver. Based on this analysis, we can check that if an application is not a driver, it contains no references to the `Io*Range` object family (which provide direct access to hardware); if the application is a driver, we check that any declaration of `Io*Range` objects conforms to the coding standards for device drivers presented in the next section. We can also check the IPC channels used by the application, to enforce any policy defined by the system. Our current system, for example, can limit drivers to `ServiceProvider` channels (which listen for service requests) and one `Extension` channel (connecting the application to its parent, in this case the kernel). The `ServiceProvider` is a second-order contract through which clients connect a device-class specific contract, such as the `SoundDeviceContract`. General applications are permitted to possess channels to access the global namespace (and therefore access data files), as well as other endpoints for interacting with drivers and other `ServiceProviders`.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest>
 <application identity="Sb16">
  <properties>
   <code main="True"
         path="/drivers/Sb16/private/Sb16.x86" />
  </properties>
 </application>
 <signatures>
  <installer>
   <secureHash id="1" value="(hash value)" />
   ...
  </installer>
  ...
 </signatures>
 <assemblies>
  ...
 </assemblies>
 <driverCategory>
  ...
 </driverCategory>
</manifest>
```

Listing 3: Installed Manifest for a Sound Blaster

If all system-specified tests succeed, we compile the entire set of assemblies into a single, statically-linked, native-instruction executable using the Bartok optimizing compiler [19]. The name of this executable and a suitable path are added as `code` properties in the `application` tree of the manifest. Following this step, we generate the signature tree that enables the system to verify that the application has not been modified, according to the system's security policy. Following this process, the manifest is placed in its distinguished location in the system namespace, and the executable is stored in the namespace according to the path attribute.

Since the final compilation is performed by the operating system during installation, we can inline trusted code from the appropriate runtime libraries. Thus for drivers we can inline instructions from the trusted `Singularity.DriverRuntime.dll` library, which is the only process runtime library that implements the `Io*Range` family of objects. By deferring linking and final compilation until the installation step, and using the application abstraction to prevent uninstalled applications from executing, we can use static analysis to verify design-time rules about how objects that access hardware are constructed, used, and destroyed. Furthermore, since the kernel application binary interface (ABI) supports versioning and is backward compatible, the entire kernel can be upgraded without impacting this installed driver; the driver will con-

tinue to use the older version of ABI calls until it is updated by its designer.

Singularity can analyze the manifest to determine if a given driver should be installed. The driver must be capable of serving some device attached to the system, as determined by matching the device signatures declared in the manifest to the hardware signatures present on the machine. The installer also inspects every channel and `Io*Range` declaration in the manifest to verify that there are no conflicts over fixed resources and no unsatisfied dependencies on channels. Additional system policy can also regulate driver updates or restrict the installation of generic drivers.

Singularity also uses the manifest at boot time, as discussed in Section 4, to ensure that no conflicts arise when devices are added or removed. The manifests aid in creating a total order in which drivers are loaded, and give the kernel sufficient information to preallocate and track all resources used by each driver. The language support we present in the following section greatly improves the accuracy of driver metadata.

3. DECLARING DRIVER RESOURCES

In order for our application abstraction to have value, the manifests must be correct and easy to produce. In the previous section, we assumed that the compiler would simply create such manifests given only source code.

To generate such output from the compiler, we created a declarative syntax for resource configuration using MSIL custom attributes [29]. These attributes are programmer-defined, statically checked, and easy to locate in the metadata of a Sing# program.

Our attributes provide the compiler with two main categories of information. First, the attributes identify the device signature[3] of the physical device the driver serves. Secondly, the attributes declare the default configuration for all hardware and IPC objects that the driver will use. This information is sufficient to generate the metadata upon which we rely.

To increase the accuracy of these attributes, we implement an additional compilation step that enforces strict rules on their use. Additionally, we use a MSIL transformation feature of the Sing# language to generate resource configuration code based on the values of these attributes. In this manner we prevent the programmer from using the attributes incorrectly, while rewarding their correct use by automatically generating boilerplate code.

3.1 Declaring Resource Requirements

The trusted Singularity driver runtime provides four objects for interacting with hardware directly, the `IoPortRange`, `IoMemoryRange`, `IoIrqRange`, and `IoDmaRange` objects. These objects all derive from the abstract `IoRange` object, and none has a public constructor. The driver runtime does not allow programs to cast arbitrary objects to these types, and since these types are reference types, they cannot be transmitted over channels. The only way to construct any of these objects is through a protected call to the trusted driver runtime.

The `Io*Range` family of objects provides full support for modern devices, such as those that use programmed I/O and memory-mapped I/O. In addition to standardizing access to these hardware resources, `Io*Range` objects also provide an interface by which the operating system can mediate dynamic resource configuration and detect inappropriate behavior, such as attempts to map an IRQ to an unavailable line or attempts to access memory mapped to another driver.

[3]Every plug-and-play ISA and PCI device can be queried during bus enumeration to determine its signature. This signature identifies the generic function of the device. PCI signatures also identify the vendor ID, a vendor-defined device ID, and a revision number [33].

```
[DriverCategory]
[Signature("/pci/03/00/5333/8811")]
class S3TrioConfig: DriverCategoryDeclaration
{
  // Hardware resources from PCI config
  [IoMemoryRange(0, Default = 0xf8000000,
      Length = 0x400000)]
  internal readonly IoMemoryRange frameBuffer;

  // Fixed hardware resources
  [IoFixedMemoryRange(Base = 0xb8000,
      Length = 0x8000)]
  internal readonly IoMemoryRange textBuffer;

  [IoFixedMemoryRange(Base = 0xa0000,
      Length = 0x8000]
  internal readonly IoMemoryRange fontBuffer;

  [IoFixedPortRange(Base = 0x03c0,
      Length = 0x20)]
  internal readonly IoPortRange control;

  [IoFixedPortRange(Base = 0x4ae8,
      Length = 0x02)]
  internal readonly IoPortRange advanced;

  [IoFixedPortRange(Base = 0x9ae8,
      Length = 0x02)]
  internal readonly IoPortRange gpstat;

  // Channels
  [ExtensionEndpoint(typeof(Extension-
      Contract.Exp))]
  internal
  TRef<ExtensionContract.Exp:Start> iosys;

  [ServiceEndpoint(typeof(VideoDevice-
      Contract.Exp))]
  internal
  TRef<ServiceProviderContract.Exp:Start> video;

  // Static accessor
  internal readonly static
  S3TrioResources Values;

  // Auto-generate resource acquisition code
  reflective private S3TrioResources();
}
```

Listing 4: Example Driver Resource Declaration

Two common patterns exist by which drivers acquire such ranges. In the case of some ISA devices and legacy PC devices for video and keyboards, the driver requires access to hard-coded resources at known places (such as the `textBuffer` in Listing 4). For most devices, however, the process of enumerating a bus to identify devices reveals not only the signature and physical location of the device on the bus, but also the full set of hardware resources that are allocated to the device.

Thus we have created two identical sets of metadata attributes for dynamic resources and fixed resources. These attributes are constructed with type-checked Sing# code and differ only in number of parameters. The dynamic versions take as an extra parameter the index of the corresponding resource in the ACPI[24], PCI[33], or PnP[10] configuration spaces.

In a similar manner, we defined attributes for the major classes of channel endpoints. The constructors for these attributes accept any channel class derived from some well-known parent; in this manner the OS need not predefine the specific protocol by which two

processes communicate, yet the OS can verify the type hierarchy defining the unknown protocol.

Listing 4 presents the full Sing# code for a class encapsulating all the hardware and IPC resources used by the Singularity S3Trio video driver. We encapsulate all of the objects in a class, derive the class from a known ancestor, and add decorations to the class itself, indicating that this class has special relevance to the installer (`DriverCategory`) and identifying the signature prefix of devices for which this driver is appropriate (in Listing 4, the prefix is a complete signature). All accesses to the class are through the static accessor, following the form `S3TrioConfig.Values.textBuffer`.

3.2 Decreasing Effort, Increasing Accuracy

At this point, it may appear that we have done little more than a system such as Javadocs [37] or Doxygen [43], which graft syntax onto the comments of a program to aid in the automated generation of documentation. Our attributes appear to merely do this in-band, employing the compiler instead of a third-party tool. However, Listing 4 is complete; in particular, the final line is not an elided constructor; it is the *entire* constructor.

By using the Sing# `reflective` keyword, we can associate a MSIL transformation known as compile-time reflection (CTR) to entirely automate the runtime calls and ABI calls required to claim exclusive ownership of the driver's resources. CTR is similar to meta-programming; programs are permitted to contain placeholder elements (classes, methods, fields, and accessors) that are subsequently expanded by a generator. Generators are written in Sing# as transforms that express both a pattern matching program structure and a code template. The transformations are applied to the compiled MSIL for each device driver at install time. The generated code can be statically checked. Functionally, the transforms are part of the trusted computing base. Listing 5 provides the result of a reflective transformation on our device driver resource constructor.

In theory, a programmer could write the exact same code, but there is no benefit to doing so. With only a small number of runtime calls for getting `Io*Range` objects, this code is simply boilerplate. Our use of CTR lets the programmer write less code, and thereby decreases the likelihood of incorrectly implementing boilerplate code. The use of CTR also encapsulates implementation-specific code that may change in future versions of the OS.

We contend that in this case CTR does more than simply reduce keystrokes; it is a bridge to thinking about resource acquisition declaratively. In this mindset, Listing 4 is Sing# syntax that declares what resources are expected, what they are to be named within the program scope, and what rule should be used to acquire the resources. The driver writer does not "get" resources; he expects the kernel to provide resources that satisfy his requirements, according to the kernel's declarative policy.

Furthermore, the driver writer and the kernel are now linked together in their dependence on accurate metadata; the kernel will use the decorations (as represented in a manifest) to allocate resources to the driver, and the driver writer will use the metadata (indirectly, through CTR and calls to the trusted runtime) to acquire and configure `Io*Range` objects. This mutual dependence creates an incentive for writing accurate metadata, and we consider it a key feature of our design.

3.3 New Semantic Rules

Without additional semantic analysis, our attributes can introduce rather than prevent errors. For example, a programmer could accidentally place a `FixedIoPortRange` attribute on an

```
[DriverCategory]
[Signature("/pci/03/00/5333/8811")]

class S3TrioConfig : DriverCategoryDeclaration {
  ...
  // Set the accessor via a static constructor
  static S3TrioResources() {
    Values = new S3TrioResources();
  }

  private S3TrioResources() {
    // Single run-time call wrapping multiple
    // ABI calls to get fully configured
    // hardware objects
    IoConfig config = IoConfig.GetConfig();

    // Debug output to log file
    Tracing.Log(Tracing.Debug,
        "Config: {0}", config.ToPrint());

    frameBuffer = (IoMemoryRange)
                  config.DynamicRanges[0];
    textBuffer  = (IoMemoryRange)
                  config.FixedRanges[0];
    fontBuffer  = (IoMemoryRange)
                  config.FixedRanges[1];
    control     = (IoPortRange)
                  config.FixedRanges[2];
    advanced    = (IoPortRange)
                  config.FixedRanges[3];
    gpstat      = (IoPortRange)
                  config.FixedRanges[4];

    iosys =
      new TRef<ExtensionContract.Exp:Start>
        ((!)Extensions.GetStartupExtension-
            Endpoint(0));
    video =
      new TRef<ServiceProviderContract.Exp:Start>
        ((!)Extensions.GetStartupService-
         ProviderEndpoint(1));

    base();
  }
}
```

Listing 5: A Post-Transformation Constructor

`IoMemoryRange` object. Fortunately, the machinery necessary to prevent such errors is simple.

The root of the problem is that the rules governing the use of MSIL attributes are more loose than we would like. We have addressed this problem by adding a compilation step for drivers that understands the following four classes of rules, which are more strict than the MSIL specification:

- **Field specificity**: An attribute decoration X may only be applied to fields of type Y.
- **Object Ancestry**: An attribute decoration X may only be applied to classes derived from class Y.
- **Required Decorations**: Every object, field, or variable of type Y must be decorated with an attribute in the set X.
- **Decoration Hierarchy**: Any instance of the decoration X1 must be within a class decorated with X2.

Our added step in the driver build process is a simple parse of MSIL assemblies, and is sufficient to ensure that all of the driver-specific attributes only appear within a class structure as shown in

Listing 4. Additionally, these rules simplify the extraction and validation of driver resource requirements and ensure the accuracy of the CTR reflective transformation

3.4 Impact

Our model simplifies the development of drivers by removing resource configuration from the domain of the programmer, and employs several mechanisms to achieve a higher degree of metadata correctness than is available in other systems. Furthermore, we have separated the configuration of drivers from the declaration of system state. The driver does not specify where its endpoints are to be connected, or how they are to be named. The driver can only communicate with the OS through its `Extension` channel, and can only listen for service requests through its `ServiceProvider`, an opaque but strongly typed connection to the rest of the system. Additionally, the driver cannot communicate with hardware except through standardized `Io*Range` objects.

Due to hardware limitations, fraudulent or incorrect use of DMA can overwrite physical memory that does not belong to the driver. We anticipate that future architectures will provide DMA memory protection to address this problem [20]. With such protection in place, a Singularity driver will not be able to affect kernel data structures directly or construct arbitrary `Io*Range` objects. With proper OS support, the driver will be completely sandboxed with regard to its I/O and IPC resource consumption.

4. OS SUPPORT

Through our declarative syntax and application abstraction, we can provide the operating system with trustable metadata for every driver. By crafting this entire driver initialization model as an end-to-end system, we can then push this trusted information directly into the OS to transform the system boot procedure so that it is more declarative, more statically verifiable, and more stable.

4.1 Installation

As we have mentioned earlier, installation is a first-class operation in Singularity. However, it is not exclusively the responsibility of a running system. Based on our declarative design, we can install applications off-line, so long as we possess a description of the hardware, a manifest for the new application, and the full declarative specification of the system as it is currently configured.

Since all drivers and applications are abstract entities, the installer reasons about them by their declarations. For example, the S3Trio driver we showed in the previous section declares a set of hardware resources upon which it depends, and a set of endpoints (in this case one `ServiceProvider`) upon which other applications can depend.

Given a bootable system, a new driver is installable if there exists a partial order that includes it and every installed driver in the system, such that all requirements of each driver are satisfied on start, and all endpoint resources provided by the driver are considered available at the instant the driver is activated. We do not concern ourselves with a total order at this time, as we are only computing the feasibility of installation.

The hardware and IPC requirements of a driver are fundamentally different in this calculation. For an IPC resource, an unlimited number of applications may connect to a single `ServiceProvider` channel, and thus once that channel is provided, we assume that it persists for the remaining lifetime of the system.

In contrast, we demand that no two drivers use the same physical resource, unless they both explicitly state in their metadata that they expect the resource to be shared (resource sharing is necessary, for example to support the master and slave drives on a legacy IDE bus). Furthermore, our installation assumes that the kernel will choose which driver to run based on a longest prefix search. Thus if a generic video driver is installed for the signature `/pci/03/00`, and our S3Trio driver serves the signature `/pci/03/00/5333/8811`, then the installation will note that if the S3Trio signature is the only video signature in the system, the generic driver will not run. System policy specifies how the installer will behave when such conflicts arise.

```
Sing#:
...
[ServiceEndpoint(typeof(NicDeviceContract.Exp))]
internal
  TRef<ServiceProviderContract.Exp:Start> nicsp;
...

Manifest:
...
<serviceProvider
    startStateId="3"
    contractName="Microsoft.Singularity-
        .Io.Network.NicDeviceContract"
    endpointEnd="Exp"
    assembly="Io.Contracts"
    version="0.0.0.0"
    culture="neutral"
    publicKeyToken="null">
 <imp>
  <inherit name="Microsoft.Singularity-
        .Channels.Endpoint" />
  <inherit name="Microsoft.Singularity-
        .Naming.ServiceProviderContract.Imp" />
 </imp>
 <exp>
  <inherit name="Microsoft.Singularity-
        .Channels.Endpoint" />
  <inherit name="Microsoft.Singularity-
        .Naming.ServiceProviderContract.Exp" />
 </exp>
</serviceProvider>
...
```

Listing 6: Code and Manifest for Network Controller

It is important to note that even after installation, we have not created much system state. Certain information, such as the location of the manifest and driver in the global namespace, is necessary. However, we still haven't given a public name to any endpoint or provided any imperative instruction for how the system should behave. Thus the installed driver is still stateless with respect to the installation order.

4.2 System Policy at Boot Time

The installation routine ensures that a driver will not be added to the system if it would conflict with existing drivers. However, this does not guarantee that the driver will run. We have necessarily limited the imperative state present in manifests up to this point, in order to ensure that the state of a running system's software is tied only to its individual components' declarative configuration in manifests, the declarative system policy, and the state of the physical resources at run time. For example, we have not yet specified what to do if multiple S3Trio video cards appear. Currently, this is handled through the system policy that is applied at run time.

In Singularity, applications communicate only through channels. Thus we can create a powerful policy for permitting (or prohibiting) driver execution simply by placing constraints on what types and counts of `ServiceProvider` contracts are allowed. Listing 6 depicts the Sing# declaration and corresponding

```
<namePolicy>
 <name contract="Microsoft.Singularity.Io-
         .Network.NicDeviceContract"
       nsName = "/dev/nic" />

 <name contract="Microsoft.Singularity.Io-
         .Network.NicDeviceContract"
       nsName = "/dev/nic"
       allowMultiple="True" />

 <name contract="Microsoft.Singularity.Io-
         .Network.NicDeviceContract"
       nsName = "/dev/nic"
       allowMultiple="True"
       limit="2"/>
</namePolicy>
```

Listing 7: Three Policies for NIC Configuration

manifest for the publicly accessible endpoint of a network interface card (NIC). Since the parameter to the attribute constructor was `typeof(NicDeviceContract.Exp)`, the manifest correlates the particular `ServiceProvider` endpoint of the driver to this channel type. Using this metadata, Singularity will not reason about the endpoint generically; instead it is a gateway through which only `NicDeviceContract` endpoints are passed. This allows a system policy for arbitrary contract types that are not known at kernel compile time.

The system policy is a set of structured XML declarations which define the behavior of a Singularity system. The system policy currently consists only of a `namePolicy` section, which regulates the creation and management of endpoints. As the OS matures, this policy is certain to grow. However, even in its current incarnation it allows the concise declaration and implementation of interesting configurations.

Listing 7 depicts three possible policies for giving names to `ServiceProvider` endpoints in NIC device drivers, all of which are expressible in the existing policy syntax. The first policy states that there is one public name that any such endpoint may have, the second states that an arbitrary number of names can be created using the `nsName` attribute as a prefix, and the third states that no more than two such global names can be created (i.e. `/dev/nic0` and `/dev/nic1`). During the construction of a total order at boot time, Singularity applies these policies to classes of drivers based on their function, rather than their name. We believe this distinction will permit more flexible system configuration, and the exploration of its implications is a future research direction.

4.3 Creating a Total Order and Loading

We now show how declarative manifests and declarative system naming policy are used to load drivers. Our installer ensures that we install only drivers that are capable of starting (i.e., their resource needs are not provably unfillable). In a similar manner, our boot sequence ensures that we only activate drivers that are capable of running (i.e. their resources needs can be satisfied at a point in the boot sequence that is consistent with a valid partial order).

At boot time, Singularity identifies the root bus of the system and enumerates it, identifying an initial set of devices and (depending on the bus type) their hardware resources. These devices include other buses, timers, and user-installed expansion cards; we think of this as the OS learning some of the state of the machine.

Given this partial state, our device activation algorithm opportunistically enumerates a bus (through a bus device driver) whenever one is found, thereby learning more state, and otherwise activates a driver for some device in the system. In both cases, the driver's requirements must be satisfied in advance.

```
[DriverCategory]
[Signature("/pci/02/00/10de/0056")]
class NvMacResources : DriverCategoryDeclaration
{
  [IoMemoryRange(0, Default = 0xfebf9000,
      Length = 0x1000)]
  internal readonly IoMemoryRange imr;

  [IoPortRange(1, Default = 0xf000,
      Length = 0x08)]
  internal readonly IoPortRange ports;

  [IoIrqRange(6, Default = 0x0b)]
  internal readonly IoIrqRange irq;

  [ExtensionEndpoint(typeof(Extension-
      Contract.Exp))]
  internal
  TRef<ExtensionContract.Exp:Start> iosys;

  [ServiceEndpoint(typeof(NicDevice-
      Contract.Exp))]
  internal
  TRef<ServiceProviderContract.Exp:Start> nic;

  internal readonly static NvMacResources Values;

  reflective private NvMacResources;
}
```

Listing 8: nForce4 Resource Declaration

For devices that use the PCI configuration space to acquire resources, the kernel uses the manifest to identify and acquire appropriate resources. To illustrate this process, we present the configuration of an nForce4 network interface controller (NIC) in figure Figure 8. In this declaration, the driver requires three `Io*Range` objects whose base addresses are determined by the PCI bus. When the PCI bus driver runs, it will identify a device whose signature starts with `/pci/02/00/10de/0056`. The bus driver will also assign to this device a set of resources from the dynamic PCI configuration space. For every physical device in on the PCI bus, the bus driver will pass to the kernel a tuple consisting of the full device signature, the physical location of the device, and the set of dynamic resources enumerated from the PCI configuration space. Other configuration spaces such as PnP and ACPI are similarly enumerated to discover devices and dynamic resources.

During the driver activation loop, if Singularity determines that an installed driver's signature is the longest matching prefix of a physical device, it will then check that every IoFixed*Range declaration in the metadata of the driver can be exclusively assigned to this instance of the driver[4]. Then the kernel ensures that every `Io*Range` object declared in the metadata can be assigned. For the nForce4 NIC, this means that the resource packet[5] returned by the bus driver must contain a memory range of at least $0x1000$ bytes in position 0, a port range that is at least 8 bytes wide in position 1, and an IRQ line in position 6. Lastly, the kernel ensures that it can create and connect all channels declared in the manifest.

We currently support the full range of PCI and legacy ISA devices using this model. There are five key features of our design:

- **Proactive Resource Tracking**: We do all tracking within the OS, without having to inspect the behavior of the driver

[4]Fixed resources exist only for legacy devices, and those devices that provide legacy support. The nForce4 NIC requires no fixed resources

[5]The PCI standard governs the number of positions in this packet and their use.

```
foreach signature in EnumerationResults
  driver = FindByLongestPrefix(signature)
  foreach IoResource in driver.manifest
    if isInUse(resource) then skip
  foreach channel in driver.manifest
    if notAvailable(channel) then postpone

  MarkInUse(signature.resources)
  MarkInUse(driver.manifest.fixed)
  MarkAvailable(driver.manifest.channels)

  Allocate(driver.manifest.fixed)
  Initialize(driver.channels)

  if Activate(driver) fails then
    cleanup(driver, signature)
```

Listing 9: Pseudocode for the Driver Activation Loop

to know what resources it uses. If a driver does not use a resource that it is allocated, we still prevent other software from using it. Additionally, we track both the resources identified by enumeration and the resources that the driver wants to claim explicitly as fixed resources. As a result we can easily support legacy hardware.

- **Strong Support for Generic Drivers**: Furthermore, we adapt to the real resources that enumeration provides, rather than performing either tracking or allocation of dynamic resources based only on driver writer expectations. Thus if the driver writer was unaware of a resource that PnP enumeration gave a device, our system will still track it safely, providing the highest assurance of correct tracking of all I/O resources. Similarly, if enumeration gives a larger range than the driver expected, we can prune the range before handing it to the driver, while still marking the full range as allocated.

- **IPC Preallocation**: Every channel to a device driver is created by the kernel and then passed to the driver. Since channel communication is usually blocking, this lets us start a batch of drivers without worrying about timing for the communication between them. If we waited until a driver's IPC was ready before we started any other application that needed to communicate with the driver, we could potentially hang forever in the kernel. While this does introduce the possibility of several drivers convoying as they wait for the first driver in the chain to complete a lengthy initialization and start processing the messages in its channel, the overall design is simpler and prevents the kernel itself from waiting. Furthermore, this allows strong isolation for IPC, as drivers can be denied constructors to IPC channels.

- **Managed Connections to Drivers**: Using our design Singularity can easily create and connect channels for communicating with device drivers. As a result, there is no need for applications to do so. Instead, the application declares in its manifest that it requires access to certain types of channels, and then relies on its application runtime to create and bind those channels. If a driver is not loaded, the system can prevent dependent applications from running, whereas if the driver is loaded, the system will connect all channels to the driver during application activation.

- **Flexibility**: There is no imperative script; our algorithm adapts to changes in the hardware configuration by resolving simple declarative statements.

Drivers for motherboard resources provide a simple, powerful example of the flexibility of our design. Some of our Athlon-based systems contained a special low pin count chip (LPC) which was responsible for remapping the IRQs of the network and IDE disk controllers. For systems with the LPC chip, failure to remap IRQs would prevent the machine from having either network or disk access.

Rather than create separate distributions of Singularity for the different hardware platforms, with special imperative boot scripts, we added a soft dependency on the LPC driver to the installed metadata of the disk and network drivers. This dependency ensures that the network and disk will not start until after the LPC, if it is found. When the LPC is not found, the disk and network drivers still start, but not until the end of the boot sequence. Any devices that require connections to the network and disk are similarly postponed.

Using these declarative dependencies, the same distribution of Singularity, with the exact same set of installed drivers, can determine not only the best set of drivers to load, but a safe order in which to load the drivers. When a driver's soft requirements cannot be met, the driver loads later in the boot sequence, without violating dependencies. When a driver's hard requirements cannot be met, the kernel will not load it. The failure to load this driver will smoothly trickle through the initialization process, and while we may load a system that does not provide all expected functionality, we will never start a system with erroneous drivers loaded.

4.4 Managed Resource Allocation

The final detail of our implementation is how we create endpoints in an OS that does not support reflection. In the case of device drivers, the `ServiceProvider` endpoint types are visible to the kernel, but our system must support arbitrary endpoint types as well. For example, an application that connects to our NIC driver will require endpoints of type `NicDeviceContract.Imp` and `NicDeviceContract.Exp`. In addition, the `Exp` end should be passed to the NIC driver before the client application starts, so that if the endpoint bind fails, the application won't start.

To support the safe creation of arbitrary endpoint types, we use the `imp` and `exp` subtrees in the metadata. The function of these trees is to provide enough information for the kernel to safely allocate a memory region that precisely matches the shape of a memory region for an unknown endpoint type, with sufficient metadata attached to that region for it to be cast as the unknown endpoint type within the client application. In the current Singularity type system, this includes the full name of every ancestor of a type, tracing back until at least the first parent type that is known by the kernel, as well as the integer index of the start state of the channel.

Using this mechanism, the Singularity kernel can completely bind the initial IPC of an arbitrary application within the kernel in time linear to the ancestry of the endpoints. This feature is the foundation for future research into extending our model to other abstract classes of applications.

4.5 Performance Impact

Our design avoids the need for most run-time checks on drivers. As a result, there is no fundamental obstacle to Singularity drivers running as efficiently as their unsafe counterparts in other operating systems. In Figure 1 we compare the average sequential disk read performance of Singularity to that of Windows XP (Service Pack 2), Linux (Fedora Core 4, kernel version 2.6.11-1.1369_FC4), and FreeBSD (version 5.3). We also include a custom build of Singularity in which bounds checks on the `Io*Range` objects were disabled. All tests were conducted on an AMD Athlon 64 3000+ (1.8 GHz)-based system with 1 GB RAM and a Western Digi-

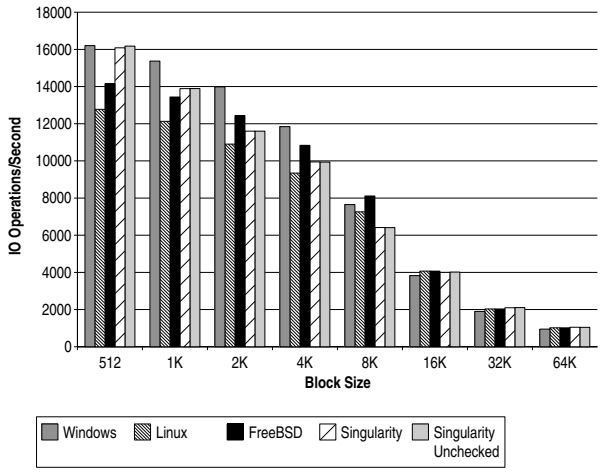

Figure 1: Raw Disk I/O Performance: Sequential Reads

tal WD2500JD 250GB 7200RPM SATA disk (without command queuing).

On each operating system we read 512MB of sequential data from the same disk partition. The benchmarks were all single-threaded, and our results are an average over seven trials. On Singularity, the benchmark communicated with the disk driver process over a channel, whereas FreeBSD, Linux, and XP used system calls.

We measured an overhead of less than 0.6% for Singularity with runtime checks, compared to Singularity without these checks. Furthermore, the disk throughput of Singularity in both cases is competitive with other systems. In return for a slight run-time overhead, our system can track track I/O and IPC resource usage at a fine granularity, before any driver runs. Our language features also aid the programmer in providing an accurate manifest, and reduce the run-time overhead of safe access to hardware. Similarly, language safety ensures that drivers cannot touch kernel data structures and cannot violate their manifests by using undeclared resources.

5. RELATED WORK

Our research touches upon several fields, most notably defect detection, application isolation, and declarative configuration.

5.1 Defect Detection

Engler et al. propose Meta Compilation in [15], a technique by which information specific to a particular domain, such as the use of a resource, is used to develop a compiler extension for identifying violations of system intent. Other defect detection tools include SLAM [5], ESP [11], and Vault [35], as well as Lint [30], LCLint [18], and FiSC [44].

In the same spirit, we have created a new tool for identifying driver errors. Our work applies to a very specific domain (metadata attributes in Sing# drivers), and is unique in that it deals with a type-safe intermediate representation of the program (MSIL), rather than source code or binaries. Furthermore, since we are not interested in supporting legacy code, we can require new programmer declarations (in the form of custom attributes) that greatly simplify the verification process without sacrificing accuracy.

5.2 Application Isolation and Reliability

Our design allows Singularity to implement an isolation model that differs from most existing operating systems. Among microkernel systems that rely on hardware for isolation, such as Mach [2], L4 [23], and Exokernel [16], isolation is achieved by placing the driver in a different hardware protection ring. As discussed previously, monolithic kernels such as Linux and Windows load drivers directly into the kernel address space. In contrast with these systems, our design relies exclusively on type safety.

SPIN [7], JX [22], and KaffeOS [3] use safe programming languages (Modula-3 and Java) for both the kernel and extensions. While they can thus achieve high degrees of protection through language features instead of hardware, our design differs in two key areas. First, we provide a single and explicit abstraction for hardware interaction through a family of objects. Coupled with our declarative metadata, this allows us to completely track resource allocation and prevent conflicts before drivers are started. Secondly, we provide an application abstraction which permits our kernel to distinguish between drivers and applications by inspecting their runtime libraries. Since this check occurs at install time in Singularity, differentiating between classes of applications is essentially free in a running system, while allowing stronger static guarantees about which applications can and cannot acquire particular resources. This also permits us to factor trusted code into the driver executable.

Nooks [39] provides multiple, configurable mechanisms for isolating drivers. While Nooks is very effective at preventing resource conflicts at run time, it suffers from at least one problem that also affects many language-based solutions: it cannot detect resource conflicts until run time. As it lacks accurate driver metadata, Nooks must wait until a conflict is detected and then recover, rather than prevent the execution of drivers that may cause conflicts. An extension to Nooks that employs shadow drivers [38] mitigates the effect of driver failure on applications. This additional indirection does not significantly impact performance.

The Xen virtual machine monitor [6, 20] isolates drivers by placing each in its own I/O virtual machine, and connecting drivers through a unified driver interface. LeVasseur et al. propose another driver model that achieves isolation through virtual machines [31]. In their design, each driver can run in a protected container, with its own full copy of the OS. Erlingsson et al. investigate a similar model for Windows drivers in [17], which delivers backwards compatibility with binary drivers, at the cost of more run-time checks.

The Devil IDL [32] provides a novel mechanism for improving driver reliability. Devil is a high-level language for specifying low-level device operations, and there is considerable overlap between Devil's abstractions and our `Io*Range` objects. As a language-only solution, Devil cannot offer extensive run-time monitoring and management, but their design, based on the recognition that driver reliability must be a design-time consideration, provides a complementary approach.

In the embedded systems domain, the HAIL language [36] is a high-level, operating system-independent language for specifying the low-level interaction between a driver and its device. HAIL provides strong guarantees and aids in the creation of more reliable drivers, but as with Devil, HAIL's focus is on a lower-level abstraction, and does not change the interaction between drivers and the kernel. We believe that HAIL is compatible with our design.

5.3 Declarative Configuration

Dodge et al. [14] outline the initialization and configuration of the Linux and OpenBSD systems in detail, describing among other things the complex process by which IPC resource handles are pre-allocated. In contrast to this mechanism, our model permits bidirectional channels to be allocated on the fly during driver initialization. Furthermore, whereas the systems they describe can make limited guarantees about the initialization of drivers, our system

allows detailed management of this process by the kernel.

Raymond proposed a declarative language for Linux kernel configuration in [34]. This model permits off-line reasoning about how a kernel is configured, but lacks the ability to analyze driver metadata to prevent the installation of drivers for hardware that is not present in the system.

There are also a number of installation management systems for the Linux and BSD Unix operating systems, such as the Redhat Package Manager [4], the Debian Package System [12], the Gentoo Portage System [21], and the venerable BSD Ports [42]. While these systems correctly model dependencies between applications at install time, they rely on external sources of metadata, rather than intrinsic qualities of the code. As a result, these tools cannot provide guarantees about whether a program is runnable, only guarantees that a program can be built and installed without error. Furthermore, these tools focus on applications, and contain only limited support for device drivers.

Compaq's Vesta project [25] also addressed the issue of software configuration, using models to configure single programs. While Vesta provided new tools to integrate revision control, dependency management, and application build, the system remains an application that runs atop the operating system, rather than a fundamental component of the system. As a result, Vesta cannot guarantee that it installed all runnable software on a given system.

DeTreville proposes a broad movement to declarative system configuration in [13], based on lambda calculus. Although he does not present an implementation, his extensive list of failure conditions for a declarative configuration model proved instructive in the design of our declarative driver syntax.

6. CONCLUSIONS

We present an end-to-end design to characterize and manage the IPC and I/O resource needs of device drivers in the Singularity OS. By leveraging the strong metadata features of MSIL, our design overcomes a traditional hurdle to accuracy by making the driver as dependent on correct metadata as the OS. As a result, Singularity knows at all times the resource requirements of every installed driver, and can verify that the declared and available resources match before allowing driver code to run. In addition, Singularity has sufficient information to monitor the communication of its drivers from the moment they begin, enabling it to prevent errors due to inappropriate resource use. By leveraging compiler and language support, the cost of this model is incurred at installation time, allowing Singularity to reason extensively about device drivers without degrading performance.

Furthermore, we have created a new first class operating system abstraction for applications, and defined a subclass of this abstraction for drivers. This abstraction enables us to reduce configuration problems, detect developer errors, and reduce developer effort. Using this abstraction, our drivers are more declarative, and push the responsibility for allocating resources into the trusted computing base.

Beyond the single function device drivers discussed in this paper, we are exploring how to extend these abstractions to cover additional software. We hope to use the generalized application abstraction to describe drivers for bus devices, subsystems like the file system and network stacks, applications ranging from web servers to GUI shells, and an entire software system. We also hope to leverage recent work in cross-process channel contracts, to create contracts that describe the hardware/software boundary between a driver and a device. These contracts will allow us to verify that the driver is interacting correctly with the hardware, and may let us to verify that hardware is interacting correctly with the driver.

Acknowledgments

We are indebted to the rest of the Singularity team at Microsoft Research, especially Mark Aiken, Paul Barham, Manuel Fähndrich, James Larus, Nick Murphy, Bjarne Steensgaard, David Tarditi, and Brian Zill. In addition, Manuel Fähndrich and Michael Carbin developed the compile time reflection implementation used to generate boilerplate driver constructor code. We are grateful to Michael Scott and James Larus for giving feedback on early drafts of this paper. We also thank our shepherd, Brian Bershad, and the anonymous reviewers, for their valuable advice.

7. REFERENCES

[1] M. Abadi, A. Birrell, and T. Wobber. Access Control in a World of Software Diversity. In *Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, Sante Fe, NM, USA, 2005.

[2] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. MACH: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX Summer Conference*, pages 93–112, Atlanta, GA, USA, 1986.

[3] G. Back and W. C. Hsieh. The KaffeOS Java Runtime System. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(4):583–630, 2005.

[4] E. C. Bailey. *Maximum RPM: Taking the Red Hat Package Manager to the Limit*. Red Hat Software, Inc., first edition, 1997.

[5] T. Ball and S. K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *The 29th Annual ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, OR, USA, 2002.

[6] 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 Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, Bolton Landing, NY, USA, 2003.

[7] 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 Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 267–283, Copper Mountain Resort, CO, USA, 1995.

[8] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilley & Associates, Inc., second edition, 2002.

[9] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. R. Engler. An Empirical Study of Operating System Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 73–88, Chateau Lake Louise, Banff, Canada, 2001.

[10] Compaq Computer Corporation, Phoenix Technologies Ltd., and Intel Corporation. Plug and Play BIOS Specification, Version 1.0a. May 5, 1994.

[11] M. Das, S. Lerner, and M. Seigle. ESP: Path-Sensitive Program Verification in Polynomial Time. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, pages 57–68, Berlin, Germany, 2002.

[12] J. Dassen, C. Stickelman, S. G. Kleinmann, S. Rudolph, S. Vila, J. Rodin, and J. Fernandez-Sanguino. The Debian GNU/Linux FAQ Chapter 6–Basics of the Debian Package

Management System. *http://www.debian.org/doc/FAQ/ch-pkg_basics.en.html*, September 2005.

[13] J. DeTreville. Making System Configuration More Declarative. In *Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, Sante Fe, NM, USA, 2005.

[14] C. Dodge, C. Irvine, and T. Nguyen. A Study of Initialization in Linux and OpenBSD. *SIGOPS Operating Systems Review*, 39(2):79–93, 2005.

[15] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 1–16, San Diego, CA, USA, 2000.

[16] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, CO, USA, 1995.

[17] U. Erlingsson, T. Roeder, and T. Wobber. Virtual Environments for Unreliable Extensions. Technical Report MSR-TR-2005-82, Microsoft Research, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, 2005.

[18] D. Evans, J. V. Guttag, J. J. Horning, and Y. M. Tan. LCLint: A Tool for Using Specifications to Check Code. In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering (SOGSPFT '94)*, pages 87–96, New Orleans, LA, USA, 1994.

[19] R. Fitzgerald, T. B. Knoblock, E. Ruf, B. Steensgaard, and D. Tarditi. Marmot: An Otimizing Compiler for Java. *Software–Practice and Experience*, 30(3):199–232, 2000.

[20] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *First Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure (OASIS) at ASPLOS'04*, Boston, MA, USA, October 2004.

[21] Gentoo Foundation, Inc. About Gentoo Linux. *http://www.gentoo.org/main/en/about.xml*, 2005.

[22] M. Golm, M. Felser, C. Wawersich, and J. Kleinöder. The JX Operating System. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 45–58, Monterey, CA, 2002, 2002.

[23] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The Performance of μ-Kernel-Based Systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP'97)*, pages 66–77, Saint-Malo, France, 1997.

[24] Hewlett-Packard Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., and Toshiba Corporation. Advanced Configuration and Power Interface Specification, Revision 3.0. September 2, 2004.

[25] A. Heydon, R. Levin, T. Mann, and Y. Yu. The Vesta Approach to Software Configuration Management. Technical Report 168, Compaq Systems Research Center, March 2001.

[26] G. Hunt, J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fändrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, 2005.

[27] G. C. Hunt and J. R. Larus. Singularity Design Motivation. Technical Report MSR-TR-2004-105, Microsoft Research, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, December 2004.

[28] G. C. Hunt, J. R. Larus, D. Tarditi, and T. Wobber. Broad New OS Research: Challenges and Opportunities. In *Tenth Workshop on Hot Topics in Operating Systems (HotOS X)*, Sante Fe, NM, USA, 2005.

[29] ISO/IEC 23271:2003. Common Language Infrastructure (CLI): Partition II: CIL Instruction Set. *http://www.ecma-international.org/publications/standards/Ecma-335.htm*, 2003.

[30] S. C. Johnson. Lint, a C Program Checker. Technical Report 65, AT&T Bell Laboratories, 1978.

[31] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 17–30, San Francisco, CA, USA, 2004.

[32] F. Mérillon, L. Réveillere, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for Hardware Programming. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, pages 17–30, San Diego, CA, USA, 2000.

[33] PCI Special Interest Group. PCI Local Bus Specification, Revision 2.3. March 29, 2004.

[34] E. S. Raymond. The CML2 Language. *http://www.catb.org/~esr/cml2/cml2-paper.html*, 2000.

[35] M. F. Robert DeLine. Enforcing high-level protocols in low-level software. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI)*, pages 59–69, Snowbird, UT, USA, 2001.

[36] J. Sun, W. Yuan, M. Kallahalla, and N. Islam. HAIL: A Language for Easy and Correct Device Access. In *The 5th ACM International Conference on Embedded Software (EMSOFT'05)*, pages 1–9, Jersey City, NJ, USA, 2005.

[37] Sun Microsystems Inc. Javadoc Tool Home Page. *http://java.sun.com/j2se/javadoc/*, 2005.

[38] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy. Recovering Device Drivers. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 1–16, San Francisco, CA, USA, 2004.

[39] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 207–222, Bolton Landing, NY, USA, 2003.

[40] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, second edition, 2001.

[41] The FreeBSD Documentation Project. FreeBSD Architecture Handbook. *http://www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/*, 2000–2005.

[42] The FreeBSD Project. FreeBSD Ports. *http://www.freebsd.org/ports/*, October 2005.

[43] D. van Heesch. Doxygen. *http://www.doxygen.org/*, 1997–2005.

[44] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using

Model Checking to Find Serious File System Errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 273–288, San Francisco, CA, USA, 2004.