Abolish Runtime Systems!
Operating Systems Should Control the Execution Environment

James Larus
Microsoft Research

Second International Conference on Virtual Execution Environments
June 14, 2006
If you come to a fork in the road, take it.
– Yogi Berra
Crossroad for Languages
Abstraction
Why Did it Turn Out this Way?

• Abstraction is good
  – simpler to build runtime on OS than bare hardware
  – more work to implement OS + runtime

• Limited memory and processor cycles limit language ambitions
  – Good OS performance difficult on limited hardware

• Expertise and goals
  – languages/systems researchers

• Compatibility/interoperability
  – existing software and languages

• Portable platform was an unfilled niche
  – many processors, many OS’es
Not the Right Path

• Language/runtime improvements do not benefit OS or system software

• Duplicate functionality
  – runtime reimplements OS features

• Elaborate runtimes increase cost of language safety
  – decrease attractiveness of better languages

• Foreclose innovation in system architecture
  – cannot trust software properties
  – rely on hardware for isolation and protection
Singularity Project

- Microsoft Research project
  - Galen Hunt & Jim Larus & many others
  - basis for building more dependable software
  - attack dependable from multiple perspectives

- New OS, languages, and tools
  - working research prototype
    - not Windows or CLR replacement
    - no compatibility

- No single magic bullet
  - mutually reinforcing improvements to languages and compilers, systems, and tools
Key Directions

1. Use safe programming languages everywhere
   - safe = type safe and memory safe (like C# or Java)
   - applications, extensions, OS services, device drivers, kernel

2. Improve system resilience despite software errors
   - failure containment boundaries between components and extensions
   - explicit failure notification model

3. Architecture supports verification
   - specify and check behavior at many levels of abstraction
   - core OS architecture facilitates static verification
Singularity OS

- Safe micro-kernel
  - 95% written in C#
    - 17% of files contain unsafe C#
    - 5% of files contain x86 asm or C++
  - kernel sealed at boot time
  - services & device drivers in processes

- Software isolated processes (SIPs)
  - all user code is verified safe
  - some unsafe code in trusted runtime
  - processes sealed at start time

- Safe and efficient communication via strong interfaces (channels)
  - channel behavior is specified, checked.
  - checked behavior allows us to make communication efficient

- Type safety is key to verification and protection
Challenge 1: Pervasive Safe Languages

- Written in extended C#
  - actually Spec# (C# + pre/post-conditions and invariants)
- Added features for systems programming
  - increase programmer control over allocation, initialization, and memory layout
- Explore language design to support programming and verification
  - message passing
  - factoring libraries into composable pieces
  - compile-time reflection
What About The Runtime?

- JVM & CLR’s design not always appropriate
  - rich runtime (“one size fits all”)
    - monolithic, general-purpose environment
    - large memory footprints (~4 MB process for CLR)
    - many dependencies (CLR PAL requires >300 Win32 APIs)
  - JIT compilation
    - increases runtime size and complexity
    - unpredictable performance
  - replicate OS functionality
    - security, threading, configuration, etc.
    - more is less
Singularity Runtime

Libraries

Application

Singularity Runtime (GC, etc.)

Whole Program Optimization

x86 Executable

TAL Proof

Singularity Process

James Larus
Abolish Runtime Systems
Small, Customizable Runtime

- Small execution environment
  - ahead-of-time, global optimizing compiler (MSR Bartok) specializes runtime and libraries
    - eliminate code for unused/disabled language features and unused application/library code
    - factorable runtime and libraries
- Runtime, garbage collector, and libraries selectable on per-process basis
  - reduce memory and computation overhead
  - enforce design discipline and system policies per process
- Eliminate OS functionality from runtime
  - security, resource allocation, etc.
- Provide OS mechanism for enforcing system policy
  - runtime can constrain behavior (e.g. driver environment)
Challenge 2: Improve Resilience

• Software errors should not cause system failure

• A resilient system architecture should
  – isolate system components to prevent data corruption
  – provide clear failure notification
  – implement policy for restarting failed component

James Larus
Process Architectures

Open Process

OS Kernel

Single Process

App

OS

James Larus

Abolish Runtime Systems
Open Process Architecture

- Open processes
  - dynamic code loading and runtime code generation
    - DLLs, Java class loading, browser plug-ins, device drivers in kernel, etc.
  - cross-process memory sharing
  - system API allows one process to alter state of another

- Near ubiquitous (Windows, Unix, etc.)
  - originated in Multics

- Shared state reduces dependability
  - 85% of Windows crashes are caused by third party code in kernel
  - interfaces between extension and host are often poorly documented and understood
  - no isolation boundary between code and extension
  - extension can access non-public interfaces (reflection)
Single Process Architecture

- All code and data in single address space
  - rely on language and memory safety to isolate components
  - dynamic code loading and runtime code generation
  - easy data sharing

- Xerox PARC (Cedar, Smalltalk, etc.) and Lisp Machine model
  - Java and .NET model as well

- Runtime is single point of failure
  - shared runtime must also meet all applications’ requirements
  - Java isolates and CLR AppDomains have complex, shared runtime

- Rely on garbage collection to reclaim resources
  - finalizers

- Difficult to constraint interactions

James Larus
Abolish Runtime Systems
Isolates and AppDomains are Still Interdependent
Singularity Sealed Processes

- Singularity processes are sealed
  - no dynamic code loading or run-time code generation
    - all code present when process starts execution
  - extensions execute in separate processes
    - separate closed environments with well-defined interfaces
  - no shared memory

- Process is fundamental unit of failure isolation

- Better: optimization, verification, security, failure handling
### Size Benefits of Sealed Processes

<table>
<thead>
<tr>
<th>Program</th>
<th>Code Whole</th>
<th>Code w/ Tree Shake</th>
<th>% Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel</td>
<td>2.37 MB</td>
<td>1.29 MB</td>
<td>46%</td>
</tr>
<tr>
<td>IDE Disk Driver</td>
<td>1.85 MB</td>
<td>455 KB</td>
<td>75%</td>
</tr>
<tr>
<td>Web Server</td>
<td>2.73 MB</td>
<td>765 KB</td>
<td>72%</td>
</tr>
<tr>
<td>Content Extension</td>
<td>2.14 MB</td>
<td>502 KB</td>
<td>77%</td>
</tr>
</tbody>
</table>

- Reduces process code size by up to 75%.
- Fewer code paths => better optimization & error analysis
Need for Lightweight Processes

• Existing processes rely on expensive hardware virtual memory and protection mechanisms
  – VM prevents reference to other processes’ pages
  – protection prevents unprivileged code from access system resources (e.g. VM page tables)

• Processes are expensive to create and schedule
  – encourages monolithic program development
  – large, undifferentiated applications
  – dynamic code loading
  – threading to allow independent control flow
Software Isolated Processes (SIPs)

- Protection and isolation enforced by language safety and kernel API design, not hardware
  - process owns a set of pages
  - all of a process’s objects reside on its pages (object space, not address space)
  - language safety ensures process can’t create or mutate references to other pages

- Global invariants:
  - no process contains a pointer to another process’s object space
  - no pointers from exchange heap into process
Interprocess Communications

• Channels are strongly typed (value & behavior), bidirectional communications ports
  – messages passing with extensive language support

• Messages live outside processes, in exchange heap
  – only a single reference to a message

• “Mailbox” semantics enforced by linear types
## Micro Benchmarks

<table>
<thead>
<tr>
<th>Athlon64 3000+ (1.8GHz) nForce4 SLI</th>
<th>Cost (CPU Cycles)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Singularity</td>
</tr>
<tr>
<td>Minimum kernel API call</td>
<td>80</td>
</tr>
<tr>
<td>Message request/reply</td>
<td>1,041</td>
</tr>
<tr>
<td>Process create &amp; start</td>
<td>388,000</td>
</tr>
</tbody>
</table>

- **Why?**
  - static verification replaces hardware protection
  - all SIPs run in ring 0
  - good optimizing compiler (not JIT)
OS Controls Resources

- OS owns, allocates, and reclaims resources
- On process termination, OS reclaims memory pages and channels
  - not dependent on finalization or garbage collection
- Clean failure notification
  - sent messages still available to other process
Would You Trust Your System to a Type System?

- Process integrity depends on type and memory safety
  - currently trust compiler and runtime
- TAL can eliminate compiler from trusted computing base
- Working on verifying the GC as well

Singularity TCB
bartok
x86
Singularity system
application verification

Sing# C# source
csc sgc
MSIL+

compiler verification
byte code verification
x86 safety proof
Hardware Protection is Orthogonal

- SIP-Phys
  - Kernel
  - System process
  - Application

- SIP-Page
  - SIP

- HIP-R0
  - Kernel domain

- HIP-R0-S
  - Non-kernel domain

- HIP-R3
  - Kernel domain

- HIP-R3-S
  - Non-kernel domain
Challenge 3: Facilitate Verification

- Verify process internals (code):
  - type safety
  - object invariants
  - method pre- & post- conditions
  - component interfaces

- Verify process externals:
  - channel contract conformance
  - resource access & dependencies

- Verify system:
  - communication safety
  - hardware resource conflict free
  - namespace conflict free

- Static verification: before code runs
public contract IoStream {
...
state Start : {
  Open? -> {
    OK! -> Opened;
    Error! -> End;
  }
}
state Opened : {
  Read? -> Data! -> Opened;
  Write? -> OK! -> Opened;
  Close? -> OK! -> End;
}
state End;
...
}

? = receive
! = send
Example: **Contract Conformance**

**Contract**

```java
public contract TcpSocketContract {
  ...
  state Connected : {
    Read? -> ReadResultPending;
    Write? -> WriteResultPending;
    GetLocalAddress? -> IPAddress! -> Connected;
    GetLocalPort? -> Port! -> Connected;
    DoneSending? -> ReceiveOnly;
    DoneReceiving? -> SendOnly;
    Close? -> Closed;
    Abort? -> Closed;
  }
  state ReadResultPending : {
    Data! -> Connected;
    NoMoreData! -> SendOnly;
    RemoteClose! -> Zombie;
  }
  ...
}
```

**Web Server (User)**

```java
... conn.SendRead();
switch receive {
  case conn.Data(readData) :
    dataBuffer.AddToTail(readData);
    return true;
  case conn.RemoteClose() :
    return false;
}
...}
```

**Missing Case**

```java
  case conn.NoMoreData() :
    contract conformance catches subtle errors and common causes of live-lock statically
```

---

James Larus  
Abolish Runtime Systems
Example: Configuration Specifications

```c
[DriverCategory]
[Signature("/pci/03/00/5333/8811")]
class S3Trio64Config : DriverCategoryDeclaration {
    [IoMemoryRange(0, Length = 0x400000)]
    IoMemoryRange frameBuffer;

    [IoFixedMemoryRange(Base = 0xb8000, Length = 0x8000)]
    IoMemoryRange textBuffer;

    ...

    [IoFixedPortRange(Base = 0x3c0, Length = 0x20)]
    IoPortRange control;

    [ExtensionEndpoint(typeof(ExtensionContract.Exp))]
    TRef<ExtensionContract.Exp::Start> pnp;

    [ServiceEndpoint(typeof(VideoDeviceContract.Exp))]
    TRef<ServiceProviderContract.Exp::Start> video;

    ...
}

requires PCI Device
requires 4MB frame buffer (from in PCI config)
requires system console buffer
requires VGA I/O ports
requires control by plug-and-play system
Provides video capability to system
```
Specification Used in Many Ways

1. Load driver
2. Allocate I/O objects
3. Create channels
Conclusion

• How can we fundamentally improve the dependability of today’s software and platforms?

• Time to rethink language, OS, and system architecture assumptions
  – OS should control application’s execution environment
  – new mechanisms to enhance system integrity, verifiability, and dependability

• Programming languages and runtimes are central to improving software

• Singularity is a complete (but simple) system
  – uses safe languages all the way down to the hardware
  – changes OS architecture improve system integrity and verification
    • eliminate wasteful duplication
  – verifies many more aspects of system behavior

• http://research.microsoft.com/os/singularity
Two roads diverged in a wood, and I—
I took the one less traveled by,
And that has made all the difference.
— Robert Frost
Large, Diverse Research Team

- Lead by Galen Hunt and Jim Larus

- **MSR Cambridge**
  - Paul Barham, Richard Black, Tim Harris, Rebecca Isaacs, Dushyanth Narayanan

- **MSR Redmond**
  - **Advanced Compiler Technology Group:**
    - Juan Chen, Qunyan Mangus, Mark Plesko, Bjarne Steensgaard, David Tarditi
  - **Foundations of Software Engineering Group:**
    - Wolfgang Grieskamp
  - **Operating Systems Group:**
    - Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, Steven Levi
  - **Security and Distributed Systems:**
    - Dan Simon, Brian Zill
  - **Software Design and Implementation Group:**
    - John DeTreville, Ben Zorn
  - **Software Improvement Group:**
    - Manuel Fahndrich, James Larus, Sriram Rajamani, Jakob Rehof

- **MSR Silicon Valley**
  - Martin Abadi, Andrew Birrell, Ulfar Erlingsson, Roy Levin, Nick Murphy, Ted Wobber