Dynamic Emulation and Fault-Injection using Dyninst

Presented by:
Rean Griffith
Programming Systems Lab (PSL)
rg2023@cs.columbia.edu

Overview

- Introduction
- Background
- Dynamic Emulation Example
- Solution Requirements
- Dyninst Modifications Necessary
- On-going Fault-injection Tool Development
- Conclusions

Introduction

- We are working on the design and evaluation of self-healing systems.
- Based on two techniques
  - Runtime-adaptations (technical)
  - Mathematical models of failures & recovery (analytical)

Role of Runtime-Adaptations

- Fault-Detection
  - Transparently adding/modifying detection mechanisms
  - Replacing/removing under-performing mechanisms
- Failure-Diagnosis
  - In-situ diagnosis of systems (drill-down)
  - In-vivo testing (ghost transactions)
- **System-Repairs**
  - **Dynamic fine-grained or coarse-grained repairs**

Dynamic Emulation Example

- Proof-of-concept dynamic emulation support for applications using Kheiron/C (mutator)
  - Allows select portions of an application to run on an x86 emulator rather than on the raw CPU
  - Security-oriented self-healing mechanism
- Allows users to:
  - Limit the impact of un-patched vulnerabilities
  - Test/verify interim (auto-generated) patches
  - Manage the performance impact of whole-program emulation

Background on the x86 Emulator

- Selective Transaction Emulator (STEM)
  - An x86 instruction-level emulator developed by Michael Locasto, Stelios Sidiroglou-Douskos, Stephen Boyd and Prof. Angelos Keromytis
  - Developed as a recovery mechanism for illegal memory references, division by zero exceptions and buffer overflow attacks

Big Picture Idea for STEM

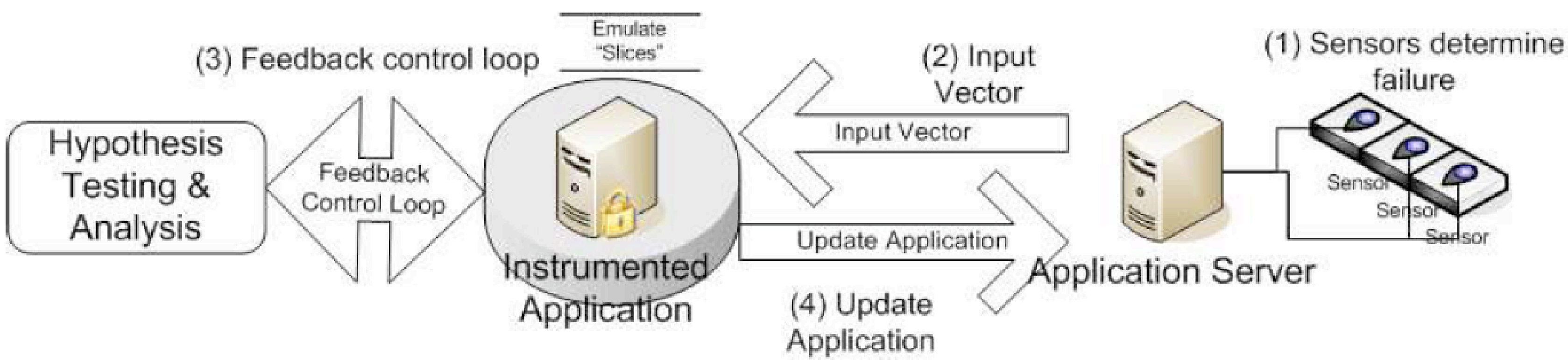


Figure 1: **Feedback control loop: (1) a variety of sensors monitor the application for known types (but unknown instances) of faults; (2) upon recognizing a fault, we emulate the region of code where the fault occurred and test with the inputs seen before the fault occurred; (3) by varying the scope of emulation, we can determine the "narrowest" code slice we can emulate and still detect and recover from the fault; (4) we then update the production version of the server.**

Building a Reactive Immune System for Software Systems,
Stelios Sidiroglou Michael E. Locasto Stephen W. Boyd Angelos D. Keromytis
USENIX 2005

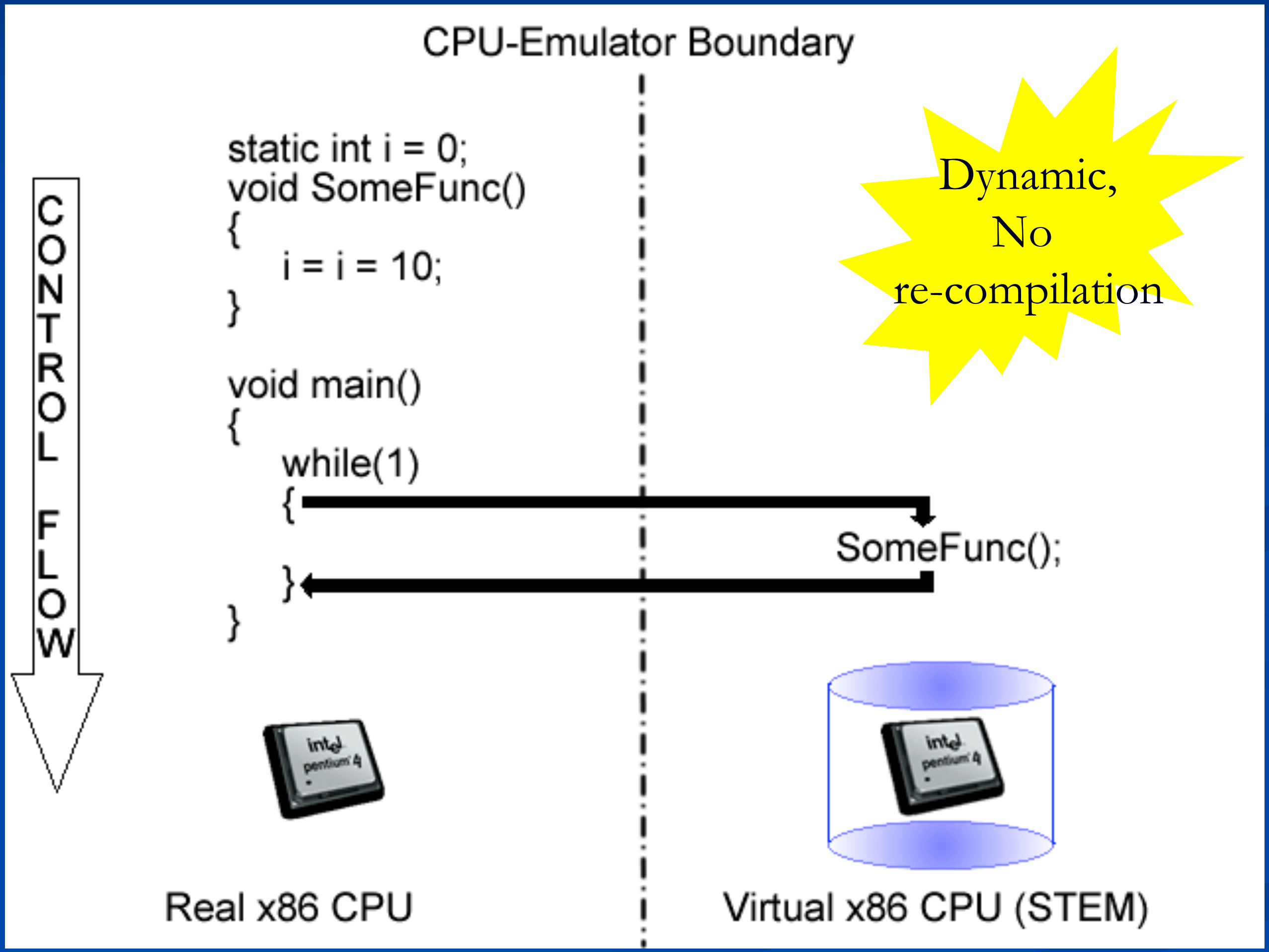
Limitations of the Original STEM

- Inserted via source-code
  - Manual identification of locations to emulate
  - Re-compilation and (static) re-linking needed to emulate different sections of an application

```
void foo()
{
    int i = 0;
    // Macro: saves gp registers
    emulate_init();
    // begin emulation function call
    emulate_begin();
    i = i + 10;
    // end emulation function call
    emulate_end();
    // Macro: commits/restores gp registers
    emulate_term();
}
```

- Minimum observed runtime over-head of 30%.

Proposed Solution



Solution Requirements

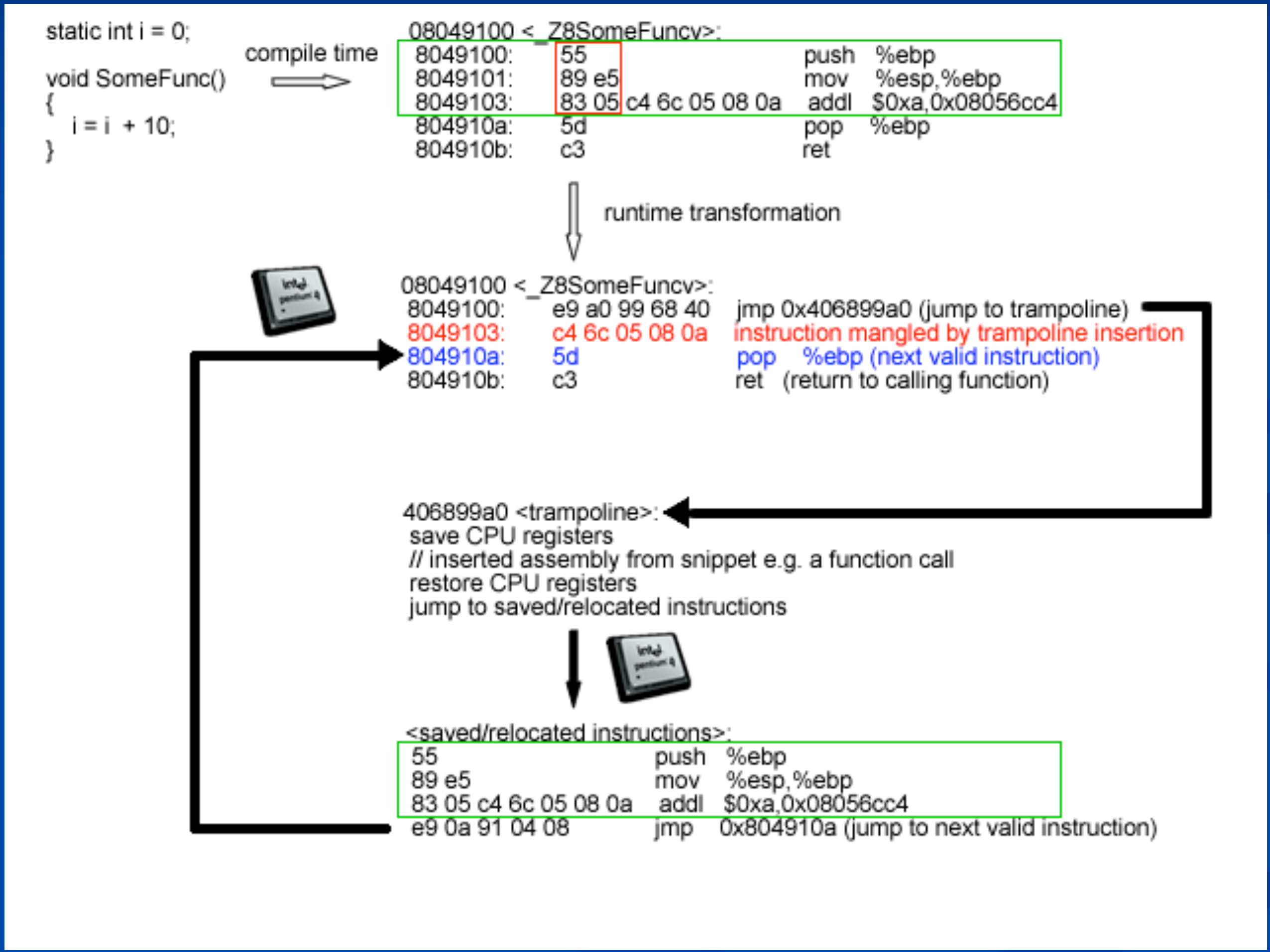
- Dynamic Loading of the STEM x86 Emulator.
- Clean CPU-to-Emulator handoff
  - Correct Emulator initialization
  - Correct Emulator execution
- Clean Emulator-to-CPU handoff
  - Correct Emulator unload

Requirements Met Out-of-the-Box by Dyninst 5.0.1

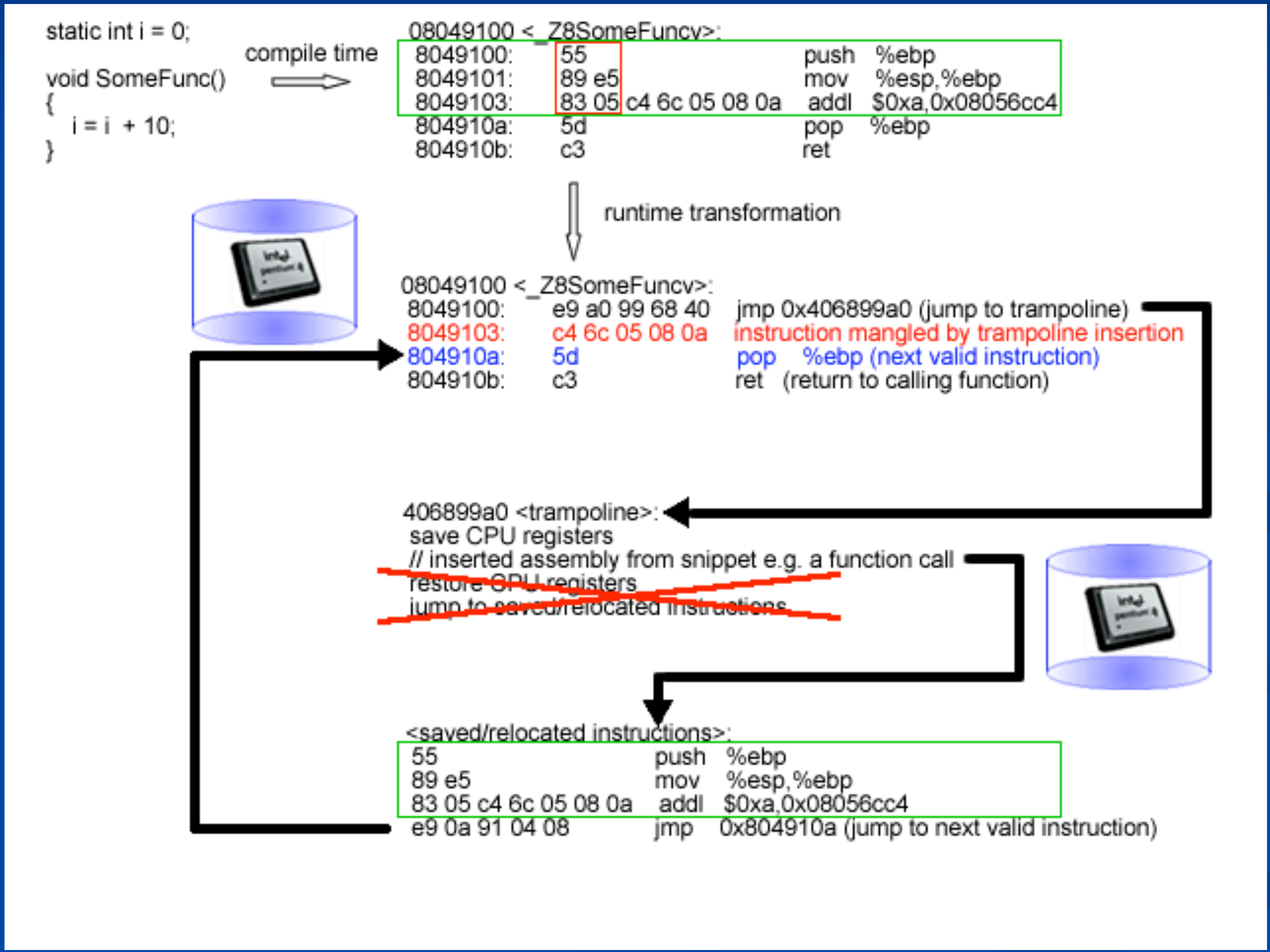
- Dynamic Loading of the STEM x86 Emulator.
- ~~Clean CPU-to-Emulator handoff~~
  - ~~Correct Emulator initialization~~
  - ~~Correct Emulator execution~~
- ~~Clean Emulator-to-CPU handoff~~
  - ~~Correct Emulator unload~~

But…with a few simple modifications to Dyninst, we are able to satisfy all these requirements.

Unmodified Dyninst Operation



Dynamic STEM Operation



Correct Emulator Initialization – Dyninst Modifications

- Emitter32::emitBTSaves modifications
  - Save CPU state before instrumentation on the real CPU stack AND at a location in the target program address space (Register storage area address)
  - Save the instructions mangled by inserting the trampoline at a KNOWN location in the target program address space (Code storage area address)
- instPoint, BPatch_point modifications
  - Added extra fields and methods to the type definitions to set/get the extra information

Dynamic Emulation Mutator Snippet

```
BPatch_point* pt = NULL;
...

pt = (*points)[0];

// Create data type
regStorageAreaType = bpatch.createScalar( "storageArea", sizeof(regData) );

// Allocate space for data type instance
regStorageAreaVar = process->malloc( *regStorageAreaType );

// Set the address of the register storage area on the instrumentation point
pt->setRegisterStorageAddress( (unsigned int) regStorageAreaVar->getBaseAddr() );

pt->setNumInstructions( pt->getNumDisplacedInstructions() );
pt->setBytesToSave( pt->getSizeofDisplacedInstructions() );
pt->setFunctionBaseAddress( (unsigned int) targetFunc->getBaseAddr() );

// Allocate space to save the displaced instructions
codeStorageAreaType = bpatch.createScalar( "codeArea", pt->getBytesToSave() );
codeStorageAreaVar = process->malloc( *codeStorageAreaType );

// Set the address of the code storage area on the instrumentation point
pt->setCodeStorageAddress( (unsigned int) codeStorageAreaVar->getBaseAddr() );
```

Correct Emulator Execution

- Register storage area address used to initialize STEM's registers
- Code storage area address used to prime STEM's execution pipeline
- STEM tracks its current stack depth
  - Initially set to 0
  - Call and Return instructions modify the stack depth
  - A return instruction at depth 0 signals the end of emulation

Correct Emulator Unload

- Cleanup
  - Copy emulator registers to real CPU registers
  - Push the saved_eip onto the real CPU stack
  - Make it the return address for the current stack frame – pop it into 4(%ebp)
  - Push the saved_ebp onto the real cpu stack
  - Restore that value into the real EBP register

Current Status

- Doesn't crash on our simple test programs.
- Correct computation results for these programs.
- Multiple emulator entries/exits e.g. in a loop.
- More refinements to x86 emulator needed to support more complicated programs
  - Emulator-state rollbacks in the works
  - Clean up the CPU-to-Emulator and Emulator-to-CPU handoffs

Role of Runtime-Adaptations

- Fault-Detection
  - Transparently adding/modifying detection mechanisms
  - Replacing/removing under-performing mechanisms
- Failure-Diagnosis
  - In-situ diagnosis of systems (drill-down)
  - In-vivo testing (ghost transactions)
- System-Repairs
  - Dynamic fine-grained or coarse-grained repairs
- **Fault-Injection**
  - **Exercise the detection, diagnosis and repair mechanisms so we can perform a quantitative evaluation**

Fault-Injection Tool Development

- Kheiron/CLR and Kheiron/JVM
  - Fault-injection tools for .NET applications and JVM applications/application-servers based runtime adaptations (bytecode-rewriting)
- Kheiron/C extensions
  - Dynamic fault-injection tool for databases using Dyninst. Specifically targeting the query (re)-planning and processing sub-systems of the database
- Device driver fault-injection tools for Linux 2.4, Linux 2.6, Windows 2003 Server and Solaris 10
  - Evaluating device-driver recovery frameworks e.g. Nooks and Solaris 10 Fault Isolation Services

Conclusions

- We have described and implemented an example of dynamically inserting and removing a recovery mechanism based on selective emulation.
- More work needs to be done to polish our prototype and experimentally evaluate the efficacy of this recovery mechanism.

Acknowledgements

- This work was conducted under the supervision of Prof. Gail Kaiser and with the help of Stelios Sidiroglou
  - We would like to thank Matthew Legendre, Drew Bernat and the Dyninst Team for their assistance/guidance as we worked with Dyninst 4.2.1 and Dyninst 5.0.1 to develop our dynamic emulation techniques.

Thank You

Questions, Comments Queries?

For more details please contact:

Rean Griffith

Programming Systems Lab Columbia University

rg2023@cs.columbia.edu