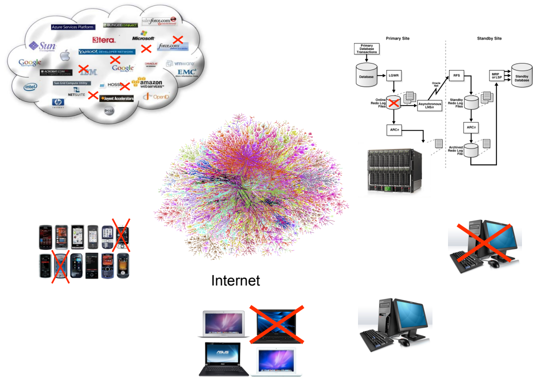
What To Do When Things Go Wrong:
Recovery in Complex (Computer) Systems

Martin Rinard
MIT EECS, MIT CSAIL
Massachusetts Institute of Technology
Cambridge, MA 02139



Fault Tolerance and Recovery

- Where are we today?
- Where can we go from here?
- What role does AOP have to play?

Hardware Fault Tolerance

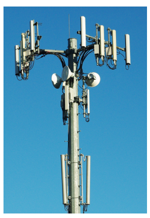
- Communication
- Storage
- Computation

Communication

11010011101100

11010011101100

Communication



11010011101100

110**0**0011101100

Communication

11010011101100

110**0**0011101100

- **First Issue**: recognize error

Communication

11010011101100 **100**

110**0**0011101100 **100**

- **First Issue**: recognize error
- **Solution**: redundancy (checksum)

Communication

11010011101100 **100**

110**0**0011101100 **100**

- **Second issue**: get right bits

Communication

11010011101100 **100**

~~110**0**0011101100 **100**~~

- **Second issue**: get right bits
- **Two solutions**:
  - Discard and Retransmit (backward error correction)

Communication

11010011101100 **100**

11010011101100 **100**

- **Second issue**: get right bits
- **Two solutions**:
  - Discard and Retransmit (backward error correction)

Communication

11010011101100 **100**

110**0**0011101100 **100**

- **Second issue**: get right bits
- **Two solutions**:
  - Discard and Retransmit (backward error correction)
  - Error correcting code (forward error correction)

Communication

11010011101100 **100**

11010011101100 **100**

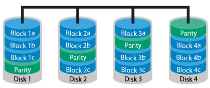
- **Second issue**: get right bits
- **Two solutions**:
  - Discard and Retransmit (backward error correction)
  - Error correcting code (forward error correction)

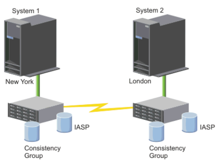
General Patterns

- Redundancy
- Error detection
- Two kinds of error correction
  - Forward error correction
  - Backward error correction (retry)
- Retries exploit **nondeterminism**

Storage

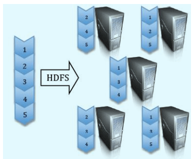
RAID





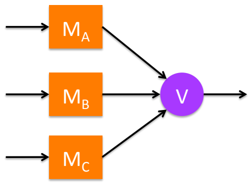
Remote Mirroring

File Replication/Distribution



Importance of Repair
Without Repair, MTTF < 2X
With Repair, MTTF > 10^4X
(Triple Redundancy)

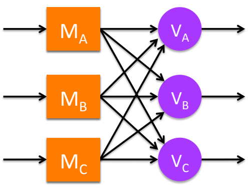
Computation



Triple Redundancy

Key Assumption: Independent Faults

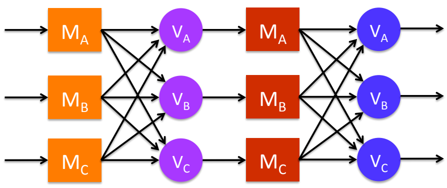
Computation



Triple Modular Redundancy

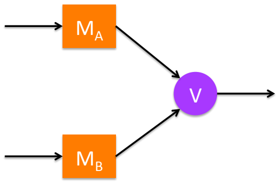
Key Assumption: Independent Faults

Computation



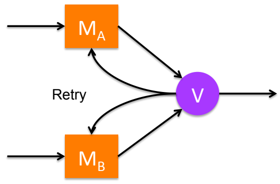
Triple Modular Redundancy

Computation



Dual Redundancy

Computation



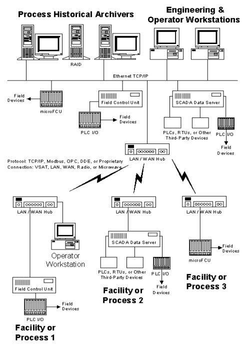
Dual Redundancy With Retry

Soft vs. Hard Errors

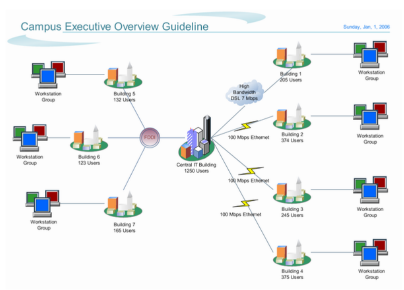
Containing Faults

- Modularity, Isolation
- Componentize the design
- Isolate components behind narrow, strictly checked interfaces
- If components fail, others keep going

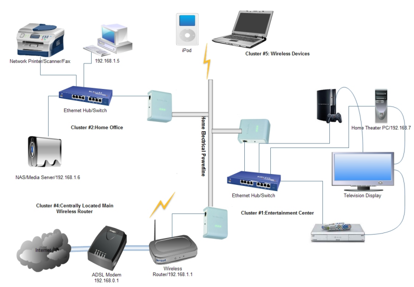
Modularity



Modularity



Modularity



Key Concepts in HW Fault Tolerance

- Redundancy
  - Spatial redundancy: checksums, parity, replication
  - Temporal redundancy: retry with nondeterminism
- Backward vs. Forward Error Correction
- Soft vs. Hard Errors
- Modularity, Isolation, Repair
- Goal of Perfection

Hardware Fault Tolerance: Current Status

Interesting Issues/Principles

Lots of Good Research

Largely Solved Problem

Hardware Fault Tolerance: Future

- Engineers will start to trade off correctness for
  - Performance
  - Reduced energy consumption
- Life will get interesting again
- Software will be exposed to hardware faults…

From Hardware to Software

- Many concepts transfer/generalize
- Important differences
  - Specification often not available for software
  - Complexity pushed onto software
  - Ease of working with technology
  - Application diversity, scale, and number
  - Failures typically caused by defects in software (not intermittent natural phenomena)
- Different tradeoffs
  - Correctness vs. functionality
  - Update/new release cost/frequency

Software Fault Tolerance

Conceptual Framework

- Errors (mistakes in thinking)
- Defects (manifestation of errors in code)
- Faults (activation/execution of defect)
- Failures (system fails to meet expectations)

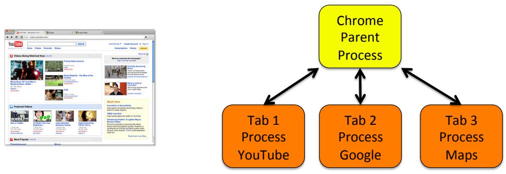
Software Fault Tolerance Classical Techniques

- Modularity
  - Processes
  - Virtual Machines
- Redundancy
  - N-Version Programming
  - Recovery Blocks
- Transactions
- Undo, Redo
- Reboot, Retry

Goal

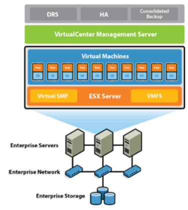
Provide abstraction of perfection

Processes + Messages



- Processes give modularity and isolation
- Messages support controlled interactions

Virtual Machines

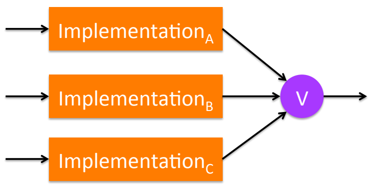


Modularity and Isolation

Redundancy

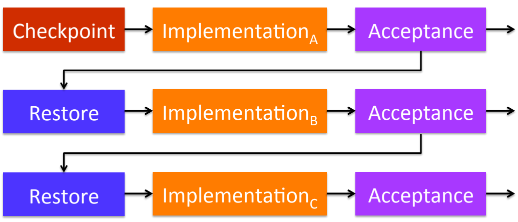
N-Version Programming

(Chen, Avizienis FTCS 1978)

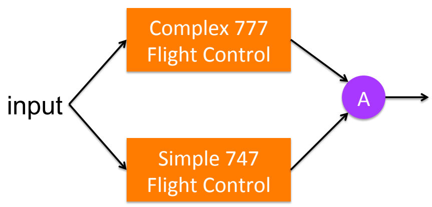


Recovery Blocks

(Horning et. al. LCS 1974)



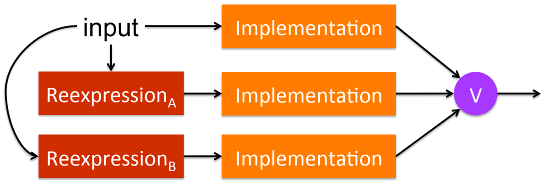
Prioritized Versions



If 777 Flight Control in 747 Envelope
Use 777 Flight Control Output
Else Use 747 Flight Control Output

Data Diversity and N-Copy Programming

(Amman and Knight, FTCS 1987)



Examples of Reexpression

- sin(x) = sin(a)sin(π/2-b) + sin(π/2-a)sin(b) choose different a,b such that x = a+b
- Reorder events for an event-processing system
- Perturb real-valued inputs by small amount
- Apply an equivalence-preserving program transformation

N-Version Programing Issues

- Correlated faults (Knight, Leveson IEEE TSE 1986)
  - Specification interpretation
  - Similar implementation choices/faults
  - Specification errors
- Duplicated implementation effort
  - Must implement multiple versions
  - Must come up with multiple ways to solve problem

Modern N-Version Programming

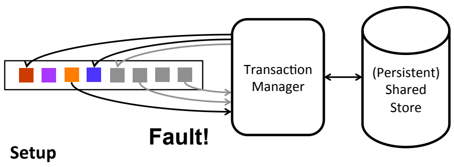
- Multiple implementations of applications
  - PDF, PNG, JPEG, WAV viewers
  - Web browsers, text editors, compilers
  - OpenOffice, Office for PC, Office for Mac
- If have a problem with one, use another!
- Worked for me preparing this talk
  - Could not print from Chrome
  - Could print from Preview

When Does Modern N-Version Programming Work Best?

- No shared specification
- No shared implementation
- No interaction between development teams

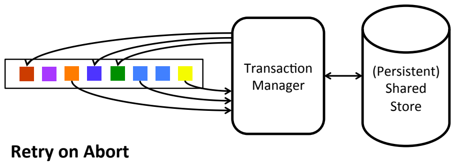
- In practice, can usually tolerate some amount of sharing/interaction
  - libc, math libraries
  - Common data format description documents

Transactions



- **Setup**
  - Sequence of operations
  - Fault causes early termination
  - Leaves store in inconsistent state
- **Solution**
  - Developer identifies transaction boundaries
  - System undoes effect of operations

Transactions



- **Retry on Abort**
  - Try transaction again
  - Most of the time it works (!!!)
- **Similar to**
  - Retransmission for corrupted network packets
  - Retry for soft hardware errors

Why Does Retry Work?

- Transaction behavior depends on two things:
  - Internal actions (deterministic)
  - External interactions with environment (nondeterministic)
    - Underlying system state
    - Parallel transactions
- Testing is very effective at identifying faults
  - In internal actions
  - Common execution environments

Why Does Retry Work?

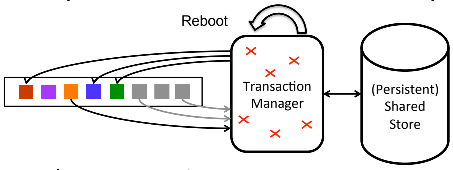
- Most faults caused by interactions with rare transient aspects of environment
- When retry, transient aspects are gone
- So back to common case and retry succeeds

Steer Retry Away from Fault

- **Dimmunix** (Jula et. al. OSDI 2008)
  Observe and avoid deadlock patterns
- **Exterminator** (Novark et. al. PLDI 2007)
  Find buffers that are too small and extend them
- **Rx** (Qin et. al. SOSP 2005)
  Rollback and execute in modified environment (memory management, timing, drop requests)
- These systems share common philosophy
  - Many possible executions, only some are fault-free
  - Find and execute one that is fault-free
  - Do not attempt to change set of executions

Transaction Complications

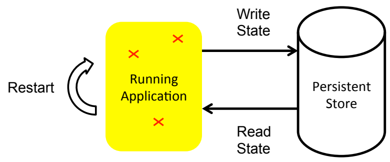
Complication One: State Decay



- State decays over time
- Decayed state causes retries to ALWAYS abort
- **Reboot** restores pristine common state
- So retries succeed, transaction commits

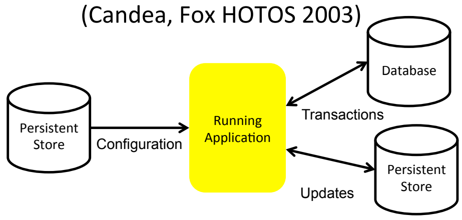
Software Rejuvenation

(Huang et. al. FTCS 1995)



Crash-Only Software

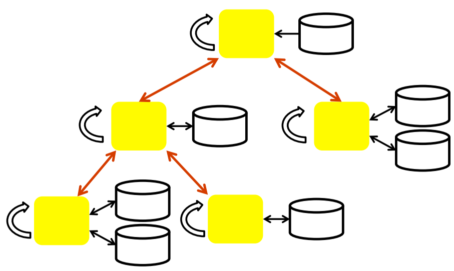
(Candea, Fox HOTOS 2003)



- ALL necessary application state stored externally in persistent storage
- Can crash and restart application AT ANY TIME

Recursive Restart

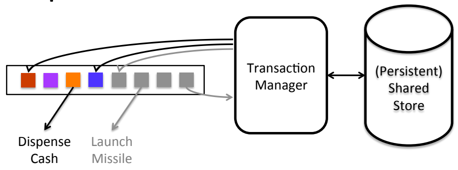
(Candea, Fox HOTOS 2001)



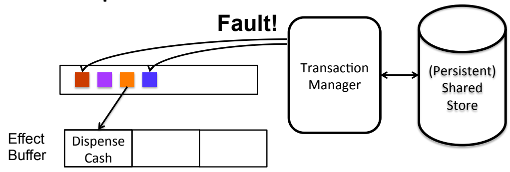
Key Insights

- All computations age - anticipate and correct problems before something goes wrong
- Abstraction barriers promote consistent data
  - Narrower, cleaner, safer interface to data
    - Session state managers, SQL
    - Save/restore procedures
  - Think more about how data stored and accessed
  - You want it to be difficult to access persistent data!
- Potential reason persistent objects not popular

Complication Two: External Effects

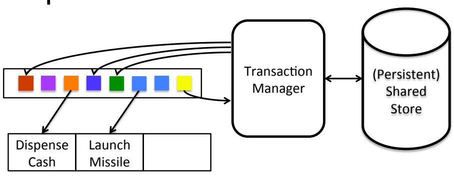


Complication Two: External Effects



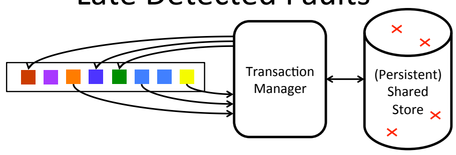
- Store external effects in buffer during transaction execution
- Clear effect buffer on abort

Complication Two: External Effects



- Store external effects in buffer during transaction execution
- Execute effects in buffer at transaction commit point
- Include confirmation checks, retry to ensure completion
- External **compensation** if can't complete effects

Complication Three: Late Detected Faults

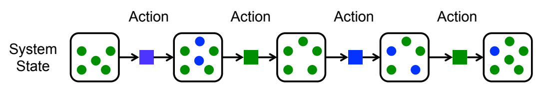


- **Problem**: transaction commits, but corrupts persistent state
- System runs for a while
- **Audit** (or external mechanism) detects corruption

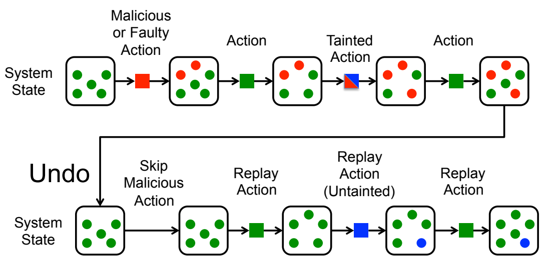
Dealing With Late Detected Faults

- Two Alternatives
- **Repair procedure** eliminates corruption (forward error correction)
- **Undo/Redo** (backward error correction)
  - **Undo** transactions until system is consistent
  - **Redo** transactions to restore system state
  - **Skip** bad transactions (if you can identify them)

Undo/Redo For Complete System



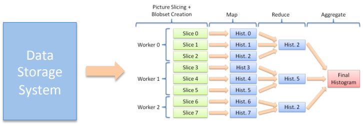
Undo/Redo For Complete System



Undo/Redo Systems

- Undoable Email (Brown, Patterson, Usenix ATC 2003)
- Taser (Goel et. al. SOSP 2005)
- RETRO (Kim et. al. OSDI 2010)
- Issues
  - Determining malicious/faulting actions
  - Accurately tracking effects (false negatives/positives)
  - Dealing with external effects
  - Redoing desirable operations in new changed state
- Complex systems programming techniques required

Special Case: Read-Only Systems



- Read-only = lightweight transactions for free
  - No need for transaction mechanism
  - No need for undo/redo
  - Can rerun/restart at any time
- Very appealing model of computation

Where Are We Today?

- Fault tolerance/recovery enormous success
- Mainstay of modern (very successful) computing and communication infrastructure

- But people still complain…
  - Systems crash, hang, misbehave
  - Security vulnerabilities (snake in computing garden)

How Do We Make Progress?

Standard Answer:
Better Engineering!

But Modern Systems Are Very Complex

You can't understand well enough to engineer…

Even if you can, not cost effective…

How Do We Make Progress?

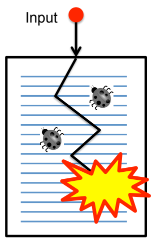
Better Answer: Change Our Perspective

What Does This Mean?

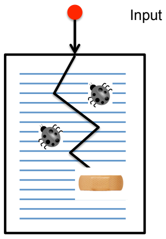
- Operate with (at most) only a partial understanding of what is going on
- Try to make things better (but not perfect)

- Techniques
  - Automatic (potentially unsound) bug fixing
  - Eliminating software fatalities
  - Performance-enhancing techniques

Automatic Bug Fixing



Automatic Bug Fixing



Goal

Automatically generate a patch that fixes the bug

Use the input to focus the patch generation and test

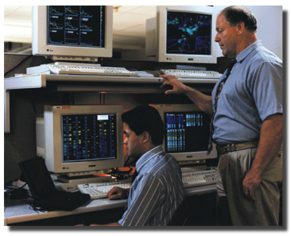
Data Structure Repair

- Basic Approach
  - Obtain data structure consistency properties
  - **Specified** (by developer)
    (Demsky et. al. OOPSLA 2003, Elklarabeih et. al. ASE 2007)
  - **Learned** (Demsky et. Al. ISSTA 2005)
- Run data structure consistency checks
  - When encounter fault
  - Before/after data structure operations
- If consistency violated, enforce invariants

What Guarantees Do You Get?

- Completely correct data structure?
  - Typically not
  - May have destroyed required information
- Consistent data structure
  - Heuristically close to correct data structure
  - Enough to keep application going

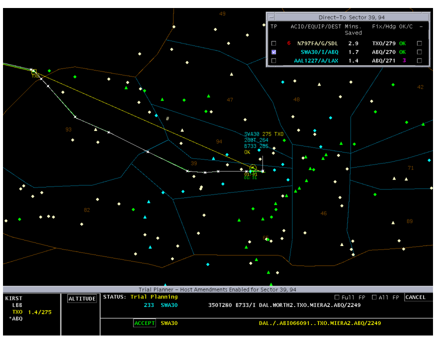
Data Structure Repair for CTAS (Air Traffic Control Software)



TMA at Fort Worth Center

FAST at DFW TRACON

CTAS Screen Shot



CTAS Bug and Repair

- Fault
  - Bug in flight plan processing (reintroduced from old version)
  - Produces bad airport index in flight plan data structure
- Workload – recorded radar feed from DFW
- Without repair
  - System crashes – segmentation fault
  - Reboot does not help – CTAS rereads flight plan, crashes
- With repair
  - Aircraft has different origin or destination
  - System continues to execute
  - Anomaly eventually flushed from system

Aspects of CTAS

- Lots of independent subcomputations
  - System processes hundreds of aircraft – problem with one should not affect others
  - Multipurpose system (visualization, arrival planning, shortcuts, …) – problem in one purpose should not affect others
- Sliding time window: anomalies eventually flushed
- Huge certification cost makes bug fixes problematic

Survival of (minor) component may enable system as a whole to survive

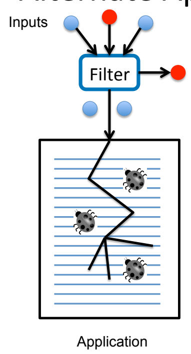
More Bug Fixing Techniques

- **ClearView** (Perkins et. al. SOSP '09)
  - Learn invariants about data that bug manipulates
  - Enforce invariants using variety of strategies
  - Choose one that works best
- **Genetic Programming** (Weimer et. al. ICSE '09)
  - Randomly generate variants around bug
  - Run generated variants on test suite
  - Choose one that works for test suite
- **DYBOC** (Sidiroglou et. al. ISC 2005)
  - Monitor function execution for faults
  - Transactionally terminate, return error code

Even More Bug Fixing Techniques

- Use specifications
- Enforce postconditions on method exit
  - Falling Back on Executable Specifications (Samimi et. al. ECOOP 2010)
  - Contract-Based Data Structure Repair Using Alloy (Zaeem et. al. ECOOP 2010)
  - Automated Fixing of Programs with Contracts (Wei et. al. ISSTA 2010)
- Can hope for completely correct patch (but you need specifications)

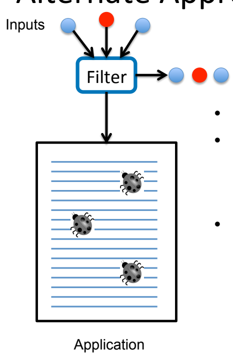
Alternate Approach: Bug Avoidance



Goal

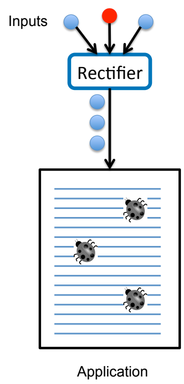
- Filter out inputs that may trigger bug
- Typical approach: anomaly detection
  - Learn constraints for typical inputs
  - Filter out inputs that are not typical

Alternate Approach: Bug Avoidance



Goal

- Filter out inputs that may trigger bug
- Typical approach: anomaly detection
  - Learn constraints for typical inputs
  - Filter out inputs that are not typical
- **Problem**: May filter out good inputs…



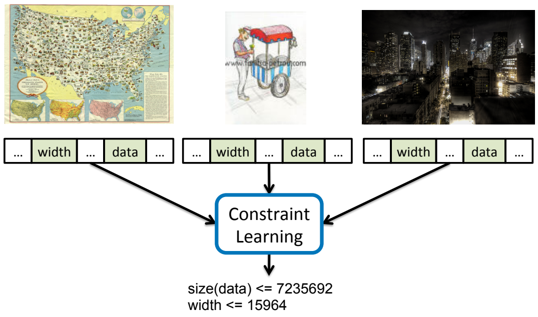
Input Rectification

(Long et. al. ICSE 2012)

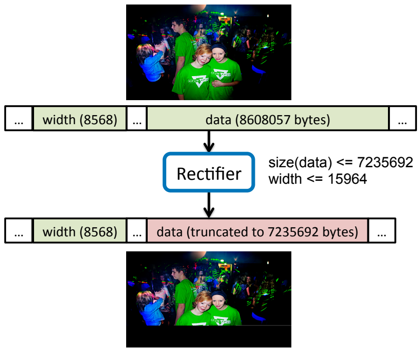
Goal

- Make ALL inputs safe to process
- Approach: Input rectification
  - Learn constraints for typical inputs
  - Enforce constraints to make ALL inputs typical

Learning



Rectification



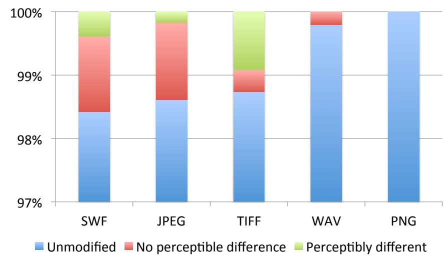
Rectification Questions

- Does it nullify defects/security vulnerabilities?
- Yes

Swfdec 0.5.5	(SWF shockwave player)
Dillo 2.1	(PNG lightweight web browser)
ImageMagick 6.5.2-8	(JPEG, TIFF image processing)
Google Picasa 3.5	(JPEG, TIFF photo management)
VLC 0.8.6h	(WAV media player)

- How much data loss is there?

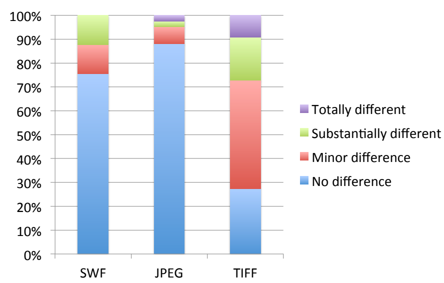
Question: How many safe files does rectifier leave intact? Answer: Between 98%-100%



Question: How much desirable data does rectifier preserve in modified files?

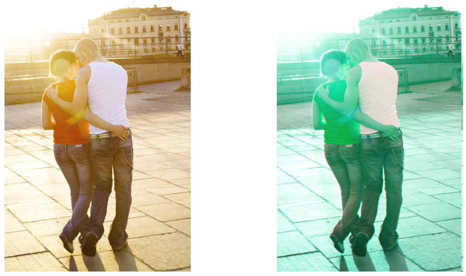
- Started with files that rectifier modified
- Mechanical Turk workers rate difference
- Workers classified files into four categories
  - No difference
  - Minor difference
  - Substantially different
  - Totally different

Mechanical Turk Classification Results (for modified files)

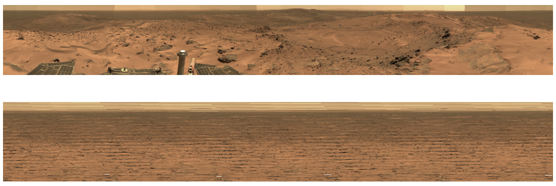


Substantially different

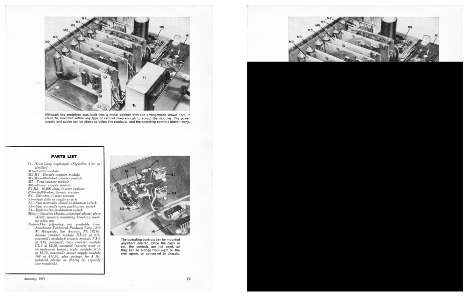
Minor difference



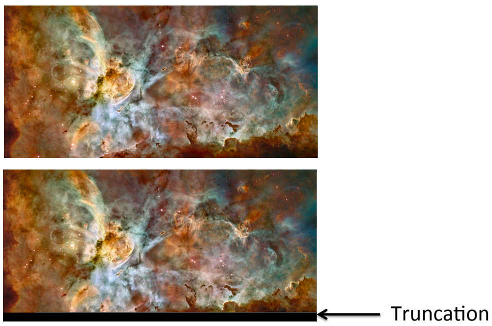
Substantially different



Substantially different



Minor difference



Minor difference

Why?

- Rectifier often modifies fields that do not affect visible data (metadata fields)
- Rectifier attempts to minimize changes (so it preserves much of useful data)

Eliminating Acute Software Fatalities

- Identify all possible fatal events
- Eliminate them
  - Memory leaks
  - Addressing errors (null references, out of bounds accesses)
  - Infinite loops
- Goal is meaningful survival, not perfection

Eliminating Fatal Memory Leaks

Original Code

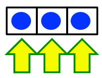
Memory Leak ↓

p = new Node();

Cyclic Allocation

Node **buffer**[B];

static int c = 0;



p = **buffer**[c++ % B];

Standard Response

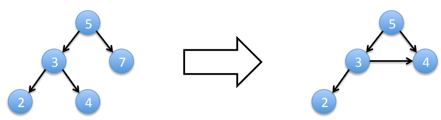
You can't do this – you might overlay live data!

↑ False ↑ True

What Happens In Practice?

- Used this technique on several programs with memory leaks [Nguyen and Rinard, ISMM 2007]
  - Squid – web proxy cache
  - Xinetd – manages connections, requests
  - Freeciv – interactive multiple player game
  - Pine – mail client
- Eliminated memory leaks
- When forced overlay of live data, programs degrade gracefully

Why?

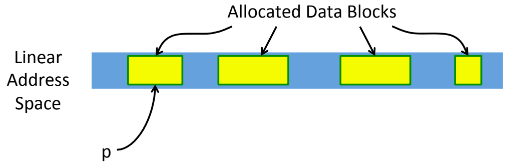


- Is data structure consistent? NO
- Consistent enough to use? YES
- Right answer some of the time? YES
- Does program survive? YES
- Replaced fatality with graceful degradation

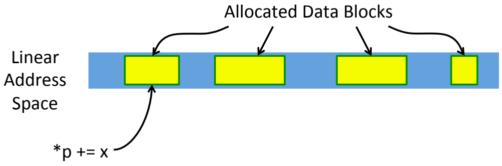
Eliminating Fatal Addressing Errors

Out of Bounds Errors
Null Pointer Dereferences

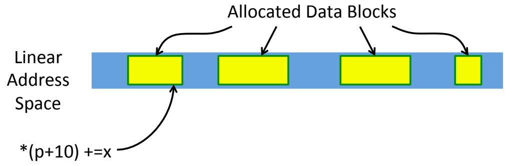
Standard C Programming Model



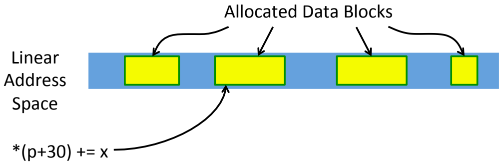
Standard C Programming Model



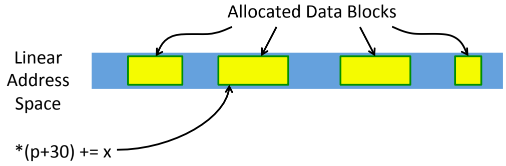
Standard C Programming Model



Standard C Programming Model



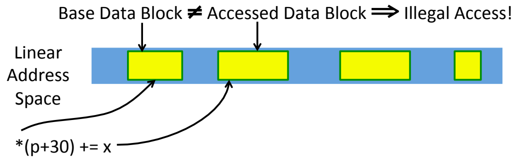
Standard C Programming Model



Bounds Violation!

Data corruption…
Segmentation violation…
Security vulnerability…

Bounds Checked C Programming Model



Track base data block for each pointer
Dynamically check that each access falls within the bounds of the base data block
If not, access is illegal

Jones&Kelly IWAD 1997, Ruwase&Lam NDSSS 2004

Traditional Bounds Check Philosophy

- Bounds violation (illegal access) is irrefutable evidence of a fault in the program
- Unsafe to continue because program is outside its anticipated execution envelope

Our Philosophy

- Programs are complex systems
- Should tolerate localized memory errors
  - Perform dynamic bounds checks
  - Discard out of bounds writes
  - **Manufacture values for out of bounds reads**
  - Continue to execute along normal path
- Called ***failure-oblivious computing***

Consequences of Failure-Oblivious Computing

- No corruption of other data blocks
- No segmentation violation
- No abnormal termination
- No addressing exceptions
- No security vulnerabilities (from out of bounds writes)

Consequences of Failure-Oblivious Computing

- No corruption of other data blocks
- No segmentation violation
- No abnormal termination
- No addressing exceptions
- No security vulnerabilities (from out of bounds writes)

But what about errors in continued execution?

Experiment

- Implemented compiler that generates failure-oblivious code
- Acquired programs (servers)
  - Pine, Mutt (mail user agent)
  - Apache (web server)
  - Sendmail (mail transfer agent)
  - Midnight Commander (file manager)
- Found bounds violation errors
  - Potential security vulnerabilities
  - Vulnerability-tracking web sites

Experiment

- Generated three versions of each program
  - SC – standard compilation
  - BC – bounds check compilation
    (terminates program on bounds violations)
  - FO – failure-oblivious compilation
    (continues through bounds violations)
- Ran each version on workload containing inputs that attempted to exploit vulnerability

Results

	Secure?			Initializes?			Continues Correctly?		
	SC	BC	FO	SC	BC	FO	SC	BC	FO
Pine	No	Yes	Yes	Maybe	Maybe	Yes	No	No	Yes
Mutt	No	Yes	Yes	Maybe	Maybe	Yes	No	No	Yes
Sendmail	No	Yes	Yes	Yes	No	Yes	No	Not Applicable	Yes
Apache	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Midnight	No	Yes	Yes	Yes	Maybe	Yes	No	No	Yes

✗ No ◇ Maybe
● Yes — Not Applicable

Why?

- Servers have short error propagation distances
  - Localized errors in one request
  - Tend not to propagate to next request
  - Inherently have good modularity
- Effect of failure-oblivious computing
  - Discarding out of bounds writes eliminates global data structure corruption
  - Keeps errors localized
  - Server survives to process subsequent requests
- Subsequent requests serviced without errors

Eliminating Infinite Loops

Jolt

(Carbin et. al. ECOOP 2011)

1. Execute program
2. Program becomes unresponsive
3. Launch Jolt
   – Bolt takes snapshots after each loop iteration
   – If two snapshots are same, infinite loop!
4. Jolt jumps to instruction after loop

5 Applications and 8 Infinite Loops

1. **ctags** : line numbers of functions in code.
   – v5.5 : one loop in fortran module.
   – v5.7b : one loop in python module.
2. **grep** (v2.5): matches regexp against files (3 loops).
3. **ping** (v20100214): icmp utility.
4. **indent** (v1.1-svr 4): indents source code.
5. **look** (v1.9.1): matches a word against dictionary file.

Question #1

Can Jolt detect infinite loops with this simple strategy?

Benchmark	Detected
ctags-f	Yes
ctags-p	Yes
grep	Yes
ping	Yes
look	Yes
indent	No

7 of 8

Question #2

Does Jolt produce a safe execution?

- Methodology
  - Validated execution with Valgrind and by hand.
  - Tested with available loop triggering inputs.
- Results
  - Yes, side effects often localized = consistent state.
  - Or, simple correctness invariants.

Question #3

Does Jolt produce a better output than Ctrl-C?

- Methodology
  - Defined output abstraction, and compared outputs.
- Results
  - Yes, errors often isolated to single output unit (e.g., file).
- Example
  - **indent:** correct indention resumes on next file.
  - **Terminating indent deletes your source code**

Question #4

Does Jolt match the developers' fix?

- Methodology
  - Manually inspected a later version of each application
- Results
  - **ctags**: no, output semantically different on some inputs
  - **grep**: jolt matches fix for two of three loops
  - **ping, indent, look**: yes, in all cases
- Example
  - **ping**: developer used **continue** instead of **break**

Observations

- Infinite loops can (and often do) frustrate users
- Infinite loops can be (and often are) simple
- Jolt enables application to produce results that can be (and often are) better than no results at all
- Jolt can (and often does) model the developer's fix

Performance-Enhancing Techniques for Software

How to Make Your Software Faster or Consume Less Energy

- Profile program
- Find loops that take most time
- Perforate the loops
  - Don't execute all loop iterations
  - Instead, skip some iterations

```
for (i = 0; i < n; i++) { … }
```

How to Make Your Software Faster or Consume Less Energy

- Profile program
- Find loops that take most time
- Perforate the loops
  - Don't execute all loop iterations
  - Instead, skip some iterations

```
for (i = 0; i < n; i++) { … }
```

⇩

```
for (i = 0; i < n; i += 2) { … }
```

How to Make Your Software Faster or Consume Less Energy

- Profile program
- Find loops that take most time
- Perforate the loops
  - Don't execute all loop iterations
  - Instead, skip some iterations
- Result
  - Program consumes fewer computational resources
  - Runs faster (or takes less energy) (or both)

Common Reaction

- OK, I agree program should run fast
- But you can't do this because you'll get the wrong result!

Our Response

- OK, I agree program should run fast
- ~~But you can't do this because you'll get the wrong result!~~
- You won't get the wrong result
- You'll get a **different** result

Not a Correctness Issue
Accuracy Issue

Exploring This Idea

(Sidiroglou et. al. FSE 2011)

- Acquire benchmarks
  - Programs
  - Inputs (training and production)
- Perform experiments
  - Apply loop perforation
  - Training runs
    - Distinguish **critical** and **perforatable** loops
    - Observe performance vs. accuracy trade off
  - Production runs on new (unseen) inputs

Parsec Benchmarks

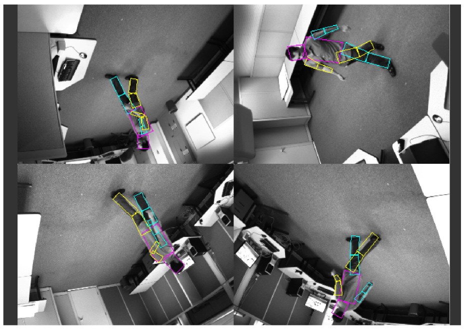
- x264 (H.264 video encoding)
- Bodytrack (human movement tracking)
- swaptions (swaption pricing)
- ferret (image search)
- canneal (digital circuit place and route)
- blackscholes (European option pricing)
- streamcluster (online point clustering)

All have some flexibility in output they produce

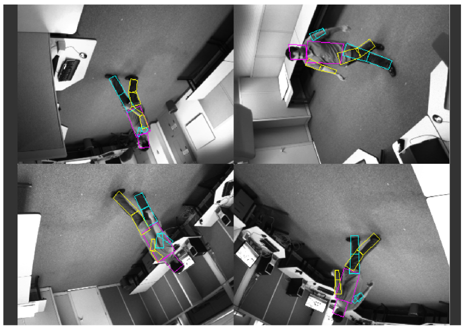
Summary of Results

- Loop perforation works
- Performance improvement
  - Typically over a factor of two
  - Up to a factor of seven
- Less than 10% change in output
- In effect, finding optimizable parts of program

Bodytrack, No Perforation



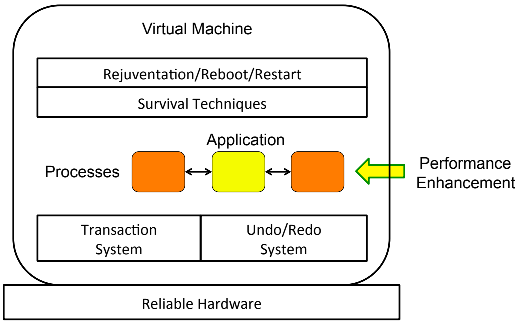
Bodytrack, With Perforation



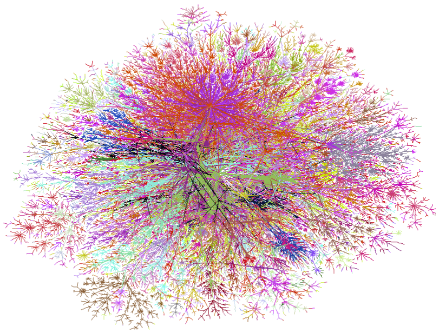
Why?

- Heuristic search guided by metrics
- Loop perforation gives new metric
  - More efficient (runs faster, consumes less energy)
  - Less accurate (but accurate enough)
- In bodytrack, metrics are error calculations
  - Between probabilistic model from previous frame
  - And image data from current frame
  - Used to obtain probabilistic model for current frame

Putting It All Together



Putting It All Together



Role of Aspect-Oriented Programming

- Current implementations
  - With compiler
  - With binary rewriting tool (Pin, DynamoRIO, …)
  - Inside operating system or transaction manager
- But implement what are essentially aspects
- Aspects should be able to help here

Role of Aspect-Oriented Programming

- Aspects provide metalevel
  - Take an existing system
  - Augment it with additional functionality
- Great for monitoring/modifying existing software
- Can make reliability/recoverability feasible/easy
- Binary AOP would be really useful

Key Techniques

- Classical techniques (perfection)
  - Processes, VMs (modularity, isolation)
  - Retry, Reboot (nondeterminism, aging)
  - Transactions (consistency in face of faults)
  - Undo/Redo (late detected failures)
- Modern techniques (survival, effectiveness)
  - Data structure repair (consistency, survival)
  - Fatality elimination (survival)
  - Performance enhancement (speed, efficiency)