Building Dependable Software

James Larus
Microsoft Research

ASPLOS XI
October 11, 2004
Computers (1994)

• Intel Pentium
  - 60-90 Mhz
  - superscalar (2 issue)
  - 8K I&D caches
  - 3.2 million transistor
  - $300-1000

• Sun SPARCStation 20
  - 50-75 Mhz
  - superscalar (3 issue)
  - 32K cache (off chip)
  - 3.1 million transistor
  - $10-20K
Operating systems (1994)
Programming languages (1994)
Subject: PizzaNet -- the killer app
From: David Farber
Date: Mon, 22 Aug 1994

FOR IMMEDIATE RELEASE

SCO AND PIZZA HUT ANNOUNCE PILOT PROGRAM FOR PIZZA DELIVERY ON THE INTERNET

"PizzaNet" Program Enables Computer Users to Electronically Order Deliveries
My family (1994)

- Don’t have a computer
- Don’t see why they would ever need one
10 years later

2004
Normal, unbelievable progress

• Moore’s Law continues
  – $500 buys $50,000 computer

• Software features increase at similar rate
  – free, instantaneous communication throughout world
  – more information than you want at your fingertips
  – digital media (photos, video, music) liberated from physical media

• Why aren’t customers happy?
My family (2004)

- Parents on second computer
  - email, web, word processing

- Scared to death they will break something
  - no idea of how it works or what goes wrong

- Installed and remotely maintained by my (sainted) brother
  - also scared to death that parents will break something
They are the norm

Expectations are different

• Most computer users are technically illiterate
  – and don’t plan to change
  – computer as appliance
    • device to get something done
    • not interesting in and of itself

• No support staff
  – muddle through as best they can

• Network opens computer to outside world
  – vandals, thieves, sleazy marketers, malware, viruses …
Software is weak link

- For most people, software
  - behaves unpredictably and inexplicably
  - fails often
  - feels insecure, corruptible
  - appears unreliable and shoddy

- Unhappy state of affairs
  - novelty is wearing off
  - day-to-day tool, not new toy
Microsoft Research and software tools

- Microsoft Research effort to improve software development in Microsoft
  - started 1998
- Premise: computer power can improve software development
  - tools **systematically** find software defects
- Aid (not supplant) developers & testers
  - manage details
  - find inconsistencies
  - ensure quality
  - goal is **not** perfection (verification)
- Success
  - tools used throughout Microsoft
  - 12% of defects in Server 2003 found by 2 MSR tools
  - tools will ship as products
What do they look for?

- Incorrect computation (≈ program verification)
  - not our goal!

- Abstraction-specific errors

- Semantic errors
  - races, deadlocks, ...

- API usage errors
  - close file twice, ignore error result, hold lock, ...

- Programming language misuse
  - uninitialized variable, null dereferences, ...
Static program analysis

- Verify property for all possible executions (and then some)
- Complements run-time checking
- Static analyses exist for only a few properties
- Choose at most one:
  - sound analysis finds all instances of error
  - complete analysis reports no false errors
- Can’t have both: Turing halting problem
- Don’t need either: heuristics
FILE* f;
if (complex_calc1())
  f = fopen(...);
...
if (complex_calc2())
fclose(f);
Improving software

- **1\textsuperscript{st} generation tools** (scalable, heuristic)
  - PREfix
  - PREfast

- **2\textsuperscript{nd} generation tools**

- **3\textsuperscript{rd} generation tools**

- **Singularity**
Scalable, heuristic tools

• Commercial challenges
  – large code bases (millions of lines)
  – language corner cases
  – open systems
  – unspecified behavior

• PREfix
  – detailed, path-by-path interprocedural analysis
  – heuristic (unsound, incomplete)
  – expensive (4 days on Windows)
  – primarily language usage defects

• PREfast
  – simple plug-ins find defects by examining functions’ ASTs
  – desktop use
  – easily customized

• Widely deployed in Microsoft
  – 1/8 of defects fixed in Windows Server 2003 found by these tools
PREfix

- Finds errors in C/C++ code
  - null pointer, memory allocation, uninitialized value, resource state
  - hardwired API

- Interprocedural
  - bottom-up traversal of call graph
  - model function by examining limited set of paths (100)
  - apply model at call site
  - expensive, batch computation for large systems

- Large effort to minimize effects of false positives
  - filtering and prioritizing error reports
  - heuristics tuned to reduce noise (at cost of precision)
**PREfix example**

```c
int myfunc(int j)
{
    int k;
    if ( j == 0 )
        k = 1;
    return k;
}
```

**Path #1**
- Reserve memory for variables j, k
- Is j initialized?
  - Evaluate j == 0
  - Result: unknown
  - Assume j == 0
- Assign: k = 1
- Is k initialized?
  - Assign: return = k

**Path #2**
- Reserve memory for variables j, k
- Is j initialized?
  - Evaluate j == 0
  - Result: unknown
  - Assume j != 0
- Assign: k = 1
- Is k initialized?
  - No! Report Uninit
  - Assign: return = k
PREfast

- **Framework** for local analysis of C/C++ code
  - abstract syntax tree from Visual Studio C++ compiler
  - plug-ins traverse trees, looking for idioms

- **Low cost checking**
  - local (function by function)
  - simple heuristics

- **Easy extensibility**
  - large community of tool builders/users

- **Provides basis for more sophisticated tools**
  - familiar infrastructure
  - integrated in complex build processes
SpecStrings

- Company-wide effort to annotate C/C++ APIs
  - standard annotation language (SAL)

- Tools built on PREfast use these annotations
  - e.g., buffer overruns

```c
BOOL WINAPI SetupGetStringFieldW( ... 
   __ecount(ReturnBufferSize) OUT PWSTR ReturnBuffer, 
   IN DWORD ReturnBufferSize, ...);

WCHAR szPersonalFlag[20];
...
SetupGetStringFieldW(&Context, 1, szPersonalFlag, 50, NULL);

PREfast: warning 202: Buffer overrun for stack buffer 'szPersonalFlag' in call to 'SetupGetStringFieldW': length 50 exceeds buffer size 40.
```
Improving software

- 1\textsuperscript{st} generation tools
- 2\textsuperscript{nd} generation tools (sound, declarative)
  - SLAM
  - Fugue
- 3\textsuperscript{rd} generation tools
- Singularity
SLAM

• Software model checking
• Input
  – C source code “as is”
  – API rules written in SLIC language
• Automatically create **abstraction** of C program
  – abstract model = Boolean program
• Systematic **exploration** of model’s state space
  – does feasible path lead to error state in SLIC spec?
• Demand-driven **refinement** of model
  – exclude infeasible paths
Rule in SLIC

state {
    enum {Locked, Unlocked}
    
    s = Unlocked;
}

KeAcquireSpinLock.entry {
    if (s == Locked) abort;
    else s = Locked;
}

KeReleaseSpinLock.entry {
    if (s == Unlocked) abort;
    else s = Unlocked;
}
The SLAM process

- #include <ntddk.h>
- C2BP predicate abstraction
- boolean program
- Bebop reachability check
- Newton feasibility check
- error path
- Harness
- SLIC Rule
- refinement predicates
Static driver verifier

Static Driver Verifier

Driver-Specific Rules

Defects

SLAM Analysis Engine

Device Driver Code

100% path coverage
Add precise, checkable API specifications to a programming language
- can this method return null?
- who owns this resource? do I have to free it?
- do I have to call these methods in a particular order?
- do these fields have data invariants I have to obey?

Documented rules in code itself

```csharp
[return:NotNull]
public string GetPage (string url);
```
API usage rules

```java
[ WithProtocol("raw","bound","connected","down") ]
class Socket {
    [ Creates("raw") ]
    public Socket (...);

    [ ChangesState("raw", "bound") ]
    public void Bind (EndPoint localEP);

    [ ChangesState(State.Any, "connected") ]
    public void Connect (EndPoint remoteEP);

    [ InState("connected") ]
    public int Send (...);

    [ ChangesState("connected", "down") ]
    public void Shutdown (SocketShutdown how);
}
```
Improving software

- 1\textsuperscript{st} generation tools
- 2\textsuperscript{nd} generation tools
- 3\textsuperscript{rd} generation tools (targeted)
  - KISS
- Singularity
New research directions

- Increased specification expressiveness
  - move beyond temporal behavior
  - specify values
  - pre/post-condition and object invariants

- Specialized problems
  - security
  - concurrency

- Combine static and run-time analysis

- New analytic techniques
  - systematic state exploration
  - theorem proving
  - SAT solvers
KISS: static checker for concurrent software

- Insight: many subtle concurrency errors manifest themselves with few context switches
- Q encodes executions of P with few context switches
  - instrumentation introduces extra paths to mimic context switches
- All-path analysis of sequential tool explores all (bounded) context switches
Example

DispatchRoutine( )
{
    int t;

    if (! de->stopping) {
        AtomicIncr(& de->count);
        // do useful work
        // ...

        t = AtomicDecr(& de->count);
        if (t == 0)
            SetEvent(& de->stopEvent);
    }
}

PnpStop( )
{
    int t;

    de->stopping = T;
    t = AtomicDecr(& de->count);
    if (t == 0)
        SetEvent(& de->stopEvent);

    WaitEvent(& de->stopEvent);
}
bool done = F;
DispatchRoutine( ) {
    int t;
    CODE;
    if (! de->stopping) {
        CODE;
        AtomicIncr(& de->count);
        // do useful work
        // ...
        CODE;
        t = AtomicDecr(& de->count);
        CODE;
        if (t == 0)
            SetEvent(& de->stopEvent);
    }
}

PnpStop( ) {
    int t;
    if ($) return;
    de->stopping = T;
    if ($) return;
    t = AtomicDecr(& de->count);
    if ($) return;
    if (t == 0)
        SetEvent(& de->stopEvent);
    if ($) return;
    WaitEvent(& de->stopEvent);
}

if ( !done ) {
    if ($) { done = T; PnpStop( ); }
}
• Trades soundness for scalability
• Cost of analyzing concurrent program $P = \text{cost of analyzing sequential program } Q$
  – size of $Q$ asymptotically same as size of $P$
• Unsoundness precisely quantifiable
  – for 2-thread program, explores all executions with up to 2 context switches
  – for $n$-thread program, explores up to $2n-2$ context switches
Improving software

- 1\textsuperscript{st} generation tools
- 2\textsuperscript{nd} generation tools
- 3\textsuperscript{rd} generation tools (targeted)
  - KISS
- Singularity
Where are we?

- Correctness tools are necessary and useful
  - amazing change in Microsoft culture

- But, tools, no matter how good, are not enough
  - correct, not prevent, defects
    - Japanese & American car production
    - only identify some defects
      - tools miss design flaws and functional errors
  - ignore problems other than coding defects
    - design, architecture, usability, testing, ...

- Focus on entire development process
  - how would software and software development evolve if dependability was primary goal?
Singularity research project

- Large, new project in Microsoft Research
  - joint effort with Galen Hunt
  - Redmond, Cambridge, Silicon Valley

- Major, synergistic changes in languages, tools, and system architecture

- Laboratory for experimentation

Verifiable languages

Correctness tools

Formal design, specification, & modeling tools

Languages

Tools

Runtime

OS

Real-time managed code

Consistent error detection and recovery

System-wide specification

Strong isolation OS

Right isolation
Singularity OS

- Favor correctness and reliability over performance
  - harder to measure, as important
- System almost entirely managed code
  - type safety & garbage collection reduce errors
- Applications entirely managed code
  - safety and platform independence
- Constrain language and programming model to favor verification
  - e.g., closed execution environment facilitates compilation and program analysis
- Strong process isolation minimizes consequence of programmer faults
- Communication explicit and carefully checked
Singularity system architecture

Current Systems

Application

EE

App Dom.

GC

...

OS

Proc

MM

Sched

DD

Singularity

Audio Driver

Device Channel

Kernel Channels

Kernel

Sched

Proc

Kernel EE

GC

MM

Unsafe Code

Application

App EE

GC'

Privacy

James Larus

Building Dependable Software
Singularity language & tools

- Spec#
  - extension to C#, with pre-/post-conditions & invariants

- Language design to support reliable software
  - problematic aspects of programming
    - error handling
    - concurrency
    - modularity
  - balance static verification and runtime mechanisms

- Language implementation issues
  - managed code to build real-time OS
  - processor-memory bottleneck

- Test new ideas against realistic, but manageable code base
  - accelerate co-evolution of languages and tools

- Employ tools throughout entire project lifetime
  - not just on mature code base
Road to dependable software

- Development tools are good starting point
  - amplify human effort & compensate for weaknesses
  - find significant defects in huge code bases
  - eliminate entire classes of defects
  - set quality bar

- Simple tools pave way for more sophisticated ones

- Need more research (& funding) for tools
  - long way to go
  - tools are crude, inaccurate, hard to use
  - limited classes of defects
  - program analysis is difficult
• Pressing challenges
  – design and modeling
    • teachable methodology for good design
    • techniques for modeling systems before building them
    • automatic ways to connect models to code
  – languages, tools, and architectures
    • goal is dependability, not performance
    • systematic defect detection
    • systems resilient to inevitable failures
  – testing improvements
    • static analysis
    • on-line monitoring (Watson)

• Adopt engineers’ mindset
  – we are not artists
  – build artifacts that work

• Industry needs your help!
Thanks for listening

http://research.microsoft.com/spt
http://research.microsoft.com/pprc
Backup slides
Linux & Windows

Red Hat Enterprise Linux AS 3
Advisories (Based on 63 advisories from 2003 - 2004)

Microsoft Windows Server 2003 Enterprise Edition
Advisories (Based on 25 advisories from 2003 - 2004)

This graph was generated by Secunia. Based on Secunia Advisories freely available at http://secunia.com/

http://secunia.com/
Solaris & Windows

Sun Solaris 9 Advisories (Based on 61 advisories from 2003-2004)


This graph was generated by Secunia.
Based on Secunia Advisories freely available at http://secunia.com/

http://secunia.com/
Mac OS X & Windows

Apple Macintosh OS X
Advisories (Based on 35 advisories from 2003-2004)

Microsoft Windows Server 2003 Enterprise Edition
Advisories (Based on 24 advisories from 2003-2004)

This graph was generated by Secunia.
Based on Secunia Advisories freely available at http://secunia.com/

http://secunia.com/
Apache & IIS

Apache 2.0.x
Advisories (Based on 17 advisories from 2003-2004)

Microsoft Internet Information Services (IIS) 6
Advisories (Based on 2 advisories from 2003-2004)

This graph was generated by Secunia.
Based on Secunia Advisories freely available at http://secunia.com/

http://secunia.com/
Lessons from tools experience

• Heuristics suffice
  – developers happy to find (some) defects
  – soundness is additional benefit

• User interface is crucial
  – PREfix spent more effort filtering than finding defects

• Developers are rational
  – usage based on expected cost-benefit tradeoff

• Simple tools pave way for more sophisticated tools
  – change developer attitude and expectations
  – eliminating simple defects exposes complicated ones