Software Robustness and Graceful Degradation in Embedded Systems

May 8, 2000

Philip Koopman
koopman@cmu.edu
http://www.ices.cmu.edu/koopman
(412) 268-5225
Embedded System = Computers Inside a Product
Outline of Talk

◆ A personal trajectory through 4 areas in the embedded systems research space

◆ Previous research areas (what's past is prologue)
  • #1: CPU design
  • #2: Hardware system synthesis

◆ Latest research results (Ballista project)
  • #3: Software robustness testing
  • Can software components be made well behaved?

◆ Current research direction (RoSES project)
  • #4: Graceful degradation/distributed embedded systems
  • Components aren’t going to be well behaved –
    Is automatic reconfiguration a silver bullet?
What’s an Embedded System?

INTEGRATED CIRCUIT

COMPUTER

NETWORKED COMPUTERS

ELEVATOR

AUTOMOBILE

SUBMARINE

CITY TRANSPORTATION SYSTEM

US HIGHWAY SYSTEM

US NAVY PACIFIC FLEET

WIRED HOUSE

INTERNET

BUSINESS / GOVERNMENT / PEOPLE
Classical, General Purpose View of Computing

- **Undergrad – Juniors:**
  - Measured by Performance
  - SPECmarks
Classical View of Computing (upper division)

- **Undergrad – Seniors:**
  - Measured by: Performance, Cost
  - Compilers & OS considered too

- **Graduate Level:**
  - Advanced performance techniques
  - Distributed systems (networks, storage)
Area #1: CPU design for embedded systems

- Let’s build a CPU that’s optimized for embedded systems!
  - Special-purpose instruction set
  - Special-purpose hardware accelerators
  - Optimized for small memory footprint
  - etc., etc., etc.

- Lessons learned:
  - General purpose processors are *Good Enough* for most things
    - Competing with Intel’s technology curve is no fun at all!
  - For better or worse, the desktop market drives the high end
    - Some room left in ultra-high-performance consumer goods (e.g., MPEG players)
  - In the real world, engineers use of-the-shelf components whenever possible
An Embedded Computer Designer’s View

- CPU + Storage + I/O
  - Measured by: Cost, I/O connections, Memory Size, Performance
Area #2: CAD Tool for System Synthesis

- Omniview Fidelity – a design-by-composition tool
  - Most Computer Aided Design (CAD) research is for synthesis, but has limited applicability to industry
  - Fidelity assembles circuit boards by selecting cost-optimized components
  - Experiment: attempt to duplicate hand-optimized design using design-by-composition tool

- Lessons learned:
  - Embedded systems need multi-technology tradeoffs
    - Analog, hardware, software, mechanical, …
  - Design tradeoffs for desktop computers are different than embedded tradeoffs
    - Desktop
      - Average operating power
      - Component purchase costs
      - Similar designs are equivalent
      - Designed by specialists
    - Embedded
      - Standby power
      - Lifecycle component costs
      - Each change requires certification
      - Designed by generalists
An Embedded Control System Designer’s View

Area #3: Software Robustness Testing

◆ Research motivation:
  • Low-cost reliability is essential for embedded systems
    – But we really don’t know how to do that
  • Component-based systems are becoming prevalent
    – Anecdotally, exception handling *may* be a big source of problems
    – *Idea:* to build robust systems, put together individually robust components
  • *Software* components are probably the most important to study
    – Increasingly, that’s where the complexity ends up…

◆ Software Dependability $\approx$ reliability + robustness + safety + security
  • Software “Reliability” $\equiv$ operates per specification
  • Software Robustness $\equiv$ acts reasonably in exceptional situations

◆ Research goal
  • Find a way to quantify robustness levels
This is a bad place to discover your RTOS is only 83.3% robust.
The *Ballista* Robustness Testing Approach

- **Use fault injection techniques**
  - “Ballista” is an ancient siege weapon for hurling big projectiles with good accuracy
  - Traditional fault injection corrupts code in system under test
    - Usual Hypothesis: “If there were a defect in the OS, the system could crash”

- **Ballista testing tool does API-level fault injection**
  - Simulates a software defect in something calling the interface
  - Does *NOT* inject a defect in the system under test itself
    - Ballista Hypothesis: “If there were a defect in a user program, the OS could crash”
**Ballista Research Challenges**

- **Scalability of testing “oracle”**
  - How do you know if the test got the right answer?
  - Usual method requires knowing expected result of each and every test
  - *Solution:* use a “crash/hang” check instead of functional correctness check
    (similar to idea to “crashme” randomized OS testing, but applicable to any API)

- **Scalability of test cases**
  - Software testing effort is usually proportional to number of functions tested
  - “Scaffolding” set-up code for tests is a large development effort
  - *Solution:* create tests based on number of data types instead of functions
    (similar to idea for Category Partitioning testing method [Ostrand & Balcer ’88], but
    without no per-test analysis required)

- **OS vendors said we’d never find anything**
  - Conventional wisdom is only remaining bugs are obscure, timing-related
  - *Solution:* they were wrong
**Ballista: Scalable Test Generation**

**API**

```
write(int filedes, const void *buffer, size_t nbytes)
```

**Testing Objects**

- **File Descriptor**
  - FD_CLOSED
  - FD_OPEN_READ
  - FD_OPEN_WRITE
  - FD_DELETED
  - FD_NOEXIST
  - FD_EMPTY_FILE
  - FD_PAST_END
  - FD_BEFORE_BEG
  - FD_PIPE_IN
  - FD_PIPE_OUT
  - FD_PIPE_IN_BLOCK
  - FD_PIPE_OUT_BLOCK
  - FD_TERM
  - FD_SHM_READ
  - FD_SHM_RW
  - FD_MAXINT
  - FD_NEG_ONE

- **Memory**
  - BUF_SMALL_1
  - BUF_MED_PAGESIZE
  - BUF_LARGE_512MB
  - BUF_XLARGE_1GB
  - BUF_HUGE_2GB
  - BUF_MAXULONG_SIZE
  - BUF_64K
  - BUF_END_MED
  - BUF_FAR_PAST
  - BUF_ODD_ADDR
  - BUF_FREED
  - BUF_CODE
  - BUF_16
  - BUF_NULL
  - BUF_NEG_ONE

- **Test Values**
  - SIZE_1
  - SIZE_16
  - SIZE_PAGE
  - SIZE_PAGEX16
  - SIZE_PAGEX16plus1
  - SIZE_MAXINT
  - SIZE_MININT
  - SIZE_ZERO
  - SIZE_NEG

**Test Case**

```
write(FD_OPEN_READ, BUF_NULL, SIZE_16)
```

- Ballista combines test values to generate test cases
CRASH Robustness Testing Result Categories

- **Catastrophic**
  - Computer crashes/panics, requiring a reboot
    - *e.g.*, `GetThreadContext(GetCurrentThread(), NULL);`
- **Restart**
  - Benchmark process hangs, requiring restart
- **Abort**
  - Benchmark process aborts (*e.g.*, “core dump”)
- **Silent**
  - No error code generated, when one should have been
    - *e.g.*, de-referencing null pointer produces no error
- **Hindering**
  - Incorrect error code generated
  - Found via by-hand examinations, not automated yet
Results for Unix Operating Systems

Normalized Failure Rate by Operating System

Operating System Tested:
- AIX
- FreeBSD
- HPUX 9.05
- HPUX 10.20
- Irix 5.3
- Irix 6.2
- Linux
- Lynx
- NetBSD
- OSF-1 3.2
- OSF-1 4.0
- QNX 4.22
- QNX 4.24
- SunOS 4.13
- SunOS 5.5

- Abort %
- Silent %
- Restart %
- Catastrophic

Normalized Failure Rate (after analysis)
Thinking of running your critical apps on NT?

Isn't there enough world suffering?
Robustness Beyond Operating Systems

- Some software is very robust
  - HLA RTI – DoD distributed simulation backplane

- Unfortunately, commercial software components tend to be non-robust
  - Unix systems often displayed a vulnerability to crashing
  - Windows CE had 28 functions that could cause a crash
  - Initial results on several CORBA implementations don’t look promising
  - Initial results on accelerated software aging tests look bad for both Linux and Windows

- How do you build a robust system from non-robust components?
  - Multi-version Unix implementations won’t work (we checked)
  - The market isn’t demanding more robust software (yet)
  - Solution: (?) build systems that degrade gracefully when components fail
Highly distributed systems

- Measured by: Product family success, life-cycle cost, dependability
- Fine-grain, system-on-chip nodes form distributed systems
  - Micro-Electrical/Mechanical Systems (MEMS) for I/O
  - Microcontroller & network connection “for free” on same piece of silicon
Area #4: Graceful Degradation

- Research motivation:
  - Future embedded systems will be low cost, but highly distributed
  - In the common case, not all components will be working
    - Degraded or failed component hardware
    - Unreliable/non-robust component software

- Existing graceful degradation techniques rely upon manual reconfiguration
  - Works fine for a few processors
  - Doesn’t scale to thousands of processors

- Research goals
  - Achieve automatic graceful degradation for a given system architecture
  - Create general principles for designing architectures that gracefully degrade
RoSES: Robust Self-configuring Embedded Systems

- **Automatic configuration management** is a unifying capability
  - Product families can include degradation as well as intentional price/performance tradeoff points

- **Consider component failure as an example:**
  - Component fails – triggers reconfiguration for degraded operation
  - Component replaced – reconfiguration to integrate repair part
  - New component added – reconfiguration to upgrade system

- **That’s a lot to attempt all at once…**
  - Static configuration at first
  - On-the-fly configuration as an eventual goal
A Simplistic Example

- Control of gasoline engine speed
  - Complicated system controls fuel if valve is installed/operational
  - But, baseline capability is retained in case of failure

**Throttled Engine Controller**

**Hit-or-Miss Constant Speed Engine Controller**

*Degrades to*
**Different Sensors / Different Capabilities**

- Similarly, different actuators have different capabilities
  - *Mobile Object Adapters* translate raw capability into desired interface

<table>
<thead>
<tr>
<th>Mobile Object Adapter</th>
<th>Mobile Object Adapters</th>
</tr>
</thead>
<tbody>
<tr>
<td>RELATIVE SPEED SENSOR</td>
<td>RELATIVE SPEED INTERFACE (TOO FAST/TOO SLOW)</td>
</tr>
<tr>
<td>LOCAL COMPUTATION, ALGORITHMS, DATA STORAGE</td>
<td>LOCAL COMPUTATION, ALGORITHMS, DATA STORAGE</td>
</tr>
<tr>
<td>EXAMPLE SIMPLE SPEED SENSOR</td>
<td>EXAMPLE HIGH-END SPEED SENSOR with MULTIPLE INTERFACES</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Embedded Network</th>
<th>Embedded Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>RPM INTERFACE</td>
<td>OVERSPEED EMERGENCY INTERFACE</td>
</tr>
<tr>
<td>PERCENT MAXIMUM INTERFACE</td>
<td>RPM INTERFACE</td>
</tr>
<tr>
<td>RELATIVE SPEED INTERFACE (TOO FAST/TOO SLOW)</td>
<td>RELATIVE SPEED INTERFACE (TOO FAST/TOO SLOW)</td>
</tr>
</tbody>
</table>
Generic RoSES System Architecture

SMART SENSORS

- Basic Sensor Device
- Baseline Sensor SW Functionality
- SW Adapter for High Level Logical Interface
- Dynamic Interface to Object Bus

SMART ACTUATORS

- Basic Actuator Device
- Baseline Actuator SW Functionality
- SW Adapter for High Level Logical Interface
- Dynamic Interface to Object Bus

Object Bus (ORB Core operating across a network)

CUSTOMIZATION MANAGER

Adapter Repository  Co-Scheduling & Assignment Tool
Functionality To Hardware Mapping

- One element of RoSES:
  Automatic allocation of HW & SW components
  - Maximize utility of functions within hardware constraints

FUNCTIONAL REQUIREMENTS

CONTROL ALGORITHMS & SOFTWARE

Dynamic Adaption Runtime System

CPUs --- MEMORY --- NETWORK

SENSORS --- ACTUATORS
Near-Term Research Challenges

- Mapping functionality onto hardware
  - Maximize utility of result given constrained resources

- Achieving real-time operation
  - Co-schedule CPU, Memory, Network usage to meet real-time deadlines

- Achieving “plug & play” capabilities
  - Is CORBA too “fat”? (how about Jini…)
  - Avoid re-inventing distributed object technology if possible!

- Testbed & demonstration
  - Generic automotive testbed
  - Apply techniques to multi-sensor vehicle navigation & other functions

- Plenty of long-term research challenges too, of course
Other Current Activities

- Chair of IFIP WG 10.4 SIG on dependability benchmarking
  - How do you get measures of system dependability that work for real-world conditions?
  - Representatives from universities, industry, government labs

- Industry-funded research efforts
  - General Motors, Bosch: graceful degradation of automotive systems
  - Adtranz: dependability analysis of train network protocol
  - Emerson, Microsoft: software robustness of Windows
  - IBM: “bulletproof Linux”
  - ABB: software robustness of embedded systems
Teaching System Architecture (current status)

- CMU ECE 18-540: Distributed Embedded Systems
  - Elevator as an example
  - Includes lightweight software engineering: requirements to validation
  - Material motivated via “war stories”

- Business issues
  - How does a particular company make its profits?
  - Non-technical constraints on solutions are a reality

- Levels of abstraction
  - Top-down decomposition + Bottom-up synthesis
  - Orthogonal building blocks (when you can find them)

- Multi-technology tradeoffs

- Non-functional requirements
  - “ilities”, safety, cost

- Life-cycle perspective
  - Requirements through disposal
  - Selected real-world issues: spare parts, cross-cultural designs, ethics
Current Research Scope

NETWORKED COMPUTERS

- ELEVATOR
- AUTOMOBILE
- SUBMARINE
- CITY
- TRANSPORTATION SYSTEM
- US
- HIGHWAY SYSTEM
- US NAVY PACIFIC FLEET
- WIRED HOUSE
- INTERNET
- BUSINESS / GOVERNMENT / PEOPLE
Phil Koopman
Jiantao Pan
Kanaka Juvva (graduating)
Bill Nace
Kobey DeVale
Ying Shi
John DeVale

Meredith Beveridge
Charles Shelton
Sandeep Tamboli (graduating)

Not shown (new additions):
Tridib Chakravarty
Beth Latronico