Information Extraction from Real-time Applications at Run Time

Sebastian Fischmeister
University of Waterloo

esg.uwaterloo.ca
Outline

• Setting the stage
• Motivate the need for information extraction
• Time-aware instrumentation
• Time-triggered runtime verification
• Conclusions
SETTING THE STAGE:
REAL-TIME SAFETY-CRITICAL EMBEDDED SOFTWARE
Embedded Systems Everywhere
## Embedded Software Everywhere

<table>
<thead>
<tr>
<th>System</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Darlington Shutdown System</td>
<td>40 000</td>
</tr>
<tr>
<td>Mars Science Laboratory</td>
<td>4 000 000</td>
</tr>
<tr>
<td>Boeing 787</td>
<td>6 500 000</td>
</tr>
<tr>
<td>Current luxury car</td>
<td>100 000 000</td>
</tr>
</tbody>
</table>
Safety-critical Real-time Systems

Physics doesn’t wait for you.
The right value too late still causes errors.
THE NEED FOR INFORMATION EXTRACTION

- Software is getting big
- We can’t comprehend it
- Bugs are real
Software is Getting Big

![Code size chart](chart.png)
Software is Getting Big

- GM car in 1981: 50,000 LOC
- GM car in 2011: 100,000,000 LOC
- Next generation car: 300,000,000 LOC
We Cannot Comprehend Software

• Software is where the innovation is happening! *Features sell, apps everywhere*

• Software size and complexity is the challenge!

Illustrating one root cause: Bridge from Tokyo to Vancouver
10 000 dots

100K?

10M?
~100M Pixels

Courtesy of Bob Mallard.
Bugs are Real

- 80% of the developer time is debugging
- 30-50% of the total cost is integration testing and debugging

<table>
<thead>
<tr>
<th>Source</th>
<th>Language</th>
<th>Failure per KLOC</th>
<th>Formal methods used?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Siemens operating system</td>
<td>Assembly</td>
<td>6-15</td>
<td>No</td>
</tr>
<tr>
<td>NAG scientific libraries</td>
<td>Fortran</td>
<td>3.00</td>
<td>No</td>
</tr>
<tr>
<td>CDIS air-traffic-control support</td>
<td>C</td>
<td>0.81</td>
<td>Yes</td>
</tr>
<tr>
<td>Lloyd’s language parser</td>
<td>C</td>
<td>1.40</td>
<td>Yes</td>
</tr>
<tr>
<td>IBM Cleanroom development</td>
<td>Various</td>
<td>3.40</td>
<td>Partly</td>
</tr>
<tr>
<td>IBM normal development</td>
<td>Various</td>
<td>30.0</td>
<td>No</td>
</tr>
<tr>
<td>Satellite planning study</td>
<td>Fortran</td>
<td>6-16</td>
<td>No</td>
</tr>
<tr>
<td>Unisys communication software</td>
<td>Ada</td>
<td>2-9</td>
<td>No</td>
</tr>
</tbody>
</table>
Information Extraction

• Information extraction helps the developer understand the program’s behavior at run time:
  – Testing, debugging, tuning, monitoring, validating, certifying

• **Goals:** Easy, low cost, readily available, deployable, and **shouldn’t break anything.**

• **Problem:** Existing approaches mostly consider logical correctness only, but **what about other properties? (e.g., timing)**
Vision & Path

Time aware instrumentation
- Coverage criterion [RTAS’09, TII]
- ISA extension [tech rep]

Time-triggered runtime verification
- Crit. CFG & sampling [FM’11]
- Mem vs. sampl. tradeoff [RV’11]

Time-triggered execution monitoring
- Preemptive [OPODIS’11]
- Super-loop [LCTES’11]

Information extraction framework for real-time safety-critical applications

Observability
- In software

Markers
- [LCTES’10]
TIME-AWARE INSTRUMENTATION


Current instrumentation methods preserve only logical correctness.

Can we capture runtime execution behavior (=variable assignments) with no or little timing interference?
Execution Time Profile

Number of observations vs. Execution time

WCET
Deadline
Idea in a Nutshell

Right shift!

Frequency

Execution time

Deadline

Original

Instrumented
Challenges

• Can we actually create this right shift?
• What will we do, if there is insufficient slack?
• What does the optimal solution look like?
Capturing on non-WCET Paths

WCET path

Basic block

Ignore assignment
On WCET path
Idea in a Nutshell

What if it doesn’t fit?

Concept of coverage
Coverage

• Coverage of an insertion point: $p$ of the last branching point

• Coverage of a path: miss ratio of assignments to logged assignments on the path

• Coverage of an instrumentation: miss ratio of on all paths

Optimality: For a given time budget, what placement of log statements yields the best coverage? [RTAS’09, TII]
Standard Toolchain

Function selection

Source analysis

Naïve instrumentation

Compilation

Execution
Time-aware Instrumentation Toolchain

- Function selection
- Source analysis
- Instrument (time aware)
- Timing analysis
- Adjust coverage
- Execution
- Compilation

Prototypes for:
- ARM9
- ATMEL
Case Study: OLPC Keyboard Controller

- Test feasibility
- Test hypothesis that shift in execution time occurs
- Experiment with time budgets
handle_power()

Approach works, but effects are limited without extra time budget.

25% of the paths share basic blocks with the WCET path.
Increasing the Time Budget

Increase in Coverage with additional Time Budget

- Small increase in the time budget has huge effects.
Case Study: Embedded FS

Instrumentation of fclose (o=0, r=0.13554)
Case Study: Embedded FS

Instrumentation of fputs (o=0, r=0.21667)

Instrumentation of fputs (o=3, r=0.63258)

Instrumentation of fputs (o=10, r=0.9697)
Ongoing Work on TAI

• What makes a program instrumentable?

• Can we transform a program to be more suitable for (time-aware) instrumentation?

• What other properties than time are of interest? *(arbitrary non-functional properties)*
Instrumentation can be time aware.

The “right shift” idea works and is technically feasible.

Long-term vision:
- New methods with better coverage
- New methods for other properties
- Software & hardware hybrid solutions
TIME-TRIGGERED RUNTIME VERIFICATION


Runtime Verification

- Observing program to check compliance with some specification.
  - Online, offline (traces)

- Example uses:
  - Runtime validation and safety
  - System steering
  - Performance monitoring and tuning
  - Debugging
An Online External RV System

- Application
  - Program
    - Observe
    - Observer
  - Recovery & steering
- Monitor
  - Eval. properties
  - Report
Problem

Current approaches are event-triggered and can lead to transient overloads at run time.

Can we observe the program with predictable overhead?
Event-based Runtime Verification

We *instrument* lines 5 and 6 such that the monitor is invoked.

```
a = scanf(...);
if (a % 2 == 0) goto 9
else {
    printf(a + "is odd");
    b = a/2;
    c = a/2 + 1;
    goto 10;
}
printf(a + "is even");
end program
```

‘b’ and ‘c’ of interest
ET has Problems (Overhead)

Spikes in overhead are a problem in real-time embedded systems.
Our idea is to **bound** the overhead of runtime verification and make it **predictable** (=> **engineerable**).

We analyze the correctness of the system in a **time-triggered** fashion:

– At the end of each period, the monitor is invoked to take a **sample** from the system to analyze its soundness.
Objective

Bounded and predictable overhead

overhead

execution

T T T
TTRV Problem 1

Identify the **sampling period**, such that the monitor observes all changes vital to evaluating the correctness of a given property.
Our Approach

C Program

Property

Control Flow Graph

Sampling Frequency
Generating CFG

```c
1:    a = scanf(...);
2:    if (a % 2 == 0) goto 9
3:    else {
4:      printf(a + "is odd");
5:*     b = a/2;
6:*     c = a/2 + 1;
7:    } goto 10;
8:    }
9:    printf(a + "is even");
10:   end program
```

Basic Block

Weight = best case execution time
Calculating Sampling Period (1)

Critical Basic Block

```c
1:    a = scanf(...);
2:    if (a % 2 == 0) goto 9
3:    else {
4:      printf(a + "is odd");
5:*    b = a/2;
6:*    c = a/2 + 1;
7:    goto 10;
8:    }
9:    printf(a + "is even");
10:   end program
```
Calculating Sampling Period (2)

Each critical basic block contains only one critical instruction
Calculating Sampling Period (3)
The *minimum sampling period* for a property is the minimum arc weight that originates from a corresponding critical basic block.
Computing the Sampling Period

Sampling Period = 1
The basic sampling period can be very small. Can we increase the sampling period?

Use **history information** to increase the minimal sampling period.
Calculating Sampling Period

The Problem in Detail

Minimum sampling period is a conservative estimate and often results in sampling with high frequency and over-sampling in some execution branches.
Solution: Leveraging Histories

\[ x := 1 \quad 1 \quad x := x^2 \quad 4 \quad x := 5 \]

\[ x := 1 \quad 5 \quad x' := x^2 \quad 5 \quad x := 5 \]
**Optimization Problem**

**Instance.** A weighted digraph $G = <V, A, w>$ and positive integers $X$ and $Y$.

**Decision problem.** Does there exist a set $U$ of vertices, such that by collapsing all vertices in $U$, we obtain a weighted digraph $G = <V', A', w'>$; where $|U| \leq Y$ and for all arcs $(u, v)$ in $A'$, $w'(u, v) \geq X$?
Mapping to ILP

\[
\begin{align*}
\text{Minimize} & \quad c.z \\
\text{Subject to} & \quad A.z \geq b
\end{align*}
\]

• Variables
  – \( x \) for collapsing vertices
  – \( a \) for arc weights
  – \( y \) as choice variables to simplify the encoding

• Constraints
  – All arc weights must be greater than the sampling period
  – Updates on arc weights when collapsing vertices
  – Loops with critical vertices must not be collapsed
Prototype Tool Chain

Property

C Program

Sampling period

CIL

CFG

ILP Model

lp_solve

Instrumentation Instructions
Experimental Setting (from MiBench)

- **Blowfish**: 745 lines of code. Results in a CFG of 169 vertices and 213 arcs. We take 20 variables for monitoring.

- **Dijkstra**: 171 lines of code. Results in a CFG of 65 vertices and 78 arcs. We take 8 variables for monitoring.

- 3 experiments
  - Event-triggered monitoring
  - Sampling-based monitoring without history
  - Sampling-based monitoring with history

- All experiments are conducted on a Mac Book Pro with 2.26GHz Intel Core 2 Duo and 2GB main memory.
Experimental Results (Blowfish – 50x)

- Overhead spikes
- Burstiness
- Predictable pattern of overhead
Experimental Results
(Blowfish – 100x)

Longer sampling period results in less overall overhead (faster execution)
Experimental Results (Dijkstra – 50x)

The diagram shows the overhead (in milliseconds) versus execution time (in seconds) for different types of algorithms. The x-axis represents the execution time in seconds, ranging from 0 to 6. The y-axis represents the overhead in milliseconds, ranging from 0.05 to 0.4. The graph includes three types of algorithms:

1. **Sampling-based with no history** represented by circles.
2. **Event-based** represented by triangles.
3. **Sampling-based 50x MSP** represented by pluses.

The data points are scattered across the graph, indicating the overhead for each algorithm at different execution times.
Experimental Results (Dijkstra – 100x)

5x faster execution

![Graph showing experimental results with different execution times and overheads. The graph compares sampling-based methods with and without history, event-based methods, and sampling-based methods with 100x MSP.]
Cumulative Overhead

Absolute overhead in fft

- **type**
  - ev
  - sb
  - sb-5

- **Axes**
  - X-axis: Time
  - Y-axis: Absolute overhead

Graph showing the cumulative overhead over time for different types.
Cumulative Overhead

Absolute overhead in bitcount

- **Type**: ev, sb

---

**Axes**:
- X-axis: Time
- Y-axis: Absolute overhead

**Trends**:
- Three distinct lines representing different types of overhead.
- The lines show an increasing trend over time.
Experimental Results (Dijkstra – Memory)
Experimental Results (Blowfish – Memory)
Optimal use of a history buffer is NP-complete with the size of the CFG.

Can we find a near-optimal solution in reasonable time?
Short Answer: Yes

| CFG Size(|V|) | ILP | Heuristic 1 (Greedy) | Heuristic 2 (VC) | Genetic Algorithm |
|----------|-----|----------------------|------------------|-------------------|
|          |     | SOF | time (s) | SOF | time (s) | SOF | time (s) | SOF |
| Blowfish | 177 | 5316 | 0.0363 | 7.8 | 0.8875 | 8 | 383 | 2.5 |
| CRC      | 13  | 0.35 | 0.0002 | 3.5 | 0.0852 | 3 | 0.254 | 1.5 |
| Dijkstra | 48  | 1808 | 0.0064 | 1.2 | 0.1400 | 1.2 | 116 | 1.7 |
| FFT      | 47  | 269  | 0.0042 | 1.7 | 0.1737 | 1.8 | 74 | 1.1 |
| Patricia | 49  | 2084 | 0.0054 | 1.4 | 0.1369 | 1.6 | 140 | 1.5 |
| Rijndael | 70  | 3096 | 0.0060 | 1.6 | 0.2557 | 2.1 | 370 | 1.9 |
| SHA      | 40  | 124  | 0.0039 | 2.2 | 0.1545 | 2.2 | 46 | 1.3 |
| Susan    | 20259 | ∞ | 3181 | N/A | 26211 | N/A | 923 | N/A |

**Table 1.** Performance of different optimization techniques.
(a) Increase in the number of execution of instrumentation instructions. (b) Increase in the maximum size of history between two samples.
Summary (TTRV)

• A time-triggered approach is a feasible approach for runtime verification
• Surprising result that TTRV can even lead to better overall performance
• Lot of open problems:
Conclusions

• Embedded software is everywhere and increasing in complexity and size.
• Many development activities require comprehending the system, and we thus need information extraction.
• Current tool only support preserving logical correctness. The presented work provides a glimpse of what can be possible.
• While other disciplines have a thorough understanding of the probe effect, software engineering considers only logical correctness.
• Understanding how to extract information from programs at run time is a widely unexplored area. (=> software probe effect beyond logical correctness)
Acknowledgements

This research was mostly supported by the Canadian tax payer.

Thank you!
Questions?

(PS: Postdoc and grad student positions available, just talk to me afterwards or email me sfischme@uwaterloo.ca)