Ji.Fi: Visual Test and Debug Queries for Hard Real-Time

Ethan Blanton[1], Demian Lessa[2], Lukasz Ziarek[1,2], Bharat Jayaraman[2]

Fiji Systems Inc.[1], SUNY Buffalo[2]

October 25, 2012

Motivation

- Debugging real-time systems is difficult
  - Symbolic debugger (gdb, jdb) overhead
  - Stop-and-examine difficult or entirely precluded
- Cost per line of certified code is very high
  - $1000 per line, versus $15-30 for non-certified code[1]
- We need more tools for SCJ and RTSJ
- We want a tool for teaching undergraduates SCJ/RTSJ

[1] Daewoo Securities Company LTS: “Software— with Crisis comes Opportunity.” 2008.

JIVE

- JIVE is a visual debugger
  - Uses extended UML object and sequence diagrams
  - Declarative queries reduce visualization complexity
  - The necessary data and no more
- Built on Eclipse
- Extended evaluation in the literature

JIVE Suitability

JIVE provides many nice features

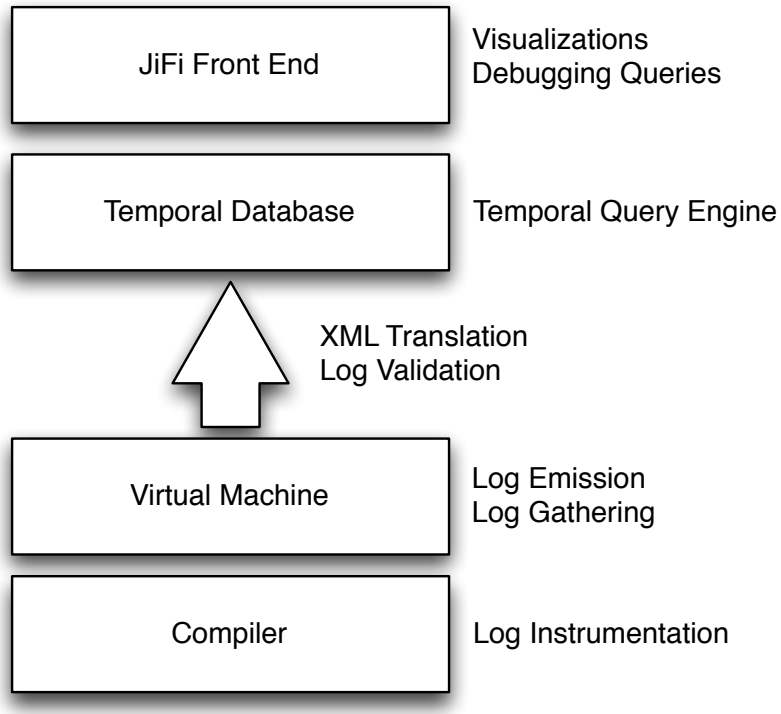
- Powerful visualization
- Forward- and reverse-stepping
- Queries to reduce cognitive burden : “when” style questions

However...

- No sense of real time, only logical time
- No offline debugger
- JDI event granularity is too fine
- No monitors or synchronization events

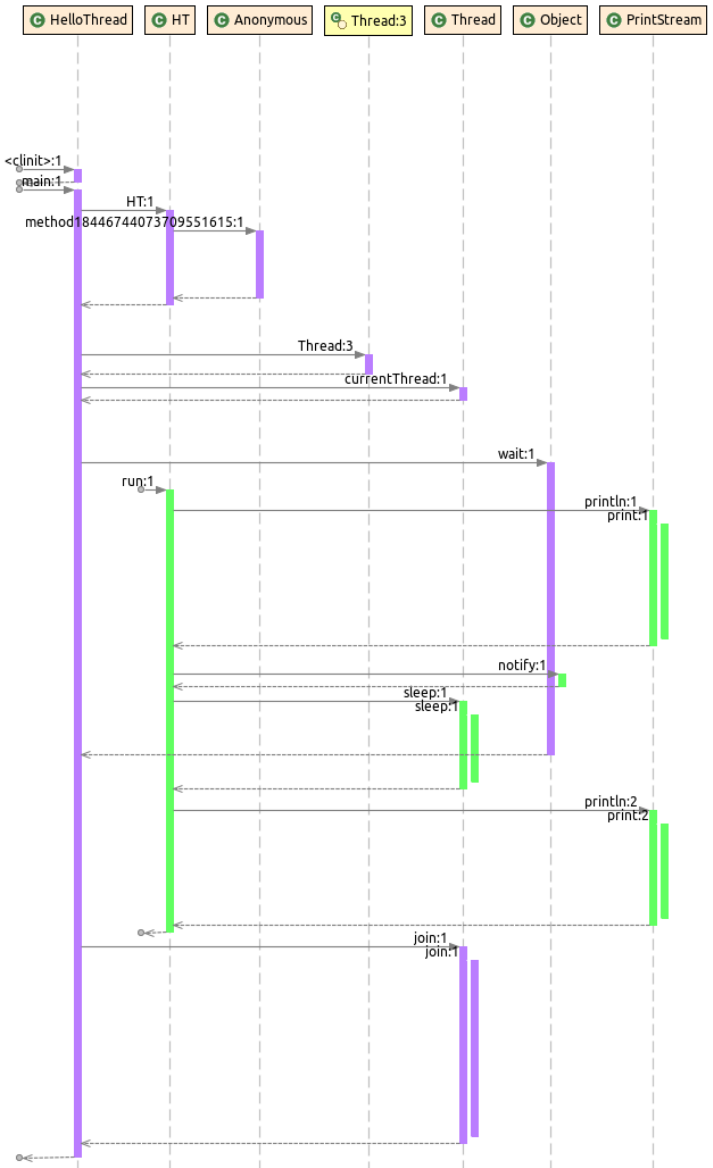
Extending JIVE: JI.FI

- System split into two levels
- Replaceable bottom half (Use your own compiler/VM!)
  - Our implementation uses the Fiji VM
- Modified JIVE top half

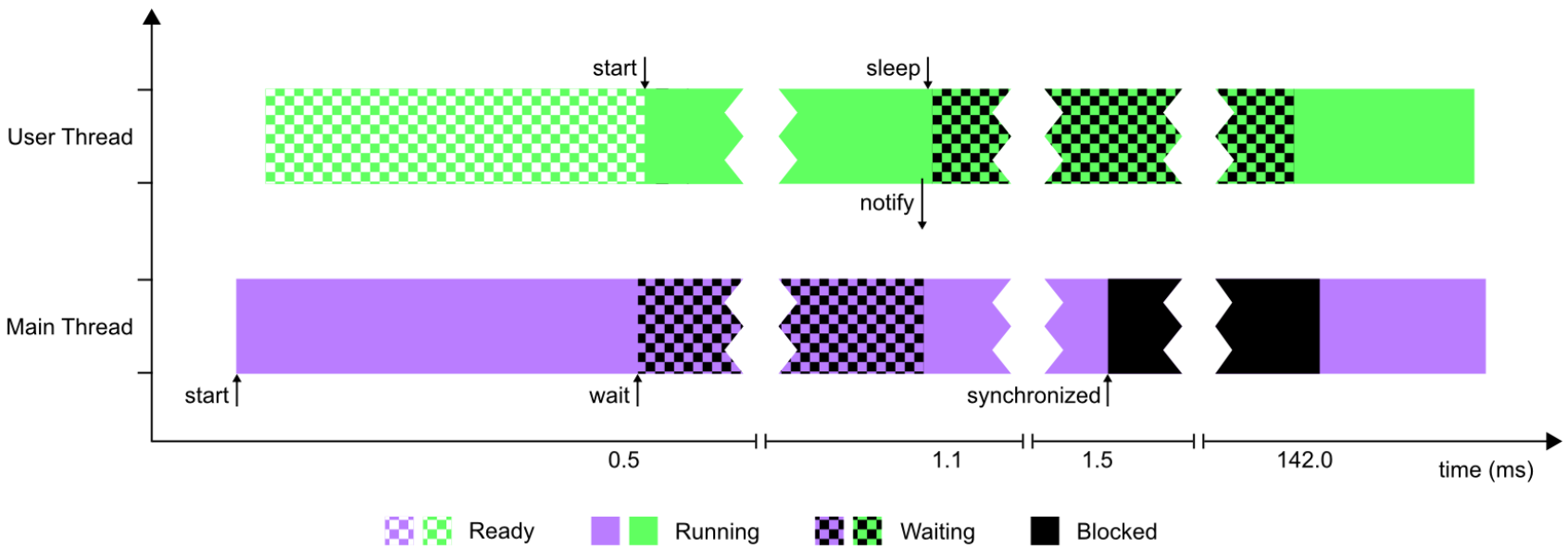


Visualizations

- RT Object Diagram
- RT Sequence Diagram
- Thread State Diagram



Thread State Diagram



Dynamic Query-Based Analysis

Temporal Queries provide analysis of temporal behavior

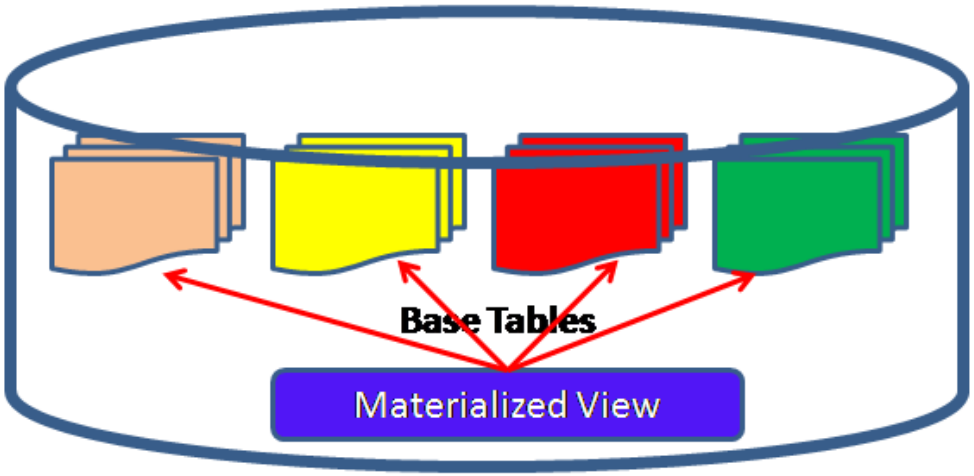
- Natural representation of a real-time system log
- Complex behaviors reduced to SQL-like expressions
- Recursive queries are also expressible
- Questions like:
  - Which monitors are accessed by multiple threads?
  - Does this execution trigger priority inversion avoidance?
  - Which periods include deadline misses?

Contended Monitors

```
1  SELECT m1.threadId AS h, m2.threadId AS b,
2         m1.monitor AS m, m2.time
3  FROM (event NATURAL JOIN event_monitor) m1,
4       (event NATURAL JOIN event_monitor) m2,
5       (event NATURAL JOIN event_monitor) m3
6  WHERE
7    m1.monitor = m2.monitor AND
8    m1.monitor = m3.monitor AND
9    m1.threadId = m3.threadId AND
10   m1.threadId <> m2.threadId AND
11   m1.time < m2.time AND m2.time < m3.time AND
12   m1.kind IN (12, 13) AND
13   m2.kind = 11 AND
14   m3.kind IN (15, 16) AND
15   NOT EXISTS (SELECT 1
16      FROM (event NATURAL JOIN event_monitor) mx
17      WHERE mx.kind IN (15,16) AND
18            m1.monitor = mx.monitor AND
19            m1.threadId = mx.threadId AND
20            m1.time < mx.time AND mx.time < m3.time);
```

Contended Monitors: Materialized Relations

```
1  SELECT holder AS h, threadId AS b, monitor AS m, time
2  FROM event NATURAL JOIN event_monitor
3       NATURAL JOIN monitor
4  WHERE
5    -- event is LOCK BEGIN
6    kind = 11 AND
7    -- monitor is locked
8    lockCount > 0 AND
9    -- lock is held by a different thread
10   holder <> threadId
```



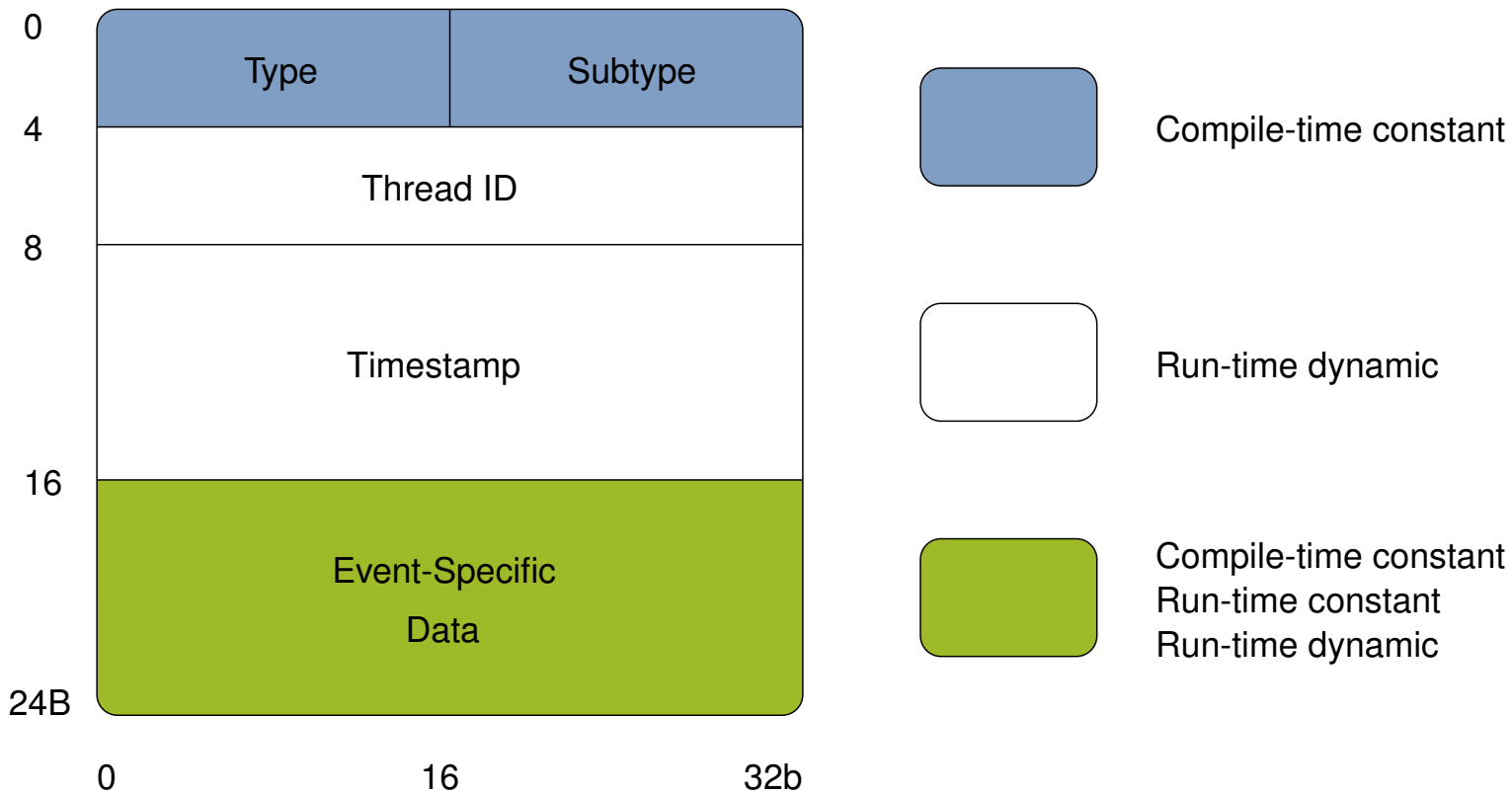
Making Queries Easier

- JI.FI will include relevant materialized relations, *etc.*
- Designing queries is still complicated
- Canned queries can be presented to users as Forms
  - Users fill out simple forms, get sophisticated answers
  - *e.g.*, “Show me every period where PIP was invoked on monitor <X>”
- More work to be done on this front

Log Structure

- There are two levels of logs:
  - Low-level VM logs
  - High-level XML logs
- Neither level of log is VM-specific
  - *Although, translation between the two may be!*
- VM logs are designed for speed and predictability
- XML logs are designed for expression and extensibility

Real-Time Log Entry



Log Buffers

- Each Java thread maintains a buffer of log entries
- Individual log entries are lock-free writes
- Handoff to a log-flushing thread has a tight critical window
- Log buffer management requires more work

Log Validation

Both low-level and high-level log manipulations involve validation

- Verify that VM log emissions are complete and correct
  - Monitor actions balance
  - Method calls nest
  - *etc.*
- Validate XML structure
- Check equivalence of VM and JIVE semantics

Runtime Instrumentation

Code paths with neat insertion points get runtime instrumentation

- Monitors
- Thread state changes
- Priority changes

```
1 static void sleep(long ms, int ns) throws InterruptedException {
2     long t = Time.nanoTime() + ms * 1000 * 1000 + ns;
3     FlowLog.log(FlowLog.TYPE_THREAD, FlowLog.SUBTYPE_SLEEP, t);
4     sleepAbsolute(t);
5     FlowLog.log(FlowLog.TYPE_THREAD, FlowLog.SUBTYPE_WAKE, t);
6 }
```

Compiler Instrumentation

Logging within user code and compiler-generated code is inserted by the bytecode-to-C compiler

- Method calls
- Returns

Instrumentation Challenges

- Keeping log fields constant
  - *e.g.*, Assign unique 32-bit integer to every method
- Filtering uninteresting methods
  - Rule-based (user-provided code vs. VM code)
  - Annotations
- Choosing minimal but sufficient data for events
- Timestamps can be very expensive!

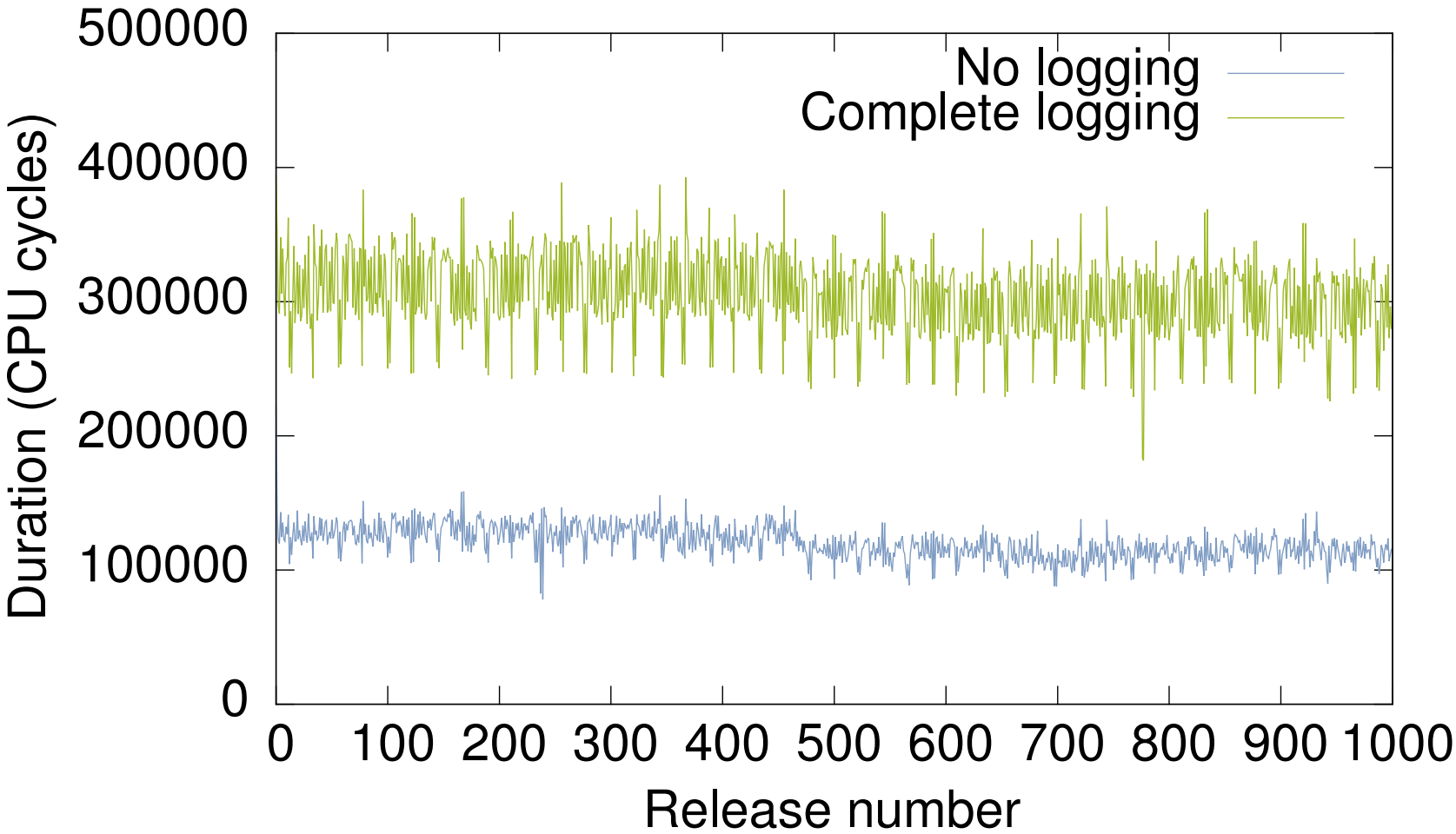
Evaluation Methodology

- Evaluated using CDj
- CDj simulates an air traffic control collision detector
- Many tunable parameters
  - 6 aircraft
  - 50 frames per second (20 ms release)
  - Varying number of frames

Computational Overhead

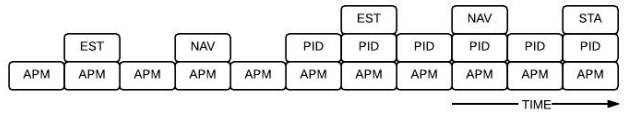
- Pure software implementation
- Architecturally optimized, but not optimized “in the small”
- 67% overhead for extensive logging
  - Covers all user-provided code plus entry into system code
  - Overhead doubles to include getters/setters/trivial methods
- Substantial overhead reduction by turning off features
- Method tracing is the most expensive event class

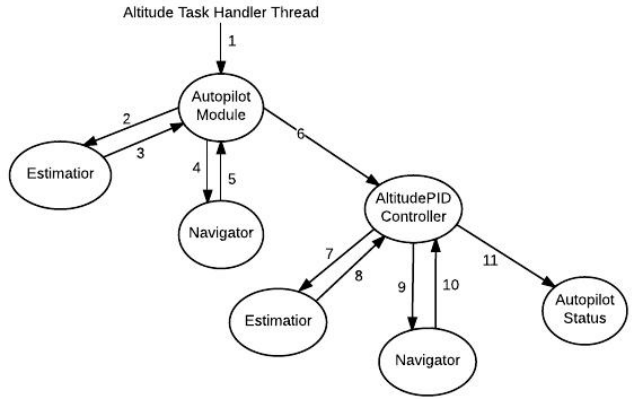
Predictability

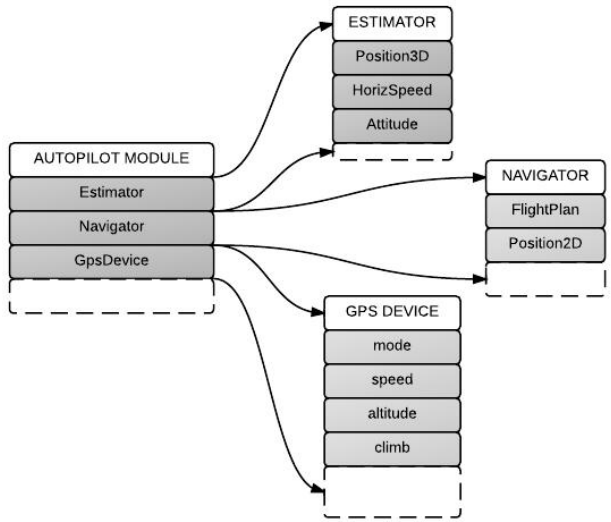


Next Steps: SCJ + RTSJ

- Visualizations : Missions, Scopes, Parameters
- Temporal Queries : Form Queries
- Logging : Scope instrumentation
- Generate scope size estimates (unlimited mode)







Next Steps: Provably Predictable Logging

- WCET Analysis of the logging framework
- Schedulabiltiy of program + logging framework
- Log impact report via Visualization

Next Steps: Multi-Run Executions

- Motivation: Cannot log everything
- Produce multiple version of the program, each instrumented to log different data
- Post execution, merge logs