J1.F1: Visual test and debug queries for hard real-time

Ethan Blanton\textsuperscript{1*}, Demian Lessa\textsuperscript{2}, Puneet Arora\textsuperscript{2}, Lukasz Ziarek\textsuperscript{1,2}, Bharat Jayaraman\textsuperscript{2}

\textsuperscript{1}Fiji Systems Inc., 19039 Auten Rd., South Bend, IN, 46637, USA
\textsuperscript{2}University of Buffalo Computer Science and Engineering, 338 Davis Hall, Buffalo, NY, 14260, USA

Received …

KEY WORDS: Real-time Java, Visual Debugging, Query Based Debugging

1. INTRODUCTION

Debugging hard real-time embedded systems is notoriously difficult, as such systems have stringent performance and correctness requirements that often preclude the use of standard debugging techniques for late-cycle debugging. For instance, a standard symbolic debugger, like GDB \[1\] or JDB \[2\], is not well suited for discovering timing errors as the debugger itself can significantly alter the program’s schedule and run time characteristics. As a result, the error the programmer is trying to isolate may not manifest itself in a debugging run or may be incorrectly deduced due to altered timing characteristics.

Debugging via print statements or log dumps, a standard alternative to symbolic debuggers, necessarily entails either extensive I/O or results in information of very low bandwidth. Inexpensive hardware signals (such as the clichéd but still popular blinking light(s)) also suffer from low bandwidth. Due to this cost-to-bandwidth trade-off, pervasive print statement-style debugging is often not realistic, and statements must be inserted and removed as a bug is tracked and eventually isolated.

With these considerations in mind, debugging of real-time embedded systems usually occurs at two granularities: high-level verification through system models and specifications (WCET analysis \[3\], schedulability analysis \[4\], etc.), and low-level debugging strategies. Classic debugging of real-time systems necessitates a low-level approach of print statements and log dumps, requiring hardware support to be effective. This support is realized in the form of specific logic for capturing and filtering traces (logs), as well as buffers to store the trace itself. Unfortunately, such a setup only supports offline debugging via stored hardware traces.

The loss of online symbolic debugging with breakpoints, stop-and-examine capabilities, and direct manipulation of program state due to real-time requirements creates a need for a replacement debugging technology. Pervasive software tracing with predictable performance coupled with offline tools capable of reconstructing detailed, symbolic debugger-style call graphs and symbolic traces can help fill this need. Although many tools exist to help real-time embedded systems developers early in the software life-cycle, for instance to help debug models \[5\], only a handful of software tools exist for late life-cycle debugging \[6\].

In this paper we present the following contributions:

\textsuperscript{*}Correspondence to: Fiji Systems Inc., 19039 Auten Rd., South Bend, IN, 46637, USA. E-mail: elb@fiji-systems.com

Copyright © 0000 John Wiley & Sons, Ltd.
\textit{Prepared using cpeauth.cls} [Version: 2010/05/13 v3.00]
1. An extensible visual debugging tool, Jt.Fi, for Java based, hard real-time embedded systems built on the JIVE [7, 8] debugging platform.
2. A JVM independent, light-weight logging format to gather relevant debugging information for a given execution run. Unlike standard Java debug logging formats, this event log can be gathered in real-time.
3. A real-time aware temporal query processing engine able to answer temporal queries about the execution of the program.
4. An extension to the Jt.Fi system for debugging Safety Critical Java (SCJ) applications.
5. A detailed performance evaluation of the Jt.Fi system.

This paper is an extended version of a work [9] that was previously published in the Proceedings of ACM JTRES 2012. This version includes details on an SCJ-specific extension to Jt.Fi and related evaluation; additional material on queries, including examples and discussion; and extended evaluation of the runtime performance and usefulness of Jt.Fi with additional benchmarks and SCJ programs.

2. MOTIVATING EXAMPLE

Real-time systems are usually concurrent, relying on multiple threads of control. When these threads execute with differing priorities, programs and runtimes must provide assurances against priority inversion, the situation where a low priority thread holds a resource that a high priority thread requires, preventing it from running. We call such assurances priority inversion avoidance protocols. Such protocols, like priority inheritance protocol (PIP) [10, 11], prevent unbounded priority inversion. Using PIP, when a high priority thread attempts to acquire a shared lock held by a low priority thread, the low priority thread is temporarily boosted to the priority of the higher priority thread until it has completed its critical region. PIP prevents unbounded priority inversion by disallowing intermediate priority threads from executing. When the low priority thread has completed, its priority is returned to normal and the high priority thread can acquire the contended resource.

Although detecting and testing for priority inversion in a real-time system is well-understood, understanding the timing effects of priority inversion avoidance mechanisms on a program schedule is more challenging, especially if the thread interactions that lead to triggering of the avoidance protocol are indirect or complex. Further complications can arise due to schedule drift. Quantifying the performance implications of such protocols is important in late-cycle system development. In this stage of system development, such effects are typically quantified through raw timing measurements gathered through specialized hardware provided on development boards and pulled through JTAG interfaces. Processing such dumps is time consuming, error prone and often platform-specific.

Jt.Fi provides a visual mechanism for viewing subtle interactions between threads as well as a temporal query engine to quickly and succinctly discover and display relevant information. To illustrate, consider the example source in Listing 1. In this example, there are two interacting threads, and each thread necessarily blocks the other at some point in the execution. This blocking action creates a priority inversion (HelloThread#main() blocks on HT#run() by way of synchronized(l)). The behavior of this listing will be examined in detail throughout this paper. By default, a thread’s priority is set to 1, the lowest user priority in the system†.

†The lowest priority is reserved for the garbage collector if the system is configured to have one. Thus, in Fiji VM the garbage collector, when present, will execute in the slack time of the system.
Listing 1: An interesting example of thread interactions. Error checking has been removed from this example for clarity.

3. JIVE OVERVIEW

JIVE is a visual debugger that represents the execution state and history of a Java program visually by means of extended UML object and sequence diagrams, respectively. In addition to traditional features such as breakpoints, variable inspection, and forward stepping, JIVE also supports advanced features such as dynamic visualizations of executions, query-based debugging, reverse stepping, and selective tracing. Declarative queries work in synergy with object and sequence diagrams to achieve scalable visualizations: queries help to focus on specific regions of the diagrams, while diagrams provide a framework for reporting query answers and rich visual context for their interpretation. Sequence diagrams provide a timeline that is especially useful for reporting answers to ‘when’ queries. Queries can also be formulated through a high-level template interface.
Figure 1. JIVE user interface showing source code, object and sequence diagrams for the program in Listing 1.
3.1. Sequence Diagrams

As noted above, JIVE captures run time interactions among objects through its sequence diagram, which consists of life lines placed horizontally across the top of the diagram and activation boxes arranged vertically along these life lines. Each life line represents one run time object and is labeled with the object’s class name and instance number. Each activation box represents the execution of one method in the context of the run time object of the corresponding life line. The color of the activation box indicates the thread on which the method call executes. Method calls and returns are depicted as solid and dashed arrows, respectively. Each call arrow is labeled with the name of the called method and an invocation number. A temporal context is also provided, in the form of a dashed horizontal line running across the diagram. The position of the temporal context, which can be controlled by the user, indicates the state of execution currently represented by the object diagram. Many of these features can be observed on the sequence diagrams of Fig. 1. Because sequence diagrams can become unwieldy for modestly sized programs, JIVE supports a number of techniques to reduce the amount of information displayed by the diagram. For instance, horizontal folding hides all nested activation boxes of a given activation, allowing users to focus on the high-level meaning of the folded activation rather than on its internal behavior. We present additional details on Sequence Diagrams in Section 4.3.

3.2. Object Diagrams

JIVE’s object diagram represents a single state of execution and visually depicts object-oriented and run time concepts, such as inheritance, objects and classes with their members and associations, method invocations within the object/class contexts on which calls are made, chaining of method invocations within threads in order to represent call stacks, etc. Additionally, object diagrams are automatically laid out and support multiple modes of visualization to control the amount of detail displayed. For example, member tables may be suppressed (as shown in Fig. 1); only objects with outstanding calls may be displayed; or all objects may be minimized. JIVE also supports user-directed filtering of method calls, hence only those that are not filtered out (‘in-model’ calls) are represented in the diagram. Despite filtering, object diagrams can still grow unwieldy and it is desirable to reduce the diagram further. Hence, we introduce a mode of visualization for object diagrams in which the diagram is continually focused on the active call stacks: objects serving as context for some outstanding method activation are visible while others are fully collapsed. We present additional details on Object Diagrams in Section 4.3.

3.3. Query-based Debugging

JIVE supports eight different kinds of form-based queries: Variable Changed, Method Called, Method Returned, Object Created, Class Invariant, Exception Thrown, Exception Caught, and Line Executed. In order to execute a form-based query, the user provides one or more required parameters and submits the form. For instance, in the Variable Changed query, the user must provide a variable name, a relational operator, and a value. (Following Eclipse’s convention, the query interface can be brought up by selecting, e.g., a variable name, on the editor window and pressing Ctrl+H.) The class name, instance number, and method name parameters, which further determine the context in which the searched variable change occurs, are all optional. After a query is executed, JIVE displays each answer as a row in the search results window, marks all query answers in the sequence diagram as red dots (see Fig. 1), and collapses all regions of the sequence diagram not related to the query answers. This provides users with a reduced view of the sequence diagram, precisely focused on the query answers. JIVE’s query interface also supports PRACTQL queries, a temporal variant of SQL. Temporal queries are formulated using a high-level point-based temporal database schema and efficiently evaluated using an off-the-shelf relational database. This allows more experienced programmers to formulate customized temporal queries when standard form-based queries are insufficient. The benefits of point-based temporal query languages as well as the design and implementation of the PRACTQL query language are discussed in detail in reference [12].
3.4. Reverse Stepping

JIVE uses the recorded trace in order to support stepping in both forward and backward directions. A toolbar on top of both object and sequence diagrams provides temporal navigation through execution states. After every step, JIVE performs a number of tasks: first, it updates the temporal context of the sequence diagram; second, it reconstructs the object diagram so that it displays the correct state of execution at the updated temporal context; and third, it synchronizes the source window in order to indicate the source line corresponding to the current state, thus providing source-level explanation for the observed state changes. The sequence diagram also serves as a temporal navigation tool: by selecting any activation box, the user can jump to any event associated with that activation or any of its child activations and, just like with the temporal navigation provided by the toolbar, the object diagram and source window are updated accordingly.

3.5. Debugging and Events

The debugger part of JIVE is implemented on top of the Java Platform Debugger Architecture (JPDA) [13], an event-based debugging architecture where debugger and debugee tiers run in separate Java Virtual Machines (JVMs). The debugger front-end and back-end communicate using the Java Debug Wire Protocol (JDWP) and the debugger front-end communicates with JIVE using the Java Debug Interface (JDI) [14]. The types of event requests supported by JDI are: virtual machine start, death, and disconnect; class prepare and unload; thread start and death; method entry and exit; field access and modification; exception; and step.

JIVE’s overall implementation is based on a model-view-controller architecture, the main components of which are illustrated in Fig. 2. JIVE’s controller has three modules: an event handler, a data model manager, and a UI engine. The event handler requests events from JPDA and processes event notifications received from JPDA. The event handler is capable of inferring additional event types not directly supported by JPDA, such as local variable changes. The data model manager receives events from the event handler and triggers appropriate model changes. Finally, the UI engine uses the models to update the object and sequence diagrams.

4. THE J.I.FI SYSTEM

J.I.FI builds on top of the JIVE visualization functionality and temporal database described in Section 3 to produce an extensible visual test and debug system for hard real-time applications.
Targeting hard real-time systems necessitates restricting the analysis to offline debugging only. Additionally, the granularity of events in the standard JIVE model (originally designed for the Java JDI) is not well suited for real-time systems. Lastly, JIVE’s native events do not support the notion of real-time, only of logical time. Jl.Fi extends JIVE into a full-fledged offline debugging system for real-time programs. The JIVE side of Jl.Fi consists of an extended temporal model supporting real-time events and states and a number of real-time visual aids. The Jl.Fi temporal database requires additional relations but the underlying PRACTQL [12] query engine remains unchanged. The Fiji VM side of Jl.Fi consists of a fast and predictable logging infrastructure exposing a useful amount of information to the JIVE models with low overhead and little impact on predictability.

4.1. Real-Time Events

JIVE relies on JPDA to obtain a stream of JDI events from a running application, from which it then derives a stream of JIVE events. The granularity of JIVE events is quite fine, allowing for precise online debugging of Java applications. This, however, is not well suited for real-time application due to its overhead. Jl.Fi utilizes a minimalistic approach for events and by design, does not support online debugging. Instead, events are logged, while still preserving real-time constraints, for offline debugging through visualizations and queries.

4.1.1. Basic Events

Five basic types of event are emitted, with each type having a number of subtypes that clarify the activity that triggered the log message. These types are:

1. **VM events**: init, shutdown
2. **Method events**: call, entry, exit, inline entry & exit
3. **Monitor events**: lock and unlock (fast & slow), wait, notify
4. **Thread events**: create, run, yield, sleep, priority change
5. **Exception events**: throw, unroll, catch, termination

For each type of event, the granularity is broken down as much as necessary to convey the critical concept and no more. For example, lock and unlock fast paths are single events, but lock and unlock slow paths are multiple events. The slow path Monitor Lock Begin event indicates that a thread is attempting to enter a monitor, while the Monitor Lock End event indicates that it has successfully done so. These two events may be separated by an arbitrarily large number of events and an arbitrarily long period of time if the monitor is currently held by another thread. This division is not required for fast path monitor log entries, because they are emitted only when there is no contention for the monitor.

In addition to type and subtype, events carry event-specific data clarifying the objects or methods to which they pertain. For example, monitor events carry an identifier for the monitor being manipulated. Method events include the method being invoked. Some event types, such as thread events, carry different information on nearly every subtype — in the case of threads, this ranges from a numeric thread ID to a priority to a real-time timestamp.

In comparison to traditional JDI events, some information is lost in the name of predictability and performance, particularly in the event-specific data attached to events. For example, method events do not indicate the specific object on which they operate. This is a conscious trade-off on verbosity (and thus predictability) versus utility, with verbosity concerns relating to both the size of individual events and the number of events emitted. Maintaining live object state would require a large number of additional events, as well as instrumentation of code that may not otherwise require instrumentation, and the addition of an extra field to method log messages. When one takes into consideration the fact that the live objects in a real-time system are often carefully accounted for in the name of predictability, the utility of this data is reduced in comparison to the cost of maintaining the state.

4.1.2. Safety Critical Java (SCJ) Event Extensions

We have also implemented an extension to the Jl.Fi system described in [9] for debugging SCJ applications. For this extension, the basic events described above are extended with additional SCJ-specific and other runtime events necessary to
make deductions about SCJ-related behaviors. The SCJ-specific events revolve around the periodic nature of SCJ missions and the semantics of SCJ memory management. The related runtime events are object allocations and reference stores to object fields and array slots, which are necessary for the debugging of scoped memory.

1. **Scope events:** alloc, free, push, pop, enter
2. **Allocation events:** object, array
3. **Reference events:** put field, put static, array store
4. **SCJ events:** begin mission, handler deadlines, new cycle, handler release

### 4.2. Temporal Queries Against the Real-Time Model

J1.Ft supports temporal queries to assist in debugging hard real-time applications. Users formulate temporal queries either directly, using the PRACTQL query language, or through one of the pre-built template queries exposed as fill-in forms. Formulating temporal queries manually can be quite complicated, as some of our examples below will illustrate. By exposing high-level template queries, J1.Ft *hides* this complexity from the user for common debugging questions. Filling in a form requires the user to indicate, for example, which threads and monitors they are interested in and the timings of interest (i.e., total duration of the execution, a given period or set of periods, or specific wall clock times). After the user fills in the form and runs a form-based query, J1.Ft generates the corresponding PRACTQL query and sends it to the temporal temporal database for execution.

The high-level template queries provided by J1.Ft allow the user to ask the following questions:

- Which threads trigger priority inversion avoidance protocol, over which monitors?
- Which monitors are contended, and by which threads?
- Which threads miss their deadline or overrun their period?
- What is the scope stack (dynamic allocation context) at allocation of a given object?
- What is the scope stack for a given thread, and in what scope is it currently executing?
- What handlers were triggered in a given period?
- What is the space usage in a scope at a given time?
- Were there any assignments that violated SCJ scoping rules?

In designing the J1.Ft temporal database, an important decision was determining which relations, if any, the database would materialize. This task, commonly known as *view materialization*, can be quite expensive and may degrade database performance significantly if not properly planned and implemented [15]. In J1.Ft, once the temporal database is loaded, it is no longer modified. Hence, the cost of view materialization is incurred only at load time while its benefits are observed repeatedly in both query formulation and evaluation. The benefit of materialized relations is illustrated through the “monitor contended” query.

#### 4.2.1. Monitor Contended Query

At a high level, this query provides to the programmer a set of monitors that were contended during an execution. We define a *contended monitor* to be a monitor $m$ which a thread $b$ attempts to acquire at a point in time when $m$ is held by a thread $h \neq b$. Specifically, this query returns a relation `contend(h, b, m, time)` where each tuple indicates that a thread $h$ causes another thread $b$ to block on monitor $m$ at the given time.

```sql
1 SELECT ml.threadId AS h, m2.threadId AS b,
   2 ml.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
```
m1.threadId <> m2.threadId AND m1.time < m2.time AND m2.time < m3.time AND
-- m1 is LOCK END/FAST LOCK
m1.kind IN (12, 13) AND
-- m2 is LOCK BEGIN
m2.kind = 11 AND
-- m3 is UNLOCK END/FAST UNLOCK
m3.kind IN (15, 16) AND
NOT EXISTS (SELECT 1
FROM event NATURAL JOIN event_monitor mx
WHERE mx.kind IN (15,16) AND
m1.monitor = mx.monitor AND
m1.threadId = mx.threadId AND
m1.time < mx.time AND mx.time < m3.time);

Listing 2: Monitor Contended Using Event Relations.

The query presented in Listing 2 uses low-level event relations event(time, kind, threadId) and event_monitor(time, monitor). Lines 3–5 join each pair of event and event_monitor relations on their time fields. The more complex part of the query is the anti-semijoin in lines 19–24, which guarantees that no monitor unlock event occurs after m1 and before m3 on the same monitor and thread as those of m1. This means that m2’s thread was effectively blocked until (at least) m3.

Another version of the Monitor Contended query is presented in Listing 3. This version takes advantage of the materialized monitor(monitor, holder, lockCount, time) relation. This is an abstract point-based temporal relation [16] that represents the state of monitors for every instant of execution. Internally, the temporal database stores these relations efficiently as concrete interval-based relations. The abstract-to-concrete mapping is performed transparently by the PRACTQL query engine [12].

The query in Listing 3 is much simpler to formulate and understand. It is interpreted as follows: a Monitor Lock Begin event happens at an instant in which the monitor is held (i.e., lockCount > 0) by a different thread. The natural join in the query is used to guarantee that monitor state tuples match the respective monitor and time fields of the event and event_monitor tuples. Hence, the monitor lock event effectively represents the instant in which the event’s thread blocks.

In addition to being much simpler to formulate and understand, the second query should be significantly more efficient than the first one in most temporal database instances. This is due to the smaller number of relations referenced in the query (three) and the absence of anti-semijoins. The first query references six relations in the FROM clause and two more in the anti-semijoin subquery. Furthermore, since the anti-semijoin subquery is correlated to the outer query via tuple variables m1 and m3, it cannot be factored out by the query engine and must be re-evaluated for every potential result tuple of the outer query. Depending on the size of the temporal database and the query plan selected by the underlying query engine, the difference in performance between these queries may be of orders of magnitude.
4.2.2. Priority Inversion Avoidance Protocol Queries  Priority inversion occurs when a low priority thread $t_1$ holds a lock on a monitor $m$ for a period of time during which another thread $t_2$ with higher priority than $t_1$ tries to acquire a lock on $m$ and is blocked. Typically this results in a boost of the priority of $t_1$ to the priority of $t_2$, to prevent unbounded priority inversion by a third thread $t_3$ of medium priority. This boosting is a priority inversion avoidance protocol called priority inheritance protocol, or PIP. Two time intervals are of interest — the priority boost time, indicating the duration of $t_1$’s execution with a boosted priority, and the acquisition delay time, indicating how long $t_2$ waited until it finally acquired the lock on $m$. Notice that the triggering of a priority inversion avoidance protocol, in the case of PIP, will only occur on monitors that are contended. We can thus start formulating queries about priority inversion avoidance protocols. In this section, we assume that priority changes only occur as a result of a priority inversion avoidance protocol in order to simplify the queries for presentation. In systems where such an assumption is not reasonable, we can predicate the queries below over the results of the contended monitor queries presented in Section 4.2.1. By first discovering contended monitors, we can distinguish between priority changes that occur in the application code or as a result of other mechanisms from priority changes that result from the attempted acquisition of a contended monitor.

To simplify query formulation, we first create a view that detects the start of a priority inversion avoidance protocol. The view, shown in Listing 4, associates a thread priority change on a thread with the closest (in time) preceding monitor lock begin event on the same thread. The join in lines 6–8 guarantees that the monitor lock event precedes the priority change event and that the thread that locks the monitor is the one boosted by the priority change event. The $\text{MAX}$ aggregate in line 4 guarantees that only the largest time of all monitor lock begin events satisfying the join and where conditions is returned by the query.

```
CREATE VIEW PI_START AS
SELECT em.monitor, em.threadId AS boosted,
    ep.threadId AS booster,
    MAX(em.time) AS fromTime,
    ep.time AS priority_change_time
FROM event_thread_priority ep
JOIN event_monitor em
ON em.time < ep.time
    AND em.threadId = ep.targetId
WHERE -- em is LOCK BEGIN, ep is PRIORITY CHANGE
    em.kind = 11 AND ep.kind = 24
GROUP BY em.monitor, em.threadId, ep.targetId, ep.time;
```

Listing 4: Protocol Inversion Start.

The actual query, shown in Listing 5, uses the view defined above to associate the start of a priority inversion avoidance protocol with the closest (in time) monitor unlock end (or unlock complete) event on the same monitor and thread as the ones in the start of the priority inversion. The join in lines 6–8 guarantees that the priority inversion start precedes the monitor unlock event and that the thread that unlocks the monitor is the one boosted by the priority inversion start. The $\text{MIN}$ aggregate in line 2 guarantees that only the smallest time of all monitor unlock events satisfying the join and where conditions is returned by the query.

```
SELECT p.monitor, p.booster, p.boosted,
    p.fromTime, MIN(em.time) AS toTime
FROM event_monitor em
JOIN PI_START p ON em.monitor = p.monitor
    AND em.time > p.priority_change_time
    AND em.threadId = p.boosted
WHERE -- em is UNLOCK END or UNLOCK COMPLETE
    em.kind IN (16, 17)
```

Copyright c⃝ 0000 John Wiley & Sons, Ltd.
Concurrency Computat.: Pract. Exper. (0000) DOI: 10.1002/cpe
GROUP BY
   p.monitor, p.booster, p.boosted, p.fromTime


We observe the non-trivial nature of this query. The GROUP BY clause in the query groups on p.fromTime, which is a field defined by the view in Listing 4 as the result of an aggregate. This means that the use of a separate query to compute this field is not only convenient, it is necessary since SQL does not support aggregates on GROUP BY.

As with the “contended monitor” query, priority inversion can take advantage of materialized views in order to simplify query formulation and improve query evaluation performance. The abstract relation threadPriority(booster, boosted, priority, time) as well as the previously introduced monitor relation are both used in the query of Listing 6. The query returns monitors that had priorities boosted, the respective booster and boosted threads, and the time during which priority was boosted. The temporal equijoin guarantees that only the times during which priority was boosted are returned.

```
SELECT m.monitor, tp.booster, tp.boosted, m.time
FROM monitor m
JOIN threadPriority tp ON tp.time = m.time
   AND tp.boosted = m.holder

Listing 6: Monitor Contended Using State Relations.
```

In comparison with the event-based query of Listing 5, the state-based query of Listing 6 is not only easier to formulate, but likely several orders of magnitude faster. The reason is that the state-based version is a simple equijoin involving two relations, which can be well optimized by the underlying query engine, while evaluation of the event-based query must serialize the evaluation of the aggregate defined in PI.START, the query join, and the query aggregate. Each of these must wait for the previous one to complete before it can be correctly evaluated.

4.2.3. Deadline Miss and Overrun Detection

The query in Listing 7 uses the SCJ extensions to J.F.I to return all thread periods whose deadlines were either met or overrun at runtime (subquery in line 1), together with those whose deadlines were missed (subquery in lines 3–7). Both subqueries reference view ThreadSchedules (Listing 8) that returns, for each thread, the time points of schedules whose deadlines were either met or overrun at runtime. The first subquery is thus straightforward. The second one computes the temporal complement of ThreadSchedules in order to return the missed schedules. It relies on the built-in True(time) temporal relation, that returns all time points in the interval $(-\infty, \infty)$. The first part of the subquery (lines 3–5) returns all time points in each thread’s execution while the second (lines 6–7) subtracts all time points corresponding to schedules whose deadlines were met or overrun. Hence, only time points corresponding to deadline misses are returned by this subquery.

```
SELECT threadId, status, time FROM ThreadPeriods
UNION
(SELECT tp.threadId, "MISSED" AS status, t.time AS period
   FROM True t, ThreadPeriods tp
   WHERE tp.threadStart <= t.time AND t.time <= tp.threadEnd
   EXCEPT SELECT threadId, "MISSED" AS status, time
   FROM ThreadPeriods)

Listing 7: Deadline Misses and Overruns
```

Table I illustrates the answers to this query. The query was formulated using the form-based interface and the answers were restricted to interval (0, 60) and thread T1 (period is 10ms). Fig. 3
Table I. Results from Listing 7.

<table>
<thead>
<tr>
<th>threadId</th>
<th>status</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>ON-TIME</td>
<td>(0 – 10)</td>
</tr>
<tr>
<td>T1</td>
<td>MISSED</td>
<td>[10 – 40)</td>
</tr>
<tr>
<td>T1</td>
<td>OVERRUN</td>
<td>[40 – 50)</td>
</tr>
<tr>
<td>T1</td>
<td>OVERRUN</td>
<td>[50 – 60)</td>
</tr>
</tbody>
</table>

Table I. Results from Listing 7.

Figure 3. Listing 7 Illustrated.

Illustrates graphically how the query is processed (top to bottom) from the raw data given at the top of the figure. The following line represents the materialized view ThreadSchedules, followed by both subqueries (lines 3–5 and lines 6–7 respectively), and finally computed deadline misses at the bottom given in blue.

Factoring out commonly used subqueries as views provides both engineering and potential performance benefits. Queries can be formulated at a higher abstraction level, with views encapsulating much of the query’s complexity. This approach yields queries that are smaller, more readable, and easier to maintain. Furthermore, frequently used views can be transparently materialized, improving the performance of all dependent queries and views in the system. For instance, the ThreadSchedules view is likely to be used across many queries and is thus a strong candidate for materialization.

```
1  CREATE VIEW ThreadSchedules AS
2      SELECT threadId, threadStart, threadEnd, status, time
3      FROM True t, ThreadScheduleEndpoints
4      WHERE periodStart <= time AND time < periodEnd;

Listing 8: ThreadSchedules
```

The ThreadSchedules view relies on yet another view, ThreadScheduleEndpoints (Listing 9), that returns the end points and status of every thread schedule observed at runtime, and the periods in which such threads were supposed to execute. This view is joined with the True relation in ThreadSchedules in order to return all time points in the periods of such schedules.

The ThreadScheduleEndpoints view uses event based relations only. It matches every thread start event with the corresponding thread end event (lines 16–17). Then, for each thread, it matches each
schedule start with every subsequent schedule event end in the same thread (lines 18–21). In order to
match each schedule start event with the correct schedule end event, the query returns the minimum
schedule end time (line 10) in every group (lines 22–23). Finally, to determine the status of the
schedule, the query checks whether the results of two integer divisions are equal. Intuitively, if the
schedule start and end times fall into the same period then the schedule was on-time for that period.
We observe that the period of a schedule is determined based on the start time of the schedule.

```
CREATE VIEW ThreadScheduleEndpoints
SELECT
  ts.threadId, ts.time AS threadStart, te.time AS threadEnd,
  ((tss.time-ts.time)/ts.periodLength) * ts.periodLength AS periodStart,
  (1+(tss.time-ts.time)/ts.periodLength) * ts.periodLength AS periodEnd,
  ts.time AS scheduleStart,
  MIN(tse.time) AS scheduleEnd,
  CASE WHEN (tss.time-ts.time)/ts.periodLength =
    (MIN(tse.time)-ts.time)/ts.periodLength
    THEN "ON-TIME" ELSE "OVERRUN" END AS status
FROM
  event_thread_start ts
  JOIN event_thread_end te ON te.threadId = ts.threadId
  JOIN event_thread_schedule_start tss ON tss.threadId = ts.threadId
  JOIN event_thread_schedule_end tse ON
tse.threadId = t.threadId AND tse.time > tss.start
GROUP BY
  ts.threadId, ts.periodLength, ts.time, te.time, tss.time
```
Listing 9: ThreadScheduleEndpoints

4.2.4. Scope Assignment Violations The next query is SCJ-specific and detects scope assignment
violations. A scope assignment event occurs in JI.FI every time a field assignment or array slot
assignment is detected in the event log. JI.FI stores both the LHS and RHS object references, their
allocation scopes, and their respective indexes in the scope stack at the time the event is processed.

```
SELECT
  time, scopeLHS, indexLHS, LHS, scopeRHS, indexRHS, RHS
FROM
  event_scopeassign
WHERE
  rhs <> 0 AND indexLHS < indexRHS
```
Listing 10: Scope Assignment Violations

The query in Listing 10 references the event_scopeassign event relation and returns all scope
assignments such that: 1) the RHS of the assignment was not NULL (rhs != 0) AND 2) the LHS and
RHS objects were allocated on memory scopes M and M' and, at the time of the assignment, the
index of M on the stack was smaller than the index of M' (indexLHS < indexRHS). This means that
the lifespan of scope M is strictly greater than that of scope M'.
Figure 4. Jt.Ft object diagram for the program in Listing 1. The main thread, shown in violet, and the user thread, shown in green, have outstanding calls to `Object.wait()` and `Thread.sleep(long, int)` respectively. This diagram illustrates the methods called by the threads in the boxes located next to the thread objects.
4.3. Real-Time Visual Aids

Being built on top of the JIVE visualization engine, J.I.Ft retains all of JIVE’s configurability, including the ability for the user to collapse and expand diagram boxes, pick representative colors, and quickly show linked diagrams. J.I.Ft renders slightly modified versions of the Object and Sequence diagrams provided by JIVE. The reason for this is that the basic real-time event log does not contain a number of events normally used by JIVE to render its diagrams. For instance, object creation and field/local variable modifications are not included by default in the Fiji VM log. Thus, object environments and field values cannot be represented in the J.I.Ft Object Diagram. Thread objects are the exception to this rule, since their creation and changes to some of their instance fields can be inferred from low-level events. Other than that, only static environments and their fields are typically represented in the J.I.Ft Object Diagram. However, static field values remain unknown (<undefined>) for the duration of execution. Fig. 4 illustrates a J.I.Ft Object Diagram for our running example. In this diagram the three thread objects represent the thread instances which are running, two for the computation threads and one for the garbage collector. Each contour representation in the diagram is a composite of nested contours, one for the object type and one for each class the object inherits from. This allows data and methods to be displayed in the right context.

Since J.I.Ft without extensions is unaware of the existence of objects other than threads, each method call in a diagram is associated with its declaring class environment instead of the actual object context on which the method was called. This essentially clusters method calls within the appropriate class. The effect on the Object Diagram is that activation records are always placed within class environments. On the Sequence Diagram, method calls are always placed under the class life lines. Thread creation is the only exception—a life line for each thread object is created by J.I.Ft and the call to their constructors is placed on the respective life line. Fig. 5 presents a J.I.Ft Sequence Diagram for our running example. For simplicity Thread lifelines are omitted in the Diagram. J.I.Ft allows for easy “collapsing” of information, allowing the user focus on relevant information. For instance, notice that there is no start method call displayed. This occurs because the thread lifeline has been collapsed in the diagram. In the diagram “Thread: 3” is a lifeline for a Thread instance object.

While these adapted diagrams are not specialized real-time visualizations, they provide useful auxiliary information for the analysis of such programs. A new visualization provided by J.I.Ft is the Thread State Diagram. This is a diagram specialized for real-time programs the purpose of which is to help users visualize interactions among the threads running in their programs. In particular, these diagrams can help identify unexpected or suspiciously long waits and blocks on user threads. J.I.Ft allows the user to select which threads and time intervals to display. The default behavior is to show all threads for the duration of the execution.

The Thread State Diagram of Fig. 6 shows the interaction between the main thread and the user thread of our running example. The main thread starts in the ready state and transitions into the running state so quickly that it is not visible in the diagram. This is quite different from the user thread, which remains in the ready state for many clock cycles before transitioning into the running state when the main thread calls its start() method. As soon as the main thread invokes wait() on the b object, it relinquishes its lock on this object’s monitor and then transitions into a wait state. Meanwhile, the user thread acquires locks on the monitors for objects l and b, in this order. Once it holds a lock on b, it calls notify() on this object, effectively waking up the main thread. Finally, the user thread sleeps for 50ms, time during which it still holds a lock on l. The main thread tries to lock object l but remains blocked while the user thread is sleeping. When the user thread wakes up, it releases its lock on b and the main thread eventually unblocks and completes execution.

In the SCJ extensions for J.I.Ft, we introduce a scope aware Object Diagram. This extension of the J.I.Ft Object Diagram uses additional events, including object allocations, to visually represent the allocation context of each object within an SCJ program execution. We represent the scopes as bounded blocks, consisting of the objects allocated in them. Fig. 7 shows a stylized scope aware Object Diagram generated for the SCJ versions of the jPapaBench benchmark. The figure depicts an illegal SCJ scope assignment in jPapaBench [17] in which the object PapaBenchUtils lives in the immortal memory and refers to an object LogUtilsImpl present in Mission Memory (which has a
Figure 5. JiFi sequence diagram clarifying thread states for the program in Listing 1.
shorter lifetime than Immortal memory). This reference assignment violates SCJ scoping rules and could lead to dereference of an invalid object pointer. More details on this bug follow, in Section 5.

4.4. Event Log Realization in Fiji VM

The Fiji VM real-time event logging infrastructure is carefully designed to log as predictably as possible. Each thread logs to a thread-local buffer (requiring no locking), and hands that buffer, via an extremely rapid handshake requiring only a few machine instructions in the critical section, to a low-priority thread dedicated to log flushing. Both the thread-local buffer and individual log messages are of a constant size known at compile time, so buffer checks and message writes are very fast. Many parts of each individual log message are also compile-time constants, and the logging code is aggressively inlined. The total code for logging an event on the fast path (when a handshake with the log flushing thread is not required) is on the order of a dozen machine instructions.

Log entries are structured to fall on natural word-size boundaries and to be as compact as practically possible. Listing 11 shows the C structure for a bare log event. The event consists of a type, subtype, thread ID, Java nanosecond timestamp, and 64 bits of event-specific data. Note that the largest word in the event is 64 bits long, and that the complete log message is three 64 bit words in length, with three bit-aligned shorter words packed into the first of these 64 bit words.

Structuring the logging path for minimum handshaking and run time processing overhead, with fixed-size buffers handed off to a log flushing thread, leads to on-disk logs that are not in strict
chronological order. Log entries for any given thread are in strict chronological order, but buffers for different threads may arrive out of order depending on buffering delays and run time behaviors. The real-time timestamp present on each log entry is sufficient to preserve total ordering for single core systems, but true total ordering ambiguity may arise in multi-core systems. We believe that this trade-off is justified in order to reduce logging overhead. In many circumstances where total ordering is critical to log coherence, it may be derived from the semantic meaning of the log entries themselves. For instance, if two threads attempt to enter a monitor at the same instant on separate cores, only one thread will succeed immediately and the other will block; the event emitted by the thread that succeeded immediately occurred logically before the event emitted by the thread that blocked. A coherent partial ordering can be easily computed offline when processing the logs.

Some log events are inserted into the VM runtime source code at critical points, such as the thread creation, destruction, and priority change events; system startup and shutdown events; and monitor events. Other events are inserted by the Fiji VM compiler into user or VM code at compile time. When the logging infrastructure is disabled at compile time, all of the code and tracking structures are elided entirely from the compiled code via macro expansions or compiler omission. Only code of particular interest is instrumented at compile time; by default, this means that user-provided code is instrumented, while the VM and Java standard libraries are not.

Method calls represent an interesting case, because they are both very common and very performance-sensitive. They have the added complexity of inline, static, and dynamic dispatches. For these reasons, method logging is inserted by the Fiji VM compiler at compile time.

Method instrumentation is performed after all optimization passes, including, critically, all inlining. Once the Fiji VM compiler has completed optimization and inlining, an additional code pass iterates over those user-provided methods with remaining non-inline invocations and inserts a Method Enter event at the beginning of each method with matching Method Exit events for every return from any method. VM-internal methods and the standard libraries are not instrumented. This process is relatively straightforward.

A more interesting operation on non-inline methods is a limited emission of information on non-instrumented methods. While instrumenting (for example) java.lang.String would lead to a lot of instrumentation that is likely uninteresting to the developer, instrumenting that the developer’s code directly invoked a method on a java.lang.String object is not. Therefore, method invocations are examined, and any point where an instrumented method invokes a non-instrumented method has event emissions inserted around the invocation, so that the log is as if the non-instrumented method logged its own invocation but none of the methods it invoked in turn — without incurring logging cost at every invocation.

The final method instrumentation is for inlined methods. Each inlined method has method entry and exit log events inserted into its inlined code, as for non-inlined methods, but in this case the nested hierarchy of inline invocations must be tracked in order to determine which inline methods should or should not be logged. The same rules are applied as for non-inlined methods, though the implementation differs somewhat. Instrumented methods emit log events, and non-instrumented methods invoked directly by instrumented code emit log events. Non-instrumented methods inlined into non-instrumented methods that are in turn inlined into instrumented code are not emitted.
makes inline method events behave just as in standard static dispatch, except that the inline method events are flagged as having been inlined by the compiler.

The necessity of compact log entries precludes storing certain information in the entries themselves. For example, Java method signatures can be hundreds of characters long, but method enter events indicate the method being entered. To handle this and other, similar cases, a unique integer ID is assigned to items of interest (such as methods that may be invoked at run time, in the case of method enter events) at compile time, and the mapping between these items and their unique IDs is recorded by the compiler.

The basic logging mechanism uses 24 byte event structures, as illustrated in Listing 11. However, extensions may require more space for event-specific data than this 24-byte structure provides. The object allocation and reference-tracking events from the SCJ extension presented here, specifically, require three or more words of data, and do not fit in the 8 bytes of event-specific space of the existing structure even on 32-bit systems. Therefore, an optional 32 byte log structure was added to the Fiji VM to make room for this additional data. Its layout is depicted in Fig. 8. The first 24 bytes of this structure are the log structure described above by Listing 11. The additional 8 byte word at the end of the structure may be included optionally at compile time if the additional details provided in allocation or reference events are required for the debugging task at hand.

It is critical to performance and predictability that as much of the log event structure as possible is constant at compile time if possible or, where compile time is not possible but run time is, at run time. The Type and Subtype fields are always compile-time constant, and the Thread ID and Timestamp fields are always dynamically computed at run time. The event-specific data fields vary by event type, but are kept constant wherever possible. In addition, where multiple values are stored in one 32- or 64-bit word, constants are grouped with constants if possible, to minimize word manipulation overhead.

4.5. JIVE Event Log

The JIVE event log differs substantially from the Fiji VM event log in both format and semantics. The JI.FI event log is a modified JIVE event log, sharing gross structure and semantics with the JIVE log. JI.FI logs are stored and exchanged in XML. A sample log entry for the beginning of a thread execution can be seen in Fig. 12. Note the real-time-specific fields in this log event, such as the absolute timestamp (JIVE logs include only the id field, which provides a logical clock time) and the real-time scheduler used for the thread in question (FIFO, in this example).

A translation tool is used to unpack the binary logs written by the VM and described in Section 4.4 into the XML logs processed by JI.FI. In addition to simple translation (for example, the event
displayed in Fig. 12 requires no additional information other than that stored in the Fiji VM event log, and can be translated directly), this translation tool draws in information produced by the compiler at compile time and state developed at processing time to produce a more complete XML log. As previously mentioned, this includes the mapping between method signatures and unique IDs used to represent methods in the binary log. It also includes maintenance of the method call stack and thread states. Some Fiji VM events may map to multiple Jt.FI events (e.g., a single Fiji VM method entry event emits two Jt.FI log events), or even none at all (as is the case for some VM setup events that are emitted before the Jt.FI model begins), so there is not a tight correspondence between the quantity of Fiji VM log events and Jt.FI log events. It turns out that there are typically many more Jt.FI log events than Fiji VM log events due to inflation caused by the aforementioned method entry translation.

During this translation process, the partially ordered event stream stored in the Fiji VM log is turned into a totally ordered Jt.FI event stream. Ordering disambiguation is currently handled entirely by the real-time timestamps on log entries, as this work focuses on single processor machines where timestamp ambiguity is not possible.

After translation, Jt.FI imports the Jt.VE-format event log, performing semantic checks at that time. The Jt.FI event log can be validated for syntax using an XML schema, but the semantics of the log are difficult to describe in an XML schema language as they must represent the complexities of Java VM semantics. The import process therefore validates each incoming event against the Jt.FI internal model as it is inserted, ensuring that the model integrity is not violated — for example, that method calls and monitor operations balance.

5. APPLICATION OF JI.FI FOR BUG DISCOVERY

Our tool provides an automated and visual mechanism for discovering and debugging errors. Using these visualizations and temporal queries we were able to find bugs in jPapaBench [17] as well as the open source SCJ (oSCJ) libraries.

5.1. Bug Discovery in jPapabench

The jPapaBench [17] benchmark is derived from a core set of the Paparazzi [18] project, an open-source UAV flight system. The benchmark manifests a scope assignment violation that our tool discovered and depicted visually through the scope aware Object Diagram. Specifically, Fig. 7 shows an object that lives in the immortal memory scope (a memory scope with a lifetime of the entire SCJ application) storing a reference to an object in a scope of lesser lifetime, mission memory (which has the lifespan of an SCJ “mission” — an abstraction for a particular set of related, co-executing real-time tasks). Although in the SCJ level 0 specification a program can contain only
one mission ‡, this mission may be executed multiple times during the life of the system. Such an assignment could therefore manifest as a dangling pointer on the second or subsequent executions of the mission.

```java
public PapaBenchUtils setLogUtils(LogUtils logUtils) {
    this.logUtils = logUtils;
    return this;
}
```

The error in jPapaBench lies in the logging facility, and is shown in the code above. The object `this` is allocated in immortal memory, while the object `logUtils` is allocated in mission memory. Such errors can be easily found via scope checks. However, they can be extremely difficult to isolate and fix as they require reasoning about the dynamic allocation contexts of objects. Moreover, due to SCJ semantics, such information cannot be extracted at the point of error. Specifically, it is very difficult to locate the objects involved in the reference store that violated the SCJ scoping rules. To do so often requires manual instrumentation and re-execution of the code. Complex errors potentially involving multiple reference assignment violations must be fixed one error at a time, as the failing scope checks abort the offending task.

Many systems allow for scope checks to be disabled to improve performance. The jPapaBench build harness does indeed disable scope checks and is likely the reason that the aforementioned bug went uncaught. In the absence of scope checks, scope assignment errors may not reliably manifest as a fault in the system. Indeed, in jPapaBench, since the main mission is only executed once, no fault occurs. If scope checks are disabled, such errors may become extremely difficult to discover. To the best of our knowledge, even exhaustive testing regimes driven by model checkers did not discover this bug §. We note that the discovery of the existence of this particular jPapaBench assignment error, which can be accomplished simply via scope checks, is not as important as the ability it demonstrates to capture salient debugging information to easily and correctly fix the error.

5.2. oSCJ Changes Between Versions

Jt.Fi was also useful in debugging the latest version of oSCJ and its integration with Fiji VM. We used our tool to discover two errors present in oSCJ. Fiji VM and oSCJ make different assumptions about which party is responsible for entering the immortal memory scope at boot time. In previous versions of oSCJ, this was handled by the oSCJ library. In version 2.0 of the library, oSCJ no longer explicitly enters the scope, resulting in objects being allocated on the VM heap. The version of Fiji VM distributed by oSCJ also does not enter the immortal memory. The implication of this bug is that objects may be allocated outside of the space prescribed by the program author, allowing the program to over-allocate the available system memory, potentially leading to application failure. This error can be displayed visually in the Jt.Fi Thread State Diagram as the immortal scope is never entered. Alternatively, the user can query the scope stack for a given object, which will show the incorrect scope stack.

Lastly, In version 2.0 of the oSCJ library, priority parameters are ignored. This results in periodic event handlers executing at the default system priority, regardless of the priority requested by the programmer at initialization time. This may lead to unexpected interruption of the real-time process by other system processes. This error can be displayed visually on the Jt.Fi Thread State Diagram as the thread priority does not change. Alternatively, the user can query the priority for a period event handler, which will show the incorrect priority assignment.

‡Levels 1 and 2 allow for more complex mission structures.
§We applied our tool to the same codebase as the citation, which is freely available at: http://d3s.mff.cuni.cz/~malohlava/projects/jpapabench/.
6. EVALUATION

The real-time executables in this section were run under Linux on a lightly loaded Intel Xeon E5410 CPU, restricted to a single core to preserve high-accuracy in-core processor timings. The executable under test was the highest priority process in the system, and the only process running with real-time privileges. Each benchmark was run ten times and the first run was dropped from any computations. For each benchmark the executions of instrumented and non-instrumented were interleaved.

6.1. General Benchmarks

6.1.1. CDx

CDx, an open source family of benchmarks with identical algorithmic behavior for different hard and soft real-time platforms, was used to evaluate the logging infrastructure’s behavior and performance. While a complete description of CDx is given in [19], we present enough basic information for readers to understand the computation performed in CDx. The evaluation in this paper uses CDj, the pure Java implementation of CDx.

The benchmark is structured around a periodic real-time thread that analyzes simulated radar frames to detect potential aircraft collisions. It can thus be used to measure the time between releases of the periodic task as well as the time it takes to compute the collisions. The need for detection of potential collisions prior to their occurrence makes CDx a hard real-time benchmark, as each frame must be processed prior to a concrete deadline.

The core of CDx is a collision detection algorithm for computing the position of aircraft in a configurably-sized airspace. The algorithm detects a collision whenever the distance between any two aircraft is smaller than a pre-defined proximity radius. The distance is measured from a single point representing an aircraft location. As locations are only known at times when the radar frames are generated, they have to be approximated for the times in between. The approximated trajectory is the shortest path between the known locations. Collision detection is split into two steps. First, the set of all aircraft is reduced to multiple smaller sets. This step allows the computation to quickly rule out aircraft that are too far from each other to collide within the duration of a radar frame. Second, for each smaller set, every pair of aircraft is checked for collision. Both the reduction and the collision check operate on pairs of 3-d vectors describing the initial position, $\vec{i}$, and the final position, $\vec{f}$, of an aircraft ($\vec{i}$ is from the previous frame, $\vec{f}$ is from the current frame). Each aircraft in the frame also contains a call sign that identifies the aircraft. A motion vector $\vec{m}$ is then defined as $\vec{m} = \vec{f} - \vec{i}$. Motion vectors are compared to check for collisions for aircraft that were not ruled out in the first step due to distance.

The benchmark executable was compiled with an un-instrumented Fiji compiler and VM configuration, as well as various levels of instrumentation. All executions of the benchmark used in these comparisons were performed by running the individual configurations ten times, round-robin, in a loop. The results from the first iteration of the loop are discarded. Where average or aggregate results are presented, all iterations other than the first are included. Where the results of a single run are presented, results from the fifth iteration of the loop are used. Correlation coefficients are computed using the Pearson method, as implemented in GNU R.

6.1.2. CDx Performance Results

Low logging overhead and high log performance predictability are critical to the viability of real-time debugging. There are several tunable parameters in the logging infrastructure configuration:

- The number of log events in each per-thread log buffer
- The method of timestamping used for event timestamps
- The quantity of instrumented code
- The event types enabled

Fig. 9(a) depicts the time spent performing detections during each release of the detection thread in CDj with and without logging enabled. The log buffer size for this run is 48 events, and all supported events are emitted for the CDj code. Execution times are in microseconds, and each release represents one of 1,000 releases in a single execution. Fig. 9(b) shows an arbitrary window
Table II. Performance of various log buffer sizes in entries, relative to 48-entry buffers. Note that the steps are logarithmic.

<table>
<thead>
<tr>
<th>Buffer Size</th>
<th>Relative Cost</th>
<th>Buffer Size</th>
<th>Relative Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>1.06</td>
<td>256</td>
<td>0.823</td>
</tr>
<tr>
<td>48</td>
<td>1.0</td>
<td>512</td>
<td>0.804</td>
</tr>
<tr>
<td>64</td>
<td>0.947</td>
<td>1024</td>
<td>0.793</td>
</tr>
<tr>
<td>128</td>
<td>0.866</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

of only 50 releases, so that the correspondence between individual releases with and without logging may be more easily seen.

The performance impact of this level of logging is significant, taking 2.5 times as long to complete each release as compared to the log-free code. However, it provides complete method logging for the user-provided code with little memory overhead (just under 1.5 kB per thread for event buffers, dependent on how often the log flushing thread can service its queue).

Increasing the size of the per-thread log buffer reduces the computation overhead of handshaking with the log flushing mechanism, but increases the memory overhead for per-thread log buffers. Table II shows the relative difference in computation overhead for various log sizes compared to the 48 entry buffer in Fig. 9(a). Each step in buffer size achieves over 5% gains in performance from 48 to 128 buffers, and nearly 5% from 128 to 256, after which the benefits decrease dramatically. Significant benefits are realized by the time the buffer reaches 64 entries in size, making this a reasonable buffer size for systems with moderate memory constraints. For systems with more memory available, the 128 and 256 event buffers fit on one or two 4 kB pages, respectively, which may be attractive for page-based allocation mechanisms. The relative impact of buffer size has proven to vary across processor architectures, but similar marked increases in performance persist as the buffer sizes increase up to 256 entries or more.

The full log of 1,000 releases of CDj from Fig. 9(a) contains about 9.26 M events, of which about 9.21 M are method events. The common Java idiom of providing objects with getters and setters, methods which perform no calculation but merely store or retrieve object state, leads to a potentially large number of method calls for which the call logging is disproportionately expensive compared to the method itself. The Fiji VM compiler employs a metric for inlining, commonly implemented in optimizing compilers, where the code size of a method is estimated and compared to the code size for a method call site. If the method body is smaller than a call site, it is always inlined unless inlining is entirely disabled. By leveraging this optimizer calculation in reverse, the Fiji VM compiler can omit logging instrumentation for most getters and setters, by eliding the instrumentation for inlined methods smaller than a call site, without losing substantial amounts of valuable information from the log. (Note that this optimization does not necessarily capture only getters and setters, but it does by...
definition capture only trivial methods.) Applying this logging optimization to the 256-entry buffer executable from Table II, which displayed a mean release duration of 245,199 ns, the number of log events emitted in a full run of 1,000 releases is reduced to roughly 4.62 M (4.57 M method events) and the mean release duration to 171,328 ns. This represents an overhead of only 81% compared to non-instrumented code, and a 50% reduction in logging overhead, in return for a negligible loss of useful information.

At least as important as the absolute overhead of logging in this context is the predictability of the logging infrastructure. In that respect, the Fiji VM implementation of the J1.F1 log structure is more than successful. The split nature of log generation and output, coupled with careful design to keep most log event data compile-time static or available via a trivial lookup, leads to a crisp linear scaling of computational overhead when logging is enabled. Looking at 1,000 releases of a typical CDj execution, the correlation coefficient for instrumented versus non-instrumented code ranges from 0.91 to 0.94 across several executions of the same executable; for comparison, correlation among executions of the non-instrumented code ranges from 0.93 to 0.96.

Fig. 10 shows a graphical representation of the correlation between the non-instrumented and instrumented versions of a particular run of CDj. The instrumented code uses 256-entry log buffers and omits very small methods. Note the linear structure of the point cloud. The horizontal axis represents the duration of a given release in the non-instrumented code, while the vertical axis represents the duration of the same release in the instrumented code. Each point represents a single release in a run of 1,000 releases. Therefore, the vertical spread for each point on the horizontal axis is the system jitter around a release of a given duration plus the instrumentation jitter.

Note that the 81% overhead figure for nearly complete method flow, monitor, and thread activity is an upper bound. The implementation evaluated here is carefully designed for predictability and performance in the large, but has not been optimized in the small, a task which may provide substantial benefits. In particular, the entire logging infrastructure uses only high-level C and Java implementation, and employs no platform-specific code. (The one possible exception to this statement, depending on how one views it, is timestamping of log messages; this uses an internal Fiji VM function that reduces to the x86 RDTSC hardware timestamp instruction on the platform under test.) Careful platform-specific fine-tuning of log entry creation to take advantage of processor strengths (particularly where 64-bit data values are concerned) may yield higher performance. Additionally, while the J1.F1 platform is designed to require no hardware assistance, the complete lack of hardware assistance is a “worst case” scenario. On platforms with fast JTAG communication, high speed bus-connected storage, hardware logging assistance, or other such features, performance may be substantially improved.
6.2. SCJ Benchmarks

We leverage a number of existing SCJ benchmarks, including SCJ versions of the jPapaBench and CDx benchmarks described above, to evaluate our tool. We evaluate several different jPapaBench flight plans, ranging in duration from about 30 seconds to 89 minutes. We have also tested our framework on the SCJ benchmarks distributed with the oSCJ library, including a graph algorithm, linked list sorting, fast-MD5, and a precision memory test. We note that the performance numbers over these smaller benchmarks were compatible with the results from CDx and jPapaBench and we focus our discussion on the latter.

We focus our detailed performance evaluation on the duration of specific real-time tasks within the larger benchmarks. We present results over a wide variety of tasks, including: a) real-world tasks extracted from jPapaBench, b) synthetic computation in a real-time setting extracted from CDx, as well as c) a non real-time synthetic MD5 kernel used for correctness testing of real-time virtual machines. The first set of timings, taken from jPapaBench, represents the most “realistic” load and behaves like a typical real-time application. The second two sets of timings are worst-case micro benchmarks to illustrate the potential impact of instrumentation and logging. All benchmarks were executed completely and fully instrumented. All overhead measurements are for releases of individual tasks within the benchmark.

For the SCJ results presented here, unless otherwise noted, we configure the logging infrastructure with the recommended configuration from Section 6.1: 256 log entries in each per-thread logging buffer, to emit many events only for user-provided code, and to omit method events for very small methods (such as getters and setters). All event types are enabled, which maximizes debugging information at the cost of increased computation and I/O overhead.

6.2.1. The jPapaBench Real-Time Application

Fig. 11 visually depicts the correlation between runtime performance in instrumented and non-instrumented code for 32 byte log events using the logging configuration described above in the same manner as 10. Each point on the graph represents a single release of the jPapaBench Autopilot Navigation task performing the “Round Trip” flight plan. This task is the most computationally intensive task in the jPapaBench task set, although it only runs 1/10 as often as some of the less intensive tasks. In the selected flight plan, it runs a total of 936 times over about 18 minutes of simulated flight time. From the structure of this point cloud, we can infer that the overhead of instrumentation is reasonably predictable. In concrete terms, the average computational duration of an instrumented release in this data set is 1.17x the un-instrumented duration, with a standard deviation of 0.06, and the correlation coefficient is 0.77. To put these variations in perspective, the period of the Navigation task is over 1 s while the vertical extent in Fig. 11 is 60 µs, many orders of magnitude smaller.
An alternate view of a portion of the data in Fig. 11 is shown in Fig. 12. This view shows 100 sequential releases of the Navigation task, with a separate curve for the fully instrumented and non-instrumented code. On this plot, the correspondence between the duration of releases for the non-instrumented and instrumented codebases is clear, with higher release times in the non-instrumented code corresponding to higher release times in the instrumented code. Several releases of the instrumented code can be seen to take disproportionately longer than their non-instrumented counterparts; for example, the double spike between releases 260 and 280. This is to be expected in certain circumstances where an uncommon code path in the instrumented code triggers a large number of events, possibly also incurring the overhead of additional log buffer handoffs. Deviations such as these account for the vertical spread in Figures 11 and 13.

6.2.2. The CDj Micro-Benchmark  Fig. 13 shows the same information as Fig. 11 for 1,000 releases of the CDj benchmark. For this plot, the average computational overhead of a CDj task release is about 3.86x with a standard deviation of 0.16, and the correlation coefficient is 0.90. As in the previous example, the variation in runtimes is several orders of magnitude shorter than the CDj task release period, which is 50 ms. The thread-local log buffers were increased to 1,024 entries for this plot, as the dense log emissions of CDj were causing handshaking overhead to increase. (256-entry log buffers displayed an overhead of about 4.4x.)

Note that the dense computation kernel of the CDj application leads to a significantly higher overhead than the jPapaBench tasks. This is due to correspondingly denser reference manipulations.
Table III. Benchmark performance and predictability.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>24 byte</th>
<th>32 byte</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ovhd</td>
<td>Corr</td>
</tr>
<tr>
<td>PB Navigation RT</td>
<td>1.15</td>
<td>0.79</td>
</tr>
<tr>
<td>PB Navigation FP4</td>
<td>1.19</td>
<td>0.92</td>
</tr>
<tr>
<td>CDj</td>
<td>3.53</td>
<td>0.95</td>
</tr>
<tr>
<td>Fast-MD5</td>
<td>3.30</td>
<td>0.06</td>
</tr>
</tbody>
</table>

Figure 14. 100 individual releases of the Fast-MD5 benchmark.

during its calculation. We note that the user may choose to disable the expensive allocation and reference store events, reducing the performance overhead to roughly that found in Section 6.1, if they are unnecessary for a given debugging task. The scope- and SCJ-specific events are far fewer than the object and reference events, and have a much lesser impact on overhead.

6.3. Overview of SCJ Benchmark Results

Table III displays the overhead and correlation coefficient metrics for a representative sample of the benchmarks on which we evaluated the SCJ logging extensions. The table abbreviations are as follows: PB is jPapaBench, RT is the Round Trip flight plan, and FP4 is Simple Flight Plan 04. The jPapaBench results use 256-entry thread-local log buffers, and the other benchmarks use 1,024-entry buffers. Results for both 24 byte and 32 byte logs are presented, and from this it can be seen that the smaller log entries, as expected, generally have both lower overhead and more predictable performance when compared to the expanded log entries. However, like the expense of the new events, the cost of these expanded log entries is offset by their utility in debugging. Comparison of the same task with different work loads (PB Navigation RT and PB Navigation FP4) shows that the overhead is similar between runs, and that the correlation between non-instrumented and instrumented runs remains good across work loads. The Fast-MD5 example is interesting because the release of the MD5 task has very little variation in release timing (mean 92,712 ns, standard deviation 1,268 ns), so the relatively small, but larger, variations in the instrumented code (mean 311,150 ns, standard deviation 10,793 ns) show up as very poor correlation. Fig. 14 shows a snapshot of 100 releases of this benchmark in the same structure as Fig. 12.

6.4. JIVE

In this section we outline JIVE performance measures and show that our prototype also has acceptable performance from a usability standpoint.
6.4.1. Log Import Loading a Fiji VM real-time event log into J1.F1 is a straightforward process. First, a Java project is created. The source code from which the log was created can be optionally added to the project, in which case J1.F1 will use it during the log import to obtain the meta-data associated with the imported events. Finally, a new offline launch configuration is created and configured for the project. This is a new feature in J1.F1 which allows a Java project to be associated with an external log file. Once the offline launch configuration is executed, the log file is loaded, internal models created, and all visualizations rendered. The current prototype is capable of loading and processing the logs, represented in an XML interchange format, at a rate of approximately 12MB/sec. For instance, a two iteration run of the CDj benchmark results in a 24.1 MB log file with 87 K events (the raw Fiji VM log file has only about 57 K events, some synthetic events are generated in the translation process). J1.F1 loads this file and renders all diagrams in 2.1sec.

6.4.2. Temporal Database Export In order to export the internal models as a temporal database, J1.F1 provides a command line interface to the export feature. The actual export uses a JDBC connection to a PostgreSQL database in order to load a model consisting of both static (e.g., meta-data) and dynamic information (e.g., events and states). The temporal database resulting from the CDj benchmark contains about 330 K tuples which are exported to the database in approximately 16.4 sec, with an average throughput of 19.8 K tuples/sec. The relatively slow database export is due to two main factors — first, the export is I/O bound, and second, every method call in the event log incurs additional costs at the database: namely, the creation and/or update of environment records and their respective members. These are essential for rendering Object and Sequence diagrams.

6.4.3. Temporal Query Evaluation We expect most queries formulated by J1.F1 users to be either temporal select-project-join (SPJ) queries or non-temporal aggregate and set/bag queries. This class of queries can be used to answer questions such as those posed in section 4.2 and many other, more complex, queries. The advantage of such queries is that they are translated by the PRACTQL query engine into standard SQL queries with a slightly modified SELECT and WHERE clauses, but no additional table references. The implication is that such queries will be optimized by the underlying SQL query engine (e.g., PostgreSQL) as well as any other SQL query in a typical database application.

For queries involving temporal aggregation, temporal set/bag operations, and temporal grouping, the performance question is still an open problem. Preliminary results obtained in the course of the development of the PRACTQL query engine suggest that the performance of such queries is just slightly inferior to their non-temporal counterparts for most cases, degrading to a worst-case quadratic performance when the structure of the temporal data is shaped as a large triangle (i.e., a set of temporal tuples with common values for their non-temporal attributes and, for every pair of tuples, the interval of one is strictly contained in the interval of the other). A full discussion of the issues involved in the performance of this kind of temporal queries as well as queries involving temporal recursion, however, is outside the scope of this paper.

7. RELATED WORK

Our work spans multiple domains, including testing and debugging techniques, visual debugging, databases, and real-time and embedded systems. We compare our systems to state of the art techniques in these various fields below. Our discussion is organized by field.

7.1. Program Debugging

We first provide an overview of dynamic techniques for program debugging and then briefly review static and model checking techniques.

Dynamic analysis is concerned with properties of a running program [20]. The landscape of dynamic analysis is extremely rich and includes techniques such as query-based, omniscient, and statistical debugging. We discuss query-based debugging in the following subsection.
Breadcrumbs [21] is a deployable dynamic tool that combines static analysis and lightweight instrumentation for precise error detection based on probabilistic calling contexts. DIDUCE [22] detects errors and identifies their root causes by dynamically formulating and updating hypothesis about likely program invariants. PQL [23] allows users to attach actions to subquery matches, enabling a simple approach to dynamic error recovery and defensive actions against security violations. DIDUCE, starting from stricter invariants and gradually relaxing them as violations are detected, in order to allow for new behavior. CCured [24] is a hybrid static and dynamic analysis tool that performs pointer analysis in C code; pointer usage is separated into those that can be verified statically and those that require instrumentation to be verified dynamically (i.e., guarded by runtime checks).

The Smalltalk debugger in [25] performs instrumentation at the virtual machine level, extends objects so they store their own histories, and rely on garbage collection to keep only reachable objects (and their histories) in memory. The Trace-Oriented Debugger [26] features stepping, state and control flow reconstitution, non-declarative queries (based on cursor and count primitives), and dynamic visualizations in the form of murals [27]. Unstuck [28] is a Smalltalk debugger that supports searches based on Smalltalk expressions and provides low-level textual views of method traces, objects, and object histories. ODB [29] is a proof-of-concept omniscient debugger for Java supporting: navigation through a previously recorded execution trace both forward and backward in time; low-level textual views of method traces, objects, stacks, and local variables; and, simple pattern-matching queries.

Holmes [30] isolates bugs by finding path profiles that are highly indicative of error; previously reported bug reports and static analysis are used to selectively instrument the program’s code. BayesDebug [31] attempts to simulate manual debugging principles from a Bayesian perspective, by assuming that only one passing and one failing test case are available. Sober [31] models evaluation patterns of predicates in both correct and incorrect runs, and identifies a predicate as bug-relevant if its evaluation patterns in correct and incorrect runs differ significantly. Liblit et al. [32] use low-overhead Bernoulli sampling for gathering execution information from multiple production sites. Their analysis identifies trends that correlate predicate violations with increased likelihood of failure.

7.2. Static and Model Checking Techniques

A number of tools explore static analysis and model checking techniques for bug detection. Many recent tools implement frameworks for programmers to write their own static checkers (e.g., Meta Compilation [33, 34], Findbugs [35], bdddbbd [36], SATURN [37]). Other tools implement specific static checks, such as Chord [38] and PREfix [39]. Test generation tools often take hybrid approaches; for instance, Agitator [40] combines static and dynamic techniques while Klee [41], CUTE [42], and Korat [43] combine static analysis and model checking. Model checking systems such as BLAST [44], SLAM [45], and Java Pathfinder [46, 47] verify whether a model of a program satisfies some form of specification: BLAST verifies temporal safety properties of C programs, SLAM verifies whether a program (e.g., windows device driver) adheres to API usage rules (e.g., windows kernel API), and Java Pathfinder performs explicit-state model checking of Java programs using both symbolic and concrete execution to identify run-time exceptions, concurrency problems, and user-specified property violations.

7.3. Query-Based Debugging

Query-based debugging was first proposed by Lencevicius et al [48]. In their approach, a query is formulated in a procedural style in the implementation language itself and run against current objects in the running program’s heap. However, there is no support for querying past program state or program control flow. A more recent tool is Whyline [49], an interrogative debugger supporting ‘why did’ and ‘why did not’ queries about program executions. It works on recorded, rather than live, executions and therefore online debugging is not supported. Whyline does not expose a query language. PTQL [50] is a relational query language with SQL-like syntax designed to query program traces online via instrumented code. Given a Java program and a PTQL query,
a compiler instruments the Java code so that the PTQL query is executed on-line. There is no support for temporal queries and query results are presented in textual form. The Trace-Oriented Debugger [26], TOD, is a scalable omniscient debugger featuring both query-based debugging and dynamic visualizations. It uses bytecode instrumentation to generate events which are recorded to a specialized database. TOD’s querying capabilities are encapsulated as high-level features: stepping, state and control flow reconstitution, and simple when/where queries over variables. These features are based on two low-level primitives, cursor and count. TOD provides high-level visualizations in the form of murals [27], which are graphs showing the evolution of event density for a given class of events over time.

7.4. Dynamic Visualizations

Recent literature provides extensive coverage of dynamic program visualization tools and techniques: [51] provides a systematic survey of program understanding through dynamic analysis and uncovers research opportunities in the area; [52] classifies a large number of dynamic visualization tools according to desirable features; [53] is a software visualization book covering static and dynamic visualizations; [54] survey eight trace exploration tools with emphasis on trace modeling, diagram abstraction, and diagram scalability with respect to navigation and size; [55] describes a case study of dynamic visualization tools and their effectiveness in providing answers to large- and small-scale program comprehension questions. A complementary overview of some dynamic visualization tools is presented next.

Ovation [56] visualizes execution traces using an execution pattern view, a form of interaction diagram that depicts program behavior. Diagrams support a number of operations such as collapsing, expanding, filtering, and execution pattern detection (e.g., repetition). Amida [57] extracts sequence diagrams from program traces and applies a dominance algorithm in order to detect and remove local objects contributing to internal behavior of dominator objects. Amida processes traces offline while JIVE displays sequence diagrams during program tracing. TPTP [58] is primarily concerned with collecting profiling data, but can represent executions as a sequence diagrams, interactively. It supports filtering and hiding methods and objects, as well as collapsing call trees. However, the latter case is not automatic. Program Explorer [59] uses merging and filtering to reduce the size of its object and interaction graphs. Programs are visualized interactively and their execution traces viewed as interaction charts which are similar to sequence diagrams. ISVis [27] uses static and dynamic analyses to construct message flow diagrams similar to sequence diagrams. These diagrams represent interaction patterns in the trace. A global view of the execution is displayed in its execution mural.

In [60] tracing is done over intervals and visualizations consist of box displays containing various statistics relevant to the particular box. The Collaboration Browser [61] recovers object collaborations from execution traces based on user-defined criteria to match and combine similar collaborations instances into higher-level collaboration patterns. Jinsight [62] provides dynamic views for detecting execution bottlenecks (Histogram View), displaying execution sequences (Execution View), showing interconnections among objects based on pattern recognition algorithms (Reference Pattern View), and displaying profiling information for method calls (Call Tree View).

One of the most popular pedagogic IDEs, BlueJ [63], is an integrated development environment (IDE) for Java specifically designed for introductory programming courses using an objects-first approach. However, BlueJ does not show the dynamic object structure nor call/execution histories. A lightweight IDE providing visualizations for a variety of data structures, jGRASP [64] generates views automatically and updates them dynamically as users step through the code. jGRASP does not feature dynamic visualizations of call/execution histories, and the dynamic visualizations of execution state focus on individual objects and, therefore, do not provide a global view of execution state. A program visualization tool that animates program executions and supports the learning process is ViLLE [65]. On the other hand, it lacks two key features: support for object-oriented programming languages and dynamic visualizations of execution state and call/execution histories.
7.5. Debugging Real-Time Embedded Systems

During the course of its development, a real-time embedded system undergoes a continuous validation and verification regimen. Part of this regimen is schedulability analysis [4], which proves whether or not the tasks that comprise a given system can meet their deadlines in the system as a whole. Recent work on multi-criteria schedulability analysis [66] has been applied for performance debugging of task models. We believe JI.FI can be leveraged to empirically validate schedules as well as identify profitable uses of slack time to reduce priority inversion avoidance overheads.

The closet work to JI.FI is TunningFork [67, 68], aimed at debugging the timing behavior and interaction of components of real-time systems. TunningFork, like JI.FI, produces interactive visualizations to aid in system comprehension. Unlike JI.FI, whose focus is on debugging within a given software component, TunningFork focuses on visualizing the interactions of multiple independently developed sub-components running on a network of servers. We believe that JI.FI could be extended to aid in cross component debugging.

AFTER [6] is a tool used in the latter stages of the software development cycle for fine-tuning a real-time embedded system to meet its timing requirements. AFTER leverages raw timing data and correlates it to a timing specification to produce a temporal image of the current system timing results. Currently JI.FI does not support testing and debugging using a system specification or schedulability analysis directly. JI.FI can provide salient information about timing analysis through the use of temporal queries. We can envision parameterizing JI.FI with a system specification to augment JI.FI’s capabilities to include automated prediction to help tune a system for its timing requirements.

Debugging real-time distributed systems typically involves light-weight replay mechanisms [69]. To achieve a replay mechanism that is light-weight, such schemes only log a sub-set of all events. The JI.FI logging infrastructure realized in Fiji VM [70, 71] employs a similar technique of emitting only key events to reduce overhead. Instead of a replay mechanism, JI.FI provides powerful temporal queries that can correlate temporally disparate events.

Many real-time systems are verified with respect to either a structural or behavioral specification. Recent work on real-time logic (RTL) [72] has provided mechanisms for systematic debugging based on incremental satisfiability counting for systems that use a behavioral specification. We envision extending JI.FI to automatically synthesize relevant temporal queries based on safety assertions provided as a part of a behavioral specification. Such a specification can also be leveraged to determine the granularity of logging and the definition of relevant events.

As the real-time community revises standards for certification, formal methods will become more widely adopted. We do not envision formal methods replacing testing and debugging tools but being used to augment them. Already the community is exploring different formal methods. For instance, static checking of SCJ compliance has been proposed and implemented through programmer annotations [73]. We believe such techniques will compliment our debugging framework.

8. CONCLUSIONS AND FUTURE WORK

We presented JI.FI, a framework for testing and debugging hard real-time embedded systems. We believe that JI.FI will prove invaluable for understanding and debugging thread interactions and use of memory areas. Our results indicate that the logging required for useful visualizations and real-time temporal queries can be accomplished in real-time without affecting the deadlines of the application when taken into account during schedulability analysis. We presented a number of real-time and SCJ-specific visualizations and temporal queries to aid in testing and debugging of such systems. We currently plan to extend our tool with support for the Real-Time Specification for Java (RTSJ) and add hardware based acceleration to the Fiji VM logging infrastructure to leverage known techniques for hardware-based log gathering. Moreover, to extend JI.FI to be parameterized by either a behavioral or structural specification to guide user queries and automated log verification and validation.
REFERENCES


