JI.FI : Visual Test and Debug Queries for Hard Real-Time

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

Fiji Systems Inc.[1]
{elb, luke}@fiji-systems.com

SUNY Buffalo[2]
{dlessa, lziarek, bharat}@buffalo.edu

ABSTRACT

Hard real-time systems have stringent timing and resource requirements. As such, debugging and tracing such systems often requires low-level hardware support, and online debugging is usually precluded entirely. In other areas, visual debugging has greatly improved program understanding and late cycle development times for non real-time applications. In this paper we introduce a visual test and debug framework for hard real-time Java applications built around the JIVE platform and realized in the Fiji VM.

Our framework, called JI.FI ['dʒɪfi], provides high-level debugging support over low-level execution traces. JI.FI provides both powerful visualizations and real-time centric temporal query support. To ensure preservation of the real-time characteristics of the application being tested and debugged, JI.FI leverages a real-time event log infrastructure that logs only relevant application and virtual machine level events, such as synchronization and modifications to priorities or thread state. Our performance results indicate that our logging infrastructure is suitable for hard real-time systems, as the performance impact is both *uniform* and *quantifiable*.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging Aids*

General Terms

Design, Measurement, Performance

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 [10] or JDB [11], is not well suited for discovering timing errors as the debugger itself can significantly alter the program's schedule and runtime characteristics. As a result, the error the programmer is trying to isolate may not manifest itself in the 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 extensive I/O or results presented 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 [36], schedulability analysis [28], *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 [19], only a handful of tools exist for late life-cycle debugging [5].

In this paper we present the following contributions:

1. A visual debugging tool, JI.FI, for Java based, hard real-time embedded systems on the JIVE [14, 13] 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, the 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. A detailed performance evaluation of the JI.FI system.

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 proto-*

JTRES 2012 October 24-26, 2012, Copenhagen, Denmark

```
1  class HelloThread {
2      public static int[] l = { 0 };
3      public static int[] b = { 0 };
4
5      private static class HT extends Thread {
6          public void run() {
7              System.out.println("Hello world!");
8              synchronized (l) {
9                  synchronized (b) {
10                     b.notify();
11                 }
12                 Thread.sleep(50);
13             }
14             System.out.println("Goodbye world!");
15         }
16     }
17
18     public static void main(String args[]) {
19         HT t = new HT();
20         Thread.currentThread().setPriority(5);
21         synchronized (b) {
22             t.start();
23             b.wait();
24         }
25         synchronized (l) {
26             t.join();
27         }
28     }
29 }
```

Listing 1: An interesting example of thread interactions. Error checking has been removed from this example for clarity.

cols. Such protocols, like priority inheritance protocol (PIP) [16, 33], 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.

JI.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.

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 time line that is especially useful for reporting answers to 'when' queries. Queries can also be formulated through a high-level template interface.

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.

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.

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. Af-

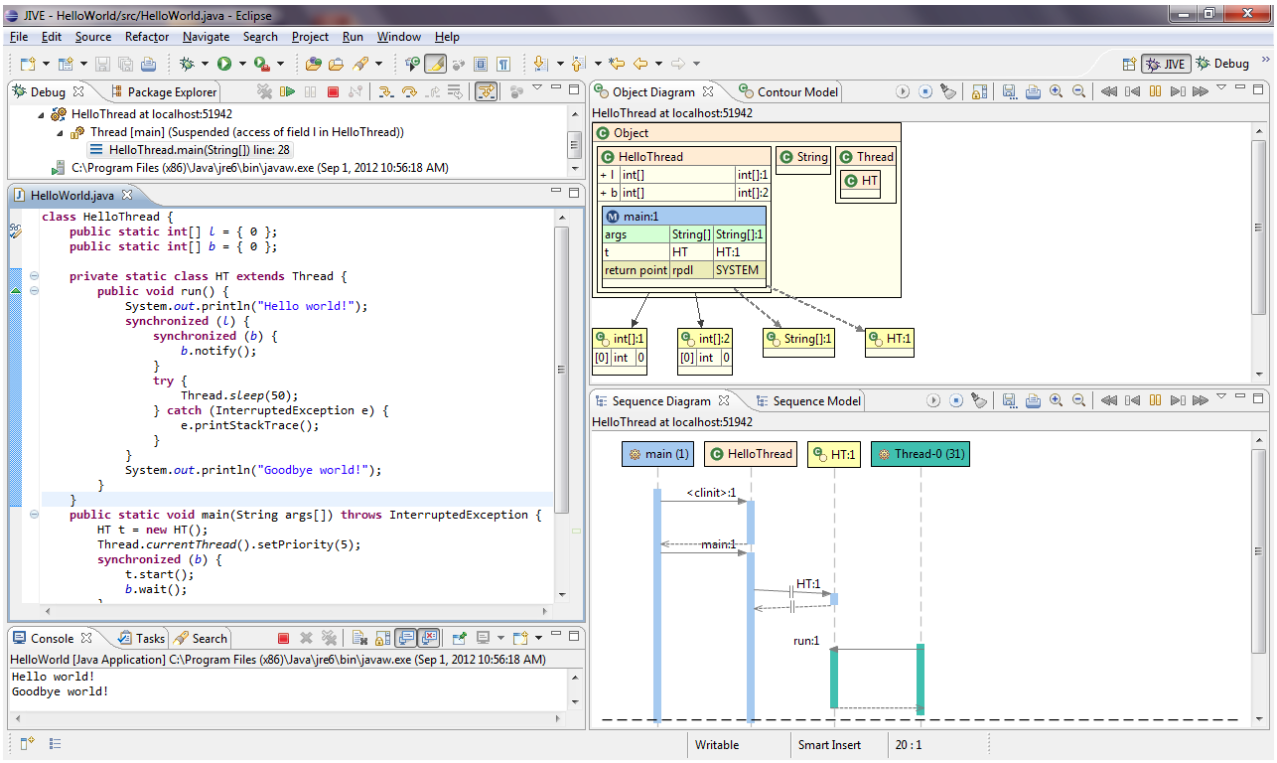

Figure 1: JIVE user interface showing source code, object and sequence diagrams for the program in Listing 1.

ter 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 [27].

Debugging and Events: The debugger part of JIVE is implemented on top of the Java Platform Debugger Architecture (JPDA) [4], an event-based debugging architecture where debugger and debuggee 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) [20]. 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.

4. THE JI.FI SYSTEM

JI.FI builds on top of the JIVE visualization functionality and temporal database described in Section 3 to produce a 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. JI.FI extends JIVE into a full-fledged offline debugging system for real-time programs. The JIVE side of JI.FI consists of an extended temporal model supporting real-time events and states and a number of real-time visual aids. The JI.FI temporal database requires additional relations but the underlying PRACTQL [27] query engine remains unchanged. The Fiji VM side of JI.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 supports events as defined by the JDI as a part of the JPDA. These events are expected to be emitted by the application while it is executing for online debugging. The granularity of events is typically quite fine, allowing for precise debugging of Java applications. This approach, however, is not well suited for real-time application due to its overheads. JI.FI utilizes a more minimalistic approach for events and does not support online debugging. Instead, events are logged, while still preserving real-time constraints, for offline debugging through visualizations and queries.

Five basic types of event are emitted, with each type having a

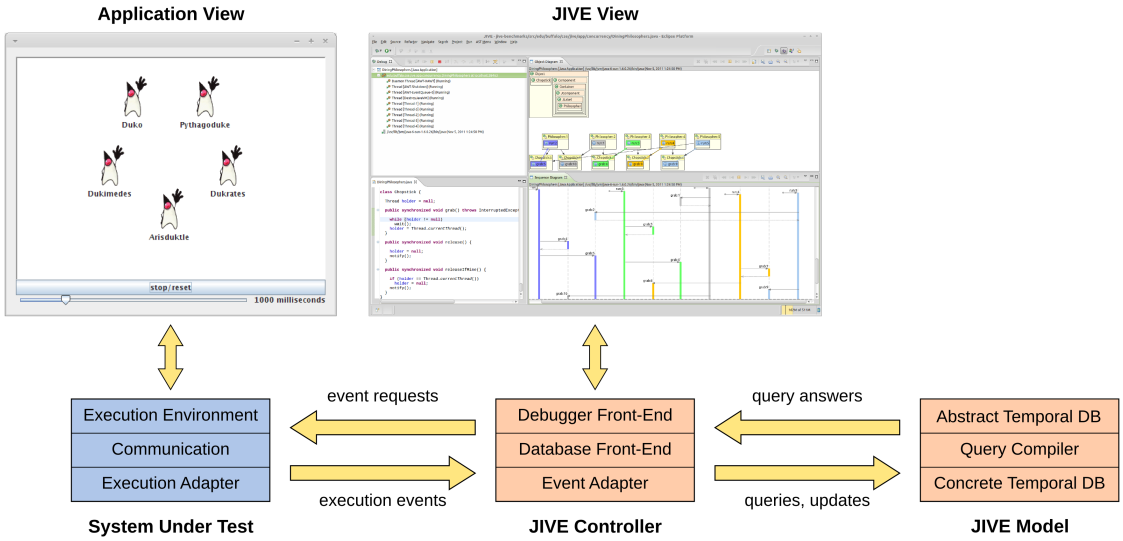

Figure 2: JIVE Model-View-Controller Architecture.

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 distinction 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.

Other Extensions: Although the current log does not contain any specific RTSJ [17] nor SCJ [2] specific events, we envision extending it to do so. Our current real-time events can be leveraged by RTSJ and SCJ implementations for basic logging. However, additional logging events for asynchronous event handling, different thread types, and memory areas are necessary to fully communicate the critical semantics of RTSJ and SCJ. To correctly handle RTSJ and SCJ extensions we envision extending the libraries with appropriate event emitting code.

4.2 Temporal Queries Against the Real-Time Model

JI.FI 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. By exposing high-level template queries, JI.FI *hides* this complexity from the user. After the user runs a form-based query, JI.FI generates the corresponding PRACTQL query and sends it to the temporal temporal database for execution.

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

- On which monitors was a priority inversion avoidance protocol triggered, and on behalf of which threads?
- What is the duration of a priority inversion avoidance protocol invocation?
- Which monitors are contended, and by which threads?
- Are there any deadline misses?

In designing the JI.FI temporal database, an important decision was determining which relations, if any, the database would materialize. This task, commonly known as *view materialization*, is quite expensive and may degrade database performance significantly if not properly planned and implemented [18]. In JI.FI, 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.

```
1  SELECT m1.threadId AS h, m2.threadId AS b,
```

```
2         m1.monitor AS m, m2.time
3  FROM (event NATURAL JOIN event_monitor) m1,
4       (event NATURAL JOIN event_monitor) m2,
5       (event NATURAL JOIN event_monitor) m3
6  WHERE
7    m1.monitor = m2.monitor AND
8    m1.monitor = m3.monitor AND
9    m1.threadId = m3.threadId AND
10   m1.threadId <> m2.threadId AND
11   m1.time < m2.time AND m2.time < m3.time AND
12   -- m1 is LOCK END/FAST LOCK
13   m1.kind IN (12, 13) AND
14   -- m2 is LOCK BEGIN
15   m2.kind = 11 AND
16   -- m3 is UNLOCK END/FAST UNLOCK
17   m3.kind IN (15, 16) AND
18   -- the monitor held by m1 is not released before m3
19   NOT EXISTS (SELECT 1
20       FROM (event NATURAL JOIN event_monitor) mx
21       WHERE mx.kind IN (15,16) AND
22             m1.monitor = mx.monitor AND
23             m1.threadId = mx.threadId AND
24             m1.time < mx.time AND mx.time < m3.time);
```

Listing 2: Monitor Contended Using Event Relations.

The benefit of materialized relations is illustrated through the "monitor contended" query. 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.

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.

The version of the "contended monitor" query presented in Listing 3 takes advantage of the materialized monitor relation. The JI.FI temporal database materializes two state relations: monitor(monitor, holder, lockCount, time) and thread(threadId, state, time). These are *abstract point-based temporal relations* [7] that represent the state of monitors and threads, respectively, for every (relevant) 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 [27].

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

Listing 3: Monitor Contended Using State Relations.

The query in Listing 3 is much simpler to formulate and understand. It is interpreted as follows: a Monitor Lock Begin event happens at the 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. 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.

4.3 Real-Time Visual Aids

JI.FI renders slightly modified versions of the Object and Sequence diagrams provided by JIVE. The reason for this is that the 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 in the Fiji VM log. Thus, object environments and field values cannot be represented in the JI.FI 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 represented in the JI.FI Object Diagram. However, static field values remain unknown ($<?>$) for the duration of execution. Fig. 3 illustrates a JI.FI Object Diagram for our running example.

Since JI.FI 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 JI.FI and the call to their constructors is placed on the respective life line. Fig. 4 presents a JI.FI Sequence Diagram for our running example.

A new visualization provided by JI.FI 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. JI.FI 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. 5 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.

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

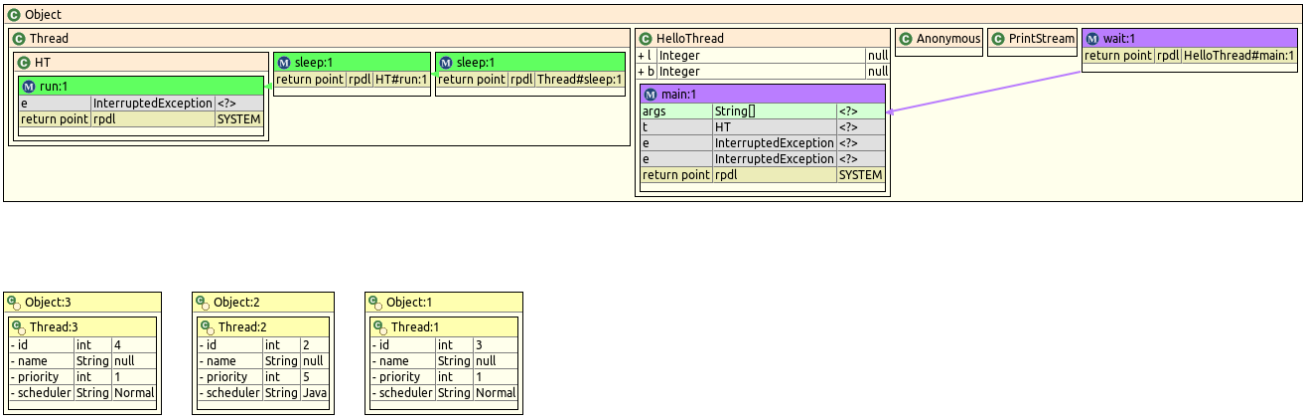

Figure 3: JI.FI object diagram for the program in Listing 1. The main thread (violet) and the user thread (green) have outstanding calls to Object.wait() **and** Thread.sleep(long, int) **respectively.**

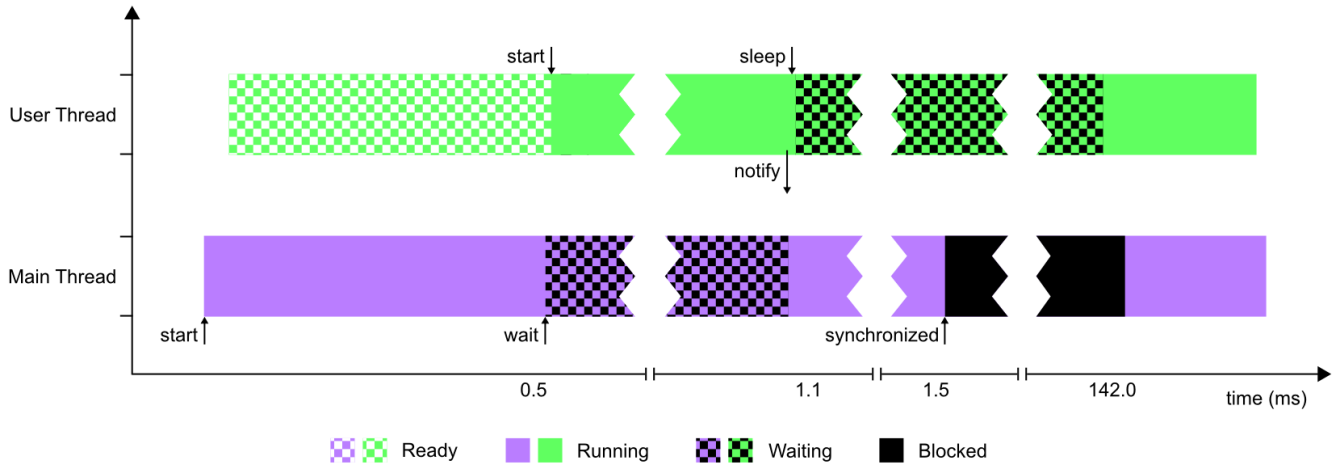

Figure 5: Thread state diagram with state changes in the main thread and the user thread. No state changes occur in the omitted intervals.

```
1  struct fivmr_FlowLogEvent_s {
2      uint16_t type;
3      uint16_t subtype;
4      uint32_t tid;
5      uint64_t timestamp;
6      uint64_t data;
7  };
```

Listing 4: Fiji VM log event structure

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 4 shows the C structure for a 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 runtime 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 may be easily computed offline when processing the logs. If a local log-buffer

overruns, the events that would cause the overrun are lost.

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. This 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.

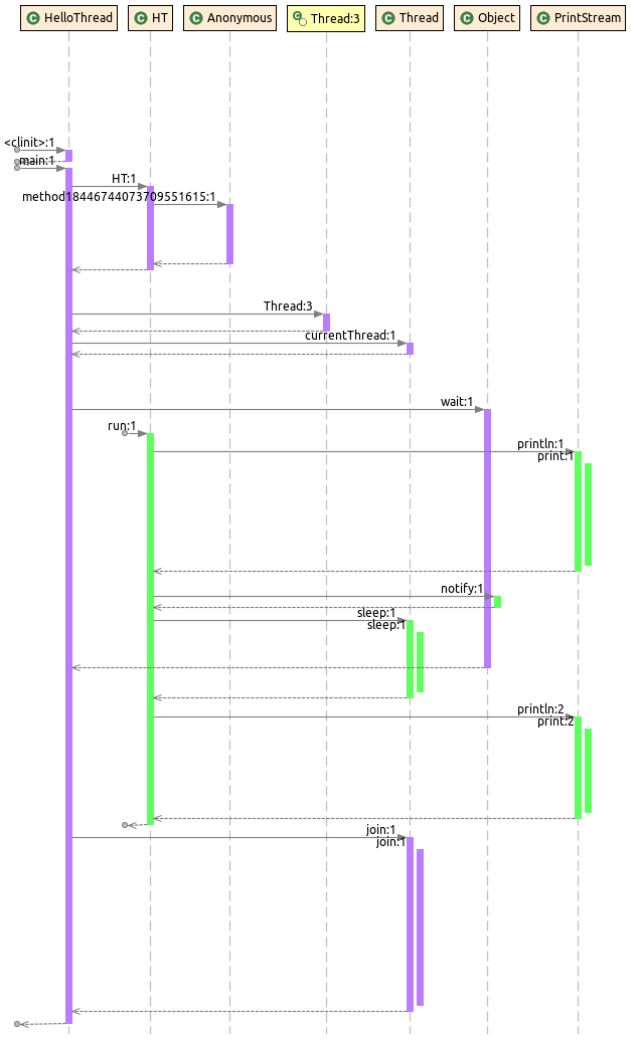

Figure 4: JI.FI sequence diagram clarifying thread states for the program in Listing 1.

5. EVALUATION

The real-time executables in this section were run on a lightly loaded Intel Core i5 2300 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.

The CDx benchmark suite [23] is an open source family of benchmarks with identical algorithmic behavior for different hard and soft real-time platforms. While a complete description is given in [22], we present enough basic information for readers to understand the computation performed in CDx.

The benchmark is structured around a periodic real-time thread that analyzes simulated radar frames to detect potential aircraft collisions. The benchmark 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.

5.1 Logging

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. 6 depicts the time spent performing detections during each release of the detection thread in CDj, the Java version of the CDx benchmark, 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 processor cycles as reported by the processor timestamping instruction, and each release represents one of 1,000 releases in a single execution.

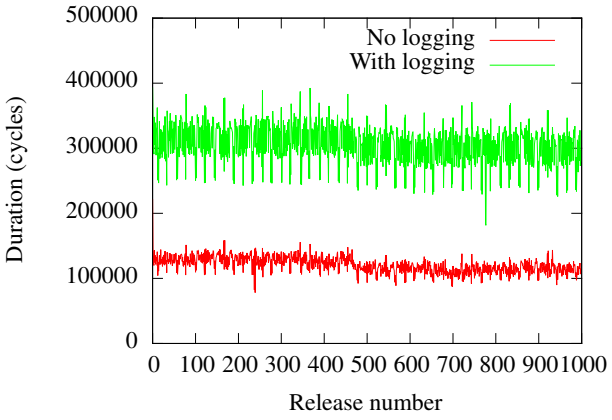

Figure 6: Raw execution time data. Jitter is caused by benchmark behavior.

Buffer Size	Relative Cost	Buffer Size	Relative Cost
32	1.06	256	0.927
48	1.0	512	0.924
64	0.975	1024	0.912
128	0.951		

Table 1: Performance of various log buffer sizes in entries, relative to 48-entry buffers. Note that the steps are logarithmic.

The performance impact of this level of logging is significant, using about 2.5 times as many processor cycles 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, depending 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 1 shows the relative difference in computation overhead for various log sizes, compared to the 48 entry buffer in Fig. 6. Each step in buffer size achieves about 2.5% gains in performance from 48 to 256 buffers, after which the benefits decrease dramatically. The most 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 full log of 1,000 releases of CDj from Fig. 6 contains about 9.27 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. Fiji VM employs a metric for inlining 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, Fiji VM can omit logging instrumentation for most getters and setters, eliding the instrumentation for inlined methods smaller than a call site, without losing substantial amounts of valuable information from the log. Applying this logging optimization to the 256-entry buffer executable from Table 1, which displayed a mean release duration of 282,132 clock cycles, the number of log events emitted in a full run of 1,000 releases is reduced to roughly 4.63 M (4.58 M method events) and the mean release duration to 201,442 clock cycles. This represents an overhead of only 67% 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 JI.FI 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 coefficient of variation of the number of cycles required for each release scale within 1% at various levels of logging, and with logging disabled.

Note that the 67% 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 JI.FI 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.

5.2 JIVE

Log Import: Loading a Fiji VM real-time event log into JI.FI 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 JI.FI will use it during the creation of the temporal database to obtain additional meta-data. Finally, a new offline launch configuration is created and configured for the project. This is a new feature in JI.FI 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). JI.FI loads this file and renders all diagrams in 2.1sec.

Temporal Database Export: In order to export the internal models as a temporal database, JI.FI 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 of environment records and their respective members. These are essential for rendering Object and Sequence diagrams.

Temporal Query Evaluation: We expect most queries formulated by JI.FI 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 sf 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.

6. RELATED WORK

Query-Based Debugging: Query-based debugging was first proposed by Lencevicius *et al* [26]. 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 [24], 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 [15] 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 [31], 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 [21], which are graphs showing the evolution of event density for a given class of events over time.

Dynamic Visualizations: Ovation [9] 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 [35] 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. TPTP [12] 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 [25] 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 [21] 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.

One of the most popular pedagogic IDEs, BlueJ [1], 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 [8] 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 [32]. 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.

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 [28], 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 [6] 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.

AFTER [5] 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 [34]. 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 [29, 30] 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) [3] 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.

7. CONCLUSIONS AND FUTURE WORK

We presented JI.FI, a framework for testing and debugging hard real-time embedded systems. 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 are currently planning on extending the real-time log definition and JI.FI visualization repertoire to include RTSJ and SCJ specific events.

8. REFERENCES

[1] BlueJ– The interactive Java environment.

[2] JSR 302. Safety critical Java technology, 2007.

[3] Stefan Andrei, Albert M. K. Cheng, Wei-Ngan Chin, and Miahi Lupu. Systematic debugging of real-time systems based on incremental satisfiability counting. In *Proceedings of the Real Time on Embedded Technology and Applications Symposium*, RTAS '05, pages 519–528, Washington, DC, USA, 2005. IEEE Computer Society.

[4] Java Platform Debugger Architecture. `http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/`.

[5] Gaurav Arora and David B. Stewart. A tool to assist in fine-tuning and debugging embedded real-time systems. In *Proceedings of the Workshop on Languages, Compilers, and Tools for Embedded Systems*, LCTES '98, pages 83–97, London, UK, UK, 1998. Springer-Verlag.

[6] Unmesh D. Bordoloi and Samarjit Chakraborty. Performance debugging of real-time systems using multicriteria schedulability analysis. In *Proceedings of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, RTAS '07, pages 193–202, Washington, DC, USA, 2007. IEEE Computer Society.

[7] Jan Chomicki and David Toman. Temporal databases. In Michael David Fisher, Dov M. Gabbay, and Lluis Vila, editors, *Handbook of Temporal Reasoning in Artificial Intelligence*, pages 429–467. Elsevier B. V., March 2005.

[8] James H. Cross, II and T. Dean Hendrix. jGRASP: An integrated development environment with visualizations for teaching Java in CS1, CS2, and beyond. *J. Comput. Small Coll.*, 23(2):170–172, 2007.

[9] Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. pages 219–234, April 1998.

[10] GDB: The GNU Project Debugger. `http://sources.redhat.com/gdb/`.

[11] The Java Debugger. `http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdb.html`.

[12] Eclipse. Eclipse Test and Performance Tools Platform.

[13] Paul Gestwicki and Bharat Jayaraman. Methodology and architecture of JIVE. In *Proceedings of the Symposium on Software visualization*, SoftVis '05, pages 95–104, New York, NY, USA, 2005. ACM.

[14] Paul V. Gestwicki and Bharat Jayaraman. Jive: java interactive visualization environment. In *Conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 226–228, New York, NY, USA, 2004. ACM.

[15] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 385–402, New York, NY, USA, 2005. ACM.

[16] J. B. Goodenough and L. Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. In *IRTAW '88: Proceedings of the second international workshop on Real-time Ada issues*, pages 20–31, New York, NY, USA, 1988. ACM.

[17] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[18] Ashish Gupta and Inderpal Singh Mumick. Materialized views. In Ashish Gupta and Iderpal Singh Mumick, editors, *Maintenance of Materialized Views: Problems, Techniques, and Applications*, pages 145–157. MIT Press, Cambridge, MA, USA, 1999.

[19] Wolfgang Haberl, Markus Herrmannsdoerfer, Jan Birke, and Uwe Baumgarten. Model-level debugging of embedded real-time systems. In *Proceedings of the Conference on Computer and Information Technology*, CIT '10, pages 1887–1894, Washington, DC, USA, 2010. IEEE Computer Society.

[20] The Java Debug Interface. `http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdi/`.

[21] Dean F. Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. pages 360–370, May 1997.

[22] Tomas Kalibera, Jeff Hagelberg, Filip Pizlo, Ales Plsek, Ben Titzer, and Jan Vitek. CDx: a family of real-time Java benchmarks. In *Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 41–50, New York, NY, USA, 2009. ACM.

[23] Tomas Kalibera, Pavel Parizek, Ghaith Haddad, Gary T. Leavens, and Jan Vitek. Challenge benchmarks for verification of real-time programs. In *Proceedings of the 4th ACM SIGPLAN workshop on Programming languages meets program verification*, PLPV '10, pages 57–62, New York, NY, USA, 2010. ACM.

[24] Andrew J. Ko and Brad A. Myers. Finding causes of program output with the Java Whyline. In *CHI '09: Proceedings of the 27th international conference on Human factors in computing systems*, pages 1569–1578, New York, NY, USA, 2009. ACM.

[25] Danny B. Lange and Yuichi Nakamura. Object-oriented program tracing and visualization. *Computer*, 30(5):63–70, May 1997.

[26] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Query-based debugging of object-oriented programs. In *Proceedings of the Conference on Object-oriented programming, systems, languages, and applications*, pages 304–317, New York, NY, USA, 1997. ACM.

[27] Demian Lessa, Jan Chomicki, and Bharat Jayaraman. Temporal data model for program debugging. In *International Symposium on Database Programming Languages*, August 2011.

[28] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20:46–61, January 1973.

[29] Filip Pizlo, Lukasz Ziarek, Ethan Blanton, Petr Maj, and Jan Vitek. High-level programming of embedded hard real-time devices. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 69–82, New York, NY, USA, 2010. ACM.

[30] Filip Pizlo, Lukasz Ziarek, and Jan Vitek. Real time Java on resource-constrained platforms with Fiji VM. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 110–119, New York, NY, USA, 2009. ACM.

[31] Guillaume Pothier, Éric Tanter, and José Piquer. Scalable omniscient debugging. *SIGPLAN Not.*, 42(10):535–552, 2007.

[32] Laakso M.-J. Kaila E. Salakoski T. Rajala, T. VILLE– A Language-Independent Program Visualization Tool. In Raymond Lister and Simon, editors, *Proceedings of the Baltic Sea Conference on Computing Education Research*, volume 88. Australian Computer Society, 2007.

[33] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

[34] Henrik Thane and Hans Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of the 12th Euromicro conference on Real-time systems*, Euromicro-RTS'00, pages 265–272, Washington, DC, USA, 2000. IEEE Computer Society.

[35] Yui Watanabe, Takashi Ishio, Yoshiro Ito, and Katsuro Inoue. Visualizing an execution trace as a compact sequence diagram using dominance algorithms. Antwerp, Belgium, october 2008.

[36] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.