Schedulable Online Testing Framework for Real-Time Embedded Applications in VM

Okehee Goh and Yann-Hang Lee

Computer Science and Engineering Department
Arizona State University, Tempe AZ, USA
{ogoh, yhlee}@asu.edu

Abstract. The paper suggests a VM-based online testing approach in which software testing is piggybacked at runtime on a system that operates to serve actual mission. Online testing in VM is facilitated with a framework that uses persistence service to initialize the testing operation with a consistent system state. The testing operation then runs in an isolated domain which can be scheduled independently of the operating version. Thus, testing operation cannot cause unbounded pause time nor spoil the normal operation. We evaluate the feasibility of schedulable online testing with a prototype developed in MONO CLI (Common Language Infrastructure) and the experiment on the prototype.

Key words: Online Testing, Virtual Machine, Real-Time Embedded Applications

1 Introduction

Nowadays, the applications of real-time embedded systems have proliferated from industrial controls to home automation, communication consumer gadgets, medical devices, defense systems and so forth. The apparent trends of the systems include sophisticated features and a short production cycle. The trends have sought solutions more in software rather than in hardware and also have led to the application of virtual software execution environment (VM) as a runtime environment. VM, populated with JVM [10] and CLI[1] [5], features high portability from using intermediate code, high productivity and reusability of object-oriented languages, and a safe runtime environment. The features are beneficial to the development of real-time embedded systems as the production cycle and cost can be reduced.

As software plays an increasingly significant role in embedded systems, the demands of upgrading software are anticipated for bug fixing and for extended functionality. In fact, most embedded systems, which have long lifetimes and require high availability, are generally passive on software upgrade because upgrading software requires to restart the systems, and the newly upgraded software

[1] CLR (Common Language Runtime), which is a CLI's implementation by Microsoft and an integral part of Microsoft .NET framework, is more popularly known than CLI.

can introduce new types of bugs and faults. Research work on online upgrade [12, 4], and reconfigurable systems [14], has been focusing on providing facilities to accomplish the software upgrade at runtime. In the meanwhile, there is no doubt as to the importance of software testing to verify the correctness, the completeness, and security, especially for mission- or safety-critical software in which even minor changes of the software require extensive testing [15].

Software testing is generally conducted in off-line environment with test cases generated by predetermined inputs. The weakness of the predetermined inputs, particularly derived from a model-driven formal system specification, is that the testing results are restricted to the correctness and completeness of a given model. Furthermore, an off-line testing environment for embedded software means that the software is tested in a simulation mode and does not participate in an actual mission. However, the testing of embedded software should be able to deal with external interferences and unexpected behavior of the target application environment. All possible inputs may not be known ahead, and the generation of complete test cases for all execution conditions is very problematical.

To overcome the aforementioned limitations, we suggest an online testing environment for software upgrades as a supplementary approach of an off-line testing. The testing of software upgrades is piggybacked at runtime on the systems that operate to serve an actual mission. Hence, the execution of the software dedicated to the actual mission coexists with the execution of the software to be tested. The apparent benefit of online testing is that testing undergoes not in a limited runtime environment but in an actual target environment connected to physical world. That is, the software testing is conducted by using actual inputs. To simplify terminologies to be used hereafter, the software for an actual mission, and the software to be tested are called software under operation (SUO), and software under test (SUT), respectively.

Most embedded applications run periodically with long lifetimes. At each period, they conduct computation by taking external input data, and then the computation results, represented as output data, is used to activate the target hardware. Some embedded applications' computation at each period is based on the state that has been accumulated from computations of preceding periods as well as the newly sampled input data. For online testing of this type of applications–*stateful software*, SUT must be able to start with the accumulated computation state of SUO, and since then, gets applied with the same inputs that SUO receives. Furthermore, if SUO is characterized by time constraints, whose virtue includes timely correctness, the online testing piggybacked has to be nonintrusive: the latency and pause time that SUO encounters due to online testing must be predictable and controllable. Certainly, any faulty behavior caused by SUT must be isolated to prevent SUO operations from any impact.

In this paper, we aim at a framework of schedulable online testing (SOTF) for real-time embedded software in VM. With the advent of an online testing request, the framework provides facilities to enable a testing mode where both SUO and SUT are executed concurrently. On the termination of testing, the system returns back to an operation mode of executing SUO only. It achieves fault isolation by executing SUT in a separate partition. By checkpointing the accumulated computation state of SUO, SUT begins to execute from a consistent state. In addition, the framework logs the external input data which SUO receives, and reconstructs the logs for the execution of SUT. Finally, SOTF employs a preemptible mechanism for checkpointing and recovery of persisted states and provides the flexibility to resume the testing anytime. Hence, the timely correctness of SUO can be ensured.

In the following section, we give a discussion of related works. The target application model of online testing is introduced in Section 3. Then, the approaches and designs of the proposed schedulable online testing framework on CLI (Common Language Infrastructure)[5]'s open source platform, MONO [17], are presented in Section 4. In Section 5, the experiment on the prototyped SOTF is used to show the space overhead incurred by the testing framework. The overhead and the source of latency of online testing with SOTF are identified. Finally, Section 6 draws a conclusion.

2 Related Works

One of well-known research areas with a key role of checkpointing and/or logging is log-based rollback recovery [6]. Logging-based recovery protocol, especially on .NET framework [2, 1], is tuned at component oriented distributed applications The work was motivated with the problems that process-based recovery protocol cannot detect the failure of components, and checkpointing/recovery in a process level is very heavy. The prototype employs .NET's *Object Serialization* and *Context* to support checkpointing/recovery and to enable interceptions of messages (to aid logging) on persistent components, respectively. The rebinding of recovered components is done through .NET *Remoting*'s registry so that other stable components can access the recovered components.

Simplex architecture[13] is to support the evolvability of dependable real-time computing systems. The architecture adopts analytical redundant logic: running a trusted version (a fault-proof component) and an upgrade version (a not-yet-fault-proof component) in parallel as separate software. The architecture has decision logic that monitors the behavior of an upgrade version. If a faulty action is detected from the version, the control of the system is switched to the trusted version. Resource isolation is emphasized to prevent a trusted version from being corrupted due to the faulty behavior of an upgrade version. Lee et al.[9] extended the Simplex architecture for online testing and upgrade of industrial controller in the Linux OS environment by applying the technique of *Process Resurrection* [8].

RAIC (Redundant Arrays of Independent Components) [11] is a technology that uses groups of similar or identical components to provide software dependability and allows component hot-swapping; the architecture allows addition or removal of components at runtime. When a component is swapped, the state transfer from an old component to a replaced component is supported, if the component is stateful. If the components are faulty, as examined using built-in testing code on the controller, the controller handles the faulty exceptions and recovers the application state so that the fault is not exposed to the applications.

The primary difference of our work from the previous works is that our framework aims a testing facility in VM that allows preemption to reduce blocking delays. Thus, flexible scheduling of testing can be carried out while ensuring the timeliness of regular service.

3 Target Application Model

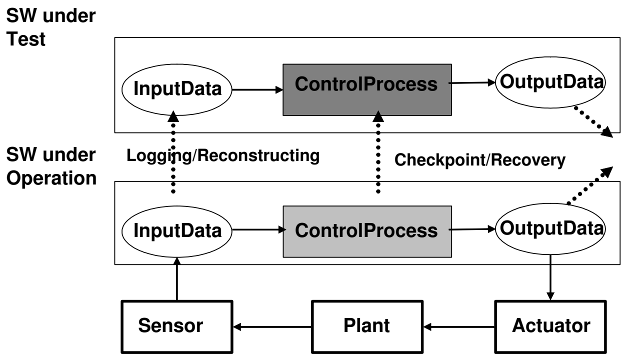

Fig. 1. Target application model of Online Testing

The target application model we envision for SOTF is a closed-loop control system. The systems basically include sensors, control processes, and actuators, and the control tasks run concurrently and periodically to control a target plant. A simplified view of the system can be described by three application-level objects for the three system components: InputData, ControlProcess, and OutputData.

Figure 1 is a simplified closed-loop control system applying the schedulable online testing. In the figure, *InputData* indicates the data collected from sensors and taken by a control process, and *OutputData* are the computational results generated by the process and passed to actuators. We assume that a control process demands upgrades to meet performance enhancement or new business requirements. The upgrade version's control process has to access the same InputData as the operation version; that is, the upgrade version maintains the same frequency and format for the access to InputData as the operation version does. It is a reasonable assumption because InputData, which is generated by a sensor as a result of monitoring the target plant, does not get changed unless the sensor or the target plant gets replaced or upgraded. The same assumption is applied to OutputData with respect to the actuator.

Consider a stateful control process where the computation result of each periodic operation depends on not only the InputData obtained at the current period but also the accumulated state from the computations of the preceding periods. An upgrade version's control process can start online testing by being initialized with the state of an operation version's control process at the moment that online testing is triggered, and by accessing the same InputData as the operation version.

The concern of online testing with real-time applications is to ensure time constraints of regular service while online testing is under way. An approach to address this issue is to schedule the testing as a background task, which runs when no real-time task is ready. One of the problems that arise under this scheduling approach is the availability of InputData. When an upgrade version is ready to run, InputData that primarily stimulates the operation version's control process, may not be available any more. Also, the initial state must be preserved until the upgrade versions start to execute. The foremost important issue is that the testing operation, the logging of Inputdata, and the preservation of the initial state must be preemptible. Thus, the testing process would not block the regular service of the target system.

4 Online Testing Service

To enable schedulable online testing, we present the approaches in VM-based real-time embedded system. Although the approaches are applicable to both CLI and JVM, we built a prototype of SOTF in MONO CLI and, in the subsequent text, refer to specific technologies and standard class library of CLI.

4.1 Online Testing Framework

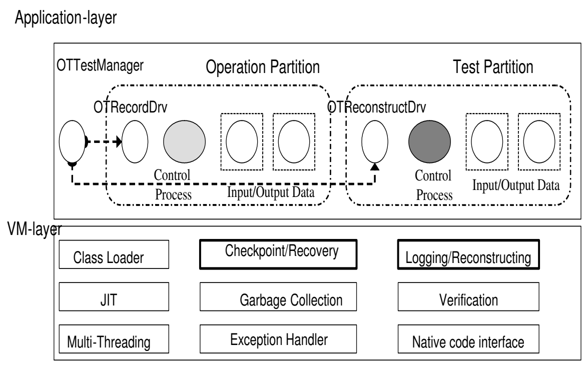

Fig. 2. Online Test Framework

SOTF consists of three tasks (in the application layer) to drive online testing, and two subsystems (in the VM layer) to aid online testing: three tasks including OTTestManager, OTRecordDriver (OTRecordDrv), and OTReconstructDriver (OTReconstructDrv), and two subsystems including a preemptible persistence system and a logging/reconstructing system.

Figure 2 illustrates the architecture of the framework of schedulable online testing. The roles of the three tasks are as follows. OTTestManager is responsible for triggering online testing when software to be tested is ready. As a response of the commands from OTTestManager, OTRecordDrv interacts with SUO to checkpoint a consistent state of SUO and log input data sampled from sensors. Correspondingly, OTReconstructDrv interacts with SUT to direct the recovery and reconstruction operations of the persisted state and the logged input data. In the figure, SUO and SUT are represented with a composition of ControlProcess, InputData, and OutputData, as a simplified model suggested in Section 3.

4.2 An Isolated Testing Environment

As SUT concurrently runs with SUO, the possible faulty behavior of SUT can affect the operation of SUO. To contain the faulty behavior of SUT, it should reside in a runtime environment separated from that of SUO. We employ *Application Domain* [3] in CLI as a facility to provide an isolated runtime environment. The application domain is a lightweight address space designed as a model of scoping the execution of program code and the ownership of resources. Sharing objects between different domains is prohibited: that is, objects in one domain cannot access objects in other domains. Creating multiple application domains by starting assembly[2] with a main entry, is supported at runtime. Additionally, CLI facilitates unloading an application domain at runtime. This allows a dynamically created SUT domain when a testing operation is requested.

4.3 Preemptible Checkpointing and Recovery

If SUO's accumulated state from the computations of preceding periods is preserved, then SUT can start with the known initial state. To transfer the state from SUO to SUT across the domain boundary, we adopt the approach of checkpointing SUO's state and recovering the state for SUT. The challenge of the approach, especially for real-time applications, is that the pause time due to the checkpointing/recovery operation may be unbounded. The unpredictable latency from checkpointing can hinder the timeliness of SUO if SUO is blocked until the checkpointing operation finishes entirely.

To make it possible to bound the pause time due to checkpointing and recovery, our prior work, *schedulable persistence system* (SP system) [7] is adopted with which the persistence service runs concurrently with real-time tasks. The minimal length of the pause time, i.e. the minimal non-preemptible region in the

[2] *Assembly* is a minimal unit of reuse, versioning, and deployment in CLI.

persistence service, can be adjusted to meet the scheduling needs of real-time application tasks.

When the persisted state is deserialized, there may be a question whether the state objects can be useful directly by the SUT. If the state objects of the control process in SUT is the same object of SUO, i.e. the names of persistent classes and persistent fields in SUT is identical to these in SUO, then the persisted objects generated from SUO can be used to initialize SUT. Otherwise, we can apply a transformation script to reconstruct the state objects for SUT based on the persisted SUO objects.

4.4 Logging and Reconstructing

After being initialized with the checkpointed state of SUO, SUT is ready to execute. It should receive the sampled input data similar to the one applied to SUO since the checkpoint. As a solution, the access to InputData by SUO's ControlProcess is logged and then the access to InputData by SUT's ControlProcess is sufficed with the logs. This logging/reconstructing requires to intercept the method calls on InputData objects. That is, the method calls by SUO's ControlProcess to read InputData is post-processed to log the sampled data, and the method call by SUT's ControlProcess to read InputData is pre-processed to reconstruct the sampled data based on the logs. The post- and pre-processing on InputData objects are done through the *Context* in CLI which provides an object with an execution scope. Additional services can be augmented during incoming/outgoing method calls on context-bounded objects which are derived from the *System.ContextBoundObject*. This feature has been employed in a logging-based recovery protocol on .NET framework [2, 1] to enable the interceptions of messages (to aid logging) on persistent components.

5 Experiments

The experiment of the SOTF prototype on MONO CLI is performed to understand the source of latency on a testing sequence and to examine the concerns of scheduling online testing in an example system. It is conducted with C# benchmarking applications on a PC workstation with 1.5GHz Pentium IV processor and 256MB memory. To have a high resolution timer and preemptive kernel, TimeSys' Linux/RK (real-time kernel v4.1.147) [16] is used. For time measurement, a standard class library *System.DateTime.Now.Ticks* is used, which gives 100ns resolution. The C# language supports five levels of thread priorities, *Highest*, *AboveNormal*, *Normal*, *BelowNormal*, and *Lowest*. The priorities, are implemented using TimeSys RK's POSIX real-time FIFO scheduling policy.

5.1 Cost Analysis for Testing Sequence

SOTF is implemented by integrating a wide range of facilities to satisfy the requirements of online testing such as an isolated testing environment, interceptions of method calls, checkpointing/recovery, and logging/reconstructing.

Using the facilities leads to some overhead. Although the amount of overhead or cost depends on the techniques employed, these types of overhead or cost are inevitable. In this experiment, we analyze the cost incurred in every stage constituting the testing sequence.

The benchmarking application, SUO, used in this experiment is a seismic event monitor, which computes the rate of seismic events by using both seismometric data newly obtained and seismometric data accumulated from a preceding duration. The seismic event monitor (SUO) runs periodically every 3ms for 1ms WCET (Worst Case Execution Time) with AboveNormal priority. The seismometric data read from its InputData object is 20Bytes. The seismometric data accumulated from prior computations, a persistent state of SUO's Control-Process, will be checkpointed to aid for online testing. The size of the persistent state, consisting of about 1000 composite objects including primitive types' fields, is about 20000Bytes. Its upgrade version, SUT, embodies a slightly different computation approach but generates basically the same results with the operation version SUO. When online testing starts, SUT runs every 1ms with Lowest priority. To just observe the overhead of the operation in a testing sequence, we allow the checkpointing on SUO and the recovery of persisted data on SUT to perform in a nonpreemptible mode. Additionally, the termination condition of testing is set to 300 periods of SUO's operation.

Stages	Time (ms)
(1) Receive a testing request	0
(2) Turn on checkpointing on SUO	10
(3) Start checkpointing/logging on SUO	12
(4) Complete checkpointing	25
(5) Start SUT	29
(6) Complete initialization for testing on SUT	686
(7) Start recovery on SUT	691
(8) Complete recovery, and start testing on SUT	699
(9) Complete logging on SUO	862
(10) Complete testing on SUT	1015
(11) Start unloading of SUT	1044
(12) Complete unloading of SUT	1109

Table 1. Time line of a testing sequence

Table 1 shows the cost incurred in every stage constituting the testing sequence. The result is chosen as one with the longest completion time (e.g. the moment that the unloading of SUT completes since the advent of a testing request) from 20 runs. The time specified at each stage is the elapsed time since

OTTestManager received the testing request. We speculate the costs involved to carry out three functions: (1) coordinating SOTF tasks (OTTestManager, OTRecordDrv, and OTReconstructDrv) and transferring information between two different application domains, (2) conducting checkpointing and recovery, and (3) starting and unloading software at runtime.

The cost in function (1) attributes to the coordination of SOTF tasks in different application domains. For instance, the OTTestManager task, receiving an event for testing, informs OTRecordDrv to prepare for testing. What is carried out in this step is that one task fires an event to wake up a dormant thread, and the data specification for testing is transferred from one application domain (where OTTestManager runs) to the other application domain (where OTRecordDrv runs). To enable the communication between tasks in different application domains, CLI's *AutoResetEvent* class and *AppDomain* class are used. The result, leading to about 10ms delay, which is quite expensive and mostly comes from marshaling (to pass objects between the domains), indicates that the efficient communication mechanism between different domains is desired.

Checkpointing and recovery operations take 13ms ((4)-(3)), and 8ms ((8)-(7)), respectively. The size of final persisted data, including metadata for the serialization protocol, is 33125Bytes. Compared to the experiment results (186ms and 69ms for serialization and deserialization respectively) by standard serialization library, the schedulable persistence system (SP system) in [7] substantially outperforms the standard serialization class libraries.

In Table 1, we also notice the cost for starting and unloading testing software in a new domain at runtime. The operations are implemented using AppDomain class's *CreateDomain*, *ExecuteAssembly*, and *Unload* methods. The delay reaches to 657ms((6)-(5)) to start a new software, and 65ms((12)-(11)) to unload the software, respectively. This noticeable delays comes from loading and compiling not only user classes of the new assembly but also a large portion of system classes referenced by the user classes.

5.2 Scheduling Online Testing and Space Overhead

In SOTF, it is imperative that the timeliness of applications (SUO and other real-time tasks) has to be guaranteed while testing is in progress. The simplest scheduling approach is to treat the testing as a background job so that the testing operations would have a minimal interference to the applications' timeliness. One of the concerns with the background testing job is the nonpreemptible regions caused by SOTF. The other concerns is the overhead of space that is reserved to save the logs. Logs produced by SUO have to remain until they are consumed by SUT. The issue of space overhead also attributes to the characteristic of embedded software testing, which requires to be exposed to the physical world for a long period. Thus, it can encounter all possible input data sets. Here, we consider a schedule example and use it to examine the space overhead in a testing process.

In this experiment, the SUO (a seismic event monitor) and its corresponding SUT (an upgrade version of the seismic event monitor) are same with ones in the

CU	T1	T2	T3	T4
0.5	0.5/5	1.6/8	1/10	1.5/15
0.6	0.5/5	1.6/8	2/10	1.5/15
0.75	1/5	2/8	2/10	1.5/15
Priority	AboveNormal	Normal	Normal	BelowNormal

Table 2. Task Sets (CU: CPU Utilization, time unit: ms)

previous experiment so that the size of persistent data for checkpointing/recovery, and the size of each log are same with the previous experiment. Additionally, service launched in an operation partition includes three more tasks besides SUO; that is, an operation partition consists of four tasks. Table 2 specifies three different task sets, and their scheduling parameters according to varying CPU utilization (CU), 0.5, 0.6, and 0.75. The table also specifies WCET, period, and priority of each task, which runs periodically; for example, T1 in the CPU utilization set 0.5 has 0.5ms WCET and 5ms period, and runs with the AboveNormal priority. Among the tasks, T1 is SUO which has a correspondent SUT. SUT, which is not specified as a task in the table because it does not account for CPU utilization, runs as a background task (with the Lowest priority). Checkpointing and recovery are carried out in a preemptible mode with the Highest priority–3ms period, and 2ms WCET. That is, when checkpointing or recovery is initiated, it runs 2ms every 3ms until it completes the requested service. In this experiment, we focus on understanding space overhead during online testing so that we ignore the testing overhead including checkpointing and recovery although it affects the schedulability of the task sets. To ease the termination condition of testing, the testing duration is limited to 300 periods of SUO operation.

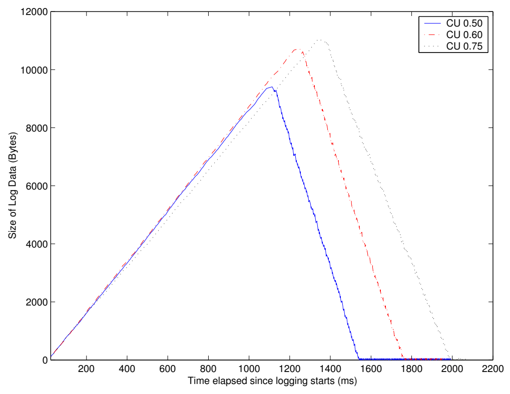

Fig. 3. Space for logs according to varying load

Figure 3 shows the space size of log data for three task sets over time until testing on SUT completes since logging on SUO started. According to the graph, the execution of SUT, conducting testing by actually consuming the logs, starts at around 1122ms, 1257ms, 1342ms for CU 0.5, CU 0.6, and CU 0.75, respectively. This start time is influenced from higher priority tasks' loads and also the overhead of online testing: as we see in the second experiment, SUT can start once the completion of checkpointing by SUO, the launch of SUT by OTTestManager, and the completion of recovery by SUT are accomplished, which take approximately 700ms. Regarding the space for logs, if the duration of SUO logging (producing logs) is not overlapped with the duration of SUT testing (consuming logs), the space for logs including metadata is 16500Bytes. In fact, the result shows that the maximum space size for logs reaches to 9407Bytes, 10727Bytes, 11057bytes for CU 0.5, CU 0.6, and CU 0.75, respectively. The experiment shows that the space for logs reaches to the maximum during the initial stage of testing; once testing by SUT starts by consuming the logs, the space required for logs becomes less than during the initial stage. It indicates that testing can be conducted for long duration without a severe burden of space if the system can guarantee the maximum space needed in the initial stage.

Besides the space issue, conducting checkpointing and recovery in a preemptible mode shows that it can keep the maximum pause time 2ms and then their response times become 18ms, and 11ms on average, respectively, due to the execution of interleaved mutators. It indicates that checkpointing and recovery do not stop application tasks for 13ms and 8ms as its nonpreemptible mode of the second experiment.

Conclusively, predicting the upper bound of memory space reserved for logs has to consider the cost and overhead of online testing and the workload of applications.

6 Conclusion

In this paper, we depict a VM-based schedulable online testing framework for testing software upgrade of real-time embedded applications. The testing can undergo with actual input data in a target runtime environment. The framework is built by integrating a wide range of mechanisms of VM, including an isolated partition for testing, preemptible checkpointing/recovery, and logging/reconstructing the sampled input data. Meanwhile, in order to prevent the testing from causing adverse effects on the ongoing regular services of the target systems, the testing task runs in the background mode and read in sampled data via a log buffer. The experiment with the prototype of the framework, developed on MONO, demonstrates the feasibility of online testing in VM environment as well as the required capacity of the log buffer.

References

1. Roger Barga, Shimin Chen, and David Lomet. Improving logging and recovery performance in phoenix/app. *ICDE*, 00:486, 2004.

2. Roger Barga, David Lomet, Stelios Paparizos, Haifeng Yu, and Sirish Chandrasekaran. Persistent applications via automatic recovery. *IDEAS*, 00:258–267, 2003.
3. Don Box and Chris Sells. *Essential .NET Volume 1: The Common Language Runtime*. Addison Wesley, 1st edition, 2002.
4. M. Dmitriev. The first experience of class evolution support in PJama. In *Proc. of The 8th International Workshop on Persistent Object Systems (POS-8) and The 3rd International Workshop on Persistence and Java (PJW3)*, pages 279–296. Morgan Kaufmann Publishers, Inc., 1998.
5. ECMA. Ecma-335 common language infrastructure, 2002.
6. E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, September 2002.
7. Okehee Goh, Yann-Hang Lee, Ziad Kaakani, and Elliott Rachlin. Schedulable persistence system for real-time embedded applications in VM. In *EMSOFT*, pages 101–108, 2006.
8. K. Lee and L. Sha. Process resurrection: A fast recovery mechanism for real-time embedded systems. In *Real Time and Embedded Technology and Applications Symposium*, pages 292–301, 2005.
9. Kihwal Lee and Lui Sha. A dependable online testing and upgrade architecture for real-time embedded systems. In *RTCSA*, pages 160–165, 2005.
10. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
11. C. Liu and D.J. Richardson. RAIC: Architecting dependable systems through redundancy and just-in-time testing. In *ICSE 2002 Workshop on Architecting Dependable Systems*, 2002.
12. Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. *Lecture Notes in Computer Science*, 1850:337–361, 2000.
13. Lui Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, July/August 2001.
14. Craig A. N. Soules, Jonathan Appavoo, Kevin Hui, Robert W. Wisniewski, Dilma Da Silva, Gregory R. Ganger, Orran Krieger, Michael Stumm, Marc A. Auslander, Michal Ostrowski, Bryan S. Rosenburg, and Jimi Xenidis. System support for online reconfiguration. In *USENIX Annual Technical Conference, General Track*, pages 141–154, 2003.
15. J.A. Stankovic. Misconceptions about real-time computing: a serious problem for next generation systems. *Computer Magazine*, pages 10–19, October 1988.
16. TimeSys Corporation. Timesys linux/real-time user's guide, version 2.0, 2004.
17. Ximian. MONO. http://www.go-mono.com.