Surviving Peripheral Failures in Embedded Systems

Rebecca Smith and Scott Rixner, ***Rice University***

https://www.usenix.org/conference/atc15/technical-session/presentation/smith

This paper is included in the Proceedings of the 2015 USENIX Annual Technical Conference (USENIC ATC '15).

July 8–10, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-225

Open access to the Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15) is sponsored by USENIX.

Surviving Peripheral Failures in Embedded Systems

Rebecca Smith
Rice University
rjs@rice.edu

Scott Rixner
Rice University
rixner@rice.edu

Abstract

Peripherals fail. Yet, modern embedded systems largely leave the burden of tolerating peripheral failures to the programmer. This paper presents Phoenix, a semi-automated peripheral recovery system for resource-constrained embedded systems. Phoenix introduces lightweight checkpointing mechanisms that transparently track both the internal program state and the external peripheral state. These mechanisms enable rollback to the precise point at which any failed peripheral access occurred using as little as 6 KB of memory, minimizing both recovery latency and memory utilization.

1 Introduction

Embedded systems drive the modern world, controlling devices all around us from toasters to automobiles. Due to resource constraints, these systems are typically programmed in low-level languages, with the application developer manually managing system resources. To increase programmer productivity, managed run-time systems are gaining popularity within the domain of embedded systems [1, 2, 3, 4]. Run-time systems ease the burden on the programmer by automating resource management and supporting high-level programming languages. The small amount of execution variability that these run-time systems introduce can be tolerated, in order to gain productivity, by many of the soft real-time workloads that are prevalent in modern embedded systems.

Embedded systems are typically event-driven, interacting with the real world by reading sensors, making decisions, and controlling actuators. Since these systems impact their environment, failures can have real-world consequences. Microcontrollers often have built-in support for reliability [19, 31], and could implement additional known processor reliability techniques [5, 41, 43]. However, the sensors and actuators at the heart of these systems are extremely susceptible to failures [9, 20]. Despite this, techniques for fault tolerance in embedded systems primarily focus on handling microcontroller failures [8, 22, 46, 48, 50], leaving management of peripheral failures entirely to the programmer.

This is a complex task for the programmer, as peripherals can fail asynchronously. The programmer would have to anticipate all possible scenarios and determine how to correctly restore the system state after an arbitrary number of instructions and peripheral accesses have executed. Such an application would be difficult to write and even harder to test, and still might have problems if peripherals fail at unanticipated and inopportune times. Therefore, it is crucial to develop mechanisms to facilitate recovery from peripheral failures.

These mechanisms should be largely transparent to the programmer, as is the case in many other domains such as large-scale distributed systems [15, 21, 25, 35, 39]. However, many existing system-level techniques require active participation in the reliability protocols by the elements that can fail; such techniques are ill-suited to handling peripheral failures in embedded systems, as these peripherals are incapable of such participation. One approach that has been effective in embedded systems is checkpointing, which enables the system to roll back to a known valid state, correct failures, and re-execute. However, existing checkpointing schemes for embedded systems do not support peripheral failures [8, 46, 50].

This paper presents Phoenix, a semi-automated peripheral recovery system built around novel checkpointing mechanisms. Phoenix provides fault tolerance for fail-stop peripheral failures, access timeouts, violations in communications protocols, and interrupt storms or other spurious interrupts. The design of Phoenix was motivated by a set of insights into the unique properties of embedded systems and their ramifications on recovery.

In particular, peripherals introduce complexities that must be addressed by any recovery process for correctness. First, many peripherals have external state, which is fundamentally different from internal program state. Internal state can be rolled back simply by resetting the memory. In contrast, the interactions that pe-

ripherals have with the real world may be difficult to undo. Furthermore, some peripherals have transient effects, while others have lasting effects. These differences influence how each peripheral must be handled during recovery. Finally, peripherals may interact with each other. Such dependencies determine the necessary actions to restore the external state. While some peripheral accesses should be replayed, in other cases it is incorrect to redo the access, so it must be skipped.

These complexities, taken in conjunction with stringent resource constraints, introduce several challenges to implementing efficient checkpointing in embedded systems. First, the external peripheral state must be restored along with the internal program state; traditional checkpointing mechanisms neglect this peripheral state. Second, simply taking a snapshot of the memory is intractable. Embedded systems have a small amount of memory, the majority of which is needed by the application itself. It is not practical to keep even a single copy of the memory at any given time. Last, the performance limitations of resource-constrained embedded systems motivate minimizing the latency of recovery. Ideally, the system would roll back to exactly the point before the particular access that failed, minimizing the number of instructions and peripheral accesses to be re-executed.

Based on these insights, the checkpointing mechanisms in Phoenix simultaneously track internal and external state, optimize memory utilization, and enable rollback to any precise point at which a peripheral failure could occur. If a failure occurs, Phoenix automatically rolls back and recovers while keeping the internal and external state consistent. For efficiency, the system only tracks state when there is a chance of failure. Checkpointing is automatically turned on when a peripheral is accessed, and turned off once all past accesses have succeeded. While checkpointing is enabled the system builds an incremental log of the state, maintaining pointers into this log corresponding to each peripheral access. This incremental approach serves two purposes. First, it only checkpoints the minimal amount of state required for rollback. Second, it allows the system to roll back to the point right before any peripheral access that could fail, thereby minimizing re-execution.

Evaluation on a set of microbenchmarks and applications showed that Phoenix is space-efficient enough to operate within the resource constraints of embedded systems. Running on a microcontroller with 96 KB of SRAM, Phoenix used on the order of 5 KB to track both the internal and external system state, leaving the majority of the space free for use by the running program. When there was a failure the overhead rose to 6 KB, as a small amount of extra state is needed during the recovery process. Furthermore, for two of the three applications studied in this paper there was no perceivable change in performance with the addition of Phoenix.

The mechanisms of Phoenix transcend peripheral failures. In particular, any recovery strategy for embedded systems should restore both internal and external state when any asynchronous failure occurs. Phoenix's techniques for logging and restoring these two types of state could be combined with the appropriate failure detection mechanisms to handle additional types of failures.

The rest of the paper proceeds as follows. Section 2 presents the key insights that motivate the design of Phoenix, and Section 3 presents the recovery procedure. Sections 4 and 5 describe the mechanisms used to implement the system. Section 6 shows the experimental evaluation of Phoenix, and Section 7 discusses related work. Finally, Section 8 concludes the paper.

2 Key Insights

Two sets of insights into the unique properties of embedded systems shaped the design of Phoenix. The first set consists of broad insights into the implications of peripherals on system behavior and therefore correct system recovery. Supplementing these, a second set of insights influenced the adaptation of a specific reliability technique, checkpointing, to the domain of embedded systems.

2.1 Peripheral State in Embedded Systems

Embedded systems are inherently event-driven, interacting with their surroundings through sensors and actuators. These peripherals have unique properties which manifest themselves in subtle and complex ways, and which must be addressed in order to correctly recover from a failure. Five insights into these peripherals drive the design of the mechanisms presented in this paper.

First, hardware peripherals introduce external state in addition to the internal state. To guarantee correct recovery from failures, Phoenix restores both types of state.

Second, the way that peripherals affect external state varies. Based on this, peripherals can be classified into four categories: stateless, ephemeral, persistent, or historical. The first category includes simple sensors such as accelerometers which cannot affect their surroundings. Ephemeral peripherals do affect the external state, albeit fleetingly; these include, for example, buzzers. In contrast, persistent peripherals have lasting effects. The state of a persistent peripheral is entirely determined by the last write to it. For instance, setting the speed of a motor causes a state change that persists until a new speed is set. Historical peripherals likewise have lasting state, but the state of a historical peripheral is an aggregation of a series of prior writes. Phoenix selects the recovery actions to perform for a given peripheral based on the classification it was registered with during initialization.

Third, peripherals do not operate in isolation; the state of one may impact the behavior of another. Such de-

pendencies are often specific to the context of a particular application. A peripheral P1 has a dependency on a peripheral P2 if P2 failing results in P1 not having its intended effect on the external state. As an example, consider an autonomous car that uses a motor and steering servo to drive. The servo will rotate the car's wheels regardless of whether the motor is functioning. However, if the servo's goal in this application is turning the car itself, the motor failing will prevent it from accomplishing its goal. Thus, the servo has a dependency on the motor.

Fourth, not all peripheral accesses can be replayed. Consider a historical peripheral such as a message passing interface between two devices. If a message is read from this interface, but an unrelated peripheral fails before the message is processed, re-executing the read is incorrect for two reasons. First, the program must process the original message. Moreover, the state of the message passing interface is not only external but shared; depending on the messaging protocol being used, the device on the other end of the connection may not re-send, so a re-read may time out. Instead, this read must be rematerialized: skipped during re-execution, in favor of re-using the original return value. In contrast, accesses to peripherals that depended on the failed peripheral must be re-executed. When a peripheral fails, the Phoenix system uses the dependency information to populate an initial set of peripherals to redo; accesses to peripherals not in this set are rematerialized. As re-execution proceeds, this set is updated as necessary to adapt to changes in state.

Finally, the lasting effects of persistent peripherals demand additional mechanisms for restoring their state. Correct restoration requires three steps. First, prior to recovering from a failure, all persistent peripherals must be put into a safe state; otherwise, they may continue operating in an erroneous state. Next, all persistent peripherals selected for re-execution must be restored to the state that they held prior to the failed access. That way, when re-execution begins, they will be in the same state that they were at this point in the original execution. Last, once re-execution is complete, all persistent peripherals whose accesses were rematerialized must be set to the last state that was rematerialized, so that they are in the correct state going forward. Phoenix automatically performs each of these three steps, ensuring a consistent state throughout the duration of the recovery process.

2.2 Checkpointing in Embedded Systems

Checkpointing is often used to provide fault-tolerance in domains such as mobile and distributed systems [25, 35, 39]. At a high level, the concept of tracking and restoring a known consistent state extends well to other domains. However, the resource-constrained nature of embedded systems presents three unique challenges to adapting existing checkpointing mechanisms.

First, all of the insights introduced in Section 2.1 must be considered when designing *any* recovery mechanism for embedded systems; checkpointing is no exception. In the context of checkpointing, this means that this external peripheral state must be logged and rolled back just like the internal program state. Phoenix therefore logs every peripheral access that is performed. During re-execution, it makes the decision of whether to re-execute or skip on the granularity of an individual peripheral access.

Second, embedded systems face tight memory constraints. For example, the TI Stellaris LM3S9B92 [47], a typical ARM Cortex-M3 microcontroller, has only 96 KB of SRAM and 256 KB of flash memory. Phoenix addresses the lack of memory space by using a logging technique that resembles journaling filesystems [10, 17, 26, 38]. Rather than preemptively checkpointing some or all of the memory, Phoenix only copies memory that has actually been changed. Moreover, Phoenix takes advantage of the fact that when there are no outstanding peripheral accesses, there is no chance of peripheral failure. Thus, Phoenix automatically disables logging once all peripheral accesses have been acked, re-enabling logging only when another peripheral access is issued.

Last, embedded systems also face time constraints. The LM3S9B92 operates at 50 MHz; therefore, minimizing the rollback latency is crucial to maintaining performance. This is magnified by the fact that even soft real-time embedded applications are typically event-driven, and thus require some degree of reactivity. Phoenix guarantees that when a peripheral fails there will be sufficient checkpointing data to roll back to the exact point of failure, avoiding unnecessary re-execution. It achieves this by maintaining one logical checkpoint for each outstanding peripheral access. Each checkpoint is identified by pointers into checkpointing structures that contain only a small subset of the memory at any given time.

3 Recovery Procedure

This section presents a semi-automated recovery procedure for peripheral failures that builds on the insights from Section 2. This procedure was implemented in Owl [3, 7], an embedded run-time system including a Python bytecode interpreter that runs on the bare metal. Peripheral accesses are structured as function calls made through Owl's native function interface. These functions access the hardware through a thin C library. While minor implementation details were tailored to Owl, the procedure and mechanisms of Phoenix transcend Owl and could be implemented in other run-time systems.

When a peripheral access fails, it must be re-executed in order to achieve correct program behavior. However, re-executing this access in the context of the arbitrary execution state in which its failure is detected may be insufficient or incorrect. First, it is likely that many byte-

codes were executed between the peripheral access and the detection of its failure. These bytecodes could have changed the program state. If the peripheral access interacts with the internal program state at all, such as through its parameters, then in order to re-execute it correctly the state of the program must be restored to the point at which the access originally occurred. Second, naively re-executing the same peripheral access may simply result in another failure. Thus, the failed peripheral must be recovered prior to re-execution. Last, some of the operations performed after the peripheral access was issued but before its failure was detected may have depended on the success of the failed access; in this case, those operations must be re-executed as well.

The Phoenix system translates these facts into a three step recovery process which automatically executes upon detection of failure. First, the internal state is rolled back to the point of the failed access. Second, the failed peripheral is recovered. Third, the correct external state is reached via redo mode execution. Within redo mode, the system re-executes the failed access, as well as all accesses to dependent peripherals, but rematerializes accesses to unrelated peripherals. Once execution reaches the point of failure detection, the system seamlessly exits redo mode and resumes normal execution.

As an example, consider Figure 1, which is sample code for an autonomous car that uses a motor and steering servo to drive and an SD card to record its movements. Assume that the peripheral access in line 3 fails, and the system identifies this failure after line 8.

Upon detecting this failure, the system will first put the motor and servo into safe states. It will then restore the internal state as it was immediately prior to line 3, resetting the value of the variable `speed` to 100. The system will recover the motor by invoking a programmer-provided recovery function, and will then enter redo mode. During redo mode, line 3 will be re-executed, since it failed initially. However, line 4 will be rematerialized, as there is no dependency between the motor and the SD card. This rematerialization is a requirement for correctness. Effectively, redo mode restores the peripheral state that would have been reached had there been no failure at all. It would be inaccurate to replay this write

```
1  # Run the motor
2  speed = 100
3  motor.run(speed)
4  SD.write('set motor to 100')
5  speed += 100
6
7  # Turn the wheels
8  servo.set_servo(LEFT)
9  ...
```

Figure 1: Sample Application Code

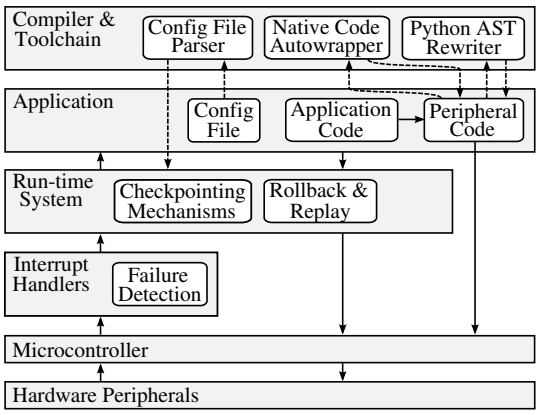

Figure 2: Phoenix System Architecture

and record the motor as having been set to 100 twice, when in fact the first attempt was unsuccessful. Next, line 5 will be re-executed, because it affects the internal state, which was rolled back. Last, the servo depends on the motor, so line 8 will also be re-executed. Finally, normal execution will resume.

4 System Mechanisms

This section presents the interlocking compiler and run-time system mechanisms which work together to carry out the procedure presented in Section 3.

At a high level, the compiler supports recovery in two key ways: it injects code to enable checkpointing prior to each peripheral access, and it auto-generates code to log the arguments and return value of each peripheral access. Then, when a failure is detected at run-time, the run-time system automatically identifies the failed peripheral, puts all persistent peripherals into a safe state, and executes the three step recovery procedure.

None of these mechanisms alone are sufficient for recovery. But, taken as a whole, they form a complete recovery system which facilitates every step from failure detection to rollback, recovery, and re-execution. Figure 2 presents an overview of the Phoenix system architecture, illustrating how the various components of the Phoenix system, in white, fit into the Owl system, in grey.

4.1 Failure Detection

Microcontrollers offer a variety of external peripheral interfaces, which notify the processor of events via interrupts. For example, the LM3S9B92 includes universal asynchronous receiver/transmitters (UART), inter-integrated circuit interfaces (I2C), and other interfaces.

Phoenix provides two system-level mechanisms to interact with these external interfaces: interrupt handlers and light-weight software shims. Since these two mechanisms are the link between the peripherals and the rest of the Phoenix system, Phoenix automatically recovers from all failures that they can detect, primarily consist-

ing of communication failures such as fail-stop peripheral failures, timeouts, and spurious interrupts. Upon detecting a failure, these mechanisms identify the failed peripheral and set the three step recovery procedure in motion by throwing an exception to the interpreter.

Most failures are detected solely via interrupt handlers. However, some external interfaces do not provide the appropriate type of interrupts for failure detection; in these cases, the interrupt handlers are supplemented by the software shims. Further, the existing software detection mechanisms could be augmented to perform additional validation that the peripheral is operating correctly, thereby extending fault coverage without any changes to the rest of the system.

4.1.1 Hardware Detection

The interrupt handlers in Phoenix manage both successful and unsuccessful peripheral accesses. Each interrupt handler begins by pinpointing the peripheral whose access generated the interrupt. During initialization (see Section 5.2), each peripheral is registered with an (interface, interface details) pair that uniquely identifies it to the interrupt handler. As an example, the pair for an I2C device would look like (I2C, (master address, slave address)). Thus, the handler identifies the source of the interrupt by searching the set of registered peripherals.

Next, the interrupt handler checks the status flags to determine whether the access succeeded. If so, the interrupt handler acks it by decrementing the appropriate counter of outstanding reads or writes. Otherwise, it throws a rollback exception to the interpreter. A special block in the interpreter handles this exception by performing the three step procedure discussed in Section 3.

4.1.2 Software Detection

Not all peripheral interfaces provide a convenient interrupt mechanism to detect success and failure. For example, the interrupts for the UART interface on the LM3S9B92 are designed to allow asynchronous transmission and reception of data; thus, they convey whether or not the device is ready for a subsequent transfer, not whether previous transfers have succeeded or failed.

To address this limitation, the system was augmented with a small buffer to hold UART data. Each entry contains a byte that was received by the UART and associated error bits. As the hardware receives data, the interrupt handler transfers the data, and any error bits, to the software buffer. UART reads made by the application then access the software buffer. When a read occurs on a byte that was successfully transferred, the peripheral access is acked, as if there had been a "success" interrupt. When a read occurs on a byte that experienced a transmission error, the recovery procedure is triggered, as if a "failure" interrupt had occurred. This is a general approach that can be used for any peripheral for which the interface does not provide interrupts that enable the system to determine the success or failure of each access.

4.2 Checkpointing

To enable rollback and re-execution, the Phoenix system performs checkpointing whenever there is a chance of peripheral failure. Checkpointing is automatically enabled prior to each peripheral access. While checkpointing is enabled, Phoenix maintains three structures: a journal, a control flow queue, and a set of rematerialization queues. These structures persist past rollback, as they live on a second recovery heap which is not checkpointed. They are freed on a rolling basis as accesses succeed. Further, once all past accesses have succeeded, rollback past the current point is impossible, so Phoenix disables checkpointing until the next access is issued.

4.2.1 Software Journal

In Owl, all of the program state is stored on the Python heap. So, when checkpointing is enabled, the history of this heap must be tracked to enable rollback of the internal program state. When an assembly language store to this heap occurs, Phoenix first records a journal entry containing the memory address to be stored and its old contents. The journal only needs space proportional to the number of stores to the Python heap since checkpointing was last enabled, rather than the entire size of the heap. On failure, the run-time system can roll back the state of the Python heap by restoring the contents of the journal in reverse order.

The prototype implementation of Phoenix keeps the journal using software that exploits existing hardware mechanisms defined by the ARM ISA. In particular, Phoenix uses the ARM Cortex-M3's memory protection unit (MPU), flash remap hardware, and system call instruction (SVC) to keep the journal. These are the only hardware requirements of Phoenix.

The journaling process proceeds as follows. First, when checkpointing is enabled, the regions of memory covering the Python heap are set read-only. This causes the memory management fault handler to run upon each store to this heap. It receives both the address of the instruction that caused the fault and the memory address of the faulting access. Based on the type of instruction (ARM has "store multiple" instructions), it determines how many journal entries to populate. Each entry contains a faulting address and the contents of that address.

If the journal fills, an exception is thrown telling the system to wait for prior peripheral accesses to be acked. Because the failure detection mechanisms only write to the recovery heap — which is not checkpointed — as opposed to the Python heap, there is no chance of deadlock. Once all accesses have succeeded, the system clears the

journal and disables logging. Execution can then resume.

After the store instruction completes, memory protection must be turned back on. The fault handler remaps the instruction that follows the store to become an SVC instruction. Note, however, that the program is executing from flash memory, so the instruction cannot be overwritten. Instead, Phoenix uses the Cortex-M3 *flash patch and breakpoint* support, which allows small regions of flash to be "patched" by remapping them to SRAM. When the system call executes, MPU protection is turned back on. The flash remapping is turned off, and the return from the SVC is routed back to the instruction following the store.

The software journal is effective, but it incurs an overhead for each protected store to the Python heap. Since two exception handlers must run for each protected store, there is a minimum overhead of 48 cycles (12 cycles to enter and 12 cycles to exit each exception handler), plus any pipeline flushes. In practice, the handlers themselves execute for hundreds of additional cycles; during evaluation on a series of microbenchmarks, the overhead per protected store averaged 308 cycles.

4.2.2 Hardware Journal

To reduce the journal overhead, the microcontroller could be augmented with a simple hardware journal. This would involve minimal changes to the ARM ISA: MPU support for an additional "log" mode, a circular buffer with entries that can hold a memory location and a 32-bit value to serve as the journal, and a pair of registers to store head and tail pointers into the journal. If the tail pointer ever becomes equal to the head pointer, the journal is full, and an interrupt would be generated. In response to such an interrupt, the Phoenix system would wait, as described in Section 4.2.1.

When a store instruction writes to memory in a region that has the log attribute set, the hardware would first store the address and the contents at that address in the journal, incrementing the tail pointer modulo the size of the journal. The stage of the store instruction that writes to SRAM would complete in three cycles instead of one. First, the original write to SRAM would be aborted. Second, the contents of SRAM would be read. Last, three writes would occur in parallel: the original write, the journal write, and the tail pointer update. This two cycle overhead is two orders of magnitude lower than that of the software journal, which would yield perceptible improvements for applications with frequent journaling.

4.2.3 Rematerialization Queues

The rematerialization queues checkpoint the external state. Each peripheral has a queue, with one entry per access. These entries serve two purposes: storing the arguments and return value for use during redo mode, and storing the rollback point in case of failure. Since peripheral accesses are structured as native function calls, the return value is not raw data from the peripheral but rather a Python object allocated on the heap. The rematerialization queue holds deep copies of these objects, allocated on the recovery heap in order to persist past rollback.

On failure, Phoenix selects the right checkpoint based on the identity of the failed peripheral. Each checkpoint corresponds to a single access, so metadata for a given checkpoint is stored in the corresponding queue entry: indices into the journal and control flow queue, as well as pointers into the other rematerialization queues.

After rollback, the system constructs a set of peripherals to redo based on programmer-specified dependencies. Each time a peripheral access call is reached during redo mode, Phoenix checks to see if the peripheral is in the redo set. If so, the function is called as usual. Otherwise, Phoenix compares the new arguments against those in the rematerialization queue entry. If they match, the function call is skipped; Phoenix pops the new arguments from the stack and pushes the old return value. If they do not match, the peripheral is added to the redo set and the function call is re-executed with the new arguments.

A given rematerialization queue entry is freed once the peripheral access it corresponds to, plus all earlier accesses, are acked, since these acks guarantee that the system will never roll back to (or past) this entry. The contents of the journal up to the index stored within this rematerialization queue entry are freed at the same time.

4.2.4 Control Flow Queue

When checkpointing is enabled, the control flow queue keeps track of the instruction pointer of each Python bytecode that is executed. During redo mode, this queue is used to determine when to resume normal execution. Redo mode is exited either when the point at which failure was detected is reached once more, or when control flow diverges from the original path. For instance, a re-executed peripheral read may return a different value the second time through and cause a different path to be taken at a branch. Phoenix would then exit redo mode. This allows the system to adapt to changes in the external state that have occurred since the initial execution.

4.3 Compilation

To enable rollback, Phoenix records the journal and control flow queue indices at each checkpoint. Checkpoints correspond to peripheral access function calls — in Python, CALL_FUNCTION bytecodes. However, it is insufficient to re-execute the failed function call with the same arguments, as they may change during recovery. For instance, an I2C device access may take the master address as a parameter. If this address was passed as a variable, and this variable was changed by the recovery function, its new value must be loaded prior to

re-execution. As such, the true checkpoint is not the CALL_FUNCTION bytecode but the first bytecode that loads an argument for the call. These checkpoint locations are difficult to identify at run-time. Thus, Phoenix introduces a new bytecode, JOURNAL_STORE, and adds a custom AST rewriter to the Python compiler which inserts a JOURNAL_STORE prior to loading the arguments for each peripheral access. On reaching this bytecode at run-time, the interpreter stores the current journal and control flow indices and enables checkpointing.

Additionally, Phoenix uses an autowrapping tool to inject code before and after each peripheral access. Prior to a peripheral access, the autowrapper adds code to create a rematerialization queue entry. After the access, it inserts code to increment the number of outstanding reads or writes to this peripheral and set the new rematerialization queue entry's return value. These additions facilitate the run-time system's job in two ways. First, tracking outstanding accesses allows the system to automatically free the checkpointing structures and disable checkpointing as accesses complete. Second, allocating queue entries is a prerequisite for rematerialization.

5 Programmer Mechanisms

Phoenix disentangles the bulk of the application code from the peripheral code. Making this peripheral code recoverable requires that the programmer follow a few simple rules during development, involving minor refactoring and a small amount of new code. For the three applications evaluated in Section 6.2, an additional 9–17 lines of Python and 66–76 lines of C were required.

5.1 Peripheral Classification

Each peripheral must extend one of four provided peripheral classes, which mirror the categories from Section 2.1; an example is shown in Figure 3. Applications using the same hardware peripheral can share the same peripheral class. The class is largely written in Python; the only C code that the programmer must write is that of the low-level access functions. These can then be called from the Python code via Owl's native function interface.

Peripheral access functions should encapsulate an atomic unit of work, since rollback and re-execution occur at the granularity of a single function call. Additionally, each peripheral class must support recovery. All peripheral classes must define a `recover` method, which the system automatically calls after rollback. This gives the programmer flexibility in deciding how to handle a failed peripheral, such as by resetting it, switching in a hot spare, or even signaling for user intervention. Yet, the programmer need never worry about the complex bookkeeping and control flow logic to determine when to call this method. This is the only method that must be defined for stateless and ephemeral peripherals.

```
1  class Motor(PersistentPeripheral):
2      def __init__(self):
3          # Initialize primary device
4          self.init(PRIMARY)
5
6      def recover(self):
7          # Switch to backup device
8          self.init(BACKUP)
9
10     def safe_state(self):
11         self.set_speed(0)
12
13     def last_state(self, *args):
14         native_write(*args)
```

Figure 3: Peripheral Class Excerpt

Because persistent peripherals have lasting state, they require two additional restoration methods, motivated by the final insight in Section 2.1: `safe_state` and `last_state`. Before rollback, Phoenix automatically invokes all `safe_state` methods to put the peripherals into states that will have minimal effects during subsequent recovery steps. For example, a motor might be set to a speed of zero. As with `recover`, the definition of `safe_state` is up to the programmer.

Defining `last_state` is trivial, as it follows a standard template: take a variable number of arguments and perform a write with those arguments. This allows Phoenix to invoke `last_state` by simply passing the arguments from a rematerialization queue entry, regardless of the write function's signature. During redo mode, the system calls `last_state` in two places. Peripherals whose accesses must be replayed are set to their last state at the beginning of redo mode; peripherals whose accesses were not replayed during redo mode are set to their last state prior to resuming normal execution.

Restoring a historical peripheral is challenging, as its state is an aggregation of multiple past writes. The naive solution is to re-execute every past write. However, this would prevent freeing the rematerialization queues, which is untenable given limited memory. While Phoenix does not currently include specialized mechanisms for historical peripherals, this does not preclude their use with Phoenix. Consider an LCD display. Many applications will update a display by completely redrawing its contents, which would work seamlessly with Phoenix. More generally, a finite number of pixels comprise the display's state. Their values could be stored in a buffer programmatically and restored by `recover`, requiring no extra support from Phoenix.

5.2 Peripheral Initialization

To enable failure detection, the peripheral class must do two things upon initialization: register the new instance and enable interrupts. Phoenix provides a

peripheral_register function which must be invoked with the interface and interface details for the peripheral, as described in Section 4.1.1. Since interrupt handlers detect success or failure, and interface information is needed to determine which peripheral generated a given interrupt, registering peripherals and enabling interrupts must occur prior to any peripheral accesses. However, registration and interrupt information can be updated at any time to reflect changes to the hardware configuration. In particular, if the recover function switches to a backup device, it should also re-register the peripheral and enable interrupts for any new interfaces.

5.3 Config File

Last, the programmer must provide a config file for each application containing peripheral metadata. First, the programmer must declare dependencies. Second, the programmer must specify how many interrupts the peripheral access functions generate. This allows Phoenix to treat an access as an atomic unit and determine when it has completed. Writing the config file proved trivial for the three applications studied in this paper.

6 Evaluation

This section will first introduce the microbenchmarks and applications used to evaluate the Phoenix system. It will then assess Phoenix, showing that its space overhead is minimal and its time overhead is completely hidden in realistic workloads.

6.1 Microbenchmarks

The microbenchmarks use two persistent peripherals, a gyroscope and compass, and follow a common structure. They are named in the form <peripherals>_<actions>, where <peripherals> is a subset of {gyro, comp} and <actions> is a subset of {r, w, c}, for read, write, compute. After the peripherals are initialized, the actions are performed in a loop in the order in which they are listed.

Even in the absence of failures, checkpointing necessarily incurs a time overhead, primarily due to journaling. Across all of the benchmarks listed in Table 1, the weighted average cost of adding an entry to the software journal was 6.2 μs. Given a 20 ns cycle length, this means that one journaled store took, on average, 308 cycles. A hardware journal, with an overhead of only 2 cycles per store, would yield dramatic improvements. Introducing a single failure incurred a relatively small additional overhead of 12–143 ms. This overhead did not exceed 45 ms for benchmarks with one peripheral; for gyro_comp_wr the overhead was larger due to additional safe_state and last_state calls.

This overhead is incurred in support of speculative execution. Another approach would be to wait for each access to be acked before proceeding. However, the length of the control flow queue reveals that Phoenix takes advantage of significant opportunities to make progress where a stop-and-wait system would not. As shown in Tables 1 and 5, the control flow queue reached lengths of 9–18 during the period in which a wait-based system would be stalling. This is non-trivial; one Python bytecode may, for example, execute an entire native function.

Table 1: Benchmark Checkpointing Structures, With and Without Failure (max live entries)

Benchmark	JNL		CFQ		RMQ (comp, gyro)	
	Failure?		Failure?		Failure?	
	No	Yes	No	Yes	No	Yes
gyro_r	220	220	9	14	(0, 2)	(0, 3)
gyro_w	165	182	7	14	(0, 2)	(0, 3)
gyro_wr	220	220	9	14	(0, 2)	(0, 3)
comp_r	144	169	6	6	(1, 0)	(2, 0)
comp_w	192	226	9	9	(2, 0)	(3, 0)
comp_wr	192	211	9	9	(2, 0)	(3, 0)
gyro_comp_wr	220	301	9	11	(2, 2)	(3, 2)
comp_wcr	190	219	9	9	(2, 0)	(3, 0)
comp_wrc	190	211	9	9	(2, 0)	(3, 0)
comp_wcrc	192	211	9	9	(2, 0)	(3, 0)

Just as the checkpointing process takes time, the structures storing the state require space. There are three checkpointing structures: the journal (JNL), control flow queue (CFQ), and rematerialization queues (RMQ). Each journal entry requires 6 bytes, as the software journal is optimized to store an offset into the 64 KB heap rather than the full address. A control flow queue entry requires 4 bytes. Phoenix uses a fixed-size journal of 512 entries, and a fixed-size control flow queue of 64 entries. Rematerialization queues are linked lists. Each entry consumes a minimum of (36 + (72 * n)) bytes, where n is the number of hardware peripherals, plus variable space for arguments. Owl's best-fit memory allocator also introduces a small amount of variance.

A long-running program may generate many entries in these structures over the course of its execution. However, since past entries are discarded as acks are received, very few entries are live at the same point in time. Table 1 shows the maximum number of live entries for each of the three checkpointing structures when the benchmarks were run with no failures and with a single failure. No benchmark required more than 301 journal entries, 14 control flow queue entries, or 3 rematerialization queue entries per peripheral. Thus, the fixed sizes for the journal and control flow queue proved more than sufficient, never exceeding 59% or 22% capacity, respectively.

Tables 2 and 3 show the total space overhead with and without failure. The checkpointing structures are the primary source of this overhead. However, Phoenix additionally maintains a small amount of metadata. Peripheral metadata tracks the active peripherals in the system,

Table 2: Benchmark Overhead, Without Failure (bytes)

Benchmark	comp_r	comp_w	comp_wr	comp_wcrc	gyro_r	gyro_w	gyro_wr	gyro_comp_wr
JNL	3120	3120	3120	3120	3120	3120	3120	3120
CFQ	284	284	284	284	284	284	284	284
RMQ	276	472	472	472	484	484	484	752
Peripheral Metadata	268	268	268	268	268	268	268	384
Recovery Metadata	24	24	24	24	24	24	24	24
Total	3972	4168	4168	4168	4180	4180	4180	4564

Table 3: Benchmark Overhead, With Failure (bytes)

Benchmark	comp_r	comp_w	comp_wr	comp_wcrc	gyro_r	gyro_w	gyro_wr	gyro_comp_wr
JNL	3120	3120	3120	3120	3120	3120	3120	3120
CFQ	284	284	284	284	284	284	284	284
RMQ	664	656	656	656	656	656	656	984
Peripheral Metadata	268	268	268	268	268	268	268	384
Recovery Metadata	304	300	300	300	304	304	304	300
Total	4436	4636	4636	4636	4632	4632	4632	5072

growing with the number of peripherals. Some of this metadata is generated at boot-time based on data parsed from the config file; the rest is created when a peripheral is initialized at run-time. The same build of the run-time system was used for all benchmarks; thus, there were two peripherals' worth of boot-time metadata in all of the benchmarks, and the total peripheral metadata size does not double on activating the second peripheral.

Recovery metadata contains information needed to perform rollback and re-execution, including which peripheral failed and which peripherals to redo. Thus, the size of this metadata is initially negligible and grows when a failure occurs. The combined overhead of both types of metadata never exceeded 408 B (gyro_comp_wr) in the benchmarks without failure; with failure, it reached a maximum of only 684 B (gyro_comp_wr).

The total space overhead of Phoenix was relatively consistent across all benchmarks. With no failures, it began at 3.9 KB for the case where a single peripheral was read but not written (comp_r), as this minimized the number of live rematerialization queue entries. Introducing writes (comp_w, comp_wr, comp_wcrc) added an additional 196 B, as a second live rematerialization queue entry is required to support `last_state`. The gyro benchmarks did not show this same increase upon adding writes, as the gyro requires a single write during initialization, and therefore even gyro_r held a second rematerialization queue entry. The largest increase occurs when additional active peripherals are introduced, as in gyro_comp_wr. This is due to two factors: extra rematerialization queue entries and extra peripheral metadata.

These space requirements do not change drastically in the case of failure; only the rematerialization queue entries and the recovery metadata consume additional space. The expansion of the rematerialization queues is due to the fact that the system maintains segments of the queues from the original execution in order to replay accesses. At the same time, replaying these accesses results in additional entries being generated. Thus, during redo mode there are brief periods in which two rematerialization queue entries exist for the same access.

Table 4: Car Interval Length (ms)

Benchmark	Minimum	Maximum	Average
Without Phoenix	30	44	30
Phoenix (No Failure)	30	44	30
Phoenix (Failure)	30	44	30

Overall, the space overhead of the Phoenix system is quite small. Across all benchmarks, the overhead never exceeded 4.5 KB when no failures occurred, nor did it exceed 5.0 KB when one failure occurred. Using a mere 4.0–5.2% of SRAM, Phoenix leaves most of the space available for the running program. At the same time, it maintains multiple simultaneous checkpoints, each of which is positioned at precisely the latest possible location to which Phoenix could roll back in order to recover.

6.2 Applications

Phoenix was also evaluated on three applications, each representative of a different pattern of peripheral accesses. The first application is an autonomous RC car. The microcontroller is attached to three types of peripherals: a motor, a steering servo, and two gyroscopes. An event loop controls the car's movements by reading from the gyro and writing to the motor and steering servo. The second application is an obstacle tracker which periodically logs the distance to the nearest obstacle by reading from one of two range finders and writing to a display.

Table 5: Application Checkpointing Structures, With and Without Failure (max live entries)

Benchmark	JNL		CFQ		RMQ (comp, display, finder, gyro, motor)	
	Failure?		Failure?		Failure?	
	No	Yes	No	Yes	No	Yes
autonomous car	220	220	9	15	(0, 0, 0, 2, 2)	(0, 0, 0, 3, 2)
obstacle tracker	346	346	18	18	(0, 2, 1, 0, 0)	(0, 2, 2, 0, 0)
virtual compass	346	346	18	18	(1, 2, 0, 0, 0)	(2, 2, 0, 0, 0)

Table 6: Application Overhead, No Failure (bytes)

Application	autonomous car	obstacle tracker	virtual compass
JNL	3120	3120	3120
CFQ	284	284	284
RMQ	876	1184	1360
Peripheral Metadata	476	352	368
Recovery Metadata	36	24	24
Total	4792	4964	5156

Table 7: Application Overhead, With Failure (bytes)

Application	autonomous car	obstacle tracker	virtual compass
JNL	3120	3120	3120
CFQ	284	284	284
RMQ	1068	1084	1584
Peripheral Metadata	476	352	368
Recovery Metadata	336	294	312
Total	5280	5134	5668

The final application uses two compasses and a display to draw a virtual compass pointing towards magnetic North. The range finder is a stateless peripheral; the compass, gyro, steering servo, and motor are persistent peripherals; and the display is a historical peripheral. Each application was evaluated on three configurations, for ten seconds each: Owl without Phoenix, Phoenix with no failure, and Phoenix with a single failure.

The autonomous car uses a control loop to query the gyro and steer. It attempts to hit a specific period (30 ms) between updates, where an update consists of reading the gyro and setting the servo. Table 4 shows the minimum, maximum, and average interval lengths, which were identical across all configurations. On average, the car hit its soft real-time deadline of 30 ms. Yet, the maximum was 44 ms. This spike was caused by ill-timed runs of Owl's mark-and-sweep garbage collector.

The obstacle tracker reads the range finder and displays the distance to the nearest obstacle. Instead of aiming for a set period between updates, it sleeps for a fixed amount of time. As a result, sleep dominates the workload, and all configurations completed the same number of iterations in the allotted time.

In contrast, the virtual compass is peripheral access-intensive. It attempts to update as frequently as possible; on each iteration, it reads the compass and draws an arrow on the display. Without Phoenix, each iteration took, on average, 1005 ms. With Phoenix enabled, an iteration averaged 1862 ms. The main reason for this slowdown is that peripheral accesses are so frequent that checkpointing was nearly always enabled. This is compounded by the fact that the display is inherently slow, as a separate native write is required for each pixel.

While the performance of the applications varied due to their disparate update patterns, the space overhead was consistently small. As seen in Table 5, each application fit easily within the 512-slot journal and 64-slot control flow queue, and the maximum rematerialization queue length was three. The total space overhead, presented in Tables 6 and 7, was comparable to that of the benchmarks, requiring a maximum of 5.0 KB (virtual compass) when no failure occurred and a maximum of 5.5 KB (virtual compass) when a single failure occurred. This overhead encompasses multiple checkpoints, one per outstanding peripheral access. In contrast, a traditional checkpointing system would require a complete copy of the heap (64 KB) for each individual checkpoint.

Reliability cannot come for free; it demands tradeoffs in time and space. Phoenix optimizes for both, saving time by enabling rollback to the exact point of failure and saving space by logging only that which is absolutely necessary. Still, a time overhead is perceptible in peripheral-intensive workloads. In such workloads, the costs of improved reliability may outweigh the benefits; yet, if reliability is critical, this tradeoff may well be worthwhile. Further, this overhead could largely be eliminated by employing a hardware journal.

However, Phoenix is well-suited to applications that access peripherals periodically; these applications experienced no observable delays during evaluation. Such access patterns are more characteristic of typical embedded workloads, which periodically monitor their surroundings via sensors and react to discrete events.

7 Related Work

Fault tolerance has been well studied in distributed systems. The elements of these systems collaborate to provide a reliable service to external clients using redun-

dancy and consensus to detect, mask, and recover from failures [15, 21, 27, 28, 36]. However, each element of the system must be able to actively participate in the reliability protocols and communicate its current status. In contrast, peripherals in embedded systems are dedicated to a specific task and are unable to participate in specialized reliability protocols.

There has also been much work on protecting conventional computing systems from device driver failure. In such systems, the operating system has reliability features that allow it to tolerate device failures if the drivers behave properly. So, the focus has been on hardening device drivers and protecting the interface between the operating system and the device driver [18, 23, 24, 29, 40, 44, 45]. Phoenix instead operates at the level of hardware peripheral reads and writes, targeting systems which lack the heavier weight isolation and protection mechanisms of device drivers and conventional operating systems.

The difficulty of exhaustively addressing all possible failure scenarios in embedded applications is widely recognized [9, 11, 30, 49]. Yet, existing techniques leave much of the burden on the programmer, resulting in increased development costs and poor scalability [32]. Efforts have been made to raise the level of abstraction of writing fault-tolerant embedded applications through model- or template-driven development [11, 49] and programming language primitives [9]. However, these application-level approaches rely on the programmer to apply the correct constructs in the right places, a non-trivial task in the face of asynchronous failures.

One recovery methodology with significant traction is rollback and re-execution [6, 12, 16, 25, 35, 37, 39, 42]. State-of-the-art rollback relies on checkpoints, or snapshots, of the system state. Checkpoints are typically either taken periodically by the system, or inserted manually by the programmer [6, 12]. Traditional checkpointing algorithms maintain one or more complete copies of the memory space; while this has been successful in mobile and distributed systems [25, 35, 39], such an overhead is infeasible in a resource-constrained embedded system. Instead of taking snapshots of the entire program state, Phoenix utilizes a logging technique that resembles journaling filesystems [10, 17, 26, 38] and some hardware transactional memory proposals [34]. Though this adds an overhead to some stores, it ensures that the system only copies the subset of memory that has been changed. Further, automated disabling of logging minimizes the number of stores burdened by this overhead.

Moreover, traditional snapshots encapsulate only the internal program state, which can be restored via rollback and re-execution. In the worst case, internal state can be restored by re-executing the entire program. In fact, a full reboot is a common approach to recovery in embedded systems [14, 33]. However, critical application state is likely to be lost on a reboot. To preserve state and minimize the amount of work that is re-executed, algorithms for mobile and distributed systems have been optimized to re-execute only a subset of processes [37, 39] or threads [13, 25] based on dependencies which are either implicitly established via messages or explicitly annotated by the compiler.

Yet, minimal efforts have been made to restore external state. Past work supplementing checkpointing with message logging acknowledged the existence of external state in the form of messages, and attempted to deal with it by skipping all message sends during re-execution [37, 42]. However, this policy leaves no room to adapt to changes during re-execution.

8 Conclusions

As embedded run-time systems grow in popularity, they demand mechanisms for increased reliability. At the same time, they present new opportunities for automating reliability at the system level. This paper has presented the design and implementation of Phoenix, a novel system for surviving peripheral failures in embedded run-time systems. Phoenix is composed of integrated system- and application-level mechanisms, which work together to efficiently record the system state and automatically recover from asynchronous peripheral failures.

Several new insights motivated the design of Phoenix. In particular, peripherals interact with the real world and with each other in ways that are substantively different than internal program interactions. Based on this, one of the key innovations of Phoenix is a novel, lightweight checkpointing system to efficiently track both internal and external state. After a failure, this enables Phoenix not only to reset the internal program state, but also to restore the external peripheral state by determining whether each peripheral access must be re-executed or rematerialized.

The microcontrollers that Phoenix targets are severely resource-constrained. Phoenix guarantees rollback to the precise point at which a failed peripheral access occurred, re-executing the minimal necessary set of actions during recovery. Further, two of the three applications on which Phoenix was evaluated experienced no perceivable overhead during normal system operation. Finally, Phoenix used no more than 6 KB to log both internal and external state for these applications.

Embedded systems interact with the real world by controlling actuators based on sensory inputs. The peripherals enabling these interactions are thus fundamental components of the system and must be reliable. By providing a complete recovery process that addresses the unique challenges of resource-constrained embedded systems, Phoenix is an important step towards improving the future of writing reliable embedded applications.

References

[1] eLua. http://www.eluaproject.net/.

[2] Micro python. http://micropython.org/.

[3] The Owl embedded Python system. http://www.embeddedpython.org/.

[4] python-on-a-chip. http://code.google.com/p/python-on-a-chip/.

[5] AUSTIN, T. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the International Symposium on Microarchitecture* (1999).

[6] BARBOSA, R., AND KARLSSON, J. On the integrity of lightweight checkpoints. In *Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium* (Nanjing, China, 2008).

[7] BARR, T. W., SMITH, R., AND RIXNER, S. Design and implementation of an embedded Python run-time system. In *Proceedings of the USENIX Annual Technical Conference* (Boston, Massachusetts, 2012).

[8] BASHIRI, M., MIREMADI, S. G., AND FAZELI, M. A checkpointing technique for rollback error recovery in embedded systems. In *Proceedings of the 18th International Conference on Microelectrics* (2006).

[9] BECKMAN, N., AND ALDRICH, J. A programming model for failure-prone, collaborative robots. In *Proceedings of the 2nd International Workshop on Software Development and Integration in Robotics* (Rome, Italy, 2007).

[10] BEST, S. JFS log: How the journaled file system performs logging. In *Proceedings of the 4th Annual Linux Showcase and Conference* (Berkeley, California, 2000).

[11] BUCKL, C., KNOLL, A., AND SCHROTT, G. Model-based development of fault-tolerant embedded software. In *2nd International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation* (Paphos, Cyrpus, 2007).

[12] CHANDY, K. M., AND RAMAMOORTHY, C. V. Rollback and recovery strategies for computer programs. *IEEE Transactions on Computers C-21* (1972).

[13] CHEN, Y., GNAWALI, O., KAZANDJIEVA, M., LEVIS, P., AND REGEHR, J. Surviving sensor network software faults. In *Proceedings of the Symposium on Operating Systems Principles* (2009).

[14] COOPRIDER, N., ARCHER, W., EIDE, E., GAY, D., AND REGEHR, J. Efficient memory safety for TinyOS. In *Proceedings of the International Conference on Embedded Networked Sensor Systems* (2007).

[15] CORBETT, J., DEAN, J., EPSTEIN, M., FIKES, A., FROST, C., FURMAN, J., GEMAWAT, S., GUBAREV, A., HEISER, C., HOCHSCHILD, P., HSIEH, W., KANTHAK, S., KOGAN, E., LI, H., LLOYD, A., MELNIK, S., MWWAURA, D., NAGLE, D., QUINLAN, S., RAO, R., ROLIG, L., SAITO, Y., SZYMANIAK, M., TAYLOR, C., WANG, R., AND WOOKDORD, D. Spanner: Google's globally-distributed database. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation (OSDI)* (2012).

[16] CUNEI, A., AND VITEK, J. A new approach to real-time checkpointing. In *Proceedings of the 2nd International Conference on Virtual Execution Environments* (Ottawa, Ontario, Canada, 2006).

[17] CZEZATKE, C., AND ERTL, A. M. LinLogFS: A log-structured file system for Linux. In *Proceedings of the USENIX Annual Technical Conference* (San Diego, California, 2000).

[18] GANAPATHY, V., RENZELMANN, M., BALAKRISHNAN, A., SWIFT, M., AND JHA, S. The design and implementation of microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (2008).

[19] GHAHROODI, M. M., OZER, E., AND BULL, D. SEU and SET-tolerant ARM Cortex-R4 CPU for space and avionics applications. In *Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale* (Avignon, France, 2013).

[20] GOPAL, K., RAMALAKSHMI, K., AND PRIYADHARSINI, C. Wide area network fault detection and routing model for wireless sensor networks. *International Journal of Communication and Computer Technologies 2, no. 7* (2014).

[21] HUNT, P., KONAR, M., JUNQUEIRA, F., AND REED, B. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the USENIX Annual Technical Conference* (2010).

[22] IZOSIMOV, V., POP, P., ELES, P., AND PENG, Z. Design optimization of time- and code-constrained fault-tolerant distributed embedded systems. In *Proceedings of the Design, Automation, and Test in Europe Conference and Exhibition* (Munich, Germany, 2005).

[23] KADAV, A., RENZELMANN, M. J., AND SWIFT, M. M. Tolerating hardware device failures in software. In *Proceedings of the Symposium on Operating Systems Principles* (2009).

[24] KADAV, A., RENZELMANN, M. J., AND SWIFT, M. M. Fine-grained fault tolerance using device checkpoints. In *Proceeedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (2013).

[25] KASBEKAR, M., AND DAS, C. R. Selective checkpointing and rollbacks in multithreaded distributed systems. In *Proceedings of the 21st International Conference on Distributed Computing Systems* (Mesa, Arizona, 2001).

[26] KONISHI, R., AMAGAI, Y., SATO, K., HIFUMI, H., KIHARA, S., AND MORIAI, S. The Linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review 40* (2006).

[27] LAMPORT, L. The part-time parliament. *ACM Transactions on Computer Systems 16* (1998).

[28] LAMPORT, L. Paxos made simple. *ACM SIGACT News 32* (2001).

[29] LEVASSEUR, J., UHLIG, V., STOESS, J., AND GOTZ, S. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the Symposium on Operating Systems Design and Implementation* (2004).

[30] LUTZ, R. R. Analyzing software requirements errors in safety-critical, embedded systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering* (San Diego, California, 1992).

[31] LYONS, W. Enabling increased safety with fault robustness in microcontroller applications.

[32] MARIANI, R., FUHRMANN, P., AND VITTORELLI, B. Fault-robust microcontrollers for automotive applications. In *Proceedings of the 12th IEEE International On-Line Testing Symposium* (Como, Italy, 2006).

[33] MATIJEVIC, J., AND DEWELL, E. Anomaly recovery and the Mars exploration rovers. In *Proceedings of SpaceOps* (2006).

[34] MOORE, K. E., BOBBA, J., MORAVAN, M. J., HILL, M. D., AND WOOD, D. A. LogTM: Log-based transactional memory. In *Proceedings of the International Symposium on High Performance Computer Architecture* (2006).

[35] NEOGY, S., SINHA, A., AND DAS, P. K. Selective recovery in distributed systems. In *TENCON* (Chiang Mai, Thailand, 2004).

[36] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference* (2014).

[37] PARK, T., WOO, N., AND YEOM, H. Y. An efficient optimistic message logging scheme for recoverable mobile computing systems. *IEEE Transactions on Mobile Computing 1* (2002).

[38] PIERNAS, J., CORTES, T., AND GARCIA, J. M. DualFS: a new journaling file system without meta-data duplication. In *Proceedings of the 16th International Conference on Supercomputing* (New York, New York, 2006).

[39] PRAKASH, R., AND SINGHAL, M. Low-cost checkpointing and failure recovery in mobile computing systems. *IEEE Transactions on Parallel and Distributed Systems 7* (1996).

[40] RENZELMANN, M. J., AND SWIFT, M. M. Decaf: Moving device drivers to a modern language. In *Proceedings of the USENIX Annual Technical Conference* (2009).

[41] ROTENBERG, E. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the International Symposium on Fault Tolerant Computing* (1999).

[42] SARIDAKIS, T. Design patterns for log-based rollback recovery. In *Proceedings of the 2nd Nordic Conference on Pattern Languages of Programs* (Boahtteigi, Saariselk, Finland, 2003).

[43] SUNDARAMOORTHY, K., PURSER, Z., AND ROTENBERG, E. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2000).

[44] SWIFT, M., ANNAMALAI, M., BERSHAD, B. N., AND LEVY, H. M. Recovering device drivers. *ACM Transactions on Computer Systems 24*, 4 (2006).

[45] SWIFT, M., BERSHAD, B. N., AND LEVY, H. M. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems 23*, 1 (2005).

[46] TABKHI, H., MIREMADI, S. G., AND EJLALI, A. An asymmetric checkpointing and rollback error recovery scheme for embedded processors. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems* (Boston, Massachusetts, 2008).

[47] TEXAS INSTRUMENTS. *Stellaris LM3S9B92 Microcontroller Data Sheet*, Oct. 5, 2012.

[48] WATTANAPONGSKORN, N., AND COIT, D. W. Fault-tolerant embedded system design and optimization consdering reliability estimation uncertainty. In *Reliability Engineering and System Safety* (2007).

[49] WILLIAMS, B. C., INGHAM, M. D., CHUNG, S. H., AND ELLIOTT, P. H. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE 91* (2003).

[50] ZHANG, Y., AND CHAKRABARTY, K. Fault recovery based on checkpointing for hard real-time embedded systems. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems* (2003).