r-Kernel: An Operating System Foundation for Highly Reliable Networked Embedded Systems

Qing Cao*, Xiaorui Wang*, Hairong Qi* and Tian He†
*Department of Electrical Engineering and Computer Science
University of Tennessee, Knoxville, TN 37996
†Department of Computer Science and Engineering
University of Minnesota, Minneapolis, MN 55455
Email: {cao, xwang, hqi}@utk.edu, tianhe@cs.umn.edu

***Abstract*—In this paper, we present r-kernel, an operating system kernel enhancement specifically designed to improve software reliability in networked embedded systems. The key novelty of r-kernel lies in that it exploits the time dimension of software execution to improve robustness. Specifically, r-kernel keeps track of the execution of applications through checkpoints. If one application has been determined to have failed, r-kernel performs rollback operations to restore its state to one of those checkpoints created earlier. For the second round of operation, r-kernel provides a safe mode environment to avoid triggering the same bugs, by dynamically redirecting system calls made by the application. Finally, if the whole system is crashed, r-kernel relies on watchdog timers to reset the node, and develops a technique called past-run reconstruction to locate and report the thread that had caused the system failure. We have implemented r-kernel based on the LiteOS operating system kernel running on the popular MicaZ platform. We demonstrate that it achieves desired goals with acceptable overhead.**

I. Introduction

One key feature of the emerging networked embedded systems, such as wireless sensor networks, is the tight interaction of software modules with the physical world [1], [2]. Because of the tremendous cost of system failures, enormous efforts are invested in debugging and testing prior to system deployment. However, as software complexity increases, failures may still occur after deployment since some bugs could slip through even the most thorough debugging and testing. Further, the unique nature and deployment environment of networked embedded systems, combined with their limited system resources, make them particularly vulnerable to new types of bugs such as those caused by interaction with the physical environment [3]. When these bugs cause failures, there are typically no patches available yet. This considerably worsens the situation, especially for those deployments that are meant to be autonomous, long term, and without human intervention.

To ensure system reliability, the most urgent thing for networked embedded systems is to ensure that the normal system operation is not interrupted despite component failures. To achieve this, the system should be robust and resilient. One commonly followed approach in embedded software development is using watchdog timers. In the normal execution of an application, the watchdog timer is reset periodically to prevent system reboots. If the system hangs, the watchdog timer will eventually time out and cause a system reboot.

While reboots have the obvious advantage to restore a system to its initial state, as it is the most tested and well-known, they also have considerable drawbacks in causing data loss and energy overhead. For example, in routing protocols of sensor networks, it is common practice for sensor nodes to keep neighbor information such as their IDs and link quality into an internal neighbor table. After reboots, such information will be lost. Reconstructing neighbor tables will incur extra energy overhead. As another example, consider an application consisting of multiple threads. If one thread fails, the consequent system reboot will cause every thread on the node to be restarted, leading to unnecessary energy costs.

The second, less obvious problem with reboots is that while they can avoid certain non-deterministic bugs in the re-execution of the same program (i.e. bugs that do not always occur in re-executions), they are useless to handle deterministic bugs, such as those caused by memory leaks or stack overflows. In fact, the same bug will likely cause reboots again and again until the energy runs out, or a patch is made available. For example, a bug related to priority inversion caused the Mars Pathfinder probe to reset again and again with the use of a watchdog timer, until a patch was available to fix this bug [4].

To address the above problems, we present r-kernel, a comprehensive, holistic solution to improve the robustness of embedded software against a wide range of failures. It targets on extremely resource constrained platforms as represented by MicaZ nodes, therefore, its design decisions have to be tailored to reduce overhead and improve efficiency.

The central idea of r-kernel is that it provides a snapshot/rollback mechanism for individual application threads. When the system does not fail, it makes progress and takes checkpoint snapshots either periodically or according to applications' needs. When a failure is detected for a particular thread, r-kernel performs rollback operations by restoring the thread to the last known correct state. If the same problem persists, further rollback operations to even earlier checkpoints are needed until the thread eventually gets rebooted. To provide robustness against bugs, every time a thread gets re-executed, it runs in a *safe mode* with a novel technique called *system call shadowing*. Specifically, this technique is based on the observation that in our target software environment, application threads are interfaced with the kernel through a suite of customized system calls. We therefore change system call implementations by replacing the original ones with safer, but more complicated ones with higher overhead. We demonstrate that they can protect against certain deterministic bugs to avoid repeated system failures.

Finally, it is possible that a bug may corrupt the entire system state, leading to unpredictable deadlocks or system freeze before the rollback mechanism can be activated. To this end, we introduce the last defense line of r-kernel, by integrating the use of watchdog timers. This mechanism causes the entire node to be rebooted when necessary, independently of the execution sequence of the CPU. To avoid information loss, r-kernel allows selectively storing critical information, such as context switch history, into non-volatile memory (such as EEPROM) at additional overhead. After the node gets rebooted, such non-volatile history, which we term as *past-run traces*, are used to infer which threads have caused failures, so that the same thread will not crash the node for the second time.

To the best of our knowledge, this is the first piece of work that implements thread-level checkpoints and rollbacks in resource constrained embedded systems. It addresses not only non-deterministic bugs, but also deterministic bugs. It integrates multiple techniques such as checkpoints, rollbacks, system call shadowing, watchdog timers, and past-run traces, so that we can achieve considerably improved system robustness with low overhead and fast recovery. This combination is unique in its kind, and most design decisions have been carefully considered to fit on the resource-constrained sensor nodes. Therefore, this work is considerably different from the checkpoint/rollback implementation in traditional systems, such as servers. For example, the checkpoint/rollback mechanisms have to be re-designed based on the particular operating system requirements; the system call shadowing technique is unique; the past-run trace reconstruction has to fit within resource constraints. Furthermore, r-kernel can be applied to deployment software environment, since it does not require additional hardware, wires, or human involvement. Finally, while it does not require modifying source code, it provides APIs for programmers and interactive commands to administrators to better fine-tune the system performance. In summary, we believe r-kernel is very valuable for achieving a new level of robustness in systems where such robustness is most valued: autonomous, resource-constrained, and networked embedded systems.

The rest of this paper is organized as follows. Section II presents the knowledge background of the LiteOS operating system on which r-kernel is implemented. Section III presents an overview of r-kernel design. Section IV presents the implementation of r-kernel. Section V demonstrates the performance of r-kernel through extensive evaluations. Section VI reviews the state of the art. Section VII summarizes this paper.

II. BACKGROUND

r-Kernel builds on the LiteOS operating system [5], a thread-based, Unix-like operating system environment developed for resource-constrained sensor networks. This section provides the necessary background on the LiteOS operating system, and provides the context for r-kernel's design choices and implementation decisions.

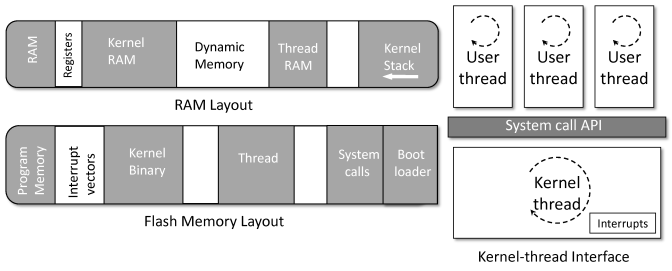


Fig. 1. The Architecture of LiteOS

LiteOS is an operating system specifically designed for ultra-low-power microcontrollers with limited RAM and storage space. LiteOS currently supports 8-bit AVR microcontroller based sensor nodes running at $1-8$ MHz that has $4-8$kB of SRAM, 128kB of flash memory, and up to 512kB of external flash storage space.

The operating system uses threads as basic units of abstraction. Threads may be created by the LiteOS kernel or other existing threads at runtime, and are written in the C programming language. The LiteOS kernel itself is a thread that is initiated during boot time, and is responsible for tasks ranging from interactions with hardware to management of user created threads. The LiteOS kernel is interfaced with user threads through a suite of system calls. System calls not only provide software compatibility by ensuring that any modification to the kernel will be transparent to the application side, but also simplifies application development by providing a unified API interface for the users. Compared to directly invoking kernel functions, each system call adds 5 instructions (10 CPU cycles), a sufficiently low overhead to be supported on microcontrollers. Figure 1 shows the conceptual memory layout of the kernel and the applications, and the system calls that bridge them. Further details on the LiteOS operating system can be found in the LiteOS paper [5].

Impact on the Design of r-Kernel Our design choices of r-kernel is significantly affected by the particular architecture and requirements of LiteOS. Specifically, the checkpoint-rollback support is enabled by the thread-oriented design of LiteOS. The snapshot images created for threads are stored in the file system of LiteOS. We note that, however, r-Kernel is not intended to be developed for LiteOS only. Some recent innovations in TinyOS, the currently most popular operating system for sensor networks, also started to adopt threads and system calls, making it possible for porting r-kernel to TinyOS.

Another impact of LiteOS is its organization of system calls. The original version of LiteOS implements a suite of 56 system calls aligned consecutively. We re-organize the memory layout of system calls so that *shadow* system calls are aligned to the end of their corresponding non-shadow system calls. This design allows new shadow system calls to be added easily, and leads to more compact code with smaller memory footprint.

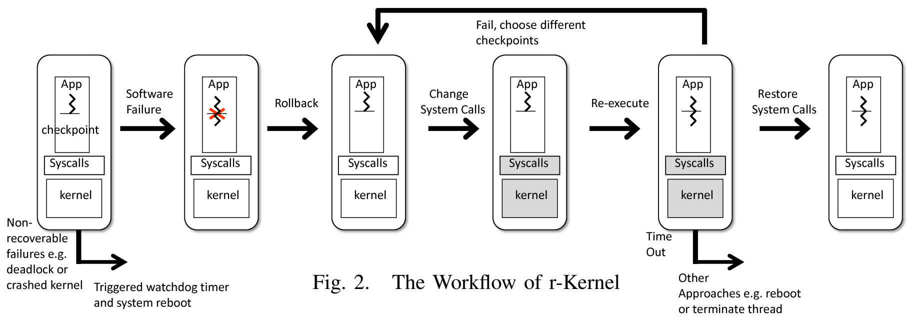

Fail, choose different checkpoints
App
checkpoint
Syscalls
kernel
Software Failure
Rollback
Change System Calls
Re-execute
Restore System Calls
Non-recoverable failures e.g. deadlock or crashed kernel
Triggered watchdog timer and system reboot
Time Out
Other Approaches e.g. reboot or terminate thread

Fig. 2. The Workflow of r-Kernel

The third impact of LiteOS is its user interaction. We extend the Unix-like interface of LiteOS to provide support for additional commands that enable snapshot and rollback operations through the shell. Such user interaction support not only allows user intervention when necessary, but also supports debugging of potential bugs.

III. System Design

The three main features of r-kernel are rollback of threads, system call shadowing, and past-run reconstruction with watchdog integration. A typical application that is built on r-kernel is assumed to consist of a collection of small, well isolated threads that collectively form the application fabric. Interprocess communication follows a messaging system, and no memory is shared between threads. Specifically, the design of r-kernel has the following goals.

A. Design Goals

- Effectiveness: this goal is measured by its ability to perform: (i) checkpoint and rollback operations accurately, (ii) avoidance of certain deterministic bugs successfully, and (iii) support for user interaction.
- Simplicity: as r-kernel provides APIs for user applications, one demanding goal is that such APIs are easy to use. Our experiences with other systems tell us that simple APIs are more likely to be adopted compared to complex ones.
- Efficiency: sensor network platforms suffer considerable resource constraints (in CPU, memory, and energy overhead). Therefore, using resources efficiently is a demanding goal. r-Kernel is designed with efficiency in mind so that it can be integrated into existing operating systems with acceptable overhead.

The above design goals motivate our design choices throughout the remaining of this section.

B. System Workflow

Figure 2 shows the workflow of r-kernel. First, the system reboots and starts the watchdog timer. It then loads application threads. There are two ways of loading a thread: internal or external. For internal loading, the thread has been compiled together with the kernel. For external loading, the thread is separately compiled, and will be loaded from the file system. Once threads are loaded, they are executed as normal. Periodically or in response to the requests of the application (through programming APIs detailed later), checkpoints will be created by the kernel. When this happens, all information needed for the rollback operation, such as thread contents, current packet queue, and file operators, are copied into a specially named file in the non-volatile flash.

As illustrated in Figure 2, rollback operations will be triggered when software failures are detected. Such detection is either application-specific, or OS-assisted. In the work flow, we assume a mechanism has been developed to determine whether failures have occurred. The evaluation section describes an OS-assisted approach to detect stack overflows as an example.

When a rollback occurs, the same program will be executed, but with a different system call environment. This mechanism is motivated by the needs to protect against certain deterministic bugs. We hope the differentiated execution environment will help the software survive the previous failure, and proceed as normal. If this is what happens, once the execution passes the previous spot of failure (as measured by timestamps), the system call environment will be restored to normal and the program execution will proceed. If such re-execution is not successful, then either an even earlier checkpoint will be tried, or the program thread will be rebooted or terminated.

C. Programming APIs and User Interface

Our design goals call for an easy-to-understand interface for users. We propose the following programming APIs. The APIs in the first category are related to checkpoints. This includes creating a new checkpoint `createCheckpoint`, checking whether a checkpoint exists `existCheckpoint`, deleting a checkpoint `deleteCheckpoint`, and rollback to a previous checkpoint `rollbackCheckpoint`. The APIs in the second category are related to the operating system environment, such as changing the mode of system calls `enableSafeMode` and `disableSafeMode`. The third category of APIs are auxiliary, such as getting the current timestamp `getCurrentTime`. Together these functions provide comprehensive control on checkpoint behavior.

Following is a simple example on the behavior of checkpoints.

```
1  checkpointIndex = createCheckPoint();
2  printf("Point A,");
3  rollbackCheckpoint(checkpointIndex);
4  printf("Point B,");
```

In this example, line 4 will never be executed as the program returns to the first checkpoint again and again. The printing result of this segment (using the serial port to receive

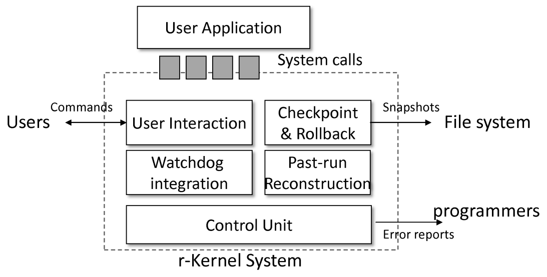


Fig. 3. The Architecture of r-Kernel

incoming data from MicaZ) as: `Point A, Point A, Point A` .... Note that "Point B" will not be printed.

Besides programming APIs, we also modified the LiteOS shell to support user side commands, to perform tasks such as creating snapshots and thread rollbacks. Combined with the debugging commands of LiteOS that allow breakpoint operations, these commands provide more flexibility to the end user to increase the level of controllability of threads for runtime environments.

D. Checkpoint and Rollback Engine

The checkpoint-and-rollback engine (CRE) is the most important component in r-kernel. Its role is illustrated in Figure 3. It takes checkpoints of the target application, and transparently rolls back the application to a previous checkpoint upon software failure or user request. At a checkpoint, it stores a snapshot of the application into external non-volatile flash as a file. We emphasize that this design choice is significantly different from related work, especially those targeting server applications [6], [7], which store checkpoint images in memory. Our design choice is motivated by the fact that the RAM is usually the most constrained resource on embedded devices. For example, on MicaZ nodes, only 4KB are provided, and only around 1KB is provided to user applications as free memory. On the other hand, snapshot images could reach 200 bytes. Hence, when multiple snapshot images need to be stored, non-volatile space has to be exploited.

Performing a rollback operation is straightforward. CRE simply reinstates the program from the snapshot associated with the specified checkpoint. As shown in the code sample in the previous section, each checkpoint is associated with a unique index, which is generated by reading the current timestamp on the node. This index is passed as a parameter for the rollback function to locate the right version of the snapshot. Finally, CRE is also responsible for other functions, such as replying to queries regarding whether a snapshot image still exists, or deleting a snapshot image from the file system.

There are two challenges for designing a practical and useful CRE. First, CRE needs to handle the interaction between the thread and the operating system in a proper way. For example, if a thread requests dynamic memory allocation, suppose that it gets a memory chunk S, makes changes to the content, and initiates a checkpoint C, then, the resulting snapshot image should include the contents of the allocated memory chunk S. The reason is that the thread might later release S, and then tries to rollback to the previous checkpoint C. Since S was allocated at checkpoint C, it needs to be properly restored during rollback to ensure an identical execution environment at checkpoint C. Generally, the challenge is that CRE should not only restore the RAM contents of a thread at a checkpoint, but also its allocated resources from the operating system.

Second, CRE needs to provide maintenance of multiple checkpoints, as this could impose a significant space overhead. For example, if periodic checkpoints are taken, old snapshot images may gradually become useless as the thread no longer needs them. Therefore, CRE should remove such out-of-date snapshots to save space. We next describe how we address these two problems in the design of CRE.

Handling Allocated OS Resources

To handle the interaction between threads and the operating system properly, the first concern is that CRE needs to keep snapshots of their allocated operating system resource. We consider three types of resources, including file control data structures for opened files, dynamic memory allocation contents, and obtained resource locks. Common to these system resources is that they may be released later during the execution of the program. When a rollback occurs, CRE tries to obtain such resources again to ensure that the same execution environment is provided to the threads. However, note that it is well possible that such system resources are temporarily unavailable during rollback. For example, a lock might be temporarily held by another thread, or a chunk of dynamic memory with the same size may not be available. In such cases, the thread will be temporarily blocked until the resource requirements are met for the rollback operation.

The second concern is related to posted tasks (from the thread) and received packets (to the thread) that have yet to be processed by the OS kernel. The reason is that these tasks or packets may affect the execution of the thread later. If they are not properly restored (i.e., they are processed by the kernel after a checkpoint, and then CRE wants to rollback to this checkpoint), the execution of the thread after rollback might be different. Therefore, both posted tasks and received messages need to be stored in the snapshot images. In practice, CRE keeps both the functional pointers for posted tasks and copies of received packets at a checkpoint. Note that, however, after rollback, the timing of the delivered packets will be different from the original, as CRE does not record when an asynchronous event gets delivered to an application. The underlying design philosophy is that r-kernel purposely introduces *nondeterminism* into the execution of the programs to avoid any potential problems during earlier rounds of execution. This makes our work considerably different from previous work that focuses on replaying the sequence of events in the exact order and timing requirements [8], [9], and better than them in avoiding the appearance of the same bugs.

The third concern relates to the changes one thread makes to the underlying OS environment. For example, during the execution of a thread, it may set the radio channel according to its own needs. To address such changes, at each checkpoint, a set of system environment parameters, including the radio frequency, power level, and percentage of CPU usage, are stored. During rollback, such parameters are restored to their earlier values so that the execution environment is identical as the original checkpoint.

The implementation of CRE does have one limitation: it does not provide rollback support on file operations. That is, if a file has been updated after a checkpoint, it will not be recovered to its original state. This design choice is motivated by the observation that in sensor network applications, most applications only use the file system as the place for storing sensing data. Even if applications fail, the sensing data are still valuable and should not be deleted. Based on this premise, we find that this design choice will fit the needs of most applications, are reasonable to make, and help conserve the most amount of useful data for application purposes.

Checkpoint Maintenance

A possible problem with maintaining multiple checkpoints is that this imposes a significant storage space overhead. One simple way to save space is that whenever a rollback to a checkpoint happens, all following checkpoints will be removed. However, on resource constrained sensor nodes, such an approach is not sufficient. To maintain checkpoints effectively, we develop an adaptive approach that maintains checkpoints based on i) the current system load; and ii) the predicted usefulness of the snapshot images. Formally, suppose r-kernel takes checkpoints periodically, let τ_i be the timestamps of the checkpoints that have been kept in the chronological order. We can use two schemes to keep those checkpoints: one is to keep intervals between checkpoints to be constant, yet only storing the most recent checkpoints; the second is to keep exponential landmark checkpoints, deleting redundant checkpoints on the run. That is, the two schemes satisfy the following equations, respectively.

$$\tau_i - \tau_{i-1} = \tau_{i+1} - \tau_i$$
$$\tau_i - \tau_{i-1} = \beta(\tau_{i+1} - \tau_i)$$

In the first scheme, only the most recent N snapshots are kept. In the second scheme, β is an exponential parameter for setting intervals for checkpoints. In both cases, r-kernel keeps track of the current CPU utilization. If the system is busier than a certain threshold, the snapshot activity is temporarily suspended to reduce its impact on system performance.

E. System Call Shadowing

While several types of failures may be avoided by re-executing the same code with different timing, such as race conditions and deadlocks, such an approach cannot address deterministic bugs, as illustrated in the following example.

```
char *buffer = malloc(100);
/*some more code here*/
buffer[100] = 1;
```

In this example, if the memory address `buffer[100]` has been allocated to another thread, it is likely that a hidden fault has been introduced into the system. If this fault manifests itself with a failure, no matter how many rollbacks are performed, the system is going to experience failures again and again. The example motivates us to design system call shadowing to avoid such deterministic bugs.

Our design stems from the observation that the library functions in LiteOS, such as `malloc`, are bridged to the kernel implementation through a suite of system calls. We employ a novel and affordable approach to redirect system calls. To this end, we modified and re-organized the LiteOS system calls in such a way that for some calls, we develop alternative implementations, and patch the kernel with their functionalities. For example, the new implementation of `malloc` no longer assigns memory chunks consecutively. Instead, it adds paddings between two assigned memory chunks. This way, the bug as shown in the previous example may be avoided if a proper padding size is chosen. We call such an approach as *system call shadowing*, and with them, we call that the applications are running in the *safe mode*.

There are two challenges on the design of shadow system calls. First, which system calls should have shadows? Second, how to modify applications to invoke the correct system calls?

We focus on two categories of system calls. The first category is related to memory operations, and the second is related to process scheduling. Specifically, the shadow system calls support the following environmental changes:

(1) Padding buffers. This approach modifies system calls such as `malloc()`, `realloc()`, `calloc()`, and `free()`. It adds fixed-size paddings to both ends of any allocated buffers during re-execution to avoid buffer overflows. Because this method reduces the amount of allocatable dynamic memory, it should only be enabled in the safe mode.

(2) Zero-filling buffers. This approach avoids some failures caused by uninitialized reads. It fills any allocated buffer space with zero values. Since it incurs extra time overhead, it should be enabled only in the safe mode.

(3) Thread scheduling. This approach modifies the priority of a thread during its execution to avoid failures caused by concurrency issues such as race conditions. Specifically, whenever a new thread is created, we artificially increase the scheduling time share of the thread so that it is less likely for this thread to experience context switches during some unprotected critical region.

The second problem is how to modify applications to invoke correct system calls. To this end, we rely on a technique called *dynamic instrumentation*, which modifies the binary images of applications to point to the right system calls (we previously adopted this approach to insert dynamic tracepoints [10]). Specifically, each invocation to a system call is started with a `call` instruction followed by the address of the system call portal. The dynamic instrumentation process walks through the binary images of the applications, and modifies the addresses of the system calls to their shadows.

F. Integration with Watchdog Timers and Past-run Information

The last defense line against failures comes from the integration of watchdog timers. These timers reset the system if the kernel enters a deadlock or other crashed mode when the kernel can no longer periodically reset the watchdog timer. When this happens, the entire kernel will reboot. When this occurs, one research challenge is how to avoid the same bug from being triggered again and again (which causes the system to enter an endless reboot loop). To this end, we develop a technique called *past-run reconstruction* to infer which thread has caused the reboot, so that the same thread would not be executed again.

Specifically, we rely on *traces* in our design of *past-run reconstruction*. To avoid traces from being lost, we store them in non-volatile storage, e.g., EEPROM (if EEPROM is not available, on-board flash storage can also be used with additional overhead). Specifically, the following two types of data are stored as traces during the normal execution: (i) the history of context switches, stored with the associated index sequence of threads being executed, and (ii) the checkpoint and rollback operations, stored with timestamps. Both types of information are stored in a circular buffer, so that the total storage space is limited. Only the most recent history is stored, while old history is automatically removed. Upon reset, r-kernel searches for such information in the circular buffer. Combined with the snapshot images that are previously stored in the file system (also non-volatile), r-kernel is able to determine the last user level thread that was running prior to the system reboot. This thread will be temporarily disabled, and reported as errors for debugging purposes. Our evaluation results show that this approach is able to pinpoint the failure thread correctly in all our benchmark cases (the evaluation section contains the details of benchmark applications).

IV. Implementation

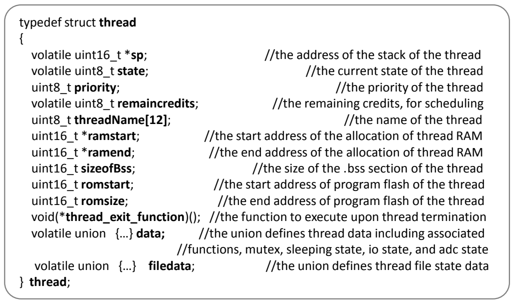
In this section, we describe the implementation choices made to r-kernel. It has been implemented on the MicaZ platform, and it is expected to work on Mica2 and IRIS as well, with small modifications. We first describe the thread rollback component, then the system call environment, then the watchdog timer integration and past-run trace reconstruction.

A. Implementation the Thread Checkpoints and Rollbacks

While the r-kernel makes several improvements over the LiteOS kernel, it also maintains the original functionality of LiteOS. All applications running on LiteOS are allowed to work. r-Kernel does not require additional hardware nor affect program execution, unless when it is turned on. Therefore, it can be left on the sensor node to be used only when needed.

The central component of r-kernel is its checkpoint and rollback system. Figure 4 shows the data structure of threads in LiteOS. In the implementation of r-kernel, the rollback functionality is implemented as a stand-alone thread that can be either internally or externally loaded. We prefer the externally loaded version since it does not consume any RAM when not activated, unlike the internally loaded version. However, the internally loaded version is faster to enable or disable. We provide both versions for users to choose from.

Once activated, the checkpoint-and-rollback thread initiates activities upon receiving inter-thread messages from user threads. As mentioned in Section III-C, the user thread can invoke several APIs to perform various activities. These APIs are implemented in the standard library of LiteOS programming environment by sending messages to r-kernel components. Specifically, in our improved LiteOS, we develop a message passing mechanism where we treat external and internal messages uniformly: all messages are addressed to a destination node with a port number. If the destination is another node, a corresponding routing/MAC layer algorithm is invoked. If the destination is 0, it means that this message is sent to the current node itself. On the thread side, each thread subscribes to one or more port numbers. Whenever an incoming message to this port number arrives, a linked function is invoked that typically wakes up the waiting thread. The thread then takes the incoming packet from the queue and processes it. When this packet comes from another thread on the same node, e.g., messages sent from the user thread to the r-kernel component thread, such a packet serves the functionality of inter-thread communication.



```
typedef struct thread
{
   volatile uint16_t *sp;                        //the address of the stack of the thread
   volatile uint8_t state;                            //the current state of the thread
   uint8_t priority;                                      //the priority of the thread
   volatile uint8_t remaincredits;               //the remaining credits, for scheduling
   uint8_t threadName[12];                                 //the name of the thread
   uint16_t *ramstart;             //the start address of the allocation of thread RAM
   uint16_t *ramend;                //the end address of the allocation of thread RAM
   uint16_t sizeofBss;                       //the size of the .bss section of the thread
   uint16_t romstart;                //the start address of program flash of the thread
   uint16_t romsize;                  //the end address of program flash of the thread
   void(*thread_exit_function)();  //the function to execute upon thread termination
   volatile union   {...} data;         //the union defines thread data including associated
                                       //functions, mutex, sleeping state, io state, and adc state
    volatile union   {...}   filedata;                //the union defines thread file state data
}  thread;
```

Fig. 4. The Data Structure of the Threads

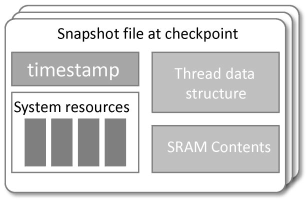
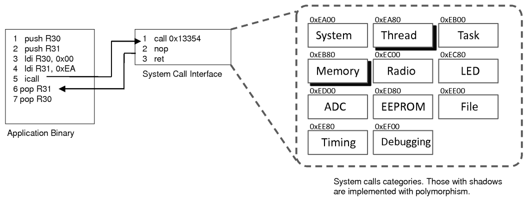


Fig. 5. Checkpoint file contents

B. Implementation of System Call Environments

To understand the implementation of system calls, Figure 6 shows the process of a typical system call initiated by the application code. Note that we organize system calls into 11 categories based on functionality. To switch between normal mode and safe mode, the entry addresses of the system calls are modified to point to *shadow* implementations using dynamic instrumentation. We add shadows to two categories of system calls: thread system calls and memory system calls. The reasons that these two categories are selected is because based on our experiences, we find it tricky to deal with bugs related to thread and memory management. Thus, we first address these bugs in our preliminary study.

1) Thread system calls: In this category, we modify a critical function `createThreadSystemCall`. More concretely, in the safe mode, when a new thread is created, there are three differences made in the safe mode: first, this thread is created with



```
push R30
push R31
ldi R30, 0x00
ldi R31, 0xEA
icall
pop R31
pop R30
```

```
call 0x13354
nop
ret
```

Fig. 6. Shadow system call implementation

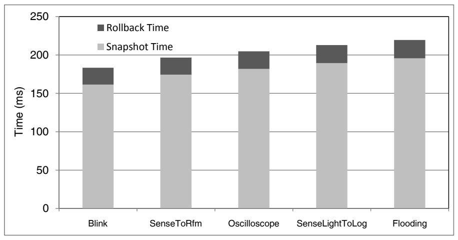


Fig. 7. Delays for Snapshot and Rollback Actions

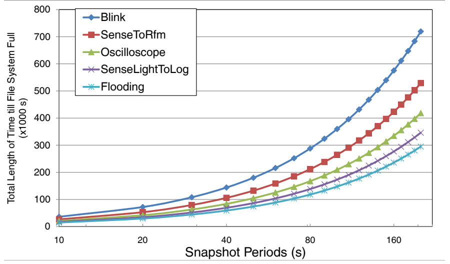


Fig. 8. Limitation of File System Capacity

an isolated chunk of RAM instead of a consecutive memory chunks to minimize its effects on other threads; second, this thread is created with a slightly larger chunk of RAM compared to requested to avoid stack overflows; third, this thread is created with the highest priority, regardless of the actual priority provided by the user, to reduce possible concurrency bugs due to context switches.

2) Memory system calls: In this category, we modify all system calls, including `malloc`, `free`, and `realloc`. More concretely, in the safe mode, there are three differences: first, to avoid dangling pointers, we do not immediately re-allocate the same chunk of memory that has been released. Instead, we follow a circular approach where the memory chunks are allocated in a circular order, so that the most recently released memory will be the last re-allocated; second, we set memory chunks with all zeros to avoid bugs caused by uninitialized reads; third, we insert paddings to the beginnings and endings of allocated buffers to reduce the possibility of buffer overflows.

C. Implementation of Watchdog Timers and Past-run Trace Reconstruction

In our implementation of watchdog timers, we modified the LiteOS kernel so that it periodically resets the watchdog timer every thirty seconds in the main loop. This value is chosen based on the observation that it is rarely the case that one application will spend more than thirty seconds in a segment of code protected by `atomic` operators. Consequently, if the watchdog timer fires, we are certain that the node is no longer responsive, and should be reset.

To help reconstruct past-run traces after a reset, we meet the challenge that after a reset, all RAM contents will be erased. Therefore, we keep a circular queue in the EEPROM to store the *major* actions taken by the kernel. Currently the following actions are stored: loading threads, context switches, checkpoint creations, checkpoint deletions, and thread terminations. Whenever these actions happen, an action index together with simple parameters are stored into the EEPROM circular buffer. Currently, we allocate 1024 bytes of the 4096 bytes of EEPROM on MicaZ for this purpose (starting from address 2701 to avoid conflicts with the file system usage of EEPROM). This way, when the node wakes up again, the contents of EEPROM together with the file system status will be used to recover the past-run traces.

V. Evaluation

In this section, we evaluate the performance of r-kernel. Our evaluation consists of three parts. First, we measure the performance of the checkpoint/rollback mechanism. Second, we measure the slowdown of applications caused by logging context switches. Finally, we present the performance of user side commands such as *snapshot* in terms of responsiveness.

We first measure the performance of checkpoint/rollback mechanisms. Figure 7 shows the performance of the snapshot and rollback actions measured in terms of delay. In this experiment, we select five benchmark applications, as illustrated on the X axis, and measure the average time for snapshot delays and rollback delays. Since these applications require increasing RAM usage, we set the memory consumption of each application, in terms of bytes, to be 100, 150, 200, 250, and 300 bytes, correspondingly. This memory allocation includes not only the static RAM usage, but also the dynamic stack of each application. We measure each application for 20 times, and plot the average delays. The time measurement process relies on a cycle-accurate timer on MicaZ nodes to collect the number of cycles between two timestamps, which is then converted into elapsed time. As illustrated in this figure, the increasing delays are primarily caused by the time on file system operations: larger programs lead to an increase in snapshot image sizes, which takes longer to create and read.

Besides the delays for typical operations, we also measure a reference timeline for snapshot (the timeline for rollback is similar), using the `blink` application as an example. The purpose of this measurement is motivated by the message-passing nature of our implementation: we want to demonstrate that such an architecture does not cause too much additional delays. Practically, when a snapshot is to be made, the user thread (`blink`) sends a message to the snapshot thread, which is then waken up, parses the message content, looks up the information of the thread that sent this message, and performs the snapshot/rollback operations accordingly. Again, this timeline is measured based on the cycle-accurate timer on MicaZ nodes. The results are illustrated in Figure 9. Observe that in this figure, the primary time delays are caused by creating a snapshot image in the file system.

Our next experiment shows the impact of file system capacity in storing snapshot images. Specifically, we increase the periods

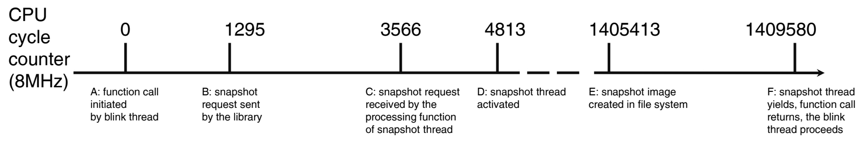
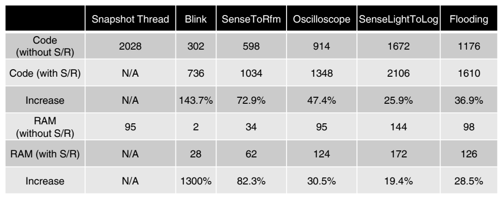

CPU
cycle
counter
(8MHz)
0
1295
3566
4813
1405413
1409580
A: function call initiated by blink thread
B: snapshot request sent by the library
C: snapshot request received by the processing function of snapshot thread
D: snapshot thread activated
E: snapshot image created in file system
F: snapshot thread yields, function call returns, the blink thread proceeds

Fig. 9. Timeline for Functional Calls



	Snapshot Thread	Blink	SenseToRfm	Oscilloscope	SenseLightToLog	Flooding
Code (without S/R)	2028	302	598	914	1672	1176
Code (with S/R)	N/A	736	1034	1348	2106	1610
Increase	N/A	143.7%	72.9%	47.4%	25.9%	36.9%
RAM (without S/R)	95	2	34	95	144	98
RAM (with S/R)	N/A	28	62	124	172	126
Increase	N/A	1300%	82.3%	30.5%	19.4%	28.5%

Fig. 10. Comparison of Memory Footprint

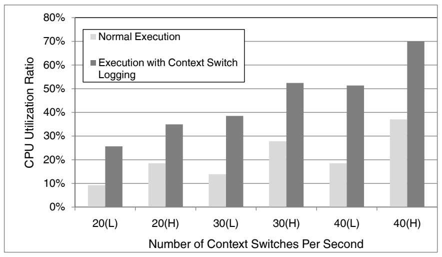


Fig. 11. Slowdown of Applications with Context Switch Logging

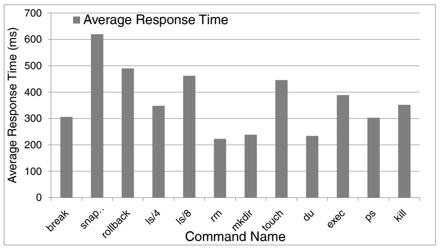


Fig. 12. Comparison of New Commands with Old LiteOS Commands

of making a snapshot from 10s to 200s, and measure how long the file system becomes full. The results are shown in Figure 8. Observe that as snapshot are created with longer and longer intervals, it takes considerably longer time for the file system (with a capacity of 512K bytes) to be full. Given the long time to fill the file system, the evaluation results demonstrate the practical value of the r-kernel system.

Another issue we are interested in is the increase of memory footprint of applications with snapshot functionality included. Figure 10 shows the evaluation results. In this experiment, we continue to use the five benchmark applications, where we compile them with and without having one snapshot/rollback (denoted as S/R) pair in the main program. Both the code size and the RAM usage (only static RAM is considered as dynamic stack usage will change over the execution of a program). The code size and RAM usage of the snapshot thread is also shown in this table. Observe that the increase for memory footprint is much smaller for larger applications (e.g., `flooding`) than for smaller applications (e.g. `blink`). The reason is intuitive: the additional overhead brought by the inclusion of snapshot functionality is almost constant. Therefore, its impact is much smaller compared when the original program has a larger footprint.

In our next experiment, we evaluate the slowdown of applications with context switch logging, a crucial component of the past-run reconstruction. Figure 11 shows the performance results. In this experiment, we use a modified version of the `Blink` application, where we insert a large amount of extra computation into the main loop for experimental purpose. We test with both low computational load (denoted as L) and high computational load (denoted as H). with different number of context switches per second. As observed in this figure, logging context switch sequences into EEPROM adds considerable overhead compared to normal execution of programs. The overhead, we argue, is tolerable: even for 40 context switches to be recorded per second with a high computational load, the CPU usage increases from around 37% to 70%, still acceptable for the application.

Finally, we evaluate the performance of snapshot/rollback commands, which are implemented in the LiteOS shell, compared to other old commands provided by LiteOS. Figure 12 shows the comparison results. As illustrated, the typical delays in invoking snapshot/rollback commands are comparable with those such as file system operations (`ls`), breakpoint creation (`break`), or process operations (`ps`). Therefore, these commands provide convenient ways to users to interact with r-kernel when needed.

VI. Related Work

How to guarantee software robustness has been extensively studied in the literature for traditional platforms such as commodity PCs, server systems, and high-performance computing. Several studies indicate that various bugs can be effectively resolved by rebooting [11], [12]. Not surprisingly, rebooting is costly in both time and energy. More recently, two techniques have been presented to improve the performance compared to whole-system rebooting. The first technique is microrebooting [13], [14], which is a fine-grained technique for surgically recovering faulty application components without disturbing the rest of the application. Microrebooting is evaluated in an Internet auction system running on an application server. It is demonstrated that microreboots recover most of the same failures as full reboots, but do so an order of magnitude faster and result in an order of magnitude less overhead. Some work, such as shadow driver [15], further tries to conceal the recovery process from users to create more robust device drivers. The second technique is based on a checkpoint-rollback mechanism [7], [16]–[18]. These mechanisms share in common that they create checkpoints for running programs, and rollback to these programs later when bugs appear. Some of these techniques also try to address deterministic bugs. For example, progress retry [17] recorders messages to increase the degree of non-determinism. Recovery blocks [18] rely on different implementation versions during rollback to avoid deterministic failures. Rx [7] relies on OS-supported mechanisms to create non-determinism artificially to avoid deterministic bugs. These previous methods, however, are developed for conventional computing platforms where computational and storage resources are much more redundant. They do not consider the unique requirements on resource concerns, operating system designs, application properties that are unique to resource-constrained networked embedded systems. Therefore, such mechanisms have to be fundamentally redesigned for networked embedded systems.

In resource constrained embedded systems such as sensor networks, the most commonly followed approach to improve system robustness has been debugging. Specifically, a wide range of tools have been developed. These approaches include source level debuggers such as AVR JTAG [19] and Clairvoyant [20], simulators such as EmStar [21], Avrora [22], and TOSSIM [23], dynamic instrumentation based methods such as declarative tracepoints [10], and record and replay methods such as EnviroLog [24]. Among these approaches, EnviroLog is remotely similar to our work since it also exploits the use of checkpoints. However, EnviroLog does not support rollback operations. Instead, it focuses on replaying environmental input to re-trigger the same bug in consecutive runs.

Despite these debugging methods, due to the tight integration of software modules and close interacting with the physical world, guaranteeing the correctness of software on embedded platforms remains an extremely challenging task. Consequently, runtime robustness has recently drawn more attention. To our best knowledge, the only work that aims to provide runtime robustness for networked embedded systems through reboots is Neutron [25], which is a version of the TinyOS operating system that efficiently recovers from memory safety bugs. Specifically, Neutron reboots the so-called recovery units, which are loosely mapped to threads, and then recovers precious state across such reboots. In this sense, Neutron is most similar to Microreboot, while in contrast, our approach pursues exploiting checkpoint and rollback. Another key difference between our work and Neutron is that after rollback, we exploit system calls to provide a *shadow* execution environment to threads. Therefore, our approach allows recovering from certain deterministic bugs, while previous methods like Neutron could not.

VII. Conclusion

In this paper, we have described our design, implementation, and evaluation of r-kernel, which runs on the MicaZ platform and combines multiple techniques including snapshot/rollback, system call shadowing, and past-run trace reconstruction. Our results are positive: through our experiments, we conclude that r-kernel can be implemented with acceptable overhead on resource constrained embedded platforms such as MicaZ nodes. In the future, we plan to release the source code of r-kernel as part of the LiteOS operating system in its future releases to facilitate adoption by users.

References

[1] T. He *et al.*, "VigilNet: An Integrated Sensor Network System for Energy-Efficient Surveillance," *ACM Transaction on Sensor Networks*, 2007.

[2] L. Luo *et al.*, "Enviromic: Towards cooperative storage and retrieval in audio sensor networks," in *ICDCS*, 2007.

[3] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli, "The hitchhiker's guide to successful wireless sensor network deployments," in *SenSys*, 2008.

[4] "What really happened on Mars?" http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html.

[5] Q. Cao, T. F. Abdelzaher, J. A. Stankovic, and T. He, "The LiteOS operating system: Towards unix-like abstractions for wireless sensor networks," in *IPSN*, 2008, pp. 233–244.

[6] J. Plank, K. Li, and M. Puening, "Diskless checkpointing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 9, no. 10, 1998.

[7] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies—a safe method to survive software failures," in *SOSP*, 2005.

[8] M. Ronsse and K. D. Bosschere, "Recplay: a fully integrated practical record/replay system," *ACM Transactions on Computer Systems*, vol. 17, no. 2, pp. 133–152, 1999.

[9] G. Altekar and I. Stoica, "Odr: output-deterministic replay for multicore debugging," in *SOSP*, 2009.

[10] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo, "Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks," in *SenSys*, 2008.

[11] J. Gray, "Why do computers stop and what can be done about it?" in *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*, 1986.

[12] T. Chou, "Beyond fault tolerance," *IEEE Computer*, vol. 30, no. 4, pp. 47–49, 1997.

[13] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Garg, and R. Gowda, "Reducing recovery time in a small recursively restartable system," in *DSN*, 2002.

[14] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot — a technique for cheap recovery," in *OSDI*, 2004.

[15] M. Swift, M. Annamalai, B. Bershad, and H. Levy, "Recovering device drivers," in *OSDI*, 2004.

[16] E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, 2002.

[17] Y. Wang, Y. Huang, W. Fuchs, and C. Kintala, "Progressive retry for software failure recovery in message-passing applications," *IEEE Transactions on Computers*, vol. 46, no. 10, pp. 1137–1141, 1997.

[18] B. Randell, "System structure for software fault tolerance," in *Proceedings of the International Conference on Reliable Software*, 1975.

[19] "Atmel Corporation. Mature AVR JTAG ICE," http://www.atmel.com/dyn/products/tools-card.asp?tool-id=2737.

[20] J. Yang, M. Soffa, L. Selavo, and K. Whitehouse, "Clairvoyant: A comprehensive source-level debugger for wireless sensor networks," in *SenSys*, 2007.

[21] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin, "Emstar: A software environment for developing and deploying wireless sensor networks," in *Proceedings of the USENIX Annual Technical Conference*, 2004, pp. 283–296.

[22] B. Titzer, D. Lee, and J. Palsberg, "Avrora: scalable sensor network simulation with precise timing," in *IPSN*, 2005, pp. 477–482.

[23] P. Levis *et al.*, "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications," in *SenSys*, 2003.

[24] L. Luo, T. He, G. Zhou, L. Gu, T. Abdelzaher, and J. Stankovic, "Achieving repeatability of asynchronous events in wireless sensor networks with envirolog," in *INFOCOM*, 2006.

[25] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr, "Surviving sensor network software faults," in *SOSP*, 2009.