On the Integrity of Lightweight Checkpoints

Raul Barbosa and Johan Karlsson
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
{raul.barbosa, johan}@chalmers.se

Abstract

This paper proposes a lightweight checkpointing scheme for real-time embedded systems. The goal is to separate concerns by allowing applications to take checkpoints independently while providing them with an operating system service to assure the integrity of checkpoints. The scheme takes error detection latency into account and assumes a broad class of application failure modes. In this paper we detail the design of the operating system service, which offers a very simple programming model to application designers and introduces only a small execution overhead for each checkpoint. Moreover, we describe the usage of model checking to ascertain the correctness of our approach.

Keywords: *fault tolerance, embedded systems, checkpointing, model checking.*

1 Introduction

Finding the right balance between dependability and the cost of redundancy is a main challenge for designers of fault-tolerant embedded distributed systems. Replication of processor nodes at the system-level is a common approach to achieve fault tolerance in such systems. Using node replication, fault tolerance can be achieved through well known techniques such as triple modular redundancy (TMR), stand-by redundancy and active redundancy. Embedded systems are, however, often required to fulfill strict design constraints concerning hardware cost, power consumption, weight and physical space. Therefore, for many embedded systems minimizing the number of replicated nodes is a primary design goal. This implies that the nodes must be made highly dependable.

One way to improve the dependability of a node is to introduce node-level techniques that can detect and recover from transient hardware faults. Such faults are expected to become increasingly important in future computer systems due to the reduction of device sizes and the associated sensitivity to environmental factors, variability and degradation of integrated circuits [3, 4].

In this paper, we investigate the use of checkpointing and rollback as a way to tolerate transient hardware faults at the node-level in embedded real-time systems. Checkpointing involves taking regular snapshots of the system state and storing them in a safe place (sometimes called stable storage). When an error is detected, the system rolls back to a fault-free state by restoring the most recent valid checkpoint. While checkpointing and rollback is widely used in applications such as database systems and transaction processing, it is less frequently used for embedded real-time systems. The main reason for this is the time overhead generated when a rollback is made and when taking the checkpoints.

In real-time systems, the correctness of a computation depends not only on the resulting value but also on the timeliness of the result. Thus, unlike general-purpose applications, the goal of checkpointing in real-time systems is to guarantee that critical deadlines are met even when errors occur. If checkpoints are sufficiently frequent, the amount of re-computation required after an error may be small enough to complete the execution before the deadline expires. However, taking checkpoints increases the execution time in fault-free cases. For this reason, in real-time systems the checkpoint interval can't be arbitrarily small and the checkpointing mechanism must have a low overhead.

One way to reduce the overhead is to store only fundamental data at each checkpoint. The ability to identify these data depends on the checkpoint level, *i.e.*, whether it is done by the kernel, by a user-mode library or by the application itself [13]. In general, only the application designer can determine which structures are fundamental. Hence, checkpointing can be efficiently implemented at the application-level, with the additional advantage of having no overhead in calls to external code. This approach is therefore attractive for real-time systems.

However, implementing checkpoint and recovery functions increases the development cost of applications. Furthermore, applying this method correctly is often non-trivial, even in uniprocessors. One reason for this is error detection latency, *i.e.*, the amount of time between the occurrence of an error and its detection. During that time interval the application may save a corrupted checkpoint and then recovery is only successful by restoring an older non-corrupted checkpoint. Due to this problem we must introduce a delay between storing a checkpoint and considering it to be reliable. Another reason is the difficulty in establishing reasonable assumptions on the failure modes of applications. A transient hardware fault may, for example, corrupt a pointer and cause an application to overwrite any previous checkpoints. Consequently, the integrity of application-level checkpoints must be assured through careful design.

We propose a novel and lightweight checkpointing technique for real-time embedded systems. It allows applications to checkpoint their state independently but provides the means for them to lock checkpoints using memory protection. The method ensures the integrity of checkpoints for a broad class of application failure modes and takes error detection latency into account. The goal is to provide the level of reliability required by high-integrity applications and meet the needs of real-time systems.

Many applications can be made fault-tolerant by checkpointing small amounts of state information (*e.g.*, control algorithms [27, 28]). Moreover, device drivers may recover transparently from failures by retrieving state information lost during a crash [10], thereby requiring a reliable mechanism for guaranteeing the integrity of driver checkpoints. To achieve this, we advocate a clear separation of concerns: each application (or driver) should be responsible for taking checkpoints, while a platform service assures their integrity. This paper describes the design of one such platform service and shows how it can be implemented on modern microprocessors.

A challenge in the development of fault tolerance mechanisms is to ensure that they are free from design faults. Over the past years, model checking has become an attractive way to increase the confidence in that a design is fault-free. We used SPIN [11] – a well established model checker for software systems – to formally verify the correctness of our approach. This paper describes in detail the formal model of the system, which was exhaustively verified using SPIN.

The remainder of the paper is organized as follows. We begin by surveying related research in Section 2. Section 3 describes the lightweight checkpointing scheme and Section 4 details the usage of model checking to verify its correctness. Section 5 introduces two extensions to the baseline method proposed in the paper. Lastly, we summarize the main conclusions of the paper in Section 6.

2 Related Research

This paper addresses the problem of checkpointing for real-time uniprocessor systems. A central problem when using checkpointing and rollback recovery in such systems is to ensure task schedulability. Given a failure hypothesis and a set of real-time tasks, one must determine if all tasks will meet their deadlines (both when errors occur and in fault-free cases). The work presented in [24] and [30], among others, studies the effect of checkpointing on the schedulability of fault-tolerant task sets.

An issue closely related to scheduling is the optimal checkpoint interval. If checkpoints are too frequent, their combined overhead is too high; if they are too sparse in time, recoveries may require too much re-computation. In real-time systems, the optimal checkpoint interval should maximize the probability of meeting deadlines when errors occur, while ensuring that deadlines can always be met in the error-free case [14]. Early work on this problem included approximations of the optimal inter-checkpoint interval, for general-purpose applications, assuming single failures with a constant rate [29]. More recent mathematical models [18] consider time-varying failure rates to determine the optimal strategy. Most mathematical models, both for general-purpose and real-time computing, assume not only the integrity of checkpoints but also that errors are detected instantly. Hence, our work can be applied to handle those assumptions.

Several existing implementations of platform services provide checkpointing for uniprocessors. These may be offered in the form of libraries supporting both transparent and non-transparent checkpointing, *e.g.*, by allowing programmers to specify which memory addresses should be excluded from the snapshots [22, 12]. Kernel- and user-level checkpointing techniques have the advantage of making it simple to protect checkpoints from faulty applications. However, these provide no automatic means for dealing with error detection latency and impose the additional overhead of calling external code, which is undesirable for real-time embedded systems.

Error recovery in communication systems has motivated extensive research on distributed checkpointing [7, 2]. Ensuring system-wide consistency when determining the recovery line may lead to the domino effect, where a sequence of rollbacks brings all nodes to the beginning of the computation. This problem can be solved by creating globally coordinated checkpoints [5]. Some of the proposed schemes deal with error detection latency [26, 15]. These assume either that at most one checkpoint can be affected by an error or that an arbitrary number of past checkpoints can be stored. Coordinated checkpointing complements our scheme, since we consider uniprocessor systems and focus on protecting the integrity of checkpoints carried out inde-

pendently by applications.

Most approaches implement checkpointing and rollback functions in software. However, the increasing concerns with transient hardware faults have led to checkpointing at the hardware-level, as a means to mitigate the effects of soft errors in mainframe computers [25, 20]. This approach is impractical for embedded systems, where cost of acquisition and power consumption are limiting factors. Nonetheless, transient hardware faults are a concern and this motivates the implementation of checkpointing in software.

3 Lightweight Checkpoints

In this paper we introduce a mechanism for assuring the integrity of application-level checkpoints. Our lightweight approach separates concerns by allowing applications to take checkpoints independently – an attractive method for real-time embedded systems – while an operating system service assures the integrity of checkpoints – a fundamental issue for building high-integrity applications.

One main concern in designing this operating system service is to ensure its real-time performance. The overhead should be as low as possible and each call to the service should be bounded in time, so that it can be used by real-time tasks. By reducing the overhead of checkpointing we can improve fault tolerance, since we are allowed to take checkpoints more frequently. Our approach requires only a very small system call to be made after each checkpoint in order to lock that checkpoint using memory protection. We begin by listing our assumptions and the describe the details of our scheme.

3.1 Assumptions

We consider a real-time operating system executing a set of tasks. Interference among tasks is assumed to be controlled both in the temporal domain, by applying a well known scheduling policy [19], and in the spatial domain, by using some form of memory protection. Tasks interact with the operating system through the *system call* interface which must be robust, *e.g.*, by using sandboxing techniques [21, 23].

Our goal is to design an operating system extension that provides a service for protecting application-level checkpoints. This service is provided through the system call interface. Under the assumption that applications are unable to interfere with one another, conflicts could only arise from calls to our operating system extension. Here, we can identify all calls through the calling process ID. Thus, we can rule out all interference among tasks and focus on a single task to describe our operating system extension.

We assume that main memory is sufficiently reliable to be used as stable storage for an embedded system. Given that our goal is to tolerate transient hardware faults and that main memory is usually protected with error-correcting codes, we can assume that checkpoints are safe when stored in main memory. In general, the DRAM cells used to build memory chips can be considered very reliable [3]. Moreover, even if a fault affects a checkpoint directly in memory, we assume that the probability of another near-coincident fault causing an application to rollback (to the corrupted checkpoint) is negligible.

3.1.1 Failure Modes and Error Detection Latency

Our checkpointing scheme addresses only transient faults directly affecting the applications. Errors affecting the operating system or the checkpointing service may be detected but we provide no means to recover from them. Typically, a real-time kernel executes less than 5% of the time [16]. During the remaining time the processor is either idle or running applications. It is therefore likely for a transient hardware fault to affect only the context of a single application. We assume that errors affecting the entire computing platform can be handled by other fault tolerance mechanisms (possibly those implemented at the system-level).

We assume that application errors can be detected within a bounded amount of time. Most error detection mechanisms take some time to discover and flag application errors [6]. There is, nonetheless, strong empirical evidence showing that the vast majority of detectable errors is detected within a bounded time interval. An example is the high success ratio of recoveries achieved through low-level checkpointing and rollback in high-end mainframe microprocessors [25, 20]. Implementing effective error detection is fundamental to ensure the success of the recovery process. Thus, if some errors remain latent for more time than expected, we can only give probabilistic guarantees that a successful recovery will eventually happen.

Error detection latency introduces problems in any scheme for checkpointing and rollback recovery. During the time between an error and its detection an application may save an incorrect state. To recover from this type of failure one must restore an older checkpoint. Due to this problem we must maintain, at any time, several past checkpoints and consider each one to be unreliable until the maximum error detection latency has passed after its creation. There is, naturally, a limit on the number of checkpoints that can be maintained in an embedded system.

The integrity of application-level checkpoints is strongly dependent on the failure modes of applications, *i.e.*, their behaviour in faulty circumstances. Since we assume that applications write their own snapshots to main memory, there is a concern that an erratic application may overwrite all previous checkpoints before the error is detected. In this paper we make no direct assumptions on the failure modes

of application. We assume only that the memory area where a checkpoint is stored can be locked from the application by using memory protection.

3.1.2 Memory Protection

The most effective means for securing memory from erroneous accesses is to use hardware-implemented memory protection. Modern microprocessors for embedded applications often include either a memory management unit (MMU) or a memory protection unit (MPU). These are able to prevent applications from accessing outside the intended memory spaces. Freescale's MPC5554 [8] is an example of a recently introduced microcontroller that fulfills these requirements.

Hardware-implemented memory protection is expected to become mainstream in highly critical real-time embedded systems. The main motivation for this is to allow multiple applications to be executed in an integrated platform while ensuring non-interference among tasks. The development of the ARINC 653 standard [1] for the aerospace industry and the AUTOSAR project [9] launched by the automotive industry are examples of initiatives that impose hardware-implemented memory protection.

We assume that applications store their checkpoints in main memory and that we can protect these checkpoints using an MMU. Each checkpointed application has a single pointer to a checkpoint area within its address space. This area can be allocated at startup or it can be statically allocated by the linker – a common design decision in embedded systems. By using an MMU, each application views its address space as a contiguous sequence of virtual addresses. These virtual addresses are then translated by the hardware to physical memory locations. We use this feature to maintain several checkpointing areas in a transparent manner to applications. Each application has a single pointer to a logical checkpoint area but this area is mapped to different physical locations during the execution.

3.2 Assuring the Integrity of Checkpoints

The fundamental requirement on the integrity of checkpoints is that an application should always rollback to a correct checkpoint upon error detection. Since, under our assumptions, an erratic application may overwrite the checkpoint area before the error is detected, we introduce a lock() system call that prevents any further writing to that area. Once a checkpoint is locked, it may be considered reliable after an amount of time equal to the maximum error detection latency has passed. Until then, we have to assume that an error may have occurred before the lock() call was made.

At some point, the application should take another checkpoint without overwriting the previous one. The concern here is that an error may occur precisely when a checkpoint is being taken. For this reason, it is common practice to have at least two checkpoint areas and switch between them [13]. We can make the switch transparent to applications by mapping the logical checkpoint area (in the virtual address space) to a different set of physical addresses. Each application keeps a single pointer to the logical checkpoint area and the lock() function makes the switch by replacing one physical checkpoint area with another.

In our case, however, two checkpoints are not sufficient to ensure that at least one of them is correct. An error may cause the application to checkpoint an incorrect state, call the lock() function and overwrite the second checkpoint area. Note that this sequence of events may occur for any arbitrarily small error detection latency.

To deal with this problem, we opted for having three checkpoints and imposing a minimum time between calls to the lock() function. The three checkpoints are used in a round-robin manner, where a lock() call always locks the most recent checkpoint and unlocks the oldest checkpoint. By having a minimum locking interval greater than the error detection latency, we can ensure that an error can affect at most the two most recent checkpoints. We can formulate this property as the following theorem.

Theorem 1. *If the minimum locking interval is greater than the maximum error detection latency, then, when an error is detected (and rollback is triggered) the oldest of three checkpoints is correct.*

Proof. Let δ denote the minimum locking interval and ϵ denote the maximum error detection latency. By definition of ϵ, an error detected at time t occurs within $[t-\epsilon, t]$. Clearly, in this time interval of length ϵ, at most one lock operation can be executed, since $\delta > \epsilon$. So we have two cases: in $[t-\epsilon, t]$, either the lock operation was not executed or it was executed exactly once.

If one lock operation was executed, then at most two checkpoints may have been affected – the most recent and the previous one. If no lock operations were executed, then the error may have affected the most recent checkpoint, but none of the other two. In either case the oldest of three checkpoints is correct. □

The theorem makes two implicit assumptions. The first is that all checkpoints contain a correct state when the first error occurs. All checkpointing schemes make this or a similar assumption which can be implemented, in our case, by taking three checkpoints at start-up. This means that an error occurring early in a program's execution will bring the computation back to its start. The second assumption is that error detection causes the execution of the erratic application to stop immediately. Since error detection is handled by the operating system, this can be implemented by transfer-

ring the execution to the checkpointing service, which will in turn call the application's exception handler.

This theorem shows that using three checkpoints is sufficient, under our assumptions, to ensure the integrity of checkpoints. However, it abstracts away most of the details involved in creating a practical implementation of our lightweight checkpointing scheme. The following sections elaborate on the necessary implementation details to create an operating system service and describe the usage of model checking to verify its correctness, thereby increasing our confidence in that all details are taken into account.

3.3 Implementation Details

The checkpoint service must allow applications to allocate memory for the checkpoint areas, so that they can save state snapshots. In real-time embedded systems this operation is usually done statically by the linker. However, it is also feasible to introduce a system call for allocating checkpoint areas. In this paper we adopt the static approach.

Another issue related to the configuration of the checkpointing service is that each application must define an exception handler. The checkpoint service transfers control to an application's exception handler when an error affecting that application is detected. The handler must be defined by the application designer in order to implement a lightweight rollback, *i.e.*, to restore the application's state from a stored checkpoint. After restoring the checkpoint, the exception handler should resume the normal execution of the application.

One important implementation detail is to ensure that the time between successive calls to the lock() function is greater than the minimum error detection latency. There are two possibilities: one may count the elapsed time or the number of instructions executed by an application between two calls. Modern microprocessors provide a wide range of performance counters that can be used to monitor diverse parameters of the execution. These can be used to count the number of instructions executed. Counting time is simpler, since we only need access to a timer, and may provide also accurate results.

Assuming that we are counting the number of instructions, the lock() function begins by checking whether the counter has incremented by a programmable amount since the application made the previous call. This implies that we must have a very accurate estimation of the maximum error detection latency. A possibility to make this estimation is to use fault injection testing. This is, however, not in the scope of this paper. As such, we assume only that this parameter can be estimated with sufficient accuracy.

In addition to ensuring a minimum time between locks, the lock() system call replaces the checkpoint areas in a round-robin fashion. Using the address translation features

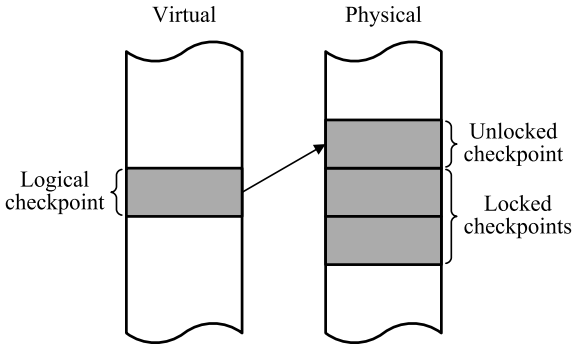

Figure 1. Logical checkpoint area (visible to the application) mapped to one of three physical checkpoints.

provided by an MMU, it replaces the mapping between the virtual addresses seen by the application with one of three distinct sets of real addresses (containing the three physical checkpoint areas). This way, the existence of three physical checkpoints is invisible to applications. Each application has only a single pointer to a logical checkpoint area which is transparently remapped to another physical area when lock() is called. This is illustrated in Figure 1.

The lock operation must be atomic in order to ensure that no checkpointing is taking place when the checkpoint areas are switched. This issue is simple to solve since a system call is typically implemented using a CPU interrupt. In this case we know that no other instructions will execute before the interrupt is handled, *i.e.*, there is no concurrency between the lock() system call and the calling application.

The worst-case execution time of the lock() system call – the only run-time overhead introduced by our scheme – is also an important implementation detail. As mentioned above, this system call begins by checking that the maximum error detection latency has elapsed since the last call (either in units of time or number of instructions) and then rotates the checkpoints. These operations are simple enough to find a bound on the execution time. Moreover, there are no complex operations involved, meaning that the total overhead to execute one lock() system call can be kept as low as few microseconds. For these reasons, we expect no difficulties in meeting the timing requirements of real-time embedded systems.

One final issue that needs to be considered is the content of checkpoints. This can be left to the application designers, since they have the best understanding of which program structures are essential. However, this may also be achieved by using a compiler-assisted technique [17]. In this paper we address only the issue of ensuring the integrity of checkpoints. However, a future extension may consider making the entire process automatic by means of tools.

4 Model Checking using SPIN

Model checking is a process for verifying whether a model fulfills a given specification. A model is an abstract description of a system, written in a formal modeling language. The system's specification is a set of properties, or logical formulæ, which the system is expected to satisfy. Model checking tools accept a model and its specification as input. Their output is either "valid", when the model is correct, or a counterexample, *i.e.*, a case where the correctness properties are violated.

SPIN [11] is an explicit state model checker. It builds a graph of the reachable states, where each vertex explicitly represents a global system state and each edge represents a possible state transition. Verifying a correctness property consists of checking that it holds in all vertices reachable from the initial state.

Model checking tools work on models of the system, which can be built before an actual implementation takes place. Thus, one of the advantages of model checking is the ability to find design faults at early development stages. Moreover, model checking brings us one step closer to an implementation, since we must build the model in a formal language. Unlike the derivation of Theorem 1, we consider all the components interacting in our lightweight checkpointing scheme, namely the application, the exception handler, the checkpointing service and the error detector.

4.1 The PROMELA Modeling Language

The formal modeling language accepted by SPIN is called PROMELA. The PROMELA language is appropriate for defining finite-state transition systems. Concurrent processes can be specified using inter-process communication via global variables (to model shared memory) or via message channels that can be synchronous or asynchronous. It features a syntax inspired by the C programming language. This is a great advantage, since it helps anyone attempting an implementation of the service to understand our scheme in detail.

The PROMELA language is formal enough to be verified using the SPIN tool. However, it maintains all the typical constructs existing in a programming language. Thus, the code provided in the remainder of the paper demonstrates the simplicity of our programming model.

4.2 Modeling the Application

In our programming model a designer stores the application state on the checkpoint area and issues a lock() system call. This is done in the app process, which is defined in Listing 1. The app process loops forever storing the application state on the checkpoint variable, which represents the pointer to the checkpoint area visible to the application. The application state is abstracted as a single bit variable, named app_state, which represents either a correct state (when its value is 1) or an incorrect state (when its value is 0). After saving its state, thereby making a lightweight checkpoint, the app process calls the lock() function by sending a message to the lock channel. This channel is read by the checkpointing service in Listing 3, as we describe later in the paper.

```
bit  app_state  = 1;
bit  checkpoint = 1;
bool exception  = false;
chan lock       = [0] of {bit};

active proctype app()
provided (!exception)
{
  do
  :: checkpoint = app_state;
     lock!0
  od
}

active proctype app_exception_handler()
provided (exception)
{
  do
  :: app_state = checkpoint;
Lrollback:
     exception = false
  od
}
```

Listing 1: Application and exception handler models.

Note that the progress of the app process is nondeterministic, *i.e.*, it may or may not execute any instructions. However, it will only execute provided that no exception has occurred (*i.e.*, when the variable exception is false). The provided clauses force the execution of the two processes to alternate. By setting exception to true the app process becomes blocked and the app_exception_handler process starts executing. The exception is raised by the checkpointing service upon error detection.

The exception handler is a very simple routine that executes only a lightweight rollback, by setting the application state to whatever is contained in the checkpoint, and returns the execution to the main body of the application. This last part is done by setting the variable exception to false, which blocks the exception handler and resumes the execution of the app process. In summary, Listing 1 contains the PROMELA code corresponding to what should be implemented by the application designer.

The application's erratic behaviour is modeled through a process that can only execute provided that the application state is incorrect (*i.e.*, the value of app_state is 0) and no exception has been raised. The code implementing this erratic behaviour is shown in Listing 2.

```
active proctype app_erratic()
provided (app_state == 0 && !exception)
{
  do
  :: checkpoint = app_state
  :: lock!0
  od
}
```

Listing 2: Model of the application's erratic behaviour.

According to our assumptions, the erratic behaviour of an application is non-deterministic. When an error occurs the application may overwrite the checkpoint area and make calls to the lock() function. It may also not execute at all, since we make no assumptions on the progress of a faulty applications. The app_erratic process executes until an exception is raised or the application state becomes correct again. Note that the subtle change in behaviour is modeled by making it possible for the app_erratic process to call lock() without saving any checkpoints and *vice versa*.

4.3 Modeling the Checkpointing Service

The checkpointing service provides the lock() function called by the application. As shown in Listing 3, the checkpointing_service process waits for any message to be inserted in the lock channel and implements the functionality described earlier in the paper. Whenever a lock() call is made, the checkpointing service will lock the most recent checkpoint and make the oldest one available to the application.

In Listing 3, the variables cp1 and cp2 represent the two checkpoint areas which are invisible to the application. When the lock() system call is made the checkpointing service swaps the areas in a round-robin manner. Note that this is done atomically in accordance with the assumptions in the previous section.

The other functionality provided by the checkpointing service is to unlock the oldest checkpoint. To implement this, it waits for messages arriving at the unlock channel. Such messages may only be sent by the error detector. The response to an unlocking event is to make the oldest checkpoint available to the application and to copy the contents of that checkpoint (which must be correct) to all other checkpoint areas. When the unlock call terminates, the checkpointing service raises an exception (by setting exception to true), thereby triggering a lightweight rollback.

```
chan unlock = [0] of {bit};

active proctype checkpointing_service()
{
  bit cp1 = 1, cp2 = 1, tmp;

  do
  :: atomic {
       lock?_ ->
       tmp = cp1;
       cp1 = cp2;
       cp2 = checkpoint;
       checkpoint = tmp
     };
Llock:
     skip
  :: atomic {
       unlock?_ ->
       checkpoint = cp1;
       cp2 = cp1;
       exception = true
     }
  od
}
```

Listing 3: Model of the checkpointing service.

4.4 Modeling Error Injection and Detection

We defined an error_injector process that sets the app_state variable to 0 (representing an incorrect state) at any non-deterministic point in time of the execution, provided that an error is not already active. This process is shown in Listing 4.

```
active proctype error_injector()
provided (app_state == 1)
{
  do
  :: app_state = 0;
Lerror:
     skip
  od
}

active proctype error_detector()
provided (!exception)
{
  do
  :: (app_state == 0) -> unlock!0
  od
}
```

Listing 4: Error injector and error detector processes.

The error_detector process, also shown in Listing 4, implements the error detection functionality. Whenever an er-

roneous state is found, the process may place a message on the unlock channel, thereby notifying the checkpointing process that an error has been found. This process executes provided that no exception is being handled at the moment, as the end result of a call to unlock is only to make the app_exception_handler executable.

4.5 Formal Specification and Verification

SPIN accepts correctness properties specified in linear temporal logic (LTL) [11]. To verify a given LTL formula SPIN creates a *never* claim which consists of the negation of the LTL formula. The verification process consists of checking that there is no possible execution matching the negated formula.

We wish to verify that, when an error is detected, the application is able to rollback to a correct checkpoint. This property is only required to hold if the error detection latency does not exceed the locking interval. As observed in the proof of Theorem 1, this means that at most one lock operation can be executed between an error and its detection. Thus, we want to verify that

$$(\mathit{rollback} \rightarrow \mathit{correct\_checkpoint})\ \mathcal{W}\ \neg(\mathit{error\_injected} \rightarrow (\neg \mathit{lock}\ \mathcal{U}\ (\mathit{lock}\ \mathcal{U}\ (\neg \mathit{lock}\ \mathcal{U}\ \mathit{rollback})))).$$

The property should be read as: *rollback* implies *correct_checkpoint*, unless more than one *lock* occurs between *error_injected* and *rollback*.

The first part of the formula states that *rollback* implies *correct_checkpoint*. This is the fundamental property that we wish to verify. However, it is only required to hold *unless* the lock() function is called more than once before an error is detected. The symbols used in the LTL formula were defined in SPIN's LTL manager as follows:

rollback app_exception_handler@Lrollback

correct_checkpoint checkpoint == 1

error_injected error_injector@Lerror

lock checkpointing_service@Llock

In addition to the usual logic connectives, the above formula uses the temporal modal operators *until* ($\mathcal{U}$) and *unless* ($\mathcal{W}$), also known as *weak until*. Note that the weak until operator is not supported by SPIN but one can use the equivalence $p\,\mathcal{W}\,q \equiv p\,\mathcal{U}\,q \vee \Box p$, which uses the operator *always* ($\Box$), to circumvent this limitation.

We began by finding a counterexample that shows that two checkpoints are insufficient to ensure, under our assumptions, that at least one of them is correct. This was achieved by removing the cp2 variable from Listing 3. This effectively means that the checkpointing service would toggle between two checkpoint areas. SPIN takes a very short amount of time to find a counterexample with the following sequence of events: an error is injected, the application makes an erratic checkpoint, locks that checkpoint and the error detector triggers a rollback recovery. In this case there is only one lock call and the application will rollback to the oldest checkpoint – which is incorrect.

The goal of the modeling effort was, nevertheless, to verify that our scheme is correct when using three checkpoints. Using the code provided in Listings 1 to 4, SPIN is able to search the state space exhaustively and confirm that the model is valid. Thus, we can have a very high confidence in that our scheme works as intended.

5 Extensions to the Baseline Approach

In this section we describe two extensions to the approach described so far in the paper. The first extension considers the usage of multi-core processors to provide increased levels of dependability. The second extension aims at optimizing the performance of rollback recoveries by allowing the application, under certain conditions, to rollback to the previous checkpoint rather than using the oldest one.

5.1 Recovery in Multi-core Processors

The baseline approach, described in the previous sections, can be extended to explore the inherent hardware redundancy available in multi-core processors. In the future we can expect multi-core processors to be used in embedded systems. Thus, we can make use of the available redundancy for improving system dependability.

Our proposal is to use the baseline checkpointing and rollback scheme for tolerating transient hardware faults. If, however, an error persists after rollback, it is likely to be caused by a permanent fault. In this case, it is beneficial to try to execute the same application on one of the remaining cores – instead of shutting down the entire processor node. To achieve this, the baseline approach can be extended to switch the application to a different core and restore a valid checkpoint. The advantage is that permanent hardware faults, such as those related to ageing, may be tolerated using checkpointing.

This approach assumes implicitly that core failures are independent. At the present moment there is little empirical evidence available to support this assumption. However, it is reasonable to assume that there will be a certain degree of isolation between cores belonging to the same integrated circuit. If some non-critical applications are being executed on the same multi-core processor, preempting them to ensure that a critical application can execute on a healthy core can increase the system's dependability.

5.2 Optimizing Rollback Performance

In the baseline approach the application rolls back to the oldest of three checkpoints whenever an error is detected. This decision is safe – as we showed in the preceding sections – but is not optimal. The reason for this is that a checkpoint can be considered reliable after the maximum error detection latency has passed without errors. This fact can be used to optimize the rollback performance.

The routine that unlocks the checkpoints upon error detection is very simple and has no notion of error detection latency. By extending this routine to use the same principle for counting time as the locking routine, we can rollback to the previous checkpoint (rather than the oldest of three) if the error is detected after the maximum error detection latency. In many cases, this means that the application rolls back to a more advanced state, thereby minimizing the amount of re-computation necessary until completion. This improves the worst-case recovery time, which has the advantage of making more task sets feasible under the same failure hypothesis.

6 Conclusions

In this paper, we investigate the use of checkpointing and rollback as a way to tolerate transient hardware faults at the node-level in embedded real-time systems. System-level techniques using replicated processor nodes are a common approach to fault tolerance. However, embedded systems must fulfill strict design constraints concerning hardware cost, power consumption, weight and physical space. In such systems a primary design goal is therefore to make nodes highly dependable.

To this end, an attractive approach for real-time embedded systems is to use application-level checkpointing, *i.e.*, to let the application designer decide the content and timing of checkpoints. However, implementing checkpoint and recovery functions correctly is often non-trivial, even in uniprocessors. If we consider error detection latency and that applications may fail in arbitrary ways, then the integrity of checkpoints must be guaranteed through careful design.

This paper proposes a lightweight scheme for ensuring the integrity of checkpoints in real-time embedded systems. The scheme allows applications to take the checkpoints while providing an operating system service for protecting their integrity. We detail the method for ensuring checkpoint integrity and make a thorough analysis of the implementation details required for building the operating system service. This service offers a simple programming model to application designers and introduces only a small execution overhead to each checkpoint.

A challenge in the development of fault tolerance mechanisms is to ensure that are free from design faults. Over the past years, model checking has become an attractive way to increase the confidence in that a design is fault-free. We used SPIN – a well-established model checker for software systems – to verify the correctness of our approach. One of the advantages of SPIN is that it accepts the PROMELA language, which has a syntax similar to the C programming language. For this reason, the code provided in this paper simplifies the work of understanding and implementing our lightweight checkpointing scheme.

We describe our formal models in detail and provide the complete PROMELA code used to verify the correctness of our approach. These models provide a very efficient method for modeling exception handlers and system calls in PROMELA, meaning that other designs involving similar components can be verified by using a similar model. Using the PROMELA code listed throughout the paper, SPIN was able to search the state space exhaustively and validate our design.

Acknowledgements

The work of Raul Barbosa has been supported by the Portuguese Fundação para a Ciência e a Tecnologia through doctoral grant SFRH/BD/18126/2004. The project was partly supported by the Saab Endowed Professorship.

References

[1] Aeronautical Radio, Inc. ARINC specification 653-1: Avionics application software standard interface, Oct. 2003.

[2] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. Addison-Wesley, second edition, 1993.

[3] R. Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, May/June 2005.

[4] S. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov./Dec. 2005.

[5] G. Cao and M. Singhal. On coordinated checkpointing in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(12):1213–1225, Dec. 1998.

[6] S. Chandra and P. Chen. How fail-stop are faulty programs? *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 240–249, Jun 1998.

[7] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, Sept. 2002.

[8] Freescale Semiconductor, Inc. *MPC5553/MPC5554 Microcontroller Reference Manual (Rev 4.0)*, Apr. 2007.

[9] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst. AUTomotive Open System ARchitecture - an industry-wide initiative to manage the complexity of emerging automotive E/E architectures. In *Proceedings of the 2004 International Congress on Transportation Electronics (Convergence 2004)*, pages 325–332, Oct. 2004.

[10] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Failure resilience for device drivers. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007)*, pages 41–50, June 2007.

[11] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[12] Y. Huang, C. Kintala, and Y. M. Wang. Software tools and libraries for fault tolerance. *IEEE Technical Committee on Operating Systems and Application Environments*, 7(4):5–9, 1995.

[13] I. Koren and C. M. Krishna. *Fault Tolerant Systems*. Morgan Kaufmann, 2007.

[14] C. M. Krishna, Y.-H. Lee, and K. G. Shin. Optimization criteria for checkpoint placement. *Communications of the ACM*, 27(10):1008–1012, Oct. 1984.

[15] P. Krishna, N. H. Vaidya, and D. K. Pradhan. Recovery in multicomputers with finite error detection latency. In *Proceedings of the 23rd International Conference on Parallel Processing (ICPP'94)*, pages II:206–210, Aug. 1994.

[16] J. J. Labrosse. *MicroC/OS-II: The Real-Time Kernel*. CMP Books, second edition, 2002.

[17] C.-C. Li and W. K. Fuchs. CATCH – compiler assisted techniques for checkpointing. In *Proceedings of the 20th International Symposium on Fault-Tolerant Computing (FTCS-20)*, pages 74–81, June 1990.

[18] Y. Ling, J. Mi, and X. Lin. A variational calculus approach to optimal checkpoint placement. *IEEE Transactions on Computers*, 50(7):699–708, July 2001.

[19] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 1(20):44–61, Jan. 1973.

[20] P. Meaney, S. Swaney, P. Sanda, and L. Spainhower. IBM z990 soft error detection and recovery. *IEEE Transactions on Device and Materials Reliability*, 5(3):419–427, 2005.

[21] D. S. Peterson, M. Bishop, and R. Pandey. A flexible containment mechanism for executing untrusted code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, Aug. 2002.

[22] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, Jan. 1995.

[23] N. Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, Aug. 2003.

[24] S. Punnekkat, A. Burns, and R. I. Davis. Analysis of checkpointing for real-time systems. *Real-Time Systems*, 20(1):83–102, Jan. 2001.

[25] K. Reick, P. N. Sanda, S. Swaney, J. W. Kellington, M. Mack, M. Floyd, and D. Henderson. Fault-tolerant design of the IBM Power6 microprocessor. *IEEE Micro*, 28(2):30–38, Mar.-Apr. 2008.

[26] L. M. Silva and J. G. Silva. Global checkpointing for distributed programs. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, pages 155–162, Oct. 1992.

[27] D. Skarin and J. Karlsson. Software implemented detection and recovery of soft errors in a brake-by-wire system. In *Proceedings of the 7th European Dependable Computing Conference (EDCC-7)*, pages 145–154, May 2008.

[28] J. Vinter, A. Johansson, P. Folkesson, and J. Karlsson. On the design of robust integrators for fail-bounded control systems. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, pages 415–424, June 2003.

[29] J. W. Young. A first order approximation to the optimum checkpoint interval. *Communications of the ACM*, 17(9):530–531, Sept. 1974.

[30] Y. Zhang and K. Chakrabarty. Fault recovery based on checkpointing for hard real-time embedded systems. In *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 320–327, Nov. 2003.