Experimental Dependability Evaluation of the Artk68-FT Real-time Kernel

Joakim Aidemark, Peter Folkesson and Johan Karlsson

Department of Computer Engineering
Chalmers University of Technology
S-412 96 Göteborg, Sweden
{aidemark, peterf, johan}@ce.chalmers.se

Abstract. This paper presents an experimental dependability evaluation of a small real-time kernel called Artk68-FT intended for distributed fault-tolerant real-time systems. A main goal of this research is to improve the dependability of such systems by using a two-level approach for tolerating transient faults. By providing mechanisms in the kernel for tolerating transient faults at the node level, the overall reliability is improved since the recovery time is much shorter at the node level than at the system level. Permanent faults and transient faults that cannot be handled at the node level have to be handled at the system level. The experimental evaluation was performed using fault injection experiments on the MC68340 microcontroller executing the kernel and three application tasks. The experimental results show that no wrong results were produced when faults were injected during execution of a critical task protected by Artk68-FT. Moreover, no application crashes were observed in the fault injection experiments with Artk68-FT compared to experiments with a version of the kernel without fault handling mechanisms.

1 Introduction

Dependability is a major concern in embedded real-time computers used for example in vehicular, aircraft or space equipment. Such systems are often safety-critical and must fulfill strict requirements regarding real-time response and fault-tolerance. Aerospace applications are primarily low volume products with moderate sensitivity on component costs while emerging safety critical application areas such as control systems in future automobiles, e.g. steer-by-wire and brake-by-wire without any mechanical backup, are high volume products and therefore more cost sensitive.

A common technique for reducing the cost of fault-tolerant distributed real-time systems is to use computer nodes that exhibit well-behaved failure semantics. One example is to use fail-silent nodes, i.e. nodes that produce either correct results or no results at all (or erroneous results that can be identified by other nodes as erroneous, e.g. through the use of checksums) [1]. FlexRay and TTP/TTA [2] are examples of communication protocols and architectures intended for safety-critical applications, which rely on fail-silent nodes.

Single node failures caused by permanent or transient hardware faults can be tolerated by using two fail-silent nodes in an active redundancy configuration. Such a

configuration, where the same operations are executed on two nodes in parallel, is able to deliver a correct service as long as one of the nodes is operational. Using nodes with less restricted failure semantics, such as those that may produce undetected erroneous results (value failures) requires majority voting to mask errors, which leads to more costly solutions.

To ensure the fail-silent property of a node, it must be equipped with internal error detection and, optionally, error recovery mechanisms. However, most implementations use only error detection for ensuring the fail-silence property. This means that the node is shutdown (directly or after a number of repeated errors) when an error is detected, regardless of whether the error was caused by a transient or a permanent fault. Thus, the detection of an error always activates the fault-tolerance mechanisms provided at the system-level (such as node membership and distributed redundancy management protocols). Recovery of transient faults, which only has a temporary effect, is usually conducted by activating a backup computer node or by reintegrating the failed node after a successful diagnostic test showing that the error was transient, e.g. [3].

Previous research has shown that transient faults are much more common than permanent faults in digital systems [4]. Common causes for transient faults are power fluctuations, electromagnetic interference or particle radiation. Heavy-ions in space and high-energy neutrons at high altitudes are known to cause soft errors in integrated circuits. Due to the reduction of features size of integrated circuits, soft errors have also become one of the most important failure mechanisms in ground-based applications. In fact, one manufacturer claim that for the most advanced CMOS devices operating in a terrestrial environment, the failure rate attributed to soft errors is higher than that of all other hardware failure mechanisms combined [5]. In general, the technology scaling increases the probability of environmentally induced transient faults [6].

In this paper, we consider a two-level approach for tolerating transient faults in distributed fault-tolerant real-time systems. The objective is to detect and recover from most of the transient faults at the node-level. The node-level mechanisms are designed to tolerate transients transparently with respect to other nodes in the system, so that in most cases, the other nodes will not notice that a transient fault has occurred. The advantage of handling transient faults at the node level is that the average recovery time is reduced substantially, which improves the overall reliability. Permanent faults and transient faults that cannot be handled at the node level must be handled at the system-level. Thus, for those faults, the node must be fail-silent.

We present an experimental evaluation of a small real-time kernel, called Artk68-FT, which we have developed to support node-level transient fault tolerance. In [7] we have evaluated the use of a time redundancy technique called Temporal Error Masking (TEM) to tolerate transient faults in the application tasks. The performance improvement rate of microprocessors and micro-controllers make time redundancy increasingly attractive for achieving fault-tolerance in real-time systems. In TEM, critical tasks are executed twice and the results compared to detect errors. A third execution is started if an error is detected by the comparison or by other error detection mechanisms. This allows transient faults to be masked by conducting a majority vote on the three results, thereby facilitating node-level fault tolerance. It is implicit that enough slack is available in the schedule to allow recovery without causing any other tasks to miss its deadline (see fault tolerant scheduling in e.g. [8]).

The Artk68-FT kernel supports preemptive fixed priority scheduling [9] and employs TEM to tolerate transient faults in the application tasks and several other mechanisms for handling faults in applications as well as the kernel. Thus, a contribution of this paper is an evaluation of TEM in combination with other fault handling mechanisms included in a real-time kernel. In particular, we evaluate the use of statically sized arrays [10] for basic kernel data-structures to improve fault tolerance. Most other real-time kernels uses dynamic data structures, i.e. linked lists, to maintain tasks. The advantage of using statically sized arrays is that the memory address of the array is fixed at compile time and can be saved in read only memory. This allows for some inherent fault tolerance, as the structures cannot be corrupted by a fault (although the contents of the structure must still be protected).

We have implemented the Artk68-FT kernel for the Motorola 68340 microcontroller and evaluated it using fault injection, a well-established experimental dependability validation technique [11]. Previous fault injection studies have shown that the behavior of operating systems in the presence of faults can be rather unpredictable even when several error detection mechanisms are included [12] [13] [14]. Application tasks or the operating system may crash or hang due to faults, often resulting in system failures. In this study, single bit-flip faults were injected into the registers of the Motorola 68340 microcontroller [15] using the GOOFI (Generic Object Oriented Fault Injection) tool [16]. A bit-flip, i.e. the inversion of a bit in a memory cell, may be the direct cause of a transient fault. A bit-flip in a register may also represent an erroneous computation due to a transient fault (pulse) in combinational logic [17]. We use software implemented fault injection (SWIFI) [11] to inject the faults during execution of both the operating system kernel and application tasks. No faults were injected into the main memory since we rely on the use of error correcting codes to mask such faults.

Fault injection experiments were also conducted on a version of the kernel without fault tolerance mechanisms, called Artk68 [18]. Artk68 uses dynamic data structures for task administration and is mainly targeted for small, embedded systems, which are not safety critical. By comparing the results of the experiments conducted on the two different kernel versions, the dependability improvements provided by the fault handling mechanisms included in Artk68-FT are evaluated.

Section 2 presents the related work. Section 3 gives an overview of the implementation of the real-time kernel describing both the Artk68 kernel and the modifications made to increase the fault tolerance in Artk68-FT. Section 4 presents the experimental setup used for validating the kernel, while the results from the fault injection experiments are presented in Section 5. Finally, the conclusions and future work are given in Section 6.

2 Related Work

Most commercial operating systems usually provide a number of mechanisms for detecting software design errors, e.g. by returning error codes from faulty function calls or detecting stack overflows caused by erroneous software. Although software design errors are not the focus of this paper, such mechanisms are also able to detect hardware errors [14]. Errors may also be detected by on-chip error detection

mechanisms in microprocessors such as illegal op-code detection, division by zero or address range checking. Many microprocessors also provide a memory management unit (MMU) that can be used to restrict memory access and thereby ensure that an application does not overwrite the memory of other applications or the kernel.

Although modern operating systems typically provide several error detection mechanisms, fault injection studies have shown that the behavior of the operating system in the presence of faults can be rather unpredictable. In [12], faults were injected into the parameters of operating system calls. The experiments showed that a fault in the parameters may cause a single task to crash or hang, or the complete operating system might crash without producing any information. In [13], faults where injected into the function parameters and the memory (code and data) areas of the Chorus microkernel. Besides crashes and hangs, the injected faults also caused the application tasks to produce erroneous results. One of the most vulnerable functions in the microkernel is the part handling the synchronization of tasks. The experiments showed that a majority of the faults injected into the parameters of the function calls to the synchronization unit resulted in an erroneous output. In [14], faults were injected into the registers and memory of a microprocessor executing the LynxOS. Almost a third of the injected faults caused the operating system to crash while slightly more than one percent of the injected faults resulted in faulty outputs being produced by the application.

The effect of the various failures is obviously application dependent. However, for safety critical real-time systems it is often important to detect any errors promptly to allow error processing to be conducted within a specified time. A crash of an application or the operating system may not result in benign failures such as fail-silent behavior, since the output may become locked to an erroneous value.

Several studies addressing fault tolerance mechanisms for handling faults both in the application and the operating system have been presented in the past. However, most approaches focus on error detection to achieve fail-silence in an active redundancy configuration and disregards node-level transient fault tolerance. In [19], error detection mechanisms integrated into the MARS operating system to provide fail-silence were evaluated. The MARS operating system supports static cyclic scheduling and includes error detection mechanisms such as *robust data structures* to allow integrity checking of data structures like linked lists, and *reasonableness checks* on parameters for detecting errors in system calls. For detecting errors in the application tasks, tasks are executed twice and the results are compared. Moreover, *timing checks* are used to detect if tasks violate any deadlines and *stack checks* are used to check if the tasks' stack limits are exceeded.

In [20], a middleware layer executing on top of an existing microkernel was evaluated. The middleware layer, called Hades, provides extensive software implemented error detection mechanisms to ensure fail-silence, e.g. tasks are checked for *deadline violations* or worst-case execution times and the arrival times for periodic or sporadic tasks are checked. Checking deadline violations in a static cyclic scheduler like MARS is rather straightforward. Violations are detected by checking if a task is currently executing when a clock interrupt is triggered. As Hades allows dynamic scheduling, the deadlines are checked by the operating system only when the task is finished, blocked or if the task is preempted. Messages are also provided with checksums. Data structures such as lists are provided with redundant information for integrity checks and checks for ensuring the validity of array indices are employed. A

so-called *flow call graph* is used to check the execution flow of the operating system. Extra code is added to the beginning and the end of every function in the operating system to check that each function is called only from pre-defined functions. In addition, instructions are inserted into unused memory areas to allow erroneous memory accesses to be detected. An error detection coverage of 99.1% was obtained from experiments injecting single bit-flip transient faults in the memory areas (code, data and stack) of both the Hades middleware and the underlying Chorus microkernel.

In [21], wrappers (software checks) are added on top of the microkernel to detect erroneous values that are passed to and from kernel components such as synchronization and scheduling components. To allow effective implementation of the wrappers, the behavior of different components are defined as predicates, e.g. a predicate may define that a task released from the semaphore queue must end up in the ready queue. However, the wrapper approach requires that the microkernel supplier implement a metainterface that allows interception of kernel calls and internal function calls

As shown, several studies have addressed error detection in the application and the operating system to achieve fail-silence for system level fault tolerance. Our objective is to evaluate mechanisms for tolerating transient faults already at the node-level. As only about 5% [22] of the total execution time is used by the kernel, we have focused on tolerating faults occurring during execution of application tasks. Faults occurring during execution of the kernel are tolerated at the system level by relying on fail-silence.

3 Kernel Implementation

A real-time operating system (RTOS) is an operating system with well-defined time constraints, i.e. the time to execute the functions in the RTOS is bounded. Most RTOS are scalable, i.e. they have a core part that implements the basic functions (called microkernel) for which extra functionality can be added depending on the application. This is done to reduce the size of the RTOS, e.g. many automotive applications do not need to include file handling and disk storage functionality. The main parts of a microkernel supporting fixed priority scheduling usually include functions for *task scheduling*, *task synchronization*, *time management*, *inter task communication*, *memory management* and *interrupt management*.

The *task scheduler* is responsible for the creation, activation, suspension and termination of tasks. It is also responsible for determining the order in which tasks are executed. The task scheduler for the Artk68 kernel supports dynamic fixed priority scheduling (FP). In FP each task is assigned a fixed priority before run-time. At run-time, the task with the highest priority is allowed to execute first. The kernel supports preemptive scheduling. This means that a lower priority task can be interrupted and suspended at any point during the execution, thus enabling execution of a higher priority task. The advantage of fixed priority preemptive scheduling over static cyclic scheduling is higher flexibility and faster response time as a higher priority task does not need to wait for a lower priority task to complete its execution. However, the kernel becomes more complex since preemptive scheduling requires that a separate stack must be maintained for each task and that access to common resources must be

synchronized to avoid concurrent access of shared data. Moreover, a fixed priority scheduling system may also experience higher *output jitter*, i.e. time variations in the delivery of result due to interference from other tasks, than a static cyclic scheduling system.

Task synchronization is used to allow tasks to wait for an event or for tasks to synchronize with each other. The Artk68 implementation uses task synchronization through semaphores, which is the most common method. A semaphore can be initialized as a binary semaphore taking only two values, or a counting semaphore that can take a range of values. The current implementation does not support a priority inheritance protocol. Therefore there is a risk for priority inversion, i.e. if three or more tasks use a specific semaphore, a lower priority task may be allowed to execute before higher priority tasks. To avoid priority inversion in this study, only two tasks may share a semaphore.

Inter task communication involves exchanging data between tasks. In our kernel, this is conducted over shared resources, i.e. the designer is required to implement mailboxes (predefined data structures) using semaphores to avoid concurrent access.

Timer management is used to schedule periodic tasks, which are executed and then suspended for a specified time interval. Timer interrupts may be implemented either tick driven or event driven. In tick driven management, periodic interrupts are triggered at regular time intervals (e.g. every 10 ms) checking if any suspended tasks should be activated. In an event driven management, a timer is set to interrupt for the task with the closest timeout. A problem with a tick driven system is deciding the interval for the periodic interrupt. Too frequent interrupts increase the overhead and too infrequent interrupts leads to *jitter*, i.e. the activation time of a task may be delayed up to one tick. Artk68 uses event driven management.

Memory management provided by the microprocessor influences how the application tasks interact with the kernel. Artk68 is implemented for the Motorola 68340 microcontroller, which does not support a memory management unit (MMU). Hence, the kernel and the tasks operate in the same address area and kernel calls are implemented as simple subroutine calls. For systems providing a MMU, it is possible to put the kernel and each task in a separate memory area. Thereby, the tasks and the kernel are protected and a faulty task is not able to corrupt other tasks or the kernel. A drawback using separate addresses is increased task switch time, as the memory areas must be altered.

Interrupt management allows efficient interaction with the environment. In Artk68, a task may be associated with a specific event through semaphores. For example, a task is suspended on a semaphore and is released by an interrupt service routine that is triggered by an external event.

The Artk68 kernel is mainly implemented in sequential Ada (no concurrency features of the Ada programming language, such as Ada tasks or protected objects, are used). In addition, a few low-level routines such as timer handling and task switch handling are written in assembly language. The basic kernel calls used to handle task scheduling and task synchronization in Artk68 are shown in Table 1.

Each task can be in the *running*, *ready* or *waiting* state. The running state corresponds to a task that is currently using the CPU. A task is ready when it is able to execute but has lower priority than the task in the running state. A task in the waiting state may be blocked for a specific time period or waiting for an event. Each task is associated with a data structure called a task control block (TCB). The TCB holds

information about the task such as the identity of the task, the pointer to the task stack where local variables are saved together with the context of the microcontroller (basically its register values) during task switch. A transition from one state to another requires a context switch, i.e. switching of the running task. The environment such as registers, program counters and stack pointer must then be altered. Although the scheduler in Artk68 only saves the stack pointer in the TCB and the remaining environment on the tasks stack, another approach would be to save the complete environment in the TCB.

Table 1. The basic kernel calls in the Artk68 kernel

Function	Description
New_process (id, priority, address..)	Create a new TCB and allocate a stack for the new task
Exit_process	Remove the task from the task set
Sleep_until (time)	Suspend a task for a given period of time
New_sem (sem_id, value)	Create a new semaphore and set an initial value of the semaphore
Wait (sem_id)	This function is called to check if a resource if free or to suspend a task until a event occurs
Signal (sem_id)	This function is called to free a locked resource or activate a suspended task

3.1 Non Fault-tolerant Kernel Implementation

Artk68 uses a traditional linked list implementation of basic data structures such as dynamic queues for maintaining the TCBs (kernels usually have at least one queue for each state a task can be in). This allows tasks to be dynamically created during run-time. The allowed number of tasks may then only be limited by the available memory space. Linked lists also simplify moving TCBs between queues by just modifying the TCB pointers. Artk68 maintains a priority-based FIFO queue for the tasks in the ready state, called *ready queue* and also for each semaphore used by tasks waiting for events. In addition, the kernel maintains a *timer queue* used for tasks that are suspended for a specific period of time. The tasks in the timer queue are ordered by their wake-up times so that the task with the closest wake-up time is placed first. Two timers are used in the kernel, one for maintaining a real-time clock and one used as event timer. When inserting a task in the timer queue, the event timer is set to expire at the point in time when the first task in the timer queue shall be released. A task in the timer queue is released by moving it to the ready queue. A dispatch is then activated to ensure that the task with the highest priority is executing.

3.2 Fault-tolerant Kernel Implementation

Artk68-FT is a redesigned version of Artk68 using statically sized arrays (lookup-tables) for the main kernel data structures instead of linked lists among other techniques in order to improve fault tolerance. Previously, statically sized arrays have been used with the objective to design a kernel with low jitter and predictable kernel overhead, e.g. in the Asterix real-time kernel [10]. Like Asterix, the Artk68-FT kernel uses an array for storing the static data of each task, e.g. the task's period time, and

one array for storing the data of each task that is changed during execution, such as the wake-up times of tasks and the pointer to the kernel's stack. In addition, one array for each semaphore queue is used. The size of an array is specified when it is initialized. Thus, the size of the TCB array corresponds to the maximum number of tasks in the system and the size of each semaphore array corresponds to the maximum number of users of the semaphore.

The main work of a scheduler essentially involves inserting and removing TCBs in and out of queues. For linked lists, this means proper maintenance of pointers. Thus, a transient fault in the registers of the CPU may affect a pointer when inserting or removing a TCB, which can corrupt the whole queue. To allow fast detection of such errors, a linked list approach may be complemented with robust data structures [19]. However, adding redundant information to the queues and performing consistency checks on each queue access may cause substantial overhead.

Instead, using arrays for storing static TCB data may provide some inherent fault tolerance as the arrays can be stored in a ROM and therefore not be erroneously overwritten by an error in the CPU. For arrays with dynamic contents, the location of the array and the index is fixed and thereby defined before run-time. Thus, each TCB is accessed through an index instead of pointers. Since the memory address of the array is fixed at compile time, it is included into the code area and may therefore also be stored in a ROM. This provides the possibility to recover from errors corrupting the index while accessing the array by simply re-executing the array access. Moreover, the Ada programming language used for our kernel implementation provides a range check that detects indices outside the bounds of the TCB array.

A requirement for Artk68-FT is that all tasks have unique priorities since the array index represents both the identity and priority of the task. Artk68-FT does not use separate data structures for ready and waiting tasks, instead the task is just marked as ready or waiting in the TCB status field. To dispatch the task with the highest priority, the scheduler has to scan the array starting from the first index for the first ready task. Thus, a drawback with the static array approach is the overhead when there are many tasks in the array since the worst search time is proportional to the number of tasks. In addition, the static array approach limits the flexibility of the kernel, e.g. it is not possible to dynamically create tasks and use round robin scheduling for tasks with equal priorities. However, if these drawbacks can be overseen, as they often can for embedded systems with small task sets, the approach provides means for cost-effective implementation of error detection. Thus, from a fault tolerance perspective, using statically sized arrays to maintain the TCBs instead of linked lists may be preferred.

Besides using static data structures, other software implemented error detection mechanisms have been added to Artk68-FT:

- *Task switch checks*: The size of the TCB tables storing static and dynamic task data is statically set to the corresponding maximal number of tasks. Ada provides constraint checks that checks that the indices to the two TCB tables are within this range. Moreover, when a task switch occurs, the state of the currently running task is updated in the dynamic TCB table and an *update TCB check* is made which checks the update by reading back the task's state. Then the highest priority ready task is selected to be running by checking the state of each task in the TCB table twice to ensure that the correct task is selected. Furthermore, the states, which can be either *running*, *ready, suspended* or *blocked*, in the TCB are

encoded using *m-of-n* codes [23] to facilitate detection of single bit errors when checking the states.

- *Stack range check:* Checks that the location of a task's stack is within the allowable range. This check is made each time a task switch occurs.
- *Timer check:* The event timer is set to expire after a defined number of *ticks*, which is calculated as *ticks = wake-up time – current time * factor*. Hence a check is made for each task that the number of ticks calculated should be less than a pre-computed *tick_check*, where *tick_check = (period – task execution time) * factor*.
- *Timer interrupt check*: Checks that at least one task is released for each event timer interrupt. Hence it may detect tasks for which the adjusted event timer value is too short compared to the task's wake-up time. A check that the event timer is restarted if there are any suspended tasks remaining is also conducted.
- *Next time check:* Checks that the computation of the next release time for periodic tasks (next time = next time + period) is made correctly by conducting the computation twice and comparing the results.
- *Semaphore checks:* The semaphore queue size is restricted to the maximum number of tasks that can be blocked on the semaphore and is checked by Ada constraint checks. Moreover, only binary semaphores which are encoded to detect single bit errors (free=01 and taken=10) have been implemented in Artk68-FT, which gives a more restricted behavior (although counting semaphores may be provided with range checks). In semaphore *wait* calls, checks are made that the semaphore's value is either free or take, otherwise an exception is raised. Similarly, when calling semaphore *signal*, the semaphore value must be taken otherwise an error is raised.

In addition to the above mentioned error detection mechanisms, Artk68-FT also provides support for temporal error masking (TEM), i.e., checking, and if possible also tolerating, errors in the task computations using time redundant execution of tasks and comparison of results. An example with three scenarios of error detection and error recovery in TEM is given in Figure 1.

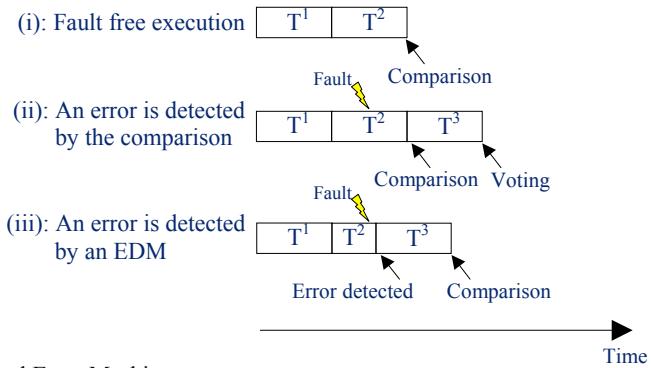

Fig. 1. Temporal Error Masking

In fault-free operation, see (i) in Figure 1, a critical task, *T*, is executed two times (denoted T^1 and T^2) and a comparison is made to detect errors. As the results match, a third copy does not have to be executed and the time may be used by other tasks. In

(ii) an error is detected by the comparison and a third copy of the task, T^3, is then executed. The results of the three copies are checked by a majority vote. If the majority voter detects two matching results, they are accepted as a valid result of the task, otherwise no result is delivered, which leads to an omission failure. In the final scenario (iii), an error is detected by an error detection mechanism (EDM). The affected copy, T^2, is then terminated and a new copy, T^3, is started immediately. The new copy will use time reclaimed from the terminated copy as well as time from any available slack. A comparison is made to confirm that the results match before a result is delivered.

Our implementation of TEM requires that a task is executed in a periodic *read input - compute - write output* loop. The input data is received in the beginning from input devices or other tasks. The input data is then processed and the results are sent to actuators or to other tasks in the system at the end of the loop, see Figure 2.

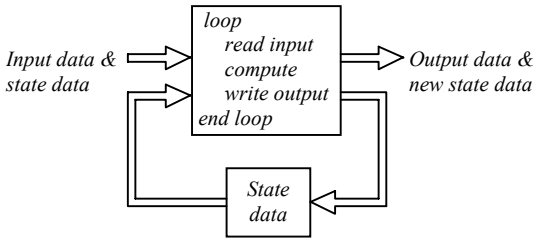

Fig. 2. Example task

To conceal the details of the error handling from the application designer in this implementation, an Ada generic package is used. The generic package contains a general algorithm for execution according to TEM. Hence, before using the algorithm, the package must be instantiated with the actual data types and functions used. An example of this is shown in Figure 3 where an instance, *T3_FT*, of the generic package *generic_TEM* is declared. The parameters to the generic package are the types for the task input and output data. Input data may typically include both sensor values and any state data, i.e. data accumulated from previous executions, while the output data include the result from the application and updated state data (see Figure 2). Parameters appended to the generic package also include the name of the functions implemented by the user, i.e. *read_input_data, calculate, write_output_data*. This is made to allow the TEM algorithm to control when new input data should be fetched, when to conduct repeated computations, and decide if a result should be delivered or not.

Tasks in Artk68 and Artk68-FT are defined as procedures. Figure 3 shows a periodic task (T3) employing TEM in the Artk68-FT kernel. Instead of calling the user-implemented functions directly, functions in the generic package are called. The function *T3_FT.read(data)*, fetches new input data and updates the global variable *data.* The function *T3_FT.compute(data, result)* computes the result for the first copy and updates the global variable *result*. The reason for saving the data in global variables (i.e. saving data in the data area instead of in the stack) is that the tasks stack is restored to an initial state (cleared) when restoring the task context after a hardware exception or an Ada exception is triggered.

The function *T3_FT.check_and_write(data, result)* conducts additional computations and compares the results to detect errors. An output is delivered if the results match. If the results do not match, a third execution of the computation is executed. The results of the three copies are then checked by a majority vote. If the majority voter detects two matching results, they are accepted as a valid result of the task, otherwise no result is delivered, which leads to an omission failure. As the new state data is part of the result, the state data is also updated when two matching results have been produced.

```
package T3_FT is new generic_TEM(input_type, output_type,
        read_input_data, calculate, write_output_data);
indata  : input_type;
outdata : output_type;
procedure T3 is
begin
  init_next_time;
  loop
     T3_FT.read(data);
     T3_FT.compute(data, result);
     T3_FT.check_and_write(data, result);
     sleep_until_next_time;
     update_next_time;
  end loop;
  exception
     when others => recover;
end T3;
```

Fig. 3. Example of an initialization of a generic package and a task employing TEM

Errors can also be detected by hardware and software EDMs. In this case, the procedure *recover* is called. The procedure *recover* restores the task's context, such as the program counter and stack pointer etc., to an initial state. Additional computations are conducted and their results are compared before delivering an output.

Before starting a third execution if an error is detected, the kernel checks whether it is feasible to re-execute the task and meet the deadline. Hence in addition to storing the tasks period time in the TCB, this also requires that the tasks worst-case execution time (WCET) and deadline to be saved.

4 Experimental Setup

This section presents the experimental setup used for evaluating Artk68 and Artk68-FT. The objective of this study is mainly to investigate the effect of transient faults occurring in the CPU during execution of basic kernel functions, i.e. the task scheduler and the task synchronization. To derive actual dependability measures for the complete kernel, such as error detection coverage [23], the injected faults should be representative of the complete set of faults which can occur in the system, requiring all different tasks execution scenarios to be considered. Nevertheless, the experiments give an indication of the coverage and the results may also be used to identify weaknesses so that appropriate error handling can be suggested.

4.1 Target System and Fault Injection Tool

Target system: The target system for our experiments is a microcontroller board featuring a 32-bit Motorola 68340 microcontroller, which contains a core processor based on the Motorola 68k architecture. The MC68340 does not provide memory protection or a floating-point unit. The MC68340 has 8 data registers (D0-D7) and 8 address registers (A0-A7) which are 32-bit wide, as well as a program counter (PC) and a status register (SR), which are all reachable by the fault injection tool. The D0 register is used both for data computations and return values when returning from function calls. Register A6 is used as a frame pointer (FP), and register A7 is used as the processor's stack pointer (SP). Parameters to functions are passed trough the stack rather than via registers. Local variables and parameters in the stack are accessed using an offset relative to the FP.

Error detection mechanisms: Table 2 gives an overview of the error detection mechanisms provided by the Motorola MC68340 microcontroller and Ada95 [24].

Table 2. Description of error detection mechanisms provided by the hardware and Ada

Selection of Ada run-time constraint checks	
Ada access check	Attempt to follow a null pointer
Ada range check	Attempt to violate a range constraint of a scalar value
Ada index check	Attempt to access an index that is not in the range of the array
Microcontroller hardware checks	
Bus error	Attempt to access non-existent memory
Address error	Attempting to access a word or a long-word on an odd memory address.
Illegal instruction	Attempting to execute a non-existing instruction
Line 1010	Attempting to execute an unimplemented instruction
Line 1111	Attempting to execute an unimplemented instruction (used for M68000 extensions)
Division by zero	Raised if a division instruction is given a divisor value of 0
Privilege violation	Attempt to execute a privileged instruction in user mode
Format error	Erroneous stack frame format when executing an RTE instruction

Fault injection tool: For the experimental evaluation, software implemented fault injection (SWIFI) with the GOOFI tool [16] is used. The SWIFI fault injection algorithm requires that a routine for receiving fault injection data from the GOOFI tool, a trap handler and a trace handler routine for performing the fault injection are located in the target system memory.

The user first selects the fault injection locations, the points in time the faults should be injected, the target system workload and the number of fault injection experiments to perform. Then, each fault injection experiment begins by reinitializing the target system and downloading the workload and fault injection data. This includes a break-point address and the number of accesses that should be made to that address until a fault is injected. In addition, the register and bit in the register in which a fault should be injected are downloaded. Then the op-code at the chosen break-point address is replaced with a software trap. After this the program is started and the system will continue as normal until the software trap is reached. Then, a jump is made to the trap routine that checks if the number of accesses to the chosen address equals the number of accesses that should be made until fault injection. If it matches, a fault is injected, otherwise the system continues (after restoring and executing the

original op-code followed by re-inserting the software trap) until the next time the trap is reached. When the application has completed, the result from each task execution, the time when each task has completed and the type of exceptions that may have been triggered is sent to the GOOFI tool for later analysis.

Intrusiveness: The time overhead (*OH*) for injecting faults using SWIFI includes the time for observing (Obs_t) the number of accesses (k) to a specific program address, as well as the time to inject a fault (FI_t), i.e. $OH = k \cdot Obs_t + FI_t$. Thus, the timing measurements in the experiments are adjusted to compensate for this overhead. The timers on the microcontroller board used have a clock resolution of approximately ±30.5 μs, which also limits the accuracy of the timing measurements. In addition, the time to inject a fault also varies ±50 μs depending on which register the fault is injected in.

Workload: The effect of a fault depends on system workload activity. In these experiments, the workload consists of the kernel and three periodic tasks (called *T1*, *T2* and *T3*). Task *T3* executes a brake-by-wire algorithm [7] and has the highest priority. Since *T3* is a typical safety critical real-time application, it is protected using TEM in the Artk68-FT kernel. Task *T1* and *T2* are non-safety critical and therefore unprotected and mainly used for exercising the semaphore routines. Both *T1* and *T2* execute the same matrix multiplication algorithm, where the output from task *T1* is used as input to task *T2*. This handled by two semaphores called *S1* and *S2,* see Figure 4. Semaphore *S1* is initialized to 0 and semaphore *S2* is initialized to 1.

```
procedure T1 is
...
begin
  next_time:= my_clock + period;
  loop
    wait(S2);
    data := read_input_T1;
    result := mult(data);
    save_result_T1(result);
    signal(S1);
    sleep_until(next_time);
    next_time:=next_time + period;
  end loop;
end;
```

```
procedure T2 is
...
begin
  next_time:= my_clock + period;
  loop
    wait(S1);
    data := read_input_T2;
    result := mult(data);
    save_result_T2(result);
    signal(S2);
    sleep_until(next_time);
    next_time:=next_time + period;
  end loop;
end;
```

Fig. 4. Task T1 and T2

In Figure 5, the first execution of each task is shown. Task *T3* executes firsts since it has the highest priority. Then task *T3* completes and is suspended until its next period. After this, task *T2* is started, but is blocked on semaphore *S1*. Task *T1* is therefore started and allowed to perform its computation. When task *T1* calls *S1*, it releases task *T2,* which preempts *T1* and completes its execution. After *T2* completes its execution, *T1* resumes and completes. Four task iterations are executed for each task before the kernel terminates.

Fault model and fault injection locations: Transient faults are modelled as single bit-flips. The single bit-flip model has become a de-facto standard in fault injection experiments, although it is not a perfect representation of all transient faults. Single bit-flip faults selected randomly using uniform sampling were injected into the data registers, address registers, program counter register and status register of the MC68340 processor.

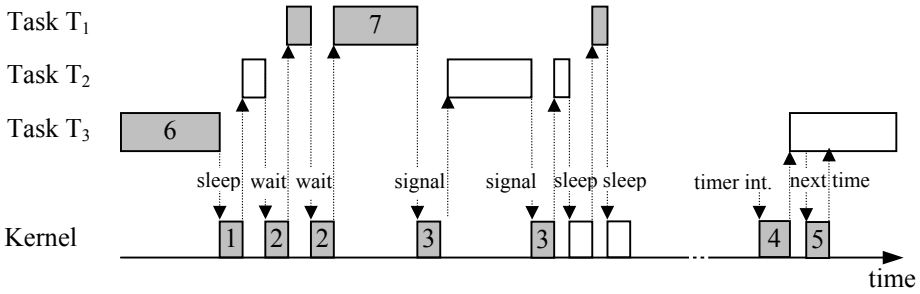

Fig. 5. Execution of tasks where faults are injected during time intervals marked in grey

Time interval for fault injection: Faults were injected during execution of the following functions: *Task switch, Wait semaphore, Signal semaphore and Timer interrup*t indicated by number 1-5 in Figure 5 and described further below. Faults were also injected during execution of task *T3* to evaluate the effectiveness of TEM. In the experiments on Artk68, faults were also injected during execution of task *T1* to investigate the effects of the faults depending on workload activity. Faults were injected during the first task iteration in each experiment and the behavior of the system was observed in the following three iterations. A break-point address corresponding to a point in time for fault injection was selected randomly using uniform sampling among the addresses obtained for each function through an execution trace from a fault free run. A detailed description of the functions executed for the fault injection experiments follows:

1. *Task switch* - A task switch occurs when task *T3* calls the *sleep_until* call which involves saving the context of the executing task, inserting the task in the timer queue and starting the timer which expires when the task should be woken the next time. After this, a dispatch is made by fetching the first task (task *T2*) in the ready queue and restoring its context.
2. *Wait semaphore* - Involves examining the value of the semaphores. The semaphore is taken if it is available, otherwise the task is suspended and inserted into the queue of the semaphore followed by a dispatch. Here, the semaphore is unavailable in the first call and available in the next.
3. *Signal semaphore* - Involves releasing the first task in the semaphores queue, i.e. moving the task from the semaphores queue to the ready queue. If the removed task has higher priority than the running task, a dispatch is made. Here, the first call to *signal semaphore* results in a task switch and the second only increases the value of the semaphore.
4. *Timer interrupt* - When a time out for a task has expired, the task is moved from the timer queue to the ready queue. If a task remains on the timer queue, the timer is restarted. A task switch occurs if the released task has higher priority than the running task.
5. *Next time* - Updates the next activation time of periodic tasks.
6. *Task T3* - Faults are injected during execution of the brake-by-wire application.
7. *Task T1* - Faults are injected during execution of the matrix multiplication.

4.2 Error Classification and Definitions

Errors can be classified into *non-effective* and *effective errors*. The non-effective errors have no effect on the system behavior, i.e. a correct result is delivered. This occurs when a fault is not activated, e.g. the location where the fault was injected is not used, or the fault is overwritten by uncorrupted data. Effective errors correspond to errors, which are detected by the error detection mechanisms or escape the mechanisms resulting in crashes or erroneous results being produced by the application tasks. A description of the effective errors is given in Table 3.

Table 3. Classification of effective errors

Effective errors	Description
Wrong result	Errors that escape the error detection mechanisms causing erroneous results to be produced by the application tasks.
Timing error	A timing error occurs when the delivery time of a correct result deviate more than ±250 μs, compared to the fault free run.
Detected errors	Errors that were detected by the error detection mechanisms
Application crash	The injected fault caused one or several tasks to stop producing any output
Kernel crash	The injected fault caused the kernel to stop working, i.e. no task is able to produce any output
CPU crash	The injected fault caused the CPU to crash

5 Results for Artk68

As shown in the Table 4, a majority of the injected faults resulted in non-effective errors. The number of non-effective errors is around 80% of the injected faults for each different kernel function. The reason for this may be that the registers in the MC68340 are sparsely used and the *liveness* of the data in registers is usually short, i.e. data is fetched into a register in one instruction and then used immediately in the next instruction. In addition, the register usage is reduced by passing parameters to functions through the stack rather than via registers. Below, an investigation of the causes for the effective errors is made for each error category.

Table 4. Results of fault injection experiments on Artk68

	Task T1		Task T3		Kernel functions									
	Matrix		BBW		Timer int.		Task switch		Sem. Wait		Sem. Signal		Next time	
No. injected faults	2207		2285		2178		1967		1936		2055		1494	
Correct result	1695	76.8%	1860	81.4%	1789	82.1%	1570	79.8%	1650	85.2%	1736	84.5%	1219	81.6%
Wrong results	166	7.5%	54	2.4%	1	0.0%	0	0.0%	1	0.1%	0	0.0%	0	0.0%
Timing errors	0	0.0%	30	1.3%	23	1.1%	28	1.4%	1	0.1%	2	0.1%	56	3.7%
Application crashes	0	0.0%	0	0.0%	1	0.0%	2	0.1%	4	0.2%	6	0.3%	0	0.0%
Kernel crashes	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
CPU crashes	52	2.4%	61	2.7%	54	2.5%	43	2.2%	42	2.2%	43	2.1%	45	3.0%
HW EDM	289	13.1%	277	12.1%	267	12.3%	292	14.8%	194	10.0%	228	11.1%	169	11.3%
Ada EDM	5	0.2%	3	0.1%	43	2.0%	32	1.6%	44	2.3	40	1.9%	5	0.3%

Wrong results occur mainly when injecting faults during execution of the tasks. However, wrong results are also produced when injecting faults during execution of the kernel Timer interrupt and Semaphore wait functions. For example, when the synchronization between task *T1* and *T2* is affected and *T2* continues to execute without waiting for task *T1* to update its input data. Synchronization may be affected due to corruption of a semaphore value, or due to an error in the program counter, e.g. when calling the *wait* function causing incorrect control flow and thus returning to the task without performing any synchronization.

Application crashes have been observed when injecting faults during execution of the kernel queue handling routines, e.g. when a TCB is not inserted properly in the queue, or the queue is corrupt after removing a TCB. Figure 6 shows an example of an application crash due to corruption of the queue after removing a TCB. Here the first task in the ready queue will be removed. Before executing the function, the queue contains task *T1* and *T2* (*queue_head→T2→T1→null*) and afterwards the queue should only contain task T1 (*queue_head→T1→null*). However, a fault injected into register a0 at address 61bc results in omitting the operation "*queue.head := queue.head.next*". Thus when the operation "*element.next := null*" is made, *queue.head.next* is also set to null which disconnects task *T1* (*queue_head→null*).

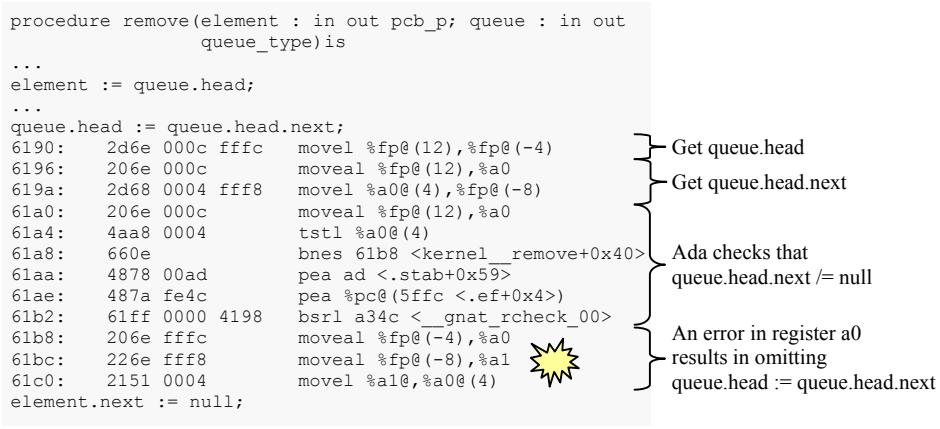

Fig. 6. Example of an error in the queue routine causing an application crash

Kernel crashes occur when the kernel stops working. No such crashes were observed during these experiments.

CPU crashes occur if the processor halts without producing any information about the encountered error. Such crashes were observed for faults affecting the stack pointer (SP). An indirect cause for faults in the SP is for example, when the CPU restores the task context (including the SP) from an invalid TCB as the pointer to the TCB was corrupted. We conjecture that these crashes are due to *double bus faults* [15], triggered when an erroneous stack pointer is used. A double bus fault occurs when a bus error or an address error is triggered during the exception processing for a previous bus or address error, i.e. when data access through an erroneous stack pointer causes an exception and the call to the corresponding exception handler causes the return address to be stored through the erroneous stack pointer which

results in a new exception. Then, the processor halts and a reset is required to resume operation.

Detected errors correspond to errors detected by the various error detection mechanisms, see Table 2. Most detected errors are due to faults injected into the PC, FP or SP register. Faults injected into the PC register are detected if, (i) the CPU tries to execute an illegal or unimplemented instruction, (ii) if the instruction is legal but nonexistent memory is accessed or, (iii) a jump to an erroneous code area is made, where the error is detected by other mechanisms related to that part of the code, e.g. a jump to the task executing the matrix operation might be detected by Ada checks such as index and range checks on the matrix arrays. Faults in the SP or the FP often leads to erroneous memory accesses, i.e. the CPU tries to access data relative to an offset from the SP and FP, which can be detected by the address and bus check. Faults in the SP and FP may also result in errors similar to faults in the PC. This is because the PC is restored from the stack when returning from a subroutine and if the SP is erroneous, the PC is assigned an erroneous value. Faults injected into the data (D0-D7) and address registers (A0-A5) may be detected by the Ada access check that detects null-pointers or by the address/bus check that detects pointers accessing nonexistent memory.

The most effective error detection mechanisms were the *bus error* (~40% of the detected errors), *line 1111* (~22% of the detected errors), *illegal instruction* (~10% of the detected errors) and the *Ada access check* mechanisms (~10% of the detected errors) for the different functions.

Timing errors relate to errors affecting the timer handling in the kernel, e.g. affecting the computation of the time-out value of a periodic task, or errors directly affecting the task's execution time. For task *T1* no timing errors are produced and for task *T3*, 35 timing errors are produced. These timing errors originate from faults injected into, e.g. registers with loop variables or the program counter causing an control flow error. Producing a correct result after a control flow error ultimately depends on the workload. For example, task *T3* includes code such as "*if x>y then a else b*", which will mask errors in *x* and *y* as long as the expression *x>y* is not affected.

Timing errors also occur when injecting faults during the execution of the *task switch* or the *timer interrupt* functions. These timing errors mainly occur due to faults affecting the computation of the time-out value for which periodic tasks are suspended.

Error propagation between tasks is observed when injecting faults in task *T1* (6 occasions). These error propagations originate from faults injected into the program counter causing an erroneous jump to task *T3* where the state data of task *T3* is updated. When the return statement is executed, the program jumps back to task T1. This results in task *T1*, *T2* and task *T3* producing a wrong output. (*T2* produces a wrong result since it uses the result of *T1* as input).

6 Results for Artk68-FT

The results of the fault injection experiments conducted on the Artk68-FT kernel are shown in Table 5.

Table 5. Results of fault injection experiments on Artk68-FT

	Task T1		Task T3		Kernel functions									
	Matrix		BBW		Timer int.		Task switch		Sem. Wait		Sem. Signal		Next time	
No. injected faults	1665		2076		1805		1866		1784		1825		1159	
Correct result	1243	74.7%	1703	82.9%	1473	81.6%	1534	82.2%	1470	82.4%	1502	82.3%	951	82.1%
Tolerated	0	0.0%	262	12.6%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
Omission	0	0.0%	18	0.9%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
Wrong results	127	7.6%	0	0.0%	0	0.0%	0	0.0%	1	0.1%	0	0.0%	0	0.0%
Timing errors	0	0.0%	19	0.9%	18	1.0%	6	0.3%	0	0.0%	5	0.3%	2	0.2%
Application crashes	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
Kernel crashes	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%	0	0.0%
CPU crashes	58	3.5%	64	3.1%	46	2.5%	42	2.3%	50	2.8%	50	2.7%	35	3.0%
HW EDM	230	13.8%	10	0.5%	217	12.0%	214	11.5%	210	11.8%	201	11.0%	125	10.8%
Ada EDM	2	0.3%	0	0.0%	14	0.8%	13	0.7%	15	0.8%	26	1.4%	8	0.7%
SW EDM	5	0.1%	0	0.0%	37	2.0%	57	3.1%	38	2.1%	41	2.2%	38	3.3%

The distribution of errors for the Matrix task is similar as for the Artk68 kernel since no additional error detection is provided for this task. No wrong results are observed when injecting faults during execution of task *T3* employing TEM. In addition, TEM allowed 74% of the effective errors to be tolerated. 26% of the *tolerated errors* are detected by the double execution and 74% are detected by the CPU hardware mechanisms. About 1% of the effective errors caused an omission failure to be produced due to lack of time to execute three times and vote (16 occasions), and when three different results are produced (2 occasions). The *timing errors* relate to non detected errors that affecting the task's execution time. The *detected errors* relate to faults causing two consecutive errors being generated, which result in a fail-silent node. It should be noted that most time is spent computing the result of the task, and therefore subjected to faults the most. More faults should be injected to reveal any weaknesses also in the TEM comparison. Moreover, the result is written to memory after two matching results. To reduce the probability that the result is affected by a fault while writing it to memory, duplication or coding may be used.

One wrong result was observed when injecting faults during execution of the Semaphore wait function. This was caused by a fault in the program counter causing an erroneous jump, which affected the synchronization between task *T1* and *T2*.

As for Artk68, no *Kernel crashes* are observed and the number of CPU crashes is similar. However, the Artk68-FT kernel significantly improved the handling of tasks as no *Application crashes* are observed.

The implemented error detection mechanisms detect between 2% to 3% of the injected faults. The *Timer check* and the *Timer interrupt check* detect most timing errors when injecting faults during execution of the task switch. However, some

errors escape these mechanisms. These errors are caused by faults affecting the execution flow (4 errors), or when the number of ticks checked by the *Timer check* is correctly below the pre-computed value (*tick_check*) but nevertheless erroneous (1 error) or errors occurring just after the *Timer check* (1 error). The *Timer check* is less effective when the event timer is restarted from the timer interrupt, as the pre-computed value is not compensated for the time reset and becomes too large. Timing errors in the remaining function relates to faults affecting the execution flow. The effect of encoding the variables holding the task state and the semaphore values was not possible to evaluate in this study since the Ada compiler used mainly handles encoded variables as immediate operands, which are compared directly to an address location, e.g. "*cmpib #3, %a0@(4)*". Since the fault injection tool only support injection of faults in registers, errors in immediate operands were not created.

6.1 Overhead

Inserting a task in the timer or ready queue in the Artk68-FT kernel requires only a table look-up which is made in constant time ($O(1)$). Whereas inserting a task in the ready queue or timer queue in the Artk68 kernel involves placing the task at the correct queue location, which has a linear time complexity ($O(n)$). For example, inserting a task in the ready queue in Artk68-FT only takes a few microseconds whereas inserting a task in Artk68 ranges from 120 μs to 240 μs in our example with three tasks. On the other hand, removing a task from the ready or timer queue in Artk68 only involves removing the task first in the queue, which is made in constant time ($O(1)$) which is measured to ~120 μs. Selecting a tasks in Artk68-FT is made in linear time ($O(n)$), which in our experiment ranged from 60 μs to 120 μs for three tasks.

The time overhead for the implemented error detection mechanisms in Artk68-FT is approximately 30 % for a task switch. The time overhead for TEM is more than 100% under fault-free conditions since each task is executed twice and extra processing time is needed for handling the time redundancy. In addition, there must be a slack in the schedule to allow for a third execution of tasks that are affected by errors. The code size of the Artk68 kernel is 28.7 kB (262 kB with Ada run-time checks and elaboration code). The size of Artk68-FT with the error detection mechanisms is 35.7 kB (269 kB) and without the error detection mechanism the size is 29.8 kB (263 kB).

7 Conclusion and Future Work

The paper has presented an experimental evaluation of a small real-time kernel intended for distributed fault-tolerant real-time systems implemented on the Motorola 68340 microcontroller. The purpose is to improve the dependability by using a two-level approach for tolerating transient faults. The kernel provides mechanisms for tolerating transient faults at the node level allowing the overall reliability to be improved since the recovery time is much shorter at the node level than at the system level. Permanent faults and transient faults that cannot be handled at the node level

have to be handled at the system level. To evaluate the fault tolerance capabilities of the kernel, fault injection experiments were conducted with two versions of the kernel, one version without any error handling mechanisms, called Artk68, and one version provided with additional error handling called Artk68-FT. The Artk68 kernel employs traditional dynamic data structures, linked lists, to maintain the tasks while the use of static data structures was investigated for Artk68-FT. The experimental results show that tasks control blocks (TCB) can be permanently disconnected from the kernel data structures by faults when single linked list are used, resulting in application crashes. No such failures were observed when static data structures were used. However, additional checks are still needed in the kernel to ensure that the correct TCB is accessed. Besides these checks, additional error handling mechanisms provided by Artk68-FT, such as, checks on stack limits and timing limits allowed many other errors to be detected. To support node-level transient fault tolerance, the Artk68-FT kernel also provides protection of tasks using temporal error masking (TEM), which is able to mask transient faults by triple time-redundant execution and voting. In the experiments, no wrong results were observed when injecting faults during execution of a task employing TEM, which clearly demonstrates the effectiveness of TEM. However, a drawback with the MC68340 microcontroller is that the processor is sometimes forced to halt, particularly when faults affect the stack pointer, then excluding the possibility of tolerating faults at on the node-level.

Future work should focus on improving the error detection capabilities in the kernel. For example, the experimental results show that the implemented mechanisms for detecting timing errors in the kernel could be improved. Checks are needed to ensure that the worst-case execution time of tasks is not exceeded. In addition, mechanisms for detecting control flow errors and a memory management unit to reduce the risk that an error in one task affects other tasks or the kernel should be investigated. Such mechanisms would lead to fewer wrong results and timing errors, and thus improve the overall error detection in the system.

8 Acknowledgements

This work was supported by ARTES and the Swedish Foundation for Strategic Research (SSF). We would like to thank Arne Dahlberg and Roger Johansson at Chalmers University of Technology for providing the source code for the Artk68 kernel, and for their technical assistance.

9 References

1. Kopetz, H.: Real-Time Systems: Design Principles for Distributed Embedded Applications, Kluwer Academic Publishers, Boston (1997)
2. Rushby, J.: A Comparison of Bus Architectures for Safety-Critical Embedded Systems, Technical Report, NASA Langley Research Center, Hampton (2003)
3. Kopetz, H. Bauer, G.: The Time-Triggered Architecture, Proc. IEEE, vol. 91, no. 1 (2003)
4. Iyer, R.K., Rossetti, D.J., and Hsueh, M.C.: Measurement and Modeling of Computer Reliability as Affected by System Activity, ACM Trans. on Comp. Systems, 4(3) (1986)

5. Baumann, R.: Technology Scaling Trends and Accelerated Testing for Soft Errors in Commercial Silicon Devices, 9th IEEE On-Line Testing Symposium, Greece (2003)
6. Constantinescu, C.: Impact of Deep Submicron Technology on Dependability of VLSI Circuits, Proceedings of the International Conference on Dependable Systems and Networks, Washington D.C. (2002)
7. Aidemark, J. Vinter, J. Folkesson, P., Karlsson, J.: Experimental evaluation of time-redundant execution for a brake-by-wire application, Proceedings of the International Conference on Dependable Systems and Networks, Washington D.C. (2002)
8. Burns, A., Wellings, A., Real-Time Systems and Programming Languages, third edition, Addison-Wesley (2001)
9. Audsley, N.C., Burns, A., Davis, R.I., Tindell, K.W., and Wellings, A.J.: Fixed Priority Pre-Emptive Scheduling: An Historical Perspective, Real Time Systems, 8(2-3) (1995)
10. Engberg, A., Pettersson, P.: Asterix: A prototype of a Small-sized Real-time Kernel, Technical Report, Mälardalen Real-Time Research Center (1998)
11. Mei-Chen, H, Tsai, T.K., Iyer, R.K.: Fault Injection Techniques and Tools Computer, Volume: 30, Issue: 4 (1997)
12. Kropp, N. P., Koopman, P. J., and Siewiorek, D. P.: Automated Robustness Testing of Off-The-Shelf Software Components, Proceedings of the International Symposium on Fault Tolerant Computing, Munich, Allemagne (1998)
13. Fabre, J-C., Salles, F., Moreno, M.R., Arlat, J.: Assessment of COTS Microkernels by Fault Injection, Dependable Computing for Critical Applications 7 (1999)
14. Madeira, H., Some, R.R., Moreira, F., Costa, D., Rennels, D.: Experimental Evaluation of a COTS System for Space Applications, Proceedings of the International Conference on Dependable Systems and Networks, Washington D.C. (2002)
15. Motorola MC68340 Integrated Processor with DMA User's Manual (1992)
16. Aidemark, J. Vinter, J. Folkesson, P., Karlsson, J.: GOOFI: Generic Object-Oriented Fault Injection tool, Proceedings of the International Conference on Dependable Systems and Networks, Göteborg (2001)
17. Fault Representativeness: Dependability benchmarking technical report IST-2000-25425 (2002)
18. Course page, real-time systems, Chalmers University, http://www.ce.chalmers.se/undergraduate/D/EDA221/ (2004)
19. Damm, A.: The effectiveness of software error-detection mechanisms in real-time operating systems, International Symposium on Fault-Tolerant Computing Systems, Washington, DC, USA (1986)
20. Chevochot, P., Puaut, I.: Experimental Evaluation of the Fail-Silent Behavior of a Distributed Real-Time Run-Time Support Built from COTS Components, Proceedings of the International Conference on Dependable Systems and Networks, Göteborg (2001)
21. Arlat, J., Fabre, J-C., and Rodriguez, M.: Dependability of COTS microkernel-based systems, IEEE Transactions on Computers, vol. 51 (2002)
22. Labrosse, J. J.: MicroC/OS-II: The Real-time Kernel, second ed. Lawrence: R&D (1999)
23. Johnson B.W.: Design and Analysis of Fault-Tolerant Digital Systems, Addison-Wessley, (1989)
24. Taft et. al.: Consolidated Ada Reference Manual: Language and Standard Libraries, Springer-Verlag Berlin (2001)
25. Rabejac, C., Blanquart, J-P., Queille, J-P.: Executable Assertions and Timed Traces for On-Line Software Error Detection, Proceedings of Annual Symposium on Fault Tolerant Computing, Japan (1996)