Paper Preprint – To Appear In:
Proceedings of the 20th Euromicro RTS Conf. 2008

Exploring the Impact of Task Preemption on Dependability in Time-Triggered Embedded Systems: a Pilot Study

Michael Short, Michael J. Pont and Jianzhong Fang
Embedded Systems Laboratory, University of Leicester, Leicester, UK.
{mjs61, mjp9}@leicester.ac.uk, fang_jz@yahoo.com

Abstract

In this paper, we explore the impact of task preemption on the dependability of a single-processor embedded control system. Our particular focus in this exploratory study is on static–priority, time-triggered scheduler architectures. The study is empirical in nature and we employ a hardware-in-the-loop (HIL) testbed, representing a cruise control system for a passenger vehicle, in conjunction with fault-injection to perform the dependability comparisons. The results we have obtained suggest that the presence of preemption may have a negative influence on dependability; however further work is needed in this area before more general conclusions may be drawn.

1. Introduction

Modern control systems are almost invariably implemented using some form of digital computer system [1]. The dominance of digital systems in this field is a consequence of the low cost, increased flexibility, greater ease of use, and increased performance of digital control algorithms when compared with equivalent analogue implementations [2] [3]. As such systems are increasingly employed in applications where their correct functioning is vital, particular attention must be focused on the dependability of such systems.

Dependability in this sense covers many attributes, for example reliability, security, timeliness and schedulability [4] [5]. In this paper, we are specifically concerned with the operational dependability (i.e. the level of software fault tolerance and reliability) of control systems implemented using a single resource-constrained embedded processor, as employed in the field. As such, we assume that appropriate analysis has been undertaken during system verification to ensure the functional correctness and schedulability of the design (e.g. [5] [6] [7]).

The particular focus is on systems in which time-triggered (TT) schedulers are employed to control the release of periodic tasks, which are in turn employed to implement the control algorithm. Often, to keep the software environment as simple as possible, instead of employing a full "real-time operating system" to dispatch the tasks, some form of scheduler is employed. In this paper, we are concerned with schedulers whose task priority are assigned during the system design phase and remain static during operation; these 'fixed priority' schedulers are generally recognized as being the most suitable for designs when dependability is a key design goal [5] [8].

The simplest form of practical TT scheduler is a "cyclic executive" (e.g. [9] [10]): this has a "time-triggered co-operative" (or "time-triggered non-preemptive") architecture. Such time-triggered co-operative (TTC) architectures have been found to be a good match for a wide range of low-cost, resource-constrained applications. TTC architectures also demonstrate very low levels of task jitter [10], and – provided that an appropriate implementation is used – can maintain their low-jitter characteristics even when techniques such as dynamic voltage scaling (DVS) are employed to reduce system power consumption [11].

Although it has many useful characteristics, a simple TTC solution is not always appropriate. As Allworth has noted: "[The] main drawback with this [co-operative] approach is that while the current process is running, the system is not responsive to changes in the environment. Therefore, system processes must be extremely brief if the real-time response [of the] system is not to be impaired" [12]. We can formally express this concern by noting that if a system is being designed which must execute one or more tasks of execution time *e* and also respond within an interval *t* to external events then, in situations where $t < e$, a pure co-operative scheduler will not generally be suitable.

Time-triggered preemptive (TTP) scheduling has been proposed as an appropriate alternative for use in such circumstances [5] [6] [13]. Of the various options available, the rate monotonic algorithm has been shown by Liu and Layland to be optimal: that is - if it is possible to schedule a task set using a fixed-priority algorithm and meet all of its timing constraints – then a rate-monotonic algorithm can achieve this [6]. More specifically, it can be shown that every task can meet its deadline if the total CPU utilization is <= 69% and the following assumptions are met: (1) all tasks are periodic and independent of each other, (2) the deadline of every task is equal to its period and (3) the worst-case execution time of all tasks is known, and (4) context switching time can be ignored ([5] [6] [10] [14]). Where such assumptions can be shown to be realistic, RM can be an attractive option. In many cases, such assumptions cannot be assumed to hold; a more complete analysis (which includes CPU overheads for example) is discussed by Katcher et al. [15].

With the increased processing power and architectural design of many commercial low-cost microcontrollers, the main drawback of the purely co-operative approach is somewhat alleviated; complex tasks can be coded to have relatively short execution times, and many time-consuming operations (such as sending multiple characters over a slow serial link) can be offloaded to dedicated on-chip hardware. In situations where task execution times may still be prohibitively long, techniques have been proposed to effectively manage these situations and automate the creation of a suitable schedule (e.g. [8] [16] [17]). In light of this, system designers may have a wider range of scheduling algorithms to select from in current designs.

Numerous papers have considered the impact of scheduling on the performance and stability of digital control systems, from theoretical, empirical and simulation-based perspectives (e.g. [18] [19] [20]). In this paper, in addition to assuming that the task set for a given implementation is both correct and schedulable, we also make the following assumptions: (1) A controller *C(s)* has been designed for a plant *P(s)* such that a required performance and stability margin has been achieved; (2) The continuous controller *C(s)* has been discretised into a suitable digital controller *C(z)*; (3) The sampling rate T of the controller *C(z)* has been selected such that under worst-case jitter conditions the lower bound on the sampling rate given by the Nyquist stability criterion is not broken [2] [3] [19].

Although several studies have sought to investigate the dependability of systems designed around both TTP and TTC architectures (e.g. see Fuchs [21] and Aidemark et al. [22] for an example of each), we have not succeeded in finding previous studies in which a direct comparison (utilizing identical hardware, task specifications and fault-tolerance mechanisms) has been made of these two approaches.

It is known that – since preemptive schedulers require task context switching - they will generally have both larger CPU overheads and RAM/ROM requirements than “equivalent” co-operative schedulers (e.g. see [9] [10] [15] [16]). As a consequence, it has also been argued that the timing properties of software code in non-preemptive designs are both easier to inspect and verify than preemptive code [5] [8] [17].

Previous research has demonstrated that both the manifestation rate of transient errors and the effectiveness of transient fault detection mechanisms in an embedded system are related (in part) to the functionality and resource requirements of its software [23] [24] [25] [26]. It thus follows that as schedulers are largely implemented in software[1], different designs may therefore directly influence the fault-tolerance properties of the resulting system.

In addition, the increases in both CPU and inter-task communication overheads in a TTP design (over a TTC design) will typically result in an increase in CPU utilization when a given system specification is implemented. Previous research has investigated a link between CPU utilization and microcontroller failure rate when a microprocessor is employed to perform cyclic control actions [27] [28] [29]; it has been suggested that an increased utilization leads to an increased failure rate due to effects such as electro-migration and increased power consumption.

In this paper, we explore these issues. More specifically, our empirical study considers the use of “standard” TTC and TTP schedulers in the implementation of a Cruise Control System (CCS) for

[1] This may not be the case for all system-on-chip designs (where, for example, the operating system kernel is implemented in hardware). Such designs raise a different set of questions about reliability: these issues are not considered in this paper.

a passenger car, and explores the differences in resource requirements and dependability of the resulting designs with 'all other things being equal'. The studies are carried out using a suitable hardware-in-the-loop (HIL) testbed, which has facilities for injecting transient faults.

2. Experimental methodology

2.1 Test facility

As noted in the introduction, the study described in this paper involved the use of a HIL simulator; the principle of HIL simulation is shown in Figure 1. The simulator is currently set up to represent the dynamics of a passenger vehicle in real-time, iterated at a rate of 1 kHz. The nature of the testbed itself, and the dynamic models used to represent the vehicle have been described elsewhere [30] [31]; we provide only a brief summary here.

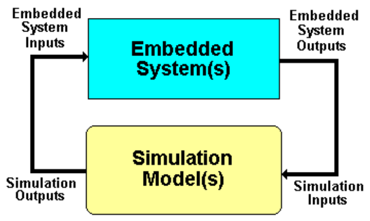

Figure 1. HIL simulation principle

Although the dynamic model of the vehicle is non-linear, it can be approximated in the operating range of the CCS by the following transfer function:

$$\frac{v(s)}{F(s)} = \frac{0.02}{15s+1} \tag{1}$$

... where *v(s)* is the velocity of the vehicle in meters per second, and *F* is the accelerating force (which is dependant on the accelerator setting, engine RPM and wheel slip conditions).

The main requirement of the CCS, which is implemented by the embedded system under test, is to provide the vehicle driver with an option of maintaining the vehicle at a desired speed without further intervention, by automatically controlling the vehicle throttle setting. It performs this function by measuring the current vehicle speed from a sensor and performing a PID calculation to determine the throttle setting. The classical form of the PID algorithm employed in this study is as follows:

$$u(t) = K_c e(t) + \frac{K_c}{T_i}\int_0^t e(t)dt + K_c T_d \frac{de(t)}{dt} \tag{2}$$

... where *u(t)* is the commanded throttle setting, *e(t)* is the error between reference (desired) speed and actual (measured) speed, and K_c, K_i and K_d are the system gains, chosen to give the desired closed loop performance [1] [2] [3]. Additionally, the module is required to indicate the current speed of the vehicle and the status of the control system to the driver, via a serial interface to an LCD. It also must interface to a number of switches to receive commands from the driver ("CCS enable", "CCS disable", "Speed up", etc).

The microprocessor employed in this study to implement each version of the CCS was the 16-bit Infineon C167 with a 20 MHz clock speed [34]. Such a device is representative of the type of embedded system currently found in many passenger vehicles. Digital signals from this device are interfaced into the simulation PC via standard parallel port interfaces; analog (sensor) signals are synthesized by the PC using low-cost AD7394 DAC chips from Analog Devices.

Overall, the CCS testbed summarized here was chosen as a representative system as it can be considered to be a critical application [32], and previous studies have shown that transient effects and processor faults (amongst other things) can be a major contributory cause to potential dangerous system failures [33].

2.2 Fault injection

In order to evaluate the dependability of each system under test, a reliable, non-intrusive and high performance fault injection protocol was created. The protocol itself operates via the PC serial port and on-chip UART of the microcontroller under test, and is based on the well-understood 'bit-flip' or 'upset' model. Such an approach has been shown to be representative of a wide range of transient faults in embedded systems [26].

The protocol operates as follows. Three control bytes are sent by the PC, invoking a high priority interrupt in the microcontroller. Fault injection is achieved by the use of pointer indirection to achieve the bit flips; bit flips in the C167 internal RAM (IRAM) areas can corrupt the system stack, registers, special function registers (SFRs) and program counter. Bit flips in the external RAM (XRAM) areas can corrupt the user stack and also the task data areas. To ensure a fair comparison between the two systems, for each fault injected in this study a random bit to flip was selected in a random memory address location from a 4.5 KB area of IRAM or a 4.5 KB area of XRAM; thus implementing a wide variety of data, control flow and CPU / peripheral configuration errors.

2.3 Overall methodology

In each of the experiments performed in this paper, the vehicle cruise control was initially enabled at 50 MPH (80.5 KPH). The 'driver' then commanded periodic speed changes from 50 MPH to 40 MPH (64.4 KPH), and vice versa, every 10 seconds. One second prior to this commanded speed change, a fault was injected into the system under test by the simulation PC. After the injection of each fault, the resulting system behavior was automatically classified using a simple model-based performance monitor to gauge the operation of the system in real-time[2]. The performance monitor compares the real system behavior with the desired system behavior, as shown in Figure 2, to detect deviations from the specification that are indicative of a system failure (e.g. sluggish / oscillatory performance, out of range or 'stuck at' errors). The performance monitor is an adapted version of the design presented by Li et al. [35]. After each resulting fault has been classified, the results are logged into a text file by the PC for later analysis.

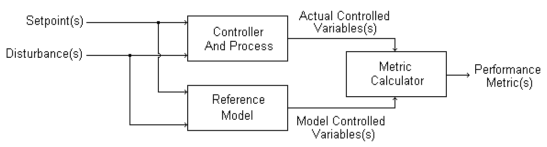

Figure 2. Performance monitor

This test strategy was employed to allow for a maximum possible coverage of critical code (sampling, control, actuation, response to driver commands) following the injection of the fault. It must be noted that there was no explicit synchronization when injecting faults; they could occur at any point in the embedded system's execution, and could not be 'blocked' in any way. In each of the tests described in this paper, the experimentation was allowed to run autonomously until 100,000 faults had been injected into the system under test, the resulting failure modes classified by the monitor, and logged into the text file. The overall testing strategy that was employed is shown schematically in Figure 3.

[2] To eliminate any 'probe effects', the performance monitor was implemented entirely on the PC.

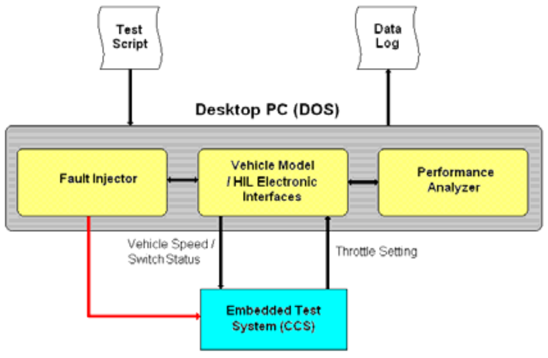
Test Script
Data Log
Desktop PC (DOS)
Fault Injector
Vehicle Model / HIL Electronic Interfaces
Performance Analyzer
Vehicle Speed / Switch Status
Throttle Setting
Embedded Test System (CCS)

Figure 3. Overall experimental methodology

3. Scheduler and task designs

3.1 TTC system

To implement the system, the first scheduler used in this study was a form of cyclic executive. The specific implementation used was based on that described previously by Pont [16], with very minor changes made to match the C167 processor employed in this study.

Please note that - because of the co-operative nature of this scheduler - no locking mechanisms were required, and inter-task communication was by means of "global" variables.

3.2 TTP system

The time-triggered preemptive (TTP) scheduler used in this study was a fully preemptive design, which was used to implement a rate-monotonic (RM) schedule of the task set described in the following section. The fast context switching mechanism of the C167 was employed in the implementation; full details of the scheduler design are documented by Fang [36].

Please note that – as discussed in the introduction - a "pure" RM schedule assumes that all tasks are independent: this was not the case in our system (indeed, it is very rarely the case in any practical design) and our scheduler implementation could therefore only provide an approximation of the RM theory. To avoid conflicts, locking mechanisms were implemented (to protect shared hardware resources). These locking mechanisms approximated the "Priority Ceiling Protocol" [37]. In "PCP", a ceiling priority is defined as the maximum priority of those tasks that take part in a priority comparison. When using PCP, the intention is to ensure that the task which is currently using the resource completes as quickly as possible.

In addition to the locking mechanism, the RM scheduler also requires a mechanism for transmitting data between tasks (the possibility of task preemption means that we cannot simply use global variables for this purpose, as we can with the co-operative design). Various mechanisms can be used for inter-task data transfers in such designs: for example, we could use the lock mechanism and global variables. A more popular technique is a message queue (e.g. see Simon [38]): such an approach was employed in this study.

3.3 Task software

Six periodic software tasks were created to implement the control system. A sensor task (T1) sampled the vehicle speed (as a voltage) through an analog-to-digital (ADC) port, re-scaled the voltage to the required speed range and performed range and range-rate sanity checks. The control task (T2) performed the PID calculation; a digitized version of (2):

$$u_j = K_P e_j + K_i \sum_{m=0}^{j} e_m + K_d (e_j - e_{j-1}) \quad (3)$$

… where u_j is the commanded throttle setting at sample j, and e_j is the error at sample j. The three system gains were chosen (manually) to give the desired closed loop performance; a critically damped 95% settling-time of approximately 6 seconds. The actuation task (T3) performed a further sanity check on the resulting throttle command and translated this output to a parallel port of the C167, to control a throttle servo actuator. These three tasks were scheduled to execute sequentially with a period of 20 ms, giving an overall sampling rate of 50 Hz.

A status-update task (T4) was executed once every 100 ms. This task monitored and "de-bounced" the "cruise enable" and "resume" switches. It also monitored switches on both the accelerator and brake pedals: if either of the pedal switches was depressed, the cruise control was disengaged. In addition, a display update task (T5) was also executed every 100 ms and sent a string of display characters to the LCD (serial protocol). Note that in the co-operative scheduler, we broke the LCD update task evenly up into smaller segments (as discussed in Pont [16]).

The final system task was the safety task (T6), also executed every 100 ms. This task serviced the watchdog timer, and also verified the system configuration (such as timer reload value, I/O configuration and interrupt settings).

These tasks and their periods T and duration D (expressed in ms) are summarized in Table I.

Table I. Task set summary

Name	Task ID	T (ms)	D (ms)
Sensor	T1	20	0.5
Control	T2	20	1.2
Actuator	T3	20	0.5
Status	T4	100	9
LCD – P	T5	100	25
LCD – C	T5	20	5
Safety	T6	100	0.1

Prior to experimentation both software implementations were verified using bounded model checking[3] techniques to ensure that run-time failures were not due to simple coding errors.

3.4 Timer tick interval

In a system based on periodic scheduler "ticks" (timer interrupts), the tick interval must be set according to the application requirements. If the tick interval is too long, the system will be unable to respond at an appropriate rate and will not – for example – be able to meet data-sampling requirements. If the tick interval is too short, the scheduling load will be unnecessarily high: this may increase both jitter levels and system power consumption.

From an analysis of the schedule, for the co-operative system we chose a timer interrupt rate of 10 ms, as this is the largest common divisor of the tasks that is also greater than the longest task duration.

[3] Please see http://www-2.cs.cmu.edu/~modelcheck/cbmc for further details of the model checker that was employed.

From Katcher et al. [15], we calculated timer bounds for the tick rate T_{TIC} of the preemptive system to be between 1.14 ms < T_{TIC} < 18.25 ms. The actual value taken was thus the integer mid point of 10 ms, identical to the co-operative system.

3.5 Transient protection mechanism

In order to increase realism of the study, a number of identical hardware and software based mechanisms were utilized (in both TTC and TTP implementations) in order to detect and correct transient errors. These mechanisms included the use of a 200 ms watchdog timer, duplex implementation of critical data with comparison, sanity checks of control signals, and a task overrun detection mechanism. The on-chip exception traps of the C167 processor were also employed, allowing detection of: stack overflow; stack underflow; illegal operands; illegal word access; protected instruction faults; and, illegal bus access.

The unused areas of FLASH memory and RAM in each design were filled with illegal operands to provide an addition means of detecting control-flow errors. On activation of any of these traps, a full system reset was forced. Finally, on system boot-up/reset, the system performed the following software-based self-tests [39]:

1. Internal RAM/register/stack validation.
2. External RAM validation.
3. ROM checksum.
4. Peripheral test (e.g. ports, timer).

This approach to fault-tolerance is illustrated in Figure 4.

4. Results and discussion

In this section we discuss the results that were obtained in the study; we begin with a comparison of the required memory resources and CPU utilization of each implementation.

4.1 CPU and memory requirements

As mentioned in the introduction, TTP schedulers are expected to have larger CPU overheads and memory requirements than equivalent TTC designs. Table 2 shows the requirements for each of the designs employed in this study. The CPU utilization *U* and scheduler overhead *W* were both experimentally determined (by directly measuring both the idle time of the microcontroller and the time spend executing scheduler code) over several thousand hyper periods of the task set, during (normal) fault-free operation. Table II also shows the percentage increase between the TTC and TTP designs.

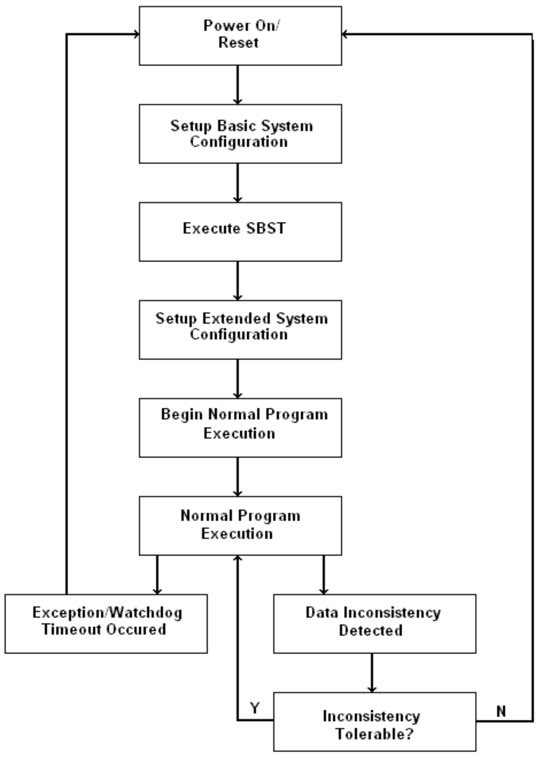
Power On/ Reset
Setup Basic System Configuration
Execute SBST
Setup Extended System Configuration
Begin Normal Program Execution
Normal Program Execution
Exception/Watchdog Timeout Occured
Data Inconsistency Detected
Inconsistency Tolerable?
Y
N

Figure 4. Transient protection mechanisms

Table II. System resource comparison

System	U (%)	W (μs)	RAM (B)	ROM (B)
TTC	45.36	39.5	650	4,774
TTP	51.44	262.2	4,882	5,030
% Increase	13.4	563.8	651.1	5.4

4.2 Fault injection results

As mentioned, in this study we were concerned with operational dependability of the systems under test. After the injection of each fault, the resulting failure behavior was therefore categorized as one of the following:

1. Effect-less: the fault was either not activated, or was tolerated by the employed fault mechanisms; it did not cause a measurable deviation of performance from the system specification.

2. Failure: the fault was neither detected nor corrected and resulted in a permanent deviation of performance from the system specification.

Based on this classification, the probability of failure PF following a transient fault was estimated. The results we obtained for each system are summarized in Table III.

Table III. Fault injection comparison

System	Faults Injected	Effect Less	Failures	P_F
TTC	100,000	99942	58	5.8×10^{-4}
TTP	100,000	99755	245	2.45×10^{-3}

4.3 Discussion

Based on the results shown in the previous two sections, it can be seen that for the two systems, exhibiting identical functionality, several noticeable differences can be observed. Firstly, as expected all measures of system resources were larger for the TTP system. The most significant differences being the increase in RAM requirements (651.1 %) and scheduler overheads (563.8 %), followed by an increased CPU utilization (13.4 %). The smallest increase was in the system ROM requirements, with a 5.4 % increase in size.

Secondly, as the fault injection results demonstrate, despite both systems exhibiting relatively good transient error recovery properties a significant difference in the failure behavior of the two systems was observed. A 322.4 % increase in the number of recorded dangerous failures was observed in the TTP system over the TTC. Thus as can be seen from the table, there is a measurably greater risk that the TTP system will exhibit dangerous failure behavior following a transient disturbance. Given that the main difference between the two system implementations was the choice of scheduler, we hypothesize that the increased complexity, overheads and resource usage in the TTP system had a direct influence on its transient error failure rate.

Based on these results, it is possible to calculate an expected mean-time-to-dangerous-failure (MTTDF) for each system implementation. Taking the probability of a transient bit flip to be approximately 10^{-9} / bit / hour for a ground based mobile installation [40], the probability of a transient error λ_T occurring in the 9KB area of memory that was considered in this study is therefore approximately 7.4×10^{-5}.

Previous works suggest a linear relationship between CPU failure rate λ_{CPU} and utilization; this relationship can be expressed as follows [27] [28] [29]:

$$\lambda_{CPU} = \lambda_{MIN} + U \cdot \Delta_U \tag{4}$$

… where ΔU is the change in failure rate due to power consumption between 0 and 100% CPU utilization. Thus the failure rates per 10^{-6} hours for the C167 CMOS technology microcontroller with 111 pins, a 128K Flash EEPROM and 256K SRAM can be calculated as shown in Table IV [41].

Table IV. Component failure rates

Component	Failure Rate
CPU (λMin)	0.69393
CPU (ΔU)	0.00067
EEPROM (λE)	0.16490
RAM (λR)	0.24320

Considering that the failure of the CPU or memory devices will result in an unpredictable failure that will cause a significant deviation of system performance from the specification, the dangerous failure rate λ_{SYS} for each system may then be calculated as follows:

$$\lambda_{SYS} = \lambda_{CPU} + \lambda_E + \lambda_R + \left(\lambda_T \cdot P_f\right) \tag{5}$$

Applying (4) and (5) to the results we have obtained, the failure rates for the TTP and TTC systems can then be calculated as shown:

$$\begin{aligned} \lambda_{TTC} &= 1.175 \times 10^{-6} \\ \lambda_{TTP} &= 1.317 \times 10^{-6} \end{aligned} \tag{6}$$

From this, it can be seen that in this study, with ‘all other things being equal’, the TTC implementation of the CCS was significantly more dependable than the TTP design. This is further illustrated when comparing the MTTDF of each system: the MTTDF of the TTC system is 97.14 years, and the MTTDF of the TTP system is 86.67 years. Although both of these values may be deemed acceptable for a CCS system, it is clear from these results that the use of task preemption in this experimental study has had a measurable impact on the dependability of the resulting system implementation, with a 12.1 % increase in the expected MTTDF in the TTC design.

5.0 Conclusions

In this paper, we have explored the impact of preemption on the operational dependability of a single-processor embedded control system, implemented using a static time-triggered scheduler. The study was empirical in nature and we employed a hardware-in-the-loop (HIL) testbed, combined with fault-injection experimentation, to perform the comparisons. The results obtained in this initial study suggest that preemption *may* have a measurable effect on dependability and fault-tolerance in embedded systems; however at this early stage it cannot be said that these results will generalize further.

Further work is required to investigate if the results found here will generalize to other processor platforms, scheduler implementations and application areas. Additionally it is unclear if more advanced fault detection schemes than those employed in this paper will minimize the differences we have described. One other point to note is that this paper has not directly considered the effects of corruptions in the program ROM in the systems under test; this may well have an additional influence on the obtained results.

Further work in this area will consider these effects, and will also consider additional case studies in which the effects of preemption *rate* may also be investigated. It is also planned to investigate preemption effects on dependability in dynamic schedulers (primarily EDF), and also its effects in multi-processor, distributed embedded systems. It is hoped that such studies may shed further light on the important issues raised in this paper.

6.0 Acknowledgements

The project described in this paper was supported by the Leverhulme Trust (Grant F/00212/D).

7. References

[1] Kilian, C.T. Modern control technology: components and systems. Delmar Thomson Learning, 2000. ISBN: 076682358X.

[2] Virk, G.S. Digital computer control systems. McGraw-Hill, 1991. ISBN: 0070675120.

[3] Astrom, K. and Wittenmark, B. Computer Controlled Systems: Theory And Design. Prentice Hall, 1997. ISBN: 0133148998.

[4] Storey, N. Safety Critical Computer Systems. Addison Wesley Publishing, 1996.

[5] Bate, I.J. Scheduling and Timing Analysis for Safety Critical Real-Time Systems. PhD Dissertation, University of York, U.K., November 1998.

[6] Liu and, C. L. and Layland, J. W. Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of the ACM 20(1), 1973.

[7] Burns, A. and Lin, T.M. An engineering process for the verification of real-time systems. Formal Aspects of Computing, Vol. 19, pp. 111-136, 2007.

[8] Xu, J. On Inspection and Verification of Software with Timing Requirements. IEEE Transactions on Software Engineering, Vol. 29, No. 8, pp. 705-720, 2003.

[9] Baker, T. P. and Shaw, A. The cyclic executive model and Ada. Real-Time Systems, Vol. 1, No. 1, pp. 7-25, 1989.

[10] Locke, C. D. Software architecture for hard real-time applications: Cyclic executives vs. fixed priority executives. Real-Time Systems, 4(1): 37-52, 1992.

[11] Phatrapornnant, T. and Pont, M. J. Reducing jitter in embedded systems employing a time-triggered software architecture and dynamic voltage scaling. IEEE Transactions on Computers, 55 (2): 113-124, 2006.

[12] Allworth, S. Introduction to real-time software design. Springer-Verlag, 1981.

[13] Audsley, N., Burns, A., Richardson, M., Tindell, K., and Wellings, A. Applying new scheduling theory to static priority pre-emptive scheduling. Software Engineering Journal 8(5), pp. 284-292 September 1993.

[14] Buttazzo, G., Marinoni, M., and Guidi, G. Energy-aware strategies in real-time systems for autonomous robots. In: Proc. of the 19th International Symposium on Computer and Information Sciences (ISCIS 2004), Turkey, 2004.

[15] Katcher, D. I., Arakawa, H., and Strosnider, J. K. Engineering and analysis of fixed-priority schedulers. IEEE Trans. on Software Engineering, Vol. 19, No. 9, pp. 920-934, 1993.

[16] Pont, M. J. Patterns for time-triggered embedded systems: Building reliable applications with the 8051 family of microcontrollers. ACM Press / Addison-Wesley, 2001. ISBN: 0-201-331381.

[17] Xu, J. and Parnas, D.L. Fixed Priority Scheduling versus Pre-Run-Time Scheduling. Real-Time Systems, Vol. 18, pp. 7-23, 2000.

[18] Palopoli, L., Pinello, C., Bicchi, A. and Sangiovanni-Vincentelli, A. Maximizing the Stability Radius of a Set of Systems Under Real-Time Scheduling Constraints. IEEE Transactions on Automatic Control, Vol. 50, No. 11, pp. 1790-1795, 2005.

[19] Bate, I., McDermid, M. and Nightingale, P. Establishing timing requirements for control loops in real-time systems. Microprocessors and Microsystems, Vol. 27, pp. 159-169, 2003.

[20] Fang, J. and Pont, M.J. Exploring the links between software architecture and PID parameters in embedded control systems. In: Proceedings of the 6th UKACC Control Conference, Glasgow, Scotland, 30 August to 1 September, 2006.

[21] Fuchs, E. An Evaluation of the Error Detection Mechanisms in MARS Using Software-Implemented Fault Injection. In: Proc. of the European Dependable Computing Conference, Springer-Verlag, Lecture Notes in Computer Science, Volume 1150, pp. 73-90, 1996.

[22] Aidemark, J., Folkesson, P. and Karlsson, J. Experimental Dependability Evaluation of the Artk68-FT Real-time Kernel. In: Proc. of the International Conference on Real-Time and Embedded Computer Systems and Applications, Göteborg, Sweden, August 2004.

[23] Rajabzadeh, A. and Miremadi, S.G. Transient detection in COTS processors using software approach. Microelectronics Reliability, Vol. 46, pp. 124-133, 2006.

[24] Benso, A., di Carlo, S., di Natale, G., Prinetto, P. and Tagliaferri, L. Control-Flow Checking Via Regular Expressions. In: Proc. IEEE Asian Test Symposium, pp. 299-303, 2001.

[25] Elder J.H. A Method for Characterising a Microprocessor's Vulnerability to SEU. IEEE Trans. on Nuclear Science, Vol. 35, No. 6, December 1988.

[26] Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.-C., Laprie, J.-C., Martins, E. and Powell, D. Fault Injection for Dependability Validation—A Methodology and Some Applications. IEEE Trans. Software Eng., vol. 16, no. 2, pp. 166-182, Feb. 1990.

[27] Ting, Y., Shan, F.M., Lu, W.B. and Chen, C.H. Implementation and evaluation of failsafe computer-controlled systems. Computers & Industrial Engineering, Vol. 42, pp. 401-415, 2002.

[28] Kapur, K.C. and Lamberson, L.R. Reliability in engineering design. New York: Wiley & Sons, 1977.

[29] Krishnan, R.V. The Impact of Workload On The Dependability Of Microprocessors Used In Control Applications. MSC Thesis, University of Illinois at Urbana-Champaign, USA, 1996.

[30] Ayavoo, D., Pont, M. J., Fang, J., Short, M., and Parker, S. A 'Hardware-in-the Loop' testbed representing the operation of a cruise-control system in a passenger car. In: Proceedings of the Second UK Embedded Forum (Birmingham, UK, October 2005), pp. 60-90, 2005. Published by University of Newcastle upon Tyne [ISBN: 0-7017-0191-9].

[31] Short, M. J. and Pont, M. J. Hardware in the loop simulation of embedded automotive control systems. In: Proceedings of the 8th IEEE International Conference on Intelligent Transportation Systems (IEEE ITSC 2005) held in Vienna, Austria, pp. 226-231, 2005.

[32] Castelli, J., Nash, C., Ditlow, C. and Pecht, M. Sudden acceleration – the myth of driver error. University of Maryland, CALCE EPSC Press, ISBN 0-9707174-5-8.

[33] Mauser, H. and Thurner, E. Electronic Throttle Control – A Dependability Case Study. Journal of Universal Computer Science, Vol. 5, No. 10, pp. 730 – 741, 1999.

[34] Infineon Technology AG. C167CS Derivatives: 16-bit single-chip microcontroller. User's Manual, V2.0, 2000.

[35] Li, Q., Whiteley, J.R., and Rhinehart, R.R. A relative performance monitor for process controllers. Int. Journal of Adaptive Control and Signal Processing, Vol. 17, pp. 685-708, 2003.

[36] Fang, J. The design of a pre-emptive scheduler for the C167 Microcontroller. Technical Report ESL 06/01, University of Leicester, 2006.

[37] Sha, L., Rajkumar, R., and Lehoczky, J. P. Priority Inheritance Protocol: An approach to real-time synchronization. IEEE Trans. On Computers 39: 1175-1185, 1990.

[38] Simon, D.E. An embedded software primer. Addison-Wesley, 1999. ISBN: 0-201-61569X.

[39] Sosnowski, J. Software-based self-testing of microprocessors. Journal of Systems Architecture, Vol. 52, pp. 257-271, 2006.

[40] Normand, E. Single Event Effects in Avionics. IEEE Trans. on Nuclear Science, Vol. 43, No. 2, 1996.

[41] MIL-HDBK-217F. Military Handbook - Reliability Prediction of Electronic Equipment. Department of Defence, Washington DC, 1990.