A Real-Time Gracefully Degrading Avionics System for Unmanned Aerial Vehicles

Belal H. Sababha
Computer Engineering Department
Princess Sumaya University for Technology
Amman, Jordan
b.sababha@psut.edu.jo

Osamah A. Rawashdeh and Waseem A. Sa'deh
Electrical and Computer Engineering Department
Oakland University
Rochester, Michigan, USA
{rawashd2, wasadeh}@oakland.edu

ABSTARACT

Graceful degradation is an approach for developing dependable safety-critical embedded applications, where redundant active or standby resources are used to cope with faults through system reconfiguration at run-time. Compared to traditional hardware and software redundancy, it is a promising technique that may achieve dependability with a significant reduction in cost, size, weight, and power requirements. Checkpointing protocols, which are necessary components of degrading systems, support task migration through state preservation. They allow real-time embedded systems to recover from any failure by restarting from the last well-defined and consistent state, thus preserving the progress of computations that have been achieved. This paper demonstrates and applies the graceful degradation concept to achieve fault tolerance in an unmanned aerial vehicle (UAV) real-time embedded system. A checkpointing protocol is used to reserve the state of the avionics of the UAV system. Faults were injected during run-time causing one of the system's stability critical control tasks to fail. The system was able successfully to recover by restarting the affected critical task(s) on a different processor with last valid consistent state(s). This paper presents the architecture, fault injection scheme, and the results of the tests performed, which demonstrate the viability of graceful degradation in our tested UAV.

I. INTRODUCTION

Modern embedded systems are becoming increasingly distributed and include an interconnecting network to facilitate collaboration of a set of processors to achieve a common goal [1]. In safety-critical applications, the correct operation is vital, requiring the use of fault tolerant techniques in applications. Fault tolerance is the ability of a system to continue operation in presence of hardware and software faults [2]-[5]. It is typically achieved through redundancy in hardware and software to enable fault detection and recovery [2]-[4]. Explicit redundancy for a non-trivial system can however be complex and costly in terms of size, weight, price, and power consumption. Graceful degradation is a promising new concept to achieving capable fault tolerance at lower cost [6]-[11]. In presence of a fault, the system switches to an operating mode that uses remaining available resources to compensate for the failure or to a mode where the affected system functionality is dropped. Reconfiguration necessitates the preservation of the recent history for every software task that is responsible of certain functionality in order to be able to restart the task later from the point that it failed at. The recent history of all parameters of a certain task is called the state of that task. Saving the state of a task to stable storage is referred to as checkpointing the state of that task [12]. When a fault is detected, the stored information is used to restart a process at an acceptable intermediate point to reduce lost computation time [12].

In previous work, several checkpointing protocols were implemented and evaluated on a distributed embedded system for graceful degradation purposes [13]-[15]. A resource limited real-time distributed embedded test-bed system was constructed [16]. The system utilizes a Controller Area Network (CAN) for communication between processing elements (PEs). Two studies were performed to evaluate the checkpointing protocols. The first study included a simulated application executing on the PE test-bed, where tasks periods, message destinations, and message frequencies are set randomly. In the second study, the protocols were applied to a feedback-control system for an unmanned aerial vehicle (UAV). The periodicity property of embedded systems was also studied.

This paper demonstrates the implementation and evaluation of a fully gracefully degrading avionics system. The avionics system of a quadrotor unmanned aerial vehicle is considered. All tasks of the attitude stability portion of the system as well as the tasks delivering the feedback information are checkpointed. A checkpointing protocol is used to checkpoint all state related parameters to a safe stable storage. A checkpointing protocol called BCS [17] was chosen due to the relatively low amount of overhead it induces compared to other checkpointing protocols. A fault injection method is implemented to kill any computation task in the system. A system manager that is responsible for fault detection, rollback and recovery is implemented on a separate MCU.

The paper is organized as follows: The next section overviews the test-bed vehicle. Section III discusses the avionics system design and implementation. The gracefully degrading avionics system and the recovery manager are overviewed in Sections IV and V respectively. Section VI shows the performance results. Finally, the paper is concluded in Section VII.

II. THE TEST-BED

The quadrotor [18] test-bed system, shown in Figure 1, consists of two subsystems. The first is a ground station that

978-1-4673-2792-3/12/$31.00 ©2012 IEEE

Figure 1. Oakland University quadrotor.

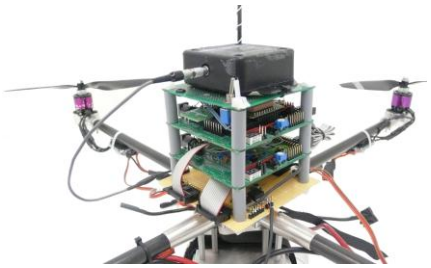

Figure 2. The Quadrotor avionics block.

is responsible for flight control, data processing, reconfiguration control, and run-time monitoring. The other subsystem is the aerial vehicle which carries the avionics and payload systems. The two subsystems are connected by three wireless links: a 1.3 GHz link for video down-streaming, a 75 MHz traditional R/C radio for manual flight control, and a bidirectional 2.4 GHz ZigBee link for telemetry and reconfiguration control.

The body of the quadrotor consists of a magnesium hub joining four carbon fiber arms. Mounted at the end of each arm is a magnesium motor mount that holds a brushless motor and propeller assembly. The avionics system's hardware is shown in Figure 2. In addition to the processors, other hardware components include an IMU, altimeter, and a modular GPS unit for attitude, altitude, and position estimation, respectively. The ground station includes a navigation algorithm, target recognition software, and the graphical user interface (GUI) for parameter monitoring and reconfiguration purposes.

More details on this custom quadrotor system design and implementation can be found in previously published papers [18],[19]. The next section describes the avionics system in more detail.

III. AVIONICS SYSTEM

The avionics system is a triple-processor setup consisting of a telemetry processor, a sensor-actuator interface processor and a control processor. All three processors are Freescale HCS12 microcontrollers running µC/OS-II™, the real-time operating system (RTOS). The three HCS12 MCUs execute different UAV tasks. The RTOS, µC/OS-II™, manages the tasks on each of the three MCUs. All MCUs are connected to a CAN bus. A forth dedicated MCU collects checkpoints written by other MCUs from the CAN bus and stores them to stable storage. Figure 3 shows the block diagram of the system.

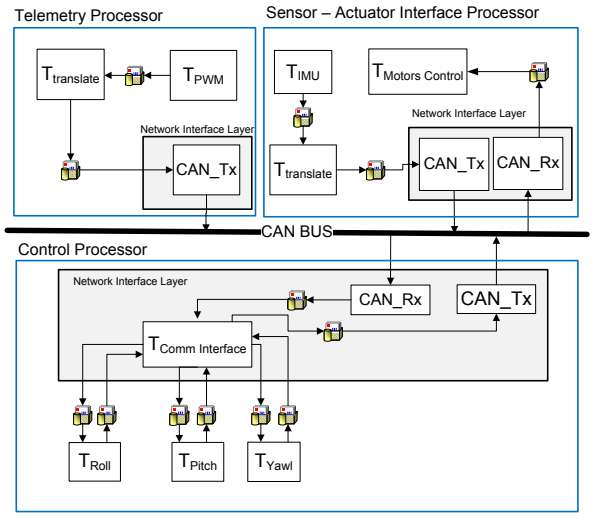

Figure 3: UAV Avionics System Block Diagram

The control processor is responsible for vehicle stabilization and navigation. It achieves these functions by executing software tasks implementing the Proportional-Integral-Differential (PID) control loops. In this project, four PID control loops are used to stabilize and control the vehicle as shown in Figure 4. The motor speed is controlled by the PID control loops according to the throttle, pitch, roll, and yaw values received by the IMU, GPS unit, and altimeter. In the PID control loops, the current errors are calibrated with PID gain constants Kp, Ki, and Kd to generate the proper motor adjusting values where Kp, Ki, and Kd are the proportional, integral, and differential gain constants respectively. The PID gains are found experimentally, and are wirelessly reconfigured by the reconfiguration host on the ground station to allow in-flight tuning.

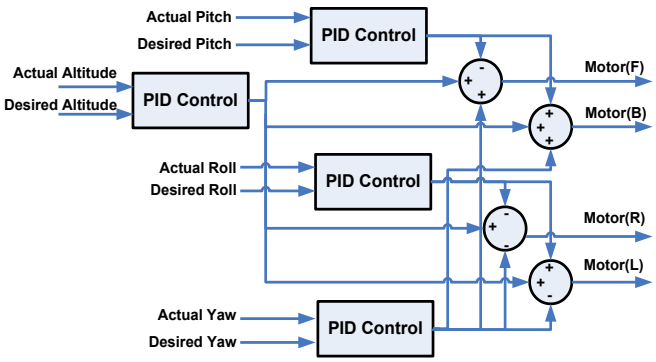

Figure 4. Control system diagram.

The main tasks running on the control processor are: T_{Comm}, T_{Roll}, T_{Pitch}, T_{Yaw}. The T_{Comm} task forwards the current and desired attitude angles to the other three tasks through OS provided Mailboxes. T_{Roll}, T_{Pitch}, and T_{Yaw} implement the roll, pitch, and yaw stability PID controllers respectively.

The telemetry processor is responsible for executing the T_{PWM} task that captures PWM from the RC receiver and translates it into desired commands. The Sensor-Actuator Interface processor is responsible for executing the tasks that do the data collection from the sensors, as well as forwarding the control commands to the motor speed controllers.

IV. The Gracefully Degrading Avionics System

The attitude stability tasks of the avionics system are the three PID control tasks described in Section III. Specifically the Roll, Pitch and Yaw PID control tasks. The control feedback task is the IMU task. The three PID tasks as well as the Motors Control ($T_{Motors\_Control}$) Task were chosen to be checkpointed because the computations they perform depend on historical data from previous computations. Figure 5 shows the data communication pattern that is continuously repeated during the process of performing attitude stability control of the unmanned vehicle.

The data checkpointed by each of the attitude stability PID tasks is composed of the accumulative error used in the integral (I) term of the PID controller as well as the previous error used by the differential (D) term of the PID controller for all three PID control tasks. The Motors Control ($T_{Motors\_Control}$) task, checkpoints the current corrections that are used to adjust the speed of the four motors propelling the quadrotor.

Each infinite loop in the Roll PID, Pitch PID, Yaw PID and Motors Control tasks performs the following main functionalities:

i. Transmits a Heart Beat message indicating that it is a live and performing its' functionality.

ii. Reads its' mailboxes for any available messages.

iii. Extracts the BCS induced control information (checkpoint index) as well as the application data.

iv. Checks the BCS forced checkpoint condition depending on the received index and decides to take a forced checkpoint or not.

v. Delivers the application data and performs the application specific computations and prepares the output that is going to be sent to other tasks in the distributed network.

vi. Piggyback the checkpointing index over all application messages that are going to be sent out.

vii. Send outgoing messages.

viii. Depending on the required checkpoint frequency, take a Local checkpoint.

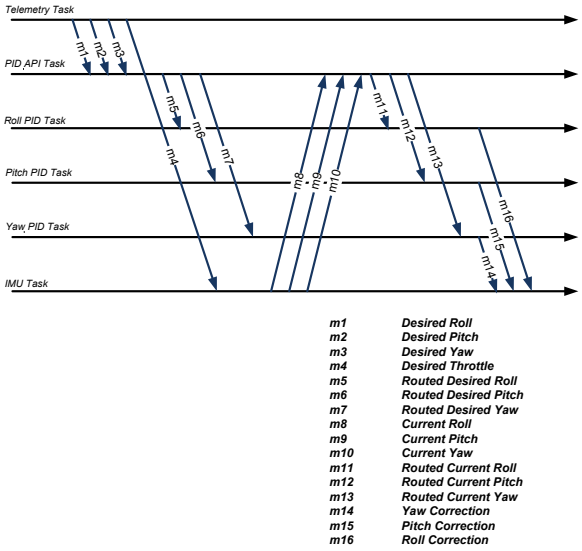

Figure 5: Attitude Stability Control Communication Pattern

V. Recovery Manager

As indicated in Section IV, each checkpointing task has a unique heart beat message broadcasted through the CAN network as long as it is functioning correctly. To simulate faulty tasks, a fault injection method is implemented through an external MCU connected to the CAN bus. With the aid of the fault injection MCU, it is possible to send a command to the RTOS running on any other MCU in the network to kill any specific task executed by that MCU. The recovery manager is also capable of sending a recover command to the RTOS running on any other MCU in the network. The recover command asks the RTOS to recreate a task and

initialize it to a certain state (recovery state). The recovery state is sent in combination with the recover command. The recovery manager is composed of several smaller tasks that maintain the following responsibilities:

i. Receive all checkpoints sent by all checkpointing tasks and save them to stable storage.

ii. Live online computation of a recovery line (Consistent global checkpoint).

iii. Perform fault detection by monitoring the heart beats of all checkpointing tasks.

iv. In presence of a faulty checkpointing task, recover all tasks in the distributed network by restarting them from the recovered state.

The data structures used to store the checkpoints are shown in Figure 6, where:

- rollTable : is an array of checkpoints received from the Roll PID task.
- pitchTable: is an array of checkpoints received from the Pitch PID task.
- yawTable: is an array of checkpoints received from the Yaw PID task.
- r: is a pointer to the rollTable's next available entry to store a newly received checkpoint.
- p: is a pointer to the pitchTable's next available entry to store a newly received checkpoint.
- y: is a pointer to the yawTable's next available entry to store a newly received checkpoint.
- i: is a pointer to the imuTable's next available entry to store a newly received checkpoint.

Figure 7, is a flowchart showing the flow for both the application tasks to be checkpointed as well as the recovery algorithm performed by the recovery manager.

VI. Performance Results

The stable memory available for checkpointing storage is around 1 Kbyte (exactly 1320 Bytes). From an application point of view, to maintain a stable flight, the acceptable execution rates for the three attitude stability PID control loops are as follows:

i. Roll PID: 20 ms (i.e. 50 Hz)

ii. Pitch PID: 20 ms (i.e. 50 Hz)

iii. Yaw PID: 60 ms (i.e. 16.7 Hz)

An experiment was conducted by injecting a fault into the roll PID control task and measuring the time to recover the system. This experiment was performed on the avionics system with a varied checkpoint frequency for all checkpointing tasks from 1 local checkpoint per execution loop to 1 local checkpoint per 30 execution loops. Figure 8, plots the recovery time versus the local checkpoint frequency.

I: Accumulative error used in the Integral term of the PID controller
D: Previous error used in the differential term of the PID controller
RC: Roll correction
PC: Pitch correction
YC: Yaw correction

Figure 6: Checkpointing storage data structure

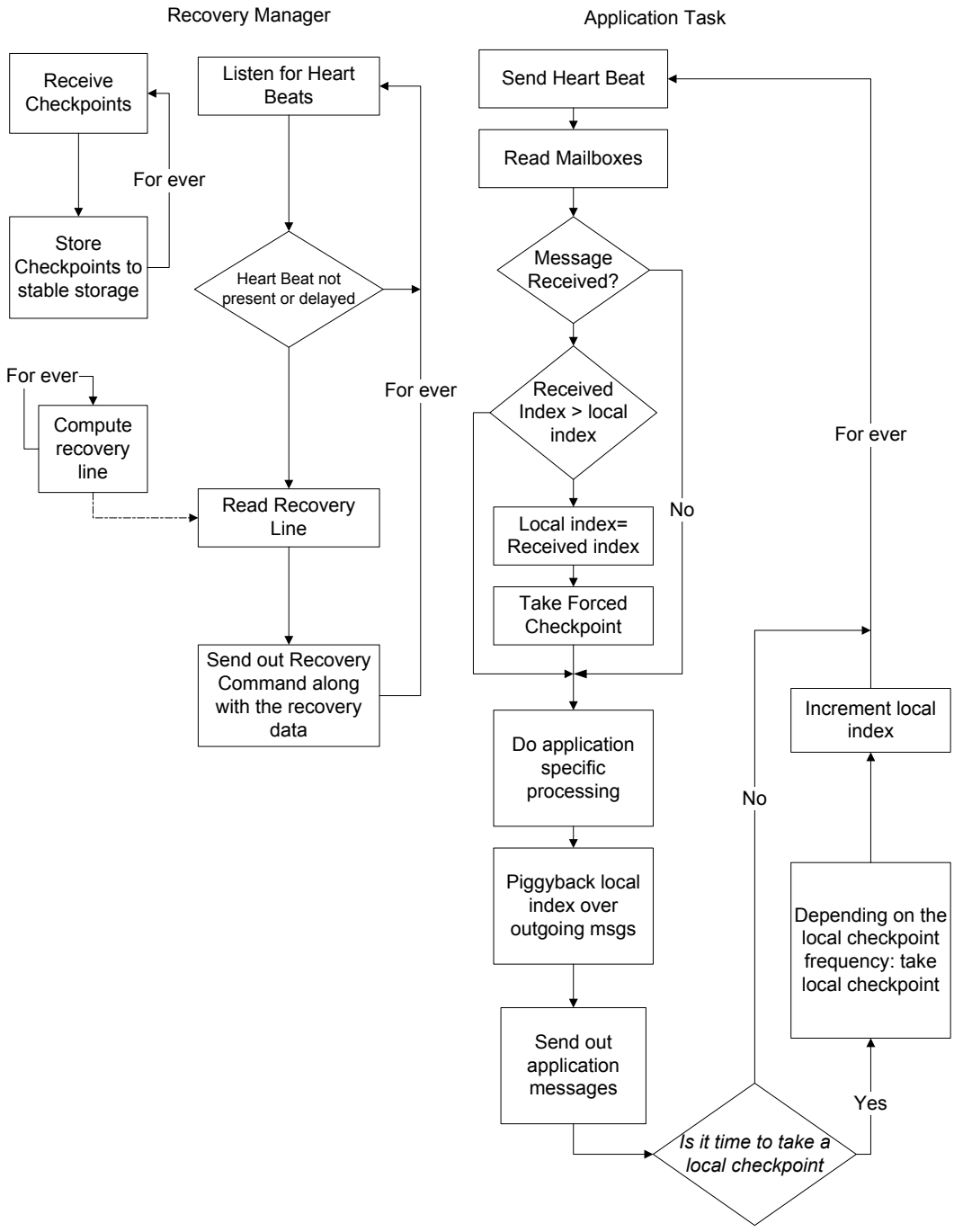

Figure 7: Application Tasks and Recovery Manager

From the figure, it can be observed that the recovery time decreases as the local checkpoint frequency increases. This is due to the following reason. The frequency of exchanging application messages in the distributed network is constant and does not change with the change of local checkpoint frequency. So, as the local checkpoint frequency increases the checkpoint indexes maintained by each of the checkpointing tasks increases rapidly. The exchange of application messages that piggyback these indexes does not

increase, hence reduced checkpoint synchronization. Furthermore, at the recovery manager's side, more checkpoints are received and required to be stored to stable storage.

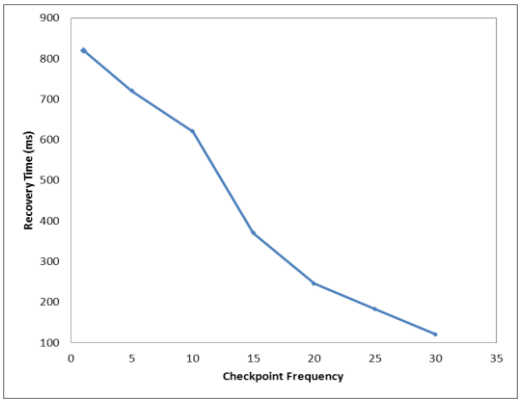

Figure 8: Recovery time at different checkpointing frequencies

However, the received checkpoints due to the lake of synchronization are more probable to not have the same index value, demanding more time to find a consistent recovery line by the recovery manager. On the other hand, as the local checkpointing frequency decreases, more application messages are available to synchronize the indexes among all checkpointing tasks. So the checkpoints received by the checkpointing manager are more probable to be synchronized, hence requiring less time to find a recovery line by the recovery manager, therefore less time to recover.

From the study, the recovery time for the system at 1 checkpoint per 30 execution loops was 120 ms. And because the execution rate for the fastest task in the system is 20ms, the system will miss 6 (120 ms/20 ms) execution loops, which is acceptable in these kinds of applications according to our flight testing. In general, to judge that a recovery time delay is acceptable or not depends on the application and how much execution loops it is going to lose during the recovery process.

VII. CONCLUSION

Graceful degradation is a promising technique that may achieve dependability with a significant reduction in cost, size, weight, and power requirements. This paper overviews a whole graceful degradation approach to achieve fault tolerance in resource constrained real-time embedded systems. The paper presented the development and implementation of an architecture for a complete gracefully degrading system that includes checkpointing coordination, checkpoint management, stable storage, and recovery management. The implementation was in form of an avionics system executing three control loops in parallel. Faults were injected during run-time causing the system's stability control tasks to fail. The system was able to recover in a very short time (around 120 ms), which was acceptable for the application to stay stable according to our flight testing.

REFERENCES

[1] Koopman, P., "Embedded System Design Issues -- The Rest of the Story", Proceedings of the 1996 International Conference on Computer Design, Austin, October 7-9 1996.

[2] A. Avizienis and J. Laprie, ``Dependable Computing: From Concepts to Design Diversity," Proc. IEEE, vol.74, no.5, pp.629-638, May 1986.

[3] A. Avizienis, J.-C. Laprie and B. Randell, "Fundamental Concepts of Dependability," Research Report No. 1145, LAAS-CNRS, April 2001.

[4] A.K. Somani and N.H. Vaidya, ``Understanding fault-tolerance and reliability," IEEE Computer, vol.30, no.4, pp.45-50, Apr. 1997.

[5] I. Koren and C. M. Krishna, Fault-Tolerant Systems, Morgan-Kaufman, San Francisco, CA, 2007.

[6] Strunk, Elisabeth A., John C. Knight, and M. Anthony Aiello, "Distributed Reconfigurable Avionics Architectures," 23rd Digital Avionics Systems Conference, Salt Lake City, UT, October 2004.

[7] R. P. Dick, N. K. Jha. "CORDS: Hardware-Software Co-Synthesis of Reconfigurable Real-Time Distributed Embedded Systems." IEEWACM International Conference on Computer Aided Design, pages 62-68, San Jose, California, November, 1998.

[8] R. Feldmann, C. Haubelt, B. Monien, and J. Teich. "Fault Tolerance Analysis of Distributed Reconfigurable Systems Using SAT-Based Techniques," In P. Y. K. Cheung, G. A. Constantinides, and J. T. de Sousa, editors, Field-Programmable Logic and Applications, Lecture Notes in Computer Science (LNCS), volume 2778, pages 478–487, Berlin, Heidelberg, Sept. 2003. Springer.

[9] Rawashdeh, O., and Lumpp, J., "Run-Time Behavior of Ardea: A Dynamically Reconfiguring Distributed Embedded Control Architecture," IEEE Aerospace Conference, IEEEAC Paper# 1516, March 2006.

[10] M.Eisenring, M.Platzner, "A framework for run-time reconfigurable systems", The Journal of Supercomputing, v.21, pp.145–159, 2002

[11] Thilo Streichert, Dirk Koch, Christian Haubelt, and Jrgen Teich, "Modeling and design of fault-tolerant and self-adaptive reconfigurable networked embedded systems," EURASIP Journal on Embedded Systems, Special Issue on Field-Programmable Gate Arrays in Embedded Systems., 2006.

[12] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-min Wang and David B. Johnson, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," ACM Computing Surveys, vol. 34, No. 3, pp. 375–408, September 2002.

[13] Belal H. Sababha and Osamah Rawashdeh, "Evaluation of Communication Induced Checkpointing in Resource Constrained Embedded Systems," The 7th International ASME/IEEE Conference on Mechatronics & Embedded Systems & Applications (MESA 2011) August 28-31, 2011, Washington, DC, USA.

[14] Belal H. Sababha and Osamah A. Rawashdeh, "Evaluation of Communication Induced Checkpointing on a CAN-Based Distributed System," The 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2010), Chicago, IL, June 29, 2010.

[15] Belal Sababha and Osamah Rawashdeh, "Evaluation of Communication Induced Checkpointing Approaches for Reconfiguration-Based Fault-Tolerance in Embedded Systems," Journal on Computing (JoC), vol. 1, no.4, January 2012.

[16] Belal H. Sababha, Osamah A. Rawashdeh, and Guangzhi Qu, "A Test-Bed for Reconfiguration-Based Fault-Tolerance in Distributed Embedded Systems," The International Conference on Information and Communications Systems (ICICS2009), Paper # 500, Amman, Jordan, Dec 20, 2009.

[17] D. Briatico, A. Ciuffoloetti, and L. Simoncini, “A Distributed Domino-Effect Free Recovery Algorithm,” Proc. IEEE Fourth Symp. Reliability in Distributed Software and Database Systems, pp. 207-215, 1984.

[18] Hong Yang, Rami AbouSleiman, Belal Sababha, Ermal Gjioni, Daniel Korff, and Osamah Rawashdeh, "Implementation of an Autonomous Surveillance Quadrotor System," Proc. of AIAA Unmanned Unlimited Conference, Paper # 2009-2047, Seattle, WA, April 6, 2009.

[19] O. A. Rawashdeh, H.C. Yang, R. AbouSleiman, and B. H. Sababha, "Microraptor: A Low-Cost Autonomous Quadrotor System," Proc. of the 2009 ASME/IEEE International Conference on Mechatronic and Embedded Systems and Applications (MESA09), DETC2009-86490, San Diego, CA, Sep 1, 2009.