DOT/FAA/AR-11/5

Air Traffic Organization
NextGen & Operations Planning
Office of Research and
Technology Development
Washington, DC 20591

Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No. 43 Phase 5 Report

May 2011

Final Report

This document is available to the U.S. public through the National Technical Information Services (NTIS), Springfield, Virginia 22161.

This document is also available from the Federal Aviation Administration William J. Hughes Technical Center at actlibrary.tc.faa.gov.

U.S. Department of Transportation
Federal Aviation Administration

NOTICE

Technical Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
DOT/FAA/AR-11/5		

4. Title and Subtitle	5. Report Date
MICROPROCESSOR EVALUATIONS FOR SAFETY-CRITICAL, REAL-TIME APPLICATIONS: AUTHORITY FOR EXPENDITURE NO. 43 PHASE 5 REPORT	May 2011
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
Rabi N. Mahapatra, Jason Lee, Nikhil Gupta, and Bob Manners*	
9. Performing Organization Name and Address	10. Work Unit No. (TRAIS)
Aerospace Vehicle Systems Institute Texas Engineering Experiment Station Texas A&M University Department of Computer Science College Station, TX 77843-3141 *Lumark Technologies, Inc. 11320 Random Hills Road Fairfax, VA 22030	11. Contract or Grant No.
12. Sponsoring Agency Name and Address	13. Type of Report and Period Covered
U.S. Department of Transportation Federal Aviation Administration Air Traffic Organization NextGen & Operations Planning Office of Research and Technology Development Washington, DC 20591	Final Report April 2009 – March 2010
	14. Sponsoring Agency Code AIR-120

15. Supplementary Notes

The Federal Aviation Administration Airport and Aircraft Safety R&D Division COTR was Charles Kilgore.

16. Abstract

This report documents the results of Phase 5 of the Aerospace Vehicle Systems Institute Authority for Expenditure (AFE) 43 Microprocessor Evaluations Project that provides research information intended to help aerospace system developers, integrators, and regulatory agency personnel in the selection and evaluation of commercial off-the-shelf microprocessors for use in aircraft systems.

This report describes the cooperative research accomplished by contributing members of the aerospace industry and the Federal Aviation Administration (FAA). The project objectives included (1) identifying common risks of using a system-on-a-chip (SoC) and mitigation techniques to provide evidence that they satisfy regulatory requirements and (2) evaluating existing regulatory policy and guidelines against the emerging characteristics of complex, nondeterministic microprocessors and SoCs to support the certification of aircraft and qualification of systems using these devices.

Complex aircraft system development requires more robust consideration of system failure and anomaly detection, correction, and recovery. The safety net approach identified in this report provides a means to address the lack of design assurance for highly integrated, complex, nondeterministic airborne electronic hardware within aircraft systems. Airworthiness assurance of these SoCs by safety net techniques may help to contain the labor burden and costs of compliance to FAA regulations while maintaining safety requirements as SoCs continue to become more complex and more widely used in aircraft systems. The safety net approach is consistent with, and does not replace, current FAA policy and guidelines.

17. Key Words	18. Distribution Statement
Microprocessor, System-on-a-chip, Avionics, Safety net, Aircraft certification, System qualification, Airborne electronic hardware, Aircraft safety, System architecture, Simulation, Operational level error detection and recovery, Integrated circuits, Avionics safety, Architecture patterns	This document is available to the U.S. public through the National Technical Information Service (NTIS), Springfield, Virginia 22161. This document is also available from the Federal Aviation Administration William J. Hughes Technical Center at actlibrary.tc.faa.gov.

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	58	

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

AEH	Airborne Electronic Hardware
AFE43	Authority for Expenditure No. 43
AMP	Asymmetric multiprocessing
ASCII	American Standard Code for Information Interchange
ATA	Advanced technology attachment
ATX	Advanced technology extended
AUD	Audio
CCB	Core complex bus
CCSRBAR	Configuration, Control, and Status Registers Base Address Register
COTS	Commercial off-the-shelf
D1	Disable control bit for DDR controller 1
D-Cache	Level 1 data cache
DDR	Device Disable Register
DEC	Decrementer
DIMM	Dual in-line memory module
DLAB	Divisor latch access bit
DMA	Direct memory access
DoS	Denial of service
DPFP	Double precision floating point
DUART	Dual universal asynchronous receiver transmitter
ECC	Error correction code
eLBC	Enhanced local bus controller
eTSEC	Enhanced three-speed Ethernet controller
FAA	Federal Aviation Administration
FEC	Fast Ethernet controller
FIFO	First-In First-Out
FPGA	Field-programmable gate array
FSRAM	Fast Static Random Access Memory
GB	Gigabytes
GNU	“GNU’s Not Unix!”—Unix-like computer operating system developed by the GNU Project
HDD	Hard disk drive
I-Cache	Level 1 instruction cache
I^2C	Inter-integrated circuit
I/O	Input/output
ID	Identification
IDE	Integrated development environment
INTx	Interrupt
IPsec	Internet Protocol Security
ISA	Instruction set architecture
Kbyte	Kilobyte
L2	Level 2 Cache
LBC	Local bus controller
LPC	Low pin count

LTIB	Linux Target Image Builder
MSI	Message signaled interrupt
NAND	Not AND
NOR	Not OR
OS	Operating system
PATA	Parallel ATA
PBIT	Periodic built-in test
PCI	Peripheral component interconnect
PCIe	PCI-Express
PEX	PCI-Express
PHY	Physical transceiver
PIC	Programmable Interrupt Controller
PME	Pattern Matching Engine
POST	Power-on self-test
PS/2	Personal System/2
R	Read
R/W	Read/Write
RAM	Random access memory
SATA	Serial ATA
SDRAM	Synchronous dynamic RAM
SEC	Security engine
SER	Serial
SerDes	Serializer/Deserializer
SGMII	Serial Gigabit Media Independent Interface
SIO	Super I/O
SMP	Symmetric multiprocessing
SoC	System-on-a-chip
SPFP	Single precision floating point
SRIO	Serial RapidIO
TB_n	Time base
TCP/IP	Transmission Control Protocol/Internet Protocol
TLU	Table lookup unit
TPM	Trusted platform module
TSEC	Triple-speed Ethernet controller
UART	Universal Asynchronous Receiver/Transmitter
UFCR	UART FIFO Control Register
ULCR	UART Line Control Register
ULi	ULi Electronics, Inc.
URBR	UART receiver buffer register
USB	Universal serial bus
UTHR	UART Transmitter Holding Register
V_{TT}	Voltage termination
W	Write
WCET	Worst-case execution time
XOR	Exclusive Or

EXECUTIVE SUMMARY

The Authority for Expenditure No. 43 (AFE43) Microprocessor Evaluations Project (Phases 1 to 5) focused on the escalating difficulties in evaluating modern, state-of-the-art, and highly integrated commercial off-the-shelf (COTS) microprocessors that (1) may not provide adequate visibility and debug features to reveal internal functionality, (2) are less predictable due to the interaction of advanced features, (3) have programmable configuration capabilities available to application software, and (4) share resources across multiple cores and devices. These COTS microprocessors exhibit nondeterministic characteristics and are consequently becoming more challenging to test and to determine that they satisfy applicable functional and safety-related requirements.

The research addressed the use of COTS microprocessors and systems-on-a-chip (SoC) in complex and safety-critical avionics. The project objectives included (1) Identify common risks of using SoCs and mitigation techniques to provide evidence that they satisfy regulatory requirements; and (2) Evaluate existing regulatory policy and guidelines against the emerging characteristics of complex, nondeterministic microprocessors and SoCs to support the certification of aircraft and qualification of systems using these devices.

The objectives of the AFE43 Project included researching an alternative method of ensuring aircraft safety when it becomes too difficult or impossible to perform design assurance on systems that use COTS microprocessors and SoCs. The AFE43 Project developed the concept of safety nets to monitor system operations, detect anomalous conditions and recover system operations adequate to meet safety and availability requirements.

A safety net is defined as the employment of mitigations and protections at a level above the functional elements containing the microprocessor(s) and/or SoCs of aircraft and system design to help ensure continuous safe flight and landing. The safety net methodology focuses on the assumption that a microprocessor will misbehave. The safety net would be deployed in the operational environment as a safety device within the aircraft. A safety net provides the ability to protect against unexpected behavior, damage, injury, and instability over the service life of the microprocessor(s) or SoCs outside, or at a level above the device itself, is necessary as appropriate for the design assurance level.

Subject matter experts from the FAA and the six industry participants of AFE43 first identified common microprocessor risks and an approach to resolve the growing issues with design assurance and certification; Phase 5 related these risks with mitigation methods associated with the multilevel safety net approach. Phase 5 developed the concepts of a multilevel safety net that established an approach to system design, including mechanisms to detect, analyze, and respond to SoC anomalous behavior at higher system levels.

The primary deliverable of the entire 5-phase AFE43 Project was a Handbook for the Selection and Evaluation of Microprocessors for Airborne Systems, and is, therefore, an important subject for this Phase 5 Report. The Phase 4 Report and this Phase 5 Report specify the research results that support the Handbook. The Handbook advocates a major shift in ensuring the safe use of

COTS microprocessors in airborne systems. Most complex hardware, including COTS microprocessors, goes through a process of demonstrating safety through the complete verification of the hardware design. The AFE43 Project has shown that this process is infeasible for some complex, nondeterministic COTS microprocessors. These microprocessors should be assumed as potentially unsafe, and system-level approaches for risk mitigation should be considered as a safety net.

Current FAA policy and guidance do not directly address the use of COTS microprocessors and SoCs in aircraft systems. However, the existing policy and guidance can be used as a basis from which the Handbook may help provide an applicant with additional information in demonstrating that their system meets the applicable airworthiness requirements.

1. INTRODUCTION.

Microprocessors and systems-on-a-chip (SoC) have become extremely complex, highly integrated, nondeterministic, and densely packaged. Recent changes in commercial off-the-shelf (COTS) microprocessors can be characterized as both physical and functional changes. Physically, transistor density has continued its exponential increase, allowing for hundreds of millions to billions of transistors to be placed on a single device. As of 2010, 65- and 45-nm devices were common in the COTS marketplace, and 32-nm and smaller devices were beginning to enter the marketplace. In addition to the decrease in device size, the functional capability of COTS devices has expanded. It is no longer necessary for different system components to be implemented as discrete devices. Instead, a single COTS SoC may contain multiple microprocessor cores, input/output devices, memory controllers, and other functionality. As a result, deterministic performance is difficult or impossible to predict in some cases. These devices require additional evaluation methods beyond that identified in current regulatory requirements to achieve the resilience required to meet safety and reliability requirements. The aircraft systems containing these COTS devices may require multilevel safety nets to be designed into them.

This report summarizes the experiments, findings, and activities of Phase 5 of the Aerospace Vehicle Systems Institute Authority for Expenditure No. 43 (AFE43): Microprocessor Evaluations Project. Phase 5 is the final phase of this project, and the primary deliverable is the "Handbook for the Selection and Evaluation of Microprocessors for Airborne Systems" [1]. Because of the Handbook's importance, it will be described in more detail in section 2. The Handbook serves as a means for system designers and regulatory personnel to have a common understanding of

- current Federal Aviation Administration (FAA) regulations and guidance regarding COTS airborne electronic hardware (initially focusing on microprocessors).
- the risks of using COTS microprocessors and SoC in airborne systems.
- safety nets—a system-level means for mitigating these risks.

In addition to describing the outcome of Phase 5 of the AFE43 Project, this report is also a source of supplemental information for the Handbook. The experimental platforms and data that assisted in forming the conclusions stated in the Handbook are detailed in this report. Also, other useful information that did not fit the theme of the Handbook can be found here.

For readers interested in understanding the development and progress of the entire project, each of the previous phases of the AFE43 Project are documented in a separate technical reports [2-5]. Phase 1 and 2 evaluated the FAA guidance and direction for the selection, design assurance, and qualification of airborne systems using COTS microprocessors and SoCs to support the certification of aircraft. These two phases attempted to find a technical solution of testing COTS microprocessors and SoCs at the component level and found that the evolving problem set was becoming unworkable.

Phase 3 included an intentional change of approach to make the AFE43 Project an industry-driven project to find an approach to solve the problem issues related to complex, modern, indeterminate, COTS microprocessors that are required to meet FAA regulations and specific safety requirements.

Phases 4 and 5 comprised a 2-year effort to develop a solution that may require additional considerations of augmenting the FAA regulations and new approaches beyond testing microprocessors and SoCs at the device level. Phase 4 identified common risk areas in complex SoCs that made testing at the component level problematic. Physical and simulated evaluation platforms were acquired during Phase 5 to determine the severity of the identified common risk areas and to determine the differences between using physical and simulated platforms. Phase 5 also developed the multilevel safety net that established an approach to system design, including mechanisms to detect, analyze, and respond to SoC anomalous behavior at higher system levels.

The Handbook was developed during Phase 5, but also received inputs from Phase 4. The Handbook advocates a major shift in ensuring the safe use of COTS microprocessors in airborne systems. Most complex hardware, excluding COTS microprocessors, goes through a process of demonstrating safety through the complete verification of the hardware design, as specified by guidance, such as Advisory Circular (AC) 20-125 and RTCA DO-254 [6 and 7]. AFE43 has shown that this process is infeasible for some complex, nondeterministic COTS microprocessors. These microprocessors should be assumed potentially unsafe, and system-level approaches for risk mitigation should be considered. This research project terms these system-level approaches as safety nets. COTS microprocessors are expected to be managed per Section 11.2 of DO-254.

2. HANDBOOK FOR THE SELECTION AND EVALUATION OF MICROPROCESSORS FOR AIRBORNE SYSTEMS.

The primary focus for the AFE43 Project Phase 5 was the development of the Handbook for the Selection and Evaluation of Microprocessors for Airborne Systems [1]. This report acknowledges the Handbook as having precedence in the definition and description of safety nets and their application.

The Handbook discusses three major topics:

- Regulatory considerations for microprocessor-based airborne applications—the FAA adoption and scoping of DO-254 to certain types of airborne electronic hardware, and current FAA policy regarding COTS microprocessors.

- COTS microprocessor and SoC risks—an identification and detailed description of three major, common risk areas present in modern COTS microprocessors and SoCs, are described in section 3 of this report.

- Safety nets—a definition and description of system-level safety net approaches, COTS microprocessor selection considerations, and examples of possible safety net designs.

Considering the growing complexity of microprocessors, the research revealed the increasing impracticality of ensuring safety at the device level alone. The combination of growing complexity of both software and hardware will drive the need to evaluate large complex systems at multiple levels, including the operational system level. The rate of change and growing complexity is accelerating, and the time between new generations of hardware and software is decreasing. The complexity of many COTS components has grown and continues to grow beyond the capabilities of DO-178B and DO-254 processes to evaluate them. The overlapping phases of development, certification, deployment, and life cycle maintenance together with COTS obsolescence accentuate the need for rapid and evolving methodologies. FAA policy, guidelines, and practices may need to be updated to accept these methodologies.

Complex aircraft system development requires more robust consideration of system failure and anomaly detection, correction, and recovery. The safety net approach identified in this report provides a means to address the lack of design assurance for highly integrated, complex, nondeterministic airborne electronic hardware (AEH) within aircraft systems. This may help to contain the labor burden and costs of compliance to FAA regulations as SOCs continue to become more complex and more widely used in aircraft systems.

2.1 HANDBOOK EXECUTIVE SUMMARY.

The Executive Summary of the Handbook describes its function and driving resolutions as follows:

> "This Handbook provides research information intended to help aerospace system developers and integrators and regulatory agency personnel in the selection and evaluation of commercial off-the-shelf (COTS) microprocessors for use in aircraft systems. AEH includes modern state-of-the-art and highly integrated COTS microprocessors that (1) may not provide adequate visibility and debug features to reveal internal functionality, (2) are less predictable due to the interaction of advanced features, (3) have programmable configuration capabilities available to application software, and (4) share resources across multiple cores and devices. These highly complex COTS microprocessors are becoming more challenging to test and to determine that they satisfy applicable functional and safety-related requirements.
>
> Resolutions to certification process challenges should offer the possibility of establishing and maintaining standards that support the continual change and growth of technologies and operations. Such resolutions can include
>
> - establishing qualitative as well as quantitative methods to certifying aircraft with embedded nondeterminate complex/critical applications.
>
> - establishing accepted standards of architectural patterns for critical, complex systems and methods for validation and design assurance.

- establishing industrywide accepted methods for design assurance of COTS microprocessor and microprocessor-based systems.

- streamlining the certification process."

2.2 HANDBOOK OBJECTIVES.

The objectives of the Handbook are to

- document common areas of concerns regarding the use of COTS microprocessors in complex and/or safety critical systems.

- provide approaches, information, and examples for mitigating the concerns through a safety net.

- provide access to the research on which the content of this Handbook is based.

- provide example approaches to resilient systems through methods defined in this Handbook under the overarching term safety nets.

- reveal how existing regulatory policy and guidance may be augmented to support the creation of resilient systems through safety net approaches safeguarding the use of microprocessor technologies in complex and/or safety critical systems.

2.3 HANDBOOK CHARACTERISTICS.

The Handbook was written for experienced system designers and regulatory personnel and intentionally not prescriptive in nature. It was intended to support the development of new approaches to the design assurance and safety evaluation leading to the approval of airborne systems. The design of an airborne system, the selection of AEH (e.g., microprocessor) devices within the system, and the architecture of the system will be unique for each application. The potential sources of nondeterminism and the challenges of design assurance of AEH devices must be determined for each system. The Handbook does not attempt to identify sources of microprocessor nondeterminism because they will be unique for each system and will proliferate in the future. System designers have to evaluate the risks associated with the candidate microprocessors and design the system architecture and the safety nets to mitigate these risks.

The Handbook does not constitute FAA policy or guidance; rather, it is the result of FAA/industry-funded research and may contribute to future policy or guidance.

2.4 SAFETY NET APPROACH.

A safety net is defined as the employment of mitigations and protections at the appropriate level of aircraft and system design to help ensure continuous safe flight and landing. The safety net methodology focuses on the assumption that a microprocessor will misbehave. The ability to protect against unexpected behavior, damage, injury, and instability over the service life outside, or at a level above the device itself, is necessary as appropriate for the design assurance level.

The safety net approach is a means to mitigate the risks associated with COTS microprocessors via both passive and active methods designed into aircraft systems. If it is not feasible to show that complex aircraft systems are sufficiently free of anomalous behavior by evaluating system components and system design, the safety net approach can mitigate unforeseen or undesirable COTS microprocessor operation by detecting and recovering from anomalous behavior at the operational system level. This approach requires the safety net to be designed as a function within the aircraft system. The safety net can include passive monitoring functions, active fault avoidance functions, and control functions for recovery of system operations. System architecture and control and recovery functions should be designed to facilitate effective system recovery from anomalous events. Safety nets should show that systems are sufficiently impervious to anomalous behavior by ensuring continuous functional availability and reliability, satisfying applicable regulations, and meeting airworthiness requirements. This includes verifying any disabled functionality from the COTS will remain inactive in the specific application.

A multilevel safety net approach is required for complex and critical applications in systems that cannot be fully assured at the component level and is significantly linked to the assigned design assurance level required by regulation, contractual obligation, and the integrated complexity at the device level. Safety net design, in general, is becoming a complex, application specific approach that will be required to detect, resolve, and validate recovery in a run-time environment to the required levels of availability and safety. The safety net approach is consistent with current FAA policy and guidelines. Refer to the Handbook [1] for a more complete discussion of safety nets.

3. THE COTS MICROPROCESSOR RISK AREAS.

This section provides an overview of COTS microprocessor trends and the risks presented by their deployment in aerospace systems.

Recent changes in COTS microprocessors can be characterized as both physical and functional changes. The physical size of transistors has been shrinking rapidly, allowing for hundreds of millions to billions of transistors to be placed on a single device. As of 2010, 65- and 45-nm devices were common in the COTS marketplace, and 32-nm and smaller devices are beginning to enter the marketplace.

The rapid reduction in transistor size has expanded the functional capability of single COTS devices. It is no longer necessary for different system components to be implemented as discrete devices. Instead, a single COTS SoC may contain one or more microprocessor cores, input/output (I/O) devices, memory controllers, and other functionality.

The result of this functional integration is reduced visibility and control of the COTS SoC and its devices. Unless the manufacturer explicitly provides mechanisms to inspect the behavior of the SoC, it is exceedingly difficult to understand how devices within the SoC operate and interact.

Additionally, the configuration of the many devices within COTS SoC is typically controlled by software settings. This reduces the confidence that the system configuration will remain stable during operation.

Ultimately, these trends in COTS SoCs provide for very high-performance systems with less power and reduced cost for the same area size. Complex interaction with new ground support systems, more capable human interfacing, and other new system functionality will eventually require the use of COTS SoCs. However, unless system designers and approval agents understand the risks of using these new devices in airborne systems and agree on adequate means to demonstrate their safe use, these economic and capability advantages will be offset by regulatory costs. This will preclude the wide adoption of COTS SoCs in airborne systems, despite their technological superiority over current solutions.

It is important to distinguish between discrete COTS microprocessors and SoCs when discussing certain elements of risk and safety analysis. The task of ascertaining safety considerations is more complicated for SoCs due to the broad variety in SoC designs. Unlike the case of discrete COTS microprocessors, where the majority of features of interest are similar across most microprocessors, SoC components tend to vary significantly based on the product selected, making safety analysis more complicated. Not only are the safety concerns due to the features of individual IP cores an issue, but interaction among them presents a verification challenge to system designers. Additionally, certain systems using SoCs may not require particular on-chip cores and would require the disabling of those cores for safety reasons.

Resources used by safety nets are expected to change as the underlying technology of the system being protected (or implemented within the safety net itself) evolves. An example of this occurs when configuration register changes (the addition of new, removal of old, and exposure of hidden/reserved registers) caused when separate, discrete microprocessors, system controllers, and other components are replaced with a single SoC. The safety net design must be examined and perhaps adjusted when such technology changes are implemented or when the device must be approved within a new application. The discussion above needs to be considered while formulating a safety net.

After reviewing the designs of a variety of modern COTS microprocessors across a group of manufacturers, product families, and technology generations, three areas of risk were identified as common to all these devices [4]:

- Visibility and Debug—The inability to observe the internal operation of the device during system use and development (see section 3.1)

- Configuration-Related Issues—Software accessibility to device configuration during system operation (see section 3.2)
- Shared Resources Effects—Performance unpredictability due to multiple on-chip shared resources (see section 3.3)

3.1 VISIBILITY AND DEBUG.

During the background research, a physical target computer environment and a simulated target computer environment were setup to perform experiments. Setting up these environments exposed visibility and debug challenges that may arise during typical airborne system development and analysis; some of these challenges are described in the Handbook [1]. The system developer should document the configuration of the test environment and identify any differences, limitations, and constraints of the simulated environment, if used, in relation to the physical environment.

3.2 CONFIGURATION-RELATED ISSUES.

Configuration register changes are a growing concern for avionic microprocessor applications since continued device integration has allowed an increasing proportion of the entire system to be configured through software. If a microprocessor is not configured properly, erroneous behavior, including improper data processing, stack overflows, erroneous interrupts, machine checks, data loss, data corruption, or inadequate throughput, may occur.

Over time, the number of configuration registers per microprocessor has grown significantly. For example, the Freescale™ MPC8572 SoC has more than 500 software-accessible configuration registers that control basic functionality of the processing cores and on-chip devices [8]. In addition to these configuration registers, various device functions may be set externally via pullup/pulldown pins that are sampled shortly after a hardware reset or internally via software. Incorrect settings or inadvertent changes are both areas of concern that must be addressed.

In general, the capabilities of most microprocessors exceed what is required by typical applications. Care should be taken to provide assurance that unused capabilities are properly disabled. In legacy avionics with many discrete system devices, this concern was addressed through physical disconnection of the unused devices from power sources and the rest of the system. However, COTS SoCs have removed the physical separation between the devices and processors and gave control of device configuration to software. Therefore, the deactivation of unused features has become an additional consideration within the set of configuration-related issues. In addition to proper deactivation of unused features and devices, system designers should be able to assure that those features and devices cannot be reactivated through erroneous software or environmental effects. Inadvertent activation of an unused device may cause unintended and undesirable operation, such as erroneous interrupts, data loss, data corruption, machine check cycles, and stack overflow, among others.

As an example of how the deactivation of unused features has changed for modern COTS microprocessors, figure 1 shows the Freescale MPC8572 device disable register (DDR).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
R W	—		PCIE1	—	LBC	PCIE2	PCIE3	SEC	PME	TLU1	TLU2	—	SRIO	—	D2	D1
Reset	0	0	n	0	0	n	n	0	0	0	0	0	n	0	0	0

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
R W	E500_0	TB0	E500_1	TB1	—	DMA1	DMA2	—	TSEC1	TSEC2	TSEC3	TSEC4	FEC	I2C	DUART	—
Reset	0	0	0	0	0	0	0	0	0	0	n	n	n	0	0	0

Figure 1. Freescale MPC8572 Device Disable Register [8]

This register is used by the system to determine whether each of the system's major on-chip devices should be enabled or disabled by setting particular register fields to 0 (enabled) or 1 (disabled). This configuration register contains 24 single-bit readable and writeable fields that are associated with the 24 on-chip devices of the SoC. For example, bits 16 and 18 control the two e500 processing cores [9], bits 2, 5, and 6 control the three PCI-Express (PCIe) controllers, and bits 24 through 27 control the four triple-speed Ethernet controllers. A single bit flip in this register can disable a critical on-chip device or one of the processors, and the experiments have demonstrated that erroneous writes to this register can render the system inoperable. Therefore, system designers should be aware of the risks of unexpected modifications to the system configuration space.

Several microprocessor devices also offer the user the flexibility of locating the configuration registers in external memory space, which makes them more vulnerable to corruption. This capability also makes the system susceptible to losing all configuration data if the pointers to the external memory locations are corrupted.

Configuration registers may inadvertently be changed by software errors (inadvertent writes), single-event upsets, hardware defects, hardware faults (such as a noisy power supply core voltage, signal integrity issues, and ground bounce), or electromagnetic interference.

In light of the risks identified with the current technology and trends and the criticality of proper microprocessor configuration register settings, the risks associated with incorrect configuration register settings should be understood. Each register should be assessed for:

- Intended setting for each register bit within the system being implemented and the reason for the selected setting.
- Identification of each operational phase at which each register is being set (i.e., initial power up, power on reset, built-in test, exception processing)
- Identification of disabled functions

- Impact to system if the state of the bit is unintentionally changed

  - It is recommended that the impact of an inadvertent change to critical registers or registers of questionable impact is verified through simulation, if possible.

  - The simulation environment should implement an algorithm that has the ability to randomly change configuration settings.

- Rate of impact to system of unintentionally changed bit, if known (i.e., inadvertently enabling a communication bus may have no impact for a major frame)

- Errata information

Additional consideration is recommended for determining the effect of configuration register default values. Vendor default values for configuration register settings are usually selected to simplify the process of getting a system "up and running" with minimal effort. These do not necessarily provide the needed configuration for the performance or exception detection needed by the intended application. Evidence that supports the assertion by the integrator that the settings are correct for the intended application should be provided. Examples include the following:

- Registers that are used to enable utilized interfaces and disable unutilized interfaces.
- Registers that are used to allocate access to shared resources (arbiters).

Beyond the general assessment recommended for all configuration registers described above, industry members have identified the following types of registers for special consideration. These registers may be helpful in constructing a safety net:

- Interrupt Mask Registers—These registers allow hardware exceptions to either generate an interrupt signal or inhibit the generation of the interrupt.

- Interrupt Cause Registers—These registers allow the apparent cause of hardware exception to be visible to software. Often, the cause is separate from the generation/masking process so that the implementer may poll the cause register. Usually, provisions exist internally to capture only one occurrence of a set of causes at a time. Therefore, the system implementer must take additional measures in the interrupt handler to capture subsequent occurrences if this is required.

- Error Counters—These registers allow accumulated error events to be visible to software. Examples of these include the error correction code (ECC) single-bit error counters and ECC double-bit error counters on memory bus interfaces.

Once this analysis has been completed, safety net methodologies can be employed to mitigate the identified risks.

3.3 RESOURCE-SHARING CONSIDERATIONS.

Modern microprocessors differ from previous technology in that many processing, memory, and I/O components reside within a single device, and many of these components are designed to be shared to optimize system performance. The multiple processing components compete to initiate requests to their memory and I/O targets. Additionally, the I/O components can initiate requests to memory targets through direct memory access. Whether initiated by a processor or I/O controller, requests travel over shared on-chip interconnects, typically a single on-chip bus. The Handbook defines shared resources as any on-chip or off-chip components that are accessible to multiple initiators. Examples of shared resources include on-chip memory controllers, hardware accelerators, level 2 (L2) and/or level 3 caches, on-chip busses, and on-chip Ethernet controllers.

Modern microprocessors feature a number of shared resources. As shown in figure 2, the two e500 cores of the Freescale MPC8572E share a single L2 cache [8]. Additionally, the two processing cores share a single on-chip bus to access the other major components of the system, including the various I/O controllers and memory controllers.

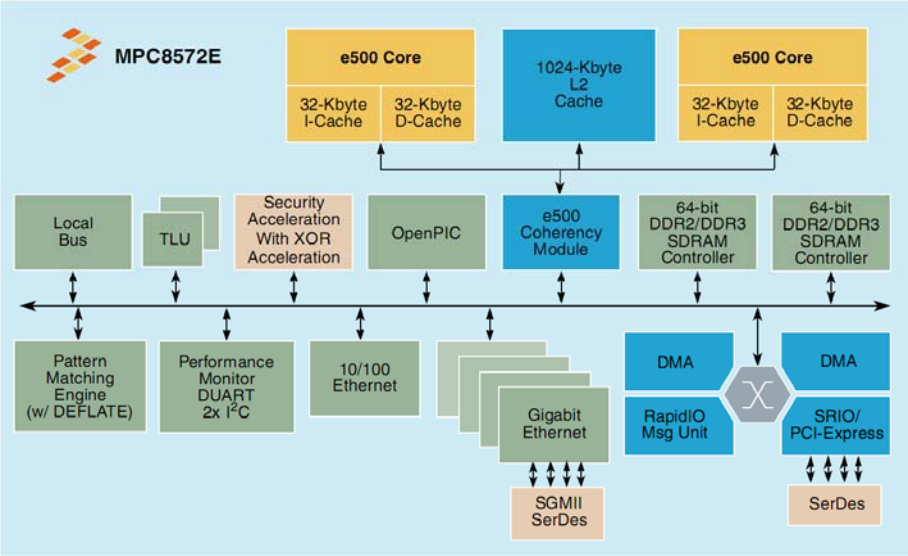


Figure 2. Freescale MPC8572E Block Diagram [8]

It is highly recommended that the system designer analyze the access protocol for these shared resources and the run-time behavior of all programs that share a given resource. Based on this analysis, the designer should ensure that even with the sharing of resources, the system will continue to run in a predictable manner (continuous operation). In some conditions, hard shutdowns can be an acceptable safety protection mechanism.

Sharing resources is a major contributor to nondeterminism and worst-case execution time (WCET) analysis challenges in modern COTS microprocessors. Nondeterminism arises because the availability of a shared resource becomes largely dependent on the run-time behavior of other processes sharing the same resource. In many cases, the run-time behavior of programs is data-dependent and cannot be predicted offline. WCET analysis depends on understanding all

conditions that lead to timing delays and then bounding for worst-case conditions. Multiple shared resources on a single device complicate this analysis due to the increase in the number of delay conditions.

Pellizoni, et al. [10], describe the challenges in predicting the WCET in a multitasking system. They show that due to the interference between cache-fetching activities and I/O peripheral transactions, tasks can suffer computation time variance of up to 46% in a typical embedded system.

To further assess the timing delays and nondeterminism caused by resource sharing, additional experiments were performed using a simple matrix multiplication program on the Freescale MPC8572 platform considering the L2 cache as the shared resource, as described in section 6. Each processing core executed its own copy of a matrix multiplication program that required the use of the shared L2 cache, due to the program size. The experiments showed that the execution time of the matrix multiplication program can increase by as much as 17% as L2 cache interference increases.

Moscibroda, et al. [11], describe that in a multicore system, multiple programs running on different cores can interfere with each other's memory access requests, thereby adversely affecting performance [11]. They show that a competing program running on one processing core can result in a denial of service (DoS) on the other processing core, due to the inherent unfairness in memory controller access policy. The performance of a blocked application can be reduced by as much as 2.9 times in a typical dual-core system. Moscibroda, et al., identify the memory access scheduling algorithm as the main source of inequality in memory access, allowing the DoS to occur. Shared resource access policy and scheduling is addressed below.

Industry trends indicate that the ratio of processing cores to various shared resources in COTS SoCs will increase over time. Instead of two processing cores sharing a common cache, memory controllers, and I/O devices, there will be four, eight, or more processing cores sharing these resources. This will increase the competition for shared resources among processing cores, worsening potential unpredictability issues.

In addition to the contention of shared resources being an area of risk for microprocessors, it is also important to consider the sharing of interfaces for the purpose of data transfer and exception signaling. Exception condition signaling mechanisms should be considered because they impact both the ability to deliver exception notifications as well as the latency of exception event notification. Examples from various forms of peripheral component interconnect (PCI) interfaces, include:

- Mechanisms that use the same channel as the one that is being monitored—Message signaled interrupts (MSI) on PCIe links that are communicated as unique bit patterns within the same channel used for bus commands, addresses, and data. If the channel is disrupted, then the notification method is disrupted as well. Also, due to the serialized nature of this mechanism notification is delayed until the traffic in front of it has been transferred. Note that the parity error and system error signals of conventional PCI buses

also fall into this category because they are generated along with the bus transaction cycles they represent and are transferred along with the transaction results.

- Mechanisms that use a different channel than the one being monitored—The conventional PCI interrupt mechanism uses discrete lines to the interrupt controller devices outside the lines that transfer bus commands, addresses, and data, though it also appears on the same connector when implemented as an external interface. Due to the nonserialized nature of this mechanism, notification is not delayed by traffic on the bus. If the address/data bus portion is disrupted, then the interrupt notification mechanism may still be intact.

4. THE COTS MICROPROCESSOR EVALUATION PLATFORM.

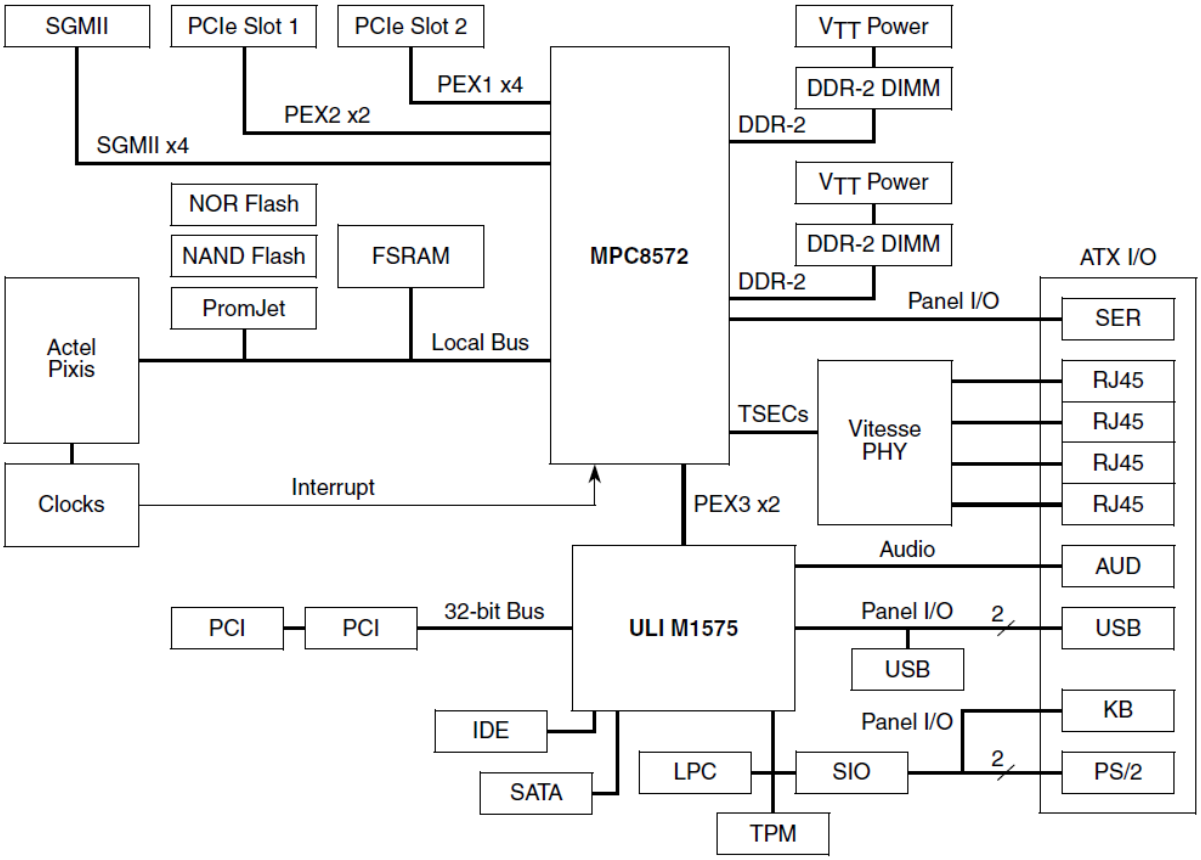
This section describes the target platforms that were established to perform experiments and evaluate the criticality of the above mentioned risk areas. Two different platforms were used for experimentation in this research:

- Physical—Freescale MPC8572DS platform
- Simulated—Simics® MPC8572DS model

Having the Simics simulated model for the MPC8572DS hardware allowed the researchers to easily perform tasks, such as fault injection and performance monitoring, which would have been more difficult to perform on the hardware platform.

4.1 PHYSICAL FREESCALE MPC8572DS PLATFORM.

The MPC8572E is a dual-core PowerPC e500v2 microprocessor that includes high-speed interconnect technology to balance processor performance with I/O throughput. In addition, the MPC8572E offers the following: a double-precision, floating-point auxiliary processing unit; 1024 Kbytes of L2 cache; four integrated 10/100/1GB-enhanced, three-speed Ethernet controllers (eTSEC) with transmission control protocol (TCP/IP) acceleration, classification capabilities, and SGMII interface capabilities; a 10/100 fast Ethernet controller (FEC) maintenance port; two table lookup units (TLU); two DDR2/DDR3 SDRAM memory controllers; a multiprocessor programmable interrupt controller; inter-integrated circuit two (I^2C) controllers; two four-channel direct memory access (DMA) controllers; an integrated security engine (SEC) with XOR acceleration; an enhanced local bus controller (eLBC); a pattern matching engine (PME) with Deflate (lossless data compression algorithm that uses a combination of the LZ77 algorithm and Huffman coding) capabilities; a general-purpose I/O port; and dual universal asynchronous receiver/transmitters (DUART). For high-speed interconnect, the MPC8572E provides a set of multiplexed pins that support two high-speed interface standards: 1x/4x serial RapidIO (with message unit) and up to x8 PCIe (with the capability to offer three independent PCIe links). The high-level integration in the MPC8572E helps simplify board design and offers significant bandwidth and performance. Figure 3 shows the physical MPC8572DS development system used for experiments.



KB = Keyboard

Figure 3. System Diagram of Physical MPC8572DS [12]

Key features of the MPC8572E device include:

- Two PowerPC e500v2 cores with 36-bit physical addressing [9]
- 1024-Kbyte L2 cache
- Integrated security engine (SEC) with XOR acceleration
- Four integrated 10/100/1GB-enhanced three-speed Ethernet controllers (eTSEC) with TCP/IP acceleration, classification capabilities, and SGMII interface capabilities
- 10/100 Fast Ethernet controller (FEC) maintenance interface
- Two DDR2/DDR3 SDRAM memory controllers
  - High-speed interfaces:
  - Three PCIe controllers
- One serial RapidIO (SRIO) controller with RapidIO messaging unit

- Pattern matching engine (PME) with DEFLATE engine
- Two table lookup units (TLU)
- Programmable interrupt controller (PIC)
- Two, four-channel DMA controllers
- Two I^2C controllers
- DUART
- Enhanced local bus controller (eLBC)
- Eight general-purpose I/O signals
- Power management (power saving modes: doze, nap, and sleep)
- System performance monitor
- System access port
- IEEE Std 1149.1™ compatible, Joint Test Action Group boundary scan
- 1023 FC-PBGA package

The MPC8572DS features memory-mapped configuration, control, and status registers for the integrated peripherals starting at an offset defined by the configuration, control, and status registers base address register (CCSRBAR). No address translation is done, so there are no associated translation address registers. The configuration, control, and status window is always enabled with a fixed size of 1 megabyte.

4.2 BOARD SUPPORT PACKAGE.

The board support package features the Linux 2.6.27.6 kernel and provides the following tools, device drivers, and additional features needed for an embedded Linux project:

- Targeting Freescale MPC8572DS board Linux 2.6.27.6 kernel supporting e500v2 core
- Supports symmetric multiprocessing (SMP) Linux on both cores
- Supports asymmetric multiprocessing (AMP) Linux on each core
- eTSEC driver to support four eTSECs on 10M/100M/1000M Ethernet function
- e500 hardware floating-point exception handler patches to support the scalar single-precision floating point (SPFP), vector SPFP, and double-precision floating point (DPFP)

- DUART driver support 115200 baud without flow control
- 32-bit PCI host driver for 33 MHz to support Intel e100, 3Com 3C905, Intel Pro1000
- 82545EM Ethernet adaptors
- PCIe host driver to support the onboard ULi (Uli Electronics, Inc.) M1575 bridge and 2x mode driver to support Intel Pro 1000 network interface card for two slots
- Both INTx and MSI are supported on PCIe 2/3.
- Serial ATA (SATA) driver in kernel to support the SATA (Advanced technology attachment, where advanced technology derived from the IBM PC/AT naming of personal computers, and attachment AT is an interface standard for the connection of storage devices such as hard disks, solid-state drives, floppy drives, and optical drives in computers) module of ULi M1575 with hard disk utilities. Hard disk can be mounted manually.
- Support for ATI RADEON X800XL and X700 video card
- Pattern matching engine driver
- TLU Unit driver
- Linux kernel booting from network, flash, or hard disk drive (HDD), with ULi M1575 SATA HDD as default boot
- Support for LAMP (Linux 05, Apache HTTP Server, MySQL (database software), and Perl/PHP/Python, which form principle/components to build a viable general-purpose web server)
- Socket buffer recycling patch for eTSEC included
- Support for SEC 3.0 mainline Internet Protocol Security (IPsec) stack and mainline Talitos driver. SEC low-level driver is included.
- I^2C driver
- Real-time clock driver
- Support for ULi parallel ATA (PATA) controller
- ULi universal serial bus (USB) controller supports keyboard, mouse, and U-disk
- Memory technology device driver supports both not OR (NOR) and not AND (NAND) flash

- Dynamic power management driver
- Watchdog driver
- Support for JFFS2 file system
- Support for the user space device management (udev) file system
- Support for CodeWarrior debug in both SMP and AMP mode
- Linux Target Image Builder (LTIB) root file system on SATA HDD automatically mounted, including native "GNU's Not Unix!"—Unix-like computer operating system developed by the GNU Project (GNU) toolchains and application packages
- SATA hard disk ghost image
- TCP/IP stack
- File transfer program client and server
- Telnet client and server
- Support for both the network file system and random access memory (RAM) disk file systems

4.3 SIMICS MPC8472DS MODEL.

The simulation platform used in Phase 5 was the Virtutech Simics model of the Freescale MPC8572DS. Research was performed using Virtutech Simics 4.0 and Virtutech Model Library MPC8572 4.0.10 [13]. As done with building software for a physical MPC8572DS, the Freescale LTIB package was used to configure and compile the system bootloader and operating system for the simulation platform. The bootloader used for the simulation platform was U-Boot 2008.10, and the operating system used was Linux 2.6.27.6.

There are several important differences between the physical and simulation platforms; however, most of these differences had no effect on most normal software execution. These differences did result in functional discrepancies between the physical and simulation platforms during extended configuration register testing, as described in section 5.

The simulated model of the MPC8572DS is a partial simulation of the hardware and is composed of the following:

- Freescale MPC8572 SoC
- RAM
- NOR Flash
- PromJet

- Pixis field-programmable gate array (FPGA)
- Ethernet PHYs

The most notable difference between the physical and simulation MPC8572DS is the absence of the ULi M1575 South Bridge chip from the simulation model. This chip provides interfaces for USB devices, audio devices, and legacy PCI devices, among others. The South Bridge chip is connected to the MPC8572 SoC via a PCIe link in the physical platform.

4.4 ANALYSIS OF PHYSICAL AND SIMULATION PLATFORMS.

Hardware and software considerations were identified for system designers in evaluating the physical and simulation MPC8572DS platforms. The following hardware considerations apply for physical evaluation platforms:

- Limited visibility into the device's internal operation—Manufacturer documentation is intended to support the use of the device based on the capabilities described in the User's Manual and on the device characteristics documented in the data sheet/errata. Internal operation is frequently proprietary information. For example, the User's Manual may document how to configure the use of cache memory, but the cache algorithm being used internally by the device may be proprietary.

- Performance monitoring—Hardware performance monitors may be provided by the manufacturer to provide insight into the internal operation of a microprocessor. These monitors allow system designers to track various system activities and performance statistics during application development and execution. Hardware performance monitors are typically used to gather L1 and L2 cache statistics and measure cycle counts to estimate application execution times. Performance monitors may also allow a system designer to observe the activity of functions resident on the microprocessor, including memory controllers, PCIe and Ethernet controllers, and DMA engines. These performance monitors are uniquely designed for each COTS microprocessor, and access to the performance monitors typically requires custom software and is not fully supported by COTS operating systems.

- Fault insertion—The ability to insert faults internal to the microprocessor may be limited or not achievable. An applicant may not be able to demonstrate that faults are correctly detected if the faults cannot be injected. The safety net methodology can then mitigate these types of faults.

The simulation evaluation platform is capable of running most of the same software as the physical platform; however, limitations do exist for custom or specialized software. System designers should not expect that every configuration register for every system component is functional. Instead, the configuration registers required for a standard bootloader and operating system (e.g., Linux) are modeled. If a custom bootloader or operating system is used, additional system modeling may be required.

The following software and hardware considerations were identified for simulation platforms:

- Software considerations:
  - Applicants should be aware that the primary focus of microprocessor simulation is on application software development. Typically, hardware evaluation is a secondary concern, if addressed at all.
  - Modeling configuration and internal registers—The simulated computer environment typically models the minimum set of configuration and internal registers to support software execution. The differences between the simulated computer environment and the target computer should be documented by the system developer as part of the test environment. Evaluating the system response in a simulated environment requires accurate modeling of all configuration registers.
- Hardware considerations:
  - Limited modeling of hardware—Hardware interfaces and modeling of microprocessor functions may be limited to those items required for application software development.
  - Timing and cycle accuracy of the simulated target computer should be assessed. If the target computer model is not cycle accurate, functions which are timing sensitive should be verified in the target computer environment.
  - Modeling device performance—Internal microprocessor performance monitors may not be modeled in the simulator.
  - Microprocessor models focus on the simulation of core central processing units; modeling of other microprocessor resident functionality may be limited.

5. CONFIGURATION-RELATED ISSUES.

This section details the experiments and their results that were performed on the hardware platform to assess the severity of issues related to unexpected modifications to configuration registers.

5.1 EXPERIMENTS AND RESULTS.

An experiment was designed to test the effects of bit changes in configuration registers. In this experiment, register bits in the universal asynchronous receiver/transmitters (UART) configuration space were changed. The CCSRBAR register holds the base address of all the memory-mapped configuration registers on the MPC8572E. The UART configuration registers are located at an offset of 0x4500 from CCSRBAR. Figure 4 shows all the configuration registers for UART0. The registers for UART1 are similarly located, starting at 0x4600.

Offset	Register	Access	Reset	Section/Page
Block Base Address: 0x0_4000				
UART0 Registers				
0x500	URBR—ULCR[DLAB] = 0 UART0 receiver buffer register	R	0x00	13.3.1.1/13-5
0x500	UTHR—ULCR[DLAB] = 0 UART0 transmitter holding register	W	0x00	13.3.1.2/13-5
0x500	UDLB—ULCR[DLAB] = 1 UART0 divisor least significant byte register	R/W	0x00	13.3.1.3/13-6
0x501	UIER—ULCR[DLAB] = 0 UART0 interrupt enable register	R/W	0x00	13.3.1.4/13-8
0x501	UDMB—ULCR[DLAB] = 1 UART0 divisor most significant byte register	R/W	0x00	13.3.1.3/13-6
0x502	UIIR—ULCR[DLAB] = 0 UART0 interrupt ID register	R	0x01	13.3.1.5/13-8
0x502	UFCR—ULCR[DLAB] = 0 UART0 FIFO control register	W	0x00	13.3.1.6/13-10
0x502	UAFR—ULCR[DLAB] = 1 UART0 alternate function register	R/W	0x00	13.3.1.12/13-16
0x503	ULCR—ULCR[DLAB] = x UART0 line control register	R/W	0x00	13.3.1.7/13-11
0x504	UMCR—ULCR[DLAB] = x UART0 modem control register	R/W	0x00	13.3.1.8/13-13
0x505	ULSR—ULCR[DLAB] = x UART0 line status register	R	0x60	13.3.1.9/13-14
0x506	UMSR—ULCR[DLAB] = x UART0 modem status register	R	0x00	13.3.1.10/13-15
0x507	USCR—ULCR[DLAB] = x UART0 scratch register	R/W	0x00	13.3.1.11/13-16
0x510	UDSR—ULCR[DLAB] = x UART0 DMA status register	R	0x01	13.3.1.13/13-17

Figure 4. The MPC8572 UART Configuration Registers [8]

5.2 TEST SETUP.

The program was run through the CodeWarrior™ integrated development environment by loading it into the memory of the hardware platform. The program used to change the register bits was also used to test the UART functions. This was done by printing the register and bit number each time a register bit was changed.

5.3 EXPERIMENTS.

The register bits were changed by XORing them with a mask in a walking one pattern. After printing the register and bit numbers, the bit was reset to its original value before changing the next bit.

5.4 RESULTS.

Table 1 shows the results of the experiment. The table only shows the bit and register numbers for which there was a deviation from the normal execution.

Table 1. Results of Experiment

Serial No.	Register No.	Bit No.	Register Name	Bit Description	Effects
1	0	3	UTHR0	Data	Newline character(s) deleted
2	0	5	UTHR0	Data	Extraneous space character inserted
3	0	6	UTHR0	Data	Extraneous @ character inserted
4	2	1	UFCR0	Receiver trigger level	Extra character repeated
5	2	2	UFCR0	Reserved	Extra character repeated
6	2	7	UFCR0	FIFO enable	Extra non-ASCII characters on reset
7	3	1	ULCR0	Set break	Some characters get translated to non-ACSII characters
8	3	6	ULCR0	Word length	Test hangs
9	3	7	ULCR0	Word length	Test hangs

It should be noted that, in each case, the program returned to the normal mode of operation after resetting the changed bit. The results can be classified based on their degree of criticality:

- Change in output data: The data output on the console was different from normal output. (For example, serial numbers 1-7)
- Change in program execution (Serial numbers 8 and 9)
- Crash: None in this experiment.

5.5 INTERPRETATION OF RESULTS.

For serial numbers 8 and 9 in table 1, a special diagnostic was required since the program did not execute normally. Using CodeWarrior debugger, it was observed that, for serial number 8, the execution did not come out of the MPCDUARTReadPool() function. The program did not exactly stop executing, but it seemed to be stuck in the function. The set break forced logic 0 to be on the serial out line and did not affect the UART buffers. In such a situation, the UART buffers get filled up, and hence, the call to printf() in the test program does not return.

5.6 A SIMPLE SAFETY NET IMPLEMENTATION FOR CONFIGURATION-RELATED ISSUES.

In this experiment, a safety net design was tested for the configuration-related issues. In an earlier test, it was observed that some UART configuration registers can lead to unwanted effects when their values are changed. It was also observed that the correct values of these registers are known for a particular use case. In this experiment, the UART configuration registers were periodically overwritten with their correct values. It was observed that unwanted effects still

exist when overwriting with correct values. The period at which the system was able to perform correctly was observed, even when the register values changed.

5.7 TEST SETUP.

The test was performed inside the CodeWarrior debug environment on the hardware platform. A UART0 line control register was used to corrupt and repair. The register was corrupted by changing its lowest bit, which changes the data to be transmitted on the UART. This data translation led to the output of non-ASCII characters on the serial port. To repair the corrupted register, the default correct value of 0x03 was written onto the register. To detect errors, the value of the register was confirmed to be 0x03. The corruption, repair, and usage were all run within a single thread on one core.

5.8 TIMER SETUP.

The decrementer (DEC) counter in the e500 cores was used to set up a 1-ms timer. The DEC counter is decremented every 8 core complex bus (CCB) clocks. In the default setup, the CCB runs at a 600-MHz frequency, so the DEC was loaded with a value of 750,000. The interrupt handler for the timer calls the timerInt function where the register corruption, repair, and usage are performed. The pseudo code for the timerInt function is shown in figure 5.

```
time++
if (time % usage_period == 0)
        if(register value incorrect)
                repairs++
if (time % repair_period == 0)
        repair register()
if (time == next_error_time)
        corrupt register()
        update next_error_time
```

Figure 5. The timerInt Function

5.9 EXPERIMENTS.

Three different periods were used for using, corrupting, and repairing the register value. The usage period and corruption period are estimates on how frequently the register is expected to be used and corrupted, respectively. Experiments were conducted for different values of the repair period to observe the number of errors. For each repair period, the test was run for 10 seconds.

5.10 TEST RESULTS.

This section discusses the results from the experiments performed for this test plan. Figures 6 through 8 show the number of errors for different values of repair, usage, and corruption periods. In all three results, no errors were detected if the repair period was less than 5 ms. At the same time, for certain values of the repair period, there was a spike in the number of errors. This happened because the interleaving of usage, repair, and corruption periods allowed the register to be used just after it is corrupted for the given corruption period. The usage period was kept constant at 15 ms.

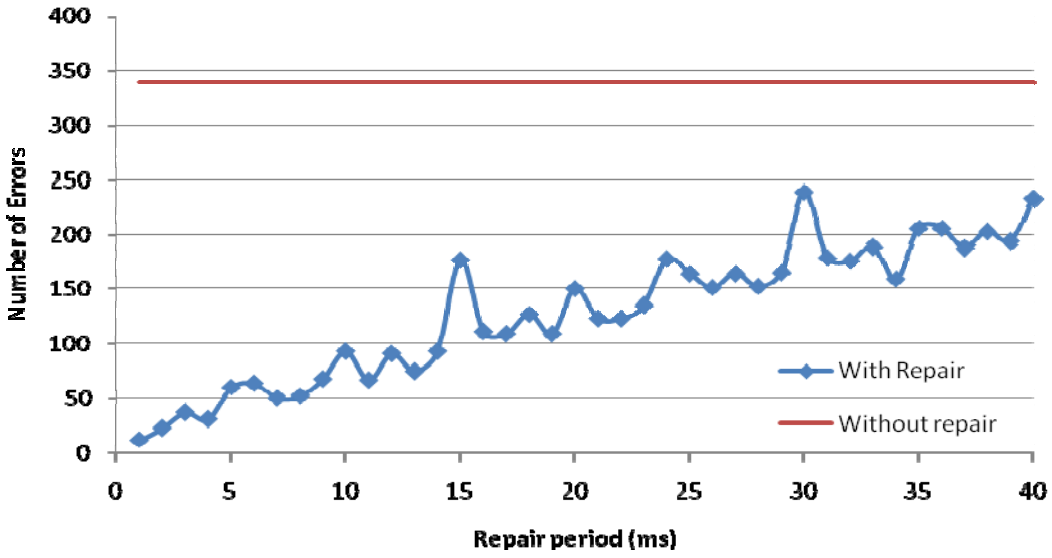


Figure 6. Corruption Interval = 100 ms

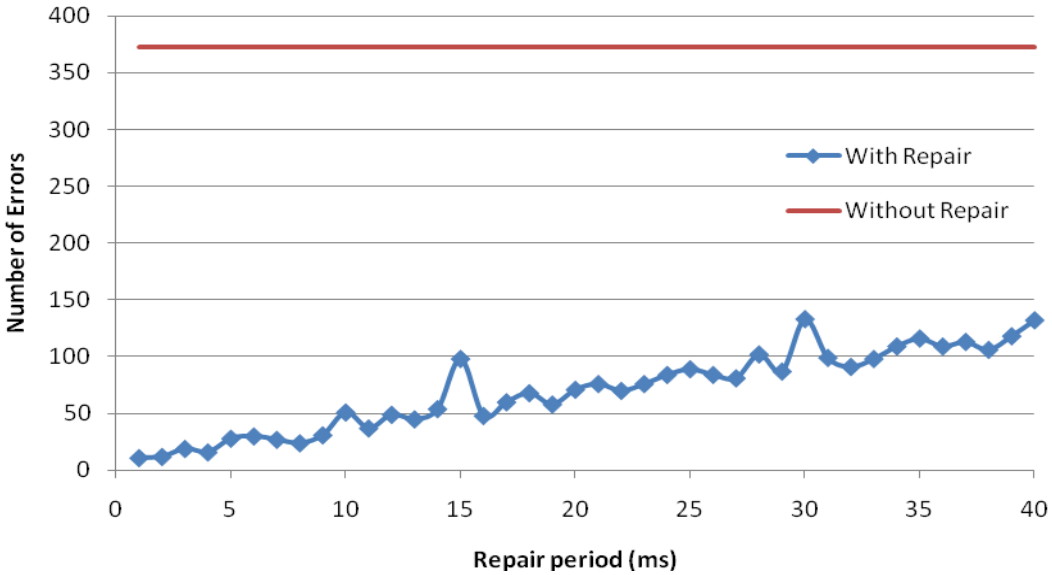


Figure 7. Corruption Period = 200 ms

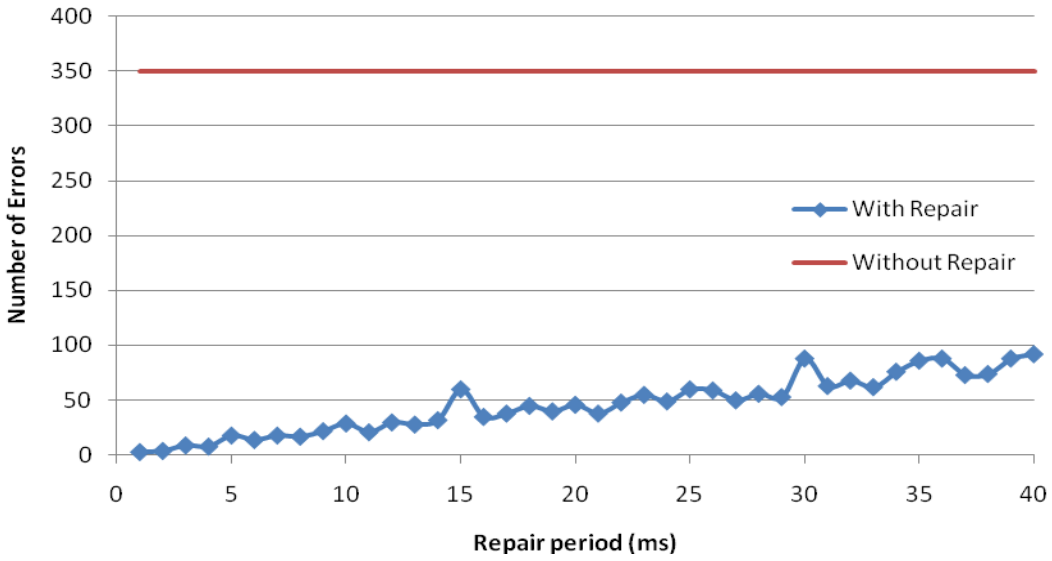


Figure 8. Corruption Period = 300 ms

Through these experiments, it was found that the number of detected errors depends greatly on the usage, corruption, and repair periods. However, these validation experiments indicate that if the configuration registers are repaired at a sufficiently high rate, the failure impact can be significantly reduced. The effectiveness of this reduction in impact can be affected by the corruption period, the specific configuration register change(s), and the design of the affected functions.

6. RESOURCE-SHARING CONSIDERATIONS.

This sections details the experiments and results performed to gain insight into the effects of resource sharing in a COTS microprocessor system.

6.1 EXPERIMENTS, TEST SETUP, AND RESULTS.

In this experiment, the timing delays, which were due to the contention by cores on L2 cache, were determined. First, a simple matrix multiplication program was used to determine its execution time by running it on only one core. This execution time was the baseline timing requirement. To test the effect of contention, the other core was activated and run another matrix multiplication program, which stressed the shared L2 cache. The delays incurred by this contention on the execution time of the matrix multiplication program were observed and measured.

6.1.1 Test Setup.

The test was performed inside the Linux operating system environment from the board support package for the hardware platform MPC8572DS development system. The taskset utility from the DENX ELDK software package was used to restrict the programs to run on a single core [14]. The execution time of the programs was obtained by reading the real-time clock device set

to a frequency of 8 MHz. The specific details about reading the real-time clock and running the programs can be obtained from the documentation in the code package for the one page test plan (Resource-Sharing Effect on Timing One Page Test Plan). Two versions of the matrix multiplication program were implemented. The first, *matmult_timer*, prints out the execution time whenever it is run. The second, *matmult*, is used for the L2 cache-stressing purposes and does not print out the execution time.

6.1.2 Experiments.

In the following sections, the timed and untimed versions of the matrix multiplication program are referred to as base matrix and contention matrix, respectively. Two different experiments were performed to test the results of contention. In the first experiment, the contention matrix size was kept constant at 10,000, while the base matrix size was varied from 20 to 700. In the second set of experiments, the base matrix size was kept constant, while the contention matrix sizes were varied.

6.1.3 Test Results.

Figure 9 shows the execution times for the first experiment where the base matrix size was varied. It is clear that the execution times in the presence of contention are greater than those without contention. The trend of the execution times is a bit unexpected though. As shown in figure 9, spikes trend at matrix sizes of 510, 590, and 670. The cause(s) of these spikes are probably the dynamic management of cache by the operating system (OS) and needs more investigation.

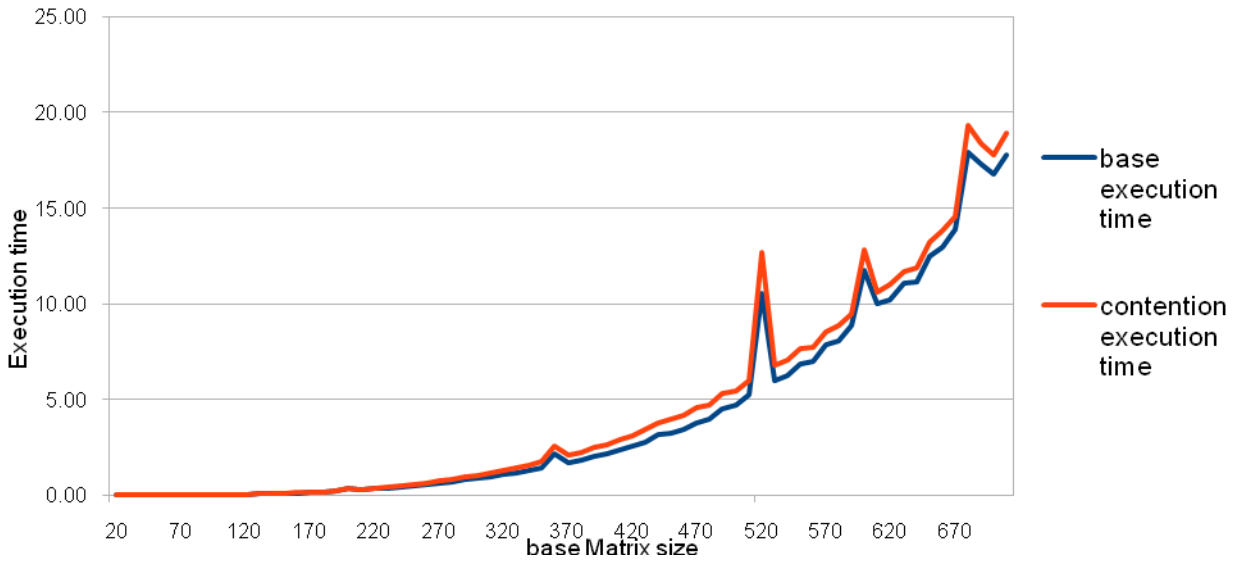


Figure 9. Effect of Contention on Execution Time

Figures 10 through 14 show the execution times for the second experiment where the base matrix sizes were kept constant. Five different base matrix sizes (200 to 600) were used for the experiment. For each case, the contention matrix size was varied from a low number (around

20) to a size roughly equal to double the size of the base matrix. The reason for choosing this range is that a small size contention matrix would provide very little stress on the L2 cache, and the execution time with contention would be roughly the same as without contention. This is clearly visible in the results. A contention matrix size of double the base matrix size ensures there is some contention on the L2 cache. As the contention matrix size is gradually increased beyond a certain value, the execution time with contention may suffer a significant increase, as shown in figures 10 through 14. But as the contention matrix size continues to increase, the execution time does not continue to increase. This again might be a result of the Linux OS performing intelligent reconfiguration of the cache that masks the effect of increase in contention.

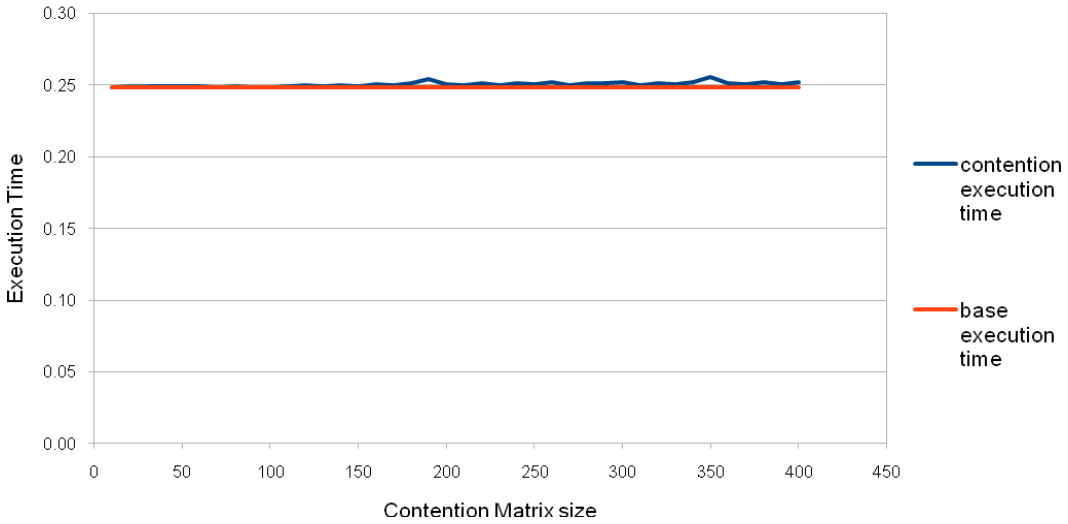


Figure 10. Base Matrix Size = 200

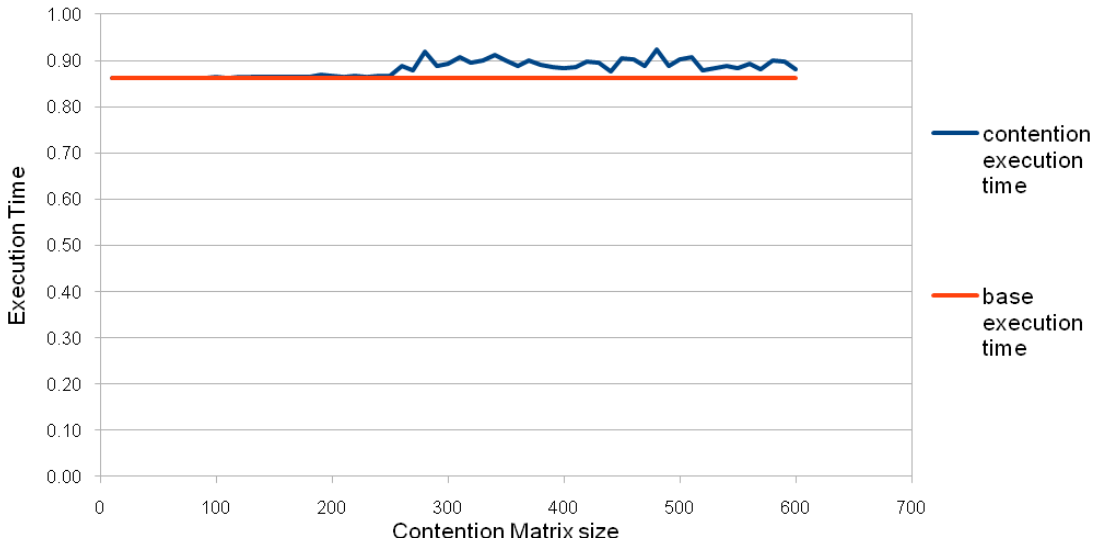


Figure 11. Base Matrix Size = 300

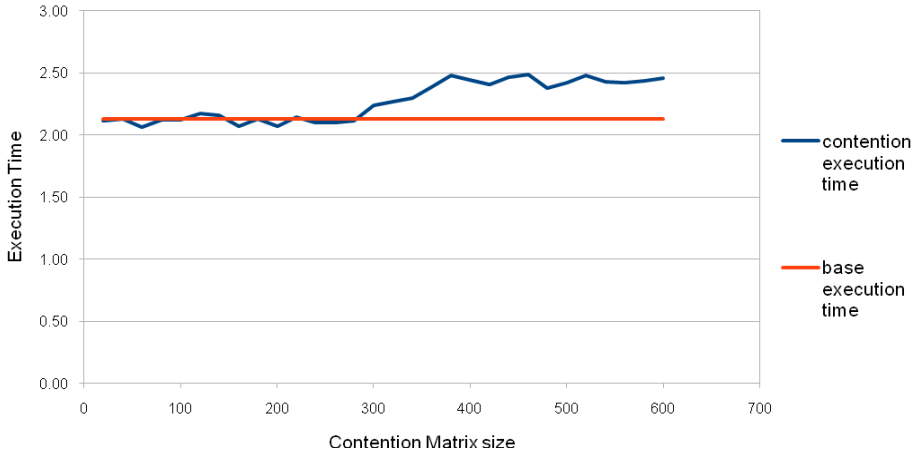


Figure 12. Base Matrix Size = 400

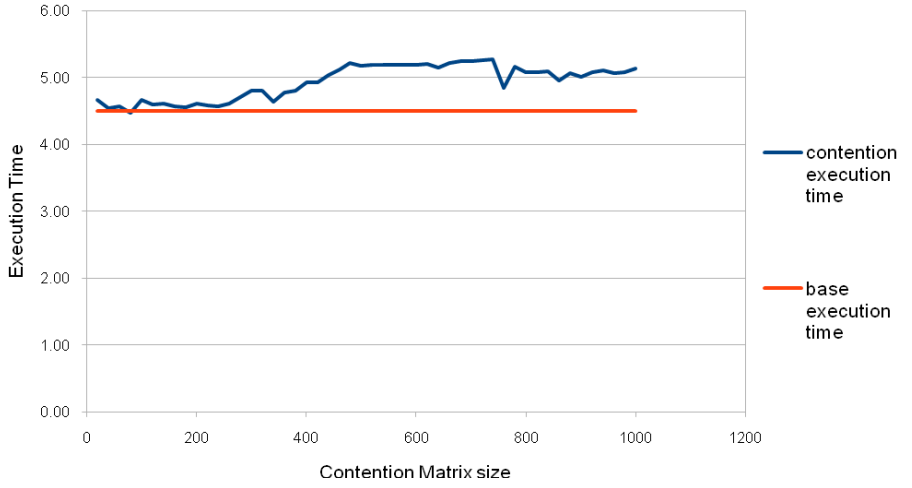


Figure 13. Base Matrix Size = 500

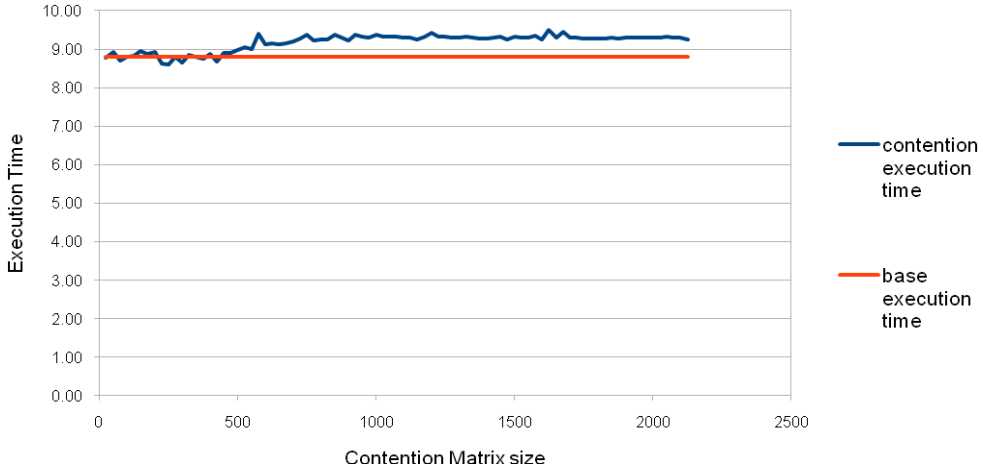


Figure 14. Base Matrix Size = 600

6.2 SUMMARY.

This experiment shows that resource sharing may lead to variation in task execution times. At the same time, the OS may perform intelligent dynamic management, which can reduce the effect of such contention. The effects of contention due to resource sharing are highly dependent on the system setup, and each system setup must be individually evaluated for the effects of resource sharing.

7. SAFETY NETS.

This section discusses safety net considerations that are not present in the Handbook [1]. It is recommended that readers first consult the Handbook for the primary considerations in understanding and building safety nets before reading this section. The following supplementary considerations are addressed here:

- The role of safety nets in ensuring predictable degraded system performance during failure
- System start-up behavior considerations when designing safety nets
- Distinguishing appropriate safety net actions, depending on aircraft conditions

To ensure a comprehensive safety net design, the following activities are needed:

- Identification of the critical functions within the system
- Identification of the components within the system that are used either directly or indirectly by the critical functions
- Identification of operational constraints for use of these components
- Evaluation of the impact to system behavior if the system is subjected to operation outside the constraints. (This includes, but is not limited to, the possible failure modes of the devices used to construct the system.)
- Identification of interfaces that can be connected to those components, both intentionally, and as a result of component failure modes, temporal or logical errors, and operation outside the constraints
- Identification of mechanisms for detecting, classifying (according to criticality of hosted functions, severity of degradation, and other factors), and responding to (recording, reporting, or taking actions) system degradation when it is detected

- Identification of the latency requirements for the response, based upon the classification of the severity of the degradation, the impact it has on critical functions, and the effect the passage of time has to either exacerbate or mediate the effects of the degradation/failure or enhance or dilute the effects of the subsequent remedial actions

Also, the following activities are needed for system design:

- Minimize the probability of undetected degradation/failure of components in critical paths.

- Minimize the probability of incorrect classification of the degradation/failure severity present in those paths.

- Minimize probability of incorrect response to detected degradations/failures present.

- Ensure that the appropriate amount of resources are allocated to mediate the effects of the degradation/failure and enhance the effects of the subsequent remedial actions in critical functions.

- Test to ensure the mediation allows the aircraft to function with the failure inserted (when possible).

The remainder of this section describes the general logic behind a safety net design and a detailed analysis of the different system conditions a designer should consider following that general logic.

The basic logic begins with the system operating within the originally conceived design constraints. The system either experiences an event that subjects it to an environment it was not designed to withstand, or the system sustains an internal failure in one (single fault) or more (cascade failure) components that are either frequently or infrequently used by the system. At this point, portions of the system may exhibit behavior ranging from being totally operational (no degradation) to being totally degraded (failed). The degradations or failure behaviors may be detectable immediately or they may only be detectible when a certain function or combination of conditions is exercised (latent). The kinds of behavior the system produces as a result of the event and degradation can range from inputs or outputs not being present when required, to inputs or outputs being present when not required. Multiple threads of these behaviors may exist within portions of the system that can be expressed as above. The logic can be applied to components in the path of critical functions as well as in the path of noncritical functions.

A safety net may perform a combination of periodic built-in test (PBIT) and software monitoring to detect the occurrence of a subset of possible constraint violations and internal failures. As stated previously, the capabilities of PBIT and software monitoring depend on the specific microprocessor capabilities.

Upon occurrence of an internal failure or design constraint violation, the system may transition to one of the following states that must be accounted for in the safety net design:

- The system may continue to operate normally.

- The system may totally fail to operate (go offline).

- The system may enter one of the following partially operational, but functionally degraded, states:

  - One or more critical functions are lost before conditions exist that require the operation of any of the lost or affected critical functions. One or more observable effects are produced when the state is entered.

  - One or more critical functions are lost during or after the conditions exist for which the operation of the affected critical function is required. One or more observable effects are produced when the state is entered.

The observed effects for the above cases are due to either the loss of observable critical or noncritical functions or injections of unintended operational conditions into observable critical or noncritical functions that are apparent upon state entry.

There are two cases where functionality is lost but no observable effects occur:

- One or more noncritical functions are lost before conditions exist that require the operation of any of the affected noncritical functions. No observable effects are produced when the state is entered.

- One or more critical functions are lost before conditions exist that require the operation of any of the affected critical functions. No observable effects are produced when the state is entered.

The presence of degraded operation with the lack of observed effects for these two cases is due to either the loss of unexercised functions or injections of unintended operational conditions into unexercised functions that are not apparent upon state entry. The loss or losses for the above two cases become apparent when one of the affected functions is required.

The observed effects for the following four cases are due to injections of unintended operational conditions into observable critical or noncritical functions that are apparent upon state entry:

- Injection of one or more unintended operational conditions into one or more critical functions before conditions exist that require the operation of any of the affected critical functions. One or more observable effects are produced when the state is entered.

- Injection of one or more unintended operational conditions into one or more critical functions during or after conditions exist that require the operation of any of the affected critical functions. One or more observable effects are produced when the state is entered.

- Injection of one or more unintended operational conditions into one or more noncritical functions before conditions exist that require the operation of any of the affected noncritical functions. One or more observable effects are produced when the state is entered.

- Injection of one or more unintended operational conditions into one or more noncritical functions during or after conditions exist that require the operation of any of the affected noncritical functions. One or more observable effects are produced when the state is entered.

The presence of degraded operation with the lack of observed effects for the following two cases is due to the injection of unintended operational conditions into unexercised functions that are not apparent upon state entry. The loss or losses for the above two cases become apparent when one of the affected functions is required.

- Injection of one or more unintended operational conditions into one or more critical functions before conditions exist that require the operation of any of the affected critical functions. No observable effects are produced when the state is entered.

- Injection of one or more unintended operational conditions into one or more noncritical functions. No observable effects are produced when the state is entered.

7.1 SYSTEM START-UP BEHAVIOR CONSIDERATIONS TO ADDRESS IN DESIGNING SAFETY NETS.

The actions performed during power application depend on the operational state of the system, the hardware and software architecture, and other factors. The following are representative examples of conditions and operations that should be considered when designing safety nets.

Maintenance start-up on ground includes the following operations:

- Maximum coverage power-on self-test (POST)

- Maximum initialization of processing hardware

- Complete initialization and test of memory subsystems. As memory is added to a system, the time needed to verify its integrity is increased. The execution speed of the processor is not the constraining factor for these tests.

- Initialization of discrete and networked I/O. As I/O is added to a system, the time needed to verify its integrity is increased. The execution speed of the processor is probably not the constraining factor for these tests.

- Run-time executive load and execution (including PBIT)

- Application software load and execution (including application software monitors)

- Ground-start event detection, processing, and optional notification

- Preparation and execution of additional diagnostic testing with attached test equipment

Normal start-up on ground consists of the following operations:

- Normal coverage POST
- Initialization of processing hardware
- Complete initialization and test of memory subsystems
- Initialization of discrete and networked I/O
- Run-time executive load and execution (including PBIT)
- Application software load and execution (including application software monitors)
- Ground-start event detection, processing, and optional notification

Ground-restart consists of the following operations:

- Normal coverage POST
- Initialization of processing hardware
- Complete initialization and test of memory subsystems
- Initialization of discrete and networked I/O
- Run-time executive load and execution (including PBIT)
- Application software load and execution (including application software monitors)
- Ground-start event detection, processing, and optional notification

Air-restart consists of the following operations:

- Time-constrained abbreviated POST

- Time-constrained abbreviated initialization of processing hardware, possibly using last applicable configuration data if it is available

- Time constrained, minimally disruptive initialization and test of memory subsystems

- Time-constrained abbreviated Initialization of discrete and networked I/O possibly using last applicable configuration data if it is available

- Time-constrained abbreviated run-time executive load

- Time-constrained abbreviated application software load

- Time-constrained application software execution in-air-restart event detection, processing, and notification

- Air-start event detection, processing, and optional notification

7.2 CRITICAL CONDITION DETERMINATION CONSIDERATIONS FOR SAFETY NET DESIGN.

Detection of critical conditions is usually accomplished by reading discrete inputs of the processing system, just after the power-up initialization and critical self-test functions have completed.

A pair of critical conditions is associated with application and removal of power. These rely on a reserve amount of power present in the system to accomplish the storage of data from the system during power interruptions, and subsequent retrieval of that data to resume operation once power has been restored. The following are examples of critical conditions and the associated design considerations that could change based on the system functions, system architecture, aircraft power sources, and aircraft interfaces.

- Example 1: Power Down Imminent Conditions

  Power down imminent indicates that the power conditions needed to maintain system operation will soon be lost, and that critical data and state information needed to resume operation when power is subsequently reapplied need to be saved. The amount of information saved may be a function of the state the system is in when this condition was entered.

- Example 2: Power Up Conditions

  Power up indicates that the power conditions needed to establish and maintain system operation now exist, and previously saved critical data and state information need to be restored, if they exist. The amount of information restored may be a function of the state the system was in when the power down imminent condition was entered.

The next pair of critical conditions are associated with the determination of on-ground versus airborne conditions.

- Example 3: On-Ground Conditions

  The presence of an on-ground condition is indicated by the presence of the weight on wheels discrete.

It inhibits the operation of signals and software execution for avionics and release systems during tests that would be hazardous to aircrews, ground crews, or others; unless intentionally overridden to exercise those signals as part of the test.

It enables the operation of signals and software execution (such as special fault isolation software) that would not be permitted in an airborne environment because they consume an excessive amount of computational resources needed to maintain safe flight or otherwise present a hazard to aircrew or others.

Detection of this condition after power-up indirectly indicates an on-ground power cycle.

Absolute determination of the on-ground condition requires additional input from other sources, such as the presence of the landing gear down and locked input. This condition cannot be achieved while in the air, except in the event of landing or through failure that asserts the required discrete inputs.

- Example 4: Airborne Condition

  The absence of an on-ground condition is indicated by the absence of the weight-on-wheels discrete.

  It enables the operation of signals and software execution for avionics and release systems during tests that would be hazardous to aircrews, ground crews, or others if they were indiscriminately executed on the ground.

  It inhibits the operation of signals and software execution (such as special fault isolation software) that would not be permitted in an airborne environment because they consume an excessive amount of computational resources needed to maintain safe flight or otherwise present a hazard to aircrew or others.

  Detection of airborne condition after power-up indirectly indicates an in-air power cycle. Absolute determination of the condition requires additional input from other sources, such as the absence of the landing gear down and locked input. This condition can be achieved on the ground, either intentionally as described above, in the event of take-off, or through failure that removes the required discrete inputs.

8. RECOMMENDATIONS FOR FUTURE RESEARCH.

The FAA has suggested a follow-on Aerospace Vehicle System Institute project to evaluate other AEH beyond COTS microprocessors and an update to the Handbook to reflect the results of the follow-on project. AEH considered in the follow-on project are expected to include FPGA-based SoCs.

Additionally, the follow-on project proposes to guide one or more pilot projects in the use of safety nets for microprocessor-based aircraft systems. Pilot projects in the continuing development, application, and refinement of the safety net approach could provide additional

content and definition to the Handbook in the form of examples and real-world analyses of the effectiveness of safety net approaches.

It is the FAA's intention to assess the use of safety nets in future microprocessor-based aircraft systems. The use of the safety net approach resulting in certification of aircraft containing systems with safety nets can lead to enhancements in FAA policy and guidance to formally accept, implement, and provide guidance for the continued use of safety nets.

Additional research should

- investigate additional functionality that can be accomplished within safety nets.
- investigate new trends in microprocessor design that may aid or hinder the implementation of safety nets.
- investigate architectural and functional requirements for the safety net monitoring itself.
- investigate the implementation of the architectural safety net examples identified in section 7.2.

The initial use of safety nets will be based on a tradeoff between the cost, difficulty, and feasibility of testing and validating the safety of complex nondeterministic SoCs versus the development of the innovative multilevel safety net approach to ensuring system safety characteristics within the operational environment. Both sides of this tradeoff will be very difficult to determine and quantify in advance. It may well be the development of additional uses for safety nets that justify their initial development.

The following sections give a brief description of specific research directions that may be valuable for future designs and regulatory focus. The first research direction, the study of virtualization in future airborne electronic hardware, covers a new trend in microprocessor design that is expected to become prevalent in a broad variety of embedded systems. The second research direction, COTS microprocessor security, is a complementary aspect of the AFE43 research, since safety and security issues typically originate from a common set of device vulnerabilities or risks.

8.1 THE EMERGENCE OF VIRTUALIZATION IN COTS MICROPROCESSORS.

The rise of virtualization features is an emerging trend in embedded COTS microprocessors and SoCs. In this context, virtualization is defined as the ability to separate an operating system from the hardware resources it manages. Instead of giving an operating system direct access to hardware, the operating system communicates with another piece of software, called a hypervisor, which is responsible for hardware management. By doing this, it is possible to run multiple, and possibly different, operating systems on a single processor, and it is also possible to strongly partition multicore resources within an SoC between operating systems. Virtualization is not a new technology, and it has seen widespread use in the enterprise market

due to performance and efficiency benefits. However, virtualization is still a relatively new trend in the embedded market.

Major instruction set architectures (ISA) have adopted virtualization extensions, including Intel's VT-x ISA and the Power ISA currently used by Freescale multicore SoC [15 and 16]. These ISA extensions incorporate hardware support for virtualization for both high performance and system dependability. Hardware support includes the addition of virtualization registers, extended page tables, and I/O memory management units [15, 16, and 17].

Virtualization in airborne systems provides some potential benefits that warrant further study, including

- the ability to run legacy and proprietary operating systems alongside newer or industry-standard operating systems within a single system, reducing the cost of application development or modification due to operating system or library dependencies.

- increased system stability and security [17].

- more hardware support for spacial and temporal separation of applications.

8.2 THE COTS MICROPROCESSOR SECURITY RESEARCH.

COTS microprocessor and SoC security issues are another largely open area of research that has not been covered by this project. While evaluating the three common risk areas of COTS microprocessors and SoCs, safety and security issues often overlapped; however, the security aspect of risk was outside the scope of the Handbook [1].

The most obvious security issues arise from the architectural trend of COTS SoCs to contain many shared resources within a single device. Specifically, some COTS SoCs may lack the resource controls necessary to ensure that different applications share a resource, such as an Ethernet controller, fairly. The virtualization trend, as described in the previous section, can provide possible solutions to these shared resource security issues.

As an example of security issues that arise from shared resources, Moscibroda, et al. [11], describe that in a multicore system, multiple programs running on different cores can interfere with each other's memory access requests, thereby adversely affecting performance. They show that a malicious program running on one core of an SoC can be written to exploit the access policy of the system's DDR memory controller. This malicious program can create a DoS attack on applications running on other cores, due to the inherent unfairness in memory controller access policy. This DoS attack essentially starves other applications of access to the memory controller, and the performance of a starved application can be reduced by as much as 2.9 times in a typical dual-core system.

Possible research in this area can include:

- the discovery and analysis of other shared resources that can lead to security risks
- temperature-based attacks that are worsened by continued technology shrinking [18]
- the development of security reports and papers that can lead to comprehensive policy and guidance on COTS microprocessor security considerations

9. SUMMARY.

Microprocessors and SoCs have become extremely complex, highly integrated, nondeterministic, and densely packaged. Recent changes in COTS microprocessors can be characterized as both physical and functional changes. Physically, transistor density has continued its exponential increase, allowing for hundreds of millions to billions of transistors to be placed on a single device. As of 2010, 65- and 45-nm devices were common in the COTS marketplace, and 32-nm and smaller devices are beginning to enter the marketplace. In addition to the decrease in device size, the functional capability of COTS devices has expanded. It is no longer necessary for different system components to be implemented as discrete devices. Instead, a single COTS SoC may contain multiple microprocessor cores, I/O devices, memory controllers, and other functionality. As a result, deterministic performance is difficult or impossible to predict in some cases. These devices require additional evaluation methods beyond that identified in current regulatory requirements to achieve the resilience required to meet safety and reliability requirements. The aircraft systems containing these COTS devices may require multilevel safety nets to be designed into them.

Subject matter experts from the FAA and the six industry participants of AFE43 first identified common microprocessor risks and an approach to resolve the growing issues with design assurance and certification; Phase 5 related these risks with mitigation methods associated with the multilevel safety net approach. Phase 5 developed the concepts of a multilevel safety net that established an approach to system design including mechanisms to detect, analyze, and respond to SoC anomalous behavior at higher system levels.

The primary deliverable of the entire 5-phase AFE43 Microprocessor Evaluations Project was a Handbook for the Selection and Evaluation of Microprocessors for Airborne Systems. The Phase 4 Report and this Phase 5 Report specify the research results that support the Handbook. The Handbook advocates a major shift in ensuring the safe use of COTS microprocessors in airborne systems. Most complex hardware, including COTS microprocessors, goes through a process of demonstrating safety through the complete verification of the hardware design. AFE43 has shown that this process is infeasible for some complex, nondeterministic COTS microprocessors. These microprocessors should be assumed as potentially unsafe, and system-level approaches for risk mitigation should be considered such as a safety net.

Current FAA policy and guidance do not directly address the use of COTS microprocessors and SoCs in aircraft systems. However, the existing policy and guidance can be used as a basis from which the Handbook may help provide an applicant with additional information in demonstrating

that their system meets the applicable airworthiness requirements. The purpose of the Handbook is to:

- Document common areas of concerns regarding the use of COTS microprocessors in complex and/or safety critical systems.

- Provide approaches, information, and examples for mitigating the concerns through a safety net

- Extend the research accomplished in AFE43 to example approaches to resilient systems through methods defined in this Handbook under the overarching term safety nets

- Reveal how existing regulatory policy and guidance may be augmented to support the creation of resilient systems through safety net approaches safeguarding the use of microprocessor technologies in complex and/or safety critical systems.

Acceptable implementation of safety nets as a design assurance mechanism and airworthiness determinant will result in the development of procedures, standards, and guidance for the use of safety nets in the certification of aircraft containing avionics systems with embedded complex COTS microprocessors or SoCs such as those described in this report.

10. REFERENCES.

1. Green, B., et al., "Handbook for the Selection and Evaluation of Microprocessors for Airborne Systems," FAA report DOT/FAA/AR-11/2, February 2011.

2. Mahapatra, R.N. and Ahmed, S., "Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No. 43 Phase 1 Report," FAA report DOT/FAA/AR-06/34, December 2006.

3. Mahapatra, R.N., Bhojwani, P.S., and Lee, J.D., "Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No. 43 Phase 2 Report," FAA report DOT/FAA/AR-08/14, June 2008.

4. Mahapatra, R.N., Bhojwani, P.S., and Lee, J.D., "Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No. 43 Phase 3 Report," FAA report DOT/FAA/AR-08/55, February 2009.

5. Mahapatra, R.N., Lee, J.D., Gupta, N., and Manners, B., "Microprocessor Evaluations for Safety-Critical, Real-Time Applications: Authority for Expenditure No. 43 Phase 4 Report," FAA report DOT/FAA/AR-10/21, September 2010.

6. AC 20-152, "RTCA, Inc., Document RTCA/DO-254, Design Assurance Guidance for Airborne Electronic Hardware," Federal Aviation Administration AIR-100, 2005.

7. DO-254, "Design Assurance Guidance for Airborne Electronic Hardware," RTCA, Inc., 2000.

8. "MPC8572E PowerQUICC III Integrated Host Processor Family Reference Manual," Freescale Semiconductor, Rev. 2, May 2008.

9. "PowerPC e500 Core Family Reference Manual," Freescale Semiconductor, Rev. 1, 2005.

10. Pellizoni, R. and Caccamo, M., "Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems," *IEEE Transactions on Computers*, Vol. 59, Issue 3, pp. 400-415.

11. Moscibroda, T. and Multu, O., "Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems," *Proceedings of the 16th USENIX Security Symposium*, 2007, pp. 257-274.

12. "MPC8572 Development System User's Guide," Freescale Semiconductor, Rev. 0, November 2007.

13. Virtutech Simics, http://www.virtutech.com, last visited 9/15/10.

14. DENX ELDK, http://www.denx.de/wiki/DULG/ELDK, last visited 9/15/10.

15. Neiger, Gil, et al., "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization," *Intel Technology Journal*, Vol. 10, Issue 3, 2006.

16. "Power ISA Version 2.06," IBM, 2009.

17. "Hardware and Software Assist in Virtualization," Freescale Semiconductor, Rev. 0, 2009.

18. Joonho, Kong, et al., "On the Thermal Attack in Instruction Caches," *IEEE Transactions on Dependable and Secure Computing*, Vol. 7, No. 2, April-June 2010, pp. 217-223.

APPENDIX A—EXPERIMENTAL PLATFORM TEST CODE

This appendix includes the test code written to generate the data reported in sections 5 and 6 of this report. This code was executed on both the physical and simulated platforms described in section 4.

A.1 CONFIGURATION-RELATED ISSUES TEST.

This test flips each individual bit of a set of configuration registers that control the Universal Asynchronous Receiver/Transmitter (UART). The registers modified by this test are listed in figure 4 of this report. Before and after each bit flip, the test prints out a statement so that the user can observe if there is any visible effect to the console due to a UART behavior change. After each bit is flipped, it is restored to its original value before the test modifies a subsequent bit.

```
#include <stdio.h>

#define UART0OFFSET 0x04500
#define CCSRBAR 0xe0000000
#define UARTREG(x) (*(volatile char *)(CCSRBAR + UART0OFFSET +(x)))
#define NCHECKED 2
typedef void (IntHndlr)(long);

extern void InterruptHandler(long cause);
asm void system_call();
int checkRegBit(int j,  int i);

asm void system_call()
{
      nofralloc
      sc
      blr
}

/* changing of some configuration register bits can cause this program
to stop.  Hence we have     this function to skip checking of bit
"bitNum" in register number "reg"
 */
int checkRegBit(int bitNum, int reg)
{
      int i;
      int checked[NCHECKED][2] =
      {
            {6,3},
            {7,3}
      };

      return 0;
```

```
	for(i=0; i<NCHECKED; i++)
	{
		if((bit==checked[i][0]) && (reg == checked[i][1]))
			return 1;
	}
	return 0;

}

//Main function

void main()
{
	int i=0,j;
	char mask;
	/*
	Because interrupt handlers contain shared code, each core needs
to register its own
	InterruptHandler routine
	*/
	register IntHndlr* isr = InterruptHandler;
	asm
	{
		mtspr SPRG0, isr
	}

	printf("OPTP-1\r\n");
	for (i=0x00; i<=0x10; i++)
	{
		mask = 1;
		for (j=0; j<8; j++)
		{
			if(checkRegBit(j,i) == 1)
				continue;
			printf("Changing bit %d of register %x.  Current
register value = %x\r\n",j,i, UARTREG(i));
			UARTREG(i) = UARTREG(i) ^ mask;
			printf("Changed bit %d of register %x.  New register
value = %x\r\n",j,i,UARTREG(i));
			UARTREG(i) = UARTREG(i) ^ mask;
			printf("reset bit %d of register %x\r\n",j,i);
			mask = mask << 1;
		}
	}


	system_call(); // generate a system call exception to demonstrate
the ISR

	while (1) { i++; } // loop forever
}
```

A.2 CONFIGURATION REGISTER SAFETY NET TEST.

This test corrupts and repairs configuration registers, and it determines how many errors occurred between corruption and repair.

```
#include <stdio.h>
#include <stdlib.h>

#define PICGCR 0xE0041020
#define UART0OFFSET 0x04500
#define CCSRBAR 0xe0000000
#define UARTREG(x) (*(volatile char *)(CCSRBAR + UART0OFFSET +(x)))
#define NCHECKED 2

typedef void (IntHndlr)(long);

extern void InterruptHandler(long cause);

void timerInt();

int time=0;
char mask=1;
int errors=0;
int regNo=3, bitNo=0;
int repair_period=1;
int read_period = 15;
int corruption_interval=300;
int next_error_time=10;
int repair_errors=0;


//Main function

void main()
{
      int i=0;

      register unsigned int period = 7500;
      register int tmpval = 0x04400000;
      /*
      Because interrupt handlers contain shared code, each core needs
to register its own
      InterruptHandeler routine
      */
      register IntHndlr* isr = InterruptHandler;
      asm
      {
            mtspr SPRG0, isr
      }

      (*(volatile int *)(PICGCR)) = 0x20000000;
      asm
      {
```

```
            mtspr DECAR, period
            mtspr DEC, period
            mtspr TCR, tmpval
            wrteei 0x1;
      }

      srand(1);
      printf("Core 0 time = %d\r\n",time);
      UARTREG(regNo) = 0x03;
      while (1)
      {
            i++;
      } // loop forever
}

//This function gets called on the timer interrupt.

void timerInt()
{
      time++;

      if(time == 10000)
      {
            UARTREG(regNo) = 0x03;
            printf("repair period = %d, Number of errors =
%d\r\n",repair_period, errors);
            repair_period++;
            if(repair_period <= 40)
            {
                  time = 0;
                  next_error_time=10;
                  srand(1);
                  UARTREG(regNo) = 0x03;
            }
            errors=0;
            return;
      }
      if(time > 10000)
      {
            return;
      }

      if(time % read_period == 0)
      {
            if(UARTREG(regNo) != 0x03)
            {
                  //printf("Error at time = %d\r\n",time);
                  errors++;
            }
      }

      if((time % repair_period == 0) && (repair_errors == 1))
      {
```

```
            if(UARTREG(regNo) != 0x03)
            {
                  UARTREG(regNo) = 0x03;
                  //printf("Repairing at time = %d, Number of errors =
%d\r\n", time, errors);
            }
      }


      if(time == next_error_time )
      {
            //printf("Corrupting at time = %d\r\n",time);
            UARTREG(regNo) = UARTREG(regNo) ^ mask;
            next_error_time += 1 + (corruption_interval
*(rand()/(RAND_MAX+1.0)));

      }
}
```

A.3 RESOURCE SHARING TEST.

This test performed matrix multiplication to generate the data reported in section 6. Before and after the matrix multiplication, the MPC8572 system-on-a-chip performance monitors were used to measure several dimensions of program execution, including:

- Level 2 cache misses
- Double data rate memory controller accesses
- Level 2 cache address collisions
- Level 2 cache line victimizations

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>

#define CCSRBAR 0xFFE00000
#define SIZE 1024*1024

void printMat(int **mat, int dim1, int dim2)
{
      int i, j;
      for(i = 0; i < dim1; i++){
            for(j = 0; j < dim2; j++){
                  printf("%d ",mat[i][j]);
            }
            printf("\n");
      }
```

```
}

int main(int argc, char **argv)
{
      int i, j, k, dim1, dim2, dim3, temp1, temp2, temp3, sum;
      int **mat1, **mat2, **matR;

      int fd;
      char *map;
      unsigned int mask, perfreg;

      fd = open("/dev/mem", O_RDWR);
      map = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd,
CCSRBAR);

      if(map == MAP_FAILED)
      {
            perror("Mapping failed\n");
            return 0;
      }
      else
            printf("MAP SUCCESS\n");

//    Set up counting
//      set FAC = 1
//      for each counter
//        set FC = 1
//        set EVENT
//        set FC = 0
//      set FAC = 0

      //Show initial state of PM Global Config
      perfreg = *(unsigned int *) ((unsigned int) map + 0xE1000);
      printf("PMGC0 = %x\n", perfreg);

      // Set FAC = 1
      printf("Setting PMGC0[FAC] = 1\n");
      *(unsigned int *) ((unsigned int) map + 0xE1000) = 0x80000000;

      //Show modified state of PM Global Config
      perfreg = *(unsigned int *) ((unsigned int) map + 0xE1000);
      printf("PMGC0 = %x\n", perfreg);

      // Set up Counter 2:  Instruction fetch -> L2 miss
      // Counter 2:  PMLCA2 = 0xE_1030, PMLCB2 = 0xE_1034, PMC2 =
0xE_1038
      *(unsigned int *) ((unsigned int) map + 0xE1038) = 0x0;
      *(unsigned int *) ((unsigned int) map + 0xE1030) = 0x007B0000;

      // Set up Counter 3:  Instruction fetch -> L2 miss
      *(unsigned int *) ((unsigned int) map + 0xE1048) = 0x0;
      *(unsigned int *) ((unsigned int) map + 0xE1040) = 0x007A0000;

      // Set up Counter 4:  Data Req -> L2 miss
```

```
      // Counter 4:  PMLCA4 = 0xE_1050, PMLCB2 = 0xE_1054, PMC2 =
0xE_1058
      *(unsigned int *) ((unsigned int) map + 0xE1058) = 0x0;
      *(unsigned int *) ((unsigned int) map + 0xE1050) = 0x00790000;

      // Set up Counter 5:  Instruction fetch -> L2 miss
      *(unsigned int *) ((unsigned int) map + 0xE1068) = 0x0;
      *(unsigned int *) ((unsigned int) map + 0xE1060) = 0x00740000;

      // Set up Counter 6:  Instruction fetch -> L2 miss
      *(unsigned int *) ((unsigned int) map + 0xE1078) = 0x0;
      *(unsigned int *) ((unsigned int) map + 0xE1070) = 0x00790000;

      // Set up Counter 7:  Instruction Fetch -> L2 hit
      *(unsigned int *) ((unsigned int) map + 0xE1088) = 0x0;
      *(unsigned int *) ((unsigned int) map + 0xE1080) = 0x00160000;

      // Set up Counter 8:  Data Request -> L2 hit
      *(unsigned int *) ((unsigned int) map + 0xE1098) = 0x0;
      *(unsigned int *) ((unsigned int) map + 0xE1090) = 0x00170000;

      // Set up Counter 9:  Data Request -> L2 hit
      *(unsigned int *) ((unsigned int) map + 0xE10A8) = 0x0;
      *(unsigned int *) ((unsigned int) map + 0xE10A0) = 0x00190000;

      // Set up Counter 9:  Data Request -> L2 hit
      *(unsigned int *) ((unsigned int) map + 0xE10B8) = 0x0;
      *(unsigned int *) ((unsigned int) map + 0xE10B0) = 0x000D0000;

      // Set FAC = 0
      printf("Setting PMGC0[FAC] = 0\n");
      *(unsigned int *) ((unsigned int) map + 0xE1000) = 0x0;

      perfreg = *(unsigned int *) ((unsigned int) map + 0xE1000);
      printf("PMGC0 = %x\n", perfreg);

      perfreg = *(unsigned int *) ((unsigned int) map + 0xE1030);
      printf("PMLCA2 = %x\n", perfreg);

      perfreg = *(unsigned int *) ((unsigned int) map + 0xE1050);
      printf("PMLCA4 = %x\n", perfreg);

      perfreg = *(unsigned int *) ((unsigned int) map + 0xE1080);
      printf("PMLCA7 = %x\n", perfreg);

      perfreg = *(unsigned int *) ((unsigned int) map + 0xE1090);
      printf("PMLCA8 = %x\n", perfreg);

      if(argc != 4){
            printf("Usage:  ./a.out dim1 dim2 dim3\n");
            exit(1);
      }

      srand(time(NULL));
```

```
      dim1 = atoi(argv[1]);
      dim2 = atoi(argv[2]);
      dim3 = atoi(argv[3]);

      printf("dim1 = %d, dim2 = %d, dim3 = %d\n", dim1, dim2, dim3);

      mat1 = (int **) malloc(dim1 * sizeof (int *));
      mat2 = (int **) malloc(dim2 * sizeof (int *));
      matR = (int **) malloc(dim1 * sizeof (int *));

      for (temp1 = 0; temp1 < dim1; temp1++){
            mat1[temp1] = (int *) malloc(dim2 * sizeof(int));
            for (temp2 = 0; temp2 < dim2; temp2++){
                  mat1[temp1][temp2] = rand()%100;
            }
      }

      for (temp1 = 0; temp1 < dim2; temp1++){
            mat2[temp1] = (int *) malloc(dim3 * sizeof(int));
            for (temp2 = 0; temp2 < dim3; temp2++){
                  mat2[temp1][temp2] = rand()%100;
            }
      }

      for (temp1 = 0; temp1 < dim1; temp1++){
            matR[temp1] = (int *) malloc(dim3 * sizeof(int));
      }

      for (temp1 = 0; temp1 < dim1; temp1++){
            for (temp2 = 0; temp2 < dim3; temp2++){
                  sum = 0;
                  for(temp3 = 0; temp3 < dim2; temp3++){
                        sum+= mat1[temp1][temp3] * mat2[temp3][temp2];
                  }
                  matR[temp1][temp2] = sum;
            }
      }

      printf("Matrix Result:\n");
//    printMat(matR, dim1, dim3);


//    Stop, read and clear counters

      // Set FAC = 1
      *(unsigned int *) ((unsigned int) map + 0xE1000) = 0x80000000;
      perfreg = *(unsigned int *) ((unsigned int) map + 0xE1000);
      printf("PMGC0 = %x\n", perfreg);

      printf("Reading Counter 2:  I -> L2 miss\n");
      perfreg = *(unsigned int *) ((unsigned int) map + 0xE1038);
      printf("PMC2 = %d\n", perfreg);

      printf("Reading Counter 3:  L2 addr collision\n");
```

```
        perfreg = *(unsigned int *) ((unsigned int) map + 0xE1048);
        printf("PMC3 = %d\n", perfreg);

        printf("Reading Counter 4:  D -> L2 miss\n");
        perfreg = *(unsigned int *) ((unsigned int) map + 0xE1058);
        printf("PMC4 = %d\n", perfreg);

        printf("Reading Counter 5:  L2 victimize\n");
        perfreg = *(unsigned int *) ((unsigned int) map + 0xE1068);
        printf("PMC5 = %d\n", perfreg);

        printf("Reading Counter 6:  L2 invalidation\n");
        perfreg = *(unsigned int *) ((unsigned int) map + 0xE1078);
        printf("PMC6 = %d\n", perfreg);

        printf("Reading Counter 7:  I -> L2 hit\n");
        perfreg = *(unsigned int *) ((unsigned int) map + 0xE1088);
        printf("PMC7 = %d\n", perfreg);

        printf("Reading Counter 8:  D -> L2 hit\n");
        perfreg = *(unsigned int *) ((unsigned int) map + 0xE1098);
        printf("PMC8 = %d\n", perfreg);

        printf("Reading Counter 9:  L2 alloc\n");
        perfreg = *(unsigned int *) ((unsigned int) map + 0xE10A8);
        printf("PMC9 = %d\n", perfreg);

        printf("Reading Counter 10:  DDR R/W from core\n");
        perfreg = *(unsigned int *) ((unsigned int) map + 0xE10B8);
        printf("PMC10 = %d\n", perfreg);

        return 0;
}
```