Multicore Operational Analysis Tooling

DESIGN DOCUMENT

Senior Design December 2024 – Team 9
Client: The Boeing Company
Advisors: Dr. Steve VanderLeest (Boeing), Dr. Joseph Zambreno (ISU), Dr. Phillip Jones (ISU)

Team Members	***Roles***
Alexander Bashara	Embedded & cache engineer
Joseph Dicklin	I/O engineer
Hankel Haldin	Platform engineer
Anthony Manschula	Project coordinator & memory engineer

Team Email:
sddec24-09@iastate.edu
Team Website:
sddec24-09.sd.ece.iastate.edu

Executive Summary

The increasing computational demand of modern avionics programs necessitates higher performance hardware platforms to support them. Among the various avenues that exist, one approach to achieving higher application performance is to utilize a multicore system. However, incorporating such systems into safety-critical applications like avionics presents a unique set of challenges when it comes to their airworthiness certification. The equipment manufacturer must be able to prove that the system is resilient to performance degradation due to shared resource "crosstalk" (also known as *resource contention*) from applications running on the platform's processor cores. Our team was tasked with building a test framework that would induce sufficient resource contention on a target piece of hardware in order to facilitate more efficient airworthiness testing of embedded Linux systems. Boeing presented our team with several requirements, including processor architecture, system form factor, system resource partitioning approach, and testing framework design. The final design utilizes a hardware platform incorporating the recommended ARMv8 processor architecture, and supports the Xilinx PetaLinux framework, allowing for streamlined system image revisions. The system image includes the Xen hypervisor, which enables the user to partition execution of different programs to distinct processor cores and quantify the effects of resource contention on worst-case program execution time. Our solution includes a front end that allows for efficient collection of execution time metrics across a variety of program types and resource contention methods. The current version of our system meets our clients' needs by providing them with a flexible framework that quantifies the effects of system resource contention on program execution time. Given that our solution is open-source, future developers may wish to make improvements in several areas, including analysis of contention mitigation methods, inclusion of a graphical interface for testing, and refinements to the results analysis tools. Overall, our solution is an important step in the landscape of modern multicore systems analysis, and its open-source nature allows for its continued use and improvement.

Learning Summary

Development Standards & Practices Used

- FAA: AC 20-193
- IEEE Code of Ethics
- CAST-32A
- SAE Aerospace Standards
- RCTA/DO-178C
- ASTM (American Society for Testing and Standards)
- POSIX (Portable Operating System Interface)
- ARINC 653
- FACE (Future Airborne Capability Environment)
- Waterfall & Agile Development Workflow

Summary of Requirements

Physical Requirements

- Architecture: System must implement a processor based on the ARMv8 instruction set
- Form-factor: Single-board computer (Raspberry Pi, Pine64 family, etc.), or FPGA board with Xilinx UltraScale+ MPSoC (Xilinx ZCU family or similar)

User Knowledge Requirements

- Working understanding of Linux environments and how they are structured in the context of embedded systems
- Worst-case execution time and its influencing factors
- Familiarity with multi-core computer architectures, caching, memory, and I/O
- Provide documentation with a sufficient level of detail to allow the user to learn any of the above at a high level

Functional and Technical Requirements

- Toolset must thoroughly and methodically stress the system in a reproducible way
- Toolset must focus on major points of resource contention (processor time, memory usage, IO bus usage, etc.)
- Accurately produce potential worst-case scenarios (rogue process uses too much CPU time/memory/IO bandwidth)
- Toolset must collect and analyze performance data to demonstrate an upper bound on worst-case execution time for our platform

User Interface and Experience Requirements

- Command-line utilities are documented with a level of detail sufficient to allow users with less technical knowledge to use the tool effectively
- Provide a user-friendly GUI for managing and interpreting test results
- Ensure consistent functionality across the GUI and command line tools

Applicable Courses from Iowa State University Curriculum

Computer Engineering

- CPRE 1850 – Introduction to Computer Engineering and Problem Solving in C
- CPRE 2880 – Embedded Systems I: Introduction
- CPRE 3080 – Operating Systems: Principles and Practice
- CPRE 3810 – Computer Organization and Assembly-Level Programming
- CPRE 4580 – Real Time Systems
- CPRE 4880 – Embedded Systems Design
- CPRE 5810 – Computer Systems Architecture

Computer Science

- COMS 3110 – Introduction to the Design and Analysis of Algorithms
- COMS 4150 – Software System Safety

New Skills/Knowledge acquired that was not taught in courses

- Application performance profiling
- Stress-NG
- Multi-core computer architecture
- Virtualization with the Xen hypervisor
- Creating embedded Linux images with PetaLinux

Table of Contents

List of Figures, Tables, and Definitions

Figures and Tables

Important Definitions and Terms

I/O – Input/Output, referring to methods of transferring data into and out of a system.

System-on-Chip (SoC) – Processor, memory, I/O, and graphics processing (sometimes) hardware all housed on a single piece of silicon.

Worst-Case Execution Time (WCET) – Maximum amount of time a program can be expected to take to execute under a set of system conditions designed to produce a worst-case load scenario.

ARMv8 – Version 8 of the ARM Instruction Set, which defines important characteristics of processors that implement it, such as supported data types, register configurations, and memory management.

Main Memory/DRAM – Large pool of reasonably fast volatile (meaning values are erased after power loss) storage that stores instructions and data of programs currently running on the system.

Cache – Small pool of very fast volatile storage that the processor can use to store frequently used data items and program instructions instead of searching for them in main memory. There may be multiple levels of cache in a processor, with some being shared between all cores, and others being private to each core.

Datapath – A path taken by a request from a particular component before reaching its destination component (for instance, the CPU to DRAM data path might be used by a request made by the processor to retrieve a value from main memory).

Virtual Machine (VM)– A software emulation of a computer's physical hardware. High speed VMs (such as the one the team is working with) may require support for this process in hardware through "virtualization hooks".

Type 1 Hypervisor – A hypervisor manages the allocation of physical hardware to all virtual machines running on a system. Type 1 hypervisors have lower resource overhead and require that they are the highest-privileged level of software running on the system (i.e., there is no other software, such as an operating system, operating between it and the physical hardware).

Xen – Open-source Type 1 hypervisor with ARMv8-based system support.

Field Programmable Gate Array (FPGA) – A special integrated circuit that contains reprogrammable logic.

1. Introduction

1.1. Problem Statement

Avionics systems are responsible for controlling many safety-critical aspects of modern airplanes, such as engine management modules and flight envelope protections. As these systems become increasingly computerized, the amount of input data they consume and act upon becomes progressively larger, thereby increasing the necessary amount of computing power avionics systems must possess. While there are multiple avenues to increasing performance in such a system, the approach this project focuses on is the use of a multicore processing platform, due to their relative ubiquity in mainstream computing. However, the use of multicore processing in safety-critical flight systems presents additional challenges over single-core processing in the way of flight certification. Because multicore systems may have several distinct programs running on multiple processing cores, those programs may try to simultaneously access the system's shared resources in such a way that is harmful to the system's performance (known as "resource contention" or "interference"). For such systems to be certified for flight, the manufacturer must be able to show that the system's performance degradation does not fall below a certain bound (i.e., the system may not slow down to a point that endangers the controllability or safety of the plane). The Boeing Company has communicated to us a need for an effective platform that allows for such quantification of performance degradation experienced in multicore systems. Within multicore systems, it is important to test the I/O (input/output) bandwidth, cache interference, CPU (central processing unit) performance and main memory performance under a worst-case set of system load conditions to determine how program execution time is affected. Such a tool is essential to safe operation of multicore systems in safety-critical environments.

This tool allows users to quantitatively identify worst-case execution times (WCET) in which resource contention is extreme. For those who are unfamiliar, worst-case execution time refers to the maximum amount of time a given process can be expected to take under worst-case system conditions. This is essential for avionics certification, as it provides a definitive way to prove that performance degradation is within an acceptable bound, or that mitigations can be applied to move the execution time inside of that bound. Overall, the desired outcome of this project is a stress-testing platform that will allow us to pinpoint areas of resource contention, apply multicore stress via those points, and apply relevant performance mitigations to the system to arrive at a WCET for our reference hardware platform.

1.2. Intended Users

When designing our empathy maps, we were able to identify several key stakeholders/users directly affected by our design project. This includes the Boeing avionics development team working hands on with the multicore stress testing platform, and the Boeing team managers in charge of a technical avionics testing team and ensuring regulatory compliance.

Looking at the user personas for each group, we can see that the avionics development team can be described as a group that manages Linux avionics development and validation for Boeing aircraft. They would need a system stress testing tool developed on an ARM-based platform with performance-related metrics like execution time, resource usages, system temperatures, etc. that is easily accessible. As this user group is directly involved in the testing of in-development avionics, a stress test tool is of the utmost value to the team.

Looking at another user group, the Boeing managers, although not needing a stress testing tool for the success of their role as a managers, they would need a stress tool easily accessible to their project engineers such that they can ensure their systems can be flight-certified under civilian or military authority. Again, as stated by the development team, this tool ensures the success of their in-development product, making the value extremely high for the company. When considering this user group from an economic standpoint, the Boeing manager role consists of client-to-team communication (e.g., economic considerations). Although less important than the safety considerations a product such as this brings up, the economic factors surrounding a stress tool are still important.

2. Requirements, Constraints and Standards

2.1. Requirements and Constraints

Physical/Resource Requirements

- ARMv8 processor subsystem
    - Choosing a hardware platform using a processor implementing the ARMv8 instruction set is critical due to its widespread usage across applications.
- Form-factor: Single-board computer (Raspberry Pi, Pine64 family, etc.) or FPGA board implementing a Xilinx MPSoC (Xilinx ZCU family)
    - The single-board computer form factor is useful in that it keeps costs down and the hardware is easy to find. However, certain FPGA boards are also acceptable if they implement an ARMv8 processor subsystem.

General User Knowledge Requirements

- Linux environments
    - Both the developers and the users must be comfortable working in Linux environments, as this is where tool development and usage will take place.
- Worst-case execution time and its influencing factors
    - Developers and users must be familiar with this concept, as it is the core metric that this project aims to determine. They must understand how both program and hardware architecture may influence it, and how to utilize those factors to produce a thorough measurement.
- Familiarity with multi-core computer architectures
    - To create tools and take measurements to determine WCET for a given multicore system, the developers must understand the data paths between various components in the system, how they interact, and how to abuse them. Additionally, to truly understand the meaning of the results produced by the tools, users must possess some amount of knowledge of the underlying hardware.
- Documentation providing a sufficient level of detail to allow the user to learn any of the above at a high level.

Since our tool will be used by both technical and nontechnical user groups, a comprehensive set of documentation is essential. This will consist of documentation regarding how to build a system image, use the tools, and view/analyze results. Additionally, as we will be handing off the project to another group, clear communication of our progress to them is critical for a smooth transition.

Functional/Technical Requirements

- Properly and methodically stress the system
- Since the toolset applies to verifying hardware for a safety-critical setting, it is important that stress applied to the system is as intense as possible. If it is not, the measurement of WCET may be based on data that were collected with a flawed methodology.
- Identify major points of resource contention (processor time, memory usage, I/O bus usage, etc.).
- The team must consider all possible avenues that resource contention could arise from. Showing evidence of thorough testing is important in being issued an Federal Aviation Administration (FAA) certification.
- Demonstrate an upper bound on worst-case execution time for our platform.
- The tool set must provide an effective way of measuring and analyzing performance metrics of various runs, both with and without the system under stress.

User Interface (UI) Requirements

- Develop a well-documented command line tool to interface with our design
    - The command line tool can be used to interact with and initiate runs with the toolset. Due to inclusion of several base programs and stress options, the team needs clear documentation for acceptable values and the corresponding behavior that they are linked to.
- Provide a user-friendly Graphical User Interface (GUI) for managing and interpreting test results, time providing
    - A GUI is much more friendly to look at than text output from a command line. The GUI should clearly present the results of a set of tests in a straightforward manner.

2.2. ENGINEERING Standards

FAA: AC 20-193

- This standard is defined by the U.S. Department of Transportation. It is concerned with the use of multi-core processors in avionics systems. Our design is directly applicable to this area, hence its inclusion.

IEEE Code of Ethics

- While this standard applies to any engineering effort, our design must ensure the public's safety. Our design provides critical information to systems whose failure could lead to severe injury or death.

CAST-32A

- This document outlines the aspects of multi-core systems of concern to the safety and performance of avionics systems and will help guide the aspects of testing that our toolset needs to achieve.

SAE Aerospace Standards

- These standards define the safety and reliability of various aspects of avionics systems. Our design will stress test multi-core systems that will support avionics systems like controls and communications.

RCTA/DO-178C

- This standard is concerned with the quality of software used in avionics systems. It defines a safety assessment process that categorizes software into five tiers of criticality. Our design must adequately characterize a hardware platform to assess the criticality of a software fault or failure.

ASTM (American Society for Testing and Standards)

- ASTM publishes technical standards that are directly applicable to many engineering efforts, including avionics.

POSIX (Portable Operating System Interface)

- POSIX defines a set of standards that ensure compatibility between operating systems. Our design will be a part of a larger set of software tools and systems. It should therefore be able to interface with these tools in a standardized way.

ARINC 653

- This standard is concerned with the space and time partitioning of safety critical avionics systems. Our design must be able to isolate and test distinct aspects of a multi-core system, like CPU usage, memory, and bus traffic to inform the decisions made by this standard.

FACE (Future Airborne Capability Environment)

- This standard defines an avionics environment for military airborne platforms. It is concerned with making real-time safety-critical computing applications more robust, portable, and secure. This is the end goal of our design.

3. Project Plan

3.1. Project Management/Tracking Procedures

Our project uses a hybrid management style, utilizing both waterfall and agile development strategies. This has allowed us to plan out the entire project broadly, then use agile development strategies to focus on week-to-week goals to meet broad deadlines. To achieve these goals, we are using Git to manage our code and tracking issues using Gitlab Issues. We also participate in weekly status meetings with our client, Boeing, to make sure that we are on the right track and that Boeing is happy with our progress. These strategies allow us to make consistent progress and track where we are relative to our goals.

3.2. Task Decomposition

We divided our project into major parts, then subdivided those parts into smaller parts that could be accomplished by one or two team members. This strategy integrates well with our management and tracking procedures. Figure 1 below details all the tasks that compose our project.

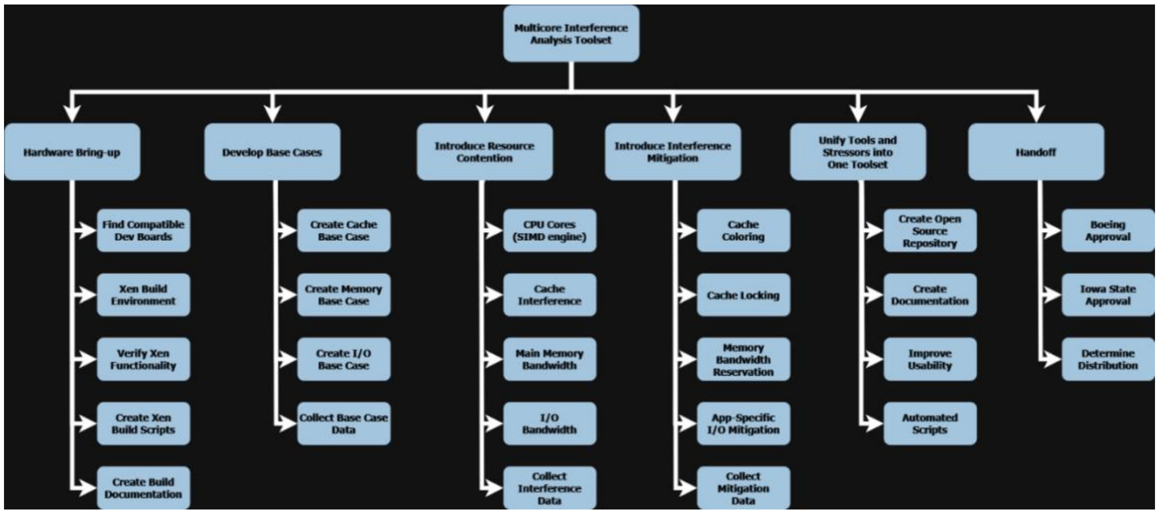


Figure 1: Task Decomposition Chart. Boxes in the second row correspond to a major milestone in our project, with the boxes under corresponding to sub-tasks needed to achieve these milestones. Each sub-task is dependent on the task above it.

3.3. Project Proposed Milestones, Metrics, and Evaluation Criteria

Metrics/Evaluation Criteria:

- Technical metrics:
    - Tool suite generates interference on all aspects of the hardware platform
        - Cache, memory, I/O buses, SIMD engines, etc.
        - Options should be configurable to service multiple different platforms
    - Worst-Case Execution Time (WCET) Criteria
        - Perform several experiments to generate interference under different types of victim programs
        - Use statistical analysis of execution time to determine an upper bound on WCET in various use cases
    - System resource usage
        - Characterize the underlying interference channel causing the WCET
- Usability metrics:

- Users rate appearance and usability of the tool suite > 7 on a scale of [1 – 10]
- Command-line version of the tool suite is as user friendly as GUI frontend

Milestones:

- Xen hypervisor is functional on our target development platform
- Identify resource contention points on our target platform
- Tools induce *some* amount of stress on the identified contention points
- Automatic testing is possible via parsing of configuration files
- Integration of testing and tools into one unified suite and made open source
- Tools thoroughly induce stress on the identified contention points
- Quantitatively prove mitigation tools improve performance of the system while contention is underway
- Obtain a worst-case execution time for our system
- Integration of testing and tools into one unified suite

3.4. Project Timeline/Schedule

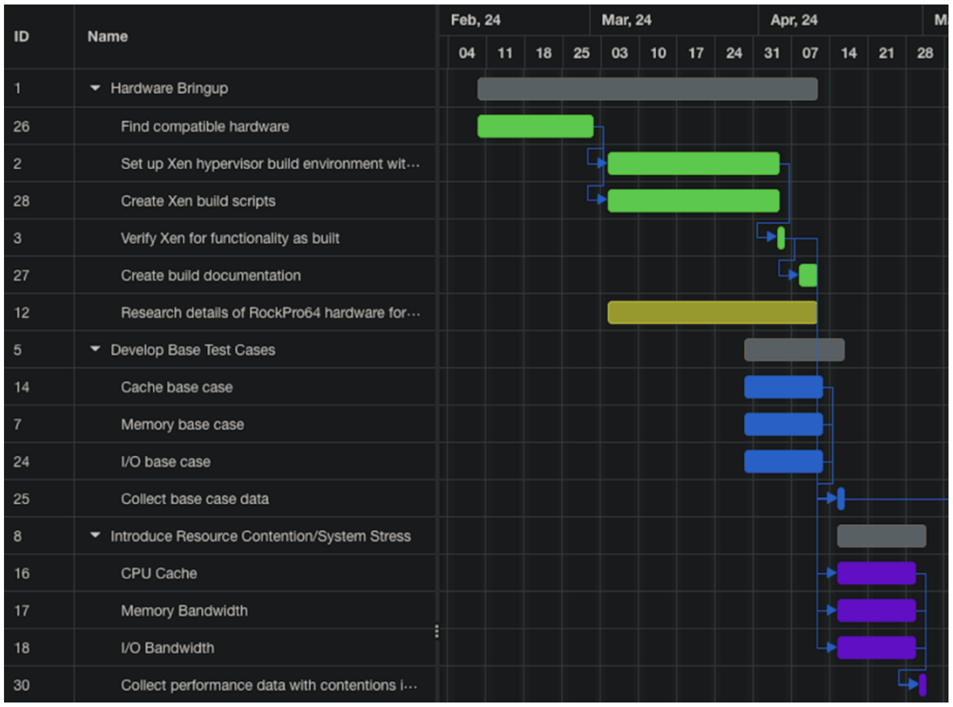

ID	Name
1	Hardware Bringup
26	Find compatible hardware
2	Set up Xen hypervisor build environment wit…
28	Create Xen build scripts
3	Verify Xen for functionality as built
27	Create build documentation
12	Research details of RockPro64 hardware for…
5	Develop Base Test Cases
14	Cache base case
7	Memory base case
24	I/O base case
25	Collect base case data
8	Introduce Resource Contention/System Stress
16	CPU Cache
17	Memory Bandwidth
18	I/O Bandwidth
30	Collect performance data with contentions i…

Figure 2: Semester 1 Project Timeline.

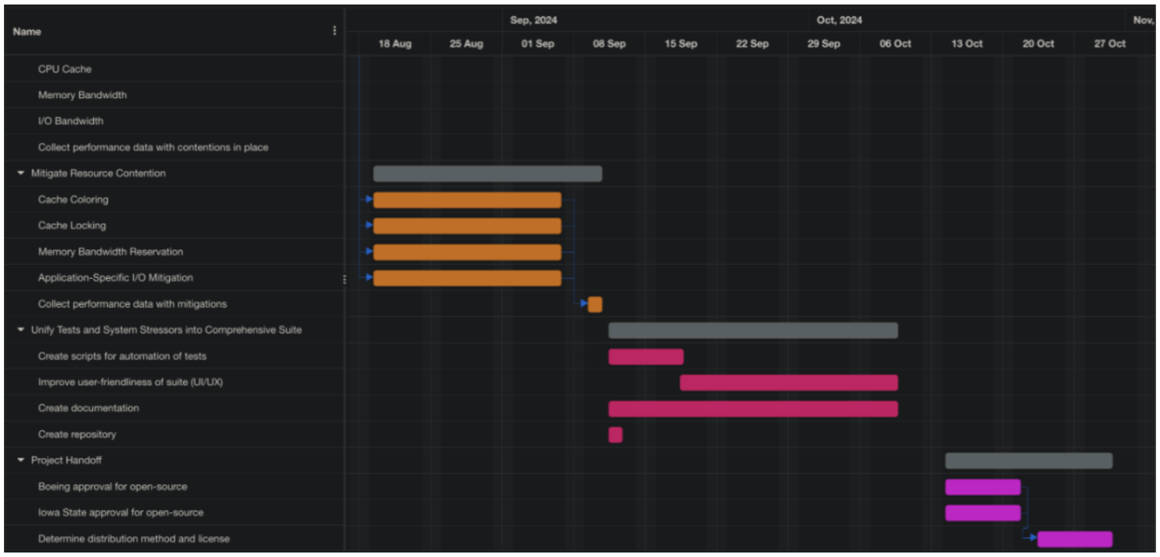


Figure 3: Semester 2 Project Timeline.

The initial project schedule was broken down into several sections. These included hardware bringup, victim program development/research, introduction of resource contention on victim programs, implementation of methods to mitigate resource contentions, development of a test suite for easy and repeatable testing, and finalization of project documentation in preparation for the handoff. Initially, the team budgeted approximately 3-5 weeks for each major section, as we had determined that to be a reasonable estimate given the research we'd done on the problem. However, for reasons discussed in the coming sections, the hardware bring up and introduction of resource contention milestones took significantly longer than the team had anticipated, which led to revision of the team's future milestones.

3.5. Risks and Risk Management/Mitigation

Hardware selection:

- *Find compatible dev board*
  - Risk: we struggle to find a board that meets our project's needs
  - Probability: .80
  - Mitigation: acquire multiple boards (within budget) to ideally find one that works
  - Mitigation: we can emulate a hardware environment in Linux
  - This was a significant issue when the project was in its infancy. Ultimately, we overcame the problem by leveraging an industry standard dev board that was available to us through a research lab but would have otherwise been too expensive for the team to purchase outright. However, the team lost a substantial amount of time working with various boards before we found one that suited our needs.
- *Xen build environment*
  - Risk: we face challenges building Xen & its toolchain for our platform
  - Probability: .75
  - Mitigation: team leverages industry experts at Boeing
  - The team encountered this issue as well during our efforts with our initial hardware, but due to our adoption of an industry-standard dev board, we were able to leverage their tools that made building Xen much more straightforward.
- *Verify Xen functionality*

    - Risk: Xen & and its toolchain do not work after installation
    - Probability: > .90
    - Mitigation: communicate our difficulties to our client to get unstuck early when we encounter issues

- *Create Xen build scripts*
    - Risk: our set up is not easily replicable / portable to a script
    - Probability: < .10

- *Risk: the selected hardware platform is incompatible with our client's and project's needs*
    - Probability: > .80
    - Mitigation: Have multiple hardware options (e.g., RockPro64, RaspberryPi 4, ZCU106)

- *Risk: we encounter trouble installing Xen on our hardware platform*
    - Probability: > .80
    - Mitigation: communicate our work and where we are stuck to the Boeing team to get unstuck

Develop Base Cases:

- *Create Cache Base Case:*
    - Risk: we cannot find all the relevant information for the cache for our given platform
    - Probability: .20 (we specifically chose platforms for which we would have this information)

- *Create Memory Base Case:*
    - Risk: we cannot find all the relevant information for the memory configuration for our given platform
    - Probability: .20 (we specifically chose platforms for which we would have this information)

- *Create I/O Base Case:*
    - Risk: we cannot find all relevant information for the I/O characteristics for our given platform
    - Probability: .20 (we specifically chose platforms for which we would have this information)

- *Collect Base Case Data:*
    - Risk: we have no way to collect relevant metrics on the researched interference channels
    - Probability: .30

Introduce Resource Contention:

- *CPU Cores (SIMD Engine)*
    - Risk: properly implementing the interference generator is more time consuming than originally planned
    - Probability: .60
    - Mitigation: communicate our work and where we are block to the Boeing team to get unstuck

- *Cache Interference*
    - Risk: properly implementing the interference generator is more time consuming than originally planned
    - Probability: .60

    - Mitigation: communicate our work and where we are block to the Boeing team to get unstuck

- *Main Memory Bandwidth*
    - Risk: properly implementing the interference generator is more time consuming than originally planned
    - Probability: .60
    - Mitigation: communicate our work and where we are block to the Boeing team to get unstuck

- *I/O bandwidth*
    - Risk: properly implementing the interference generator is more time consuming than originally planned
    - Probability: .60
    - Mitigation: communicate our work and where we are block to the Boeing team to get unstuck

- *Collect Interference Data*
    - Risk: our test base cases do not adequately stress the system (i.e., demonstrate WCET)
    - Probability: .60
    - Mitigation: we can use our client's expertise in the given domain to increase the likelihood that our test cases demonstrate the WCET for our hardware platform

Unify Tools and Stressors into One Toolset:

- *Create Open-Source Repository*
    - Risk: we are not able to create a public repository due to NDA
    - Probability: > .50 (?)
    - Mitigation: we need to communicate with both Boeing and Iowa State University early on to determine which parts of the project can and cannot be open-sourced

- *Create Documentation*
    - Risk: development of the project was not continuously documented, and knowledge is lost
    - Probability: .60
    - Mitigation: maintain light documentation of work throughout the project so it can be expanded on during this stage

- *Improve Usability*
    - Risk: our tool is not intuitive to use for our user base
    - Probability: .70
    - Mitigation: perform a usability study with our Boeing clients to improve the usability of our project

- *Automated Scripts*
    - Risk: the scripts we produce are not able to be reused by users of the project
    - Probability: .20

Handoff:

- *Boeing Approval*
    - Risk: Boeing does not approve the handoff of our project due to NDA
    - Probability: .30

- *Iowa State Approval*
    - Risk: Iowa State University does not approve the open sourcing of the project
    - Probability: .50

    - Mitigation: Communicate with both Boeing and Iowa State University early on to determine what parts of the project can be open sourced
- *Determine Distribution*
    - Risk: there is no platform we can publish our project on or no license that applies to its distribution
    - Probability: .10

3.6. Personnel Effort Requirements

Using the task decomposition table from 3.2, we separated the major tasks into the rows of the table below with the smaller sub-categories/tasks placed along the columns. Using our best judgement, we assigned rough time estimates for each sub task using the assumption that a single team member or two would be assigned to each task.

Hardware Bring-up	**Develop Base Cases**	**Introduce Resource Contention**	**Unify Tools and Stressors to One Toolset**	**Handoff**
Compatible Dev Boards (24hrs)	Cache Base Case (15hrs)	CPU Cores (25hrs)	Create Open-Source Repository (15hrs)	Boeing Approval (1hr)
Xen Build (120hrs)	Main Memory Base Case (15hrs)	Cache Interference (25hrs)	Create Documentation(60hrs)	Iowa State Approval (1hr)
Verify Xen (50hrs)	I/O Base Case (15hrs)	Main Memory Bandwidth (25hrs)	Improve Usability (60hrs)	Determine Distribution (4hr)
Xen Scripts (30hrs)	Collect Base Case Data (15hrs)	I/O Bandwidth (25hrs)	Automated Scripts (30hrs)	
Build Documents (15hrs)		Collect Data (75hrs)	Troubleshooting (80hrs)	

Figure 4: Personnel effort table along with hour estimations

Total: 725 hours across 4 members

3.7. Other Resource Requirements

The main requirement for this project is an ARM development board that supports Xen Hypervisor; for our project that is a Xilinx ZCU106 development board and an SD Card. To use this tool, the user will also need a computer with a USB Type-A port to connect to the target ZCU106 and have the required software dependencies installed, such as a version of Python >= 3.10 and Pyserial.

4. Design

4.1. Design Context

4.1.1. Broader Context

Our design is situated entirely within the avionics community. Although a multicore stress test tool is usable for other areas of engineering design, our hardware specific requirements and relevant engineering standards narrow down the design scope to avionics. The tool is currently being designed for Boeing, but as our product is being released as an open-source asset that 3rd party companies could use the tool to design safety critical systems. With the growing complexity of avionics platforms, it is imperative that multicore systems become reliable enough to help increase the efficiency of avionic hardware.

Area	Description	Examples
Public health, safety and welfare	Our project affects several stakeholder groups including engineers and other employees of Boeing, as well as the general public that could use products incorporating components that were tested with our solution.	Ideally our testing system contributes to safer software and more advanced performance mitigation techniques in the avionics space, along with a better understanding of execution times in the context of a multicore systems experiencing a high load. If such testing is not thorough enough, loss of property and life, as well as reputation could occur for our stakeholders.
Global, cultural and social	Our project focuses on the verification of multicore systems in safety-critical applications specifically avionics. A project's effectiveness and safety are only as good as the culture of the workplace it is developed in.	Usage of the MOAT solution could contribute to a positive change in safety culture in the workplace where it is being applied by providing a more accessible toolset for safety-critical verification.
Environmental	Our project's primary environmental impacts include energy consumption and resource usage (specifically rare earth metals). Because our tool is merely a software device, the main aera of concern is wasted energy usage.	Realistically, the energy impact is rather low when comparing our tool to the broader avionics system. Nonetheless, the testing software will require some level of energy use to run, in turn affecting energy production factors surrounding the environment.
Economic	To get our testing tool up and running, certain hardware must be purchased. In addition to the specific board for testing, a serial to display adapter is required to properly run/debug the system. Upfront costs are trumped by the reduction in	Although not expensive, buying the hardware to run Xen can be considered an economic consideration. The byproduct, however, is the better development of products from a

	economic factors, such as safety hardware, systems re-design factors, etc.	safety standpoint, which allows for the reduction of backup systems onboard.

Figure 5: Areas of ethical concern

4.1.2. Prior Work/Solutions

Our work developing this project is motivated by our client's research in multicore integrated modular avionics [2]. The overarching use case for our design is a tool suite that supports the work of engineers to ensure safety critical systems react to inputs within a deadline. The final deliverable of our project is a tool-suite that provides an engineering team with relevant data regarding a multicore system's WCET [3]. Our design also bolsters the efficiency of the verification process in that it allows a system designer to partition a multicore system into discrete subcomponents and characterize the Worst-Case Execution Time (WCET) on a case-by-case basis. This reflects the verification process in our client's industry [1]. Ultimately, our efforts are directed towards implementing a framework as outlined by our client's research [4].

There are several products on the market that are similar to our design. They range from professionally developed closed source tools to open-source academic projects. These tools provide us with a frame of reference in the market showing us how our tool suite can address specific needs. Each one of the designs, detailed along with their pros and cons, are shown in figure 6 below.

	Multicore Operational Analysis Tool (MOAT)	RapiDaemon	Multicore Test Harness	OTAWA (Open Tool for Adaptive WCET Analysis)	MASTECS
Pros	Open source, built with modern hardware in mind.	Designed by a team of professional engineers, specifically for compliance testing.	Open source & has good instructions.	Open source, supports several ISAs (e.g., ARM, RISC-V, etc.).	Joint project offering timing analysis software, consulting, and documentation.
Cons	Less polished than competitors due to time restrictions.	Closed source & expensive.	Very old, not built for Xen, development stopped four years ago.	No recent builds, not well-known.	Received funding from EU, so potential issues with portability to North America.

Figure 6: Analysis of market competition

4.1.3. Technical Complexity

Our project approach leverages and aggregates several core computer engineering and embedded systems concepts. One of our project's challenges is choosing a hardware platform that accommodates the tools we require and meets our client's architecture requirements. This requires familiarity, if not relatively advanced knowledge, in several domains to adequately implement our design.

Our project requires the interoperability of the following tools, components and subsystems:

- How a computer's memory hierarchy works
    - The design requires that we understand a modern computer's memory hierarchy to generate as much traffic to the next level of storage as possible
- Mechanics of multi-level caching in an ARM processor
    - Our team must gather detailed information for the cache structure of our chosen architecture
    - The research we do must allow us to maximize the number of cache misses
- Understanding of our platform's bus and I/O layout
    - A common source of interference in multicore systems comes from contention on shared communications pathways between cores
    - Our design must be aware of these pathways and exploit them to uncover interference channels
- Knowledge of multicore processor systems
    - Our design must be able to demonstrate that an application running on one core can interfere with the proper execution of another application on a different core
    - This requires a deep understanding of how cores that make up a multicore system arbitrate resource sharing
- Software development knowledge
    - Our final deliverable is a test suite that has a UNIX-like command line interface and various supporting Python libraries. This requires an adequate knowledge of software development tools and practices (e.g., version management, software libraries, etc.)
    - Addition knowledge is required for the writing of POSIX (Portable Operating System Interface) compliant software at the OS level to be distributed to a wider audience
- FPGA (Field Programmable Gate Array) Development Boards
    - Interfacing with embedded systems
- SBCs (Single Board Computers)
    - Requires knowledge of how to use resource-constrained computing environments
    - Evaluate the costs and benefits of several different boards
    - Requires the ability to efficiently read data sheets to find relevant information
- How to configure and use a type-1 hypervisor
    - Using the hypervisor correctly depends on the team's understanding of isolating computer system resources
    - Requires knowledge regarding the virtualization of computer hardware and subsystems
    - A type one hypervisor runs directly on computer hardware, so configuration knowledge is necessary
- Profiling and analyzing application performance
    - Our team must know how to leverage and use existing performance tools
    - Our design also necessitates that we know how to install these tools on our platform

In addition to the preceding technical aspects, our design is also constrained by our client's ISA (Instruction Set Architecture) requirements. The project description stipulates that the final version of our design must run on an ARM multicore processor. Ideally, our client would like us to choose a multicore processor that has around six cores.

Due to its scope this project, once implemented, will exceed current industry solutions in that it provides an open-source alternative to existing Worst-Case Execution Timing software. The team must synthesize multiple domains of computer engineering, computer science and engineering economics to deliver the final form of our client's request.

4.2. Design Exploration

4.2.1. Design Decisions

Per our client Boeing's guidance, we have been able to condense our project into 3 broad categories: Hardware Platform, Resource Contention Channels and Hypervisor. These three categories are elaborated upon below, allowing us to describe the decision-making process in detail, providing insight into our project's engineering plan.

4.2.1.1. Hardware Platform

The first major decision that was necessary for our team was the choice of a hardware platform for our framework to run on. Three stipulations were given to the team by Boeing. These stated that our hardware should utilize an SoC (system-on-chip) that contains at least two processor cores, that those cores implemented the ARMv8-A instruction set, as well as that the hardware should be in the format of a single-board computer (SBC). From there, the team identified several other constraints to guide our decision, including hardware availability, age, compatibility and feature set. To ensure that we could begin work on the project in an expedient manner, the platform we selected needed to be in stock with an estimated shipping time of no more than one to two weeks.

Hardware age was another critical consideration that the team had to make, as the ARMv8-A instruction set was publicly released in 2011. As many improvements have been made to SoCs utilizing ARMv8-A technology since 2011, it was critical that the team found a platform released within the last 5-6 years, ensuring that our testing was relevant by considering hardware of a more modern design. The feature set of our hardware was also something that had to be accounted for. This includes things such as peripheral connectivity (USB, Ethernet, PCIe (Peripheral Component Interconnect Express)) and memory technology/capacity. These features would be critical for allowing us the most flexibility in exploiting resource contention channels as described in the following section.

4.2.1.2. Interference Channels

The primary motivation of our work concerns developing a tool set that will help our client in verifying multicore avionics systems for compliance with airworthiness regulations, as outlined in FAA Advisory Circular 20-193. Part of this verification process involves identifying and characterizing performance detriments resulting from simultaneous access of shared device resources. For example, memory, Level 2 processor cache and I/O (input/output) subsystems (via USB Ethernet, PCIe or USB). Using our platform feature set as a guide, as stated in section 4.2.1.1, the team had to make several decisions on which resources would be targeted, as well as how they would be exploited (simulating a potential bad actor in real avionics system) to show we are achieving a true worst-case scenario.

The first area of contention the team identified was the processor cache. The processor cache is responsible for storing data that has been accessed recently or frequently (depending on the data replacement algorithm used by the designer), and considerably boosts system performance by reducing the frequency of main memory accesses (which are much slower to perform than cache accesses). Among some other methods, the team chose an attack vector known as "cache thrashing", which is a programming technique designed to maximize cache misses, meaning that data stored in the cache will frequently have to be ejected, and new data read from main memory. This creates a considerable amount of stress on the cache subsystem and could lead to performance degradation in other programs trying to utilize the cache.

The second area of contention that the team identified was the memory subsystem. Any application running on the target platform will need to utilize memory to store information about the work that it is performing, and as such, is a prime area for resource conflict. While modern systems have sufficient protections in place to prevent programs from using too much memory capacity, far fewer protections are in place to prevent programs from using too much memory bandwidth. Bandwidth refers to the rate that information can flow between two given subsystems on a device. In this case, focus is on the CPU-to-DRAM data path, as the CPU often performs processing tasks on program data that is stored in main memory. If a program utilizes an excessive amount of memory bandwidth, the performance of other applications running on the system could experience unpredictable latency when accessing data. For this reason, it is critical that the effects of memory bandwidth contention are analyzed.

The third area of contention that the team identified was the I/O (input/output) subsystem, which handles access to peripheral devices over a variety of protocols. Given that modern avionics' applications require a vast amount of throughput, it is imperative that our testing suite analyses the operation of I/O when interference arises. Such interference concerns include bandwidth limitations, DMA (Direct Memory Access) stressing, resource contention overlapping, end-to-end latency and delay jitter. If a program overloads the available bandwidth, the performance of other important subsystems may take a hit, leading to eventual failure.

4.2.1.3. Hypervisor

When it comes to the choice for hypervisors, the team had only one choice: Xen. Our reason for using Xen is primarily informed by two main reasons: Xen is the only major Hypervisor to support the ARM Architecture, and Boeing specified that our team use Xen in our design. While alternatives like KVM (Kernel-based Virtual Machine) exist, Xen is better suited for use in the avionics industry. Within the Xen hypervisor, the team has elected to use an Ubuntu ramdisk and minimal root filesystem to avoid introducing additional overhead from an operating system that could skew test results. This will allow us to thoroughly test the worst-case execution time while running the interference generators.

4.2.2. Ideation

Per Boeing's request, we are using an ARM SoC and Xen Hypervisor. Boeing's suggestion stems from this platform's use in safety-critical avionics systems. This enables our design to closely mirror what is used in industry. There are many different options when it comes to ARM SoCs, so our team had to select one that meets the requirements of Xen and has multiple cores.

4.2.2.1. Raspberry Pi

The Raspberry Pi is a very popular ARM Multicore SBC with lots of community support. This led our team to consider this board first. We quickly were able to find others who had gotten Xen running on Raspberry Pi's, but the documents and code repositories were a few years old. Our team had a Raspberry Pi already, so we decided to start with this board. Early in development we found that the bootloader used by the Raspberry Pi board is a very proprietary and closed source. This was a major roadblock for our system as it prevented us from getting Xen working. This led us to move away from the Raspberry Pi and pursue other options.

4.2.2.2. RockPro64

The RockPro64 is a popular alternative to the Raspberry Pi and is made by a group known as Pine64. This board meets the ARM requirements of our design and is readily available. In searching for Xen documentation, we found a few references to Xen support in documentation and others getting Xen working on the platform. This platform meets the ARM SBC requirements of our project and has easily accessible and open-source documentation so that we can thoroughly test the platform. This board showed promise for running Xen, but ultimately had issues with the provided tools to get Xen running.

4.2.2.3. Avnet ZUB-1CG

The Avnet ZUB-1CG is an affordable Xilinx Ultrascale+ MPSoC based development board. Our team started to look at this board as a hardware platform as Xilinx is a major contributor to the Xen Hypervisor project and therefore many of their devices support Xen. Our team found lots of documentation about Xen on Ultrascale+ MPSoCs, including how to build Xen on PetaLinux, Xilinx's embedded Linux platform, and how to configure Xen DomU's. While working on getting PetaLinux built with Xen for this board, our team discovered that the ZUB-1CG is not an officially supported version of the Ultrascale+ MPSoC for Xen. This led us to move away from this specific version of the Ultrascale+ MPSoC platform for something with a bit more support and power.

4.2.2.4. Hikey 960

When looking for ARM SBCs that supported Xen, our team found the Hikey 960 which Xen had listed as a supported platform. As we started looking into this platform further it appeared that it met our requirements. The issue was that this board was very old and no longer being produced so the team was unable to source one. This caused the team to quickly move away from this platform.

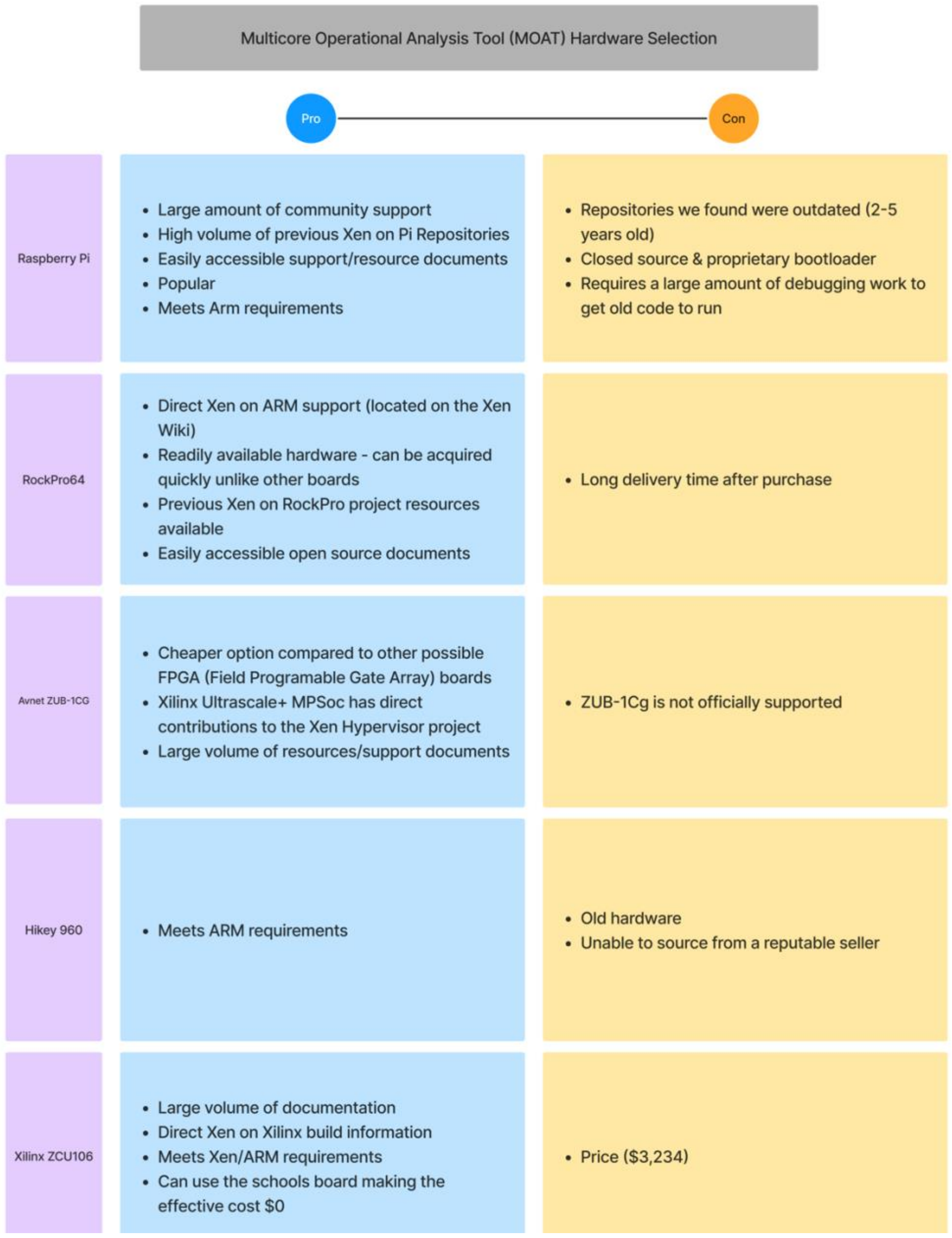
4.2.2.5. Xilinx ZCU106

The ZCU106 Evaluation Kit is a development board based around the XCZU7EV Ultrascale+ MPSoC. This board, as with the Avnet ZUB-1CG, has lots of documentation from Xilinx about how to build Xen for the Ultrascale+ MPSoC platform. This board is also referenced in the PetaLinux build software that supports Xen. After researching and working with the boards listed above, and due to the existence of the PetaLinux framework and extensive availability of documentation, we landed on this platform as our final candidate.

4.2.3. Decision-Making and Trade-Off

When determining the hardware platform our team was going to use, we looked across the internet on sites such as the Xen wiki to consolidate a list of possible solutions. This netted us the pros & cons list included below in Figure 7. To help gauge what our client desired, we presented our findings to Boeing during our weekly team-client meetings. Over the course of the weeks following, along with testing and hardware bring up, we were able to check boards off the list until we ended with our current board of choice, the ZCU106. As most of the boards on the list complete our task, there are small differences such as the delivery time, document availability and most importantly, the repo access that guided our final decision.

As noted below, when working with the Pi, we quickly found that the bootloader was closed source and complex to work with. Although the large amount of community support is a good component when considering the longevity of a product, the amount of work required early on deterred us from this option. After moving away from the Pi, the team discussed with ETG the possibility of purchasing a board from a 3rd party non-name brand store. Due to policies in place, the team was forced to scrap this board due to limitations in availability from reputable locations. This led the team to proceed with purchasing the RockPro64. Although not our first choice, tests and client guidance directed us in this direction. After putting in a considerable amount of time with this board, we came very close to getting a functional installation of Xen. However, around this time we also began researching and working with the Xilinx ZCU106. The availability of documentation and compatible build tools led us to shift our focus solely to working with this board.



Multicore Operational Analysis Tool (MOAT) Hardware Selection

	Pro	Con
Raspberry Pi	• Large amount of community support • High volume of previous Xen on Pi Repositories • Easily accessible support/resource documents • Popular • Meets Arm requirements	• Repositories we found were outdated (2-5 years old) • Closed source & proprietary bootloader • Requires a large amount of debugging work to get old code to run
RockPro64	• Direct Xen on ARM support (located on the Xen Wiki) • Readily available hardware - can be acquired quickly unlike other boards • Previous Xen on RockPro project resources available • Easily accessible open source documents	• Long delivery time after purchase
Avnet ZUB-1CG	• Cheaper option compared to other possible FPGA (Field Programable Gate Array) boards • Xilinx Ultrascale+ MPSoc has direct contributions to the Xen Hypervisor project • Large volume of resources/support documents	• ZUB-1Cg is not officially supported
Hikey 960	• Meets ARM requirements	• Old hardware • Unable to source from a reputable seller
Xilinx ZCU106	• Large volume of documentation • Direct Xen on Xilinx build information • Meets Xen/ARM requirements • Can use the schools board making the effective cost $0	• Price ($3,234)

Figure 7: Decision matrix for hardware selection.

4.3. Final Design

4.3.1. Overview

The system design utilizes a Xilinx ZCU106 FPGA Development Board. A critical part of our design is a software component known as a hypervisor, which allows for fine-grained control over the hardware resources that applications use on the system. Coupled with a base test, and programs designed to stress (in other words, use computing resources) the system in very specific ways, this fine-grained control is essential in ensuring precision in our comparative performance measurements. The purpose of taking comparative performance measurements is to show that a given type of stress cannot reduce the system's performance beyond a certain point. Sections 4.3.2 and 4.3.3 and the figures contained therein describe the components and how they operate in a more technical capacity.

4.3.2. Detailed Design and Visual(s)

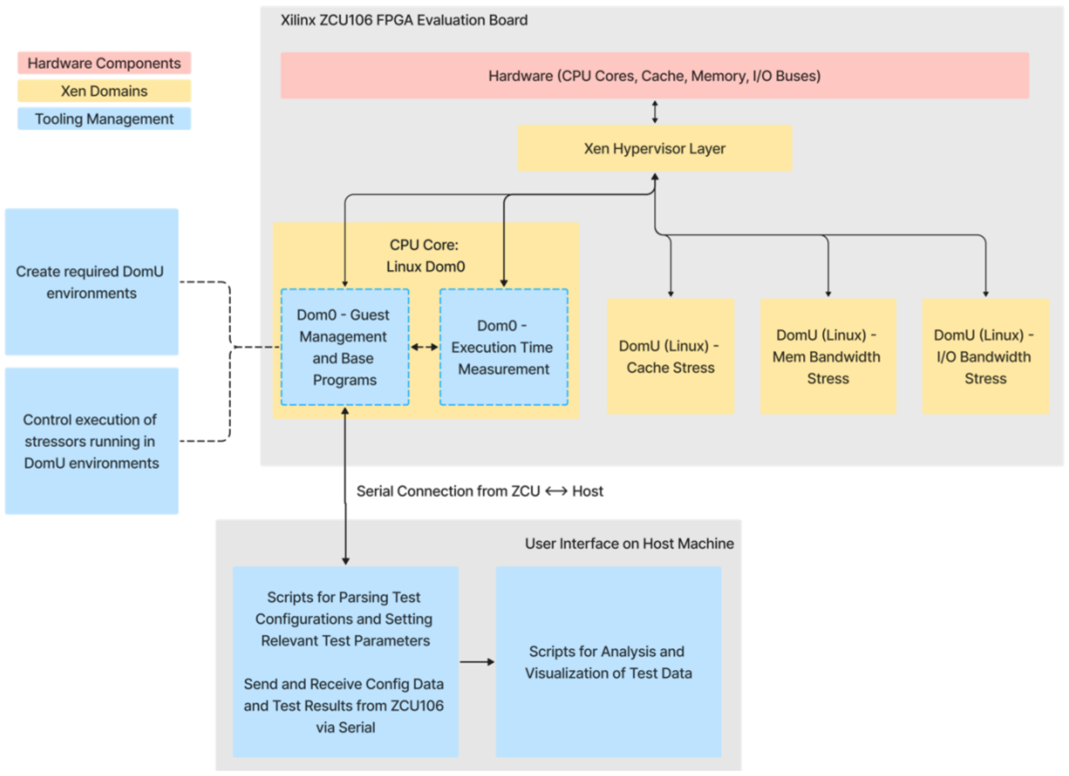

Hardware Components
Xen Domains
Tooling Management
Xilinx ZCU106 FPGA Evaluation Board
Hardware (CPU Cores, Cache, Memory, I/O Buses)
Xen Hypervisor Layer
CPU Core: Linux Dom0
Dom0 - Guest Management and Base Programs
Dom0 - Execution Time Measurement
DomU (Linux) - Cache Stress
DomU (Linux) - Mem Bandwidth Stress
DomU (Linux) - I/O Bandwidth Stress
Create required DomU environments
Control execution of stressors running in DomU environments
Serial Connection from ZCU ⟷ Host
User Interface on Host Machine
Scripts for Parsing Test Configurations and Setting Relevant Test Parameters
Send and Receive Config Data and Test Results from ZCU106 via Serial
Scripts for Analysis and Visualization of Test Data

Figure 8: Block diagram of the system, including hardware and software components and how they are connected.

Figure 8 presents the schematic of our system. It consists of several components, including the hardware contained in the SoC, the Linux kernel which hosts various kernel modules, Xen and its associated domains and the various actions that will be performed by the domains running under Xen.

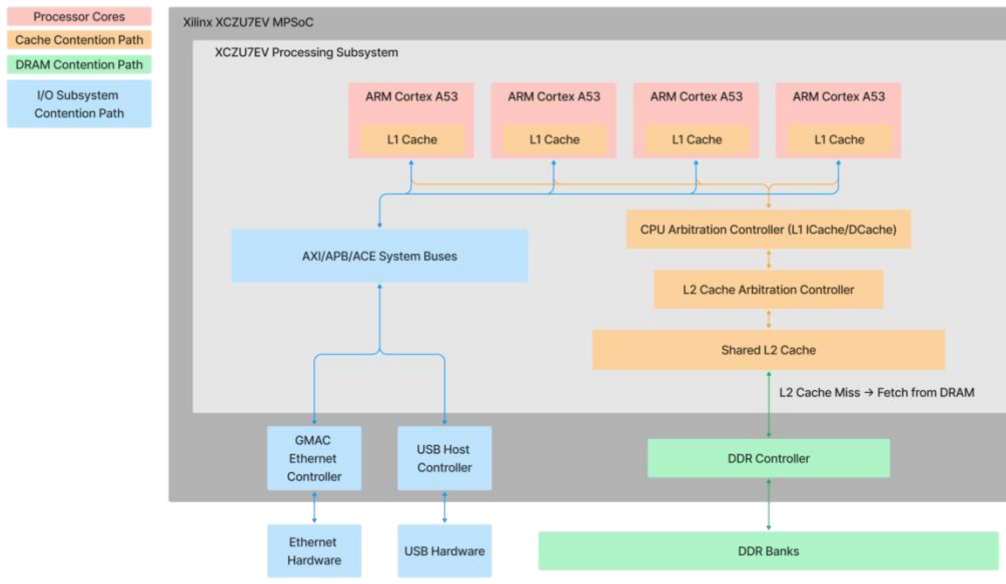


Figure 9: Components and interference channels on XCZU7EV MPSoC and associated hardware

Figure 9 presents the potential interference channels that exist on the processing subsystem of our hardware. The process of stressing the cache occurs through the CPU to cache data path, which can be seen in orange above. Each processor core incorporates its own level 1 cache for instructions and data, however, all four cores share a cache arbitration controller and a single level 2 cache. Since caching data significantly increases application performance, the DomU environment running this stressor will be executing a program that is very cache-unoptimized, meaning that L2 cache misses are frequently occurring. This would replace the data that is present in cache for other programs running on other cores, and in theory, this should affect their performance as the data they had cached is no longer available.

The memory stress environment will follow a similar approach to cache stress. Since cache misses necessitate access to main memory, abusing this property can generate a very large amount of traffic to main memory banks, thereby stressing the available memory bandwidth for all programs on the system. Lastly, the I/O subsystem stress occurs via a system bus that is present on the CPU subsystem and facilitates communication with other controller modules present on the SoC. Multiple programs may be communicating information over a local or wide area network or via other USB peripherals, so generating a large amount of I/O traffic from a DomU on one core will stress the Quality-of-Service behavior (the manner in which I/O traffic is prioritized for transmission) of the shared Ethernet and USB controllers that manage accesses from all components on the SoC.

4.3.3. Functionality

We will begin with Xen - Xen is an open-source hypervisor that creates and manages VMs that run a variety of guest OSs. Xen guests may also be of the bare-metal variety, in which a program is compiled into assembly language and run directly on hardware without an operating system. As defined in our list of terms, Xen runs directly on hardware, making it a type-1 hypervisor. This is beneficial to us because it not only reduces resource overhead, but also affords us increased granularity when it comes to allocating resources to the various domains we will create and manage under it. Xen has two types of domains (virtual machines) that the user can create: DomO and DomU. DomO is the manager domain in which configuration of the main hardware and various DomU environments (guest domains/virtual machines) takes place, and this configuration is performed via hypercalls, which go up through the kernel to the hypervisor running on hardware. Once a guest VM (DomU) is configured, it can be set up to run another operating system (such as Linux), or to run a bare-metal program. A bare-metal program is a program that has been compiled down from high-level source code (like C) to the assembly code corresponding to the platform it will be running on. The advantage of this method is that no resource overhead will be consumed by an operating system running on that guest domain, as the program is being executed directly on hardware. However, it is significantly more complex to implement, so in order to expedite product bringup, our team has elected to utilize a lightweight Linux installation to host the contention generators in DomUs.

The user-facing portion of the tool set is responsible for parsing test configurations and setting relevant parameters. It communicates with the development board, and by extension, DomO, over a serial connection, where it manages the complete test flow. It instructs DomO to boot certain DomUs and start a given type of resource contention generator on it, after which the victim program is started in DomO. The configuration file also specifies the number of times the victim program should execute, which allows us to observe run-to-run inconsistencies and any other statistical trends that may arise. After the test loops have completed, the DomUs are shut down, and the next test in the list is prepared. After all tests have completed, the front end analyzes the collected information to extract key metrics, including WCET, average execution time, and standard deviation, and also plots the run times of each test loop.

Xen DomO is responsible for several essential tasks. The first is managing guest domains that host the resource contention generators. The guest domains are not statically assigned any information other than the amount of DRAM and CPU core(s) that is/are reserved for it. As mentioned above, the front-end interface communicates with DomO to issue commands telling it to boot DomUs assigned to the cores that have been selected for a given test, which happens through the Xen Management Toolchain (“xl”). DomO must also store test results in a consistent format that can easily be parsed for execution time data that will be analyzed later. The ability to analyze this information for our base program is essential to determine that one, our interference generators are generating resource interference, and two, how much of a detriment to performance is observed to the base test when resource contention is enabled.

Xen DomU environments are less complex than DomO since they are purely responsible for executing a program (or programs) designed to stress a certain part of the system. In our case, we are interested in targeting the shared (L2) CPU cache subsystem, the data path between the CPU cores and main memory, and the data path from the CPU cores to the I/O subsystem on the Xilinx XCZU7EV MPSoC. These subsystems were chosen because they are shared between processor cores, meaning that programs on independent cores could still affect each other by misusing (stressing) the various subsystems (refer to Figure 9 for visualization).

4.3.4. Areas of Challenge

Our client’s requirements led us to a narrow range of choices for hardware, but despite this one of the greatest challenges was getting every tool we needed to implement our design to work on our hardware platform. Getting the Xen hypervisor installed on one of the platforms we identified took most of the first semester and was a major impediment to both our team's ability to satisfy requirements and make technical progress.

We overcame this challenge by dividing the team's efforts between two hardware platforms in the early stages of our project. We researched and implemented the initial stages of our design concurrently on two different platforms: the Xilinx ZCU106 and the Pine RockPro64. Our client was also able to provide guidance in configuring the software and hardware components we needed for our design. Eventually, it became evident that the ZCU106 was much more conducive to our design and at our client's request we focused our efforts on this platform.

With the groundwork established for our design, our team started to encounter obstacles related to our user's needs. For example, one of the early iterations of our design induced stressors on our platform and collected relevant timing information. This prototype required the user to manually type in the commands to produce the desired output. During a presentation of this prototype to our client, one of our users pointed out a strong use case for running stressors automatically via a configuration file. User feedback required our team to react quickly to user needs as they came up. We overcame this issue by taking a modular design approach for the front-end that had the flexibility to accommodate users' desires on the fly.

4.4. Technology Considerations

The following section details the technologies that our design uses. Each technology is concerned from the standpoint of its strengths and weaknesses, as well as viable alternatives when applicable.

4.4.1. Hardware Platform

We initially chose the RockPro64, which is a Single Board Computer (SBC) that closely aligns with what our client requested in terms of a hardware platform. It uses the ARM instruction set and has a multi-core heterogeneous processor. This architecture allows our team to partition the system resources such that a victim application running on one core can have its performance degraded by a stressor running on another core. While we were able to find abundant hardware documentation for the platform online between the official manufacturer and an online wiki, implementing our design on this platform proved challenging. The existing documentation provided us with a detailed description of the boards microarchitecture but finding information related to installing the Xen hypervisor on this platform was scarce to non-existent.

Relative to other SBCs that we considered, like the Raspberry Pi 4 or HiKey 960, the RockPro64 was the most well-documented which motivated our decision to pursue it initially. One aspect we considered in our initial analysis was that Xen may be easier or better suited to running on an FPGA like the Xilinx ZCU board, which we ultimately found to be true. We speculate that many embedded applications that use Xen run on an FPGA board rather than a commodity SBC like the RockPro.

4.4.2. Xen Hypervisor

Xen is an open-source type I hypervisor with abundant documentation. Additionally, our client has several engineers with experience using Xen for different applications. This allows us to create virtual machines on our hardware platform to properly isolate the system's hardware resources to properly characterize interference channels.

The challenge of using this technology lies in its learning curve. Xen is widely used in industry, but it is commonly running on hardware different than what our team is using. This has meant that we have spent more time than we had originally expected troubleshooting Xen issues when it was originally meant to simply be a supporting technology to our overall design. The alternative to using Xen is that our project can in some capacity be emulated in a Linux environment.

4.4.3. Stress-NG

Stress-NG is a software library used to stress the various subsystems of a computer. The primary advantage that this technology lends to our project is that it implements many stress base cases. This is useful for our project in that it allows us to uncover previously unidentified interference channels on our hardware platform, which is the fundamental goal of our design. Stress-NG is ultimately an accelerator for our work; it allows us to get to the most important parts of our design sooner. Without it, we would need to write custom stressors for each base case.

Stress-NG's weaknesses lie in that it is a general library intended for use in various systems. If there is a specific aspect of our platform we wish to exploit, we may need to write custom software to accomplish the behavior we want. This can prove to be a time-consuming process given how closely our software is integrated with the hardware. There are not many alternatives to Stress-NG, so writing our own stressors in C and assembly are likely not a viable option. This could be an area that is open for future work, given an individual that would like to contribute to the open-source software.

5. Testing

Our testing philosophy is centered around isolating systems and performing unit tests to ensure that they fit into our larger design. Due to the multiple layers of hardware and software to successfully implement our project, most of our tests are focused on ensuring that the system has the same functionality after a unit is integrated as it did before. This allows us to build on a solid base.

Naturally, most of our testing is regression testing. The scripts we have stored in our GitLab repository allow us to iterate quickly on our design and recover to a working state if a regression test fails. This aids not only the reproducibility of our work but also its dissemination to a wider audience.

The primary challenge of testing our project also stems from the scope of testing that must be performed. Our design requires us to perform testing at the hardware, software, and user levels. To test the project at all these levels, the team tests each platform together whenever possible. This allows for streamlined integration testing and making sure that changes

5.1. Unit Testing

To test the platform, the team utilized real world hardware to verify that the system functioned. The team was able to test each software component added in PetaLinux after the build was completed on a ZCU106. Each unit of the interface software was tested continuously by performing basic tests on a development machine, as well as testing the connection with the actual hardware. Much of the team's testing was performed on the whole system in order to avoid integration bugs.

5.2. Interface Testing

When testing the control interface, the team first tested locally on computers to make sure that we had basic functionality and were able to do things such as read and write over serial ports. As the interface became more complex, the team used the ZCU106 to make sure that all the interface connections were working. This allowed for fast revisions to be made to the interface to maintain and improve functionality.

5.3. Integration, System, and Regression Testing

The team focused on continuously integrating the components of the project as they were being developed. This allowed for the whole system to be tested after each change to any of the components. The whole system was tested on a consistent basis in order to collect data for analysis which allowed the team to find and correct bugs quickly and efficiently. Another benefit of this testing strategy was it allowed the team to make sure new features didn't affect old functionality. Using the system to collect the required data required all the old features to be operational, which shows that the system did not regress.

5.4. Acceptance and User Testing

Acceptance testing was handled by presenting the program's output at different stages to our client for their approval. The team was able to add and test features between these meetings, then present the new outputs and data to the client for direct approval. This made sure that the team's project met the requirements and performed the way that the client was expecting.

5.5. Results

The results of the team's testing were positive. The system generates interference, and the software can collect data and categorize the interference. The testing strategies that were used resulted in quick and efficient identification of issues which allowed the team to address them rapidly. The constant communication with the client resulted in a focus on what was important during testing and led to a better product with useful features.

6. Implementation

Our team's implementation largely matched our final design in terms of core functionality, with a few changes to certain areas due to time constraints. We will break the implementation process down into several sections, including hardware and hypervisor bring up, resource contention generator setup, and front-end interface development.

6.1. Hardware and Hypervisor Bringup

Our team utilized the PetaLinux framework, a Linux distribution that includes build tools and options specific to Xilinx ZCU development boards, to get Xen running on our hardware. While most of the configuration was left as default, there were a few important things that had to be modified to accommodate the inclusion of Xen, including several package groups and hardware device tree entries. We also needed to include Stress-NG in our PetaLinux build, since we had decided to use some of its functionality to serve as our victim program on which we would take execution time measurements. Since PetaLinux is essentially an interface for a Xilinx-customized version of Yocto, a popular tool for building embedded Linux images, adding a custom BitBake layer that built and packaged Stress-NG was straightforward. After allowing the complete PetaLinux image to build, the team had to configure some ancillary files for the embedded bootloader that inform the hardware where various system images are located during bootup. This was achieved by using a tool from the Xen Project called ImageBuilder which generated a bootloader configuration file based on the parameters and image sizes of our files that were built with PetaLinux. The end result was a disk image that could be flashed onto an SD card and booted on our target hardware. The team thoroughly documented this process in the event that changes to the PetaLinux image were necessary, which turned out to be a very helpful decision.

6.2. Resource Contention Generator Setup

The team ideated through several approaches to generating interference on our system. Eventually, we landed on using Stress-NG due to its wide variety of stress test options. Because of this, the team also pivoted away from the idea of running interference generators as bare-metal applications and on to running a lightweight Ubuntu environment in the DomUs. To run guest OSes in the DomUs, a decompressed Linux kernel image, a ramdisk (used to provide limited drivers during system boot), and a root filesystem (for permanent data storage) are required. To achieve this transition, the team downloaded an ARM-compatible version of Ubuntu and extracted the kernel image and ramdisk from it. We then downloaded an Ubuntu Base root filesystem, an extremely lightweight disk image intended for virtualization uses. We then used a tool called *virt-customize,* which allowed us to install packages (such as Stress-NG) on to the root filesystem image without having the guest system booted. This was essential because the guest systems do not contain a package manager and do not have a network connection, which would make it impossible to install packages with the system up. In our final design, there exists three separate DomU environments, one for each of the remaining CPU cores outside of DomO.

6.3. Front-End Interface Implementation

The final component of the design involved creating a method of interfacing the user with the hardware that we had brought up, as well as analyzing the test data that we collected. To achieve this, the team elected to utilize the on-board serial connection of the ZCU106. Data is transmitted over the serial connection via a custom Python framework that presents the user with several options. These options include:

- Terminal mode (allows the user to directly issue commands to the board, useful for troubleshooting)
- Batch test from configuration file
- Single test from command-line input

Terminal mode is straightforward and allows direct issuance of any command that the user wishes to execute in the DomO environment. This mode proved to be useful when debugging initial versions of the design. Batch testing mode allows the user to specify multiple sets of test parameters, such as interference types and associated cores to run them on, victim program details, and test name, in a YAML file. This YAML file is parsed by the interface, and the tests are then run on the board with no user interaction required. This feature was very useful when collecting test results, as it allowed us to have tests running in the background while working on other aspects of the system. The user may also choose to run a single test, manually specifying interference types, cores, and the victim program in the command line. The inclusion of these options makes the frontend a flexible and valuable component of our overall system design.

The final component of the frontend consists of results parsing and analysis. As the team elected to use Stress-NG in DomO as our victim program, we were able to leverage its support for outputting results in a YAML format, which allowed us to use Python to easily parse and analyze the large amount of data generated by various test runs.

6.4. Design Analysis

The implementation of the team's design works well. The system has features to make automated testing much easier, as well as providing a prebuilt platform for the hardware. This is demonstrated by the interference results that have been collected over the course of the project, as well as the client's approval of the system's results. Overall, the system's design is efficient and functional while still being feature rich.

Something that needs to be improved is the process for parsing the collected data. In the current form, this requires the user to shut down the board, pull the logged files off of the memory card manually, and then run a separate parsing program on the results. Ultimately this should be integrated as a function of the main control program. One of the reasons this limitation exists is due to the serial connection between the host and target, and the software libraries used to establish that connection.

Additionally, the team did not have sufficient time to implement resource contention mitigations, as prior milestones in our design process consumed more time than initially anticipated. This area could be a great focus for future project efforts.

7. Professional Responsibility

7.1. Areas of Responsibility

Area of Responsibility	IEEE Code of Ethics	NSPE Code of Ethics
Work Competence	Ensure that the team has the ability and knowledge to complete every project completely and safely.	Engineers shall undertake assignments only when qualified.
Financial Responsibility	Provide clients and users with accurate and realistic estimates for the costs associated with the projects.	Act as good agents on a client's behalf.
Communication Honesty	Provide honesty in all aspects of feedback, criticism, and design abilities.	Avoid deceptive acts & make statements in a truthful manner.
Health, Safety, Well-Being	The health, safety, and well-being of the users and public is the number one priority.	Hold paramount the safety, health, and welfare of the public.
Property Ownership	Treat others and their property fairly and with respect.	Act as good agents on a client's behalf.
Sustainability	Strive to implement positive sustainability practices to better the lives of users and surrounding communities.	Adhere to the principles of sustainable & development in order to protect the environment for future generations.
Social Responsibility	Act as responsible members of society and strive to better one's surroundings.	Honorably conduct themselves as to enhance the reputation of the profession.

Figure 10: IEEE and NSPE Code of Ethics Aeras of Responsibility

Comparing IEEE's code of ethics to NSPE (National Society of Professional Engineers) we can see a lot of the same values; prioritize safety and health while avoiding un-honorable acts, etc. At face value IEEE can be described as a worldwide association of electronic and electrical engineers, which differs from NSPE which encompasses primarily U.S. based engineers.

7.2. Project Specific Professional Responsibility Areas

- Work Competence – This Area of Responsibility is extremely important for our project as we are dealing with safety-critical systems. If our system performs incorrect analysis on multicore embedded systems, it could lead to failures in airplanes mid-flight and potential loss of life and property. – ***Performance (High)***
- Financial Responsibility – Our project has a low cost as it is primarily software based and is portable across the ARM architecture. – ***Performance (Low)***
- Communication Honesty – This area applies to our project as we are completing work for our client, Boeing. It is very important that we keep communication open and keep our client informed about our progress, any challenges encountered, and any issues with our

work. In a broader context, the results of our work could inform the safety decisions across the avionics industry. This means that our team has a strong professional and ethical responsibility to communicate clearly, unambiguously, and with absolute transparency. – ***Performance (High)***

- Health, Safety, Well-Being – The project does not directly affect people or have many health risks. As stated earlier, however, if our project is not built well, it could lead to failing avionics mid-flight. – ***Performance (N/A)***
- Property Ownership – Our project is Open Source and will ultimately be available for anyone to use. This leads to Property Ownership not being something that affects our project. – ***Performance (N/A)***
- Sustainability – This Area of Responsibility is also a low priority for our project as it has very little direct environmental impact. Our software also runs on hardware that consumes very little power. – ***Performance (N/A)***
- Social Responsibility – The Social Responsibility of our project is to improve the efficiency of Avionics systems while maintaining the highest levels of safety. – ***Performance (Medium)***

7.3. Most Applicable Professional Responsibility Area

For our Multicore Analysis Tooling project, Work Competence is the most important Area of Responsibility. If the team does not complete the project competently, there could be serious consequences, such as airplane crashes due to the failure of avionics systems. For this reason, the team has spent lots of time researching methods to validate WCET and use industry contacts to verify the correctness of our system.

8. Conclusions

8.1. Summary of Progress

Our team produced a test framework that characterizes the multicore interference of test programs in the presence of competing processes. We translated our client's hardware platform requirements into a hardware / software system that allows a user to test a base program against multiple sources of interference. Our team also produced software on top of this system that characterizes multicore interference in terms of its worst-case execution time.

In addition to satisfying the functional requirements of our client, we were also able to accommodate our users' needs by making our design configurable. Specifically, our design allows a user to run specific tests in addition to automated scripts. Our team also documented each phase of our design process, aiding both the extensibility and reproducibility of our design.

8.2. Value Provided

The system developed by the team provides a convenient and useful way to categorize multicore interference on safety critical systems. This system meets the user needs for functionality and creates an easy test harness for them to use during designs of other systems. This system successfully shows the worst-case interference of shared resources. This design can not only be applicable to the aerospace industry, but also any other safety critical systems that utilize multicore systems. The flexibility of the design could allow for it to be used in many industries, and on many products. The team has seen evidence of the product's value in several conversations with our client, in which we discussed ways that our product can be integrated with their design flow, and design considerations that the team should take as we wrap up implementation.

8.3. Next Steps

As our work on this project is ending, we have taken the necessary steps to ensure that our project can be handed off successfully. In our case this means getting permission from both Iowa State University and our client Boeing to open-source our work. Handing our work off to the open-source community allows our project to continue to grow and evolve beyond our input.

Specifically, we would like to see our design offer a more comprehensive suite of interference channels. This could include combinations of existing interference channels or aspects of our platform that we did not consider. Another useful aspect for future development would be the implementation of interference mitigations, which we were unable to experiment with due to time constraints, as we previously described.

Our design could also continue to develop in terms of its accessibility and ease of use. Currently, users interact with the design strictly via a command line interface. Future development could add a simple GUI to add a visual method of interacting with the tool. Lastly, the tool could support automation by running preconfigured stress tests or accepting scripts that a user has written themselves.

9. References

[1] S. H. VanderLeest and C. Evripidou, “An Approach to Verification of Interference Concerns for Multicore Systems (CAST-32A),” *SAE International Journal of Advances and Current Practices in Mobility*, vol. 2, no. 3, pp. 1174–1181, Mar. 2020, doi: https://doi.org/10.4271/2020-01-0016.

[2] S. H. VanderLeest and D. Matthews, “Incremental Assurance of Multicore Integrated Modular Avionics (IMA),” in *IEEE Xplore*, ieeexplore.ieee.org: IEEE Xplore, Nov. 2021. Accessed: Jan. 16, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/9594404

[3] S. H. VanderLeest and S. R. Thompson, “Measuring the Impact of Interference Channels on Multicore Avionics,” *arXiv.org*, Jan. 06, 2021. https://arxiv.org/abs/2101.02204 (accessed Apr. 16, 2024).

[4] S. H. VanderLeest, J. Millwood, and C. Guikema, “A Framework for Analyzing Shared Resource Interference in a Multicore System,” in *IEEE Xplore*, IEEE Xplore: IEEE Xplore, Dec. 2018. Accessed: Jan. 16, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/8569651

10. Appendices

Appendix 1 – Operation Manual

This appendix provides instructions on configuring and using the stress generation environment and tool suite. This guide assumes that the user has a functional PetaLinux installation on their test hardware. Instructions for building and installing PetaLinux can be found in Appendix 2.

Configuring DomU Environments

While the user can theoretically use any version of Linux for their DomU environments, we have pre-configured a minimal Ubuntu 22.04 system that the stress generators will run in. These files can be found under the ZCU106/DomU/Ubuntu folder in the code repository. This folder contains several important files:

- Vmlinux-5.15.0-119-generic
  - This is the Linux kernel that the DomU environments will load.
- Ubuntu-base-rootfs-stressng.tar.gz
  - This is the compressed filesystem image. It provides persistent storage and the necessary tools for the DomUs to induce resource contention on the system.
- initrd.img-5.15.0-119-generic
  - This is the ramdisk that the system will utilize on bootup. It provides a minimal set of drivers that the system needs to boot and fully function.
- Ubuntu-base-stressng.cfg
  - This is the DomU configuration file. It specifies the amount of CPU cores and memory the DomU may utilize, and what disk image it should load.

After locating these files, the user will need to move them to an appropriate directory on the SD card containing the PetaLinux installation. The framework expects these files to be located at [**root partition**]/**ubuntu** on the SD card. When the SD card is mounted to a device that is NOT the ZCU board, the The user should then decompress the root filesystem image to that folder using a tool like gzip. Once the image has been untarred, the user will need to create two more copies of the

DomU configuration files and root filesystem images, as one copy is necessary per core that is used to generate interference. It is not necessary to copy the ramdisk and kernel images, as those items are read-only and not specific to a given core. The configuration files should be renamed to core1.cfg, core2.cfg, and core3.cfg, or up to however many cores the user's target system has. The untarred root filesystem images should also be renamed according to this scheme. Each configuration file will also need to have two parameters modified to point to the correct disk image and assigned its corresponding CPU core. This should be done as follows:

- core[x].cfg
    - disk=['/run/media/root-mmcblkop2/ubuntu/core[x].img,raw,xvda[x],rw']
    - cpus=[x]
    - Replace [x] with the number of the core you are configuring

Once this configuration has been completed, the user can move on to setting up the test framework with their desired contention generation types and baseline programs.

Using the Test Framework

The test framework is provided in the form of a Python file. The device that the user is hosting the test framework on should be a separate device from the test board and should be running Linux and a version of Python >= 3.10. The user should first proceed by attaching the serial connection from the test board to their host device. The framework assumes that the serial connection from the host device to the test board is enumerated as ttyUSB0 on the host system. The user should verify this before proceeding and change the device name in the main() function if necessary.

When the user first runs the framework, they are presented with three choices: Terminal mode, MOATerm mode, and batch mode. Each of these options has a specific purpose. Terminal mode serves as a pure serial console and allows the user to directly issue commands and read responses from DomO. This mode is useful when debugging issues on the board. MOATerm mode is the first of two test modes that the framework supports. MOATerm allows the user to manually specify test parameters over one test run only. The user may specify the following test parameters:

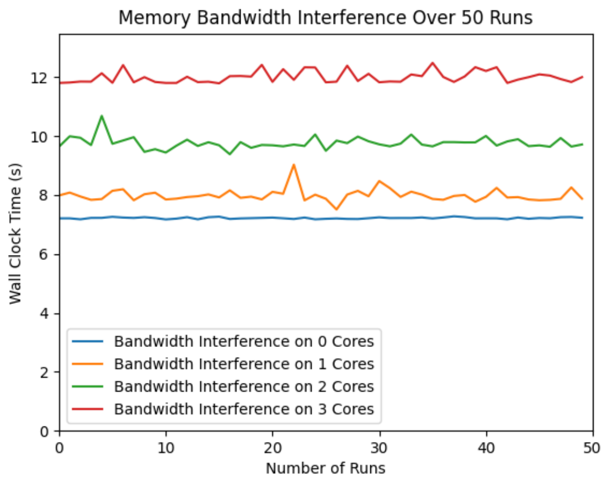
- Friendly Test Name (used for results directory naming)
- Number of CPU cores to run interference generation on
- Type of interference to be run on a given CPU core
- Victim program that will be analyzed for execution time metrics
    - Must be in the form of a valid Stress-NG command
- Number of times to loop the victim program

Once the user has specified those parameters within the terminal window, the tool takes control, booting the appropriate DomUs associated with the cores chosen, starting the interference generation, and running the victim program. Once the victim program has run the specified number of times, the tool returns and the test is complete.

The final test option, batch mode, allows the user to write the above parameters into a YAML file and execute multiple tests sequentially. This mode is useful for running many different combinations of base programs to generate a large number of test results with minimal user interaction. Details on the exact formatting of the YAML file can be found in the code repository.

Analyzing Test Results

Once the user has completed their desired number of test runs using one of the two available modes, it is necessary to perform statistical analysis on the resultant metrics in order to generate some insights. This is achieved with a separate Python file, named yaml_parser.py. In order to use this file, the user needs the matplotlib and yaml Python packages installed on their host system. The user should first remove the SD card from their test board and attach it to their host machine. Test results are stored in the directory [**root partition**]/[**friendly test name**] on the SD card. Given the way the YAML parser works, data analysis can only be done for one group of runs at a time, for instance [base run, 1 core interference, 2 core interference, etc.]. The user should edit the data_dirs array to point to each of the directories that contains results to a corresponding group of runs. Once data_dirs has been updated, the user should run the program, which will aggregate all the data collected from the base program in each of the directories specified, and provide the average, worst-case, and standard deviation of the execution time from the data in each directory. It will also plot the execution times for each run in the dataset for visualization purposes. An example of such functionality is shown below.



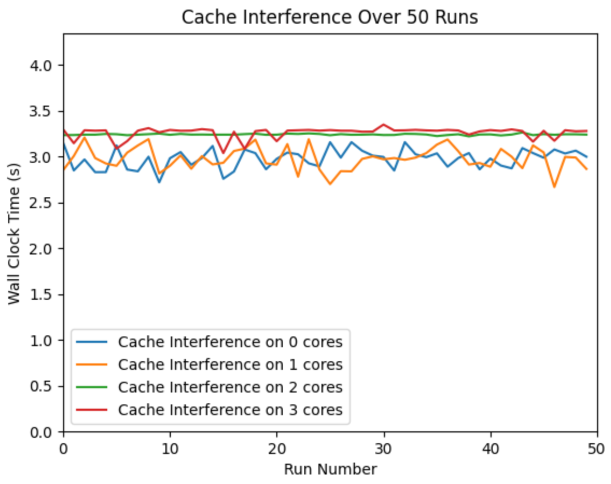


Figure 11 (top): Execution time of a test over 50 runs, exhibiting a large amount of architectural interference.
Figure 12 (bottom): Execution time of a test over 50 runs, demonstrating a small amount of architectural interference.

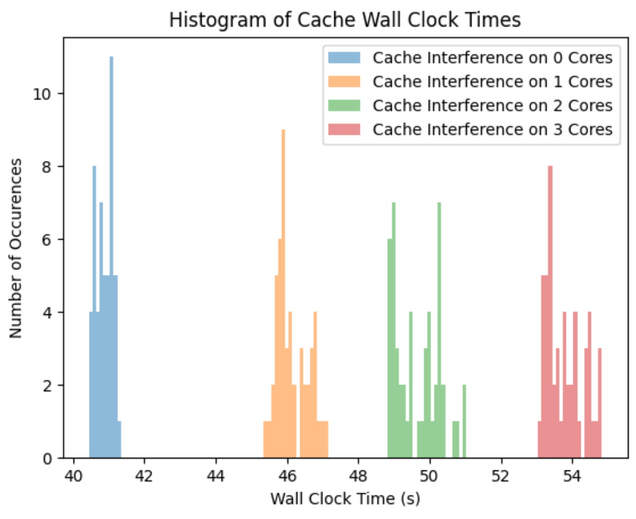


Figure 13: A histogram generated by the results analyzer. It shows the distribution of execution times over a victim program subjected to varying amounts of interference.

Appendix 2 – Building a PetaLinux Image

The code repository linked in appendix 5 contains a pre-built version of PetaLinux that the user may choose to use if it meets their needs. However, if the user wishes to make any additional modifications, such as adding additional packages or configuration options, instructions to do so can be found under the ZCU106 folder of the GitLab repository.

Appendix 3 – Alternative/initial version of design

As previously mentioned in this document, the team initially put a good amount of work into trying to work with the RockPro64 for this project. As such, the team had centered most of the design document from semester 1 around its architecture. The team had also developed several scripts that were intended to automate the build process under the assumption that we were able to get Xen working on that hardware. Late in the semester, unfortunately, the team discovered that the RockPro64 was ultimately unsuitable for our project due presence of alternative hardware with better documentation and due to the challenges, we'd experienced trying to get the hardware up and running. While much of our conceptual research we'd done on ARM-based processing systems still applied to our new hardware, all the work we'd done on documenting and automating the image build process for the RockPro64 was essentially scrapped as it was no longer relevant.

Appendix 4 – Other Considerations

- Building and configuring embedded Linux software is often "like putting Windows on an iPhone" – Steve VanderLeest, our advisor.

Appendix 5 – Code

- The code is open-source on GitHub as per our client's request.

Appendix 6 - Team

Team Members

- Alex Bashara – Embedded and Cache Engineer
- Joseph Dicklin – I/O Engineer
- Hankel Haldin – Platform Bring up Engineer
- Anthony Manschula – Project Coordinator and Memory Engineer

Required Skill Sets for Your Project

This project requires familiarity with computer architecture and an understanding of how code runs at a low level and knowledge of real-time embedded systems.

Skill Sets Covered by the Team

- Alex Bashara – Real Time Embedded Systems
- Joseph Dicklin – Low Level Electrical Systems
- Hankel Haldin – Low Level System Programming
- Anthony Manschula – Computer Architecture Knowledge

Project Management Style Adopted by the Team

The team adopted an agile project management style, focusing on weekly meetings with the team and advisors to checkpoint progress and set goals for the following week. Issues have been tracked in Gitlab with weekly updates during our advisor meetings.

Individual Project Management Roles

- Alex Bashara – Cache Interference Lead
- Joseph Dicklin – I/O Interference Lead
- Hankel Haldin – Hypervisor Lead
- Anthony Manschula – Memory Interference Lead

Team Contract

Team Name: MOAT

Team Members:

1) Alexander Bashara	2) Anthony Manschula
3) Hankel Haldin	4) Joseph Dicklin

Team Procedures

1. ***Day, time, and location (face-to-face or virtual) for regular team meetings:*** Sundays at 12pm in person (Subject to Change).

2. ***Preferred method of communication updates, reminders, issues, and scheduling (e.g., e-mail, phone, app, face-to-face):*** Discord, GitLab Boards (issue tracking, management), Trello

3. ***Decision-making policy (e.g., consensus, majority vote):*** Consensus.

4. ***Procedures for record keeping (i.e., who will keep meeting minutes, how will minutes be shared/archived):*** Will take meeting notes and minutes in the shared OneNote notebook.

Participation Expectations

1. ***Expected individual attendance, punctuality, and participation at all team meetings:*** Attendance to all class sessions, timely attendance to out of class meetings (what the team deems needed per week), participation in Boeing meetings (ideas, questions, etc…), as well as other/misc. requirements discussed within the team.

2. ***Expected level of responsibility for fulfilling team assignments, timelines, and deadlines:*** Be responsible and proactive about meeting deadlines and be sure to communicate in a timely manner if you do not believe you will be able to meet a deadline. Team members should be able to problem solve on their own but not be afraid to ask a question if they cannot find an answer.

3. ***Expected level of communication with other team members:***

   Respond to Discord messages within a day's time.

   Group members should provide notice if they are not able to attend a meeting. If a group member is stuck or otherwise having difficulty completing work, they should inform the group so continued progress can be made on the project.

4. ***Expected level of commitment to team decisions and tasks:*** If a group member takes responsibility for a task, he should be committed to completing it by assigning a deadline to it. If a group member is blocked on the completion of a task, it is that person's responsibility to ask for help.

Leadership

1. ***Leadership roles for each team member (e.g., team organization, client interaction, individual component design, testing, etc.):*** Team organization & client interaction – Anthony, Hardware research – Joe, Embedded engineering and testing lead – Alex, Hypervisor & platform bring up – Hankel

2. ***Strategies for supporting and guiding the work of all team members:*** Be transparent with your skills and deficiencies, allowing your fellow team members to aid in the completion of your individual role.

3. ***Strategies for recognizing the contributions of all team members:*** Talk about what each team member accomplished over the week and what their goals are for the next week.

Collaboration and Inclusion

1. ***Describe the skills, expertise, and unique perspectives each team member brings to the team:*** Each member comes from a similar knowledge base in terms of computer and electrical engineering knowledge. However, each member has a different focus in terms of specialized expertise (digital logic, operating systems, etc.).

2. ***Strategies for encouraging and supporting contributions and ideas from all team members:*** Everyone on the team has a unique skill set and chance to contribute in a meaningful way, so clearly communicate what your strengths are. Help others when you have the chance to and take the opportunity to move the project forward when they are presented.

3. ***Procedures for identifying and resolving collaboration or inclusion issues (e.g., how will a team member inform the team that the team environment is obstructing their opportunity or ability to contribute?):*** Members should be open about their progress and any environment issues in meetings so that they can be addressed quickly.

Goal-Setting, Planning, and Execution

1. ***Team goals for this semester:*** TBD. As stated above, we must first meet with our Boeing contact to get an in-depth overview of the complete project. Once we know that, we will discuss within the team what is reasonably achievable given the semester period.

2. ***Strategies for planning and assigning individual and teamwork:*** Plan the major deliverables after our first meeting with Boeing rep. From there, fill out sub-tasks necessary to achieve these deliverables on time. Tasks will be assigned based on knowledge level and confidence in the subject. At the weekly member meetings on Sunday, tasks will be reviewed in terms of status and time frame, and other tasks will be created or removed as necessary to keep the project organized.

3. ***Strategies for keeping on task:*** Set meaningful and achievable goals each week so that we can continuously make progress on the project.

Consequences for Not Adhering to Team Contract

1. ***How will you handle infractions of any of the obligations of this team contract?*** Have a civil discussion as a team to see how we can fix the issue.

2. ***What will your team do if the infractions continue?*** If the initial intra-team discussions prove unproductive, discuss recommendations with the project advisor.

****

a. I participated in formulating the standards, roles, and procedures as stated in this contract.
b. I understand that I am obligated to abide by these terms and conditions.
c. I understand that if I do not abide by these terms and conditions, I will suffer the consequences as stated in this contract.

1) Joseph Dicklin DATE 1/30/2024

2) Hankel Haldin DATE 1/30/2024

3) Alexander Bashara DATE 1/30/2024

4) Anthony Manschula DATE 1/30/2024