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 as well as 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 hardware platform that our framework would run on. The three stipulations given to the team by Boeing were that our hardware should utilize an SoC (system-on-chip) that contains at least two processor cores, that those cores implement the ARMv8-A instruction set, and 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, 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 and 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 Resource Contention 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 and how they would be exploited (simulating a potential bad actor in a real avionics system) in order 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 datapath, as the CPU often performs processing tasks on program data that is stored in main memory. In the event that 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's imperative that our testing suite analyses the operation of I/O where interference arises. Such interference concerns include bandwidth limitations, DMA (Direct Memory Access) stressing, resource contention overlapping, end-to-end latency, and delay jitter. In the event that 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. This is due to 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. Within the Xen hypervisor, the team has decided to use bare-metal DomU programs to not deal with multiple OSes taking up compute power on the chip. 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. 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 very quickly were able to find others who had gotten Xen running on Raspberry Pis, 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 made by 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. For these reasons, this has been the main platform the team has been developing.

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.

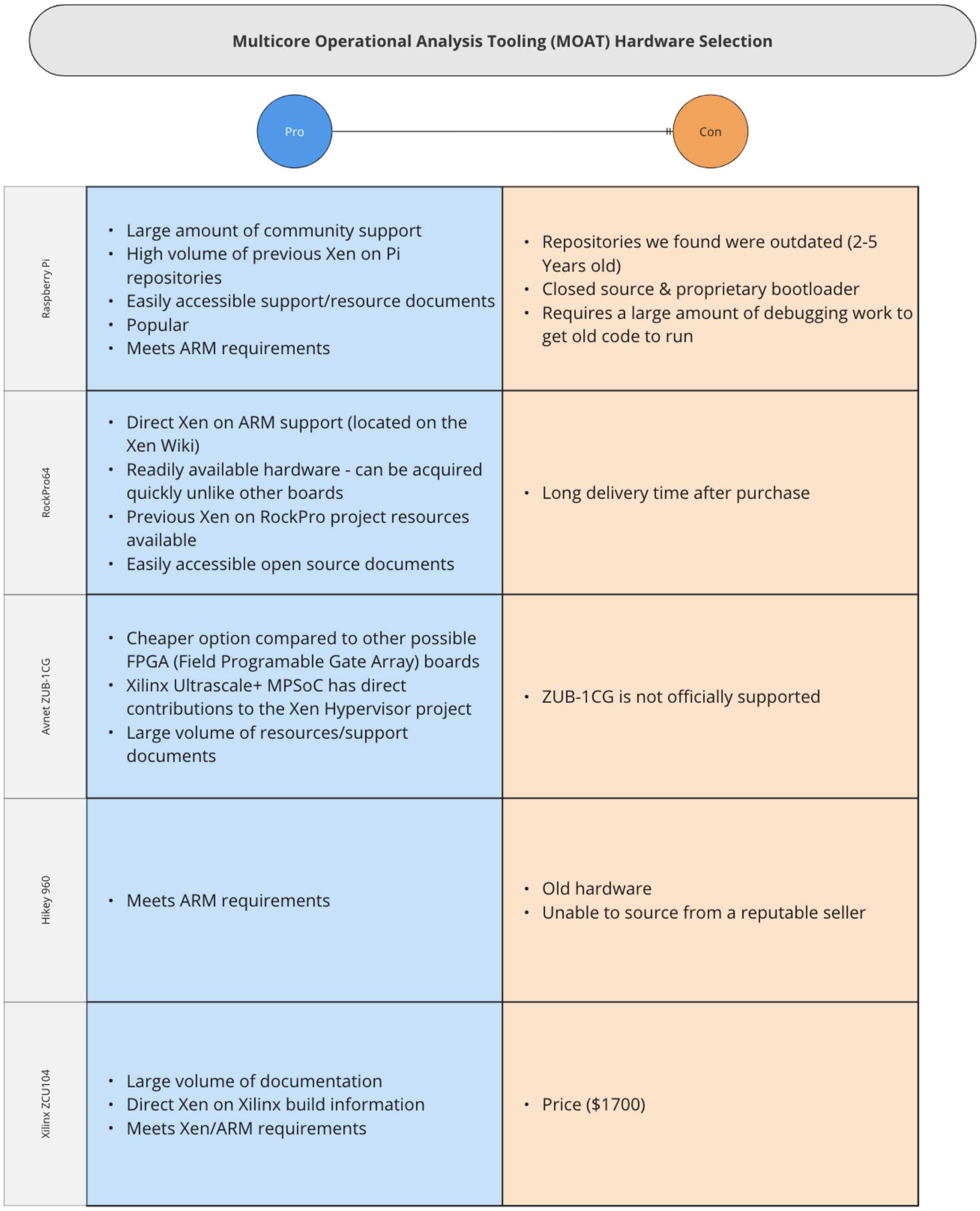
4.2.2.5 Xilinx ZCU104

The ZCU104 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. This information has led the team to consider this board as a potential hardware platform that would work well with Xen and multicore testing.

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 1. 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 RockPro 64. 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. As for the remaining FPGA board, the Xilinx, per Boeing's request, we are proceeding with this hardware as a backup in case of unique quirks or failures of our current primary system.



Multicore Operational Analysis Tooling (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 ZCU104	• Large volume of documentation • Direct Xen on Xilinx build information • Meets Xen/ARM requirements	• Price ($1700)

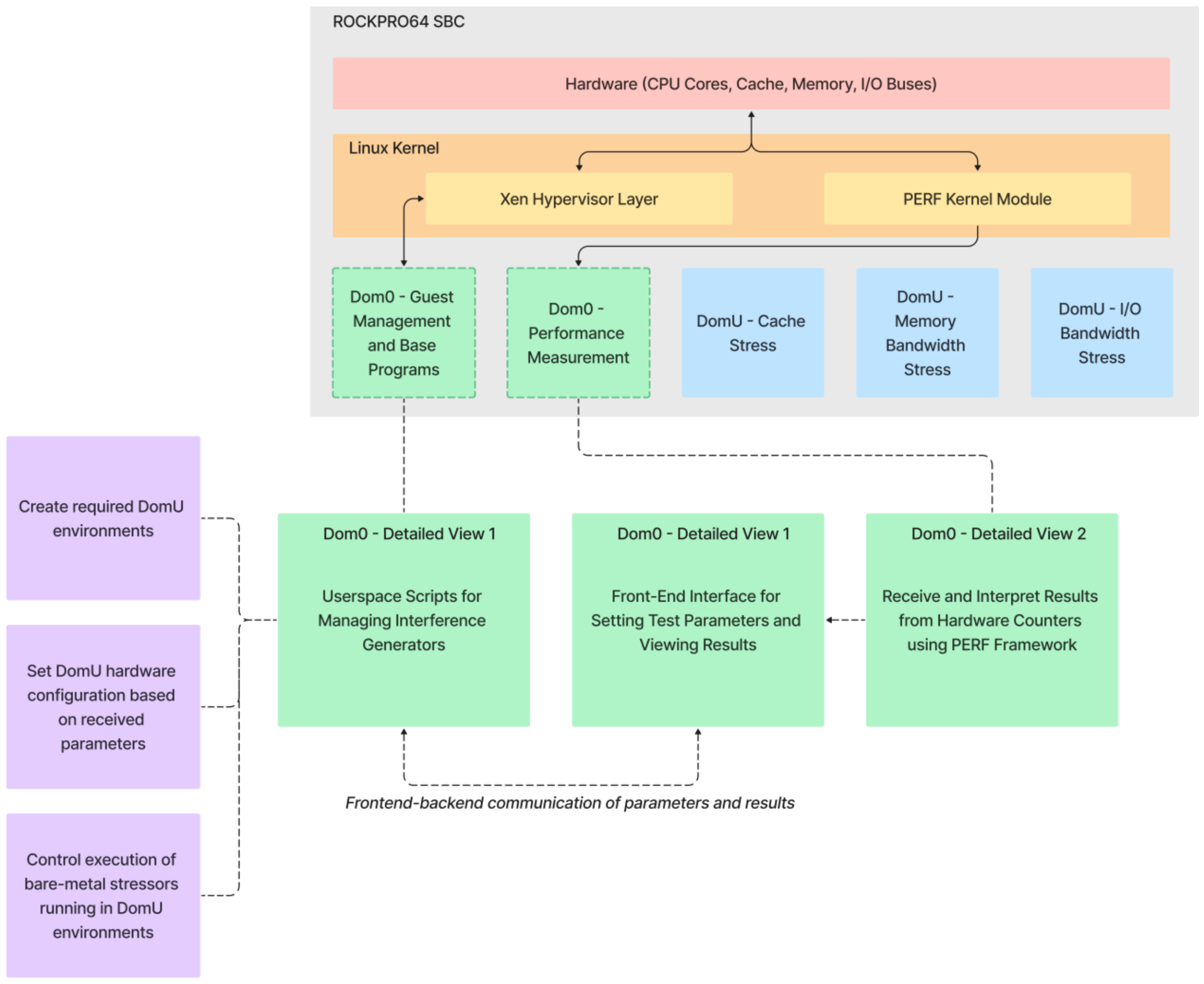
Figure 1: Decision matrix for hardware selection.

4.3 Proposed Design

4.3.1 Overview

Our current design utilizes a single-board computer (a Raspberry Pi would be a familiar example), that contains various pieces of hardware and is a fully functioning system while maintaining a small size. 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 can 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 describe the components and how they operate in a more technical capacity.

4.3.2 Detailed Design and Visual(s)



ROCKPRO64 SBC
Hardware (CPU Cores, Cache, Memory, I/O Buses)
Linux Kernel
Xen Hypervisor Layer
PERF Kernel Module
Dom0 - Guest Management and Base Programs
Dom0 - Performance Measurement
DomU - Cache Stress
DomU - Memory Bandwidth Stress
DomU - I/O Bandwidth Stress
Create required DomU environments
Set DomU hardware configuration based on received parameters
Control execution of bare-metal stressors running in DomU environments
Dom0 - Detailed View 1
Userspace Scripts for Managing Interference Generators
Dom0 - Detailed View 1
Front-End Interface for Setting Test Parameters and Viewing Results
Dom0 - Detailed View 2
Receive and Interpret Results from Hardware Counters using PERF Framework
Frontend-backend communication of parameters and results

Figure 2: Block diagram of the system, including hardware and software components and how they are connected.

Figure 1 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 as well as Xen and its associated domains, and the various actions that will be performed by the programs running under Xen.

We will begin with Xen - Xen lies within the Linux kernel and 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.

Our Xen DomO is responsible for several essential tasks. The first is managing guest domains and base programs that our team will be using to generate performance data. “DomO - Detailed View 1” in figure 1 above illustrates the actions that DomO will need to implement. The first is presenting the user with an interface capable of accepting test configuration parameters and displaying test results. This interface will be command-line based and will store test parameters, such as base test, selected points of resource contention (blue boxes in Figure 1), and any guest hardware properties (assigned CPU cores, amount of RAM, etc.) in a configuration file. The configuration file will then be read by the backend scripts, which are written in BASH, that will execute commands using the Xen management toolchain (“xl”) to instantiate one or more Xen DomU environments running bare-metal programs to initiate resource contention. The second task that DomO is responsible for performing is receiving and interpreting results from the PERF kernel module. The PERF module is a part of the Linux kernel which communicates with performance counters embedded in the hardware of the SoC and then makes the data available to user-space programs. Performance counters provide useful metrics such as processor cycles taken to execute a given program, and the number of cache misses. 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 Rockchip RK3399 SoC. 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. An illustration showing the channels that these methods take through the SoC are provided below.

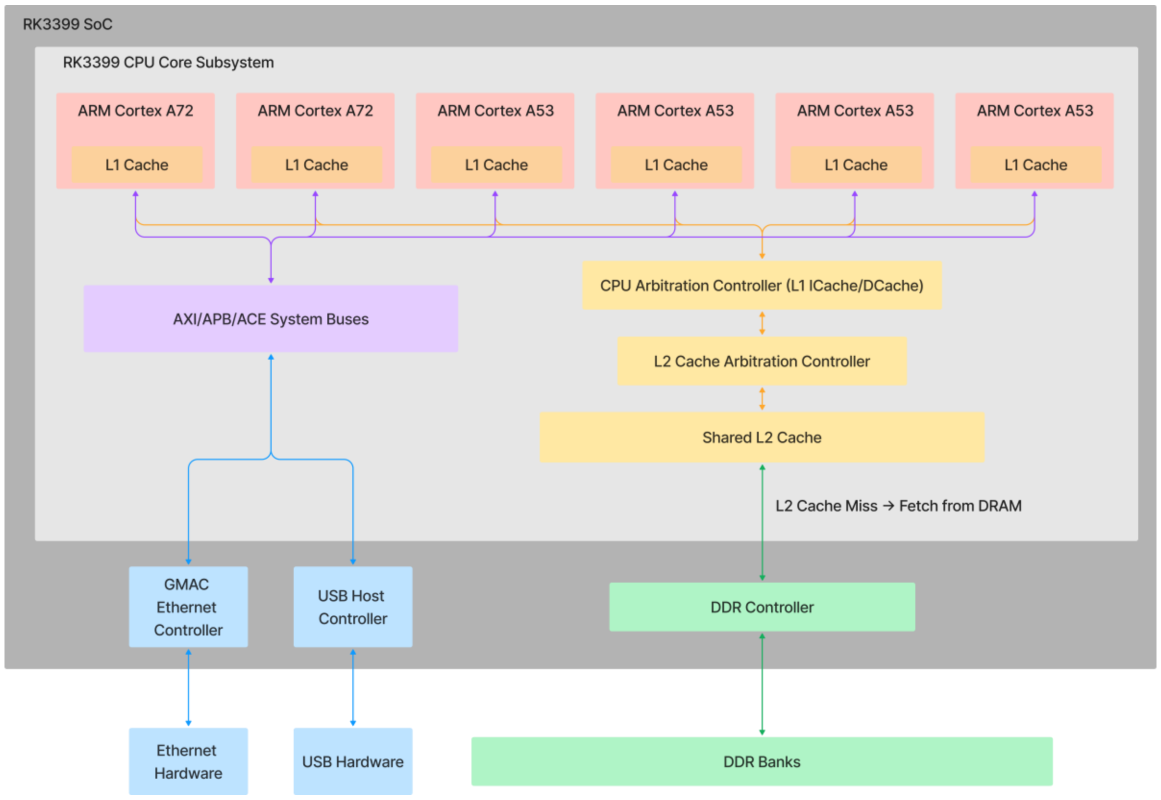


Figure 3: Components and interference channels on RK3399 SoC and associated hardware

The process of stressing the cache occurs through the CPU to cache data path, which can be seen in orange above. As can be seen, each processor core incorporates its own level 1 cache for instructions and data, however, all six 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 as the cache stress. Since cache misses necessitate an 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

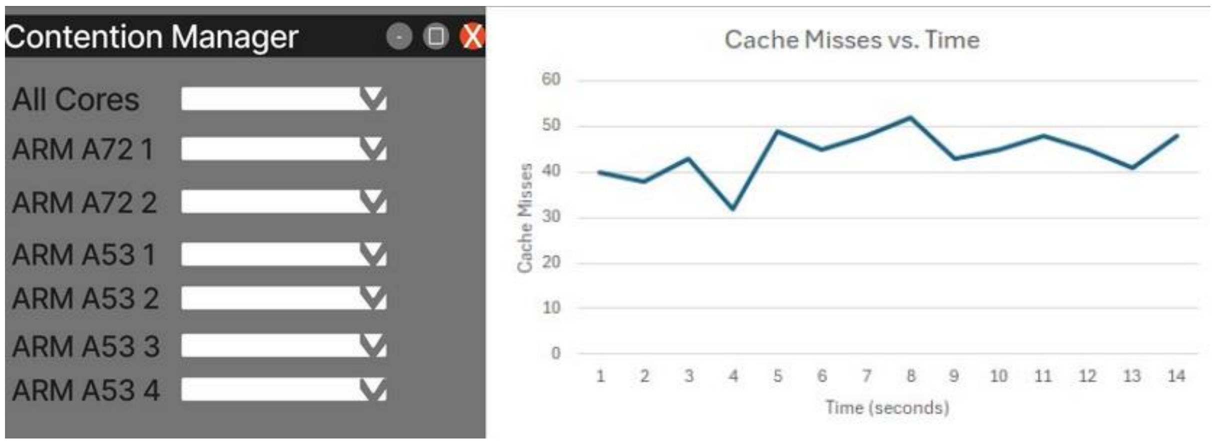


Figure 4: Wireframe diagram of a mock GUI user interface

Our end users have two options for interfacing with our design. The user will interact with our design primarily via a command line interface in which they type predefined commands that carry out various operations. Upon entering a command specifying several parameters, the application will run, and the tool will produce a file that details the performance output of the test. The command line tool allows more technically advanced users to interact with our tool in a programmatic fashion; that is, a user would be able to write scripts for the command line tool to run a variable number of tests and store the output somewhere. A final version of our design will also provide a visual GUI interface, as shown in figure 2, that outputs visual feedback on how the stressors are affecting the system's performance. The GUI allows the tool to be used in a more intuitive manner for less technically inclined users. Furthermore, the graphs the GUI produces are also of value to technical users when communicating results to non-technical users.

4.3.4 Areas of Concern and Development

At this stage of our project, our design is a rough outline of the major components of our client's requirements. There are many components to our design that require us to iteratively refine details as we work closely with our client week-to-week. Our design satisfies the hardware requirements in that we are working with an ARM based SoC running a Xen hypervisor. Currently, the team's effort is focused on identifying interference channels for our hardware platform. This is the primary interest of our client and deliverable of our design. Our client has also requested a command line-like utility that has all its functionality mirrored in a GUI interface. The final aspect of our design, the user interface, has been prototyped but not implemented.

Our team has several concerns for delivering a finished product to our client. The primary concern of our design is centered around time. To deliver a final product that satisfies all user requirements requires our team to closely adhere to an established project schedule to allow for adequate time in each stage of our design. Various technical aspects of our design may require debugging and research to be implemented correctly. Subsequent stages of the project may be hindered by getting stuck in the early stages.

Another concern is focused on how our design will capture the metrics relevant to our client's requirements. Ultimately, our design seeks to prove the Worst-Case Execution Time (WCET) for a given hardware platform, but we have not clearly defined how this can be achieved. If this aspect is missing from our design, the user will not be able to use the tool as intended and our client will not receive the product they requested.

Our weekly meetings with our client are likely the best method to get unstuck, clearly define the expectations of the final product, and properly address these concerns. To resolve our concerns of being blocked on technical issues, we can schedule time with the engineers our client includes in our weekly meetings to get unstuck. Regarding gathering performance metrics, we can articulate our concerns to our client and ask for guidance on how this information can be gathered from our current design.

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 RockPro64

The RockPro64 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. Furthermore, we were able to find abundant hardware documentation for the platform online between the official manufacturer and an online wiki. This aids our overall design in that the more knowledge we have of the underlying hardware, the better able we can write stressors for our platform.

Relative to other SBCs that we considered, like the Raspberry Pi 4 or HiKey 960, the RockPro64 was not only the easiest to work with but also the most well-documented. One weakness is that Xen may be easier or better suited to running on an FPGA like the Xilinx ZCU board. We speculate many embedded applications that use Xen run on an FPGA board rather than an SBC. Ultimately, its weaknesses are negligible when one considers that it is the only board that meets our design requirements, is easy to work with, and relatively cheap compared to other options.

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 (i.e., Stress-NextGeneration) 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.

4.4.4 Perf

Perf is a performance tool built into the Linux kernel that will allow us to gather performance metrics. This is extremely useful to our design when we are trying to characterize how our stressors are affecting victim applications. Its primary strengths are that it is built into the Linux kernel meaning we already have access to it on our target platform. Furthermore, perf also has extensive documentation which we can reference to better understand how to use it for different aspects of our design. The primary weakness in the context of our design is that it requires us to learn how to use the tool and integrate it into our larger design.

One alternative to perf is gprof, which is another performance analysis tool for Unix like applications. The drawback of using this approach is that our client has experience with using perf, so we would likely need to learn how to use a new tool to accomplish something that we already know how to use.

4.5 Design Analysis

At this point in the project, the team is focused on hardware bring-up and researching interference channels. For hardware bring up, the team is working with the RockPro64 to get a working version of Xen Hypervisor on it and make sure that it can be configured with the tools that we need. The team is also working on getting a Petalinux image built so that we can pursue an FPGA based backup platform for further development of the tools. For the interference channel research, the team has divided the interference channels into Cache, Memory Bandwidth, and I/O. The team has been spending time looking at the architecture of these systems and seeing how they can be exploited to interfere with other cores.

Using what the team has learned about the ARM architecture and Xen Hypervisor, the design has been tweaked and improved to attempt to minimize issues. The biggest issues appear during the hardware bring-up which is nearing completion. From there, time will be spent writing software to exploit the shared resources on the hardware. The team should be able to proceed with the planned design with little interference from scheduling or other unforeseen issues. The project is overall feasible, and the team is confident in the ability to complete it.