Multicore Operational Analysis Tooling (MOAT)

SENIOR DESIGN DEC'24 TEAM 09

Anthony Manschula, Alex Bashara, Joseph Dicklin, Hankel Haldin
Client: The Boeing Company
Advisors: Steve VanderLeest (Boeing), Joe Zambreno (ISU), Phillip Jones (ISU)

IOWA STATE UNIVERSITY

Project Plan & Management

Problem Statement

- Increasing computational demand of avionics programs necessitates higher performance systems
- Multicore systems can experience interference between shared resources
  - Can cause system to stall for access and introduce unpredictable delay
  - Practical example: Radio control program interferes with flight control program
- **Airworthiness certification: Must bound the worst-case execution time to guarantee that safety-critical processes complete in a certain amount of time**
- How can we create an efficient multicore system while maintaining maximum safety?

Conceptual Sketch

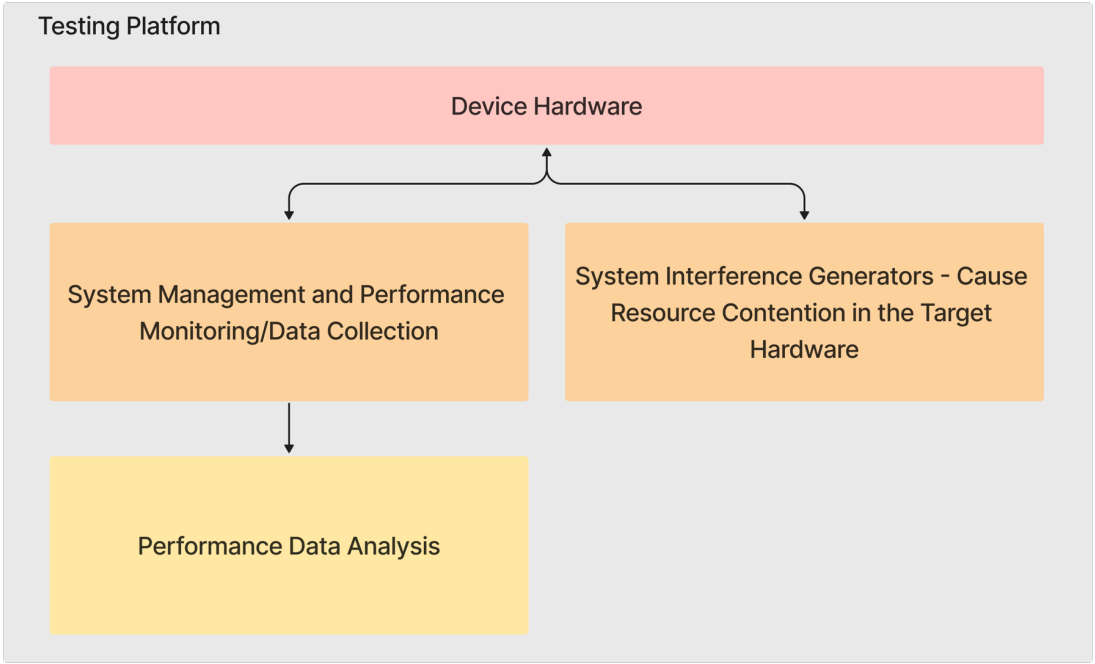


Figure 1: High-level view of the subsystems that comprise our design

Market Survey & Research

	Multicore Operational Analysis Tool (MOAT)	RapiDaemon	MASTECS	OTAWA (Open Tool for Adaptive WCET Analysis)	Multicore Test Harness
Pros	Open source, built with modern hardware in mind	Designed by a team of professional engineers, specifically, for compliance testing	Offers timing analysis software and multicore performance consulting	Open source, supports several ISAs (e.g., ARM, RISC-V, etc.)	Open source & has good instructions
Cons	Produced under tight time deadlines	Closed source & expensive	Potential portability issues to North America	No recent builds, not well-known	Very old; development stopped four years ago

Figure 2: Market research summary

Functional 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 (i.e., a 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

Non-Functional 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)
- **Hypervisor** – Our design must use a type 1 hypervisor (Xen) to partition underlying system resources

Other Constraints and Considerations

User & Developer 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

User Interface Requirements:

- Command-line utility for automated testing
- GUI for easy interpretation of results by less technical users

Project Risks & Mitigations

Risk	Mitigation
Challenges building hypervisor	Team leverages industry experts at Boeing
Team lacks info to implement effective base test cases and interference generators for the target platform	Selected hardware platform should have thorough documentation available
Project documentation is insufficient or ineffective	Ensure that documentation is created and updated for every task that team members complete
Project runs into issues regarding handoff and open-sourcing	Ensure all permissions and licensing are determined well in advance of the project's completion

Figure 3: Project Risks & Mitigations

Resource & Cost Estimate

- **Primarily a software project**
  - Linux, Xen, and build tools are open source
- **Requires an ARM based development board**
  - Xilinx ZCU106 - $3,234.00
  - Pine ROCKPro64 - $79.99

Project Milestones & Schedule

Deliverable	Delivery Date
A report showing functioning hardware along with installation scripts & documentation	April 5th, 2024
A report outlining the interference channels identified by research	April 11th, 2024
Presentation and report showing performance data generated from interference tests without any mitigations	April 29th, 2024
Comparative analysis report on interference tests with and without performance mitigations enabled	September 10th, 2024
Application with UI enhancements and a report detailing statistical analysis of WCET	October 31st, 2024

Figure 4: Project Milestones & Schedule

System Design

Hardware/Software Utilization

- **Hardware – Embedded Systems**
  - Pine ROCKPro64 Single-Board Computer
  - Xilinx ZCU106 FPGA Development Board
- **Software**
  - Linux for ARM-based Embedded Systems
    - Yocto Project & PetaLinux
  - Hypervisors
    - Xen

Component Decomposition

- **Hardware** – Consists of processor cores, memory, I/O, etc.
- **Xen Hypervisor** – Manages hardware resource allocation to domains (guests) running on the system
- **Domain 0 (Dom0)** - Linux environment manages configuration and operation of guest domains (DomU's), runs user interface utilities, and performs system performance monitoring
- **Guest Domains (DomU)** - Hosts interference generators in bare metal and Linux environments
- **Interference Generators** – Generate resource contention in shared hardware, such as L2 cache or main memory

Block-Level System Diagram

How do the system components interact?

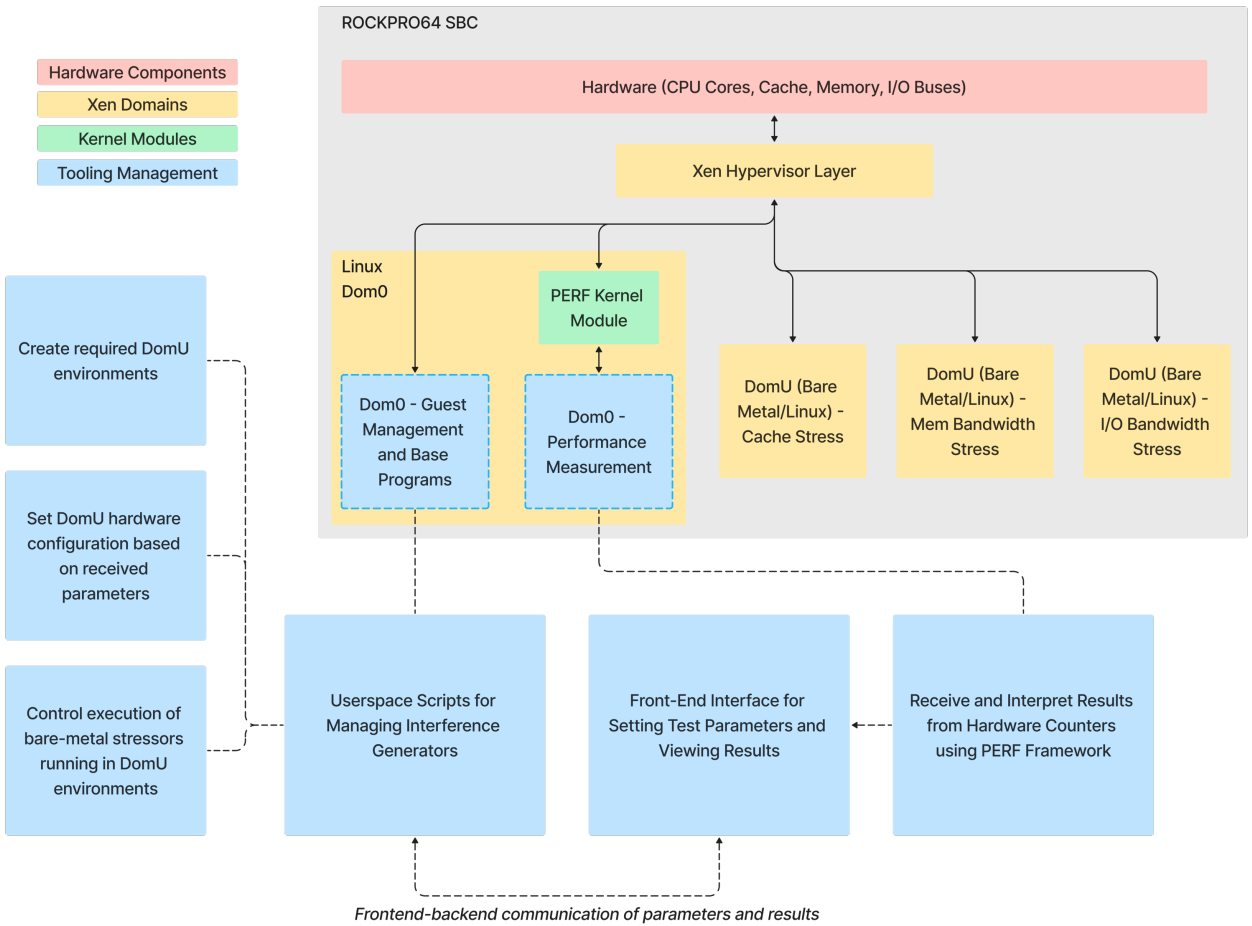


Figure 5: Block-Level System Diagram

Building Blocks – Xen Hypervisor

- A system with a functioning installation of Xen is critical to executing our project and fulfilling requirements
- Yocto Project and PetaLinux
  - Common frameworks for building embedded Linux images for ARM-based platforms
  - **Not trivial:** Including Xen in these builds requires hardware-specific changes
  - The team has documented processes and created scripts to ease future build efforts

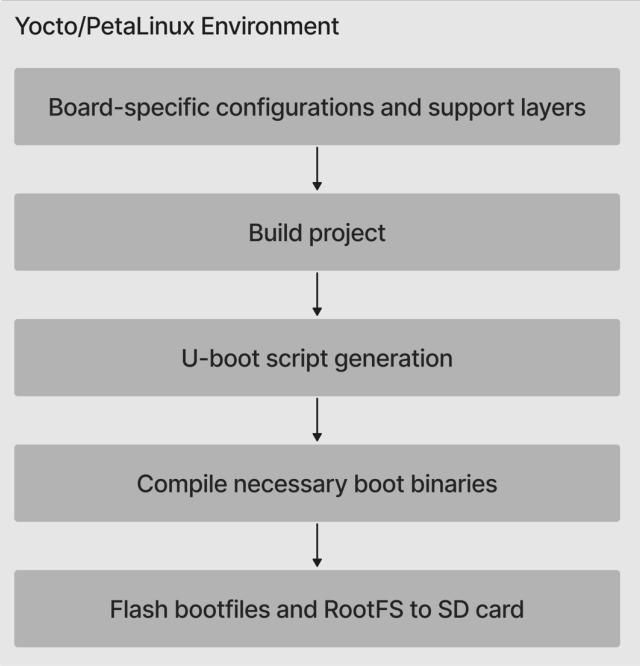


Figure 6: Image configuration and build process

Building Blocks – How is Interference Created?

- Caused by simultaneous utilization of shared resources both on and off the SoC
  - e.g., Cache, Memory, I/O
- A process on one core may have to wait for a process on another core to finish its task
  - Can lead to delays in processing
- One core may evict data that is needed by another core
  - Causes more cache misses and longer memory access times
- Can force interference by abusing the processor architecture and structure

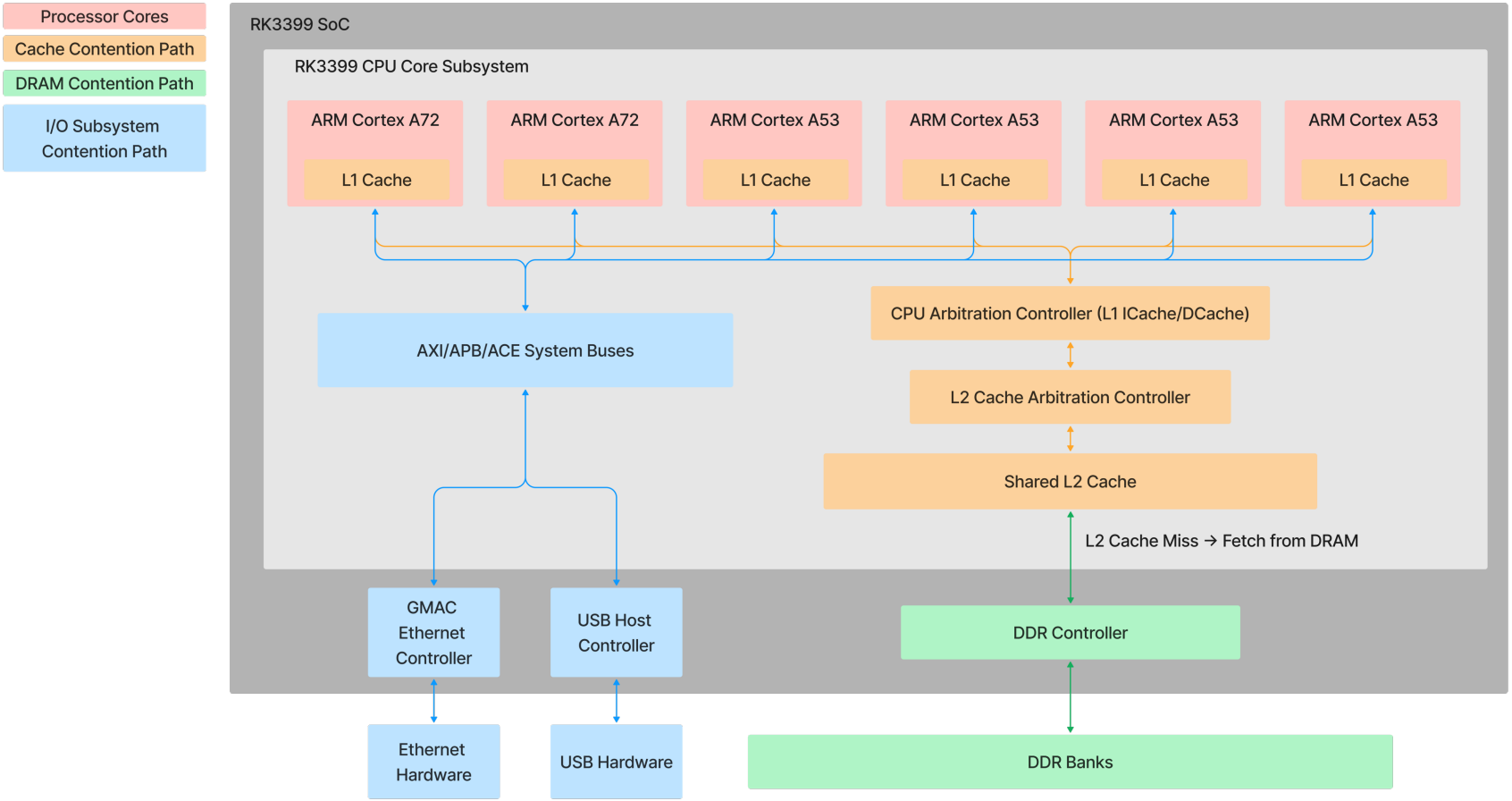


Figure 7: RK3399 SoC Shared Resource Diagram

Test Cases – Interference Case Development (1/2)

- Interference test cases are designed to induce maximum contention in the target resource
  - Goal: Create a worst-case scenario for our control/base program as it also tries to utilize that resource
- How do we make sure our test cases are effective?
  - Performance profiling
    - CacheGrind – Cache simulation: hits, misses, memory access
    - PERF – Measure hardware performance counters

Test Cases – Interference Case Development (2/2)

CacheGrind simulation for an interference generator provides estimated instruction and data cache hits and misses for a particular piece of code:

```
for (int i = 0; i < 262144 ; i++) {
    cacheLineArray3[i % 16384].array64B[0] = 0xABCD;
}
```

Figure 8: Excerpt of source code of work-in-progress memory interference generator

```
--------------------------------------------------------------------------------
-- Annotated source file: /home/tonymanschula/Desktop/sddec24-09/Programs/Mem/mem_stress_cache.c
--------------------------------------------------------------------------------
Ir____________ I1mr____ ILmr____ Dr____________ D1mr____ DLmr____ Dw____________ D1mw__________ DLmw__________
2,883,584 (30.1%) 0       0       524,288 (24.6%) 0       0       262,144 (32.8%) 262,143 (33.3%) 262,143 (33.3%)
```

Figure 9: CacheGrind simulation output for source in Figure 8

Conclusion

Current Project Status

- Hardware bring up proved to be more time consuming than we initially anticipated
  - Largely due to the time investment of researching and debugging hardware issues
  - We still managed to get a functioning system (April 5th vs. April 14th)
- Despite overlap, we continued our research on each candidate platform
  - This allowed us to make progress developing base cases for our platform
- Developed a strong foundation to continue stress test program development
  - Interference testing report will be completed when class resumes in the fall (originally April 29th)

Task Responsibility and Contributions

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

Plans for Future Work

- Generate interference
- Mitigate interference
- Create a frontend user interface
- Explore other forms of interference
    - Cache Coherency
- Identify existing gaps in preparation for project hand-off

Questions?

Supplemental Material

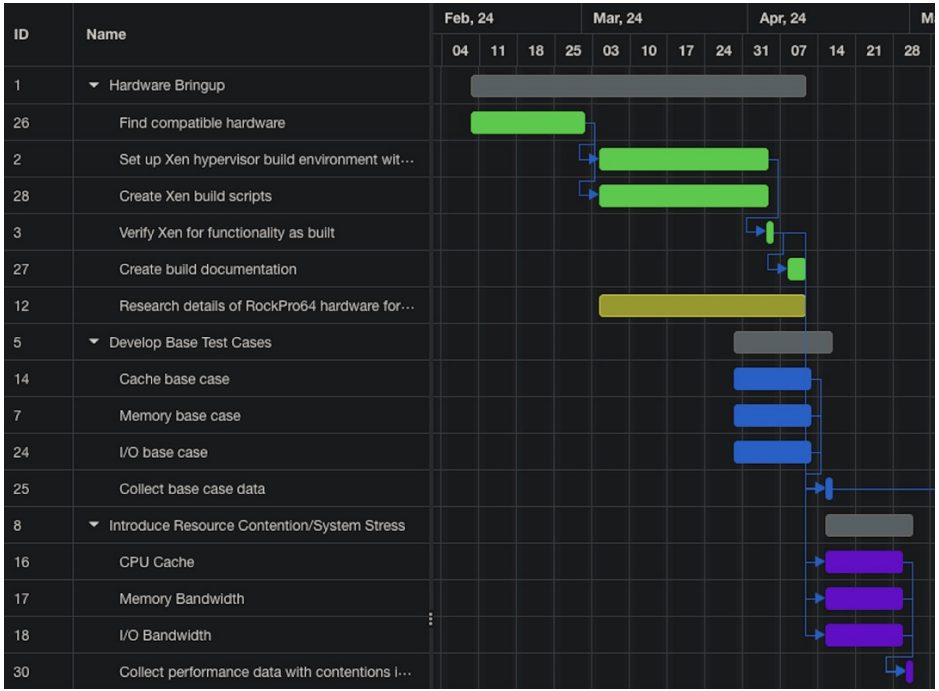
Important Engineering Standards and Advisories

- FAA AC20-193
  - This advisory is concerned with the use and compliance of multi-core processors in avionics systems.
- CAST-32A
  - Position paper arguing on safety, performance, and integrity of airborne software operating on multicore systems.
- ARINC 653
  - Defines acceptable methods of resource partitioning on hardware running avionics programs

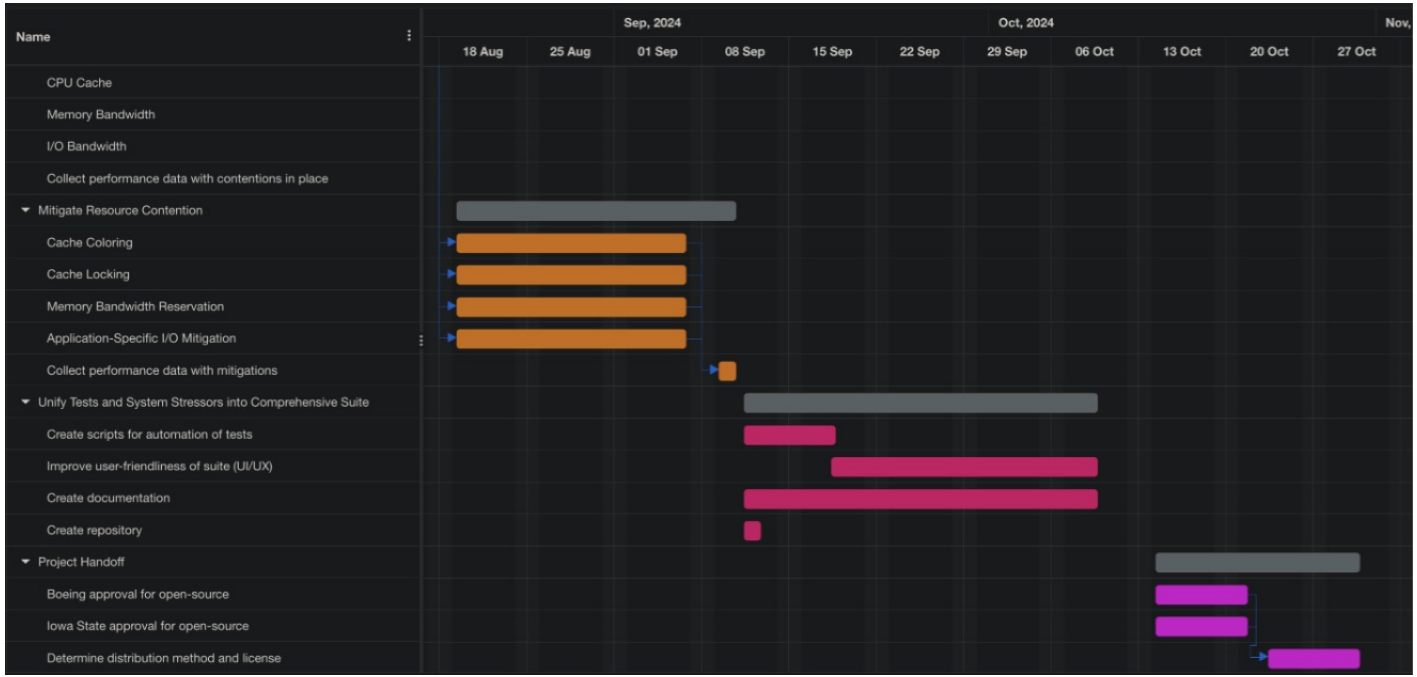
Users

- Avionics Engineers
  - Responsible for developing and validating avionics systems
  - Need a stress testing tool for their ARM-based hardware development platform
    - Allows for effective validation of their work as engineers
- Avionics Engineering Managers
  - Manage a team of Avionics Engineers
  - Provide evidence that the projects they are managing can be certified under military and civilian authority

Project Timeline (Gantt Chart)

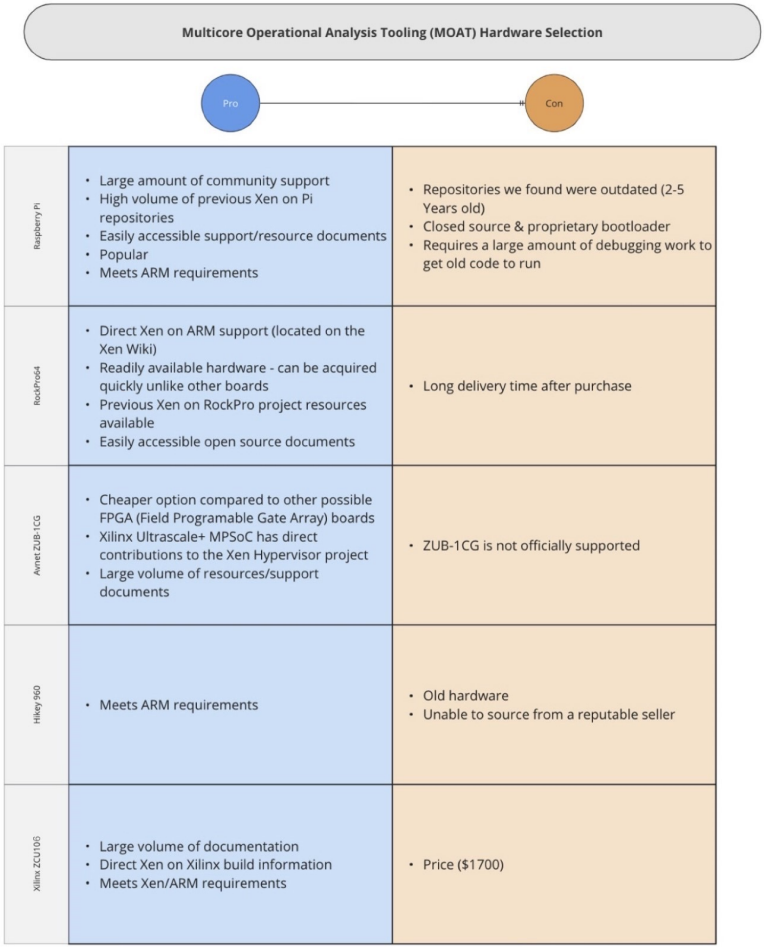


Project Timeline (Gantt Chart)

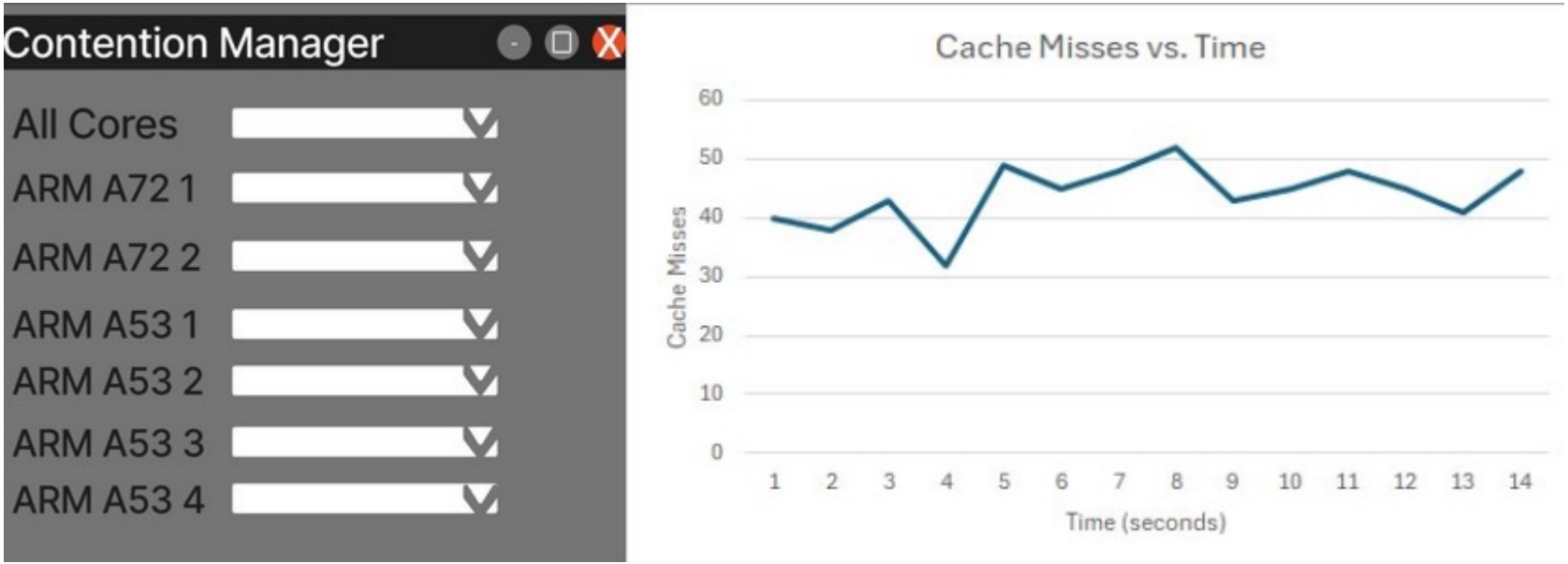


Hardware Selection Matrix

- Selection process involved evaluating several different platforms for cost, features, and support for the tools necessary for our project



GUI Example


Contention Manager
All Cores
ARM A72 1
ARM A72 2
ARM A53 1
ARM A53 2
ARM A53 3
ARM A53 4
Cache Misses vs. Time
Cache Misses
60
50
40
30
20
10
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Time (seconds)

Source Code of Mem Stress Program

```
struct cacheLine {
    //64 bytes total per struct
    //L2 has 64-byte lines
    unsigned int array64B[16];
};
```

```
int main(int argc, char* argv[]) {
    // printf("cacheLine size is %d bytes\n", sizeof(struct cacheLine));

    //1MiB arrays
    //L2 on the RK3399 is 1MiB, 16-way associative
    cacheLineArray1 = (struct cacheLine*)malloc(16384 * sizeof(struct cacheLine));
    cacheLineArray2 = (struct cacheLine*)malloc(16384 * sizeof(struct cacheLine));
    cacheLineArray3 = (struct cacheLine*)malloc(16384 * sizeof(struct cacheLine));

    signal(SIGINT, graceful_exit);


    for (unsigned int i = 0; i < 262144 * 1000 ; i++) {
        /*
            Each array is 1MiB, and L2 is 1MiB total. It should never be possible for any
            of these arrays to exist entirely in L2 or L1 given the sequential access order.
            By the time we loop back to the first index of the cacheLineArrays, the cached data for that
            index will already have been ejected ~11000 iterations ago.
        */
        cacheLineArray1[i % 16384].array64B[0] = 0xDEAD;
        cacheLineArray2[i % 16384].array64B[0] = 0xBEEF;
        cacheLineArray3[i % 16384].array64B[0] = 0xABCD;
    }

    free(cacheLineArray1);
    free(cacheLineArray2);
    free(cacheLineArray3);
}
```

Example Perf Output

```
xilinx-zcu106-20231:~$ perf stat -e cpu_cycles,l2d_cache,l2d_cache_refill,mem_access ./
mem_stress_cache_aarch64

 Performance counter stats for './mem_stress_cache_aarch64':

      25916761481      cpu_cycles:u
       1631947979      l2d_cache:u
        779192265      l2d_cache_refill:u
       3145736108      mem_access:u

     21.777850882 seconds time elapsed

     21.766157000 seconds user
      0.000000000 seconds sys
```