Degree Project in Computer Science and Engineering

First cycle, 30 credits

Hardware Extensions for Memory Capability Revocation in the S3K Framework

A Software-Managed Capability Lookaside Buffer Architecture

XIAOTIAN JIANG

Hardware Extensions for Memory Capability Revocation in the S3K Framework

A Software-Managed Capability Lookaside Buffer Architecture

XIAOTIAN JIANG

Master's Programme, Embedded Systems, 120 credits
Date: June 23, 2026

Supervisor: Roberto Guanciale
Examiner: Ingo Sander
School of Electrical Engineering and Computer Science
Swedish title: Hårdvarustöd för återkallning av minneskapabiliteter i S3K-ramverket
Swedish subtitle: En mjukvaruhanterad arkitektur för en Capability Lookaside Buffer

© 2026 Xiaotian Jiang

Abstract

Capability-based systems provide a principled way to isolate software components and manage access to shared resources. In mixed-criticality embedded systems, however, capability revocation must be both secure and temporally predictable. The S3K partitioning kernel addresses this problem by storing derived capabilities in a contiguous capability table, but its baseline memory-capability revocation still relies on a software loop and system-call path. This thesis proposes a software-managed Capability Lookaside Buffer (CLB) for S3K and implements it as an in-core RISC-V extension in CVA6. The design introduces user-level and machine-level CLB instructions, a hardware capability cache for fast memory access checks, and a command state machine for memory-backed capability operations such as fetch, delete, and revoke. The mechanism is first explored architecturally in gem5 and then integrated into CVA6 RTL for Verilator-based evaluation. The results show that capability fetch decreases from 631 cycles to 19 cycles, capability deletion decreases from 617 cycles to 39 cycles, and revocation achieves speedups between 11.70x and 26.50x across the measured subtree sizes. The CLB hit path for ordinary loads and stores introduces no observable cycle overhead in the reported measurements. FPGA implementation on a Cheshire-based CVA6 SoC targeting a Digilent Genesys 2 board adds 6.12 percent LUTs and 3.00 percent flip-flops at the full-SoC level, without increasing BRAM or DSP usage. These results show that S3K's deterministic capability-management model can be accelerated by a concrete in-core RISC-V RTL extension while preserving software control over policy.

Keywords

Capability systems, RISC-V, S3K, memory revocation, real-time systems

Sammanfattning

Kapabilitetsbaserade system erbjuder ett principfast sätt att isolera programvarukomponenter och hantera åtkomst till delade resurser. I inbyggda system med blandad kritikalitet måste återkallning av kapabiliteter dock vara både säker och tidsmässigt förutsägbar. Partitioneringskärnan S3K hanterar detta genom att lagra härledda kapabiliteter i en sammanhängande kapabilitetstabell, men dess grundläggande återkallning av minneskapabiliteter bygger fortfarande på en mjukvaruloop och systemanropsväg. Denna avhandling föreslår en mjukvaruhanterad Capability Lookaside Buffer (CLB) för S3K och implementerar den som en kärnintern RISC-V-utökning i CVA6. Konstruktionen introducerar CLB-instruktioner för användar- och maskinläge, en hårdvarubaserad kapabilitetscache för snabba åtkomstkontroller till minne samt en kommandostyrd tillståndsmaskin för minnesbaserade kapabilitetsoperationer såsom hämtning, borttagning och återkallning. Mekanismen undersöks först arkitekturellt i gem5 och integreras därefter i CVA6 RTL för utvärdering med Verilator. Resultaten visar att hämtning av kapabiliteter minskar från 631 cykler till 19 cykler, borttagning av kapabiliteter minskar från 617 cykler till 39 cykler, och återkallning uppnår hastighetsökningar mellan 11,70x och 26,50x för de uppmätta delträdsstorlekarna. CLB-träffvägen för vanliga laddningar och lagringar medför ingen observerbar cykelkostnad i de rapporterade mätningarna. FPGA-implementering på ett Cheshire-baserat CVA6-SoC för ett Digilent Genesys 2-kort ökar resursanvändningen med 6,12 procent LUT:ar och 3,00 procent vippor på fullständig SoC-nivå, utan att öka användningen av BRAM eller DSP. Dessa resultat visar att S3K:s deterministiska modell för kapabilitetshantering kan accelereras med en konkret kärnintern RISC-V RTL-utökning samtidigt som policykontrollen bevaras i mjukvara.

Nyckelord

Kapabilitetssystem, RISC-V, S3K, memory revokering, realtidssystem

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Roberto Guanciale, for his guidance throughout this thesis project. His feedback on the CLB design helped me refine the hardware/software boundary, clarify the role of the S3K capability table, and better understand the trade-offs between software-managed policy and hardware acceleration. Our discussions also guided me to explore capability-based systems more broadly, including work such as CAPSTONE and the historical perspective provided by Levy's book on capability-based computer systems. Without his guidance, I would likely not have developed the same understanding of the computer-security research context that surrounds this work. I greatly appreciate these discussions and the direction they provided during the project.

I am also deeply grateful to Henrik Karlsson, whose foundational work on the S3K framework inspired this hardware acceleration project. I greatly appreciate his willingness to share his insights and for his technical assistance regarding the S3K capability mechanism.

Furthermore, I would also like to thank my examiner, Ingo Sander, for evaluating this thesis and for providing the academic framework in which the project was carried out. I am grateful to KTH Royal Institute of Technology and the School of Electrical Engineering and Computer Science for providing the environment and resources that made this work possible.

Finally, my deepest thanks go to my family and friends for their unwavering moral support, understanding, and encouragement throughout my master's studies at KTH Royal Institute of Technology in Stockholm. Their encouragement was important during the long process of developing, implementing, and writing this thesis.

I also acknowledge the use of generative AI tools, including ChatGPT, during the preparation of this thesis. These tools were used primarily for translation, language refinement and code asssitance. All architectural design, implementation, experimental results, analysis, and conclusions presented in this thesis are my own work.

Stockholm, June 2026
Xiaotian Jiang

Stockholm, June 2026
Xiaotian Jiang

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Problem . . . 2
1.2.1 Original problem and definition . . . 2
1.2.2 Scientific and engineering issues . . . 3
1.3 Purpose . . . 4
1.4 Goals . . . 4
1.5 Research Methodology . . . 5
1.6 Delimitations . . . 5

2 Background **7**
2.1 Motivating End Applications . . . 7
2.1.1 The Broader Context of Mixed-Criticality Systems . . . 7
2.1.2 Dynamic Memory Reallocation in Avionics . . . 8
2.1.3 Temporal Memory Safety Bottleneck . . . 8
2.1.4 The Need for Hardware-Accelerated Memory Revocation . . . 9
2.2 State of the Art in Memory Protection . . . 10
2.2.1 Page-Table-Based Memory Protection . . . 10
2.2.2 Tagged Memory . . . 11
2.2.3 Capability-Based Protection . . . 11
2.2.4 Comparative Analysis . . . 13
2.3 Capability-Based Systems . . . 14
2.3.1 Historical Capability Systems . . . 14
2.3.2 CHERI-Style Tagged Capabilities . . . 15
2.3.3 Software-Managed Capability Systems . . . 16
2.3.4 Hardware-Native Capability Architectures . . . 16
2.3.5 Contiguous Software Tracking and the S3K Approach 17
2.4 Hardware-Software Co-Design for Privileged Operations . . . 19

2.4.1 The HW/SW Co-Design Sweet Spot for Capabilities . 19
2.4.2 Hardware-Driven Real-Time Schedulers 19
2.5 Summary 20

3 Methodology and Architectural Design 23
3.1 Research Paradigm and Method Selection 23
3.1.1 Rationale for the Chosen Methodology 23
3.1.2 Development Process for RISC-V Hardware Extensions 24
3.1.3 Rejected Methods: Machine-Mode ISA Extension . . 25
3.1.4 Societal, Ethical, and Sustainability Aspects 26
3.2 Hardware Acceleration and Proposed Architecture Models . . 27
3.2.1 The Take-Grant Protection Model and Hardware Boundaries 27
3.2.2 User-Level RISC-V Custom Instructions 28
3.2.3 Hardware Acceleration of Tree Traversals 28
3.2.4 Software-Managed Capability Lookaside Buffer (CLB) 29
3.2.5 WCET Evaluation Requirements and Hardware Properties 30
3.3 Implementation Roadmap and Evaluation Framework 31
3.3.1 Architectural Modeling 31
3.3.2 RTL Integration 31
3.3.3 RTL Measurement 31
3.4 Timing and Predictability Integration 32
3.4.1 Test environment 32
3.4.2 Hardware/Software to be used 32
3.5 Assessing reliability and validity of the data collected 32
3.5.1 Validity of method 32
3.5.2 Reliability of method 33
3.5.3 Data validity 33
3.5.4 Reliability of data 33
3.6 Planned Data Analysis 33
3.6.1 Data Analysis Technique 34
3.6.2 Software Tools 34
3.7 Evaluation framework 34
3.8 System documentation 34

4 Microarchitectural Design and Implementation 37
4.1 CLB Microarchitecture and Hardware Interfaces 37
4.2 Decoupled Configuration and Verification Latch 38

4.3 Custom RISC-V ISA Extensions . . . 39
4.3.1 Return Value Contract . . . 39
4.4 Execution Flows and Hardware State Machines . . . 40
4.4.1 Kernel Trap Handler Flow (`mclb.fill`) . . . 40
4.4.2 User-Mode Fetch Flow (`uclb.get`) . . . 41
4.4.3 User-Mode Deletion Flow (`uclb.delete`) . . . 42
4.4.4 User-Mode Revocation Flow (`uclb.revoke`) . . . 42
4.5 Simulator Integration (gem5) . . . 44
4.6 Register-Transfer Level (RTL) Implementation . . . 47
4.6.1 Purpose of the RTL Implementation . . . 47
4.6.2 CVA6 Execution Model and Integration Point . . . 47
4.6.3 Access-Control and Memory-Command Paths . . . 49
4.6.4 Implementation Constraints and Verilator Simulation . . . 49

5 Results and Analysis **51**
5.1 Experimental Setup . . . 51
5.2 RTL Capability Operation Latency . . . 52
5.3 RTL Access-Control Overhead . . . 53
5.4 RTL Revocation Scaling . . . 53
5.5 FPGA Resource Overhead . . . 55
5.6 Discussion . . . 57
5.6.1 Latency Improvement Analysis . . . 57
5.6.2 Revocation Scaling . . . 57
5.6.3 Predictability and WCET Implications . . . 58
5.6.4 Access-Control Fast Path . . . 59
5.6.5 Implementation Cost . . . 60
5.6.6 Limitations and Future Work . . . 60

6 Conclusion **61**

References **63**

List of acronyms and abbreviations

ADI	Application Data Integrity
CDT	Capability Derivation Tree
CHERI	Capability Hardware Enhanced RISC Instructions
CLB	Capability Lookaside Buffer
CSR	Control and Status Register
ctable	capability table
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
IPI	Inter-Processor Interrupt
ISA	Instruction Set Architecture
LSU	Load/Store Unit
MCS	Mixed-Criticality System
MMU	Memory Management Unit
MTE	Memory Tagging Extension
PMP	Physical Memory Protection
PTE	Page Table Entry
RTL	Register-Transfer Level
S3K	Secure and Safe Separation Kernel
SoC	System-on-Chip
SPM	Scratchpad Memory
TCB	Trusted Computing Base
TLB	Translation Lookaside Buffer
UART	Universal Asynchronous Receiver/Transmitter
WCET	Worst-Case Execution Time

Chapter 1

Introduction

This chapter describes the specific problem that this thesis addresses, the context of the problem, the goals of this thesis project, and outlines the structure of the thesis.

1.1 Background

Memory safety vulnerabilities have been a persistent and dominant problem in system programming for decades. Software written in low-level languages like C and C++ is highly susceptible to memory corruption, which often leads to critical security exploits. These foundational vulnerabilities become significantly more dangerous in modern Mixed-Criticality System (MCS). In end application scenarios such as avionics, autonomous driving domain controllers, and Industrial IoT (IIoT), hardware resources must be dynamically shared among components with largely different real-time constraints and trust levels [1]. For example, avionics systems must strictly guarantee the Worst-Case Execution Time (WCET) of critical flight control loops while safely and dynamically allocating residual CPU time to variable communication tasks. Similarly, autonomous driving controllers must isolate hard real-time trajectory planning from vulnerable services [2].

To enforce strict physical and temporal isolation without sacrificing this agile resource sharing, these systems increasingly rely on the concept of capabilities, which are unforgeable tokens of authority that represent access rights to specific hardware resources, such as memory or time slices. By granting and delegating capabilities, a trusted system monitor can securely coordinate resources based on the system's operational phase (e.g., an aircraft switching from cruise to landing). Driven by these

rigorous security and isolation demands, the research community and industry have developed hardware capability-based architectures, most notably the Capability Hardware Enhanced RISC Instructions (CHERI) [3]. CHERI provides a robust hardware foundation for spatial memory safety (preventing buffer overflows) by extending traditional pointers with architectural metadata—such as base, bounds, and permissions—to strictly control memory access at the hardware level.

However, while CHERI effectively mitigates spatial violations, the dynamic nature of MCS inherently requires privileges to be frequently reclaimed. When memory is freed or a time slice is withdrawn, any dangling capabilities remaining in the system must be invalidated before the underlying resource can be safely reused. Existing software-based solutions for capability revocation, such as CHERIvoke [4] and Cornucopia [5], utilize a ”quarantine and sweep” model that requires scanning the entire memory space. Because the execution time of this global sweep depends heavily on the total memory size and the density of pointers, the process is fundamentally non-deterministic and yields unpredictable WCET.

To overcome this non-determinism, the Secure and Safe Separation Kernel (S3K) [6] framework introduces a novel capability-based multi-core partitioning kernel designed for embedded RISC-V systems. S3K provides spatial and temporal isolation by treating execution time as a first-class capability and organizing all derived capabilities into a contiguous, pre-allocated memory region known as a capability table (ctable). Although this contiguous data structure algorithmically bounds the revocation process, S3K currently executes this revocation via an incremental software loop. This project focuses on designing hardware extensions to accelerate this specific revocation mechanism, aiming to bridge the gap between S3K’s strict real-time safety models and practical microarchitectural performance.

1.2 Problem

The core issue lies in the tension between strong temporal memory safety and the strict deterministic timing requirements of real-time systems.

1.2.1 Original problem and definition

For safety-critical systems (e.g., avionics, automotive electronics), execution must be highly predictable. Mainstream capability revocation methods (like CHERIvoke) rely on sweeping memory, which causes unpredictable execution

pauses and violates the WCET constraints required by real-time schedulers. S3K avoids global memory sweeping by restricting capability storage to a centralized ctable. However, the current implementation of capability revocation in S3K relies entirely on an incremental software loop that iterates through the ctable to read and modify capability metadata. This software-driven approach acts as a significant performance bottleneck in the kernel's execution flow.

1.2.2 Scientific and engineering issues

For hard real-time systems, establishing a precise Worst-Case Execution Time (WCET) is a fundamental requirement. However, deriving strict WCET bounds using traditional methods—such as Static Deterministic Timing Analysis (SDTA) or Measurement-Based Probabilistic Timing Analysis (MBPTA)—is difficult when the underlying execution is subjected to unpredictable microarchitectural interference [7].

Although S3K structurally limits capability searches to a contiguous ctable, executing this revocation via software deeply contradicts the goal of deterministic microarchitectural execution. Specifically, this software-driven approach exhibits three critical engineering flaws:

- **Lack of Atomicity:** Because a single revocation might require deleting dozens of child capabilities, doing so in a software loop without preemption would block the kernel from handling urgent interrupts. Consequently, S3K has to insert artificial preemption points into the loop to ensure predictability.

- **Context Switch Overhead:** Current capability management relies on user-level processes trapping into the kernel to execute capability operations via system calls. This introduces significant context switch overhead, forcing the processor to save the execution state, execute the multi-instruction ctable traversal, and restore the state, which heavily degrades overall system performance.

- **Execution Latency and Pipeline Vulnerability:** Reading, evaluating, and clearing capability metadata word-by-word across multiple clock cycles is inherently slow. The time complexity of $O(C)$ (where C is the size of ctable) leaves the software loop highly vulnerable to microarchitectural stalls (such as data hazards and cache misses) that dedicated hardware state machines could easily bypass.

1.3 Purpose

The purpose of this thesis is to bridge the gap between the theoretical model of S3K for real-time safety and its practical microarchitectural performance. By shifting the core revocation loop from software into a custom RISC-V hardware primitive, this project aims to reduce the performance overhead of capability revocation by orders of magnitude. Fulfilling this purpose will benefit the safety-critical and security-critical systems industry (such as avionics and IoT), which urgently requires memory safety solutions that do not violate strict real-time performance and predictability. Furthermore, the project contributes to the academic community by exploring the feasibility of hardware/software co-design for secure separation kernels on the RISC-V architecture.

1.4 Goals

The overall goal of this project is to design, model, and evaluate a set of custom RISC-V hardware extensions—specifically introducing a Software-Managed Capability Lookaside Buffer (CLB) architecture—to accelerate the S3K kernel's ctable-based revocation model. This goal is divided into the following four sub-goals:

1. **Architectural Co-Design:** Design hardware primitives and microarchitectural structures for capability revocation, defining the interaction between the processor pipeline and the memory subsystem.

2. **Architectural Modeling:** Build an architectural model of the proposed instructions and CLB behavior to explore the design before committing to Register-Transfer Level (RTL) integration.

3. **RTL Implementation:** Implement the proposed CLB mechanism in a real RISC-V RTL core, using CVA6 [8] as the integration target.

4. **Quantitative Evaluation:** Measure and analyze operation latency, access-control overhead, and revocation scaling using the CVA6 RTL implementation under Verilator [9].

1.5 Research Methodology

This project adopts a structured, iterative methodology combining analytical study, architectural co-design, and simulation-based evaluation:

- **Architectural Co-Design:** Formulate the new hardware instructions and the CLB architecture. Specify revocation metadata and design the state machine required for contiguous table traversal.
- **Architectural Modeling:** Use gem5 [10] as an exploratory model for the custom instruction semantics and high-level CLB behavior.
- **RTL Evaluation:** Integrate the CLB into CVA6 and collect cycle-level results from the Verilator RTL simulation.

1.6 Delimitations

To maintain a focused scope, the following delimitations are applied to this project:

- **Target Framework:** The hardware acceleration is strictly tailored to the S3K partitioning kernel and its specific Capability Derivation Tree (CDT) and ctable layout. It is not designed to accelerate general-purpose capability systems like CheriBSD.
- **Target Architecture:** The custom extensions are designed exclusively for the open-standard RISC-V Instruction Set Architecture (ISA).
- **Evaluation Scope:** Primary performance validation relies on Verilator simulation of the CVA6 RTL implementation, while gem5 is used only as an architectural modeling aid. Full chip tape-out and board-level deployment are out of scope.

Chapter 2

Background

This chapter provides the foundational knowledge required to understand the context of the thesis. It outlines the motivating end applications, reviews memory protection and access-control mechanisms, introduces capability-based systems, establishes the theoretical security models governing access control, and explores existing architectural trends in transferring privileged software management to hardware.

2.1 Motivating End Applications

The architectural design of the S3K [6] kernel and the proposed Capability Lookaside Buffer (CLB) are fundamentally motivated by the rigorous requirements of dynamic Mixed-Criticality Systems (MCS). In these environments, hardware resources must be agilely reallocated without violating hard real-time constraints.

2.1.1 The Broader Context of Mixed-Criticality Systems

Modern embedded systems—ranging from autonomous driving platforms and industrial robotics to advanced aerospace engineering—are increasingly consolidating components of varying criticality onto single multi-core Systems-on-Chip (SoCs). In a traditional static system, memory and execution time are rigidly partitioned at boot time, which ensures isolation but leads to severe resource underutilization.

To handle complex and highly variable modern workloads, next-generation MCS must support dynamic resource sharing. However, allowing untrusted

components to dynamically share and release physical memory introduces a critical tension: the system must retain the flexibility of dynamic allocation without ever compromising the deterministic Worst-Case Execution Time (WCET) guarantees required by safety-critical control loops.

2.1.2 Dynamic Memory Reallocation in Avionics

To illustrate this tension, we refer to the Integrated Modular Avionics (IMA) [11] scenario presented in the foundational S3K literature [6]. An avionics platform typically hosts multiple isolated domains: a hard real-time Flight Control (FC) domain, alongside less-trusted, highly variable Public Communication (PC) and Secure Communication (SC) domains.

Although the baseline S3K framework successfully demonstrated how CPU time slices can be dynamically reallocated to accommodate fluctuating communication workloads, modern IMA systems equally demand agile memory reallocation. For instance, during a normal flight phase, the PC domain may be allocated large shared memory buffers to process routine telemetry data. Yet, during a mission-critical phase (e.g., handling bursty encrypted traffic or mitigating an active cyber-attack), a trusted Communication Monitor (CM) must rapidly reclaim these memory buffers from the PC domain and reallocate them to the SC domain, without compressing the strict WCET window of the FC domain [11].

2.1.3 Temporal Memory Safety Bottleneck

This dynamic memory reallocation introduces a severe security challenge: maintaining strict temporal memory safety. To safely recycle freed memory and prevent catastrophic Use-After-Free (UAF) exploits or cross-domain information leaks, the system must exhaustively revoke all previously granted capability tokens associated with that specific memory region before it can be securely handed over to the SC domain.

Currently, this capability revocation relies heavily on software-driven mechanisms. Whether traversing a Capability Derivation Tree (CDT) in systems like seL4 [12] or iterating through a contiguous capability table (ctable) in the baseline S3K implementation [6], the process involves a software loop. The execution latency of this iteration scales linearly with the number of derived capabilities ($O(N)$).

In a hard real-time environment, this software dependency is a critical bottleneck. A single memory reallocation event could trigger an unpredictable

software sweep that consumes thousands of CPU cycles, pollutes the microarchitectural state (such as L1 caches and branch predictors), and directly threatens the deterministic WCET boundaries. Furthermore, to prevent starving hardware interrupts, software revocation loops must be artificially preemptible, which inherently breaks the atomicity of the security operation.

2.1.4 The Need for Hardware-Accelerated Memory Revocation

S3K overcomes the rigidity of traditional partitioning by treating both execution time and memory as dynamically delegatable, yet strictly isolated, capabilities. While S3K's time slice delegation allows a trusted Communication Monitor (CM) to efficiently reallocate CPU budgets (e.g., expanding the allowable WCET for the critical FC domain from 2 ms to a much safer 5 ms) [6], dynamic systems must also frequently reallocate physical memory resources.

For example, when switching flight phases, the CM might need to reclaim shared memory buffers from the PC domain to allocate them to the SC domain, or swiftly isolate the memory space of a compromised radio driver. This dynamic memory reallocation introduces a critical security bottleneck: ensuring strict temporal memory safety.

To safely recycle freed memory and prevent Use-After-Free (UAF) exploits, the kernel must exhaustively revoke all previously granted memory capabilities associated with that region. In the current S3K implementation, memory revocation relies on software iterations that traverse the capability table. The unpredictable execution latency of this software loop consumes the precious CPU cycles that dynamic scheduling aims to save, threatening to compress the absolute execution window of the FC domain.

This specific bottleneck creates a rigid necessity for custom hardware acceleration. While S3K safely handles time slice scheduling via software mechanisms, migrating the low-level memory capability revocation walk to dedicated hardware can reduce the cost of reclaiming delegated memory. The goal is not to make subtree revocation algorithmically constant time, but to execute the bounded ctable traversal with lower latency and less software-control overhead.

2.2 State of the Art in Memory Protection

Memory protection mechanisms define how software components obtain, use, and lose access to memory resources. This thesis focuses on capability-based protection, but capability systems are only one point in a broader design space that also includes page-table-based protection and tagged-memory mechanisms. These mechanisms differ in what object they protect, where authority is represented, how access rights are updated, and whether they naturally support delegated authority. This section positions capability-based protection within that broader space before the next section discusses representative capability-based systems.

2.2.1 Page-Table-Based Memory Protection

Page-table-based memory protection is the dominant paradigm in general-purpose operating systems. In this model, protection is intrinsically linked to address translation. The operating system kernel maintains a centralized hierarchical data structure (the page table) that maps virtual addresses to physical addresses and encodes access permissions (read, write, execute) for each page.

Because authority is centralized within these OS-managed tables, capability delegation (in the strict sense) does not exist natively at the hardware level; processes must invoke the kernel, for example through system calls or IPC-based resource-management interfaces, to alter sharing policies. When the kernel needs to revoke a process's access to a memory region, it modifies or removes the corresponding Page Table Entry (PTE) [13].

However, since modern processors cache these translations in a Translation Lookaside Buffer (TLB), modifying the PTE in memory is insufficient. The kernel must actively invalidate the cached entries across all processor cores. This process, known as a *TLB shootdown* [14], typically requires sending Inter-Processor Interrupts (IPIs) to halt other cores and force them to flush their TLBs. This reliance on global synchronization introduces unpredictable microarchitectural interference, severely complicating WCET analysis for hard real-time systems. Additionally, the coarse page-level granularity (typically 4KB) is ill-suited for the dynamic sharing of small, specialized data structures.

2.2.2 Tagged Memory

Tagged memory architectures, such as ARM's Memory Tagging Extension (MTE) [15] or SPARC's Application Data Integrity (ADI) [16], operate by enforcing strict tag equivalence between pointers and physical memory at the hardware level. The architecture associates a small metadata tag, which often referred to as an allocation tag, with each fixed-size granule of physical memory (e.g., every 16 bytes). Concurrently, pointers utilized by the software are extended to incorporate a corresponding reference tag. The hardware validates that the pointer's reference tag matches the memory's allocation tag upon every load and store instruction.

When memory is freed or reallocated, a tagged memory system can invalidate stale references by assigning a new allocation tag to the recycled physical memory region. Any existing dangling pointers scattered throughout the system will retain the obsolete reference tag, triggering a hardware fault upon dereference [15] [16]. This mechanism catches temporal vulnerabilities without requiring a global memory sweep or complex software tracking.

Despite this efficiency in enforcing temporal safety, purely tag-based access control lacks the semantic richness required for complex access policies. It protects the physical location itself rather than representing a transferable right, and therefore cannot express restricted delegation or track resource derivation. It is worth noting that the underlying hardware mechanism of tagged memory is highly versatile; architectural capability designs, such as CHERI [3], successfully repurpose memory tags (utilizing a hidden validity bit per memory word) to guarantee the unforgeability of capability tokens. However, as a standalone access-control paradigm, memory tags cannot encode ownership, making them insufficient for systems like S3K [6] that require dynamic resource partitioning.

2.2.3 Capability-Based Protection

Capability-based protection represents access authority explicitly as unforgeable tokens. A capability names a protected resource and securely encodes the specific rights that the holder may exercise over that resource. Unlike page-table-based protection, where authority is derived implicitly from OS-controlled mappings, a capability functions as a first-class object of authority. Furthermore, unlike tag-only memory mechanisms that merely protect physical locations, a capability can encode rich, resource-specific metadata such as ownership, bounds, permissions, and derivation state. Figure 2.1 illustrates this model. A process holding a capability can use it

to access a protected resource and can derive a more restricted capability for another process. This delegated authority is the property that makes capabilities useful for dynamic resource sharing, but it also creates the revocation problem addressed in this thesis: before a resource is reused, any derived authority to that resource must be withdrawn.

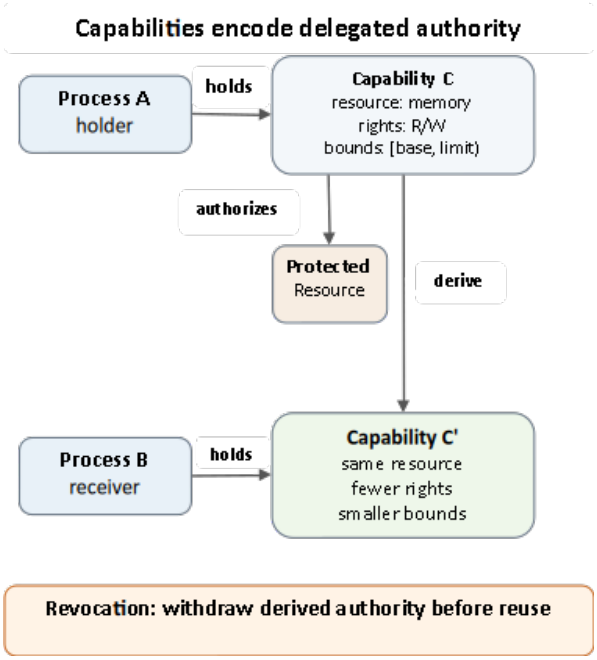

Figure 2.1: Capability-based protection as delegated authority. A capability encodes access rights to a protected resource, derivation creates restricted authority for another holder, and revocation withdraws derived authority before resource reuse.

This explicit representation of authority makes capability-based protection exceptionally well suited for systems that require fine-grained and dynamic resource management. A protection domain can hold a capability to access a resource and later dynamically delegate a restricted version of it to another domain through controlled kernel operations. This property is crucial for systems such as S3K [6], where capabilities are not restricted to memory protection but form a unified model representing time slices, IPC channels, monitor rights, and other privileged resources.

However, the power to freely delegate authority introduces a significant revocation challenge. When a shared resource is reclaimed, the system must guarantee that all derived or transferred authority referring to that resource is invalidated before the underlying hardware is safely reused. Because authority

can be widely distributed across processes, kernel objects, or hardware registers, invalidating this distributed authority typically requires exhaustive scanning, complex indirection tables, or software traversal of derivation metadata. These mechanisms inherently introduce severe trade-offs in lookup cost, memory overhead, and most critically for real-time systems, execution timing predictability.

2.2.4 Comparative Analysis

To clearly illustrate why capability-based systems are uniquely suited for dynamic Mixed-Criticality Systems (MCS) like S3K—and why they inherently suffer from revocation bottlenecks—we compare the three paradigms across key dimensions in Table 2.1.

Table 2.1: Comparison of Memory Protection Paradigms

Feature	Page Tables	Tagged Memory	Capabilities
Authority Locus	Tables	Physical tags	Tokens
Granularity	Page-level	Granule-level	Byte-level
Delegation	Poor	None	Excellent
Revocation Mechanism	TLB Shootdown	Update allocation tag	Traverse & invalidate
Predictability	Poor (IPI jitter)	Low (tag check overhead)	Poor (Software search)

As summarized in Table 2.1, **Page Tables** provide a mature mapping-based mechanism, but their coarse granularity and reliance on TLB shootdowns introduce unpredictable execution jitter, violating the strict WCET constraints required by hard real-time schedulers. **Tagged Memory** is highly efficient at catching dangling pointers and moves selected access checks into hardware, but it lacks the semantic richness to express *delegated authority* and ownership.

For this thesis, the relevant observation is that **Capability-Based Protection** provides the right abstraction for S3K's dynamic resource-management model, but it also introduces the exact bottleneck addressed by the proposed CLB. S3K keeps derivation state explicit in a ctable, making revocation structurally bounded. The CLB then accelerates access checks and memory-backed capability operations to deterministic hardware speeds while preserving software control over policy.

The following section will further discusses representative capability-based systems and how they differ in capability representation, tracking and

revocation.

2.3 Capability-Based Systems

2.3.1 Historical Capability Systems

Capability-based systems originate from the problem of protection and controlled sharing in multiprogrammed and time-sharing computers. In this setting, multiple programs or users had to access shared objects without giving each other unrestricted authority over the whole machine. Early work by Dennis and Van Horn described a capability as a protected reference that combines the designation of an object with the rights that the holder may exercise over that object [17]. The central idea is therefore not merely memory bounds checking, but explicit authority representation: a protection domain can hold, pass, and use authority through capabilities rather than relying only on global access-control lists or address-space mappings.

Levy's survey of capability-based computer systems provides a historical overview of this design space and shows that early capability systems explored both architectural and operating-system implementations [18]. The Cambridge CAP computer is a representative hardware-supported design: it provided protected capability state and used capability-controlled access to memory segments and protected procedures [19]. HYDRA represents a different point in the design space, where the operating-system kernel structured resources as protected objects and mediated access through capabilities [20].

Revocation was already a recognized issue in these historical systems, but it was usually handled through indirect mechanisms rather than through a fast revoke primitive. Levy describes systems that relied on object deletion, protected capability storage, reference tracking, or garbage collection to ensure that stale capabilities did not authorize access to reused objects [18]. For example, HYDRA used reference information and garbage collection, while IBM System/38 made an explicit trade-off between revocable profile-checked pointers and more efficient authorized pointers [21]. This illustrates an early form of the same tension addressed in this thesis: capability delegation is useful for flexible sharing, but revocation requires additional tracking or indirection.

These systems were developed in a period when the mainstream hardware and software ecosystem was also converging on virtual memory, page tables and privilege rings. Those mechanisms were simpler to integrate with the

Unix and C programming model, where protection is largely organized around processes and address spaces. Capability systems therefore remained highly influential in protection and operating-system research, but did not become the dominant general-purpose hardware-software protection interface.

This historical background is relevant to this thesis because it clarifies the broader role of capabilities as a resource-authority model. Modern systems instantiate this model in different ways: CHERI [3] provides architectural tagged capabilities for pointer-level protection, while systems such as seL4 [12], Barrelfish [22], and S3K [6] maintain capability state primarily in software. Across these implementations, a recurring difficulty is revocation. Once authority can be delegated or derived, the system must be able to identify and invalidate derived authority before the underlying resource is safely reused. Historical systems addressed this through mechanisms such as protected capability storage, indirection, object deletion, or kernel-managed capability state, each introducing trade-offs in lookup cost, implementation complexity, and revocation latency. S3K's ctable-based design can be seen as a modern software-managed response to this problem: it keeps derivation state explicit and contiguous, while the proposed CLB accelerates the common access-check and revocation path without moving the full protection policy into hardware.

2.3.2 CHERI-Style Tagged Capabilities

The Capability Hardware Enhanced RISC Instructions (CHERI) architecture extends traditional RISC ISAs by replacing standard integer pointers with architectural "fat pointers" [3]. These extended pointers encapsulate metadata, such as base addresses and access permissions, protected by a hidden hardware tag bit. While CHERI effectively eradicates spatial memory vulnerabilities at the hardware level, its baseline architecture intentionally omits the tracking of capability provenance (derivation history).

Consequently, achieving temporal memory safety in CHERI relies entirely on software-driven garbage collection mechanisms, such as CHERIvoke [4] or Cornucopia [5]. These solutions employ a "quarantine and sweep" strategy: when memory is freed, it is placed in a quarantine buffer. Once the quarantine reaches capacity, a global revocation epoch is triggered. The system must perform a comprehensive memory sweep, sequentially reading memory words across the entire virtual or physical address space to locate and invalidate the hardware tags of dangling pointers [4] . The execution time of this sweep scales linearly with the total size of system memory ($O(M)$), yielding a non-

deterministic pause that is highly destructive to the tight WCET boundaries required by hard real-time systems.

2.3.3 Software-Managed Capability Systems

To avoid the crippling overhead of global memory sweeping, systems historically rely on complex software protocols to track capability derivation. Formally verified microkernels like seL4 meticulously maintain a Capability Derivation Tree (CDT) in kernel software to precisely track parent-child provenance. During a revocation system call, seL4 relies on this CDT to recursively trace and destroy all descendant capabilities [12]. However, seL4 constructs its CDT using nodes connected via doubly linked lists that are dynamically allocated and scattered across physical memory. Traversing this fragmented data structure inherently involves unpredictable pointer chasing. This random memory access pattern triggers severe cache misses, making the execution time of revocation highly variable and dependent on the tree's depth.

Alternatively, the Barrelfish multikernel addresses heterogeneous multicore systems by abandoning shared memory assumptions. It maintains capability consistency across isolated cores via explicit message passing [22]. While this avoids shared-memory fragmentation, capability revocation in Barrelfish requires triggering complex two-phase commit protocols and cross-core message broadcasting to ensure global state consistency. This distributed, network-like coordination introduces massive latency jitter. Ultimately, both shared-memory tree traversals and distributed message protocols fail to satisfy the strict WCET constraints required by hard real-time environments.

2.3.4 Hardware-Native Capability Architectures

Recognizing the fundamental limitations of software-managed capability operations, recent architectural research has explored more hardware-native solutions. A prominent example is CAPSTONE [23], which addresses temporal safety by introducing hardware-native capability management directly into the processor's ISA.

CAPSTONE extends the capability model with "Linear Capabilities" and explicitly unforgeable "Revocation Capabilities." When a capability is delegated, the hardware generates a corresponding revocation token. To reclaim access, the owner invokes a specialized hardware instruction using this token, which directly invalidates the derived capabilities without requiring any software-level memory sweeping or tree traversal.

While this purely hardware-driven approach eliminates the software scanning overhead and provides trustless revocation, it introduces significant trade-offs. It requires deep, invasive modifications to the core CPU pipeline, the memory hierarchy, and the foundational ISA. Furthermore, resolving complex chains of capability delegation purely in hardware logic incurs substantial silicon area and complexity costs, making it difficult to adopt in resource-constrained embedded systems.

2.3.5 Contiguous Software Tracking and the S3K Approach

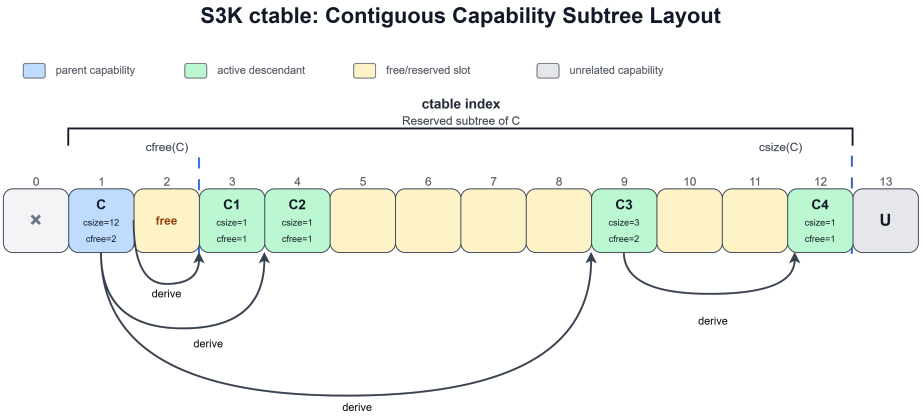

Figure 2.2: S3K capability-table organization. A parent capability reserves a bounded contiguous subtree in the ctable; `csize` gives the subtree bound and `cfree` identifies the remaining free derivation space.

The S3K framework [6] seeks a middle ground between page-table-based protection, tag-only memory-safety mechanisms, fragmented software capability protocols, and invasive hardware capability ISAs. It entirely abandons the dynamic creation of kernel objects, unpredictable linked lists, and complex message passing. Instead, S3K strictly confines all derived capabilities into a statically pre-allocated, physically contiguous one-dimensional array known as the capability table (ctable). Figure 2.2 illustrates this contiguous organization. Each capability occupies an index in the ctable and carries metadata that bounds its local derivation space. The `csize` field defines the size of the reserved subtree controlled by the capability, while `cfree` separates unused slots for future derivation from slots already occupied by derived descendants. As a result, a logical capability derivation

tree is represented by non-overlapping contiguous regions in the table rather than by dynamically allocated linked nodes.

This physical layout fundamentally alters the complexity of tracking sources, reducing the software overhead from an unpredictable graph traversal or network broadcast to a bounded and linear memory sweep ($O(C)$, where C is the maximum reserved subtree size). However, despite this structural advantage, the baseline S3K implementation still suffers from microarchitectural limitations due to its reliance on an incremental software loop to execute the revocation.

Executing a multi-instruction capability sweep in software pollutes the microarchitectural state (e.g., filling L1 caches and branch predictors with capability sweeping logic) and incurs massive context-switching overhead because the kernel must insert preemption points to avoid stalling hardware interrupts. Therefore, while S3K optimizes the data structure, the fundamental reliance on a software execution path remains a bottleneck. The proposed CLB builds on the same contiguous ctable model but maps the common access-check and memory-backed capability-management operations onto an in-core hardware fast path, while leaving ownership, refill policy, and higher-level resource management under S3K software control.

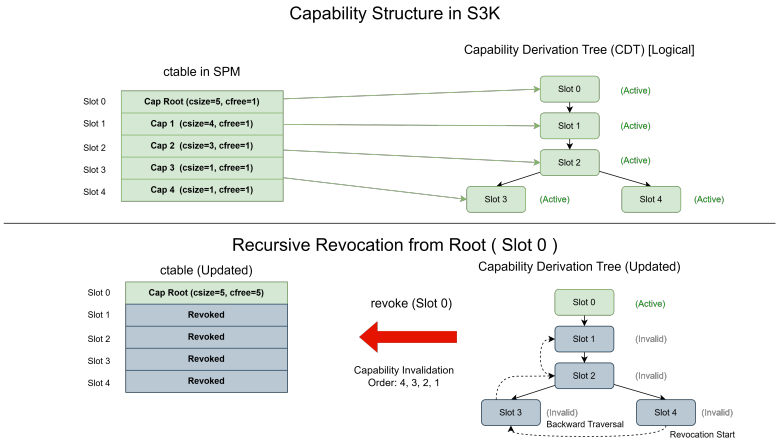

Figure 2.3: S3K Revocation Process

2.4 Hardware-Software Co-Design for Privileged Operations

The concept of offloading privileged software management functions to specific hardware, as proposed in this thesis, is a well-established architectural paradigm. It aligns with a broader industry evolution aimed at eliminating software bottlenecks and securing deterministic performance through hardware/software co-design.

2.4.1 The HW/SW Co-Design Sweet Spot for Capabilities

Within the specific domain of capability systems, the CHERI project perfectly exemplifies this successful transition for spatial safety. By extending the CPU registers and instruction set, CHERI integrate capability bounds-checking directly into the processor's pipeline [3], atomizing what previously required heavy software instrumentation.

However, as discussed in the capability-system survey above, pushing temporal safety entirely into hardware logic (e.g., CAPSTONE [23]) requires highly invasive ISA modifications and vast silicon area to resolve complex delegation chains.

A HW/SW co-design approach offers a pragmatic middle ground. By utilizing the S3K framework's contiguous capability table (ctable) as the software foundation, the complex policy formulation and system structure remain securely in software. Simultaneously, the specific microarchitectural bottleneck—the sequential software loop used for revocation—can be offloaded to a dedicated, non-invasive hardware accelerator. This reduces kernel-entry overhead, software loop control, and per-entry instruction overhead without incurring the massive architectural overhead of a pure hardware ISA overhaul.

2.4.2 Hardware-Driven Real-Time Schedulers

Outside the realm of security, the real-time systems domain also has a similar trend in transferring OS management to hardware to eliminate execution jitter. When software schedulers process massive concurrent tasks at microsecond granularity, traversing scheduling data structures (e.g. red-black trees or linked lists) introduces unpredictable latency.

Consequently, researchers and industry have developed Hardware RTOS architectures that harden core scheduling services—such as task ready queues, priority encoders, and context switch triggers—into parallel Finite State Machines (FSMs) directly adjacent to the CPU pipeline [24]. By eliminating the software execution paths and memory traversals, these hardware schedulers achieve nanosecond-level, near-zero jitter task switching.

The transition from software memory address checker to hardware TLBs, from software bounds-checking to CHERI, and from software task scheduling to hardware RTOS, represents a clear architectural track. The proposed hardware-accelerated capability revocation adheres strictly to this paradigm: utilizing hardware to execute bounded and also deterministic operations while leaving complex policy formulation to the software kernel.

2.5 Summary

In conclusion, this chapter has established the clear necessity for hardware-accelerated memory capability revocation in modern dynamic Mixed-Criticality Systems (MCS). While these systems demand dynamic memory agility to maximize resource utilization, they strictly forbid any operations that compromise Worst-Case Execution Time (WCET) determinism.

A review of memory protection mechanisms shows that page tables, tagged memory, and capability-based protection solve related but different access-control problems. Page tables provide a mature mapping-based mechanism, while tagged memory can move selected access checks into hardware. Capability-based systems are the closest match for this thesis because they explicitly represent delegated authority, ownership, and fine-grained resource rights. However, existing capability-based approaches still leave an important gap for hard real-time systems: purely software-driven approaches—whether relying on global memory sweep (CHERI [3] software ecosystems), fragmented derivation trees (seL4 [12]), or distributed message passing (Barrelfish [22])—introduce unacceptable execution jitter and cache pollution. In contrast, hardware-native capability architectures (CAPSTONE [23]) offer high performance but require invasive core modifications that are often prohibitive for resource-constrained embedded systems.

The S3K framework provides a superior structural foundation by flattening capability derivation trees into a statically allocated, contiguous array. However, its continued reliance on a preemptible software loop to execute revocation remains a critical microarchitectural vulnerability.

By analyzing successful historical trends in HW/SW co-design—such as

TLBs, hardware-driven schedulers, and CHERI's spatial safety mechanisms—this chapter demonstrates that offloading specific OS bottlenecks to dedicated hardware is a highly effective paradigm. Therefore, exploring a dedicated hardware extension designed specifically to accelerate S3K's contiguous revocation process is strongly motivated, promising a path toward lower-latency and more predictable memory-capability management.

Chapter 3

Methodology and Architectural Design

This chapter describes the research methodology used in this thesis to design, implement, and evaluate the proposed hardware acceleration. It justifies the selection of the simulation environment, outlines the standard development process for RISC-V hardware extensions, defines the theoretical hardware/software co-design boundaries, and establishes the strict microarchitectural requirements and test environments for the proposed Capability Lookaside Buffer (CLB).

3.1 Research Paradigm and Method Selection

To bridge the gap between theoretical real-time capability models and actual silicon performance, this project adopts a quantitative, simulation-based hardware/software co-design methodology.

3.1.1 Rationale for the Chosen Methodology

A critical decision in architectural research is selecting an appropriate simulation environment. The RISC-V ecosystem offers several mainstream simulators, each designed with different trade-offs between execution speed and architectural fidelity.

By utilizing this methodology, the project applies several core engineering skills:

- **Instruction Set Simulators (e.g. Spike):** Spike [25] serves as the golden standard reference model for RISC-V ISA compliance and software logic verification. It executes instructions functionally at extremely high speeds. However, it operates without any concept of clock cycles, pipelining, or memory latency, making it unsuitable for microarchitectural timing analysis.
- **System-Level Emulators (e.g. QEMU)** QEMU [26] utilizes dynamic binary translation to achieve massive execution speeds, making it the ideal tool for booting entire operating systems and developing software stacks efficiently. To achieve this high throughput, QEMU abstracts away hardware details, employing a simplified memory model that does not accurately reflect cache hierarchies, pipeline stalls, or bus contention.
- **Architectural simulators (e.g. gem5):** The gem5 simulator [10] is a heavyweight, event-driven simulator that can model processor and memory-system behavior at a higher abstraction level than RTL.
- **RTL simulators (e.g. Verilator):** Verilator compiles SystemVerilog RTL into an executable simulation model. It is slower than functional emulation, but it runs the actual hardware implementation and is therefore the most direct source for the CVA6 integration results in this thesis.

Spike and QEMU are useful for rapid software prototyping and functional checks, but they abstract away the pipeline and memory interactions that matter for the proposed Capability Lookaside Buffer (CLB). gem5 was therefore useful during architectural exploration, where the instruction semantics and high-level memory interactions could be modeled before RTL integration. The final performance evidence in this thesis, however, is based on the CVA6 SystemVerilog implementation running under Verilator, because that environment executes the actual CLB command state machines, scoreboard integration, and memory-request path.

3.1.2 Development Process for RISC-V Hardware Extensions

To ensure engineering feasibility and align with modern silicon design practices, the development of the hardware extension follows a standardized,

top-down Hardware/Software co-design process that is widely adapted in RISC-V extension development:

- **ISA-Level Specification Definition:** The process begins by utilizing the custom opcode space reserved in the RISC-V architecture to precisely encode the new custom instructions (e.g., urevoke). This involves defining instruction formats, register addressing modes, and behavioral semantics.
- **Toolchain and Software Adaptation:** To allow the S3K kernel to emit these new instructions, the compiler toolchain (such as GCC or LLVM) must be patched. In early stages, this could be achieved by using C-level inline assembly macros that inject the raw hexadecimal machine code directly into the software execution flow.
- **Agile RTL Modeling (Chisel):** Moving beyond traditional Verilog, the RTL microarchitecture is constructed using Chisel (Constructing Hardware in a Scala Embedded Language) [27]. Chisel's object-oriented nature allows for highly parameterized generation of the CLB state machines and SRAM lookup structures.

In summary, adopting this full-stack development process bridges the gap between theoretical architectural design and practical engineering. By systematically progressing from ISA definition to RTL integration, the project shows that the proposed CLB extension is not merely a conceptual software model, but a hardware mechanism that can be mapped into a concrete RISC-V core and evaluated using RTL simulation.

3.1.3 Rejected Methods: Machine-Mode ISA Extension

During the architectural exploration phase, an intermediate hardware acceleration scheme—referred to as the "Lightweight ISA Extension"—was initially considered as the baseline and ultimately rejected. This discarded approach proposed accelerating the software traversal loop by introducing a custom Machine-Mode instruction (e.g. crevoke). In this design, a hardware Finite State Machine (FSM) would take over the bus to rapidly traverse the physically contiguous ctable in Scratchpad Memory (SPM).

To preserve real-time constraints and avoid blocking system interrupts during a long memory traversal, the hardware instruction was designed to

monitor the interrupt line. If an interrupt occurred, the hardware would write its current progress back to a source register and retire early, allowing the kernel to handle the interrupt and seamlessly resume the instruction afterward.

Although this approach successfully provided deterministic instruction latency, maintained compatibility with existing Physical Memory Protection (PMP) logic, and solved the interrupt blocking issue with low hardware complexity, it was fundamentally flawed at the system level. Because the instruction was privileged to Machine Mode, invoking it still required the user process to issue a system call, and the overhead of context switch remained high. As a result, optimizing only the loop while retaining the trap overhead yielded diminishing performance returns bounded by memory bandwidth, leading to the rejection of this method.

3.1.4 Societal, Ethical, and Sustainability Aspects

From a societal perspective, this thesis targets systems where memory-safety failures may have consequences beyond ordinary software crashes. Safety-critical and security-critical platforms such as avionics, autonomous vehicles, and industrial control systems increasingly consolidate workloads with different criticality levels on shared hardware. In such systems, a memory revocation failure may lead to use-after-free vulnerabilities, cross-domain interference, or leakage of sensitive data. By reducing the cost of capability revocation while preserving the S3K protection model, the proposed CLB contributes to safer and more predictable resource management in critical embedded systems.

From an economic sustainability perspective, the work supports more efficient use of embedded hardware resources. Static partitioning is simple, but it can require over-provisioning because resources must be reserved for worst-case demands even when they are not continuously needed. S3K addresses this problem by enabling controlled dynamic reconfiguration, and the CLB reduces the runtime cost of memory-capability management in that setting. If such mechanisms are used in real systems, they may reduce the need for redundant hardware capacity and allow stronger isolation guarantees to be maintained on resource-constrained platforms.

The ecological impact of the work is more indirect. The proposed hardware extension increases logic utilization, and any additional hardware may increase silicon area and potentially energy consumption. This thesis reports Field-Programmable Gate Array (FPGA) resource overhead, but it does not provide a complete power, energy, or lifecycle assessment.

Therefore, no strong ecological claim is made. The possible ecological benefit is limited to an indirect one: if predictable dynamic resource sharing reduces over-provisioning or avoids deploying additional processing units, it may contribute to better hardware utilization. This remains a topic for future evaluation.

Ethically, the main positive impact is improved safety and security for systems that interact with people or critical infrastructure. The main risk is that hardware enforcement mechanisms are difficult to change after deployment; design errors in the CLB or in the hardware/software interface could create new failure modes. For this reason, the design intentionally leaves policy decisions in the S3K software layer and limits the hardware extension to accelerating bounded access-check and revocation operations.

3.2 Hardware Acceleration and Proposed Architecture Models

To address the limitations of software loops and trap overheads, this section defines the theoretical boundaries of the hardware and introduces the specific User-Level architectural models.

3.2.1 The Take-Grant Protection Model and Hardware Boundaries

To define the precise architectural scope of the hardware acceleration, this design is strictly grounded in the Take-Grant Protection Model. This mathematical framework analyzes security by defining access control through four foundational primitives, namely *Take*, *Grant*, *Create*, and *Remove* (Revocation). When mapping these theoretical primitives to the system architecture, a critical decision is determining whether the hardware extension must support the *Grant* and *Take* operations.

In the context of the S3K partitioning kernel, the *Create* and *Grant* primitives are realized through explicit capability derivations (the derive system call) and IPC-based capability transfers. According to the S3K design principles, these operations are already meticulously implemented as strict constant-time ($O(1)$) algorithms within the software kernel. They inherently require complex policy enforcement, such as verifying csize and cfree boundaries in the capability table (ctable), confirming ownership, and

maintaining the monotonic properties of domains to ensure strict spatial and temporal isolation [6].

In adherence to the operating system design principle of separating mechanism from policy, the hardware extension intentionally excludes the implementation of the *Grant*, *Take*, and *Create* primitives. Embedding these intricate $O(1)$ software policies into silicon would result in inflexible hardware without significant performance gains, directly violating S3K's philosophy of maintaining a minimal Trusted Computing Base (TCB). Therefore, these operations remain strictly managed by the formally verified S3K software kernel, acting as the policy layer.

Consequently, the proposed hardware extension is exclusively designed to serve the Remove primitive. As highlighted in the S3K specifications, revoke is the only system call whose execution time is linear (bounded by csize) rather than constant, relying on incremental software deletion and preemption points to maintain responsiveness. The sole purpose of the hardware acceleration is to make this bounded traversal cheaper and more predictable by executing the ctable metadata walk in a hardware command state machine while preserving the subtree-size dependence of the operation.

3.2.2 User-Level RISC-V Custom Instructions

To avoid the significant context switch costs associated with trapping into the kernel, the proposed architecture introduces a set of custom RISC-V instructions that can be executed directly in User Mode. By completely bypassing the ecall interface for routine invalidations, this design entirely eliminates the context switch and the high microarchitectural costs associated with the fence.t flush.

3.2.3 Hardware Acceleration of Tree Traversals

The concept of accelerating structured traversals in hardware is well-established in domains such as ray-tracing. In S3K, the relevant structure is not an arbitrary graph but a contiguous ctable layout. The proposed hardware therefore does not need to chase scattered pointers or execute a software loop for every metadata step. Instead, the CLB command state machine walks the relevant ctable metadata directly through the memory path, reducing instruction overhead and branch behavior while still performing the required per-descendant reads and writes.

3.2.4 Software-Managed Capability Lookaside Buffer (CLB)

To directly address the S3K revocation bottleneck and replace the existing PMP-based enforcement mechanism, the core engineering contribution of this thesis is the design of a Capability Lookaside Buffer (CLB). Inspired by software-managed TLBs, the CLB is modeled as a fast, on-chip cache situated between the ctable in SPM and the processor's Load/Store Unit (LSU). Its conceptual model operates as follows:

- **Software-Managed Miss Handling:** Instead of relying on a complex and potentially vulnerable hardware table walker, whenever a process attempts to access a capability that is not present in the CLB (a CLB Miss) or encounters an invalid state, the hardware triggers a synchronous exception. The processor traps into the S3K kernel, where a verified software handler evaluates the capability derivation tree, validates permissions, and explicitly loads the valid capability into the CLB. This approach ensures the kernel maintains complete visibility and absolute authority over capability usage.
- **Software-Managed CLB Entries:** Capabilities successfully loaded into the CLB by the kernel contain the metadata required for fast access checks, including ownership, ctable index, bounds, and permissions. The kernel remains responsible for choosing which line to refill and for validating the authoritative metadata in the backing ctable.
- **Memory-Backed Revoke Command FSM:** When a user process executes `uclb.revoke`, the hardware launches a commit-safe command state machine that accesses the backing ctable, clears child ownership, updates the parent metadata, invalidates matching cached child entries, and updates the cached parent metadata. This accelerates the revocation walk, but the latency still depends on the number of descendants reclaimed.

While this architecture avoids the system-call path for user-level CLB operations and reduces the cost of ctable metadata traversal, it inherently requires navigating a trade-off: higher hardware complexity. Implementing a new functional unit and maintaining consistency between the CLB and the backing SPM presents significant engineering challenges that will be addressed in the implementation phase.

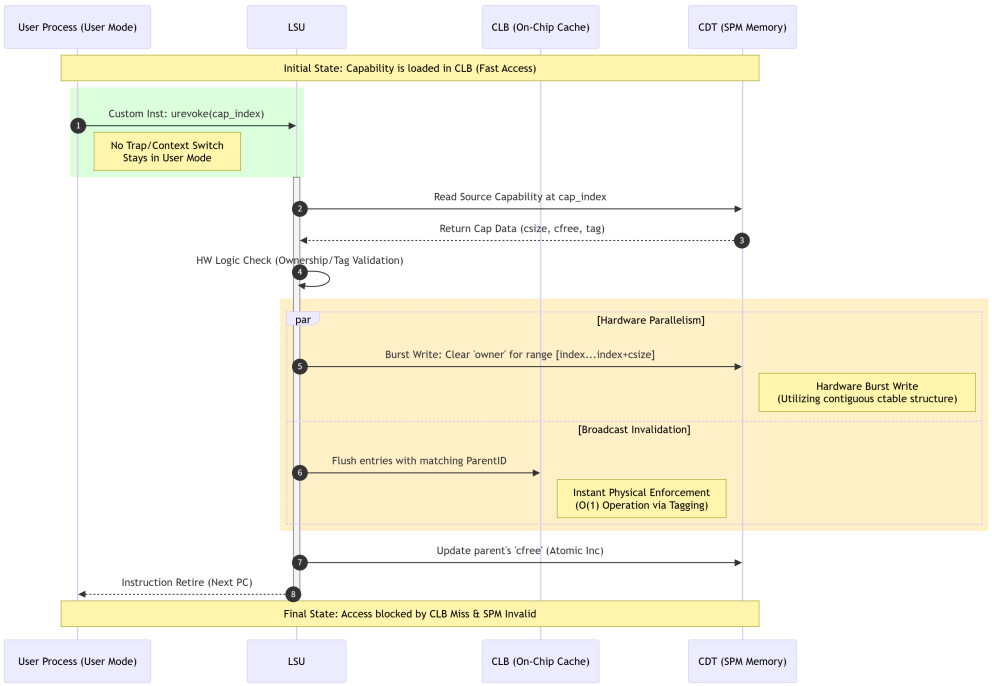

Figure 3.1: CLB Workflow

3.2.5 WCET Evaluation Requirements and Hardware Properties

Based on the theoretical constraints of the Remove primitive, the CLB hardware architecture must exhibit the following three critical microarchitectural properties to satisfy the WCET requirements of hard real-time systems:

- **Commit-Safe Side Effects:** Destructive commands such as `uclb.delete` and `uclb.revoke` must not modify the backing ctable speculatively. The CLB therefore starts these command state machines only after the corresponding instruction is known to commit.
- **Bounded Traversal Cost:** The execution time of a revoke operation remains related to the number of descendants that must be reclaimed. The CLB requirement is a bounded hardware traversal over the contiguous ctable that reduces the fixed trap cost and the per-descendant software overhead.
- **Pipeline Overhead and Jitter Reduction:** Since the ctable resides in the SPM, traditional cache pollution is already mitigated. However,

executing a software loop still requires instruction fetching, decoding, branching, and repeated metadata handling. The CLB reduces this overhead by moving the metadata walk into a hardware command state machine while keeping the authoritative policy in software.

3.3 Implementation Roadmap and Evaluation Framework

The implementation follows a model-to-RTL roadmap. The custom instruction semantics and the CLB state-machine behavior are first explored at an architectural level. The design is then integrated into CVA6 as a concrete RTL implementation, and the final performance measurements are collected from the Verilator simulation of that RTL system.

3.3.1 Architectural Modeling

The initial stage establishes the behavior of the custom CLB instructions and the expected interaction between the CLB and the S3K capability table. The gem5 model is used in this stage as an architectural aid: it helps validate the instruction semantics and the intended hardware/software split before the mechanism is mapped into a concrete processor implementation.

3.3.2 RTL Integration

The second stage implements the CLB as an in-core extension of CVA6. In this stage, the decoder, scoreboard dispatch, commit-safe CLB functional unit, load/store access-control path, and memory-backed command state machine are all part of the SystemVerilog design. This step checks that the mechanism can be realized inside a real open-source RISC-V core rather than only as an abstract simulator model.

3.3.3 RTL Measurement

The final stage evaluates the implemented design using Verilator. The same CVA6 RTL model is used for the baseline software path and the CLB-enabled path, so measured differences can be attributed to the CLB mechanism rather than to a different simulated core. The measurements focus on capability-operation latency, load/store access-control overhead, and revocation scaling.

3.4 Timing and Predictability Integration

3.4.1 Test environment

The evaluation uses a single-core bare-metal S3K configuration running on the Verilator model of the modified CVA6 RTL core. The baseline configuration keeps the original PMP plus system-call capability-management path, while the CLB configuration enables the custom CLB instructions and CLB-based access control. Cycle measurements are taken around the target operations using the RISC-V architectural cycle counter.

3.4.2 Hardware/Software to be used

The project uses both architectural simulation and RTL simulation tools. Their roles are separated as follows:

- **Architectural Modeling:** gem5 is used as a higher-level model for exploring instruction semantics and design structure.
- **RTL Simulation and Implementation Analysis:** The CVA6 SystemVerilog implementation is simulated with **Verilator**. This flow is the primary source for the performance results and provides the RTL basis for later implementation-cost analysis.

3.5 Assessing reliability and validity of the data collected

3.5.1 Validity of method

The primary objective of this project is to evaluate the predictability and performance of a microarchitectural extension (the CLB). Using the CVA6 RTL implementation as the main measurement platform gives high methodological validity because the measured behavior includes the actual CLB state machines, scoreboard integration, load/store permission checks, and memory-command path. This avoids relying on an abstract timing model for the main performance claims.

3.5.2 Reliability of method

The reliability of the methodology is supported by the deterministic nature of RTL simulation. Unlike measurements on a physical board, the Verilator environment avoids external noise such as dynamic clock scaling, thermal variation, and board-level interference. Repeating the same workload on the same RTL configuration should therefore reproduce the same architectural behavior and cycle-counter measurements.

3.5.3 Data validity

To ensure data validity, the comparative evaluation will strictly isolate the hardware extension as the sole independent variable. The experimental group (S3K utilizing the hardware CLB and custom urevoke instruction) and the control group (unmodified S3K utilizing the software-based revocation loop) will be subjected to the exact same workloads, executing the exact same capability tree deletion tasks within the same contiguous memory layout. This rigorous control of variables guarantees that any observed reduction in latency or execution jitter is exclusively attributable to the proposed hardware extension.

3.5.4 Reliability of data

The data collected during the evaluation consists of cycle counts measured with the RISC-V architectural cycle counter. The measurement points are placed around the operations being compared, which excludes boot and initialization overhead and keeps the reported values focused on the CLB behavior under study.

3.6 Planned Data Analysis

The data analysis will primarily employ quantitative comparison techniques. The execution latency (in clock cycles) of the revocation process will be measured across varying Capability Derivation Tree (CDT) sizes (e.g., revoking 1, 4, 16, and 64 child capabilities). The analysis will focus on two key metrics:

- **Latency Reduction (Performance):** Calculating the speedup ratio between the software baseline and the hardware-accelerated CLB approach.

- **Revocation Scaling and Predictability:** Analyzing how revocation latency changes as the number of descendants increases. The expected outcome is that the software path scales with both subtree size and system-call overhead, while the CLB path keeps the same subtree-size dependence but lowers the average latency and per-descendant cost by moving the metadata walk into hardware.

3.6.1 Data Analysis Technique

3.6.2 Software Tools

The output logs generated by the Verilator experiments are parsed and aggregated using Python. The resulting data are used to generate the latency, access-overhead, and revocation-scaling figures in the results chapter.

3.7 Evaluation framework

The evaluation framework compares two configurations of the same S3K system:

- **Baseline Configuration:** The original PMP plus system-call path for memory-capability management.
- **Experimental Configuration:** The CLB-enabled path using custom CLB instructions and CLB-based access control.

The measured operations cover individual capability management, ordinary protected memory accesses, and revocation over capability trees with different shapes and descendant counts.

3.8 System documentation

To ensure reproducibility and facilitate future research within the S3K framework, the project will be thoroughly documented across all system layers:

- **Hardware Extensions:** The CVA6 RTL changes and the architectural model of the CLB will be documented using inline comments and block diagrams illustrating the CLB state machine.

- **Kernel Software:** Patches applied to the S3K kernel (C code and inline assembly) to invoke the new custom instructions will be documented using Doxygen-style inline comments.
- **Deployment:** A comprehensive README file will be provided in the project's Git repository, detailing the toolchain setup, compilation steps, and commands required to reproduce the Verilator simulation flow and data analysis graphs.

Chapter 4

Microarchitectural Design and Implementation

This chapter details the engineering and implementation of the proposed CLB architecture. Adhering to the hardware-software co-design methodology, the implementation is divided into the microarchitectural data structures, the specification of the custom RISC-V ISA extensions, the execution flows of the hardware state machines, the architectural gem5 model, and the CVA6 RTL realization.

4.1 CLB Microarchitecture and Hardware Interfaces

The core of the hardware extension is the CLB, implemented as a highly associative SRAM structure (e.g., 8-way to 32-way) strictly dedicated to memory bounds and permissions enforcement. To minimize complex hardware logic area and adhere to the S3K kernel's minimal TCB philosophy, the CLB replacement policy is entirely software-managed.

The CLB hardware features decoupled interfaces to maintain strict security boundaries:

- **LSU Interface (Read-Only):** Queried automatically during processor memory accesses to intercept and block illegal transactions based on valid CLB entries.
- **Kernel Management Interface (Write-Only):** Populated exclusively by the machine-mode kernel via the custom `mclb.fill` instruction.

- **SPM Memory Interface:** Used by the CLB command state machine to read and update the backing ctable during memory-backed operations such as `uclb.get`, `uclb.delete`, and `uclb.revoke`.

Each entry within the CLB array stores the metadata required for fast authorization and boundary checking in the current RTL implementation. The mapping between the hardware fields and the S3K software `mem_t` structure is defined in Table 4.1. Memory bounds are represented explicitly as an inclusive base and an exclusive limit, matching the setup and capability-fetch registers exposed by the implemented interface.

Table 4.1: CLB entry bit-field organization in the current RTL implementation

Field	**Width (bits)**	**`mem_t` Mapping**	**Description**
Valid	1	N/A	Entry validity flag.
PID	16	`owner`	S3K process identifier for context-aware enforcement.
Cap_Index	11	Array Index	The capability index in the `ctable` (Max 2048 entries).
Cap_Free	9	`cfree`	Number of free slots available (identifies active children range).
Cap_Size	9	`csize`	Number of slots reserved for descendants (Range: [1, 256]).
Perms	3	`rwx`	Read/Write/Execute permissions.
Base	64	`base`	Inclusive lower bound of the memory range.
Limit	64	`base + size`	Exclusive upper bound of the memory range.

4.2 Decoupled Configuration and Verification Latch

To prevent cross-privilege state clobbering and mitigate information leakage, the machine-mode fill interface and the user-mode fetch interface are physically separated. Furthermore, a "Commit-after-Verify" security model is enforced using an **Internal Verification Latch**. This microarchitectural, software-invisible structure temporarily stages data fetched from the SPM. The

hardware performs strict ownership verification on this latch before altering any architectural state.

Machine-Mode Setup CSRs (Write-Only): The kernel uses these registers to prepare capability deployments. They include `mctable_base` (physical base address of the `ctable` in SPM), `mclb_setup_base`, `mclb_setup_limit`, and a packed `mclb_setup_meta` register holding the capability index, free slots, size, and permission bits.

User-Mode Capability Fetch Registers (CFR, Read-Only): User processes securely access capability boundaries through `uclb_cfr_base`, `uclb_cfr_limit`, and `uclb_cfr_meta`. These registers are updated by the hardware only upon successful authorization from the Internal Verification Latch.

4.3 Custom RISC-V ISA Extensions

To interface with the CLB, seven custom instructions were encoded into the RISC-V ISA utilizing the custom opcode space (`0001011`). These instructions are divided into privileged management operations and user-mode execution primitives. The encoding details are provided in Table 4.2.

Table 4.2: Custom Instruction Encoding (Opcode: `0001011`)

Instruction	funct7	rs2	rs1	funct3	rd
`mclb.fill`	`0000000`	`00000`	`rs1`	`000`	`00000`
`mclb.flush`	`0000000`	`00000`	`00000`	`001`	`00000`
`mclb.read`	`0000000`	`00000`	`rs1`	`010`	`rd`
`mclb.invalid`	`0000000`	`00000`	`rs1`	`011`	`00000`
`uclb.revoke`	`0000000`	`00000`	`rs1`	`100`	`rd`
`uclb.get`	`0000000`	`00000`	`rs1`	`101`	`rd`
`uclb.delete`	`0000000`	`00000`	`rs1`	`110`	`rd`

4.3.1 Return Value Contract

All user-mode operations (`uclb.*`) adhere to a non-trapping, inline error-reporting model via the destination register `rd`. Instead of triggering costly hardware exceptions on access violations, the hardware returns status codes to facilitate user-level control flow:

- `0`: SUCCESS / COMPLETED.

- `>0`: INCOMPLETE. Represents the remaining capability count, utilized exclusively by `uclb.revoke` during preemptions.
- `-1`: ERROR_OWNERSHIP_MISMATCH. Indicates a security violation.

4.4 Execution Flows and Hardware State Machines

The microarchitectural implementation relies on dedicated FSMs to orchestrate memory transactions securely without exposing intermediate states.

4.4.1 Kernel Trap Handler Flow (`mclb.fill`)

When a user process attempts to access a memory location which holds a valid capability that is not currently cached in the CLB, the LSU hardware triggers an exception. The S3K kernel handles this trap through the following workflow:

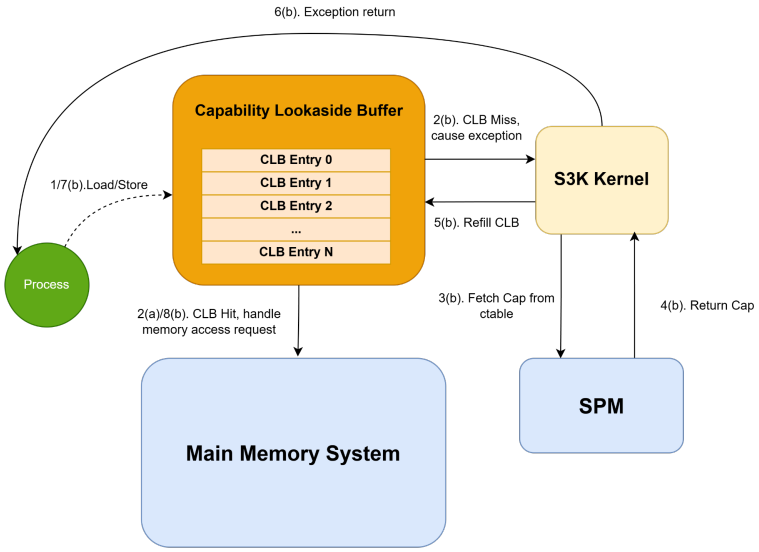

Figure 4.1: Hardware and software execution flow of the Kernel Trap Handler (`mclb.fill`).

1. **Search:** The kernel searches the `ctable` in the SPM for the faulting capability.

2. **Calculate Bounds:** The kernel calculates the physical boundaries (e.g., `Limit = base + size`).

3. **Stage Metadata:** The kernel packs the metadata and writes it into the Machine-Mode Setup Control and Status Registers (CSRs).

4. **Hardware Commit:** The kernel executes `mclb.fill rs1`, where `rs1` specifies the hardware CLB line index determined by the OS replacement policy. The hardware atomically copies the CSRs and the `Current_PID` into the target slot and asserts the Valid bit. The kernel then issues `mret`.

Additional machine-mode primitives include `mclb.flush`, which rapidly invalidates all entries within the CLB array in a single clock cycle (typically executed during context switches to prevent cross-process leakage), and `mclb.invalid`, which targets a specific entry.

4.4.2 User-Mode Fetch Flow (`uclb.get`)

This instruction safely exposes capability boundaries to user software without invoking system calls.

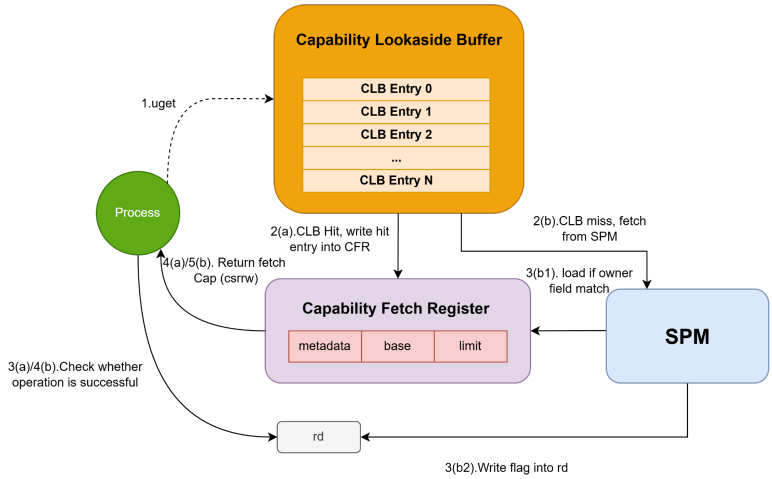

Figure 4.2: Execution flow for securely fetching capabilities in user mode (`uclb.get`).

1. **Secure Load:** The FSM issues an SRAM load from the SPM `ctable` into the Internal Verification Latch.

2. **Verify:** If `Latch.Owner != Current_PID`, the hardware aborts the fetch, preserves the existing CFRs to prevent leakage, and writes `-1` to `rd`.

3. **Commit:** If the ownership matches, the hardware copies the latch data into the `uclb_cfr_*` CSRs and writes `0` to `rd`.

4.4.3 User-Mode Deletion Flow (`uclb.delete`)

This flow atomically verifies and destroys a single capability entry across the memory hierarchy.

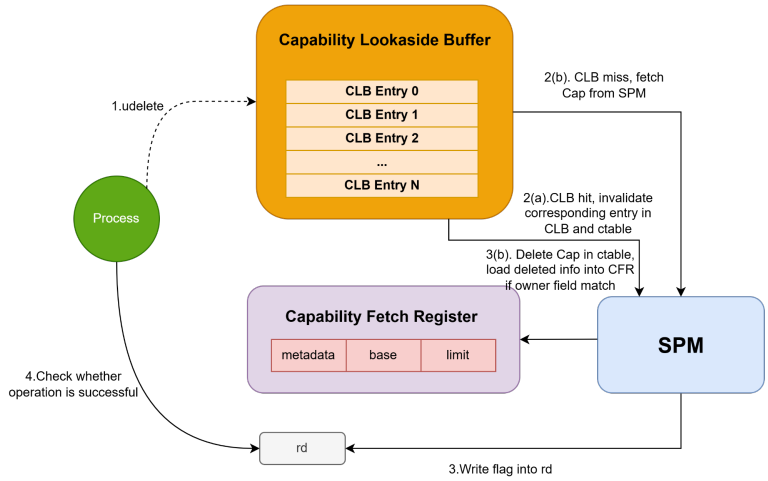

Figure 4.3: Atomic verification and destruction process across the memory hierarchy (`uclb.delete`).

1. **Secure Load & Verify:** Similar to the fetch flow, the entry is loaded into the Verification Latch. If ownership mismatches, the hardware aborts the destructive write and returns `-1`.

2. **Execute:** Upon successful verification, the FSM simultaneously invalidates the corresponding index in the CLB cache and zeroes out the entry in the backing SPM, concluding by writing `0` to `rd`.

4.4.4 User-Mode Revocation Flow (`uclb.revoke`)

The `uclb.revoke` instruction addresses the primary execution bottleneck of the S3K kernel. Because a single revocation might target an entire subtree,

executing this atomically in hardware could block urgent system interrupts. To resolve this, the FSM employs a "Chunked Execution" model.

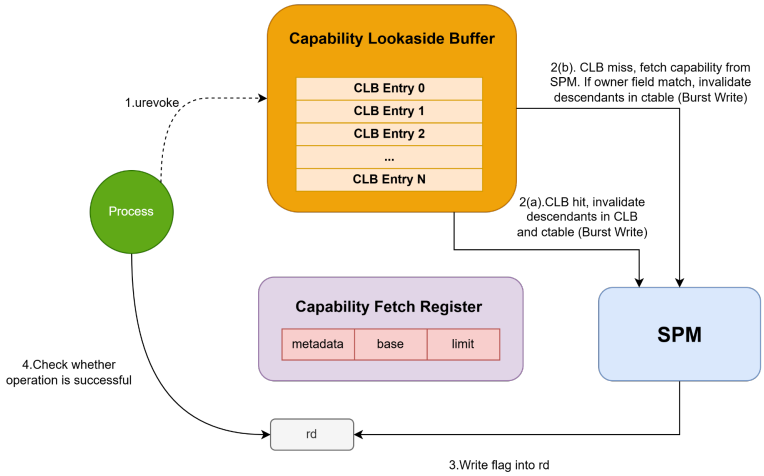

Figure 4.4: Chunked execution model and state machine flow for capability revocation (`uclb.revoke`).

1. **Secure Load & Verify:** The FSM loads the target capability's metadata into the Verification Latch. An ownership mismatch immediately aborts the operation (returning `-1`).

2. **Chunked Execution:** The FSM initiates a Burst Write to zero out the descendant range in the SPM. If a hardware interrupt line asserts during this burst, the FSM safely halts at a chunk boundary, commits the current progress back to the SPM, and writes the remaining capability count (`>0`) to `rd`.

3. **Software Resumption:** The user-level loop observes the `>0` return value, branches back to the start of the instruction, and allows the core to service the pending interrupt. Upon return, the instruction seamlessly resumes the burst write.

4. **Completion:** Once the entire descendant range is successfully zeroed out, the hardware writes `0` to `rd`, and the software loop exits.

4.5 Simulator Integration (gem5)

To evaluate the proposed CLB as a microarchitectural mechanism rather than as an isolated ISA extension, the design was integrated directly into gem5's RISC-V full-system platform [10]. The implementation spans the decoder, the memory-management path, and the memory system. First, gem5's RISC-V decoder was extended to recognize the custom `0001011` opcode space and dispatch the newly introduced `mclb.*` and `uclb.*` instructions. Second, the CLB itself was modeled as a dedicated gem5 `SimObject` attached to the RISC-V Memory Management Unit (MMU). This model stores the CLB entries, the machine-mode setup state used by `mclb.fill`, the software-visible CFR state, and the internal control state required by `uclb.get`, `uclb.delete`, and `uclb.revoke`. Third, the normal instruction-fetch and data-access paths were modified so that each translated memory access is checked against the CLB before the request is allowed to proceed. In the experimental configuration, the CLB therefore replaces the original PMP-based memory capability enforcement in the simulated S3K system.

An important modeling decision is that the CLB is not represented merely as a local permission checker. In the gem5 model, it is also an independent memory-system master with its own memory-side request port. Consequently, whenever `uclb.get`, `uclb.delete`, or `uclb.revoke` must consult or update the backing ctable in SPM, the corresponding reads and writes are issued by the CLB itself rather than being disguised as ordinary LSU traffic from the processor core. This decoupling makes CLB-generated memory traffic explicit in the simulation and allows the measured latency of capability operations to include realistic response time, retry behavior, and interconnect contention.

The same implementation also preserves the software-managed nature of the design. The CLB performs the fast-path permission check for ordinary instruction fetches, loads, and stores, but a miss still raises a synchronous exception that traps into the S3K kernel. The kernel then inspects the authoritative ctable in SPM, decides the replacement target according to the software policy, programs the setup CSRs, and completes the refill with `mclb.fill`. In this way, the gem5 model reflects the intended hardware/software split of the proposal: hardware accelerates the repeated enforcement and revocation operations, while the kernel remains responsible for policy decisions and refill authority.

The timing model was implemented on both `TimingSimpleCPU` and `MinorCPU` using the same latency decomposition. For ordinary protected

Table 4.3: Instruction-level latency decomposition used by the gem5 CLB model.

Instruction	**Condition**	**Latency model**
`mclb.fill`	Local refill commit	$T_{\text{op}} = T_{\text{pipe}} + 1$
`mclb.flush`	Flush all entries	$T_{\text{op}} = T_{\text{pipe}} + 1$
`mclb.invalid`	Invalidate one line	$T_{\text{op}} = T_{\text{pipe}} + 1$
`mclb.read`	Read one CLB line into CFR	$T_{\text{op}} = T_{\text{pipe}} + 1$
`uclb.get`	CLB hit	$T_{\text{op}} = T_{\text{pipe}} + L_{\text{lookup}}$
`uclb.get`	CLB miss	$T_{\text{op}} = T_{\text{pipe}} + L_{\text{lookup}} + T^{\text{rd}}_{\text{ctable}}$
`uclb.delete`	CLB hit	$T_{\text{op}} = T_{\text{pipe}} + L_{\text{lookup}} + T^{\text{wr}}_{\text{ctable}}$
`uclb.delete`	CLB miss	$T_{\text{op}} = T_{\text{pipe}} + L_{\text{lookup}} + T^{\text{rd}}_{\text{ctable}} + T^{\text{wr}}_{\text{ctable}}$
`uclb.revoke`	Revoke of N child capabilities	$T_{\text{op}} = T_{\text{pipe}} + L_{\text{lookup}} + T^{\text{rd}}_{\text{parent}} + \sum_{i=1}^{N} \left(T^{\text{rd}}_{\text{child},i} + T^{\text{wr}}_{\text{child},i} \right) + T^{\text{wr}}_{\text{parent}}$

memory accesses, the CLB contributes an explicit lookup delay before the underlying fetch or data request is issued. For low-complexity management instructions, such as `mclb.fill`, `mclb.flush`, `mclb.invalid`, and `mclb.read`, a one-cycle local execution model is sufficient because these operations only manipulate internal CLB state or already-latched configuration values. By contrast, `uclb.get`, `uclb.delete`, and `uclb.revoke` are modeled as compound operations whose total latency is decomposed into local CLB work and, when required, one or more CLB-generated ctable accesses. This decomposition keeps the architectural model internally consistent across both CPU models.

Although the two CPU models differ internally, the microarchitectural interpretation is the same. In `TimingSimpleCPU`, the detailed model captures the delay of the local CLB lookup as well as the full round-trip latency of any backing ctable request generated by the CLB. In `MinorCPU`, the same events are placed closer to the in-order pipeline semantics: ordinary protected memory accesses pay the lookup cost before entering the data-access path, simple `mclb.*` maintenance operations occupy a single execute slot, and `uclb.get`, `uclb.delete`, and `uclb.revoke` are represented as multi-cycle operations whose local lookup, request generation, memory response, and architectural completion are accounted for separately. As a

Table 4.4: Configurable parameters of the gem5 CLB architectural model.

Parameter	**Type**	**Role in the model**
`enable`	Boolean	Global runtime enable for CLB-based access control
`clb_entries`	Integer	Number of CLB lines.
`clb_lookup_latency`	Cycles	Generic CLB lookup latency

result, both CPU models expose the same dominant timing factors, while `MinorCPU` additionally captures their interaction with pipeline back-pressure and instruction issue.

Table 4.3 summarizes the latency decomposition used in the gem5 implementation. The notation is intentionally parameterized so that the same timing structure can be instantiated on both `TimingSimpleCPU` and `MinorCPU` while still allowing the memory-system contribution to depend on the surrounding gem5 configuration.

In Table 4.3, T_{pipe} denotes the core-local execution overhead of issuing and retiring the custom instruction, while L_{lookup} denotes the local CLB lookup latency. The terms $T^{\text{rd}}_{\text{ctable}}$ and $T^{\text{wr}}_{\text{ctable}}$ denote the observed response time of CLB-generated reads and writes to the backing ctable. They are therefore not fixed properties of the CLB alone, but depend on the simulated memory hierarchy and interconnect. This distinction is crucial: the gem5 model does not merely assign constant delays to capability operations, but explicitly separates local CLB processing from memory-system latency. In particular, `uclb.revoke` follows the same modeling principle as the other user-level CLB operations: one local lookup phase followed by a sequence of explicit memory requests generated by the CLB state machine.

The configurable parameters used to instantiate this timing model are summarized in Table 4.4. Parameterization is intentionally kept compact: the CLB capacity and the generic lookup latency are configured explicitly, while the remaining contribution to instruction latency emerges from the surrounding gem5 memory-system configuration. This is especially important for `uclb.revoke`, whose latency is determined by the number of generated ctable transactions and their response time.

From a modeling perspective, the most important consequence of this integration is that the same architectural model can represent both the original PMP plus system-call-based capability-management path and the CLB-enabled path. This made gem5 useful for early design exploration before

the mechanism was implemented in CVA6 RTL.

Finally, the gem5 model should be understood as an architectural model rather than the source of the final performance claims. In particular, the SPM backing the ctable is represented through gem5's memory system abstraction rather than through an RTL memory path, and the detailed timing of simple maintenance operations is intentionally collapsed into the default one-cycle pipeline cost. The final cycle results in this thesis are therefore taken from the CVA6 RTL implementation under Verilator.

4.6 Register-Transfer Level (RTL) Implementation

4.6.1 Purpose of the RTL Implementation

The gem5 implementation described in Section 4.5 provides an architectural model of the proposed mechanism. The RTL implementation is the concrete realization used for the main performance analysis: it maps the same CLB mechanism into an open-source RISC-V core. In this project, the CLB is implemented in SystemVerilog inside CVA6 and verified using Verilator.

The RTL implementation targets the single-core bare-metal S3K configuration used throughout this thesis. It provides the concrete hardware basis for integrating the CLB into the processor pipeline, preserving precise architectural side effects, and supporting subsequent analysis of implementation overheads.

4.6.2 CVA6 Execution Model and Integration Point

CVA6 is an in-order, single-issue RISC-V core. Instructions enter the issue path in program order, but the core uses a scoreboard to track in-flight instructions and their destination registers. This scoreboard-based organization allows different functional units to complete independently and write back out of order, as long as the architectural dependencies recorded by the scoreboard are respected. This execution model is suitable for the CLB because a CLB instruction can be registered as a separate functional unit (FU), in the same broad style as the existing execution resources, without turning the CLB into an external accelerator.

In the RTL implementation, the CLB is integrated natively inside CVA6 as its own functional unit. The instruction decoder recognizes the S3K custom-

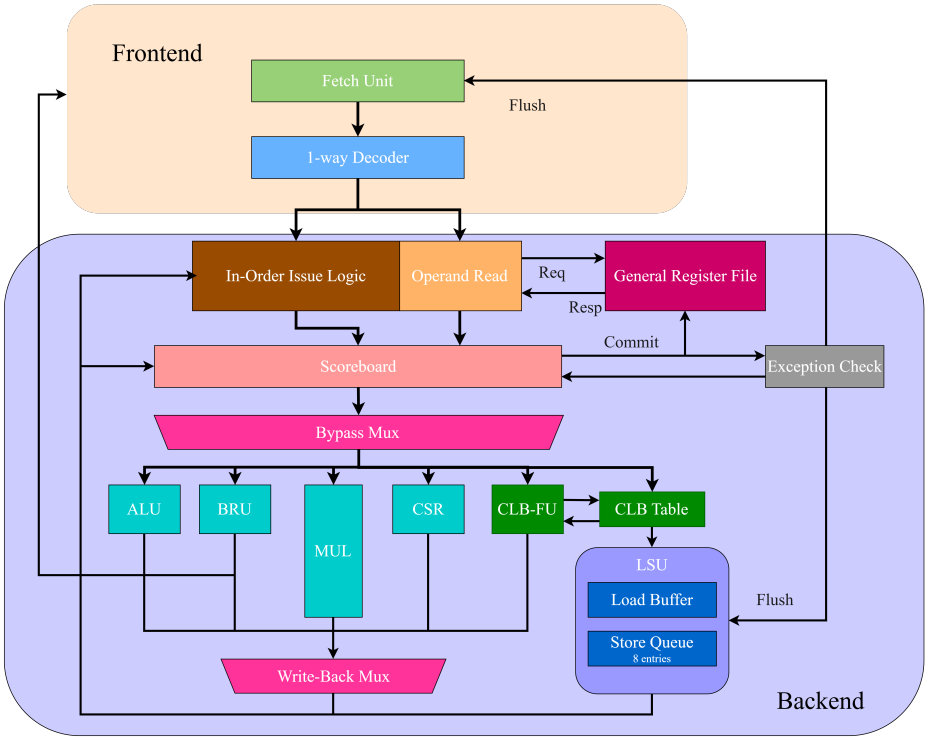

Figure 4.5: Simplified CVA6 execution structure relevant to the CLB integration. CVA6 issues instructions in order, while its scoreboard allows independent functional units to complete and write back out of order.

0 instruction encodings and emits CLB operations into the scoreboard. The issue stage then dispatches these operations to the dedicated CLB FU.

The CLB FU is deliberately split from the state machine that performs the actual capability-table operations. The front-end FU buffers the issued instruction, records its transaction identifier, and waits until the same scoreboard entry reaches commit. Only after this commit acknowledgement does it launch a command into the CLB command unit. This is essential for destructive operations such as `uclb.delete` and `uclb.revoke`: their writes to the backing ctable must not occur speculatively. As a result, CLB side effects follow a store-like architectural discipline while still returning their final result through the normal CVA6 writeback path.

4.6.3 Access-Control and Memory-Command Paths

The LSU queries the CLB table for user-mode memory accesses. A load or store may proceed only if the CLB contains a valid entry for the current process whose range covers the complete access and whose permissions authorize the requested operation. If no such entry exists, or if all matching entries lack the required permission, the LSU raises a CLB access fault. The S3K trap handler then inspects the authoritative ctable, chooses a replacement line according to the software policy, and refills the CLB. When multiple cached capabilities overlap, the access-control logic uses union-of-capabilities semantics: any matching entry with sufficient permission can authorize the memory access.

Memory-backed CLB commands need to read and update the S3K ctable. In the RTL implementation, the CLB command unit reuses an existing CVA6 `dcache` request path inside the core, which in CVA6 is initially preserved for external accelerator. The current S3K target is single-core, runs without virtual memory, and does not require RVV, CV-X-IF, or an external accelerator in the intended embedded partitioning-kernel use case. Under these assumptions, reusing the otherwise available `dcache` port is a pragmatic implementation choice that keeps the CLB in-core and avoids introducing a separate bus-visible peripheral.

4.6.4 Implementation Constraints and Verilator Simulation

The RTL implementation is intentionally scoped to the S3K configuration evaluated in this thesis rather than to every possible CVA6 configuration. It assumes a single-core system and disables the MMU because the current S3K

kernel does not use virtual addresses. The RTL configuration also disables RVV, CV-X-IF, and external accelerators because these features are outside the target S3K bare-metal use case and would require additional writeback or memory-port arbitration. Multi-core CLB coherence and virtual-memory interaction are left outside the scope of this RTL implementation.

The same RTL system can boot in a Verilator simulation environment that models the surrounding S3K bare-metal SoC. The simulator loads the S3K kernel ELF and a user application ELF into the simulated memory map, then exposes Universal Asynchronous Receiver/Transmitter (UART) output as the observable program result. This allows the CVA6 RTL implementation, the CLB state machines, and the S3K software stack to run together in a full-system simulation flow without requiring board-level deployment.

Overall, the RTL implementation shows that the proposed CLB maps cleanly onto CVA6 as an in-core extension with commit-safe side effects and direct access to the backing capability table. This provides the RTL basis for later implementation-overhead analysis.

Chapter 5

Results and Analysis

This chapter evaluates the proposed CLB mechanism using Verilator simulation of the CVA6 RTL implementation described in Section 4.6. This environment executes the actual SystemVerilog CLB state machines, the CVA6 scoreboard integration, the reused dcache request path, and the S3K software stack. The resulting measurements are therefore used as the source for operation-level latency, revocation scaling, and access-control overhead.

5.1 Experimental Setup

The RTL experiments run a single-core bare-metal S3K system on the Verilator model of the modified CVA6 core. The baseline configuration uses the original software capability-management path, where user applications invoke `s3k_mem.*` system calls and memory access permissions are enforced through the existing kernel-managed protection mechanism. The CLB configuration enables the custom `uclb.*` instructions, CLB-based memory access control, and the RTL command state machine that accesses the ctable in SPM.

When a CLB miss occurs, refill and replacement follow the software-managed LRU policy implemented in the S3K kernel.

Cycle measurements are taken with the RISC-V `cycle` counter around the target operation, so boot and initialization overhead are excluded from the reported results.

Table 5.1: Measurement categories used in the main RTL evaluation.

Measurement category	What is measured	Reported data
Correctness gate	Functional correctness of CLB commands and protected memory accesses	Pass/fail
Capability operation latency	Average latency of capability fetch and deletion operations	Avg. cycles & speedup
Access-control overhead	Per-access cost of CLB-based authorization for ordinary memory operations	Cycles/access (CLB hit path)
Revocation scaling	Revocation latency and bounded scaling as the number of descendants increases	Avg. cycles & speedup

5.2 RTL Capability Operation Latency

The first group of measurements isolates the average latency of individual capability-management operations. The baseline uses the original software path, while the CLB configuration executes the corresponding hardware commands directly and still preserves commit-safe side effects for destructive operations.

Table 5.2: Measured operation-level latency in the Verilator RTL model.

Operation	Baseline cycles	CLB cycles	Speedup
Get	631	19	33.2×
Delete	617	39	15.8×

Both operations benefit substantially from the CLB path. The largest gain appears in capability fetch, where the average latency drops from 631 cycles for the software path to 19 cycles for the direct CLB instruction. Capability deletion remains slightly more expensive because it still performs a commit-safe destructive update to the backing ctable, but it still reduces average latency from 617 cycles to 39 cycles. These results indicate that most of the baseline cost comes from trap entry, kernel-side capability handling, and return-path overhead rather than from the small amount of metadata movement itself.

5.3 RTL Access-Control Overhead

The second group of measurements evaluates the cost of using the CLB as the memory access-control mechanism. In the CLB configuration, user-mode loads and stores are checked against the CLB table before the LSU allows the access to proceed. A cached matching capability represents the common hit case. A missing capability triggers a CLB access fault, after which the S3K trap handler can refill the CLB using the software-managed replacement policy.

Table 5.3: Measured access-control overhead in the Verilator RTL model.

Access pattern	**Baseline cycles/access**	**CLB cycles/access**
Load, CLB hit	14	14
Store, CLB hit	8	8
Mixed load/store, CLB hit	18	18

This measurement is important because the CLB is not only a command accelerator. It also replaces the fast-path authorization mechanism for ordinary user memory accesses. In the cached hit path, the CLB check is effectively free at this measurement granularity: load, store, and mixed load/store accesses match the baseline at 14, 8, and 18 cycles per access respectively. Miss handling remains a software-managed slow path: a missing entry raises a synchronous exception, after which the S3K kernel searches the authoritative ctable and refills the CLB according to its replacement policy. The frequency and cost of this path depend on the user program's capability working set, access locality, the kernel refill policy, and the trap path, so refill behavior is treated qualitatively rather than as a general capacity result.

5.4 RTL Revocation Scaling

Revocation is the main operation targeted by the proposed hardware extension. The baseline path revokes memory-capability subtrees through `s3k_mem_revoke`, which enters the kernel and updates the software ctable. The CLB path performs the same revocation through `uclb.revoke`, which executes the traversal in hardware while preserving the architectural remaining-count return convention for interrupt-safe preemption.

The experiment uses both flat and chain-shaped capability trees. The flat profile stresses fan-out from a single parent, while the chain profile stresses backward revocation through nested descendants. Reporting both shapes avoids overfitting the evaluation to one favorable ctable layout.

The baseline in this experiment is the original S3K memory-capability path using PMP-based memory protection and the `s3k_mem_revoke` system call. The CLB configuration enables CLB access control and performs revocation through user-level `uclb.revoke` instructions. Both configurations were executed on the same CLB-capable CVA6 Verilator model, so the comparison isolates the software path versus the CLB command path rather than comparing different cores or memory systems.

Tables 5.4 and 5.5 summarize the measured average revocation latency.

Table 5.4: RTL revocation scaling for flat capability trees.

Descendants	PMP+syscall avg	CLB avg	Speedup (×)	Syscall calls	CLB calls
1	748	31	24.13	1	1
2	821	39	21.05	1	1
3	888	50	17.76	1	1
4	953	61	15.62	1	1
5	1 022	72	14.19	1	1
6	1 091	83	13.14	1	1
7	1 160	94	12.34	1	1
8	1 229	105	11.70	1	1

Table 5.5: RTL revocation scaling for chain-shaped capability trees.

Descendants	PMP+syscall avg	CLB avg	Speedup (×)	Syscall calls	CLB calls
1	742	28	26.50	1	1
2	821	39	21.05	1	1
3	888	50	17.76	1	1
4	953	61	15.62	1	1
5	1 022	72	14.19	1	1
6	1 091	83	13.14	1	1
7	1 160	94	12.34	1	1
8	1 229	105	11.70	1	1

The results show that CLB-assisted revocation is substantially faster across

the measured descendant counts. For a single child, the CLB path reduces average latency from 748 cycles to 31 cycles in the flat profile and from 742 cycles to 28 cycles in the chain profile. At eight descendants, the uninterrupted hardware path still completes the subtree with one architectural revoke call and reduces average latency from 1229 cycles to 105 cycles, corresponding to an 11.70× speedup in the flat profile and the same 11.70× speedup in the chain profile.

The call-count columns are important for interpreting the result. In the reported measurements, neither path is interrupted, so both the baseline `s3k_mem_revoke` system call and the RTL `uclb.revoke` path complete with one architectural call per subtree. The speedup therefore comes directly from moving the revocation walk itself into the CLB command state machine rather than from reducing a repeated software retry loop. The CLB ISA still retains the visible remaining-count return contract for timer-interrupt preemption, but the measurements reported here exercise the uninterrupted fast path.

5.5 FPGA Resource Overhead

The previous sections evaluate the performance benefit of the CLB. This section evaluates the corresponding implementation cost. The comparison uses two single-core S3K-capable CVA6 SoC implementations: a baseline design without the CLB, and a CLB-enabled design in which the additional logic is integrated inside the CVA6 core. The purpose of this comparison is to quantify the resource and timing overhead of adding the CLB to an otherwise equivalent processor and SoC configuration.

Both designs were synthesized and implemented using the same Cheshire-based CVA6 SoC target, matching the SoC integration style used by the original S3K work [6]. The FPGA target is the Digilent Genesys 2 board, which uses a Xilinx Kintex-7 XC7K325T device; the SoC clock is constrained to 50 MHz. The SoC wrapper, UART, timer, memory system, and board-level debug infrastructure are therefore common to both designs. As a result, the resource difference primarily reflects the additional in-core CLB structures, the CLB command state machine, and the associated integration logic in the CVA6 pipeline.

At the full-SoC level, the CLB-enabled implementation increases LUT usage by 5,742 LUTs, or 6.12%, and flip-flop usage by 2,482 FFs, or 3.00%. It does not increase the number of block RAMs or DSP blocks. This matches the structure of the design: the CLB is mainly composed of control logic, registers,

Table 5.6: FPGA routed utilization of the baseline and CLB-enabled SoC implementations.

Resource	Baseline	CLB	Abs. increase	Rel. increase
LUTs	93,761	99,503	+5,742	+6.12%
FFs	82,846	85,328	+2,482	+3.00%
RAMB36	169	169	0	0.00%
RAMB18	22	22	0	0.00%
DSP	16	16	0	0.00%

comparators, permission checks, and finite-state machines rather than large memories or arithmetic datapaths.

The core-local overhead is larger than the full-SoC percentage because the common SoC infrastructure dilutes the relative increase in the top-level utilization numbers. Within the CVA6 core hierarchy, LUT usage increases from 25,914 to 31,578, corresponding to a 21.9% increase, while flip-flop usage increases from 14,327 to 16,821, corresponding to a 17.4% increase. This increase is expected because the CLB table performs parallel matching and range/permission checks, while the command state machine implements memory-backed `get`, `delete`, and `revoke` operations. Some of the overhead is also absorbed into existing CVA6 modules, including instruction decoding, CSR state, LSU access checks, dcache-port arbitration, and scoreboard dispatch.

The post-route timing result remains positive for both designs at the 50 MHz SoC clock. The baseline implementation reports a WNS of 0.170 ns, while the CLB-enabled implementation reports a WNS of 0.112 ns. The reduced slack indicates that the CLB integration adds some timing pressure, but both implementations still meet the target timing constraints on the selected Kintex-7 FPGA. Therefore, the FPGA implementation results support the practical feasibility of the proposed CLB as an in-core CVA6 extension, while showing that its cost is primarily a moderate LUT/FF increase without additional memory or DSP resources.

5.6 Discussion

5.6.1 Latency Improvement Analysis

The results show that the CLB improves capability-management latency by removing the dominant software path overhead. In the baseline configuration, even simple capability operations require a transition into the kernel, kernel-side metadata handling, and a return to user mode. The CLB path replaces this sequence with dedicated instructions and a hardware command state machine that operates directly on the ctable. This explains why `get` benefits the most: it is primarily a metadata fetch, so bypassing the system-call path reduces the average cost from 631 cycles to 19 cycles. The `delete` and `revoke` operations remain more expensive because they perform commit-safe destructive updates to the backing ctable, but they still avoid most of the kernel control-flow overhead and remain substantially faster than the software path.

5.6.2 Revocation Scaling

The revocation results also show an important scaling pattern. The CLB path achieves a 24.13× speedup for a single descendant in the flat profile and a 26.50× speedup in the chain profile, while the speedup at eight descendants is 11.70×. This decrease in speedup ratio reflects the changing balance between fixed and per-descendant costs. For small subtrees, the baseline cost is dominated by the fixed system-call and trap overhead, so replacing that path with a CLB instruction produces a very large ratio. As the number of descendants increases, both the baseline path and the CLB path must process more child metadata. The fixed trap overhead is then amortized, and the measured speedup approaches the ratio between the two per-descendant update paths. The CLB remains faster because it avoids software loop control, kernel entry and exit, and repeated instruction-level metadata handling, but it still performs real ctable memory reads and writes.

The similarity between the flat and chain-shaped revocation measurements follows from the S3K ctable layout. Although the logical capability trees differ, the hardware-visible work is dominated by reclaiming metadata from the contiguous table representation. In the current implementation, both profiles therefore reduce to a sequence of child metadata updates and parent metadata adjustments. This is consistent with the design goal of exploiting S3K's contiguous capability representation rather than building a general

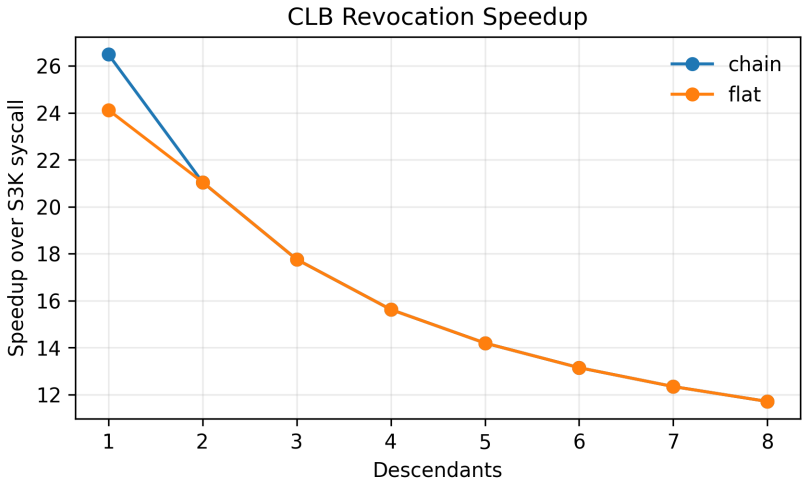

Figure 5.1: Speedup of CLB-assisted revocation over the software path.

graph-walking capability engine.

5.6.3 Predictability and WCET Implications

Although this thesis does not perform a full formal WCET analysis, the results provide evidence that the CLB design better matches the predictability goals of S3K than the original software revocation path. As shown in the revocation-scaling results, the operation still depends on the number of descendants that must be reclaimed. This dependence is expected and compatible with S3K: the number of descendants is bounded by the reserved ctable subtree size, represented by the capability metadata, so the operation remains bounded by explicit S3K metadata that is decided when the kernel is compiled.

The CLB also makes the remaining cost more structurally predictable. In the evaluated implementation, revocation operates over the contiguous ctable stored in SPM, and the hardware command state machine performs a regular sequence of metadata reads and writes. This removes much of the system-call entry and exit, software loop control, branch behavior, and repeated instruction-level metadata handling from the critical path. The measured flat and chain results support this interpretation: once the number of descendants is fixed, the logical tree shape has little effect on the observed latency because the hardware-visible work is dominated by the number of contiguous metadata entries reclaimed.

This structure also gives a simple upper-bound form that could be used as

an input to a later WCET analysis. For a revoke operation that reclaims at most D descendants, where $D \leq \texttt{csize}$, the uninterrupted hardware traversal can be modeled as

$$T^{\text{CLB}}_{\text{revoke}}(D) \leq T_{\text{setup}} + D \cdot T_{\text{entry}} + T_{\text{finish}}. \tag{5.1}$$

Here, T_{setup} captures instruction issue, commit, and initial metadata verification; T_{entry} is an upper bound for one child-entry reclaim step, including the required ctable metadata reads, writes, and local CLB invalidation; and T_{finish} captures the final metadata update and architectural writeback. The equation is not presented as a measured WCET result, since this thesis reports average RTL cycle measurements rather than a formal timing proof. Its purpose is to show how the CLB changes the shape of the timing argument: the cost is bounded by explicit S3K metadata and a regular per-entry hardware step, rather than by global memory sweeping or fragmented pointer traversal.

Finally, the CLB preserves S3K's need for interrupt responsiveness. A large revocation is not required to execute as one non-preemptible block. The `uclb.revoke` instruction preserves the remaining-count convention: if a timer interrupt must be serviced, the hardware can retire at a safe point and report how many descendants remain, allowing software to resume the operation later. Therefore, the proposed design does not eliminate the descendant-count term, but it turns revocation into a bounded and more stable hardware-managed traversal.

5.6.4 Access-Control Fast Path

For ordinary memory accesses, the CLB hit path does not add observable overhead at the measurement granularity used in this evaluation. Loads, stores, and mixed load/store sequences match the baseline cycle counts when the relevant capability is already cached. This is a necessary property for the design: if the CLB accelerated rare management operations but slowed down every protected load and store, it would not be a viable memory access-control mechanism. Miss handling remains intentionally software-managed. A CLB miss raises a synchronous exception, after which the S3K kernel validates the authoritative capability metadata and refills the CLB according to its replacement policy. The frequency of this path depends on the user program's capability working set, access locality, and kernel refill policy. Therefore, CLB capacity should not be interpreted like an ordinary data-cache size alone; it is also shaped by how software derives and uses capabilities.

5.6.5 Implementation Cost

The FPGA results indicate that the implementation cost is acceptable for the evaluated S3K-aligned target. At the full-SoC level, the CLB-enabled design adds 6.12% LUTs and 3.00% flip-flops, with no additional BRAM or DSP blocks. The design also meets timing on the Cheshire-based Genesys 2 Kintex-7 target at 50 MHz. The core-local overhead is larger than the full-SoC percentage, which is expected because the CLB is an in-core structure: the table lookup logic, range and permission checks, command state machine, CSR state, LSU integration, scoreboard dispatch, and dcache-port arbitration all sit inside or adjacent to the CVA6 core. This resource profile is a reasonable trade-off for a mechanism that removes the dominant software path from capability fetch, deletion, and revocation. However, these FPGA numbers should be interpreted as an implementation-cost result for this target, not as a final ASIC PPA claim or a maximum-frequency characterization.

5.6.6 Limitations and Future Work

The main limitation of the current design is that it focuses on the single-core S3K configuration. The original S3K work targets multicore partitioning systems and discusses synchronization points, kernel locking, and inter-core effects [6]. A core-local CLB introduces a new consistency question in that setting: after one core deletes or revokes a capability, other cores may still hold cached CLB entries derived from the same ctable metadata. A multicore version of this design would therefore need a bounded synchronization protocol for cross-core CLB invalidation.

One possible direction is to protect ctable metadata updates and cross-core invalidation with a small hardware spinning lock or hardware arbiter. Such a mechanism could turn revoke/delete into a bounded critical section that updates the authoritative metadata and invalidates affected remote CLB entries before the capability can be used again. Another direction is to align CLB synchronization with S3K synchronization points, so that remote invalidation happens at controlled scheduling boundaries rather than through arbitrary asynchronous shootdowns. A third possibility is to tag CLB entries with epochs or per-process generation counters, allowing revocation to advance a generation value and letting each core reject stale entries locally. All of these approaches require careful interrupt- and preemption-safe design; otherwise, the benefit of accelerating revocation could be replaced by lock contention or inter-core timing jitter.

Chapter 6

Conclusion

This thesis designed and evaluated a software-managed Capability Lookaside Buffer for accelerating S3K memory capability management. The design keeps policy in the S3K kernel while moving repeated, low-level enforcement and revocation mechanisms into hardware. It introduces custom CLB instructions for capability fetch, deletion, and revocation, together with a CLB table that supports fast-path access checks for user-mode memory operations.

The work progressed from architectural modeling to RTL implementation. The proposed mechanism was explored at the architectural level and then integrated into CVA6 as an in-core extension. The final RTL implementation includes a native CLB functional unit, LSU access-control checks, CSR support, and a memory-backed command state machine that accesses the S3K ctable. This implementation demonstrates that the proposed mechanism can be mapped onto a real open-source RISC-V core without modeling it as an external MMIO device or a separate SoC peripheral.

The evaluation shows substantial latency improvements. Capability fetch decreases from 631 cycles to 19 cycles, while capability deletion decreases from 617 cycles to 39 cycles. Revocation achieves speedups from 11.70× to 26.50× across the measured subtree sizes and shapes. At the same time, the CLB hit path for ordinary loads and stores introduces no observable cycle overhead in the reported measurements. The FPGA implementation adds 6.12% LUTs and 3.00% flip-flops at the full-SoC level, adds no BRAM or DSP blocks, and meets timing on the 50 MHz Cheshire-based Genesys 2 target.

The validation in this thesis is prototype-level validation, not a complete correctness proof or a complete WCET proof. The implementation is evaluated through functional programs, Verilator RTL simulation, revocation-scaling experiments, access-control measurements, and FPGA implementa-

tion. These experiments provide evidence that the proposed CLB mechanism works as intended in the evaluated single-core S3K scenarios and that its revocation behavior remains bounded by explicit S3K metadata. They do not, however, exhaustively verify all possible capability-table states, command-state-machine transitions, interrupt interleavings, or exceptional cases.

Full assurance would require a formal specification of the CLB/S3K semantics, proof or model checking of all relevant state-machine behavior, verification of interrupt and malformed-operation corner cases, and scheduler-level WCET or schedulability analysis. Multicore coherence would additionally require reasoning about cross-core invalidation and synchronization. The presented design should therefore be interpreted as a feasible and evaluated RTL prototype, rather than a complete security-certified implementation.

Overall, the results support the central claim of this thesis: S3K's deterministic, ctable-based capability-management model can be accelerated effectively by a tightly integrated RISC-V hardware extension. The CLB does not replace the S3K kernel's authority over policy, ownership, or refill decisions. Instead, it moves the timing-sensitive enforcement and revocation mechanisms closer to the processor pipeline, reducing the cost of operations that would otherwise require repeated software traversal and kernel entry. Future work should extend this design to multicore synchronization, virtual-memory interaction, higher-frequency and ASIC-oriented implementation studies, and larger end-to-end mixed-criticality workloads.

References

[1] A. Burns and R. I. Davis, "Mixed Criticality Systems: A Review," Department of Computer Science, University of York, Tech. Rep. v1.1, 2013. [Online]. Available: https://www-users.york.ac.uk/~ab38/review.pdf

[2] ——, "A Survey of Research into Mixed Criticality Systems," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 1–37, 2017.

[3] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, ser. SP '15. USA: IEEE Computer Society, 2015. doi: 10.1109/SP.2015.9. ISBN 9781467369497 p. 20–37. [Online]. Available: https://doi.org/10.1109/SP.2015.9

[4] H. Xia, J. Woodruff, S. Ainsworth, N. W. Filardo, M. Roe, A. Richardson, P. Rugg, P. G. Neumann, S. W. Moore, R. N. M. Watson, and T. M. Jones, "CHERIvoke: Characterising Pointer Revocation Using CHERI Capabilities for Temporal Memory Safety," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-52. New York, NY, USA: Association for Computing Machinery, 2019. doi: 10.1145/3352460.3358288. ISBN 9781450369381 p. 545–557. [Online]. Available: https://doi.org/10.1145/3352460.3358288

[5] N. Wesley Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. Tomasz Napierala, A. Richardson, J. Baldwin, D. Chisnall, J. Clarke, K. Gudka, A. Joannou, A. Theodore Markettos, A. Mazzinghi, R. M. Norton, M. Roe, P. Sewell, S. Son, T. M. Jones,

S. W. Moore, P. G. Neumann, and R. N. M. Watson, “Cornucopia: Temporal Safety for CHERI Heaps,” in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020. doi: 10.1109/SP40000.2020.00098 pp. 608–625.

[6] H. Karlsson and R. Guanciale, “Partitioning Kernel With Capability Controlled Temporal and Spatial Partitioning,” in *2025 IEEE Real-Time Systems Symposium (RTSS)*, 2025. doi: 10.1109/RTSS66672.2025.00015 pp. 68–81.

[7] J. Abella, C. Hernandez, E. Quiñones, F. J. Cazorla, P. R. Conmy, M. Azkarate-askasua, J. Perez, E. Mezzetti, and T. Vardanega, “WCET Analysis Methods: Pitfalls and Challenges on Their Trustworthiness,” in *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, 2015. doi: 10.1109/SIES.2015.7185039 pp. 1–10.

[8] OpenHW Group, “CVA6: An Application Class RISC-V CPU Core,” https://docs.openhwgroup.org/projects/cva6-user-manual/, 2026, accessed: 2026-06-02.

[9] Verilator Project, “Verilator,” open-source SystemVerilog simulator. Accessed: 2026-06-02. [Online]. Available: https://github.com/verilator/verilator

[10] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B. R. Bruce, D. R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S. A. R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T. M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L. E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M. D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D. A. Wood, H. Yoon, and Éder F. Zulian, “The gem5 Simulator: Version 20.0+,” 2020. [Online]. Available: https://arxiv.org/abs/2007.03152

[11] P. J. Prisaznuk, "ARINC 653 Role in Integrated Modular Avionics (IMA)," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008. doi: 10.1109/DASC.2008.4702770 pp. 1.E.5–1–1.E.5–10.

[12] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal Verification of an OS Kernel," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009. doi: 10.1145/1629575.1629596. ISBN 9781605587523 p. 207–220. [Online]. Available: https://doi.org/10.1145/1629575.1629596

[13] RISC-V International, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, RISC-V International, 2024, accessed: 2026-06-03. [Online]. Available: https://docs.riscv.org/reference/isa/priv/

[14] N. Amit, A. Tai, and M. Wei, "Don't Shoot Down TLB Shootdowns!" in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20. New York, NY, USA: Association for Computing Machinery, 2020. doi: 10.1145/3342195.3387518. ISBN 9781450368827. [Online]. Available: https://doi.org/10.1145/3342195.3387518

[15] Arm Ltd., "Introduction to the Memory Tagging Extension," https://developer.arm.com/documentation/108035/0100/Introduction-to-the-Memory-Tagging-Extension, accessed: 2026-06-03.

[16] Oracle, "Using Application Data Integrity (ADI)," https://docs.oracle.com/en/operating-systems/solaris/oracle-solaris/11.4/prog-interfaces/using-application-data-integrity-adi.html, accessed: 2026-06-03.

[17] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Commun. ACM*, vol. 9, no. 3, p. 143–155, Mar. 1966. doi: 10.1145/365230.365252. [Online]. Available: https://doi.org/10.1145/365230.365252

[18] H. M. Levy, "Capability- and Object-Based System Concepts," in *Capability-Based Computer Systems*. Digital Press, 1984, pp. 1–20. ISBN 978-0-932376-22-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780932376220500065

[19] R. M. Needham and R. D. Walker, "The Cambridge CAP Computer and Its Protection System," *SIGOPS Oper. Syst. Rev.*, vol. 11, no. 5, p. 1–10, Nov. 1977. doi: 10.1145/1067625.806541. [Online]. Available: https://doi.org/10.1145/1067625.806541

[20] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System," *Commun. ACM*, vol. 17, no. 6, p. 337–345, Jun. 1974. doi: 10.1145/355616.364017. [Online]. Available: https://doi.org/10.1145/355616.364017

[21] M. E. Houdek, F. G. Soltis, and R. L. Hoffman, "IBM System/38 Support for Capability-Based Addressing," in *Proceedings of the 8th Annual Symposium on Computer Architecture*, ser. ISCA '81. IEEE Computer Society Press, 1981, pp. 341–348.

[22] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The Multikernel: A New OS Architecture for Scalable Multicore Systems," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: Association for Computing Machinery, 2009. doi: 10.1145/1629575.1629579. ISBN 9781605587523 p. 29–44. [Online]. Available: https://doi.org/10.1145/1629575.1629579

[23] J. Z. Yu, C. Watt, A. Badole, T. E. Carlson, and P. Saxena, "CAPSTONE: A Capability-Based Foundation for Trustless Secure Memory Access," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23. USA: USENIX Association, 2023. ISBN 978-1-939133-37-3

[24] P. Kohout, B. Ganesh, and B. Jacob, "Hardware Support for Real-Time Operating Systems," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, ser. CODES+ISSS '03. New York, NY, USA: Association for Computing Machinery, 2003. doi: 10.1145/944645.944656. ISBN 1581137427 p. 45–51. [Online]. Available: https://doi.org/10.1145/944645.944656

[25] RISC-V International, "Spike: RISC-V ISA Simulator," 2024, accessed: 2024-05-20. [Online]. Available: https://github.com/riscv-software-src/riscv-isa-sim

[26] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator,” in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46. [Online]. Available: https://www.usenix.org/conference/2005-usenix-annual-technical-conference/qemu-fast-and-portable-dynamic-translator

[27] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, “Chisel: Constructing Hardware in a Scala Embedded Language,” in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC ’12. New York, NY, USA: Association for Computing Machinery, 2012. doi: 10.1145/2228360.2228584. ISBN 9781450311991 p. 1216–1225. [Online]. Available: https://doi.org/10.1145/2228360.2228584

TRITA-EECS-EX-2026:428
Stockholm, Sweden 2026

www.kth.se