SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Online OS reconfiguration for Cloud DBMS

Hristina Yordanova Grigorova

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Online OS reconfiguration for Cloud DBMS

Online OS Rekonfiguration für Cloud DBMS

Author:	Hristina Yordanova Grigorova
Examiner:	Prof. Dr. Pramod Bhatotia
Supervisor:	Ilya Meignan--Masson
Submission Date:	28.08.2025

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 28.08.2025 Hristina Yordanova Grigorova

Acknowledgments

I want to sincerely thank Prof. Dr. Pramod Bhatotia for supervising this thesis and giving me the opportunity to explore this topic.

I am also deeply grateful to my advisor Ilya Meignan--Masson, for his understanding and patience throughout this entire process. I highly appreciate the guidance and time he dedicated to providing detailed responses to all my questions. This thesis wouldn't have been possible without his support and expertise, and it has been a genuine pleasure working with him.

Abstract

Cloud-native database systems must evolve beyond static, single-tenant designs in order to meet the demands of the dynamic, multi-tenant, and complex cloud environment, where flexibility, elasticity, and safety are essential. Traditional DBMSs, designed for static workloads and resources, combined with bulky general-purpose operating systems, fail to fully leverage cloud capabilities. Previous approaches have attempted to integrate the DBMS with the Operating System (OS) or individually extend only one layer, but these solutions are limited and do not satisfy all requirements of cloud-native systems.

A cloud-native DBMS should be lightweight, high-performance, scalable, elastic, and extensible, capable of dynamic adaptation to variable workloads and heterogeneous hardware infrastructures. Our proposed system is guided by these principles, integrating the fundamental design goals of cloud-native applications. By leveraging unikernels, our approach provides a minimal and specialized execution environment, achieving a tighter OS/DBMS integration while reducing overhead. Dynamic reconfiguration support allows the system to adjust to the rapidly changing cloud conditions, meeting both tenants' and providers' requirements while meeting service-level agreements. Additionally, an eBPF-based extension mechanism ensures the system remains safe and isolated while supporting extensibility.

We demonstrate that enabling runtime policy reconfiguration can be implemented with negligible performance and memory overhead, achieving millisecond-scale reload cycles for behavior updates. Our results show that the proposed design effectively balances high performance, adaptability, and safety. Overall, this work demonstrates a practical approach to extensible, cloud-native DBMS deployments by embedding a safe eBPF-style runtime component and a DBMS engine instrumented with extension points into a minimal unikernel, enabling dynamic programmability of core behavior.
The source code for this work is available at:
https://github.com/hristinagrig/BachelorThesis

Contents

1 Introduction

Cloud computing has become widely adopted and now plays a major role in the field of information technology, delivering on-demand compute capacity, storage, and services over the internet by pooling distributed resources across tenants. Cloud platforms offer on-demand self-service, ubiquitous location-independent access to resources, risk displacement, pay-per-use pricing, and reliability [SMM14]. At the same time, location transparency, multi-tenancy, and shared resource pooling raise privacy and security concerns. Consequently, cloud mechanisms must preserve strict isolation across tenants and enforce the least privilege model.

With the rise of cloud computing technologies, databases have also moved to the cloud [Al 13]. Cloud Database Management Systems (DBMSs) are provisioned and operated on cloud infrastructure across data centers. They must be elastic in order to handle variable workloads and provide easy management and accessibility while maintaining safety in the form of correctness and tenant isolation. Traditional DBMSs - designed for relatively static, single-tenant deployments - must be re-designed to exploit cloud capabilities.

Managing large amounts of data while ensuring optimal resource utilization remains a central challenge [AS13]. Cloud-aligned design goals include dynamic scalability, availability, reliability, and on-demand resource provisioning. To address the unpredictability of the cloud environment, many modern systems adopt disaggregation, separating compute from storage (e.g., Aurora's scale-out storage [Ver+17], SQL Server's cloud architecture "Socrates" [Ant+19] and others [Elh+22], [Dag+16], [Gup+15]). While such architectures enable elasticity and scalability, multi-tenancy shifts access patterns in ways that traditional, static designs handle poorly, motivating extensibility.

Decades of research on extensible Database Management System (DBMS) architectures ([Sch+86], [Car+86], [Gut+05]) lay the groundwork for flexible, modular systems that can be customized, enabling the development of domain-specific database engines. In the cloud era, these extensibility principles should be treated as essential: cloud-native DBMSs need the flexibility to tailor behavior to the needs of providers and tenants. Critically, this extensibility must work at runtime.

Today, most cloud applications, including DBMSs, are built over general-purpose operating systems that are heavyweight, resource-hungry, and expose a large attack surface [Pav16]. With limited host resources, reducing per-VM overhead is critical for

cost efficiency. Unikernels address this issue: they are specialized, single-address-space machine images that link an application with only the OS functionality it requires using library operating systems (libOSes), yielding smaller images with faster boot times and reduced attack surface ([Ima18], [Kue+21], [Sysa]). These properties align well with cloud-native trends: enabling small, single-purpose images, lowering memory footprint, and boosting performance by avoiding kernel/user context switches. Unikernels, however, do not solve the challenge of static, ill-suited OS functionalities. They offer limited flexibility and extensibility.

What is needed is dynamic extensibility that will allow the DBMS to efficiently adapt to the rapidly changing, dynamic cloud environment. By supporting runtime adaptation, the system can better accommodate workload variability and evolving hardware infrastructure. Live reconfiguration support can help reduce overheads and enable fine-grained tuning for performance isolation, cost efficiency, and meeting service-level objectives in multi-tenant environments.

We propose a system that enables safe, dynamic reconfiguration of unikernel-based DBMSs through a lightweight extension mechanism. Safety is enforced by verifying and executing extensions in an isolated environment. Furthermore, extensions can interact with the system only through a restricted, predefined helper interface. Concretely, we embed a tiny Runtime for extension support inside a unikernel and package it with the DBMS. The DBMS exposes well-defined hookpoints at key decision sites. At runtime, custom logic can be injected and swapped - without reboots.

We realize a tight OS/DBMS integration by building the OSv [Sysa] unikernel with DuckDB [25]. For the extension mechanism, we use an eBPF-style approach via uBPF [Net], a userspace execution engine for BPF programs. The embedded Runtime loads, compiles, and executes extensions whenever a hookpoint is triggered.

The mechanism is designed to support safety, live adaptability, portability, and low overhead. Our evaluation shows that live reconfiguration support can be achieved without significant performance and memory overhead, and that reloading extensions is substantially faster than static reconfiguration methods that require rebooting the system.

Overall, our work demonstrates that adaptive behavior at the OS/DBMS boundary is practical and safe when combining the eBPF execution model with the lightness of a unikernel environment. It provides a practical path towards extensible cloud-native DBMS deployments that can be tuned at runtime to the dynamic cloud environment.

2 Background

2.1 Cloud-Native Database Systems

Cloud computing is a form of service that provides on-demand access to both physical and virtual computing resources [SS25].

The term "cloud-native" is used for applications that are built, deployed, and managed in a cloud computing environment [Amand]. Cloud-native technologies are expected to adapt to the dynamic and rapidly changing nature of the cloud.

Traditional database systems cannot fully leverage the capabilities of the cloud because they are static, designed for monolithic on-premise environments. That is why a new architecture model is required to support the need for scalability, resilience, and programmability. A cloud-native database should be able to provide rapid elasticity, high availability, and performance, and should continuously adapt to dynamic changes in resource usage [Li+25].

From the user's perspective, cloud-native databases have several particularly appealing features [Don+24a]:

- Elasticity: The database system relieves the customer from managing the computing capabilities by dynamically provisioning and releasing resources.
- Availability: High-availability and recovery are ensured through replication, redundancy, and multi-region deployment.
- Flexibility: Cloud-native applications shift responsibilities from the customer to the provider, offering an out-of-the-box deployment model and automated management. The ability to customize behavior lets clients adapt the system to their individual needs.

From cloud vendors' perspective, one of the main advantages is [Don+24a]:

- Improved Resource Utilization: The database system dynamically adjusts the allocation of resources to different users, based on workload demand, which increases efficiency and cost-effectiveness.

Both perspectives highlight that one of the most valued qualities of cloud-native databases is the ability to adapt rapidly to change. Workloads, service level agreements,

and infrastructure conditions frequently fluctuate in the cloud, and maintaining performance under such variability requires programmability and adaptiveness to be at the core of the system. In practice, this means that policies, parameters, and internal behavior should be adjustable to align with current needs.

2.2 Reconfiguration in Cloud Database Systems

Database systems are simply applications that run in the user space and need the services of an operating system to access hardware resources such as memory, storage, and networking. A DBMS, therefore, relies not only on its own internal mechanisms but also on the functionality provided by the OS. As a consequence, reconfiguration of the database system can take place at two layers:

2.2.1 User-level

User-level reconfiguration allows the application logic and its policies to be changed and refers to modifications made within the database engine itself. This can include, for example, adapting query execution strategies, modifying buffer sizes, and switching indexing policies. In this layer, *what* the database does is modified and extended.

2.2.2 OS-level

OS-level reconfiguration, on the other hand, modifies the environment that the database system runs on and tunes how the platform executes logic. This includes tuning kernel-level mechanisms such as CPU scheduling, memory allocation strategies, and network handling. At this level, *how* the platform executes and manages the database's logic is adjusted.

A cloud-native application can benefit from both. The DBMS can retune itself in response to workload shifts, while the host can specialize in low-level behavior. A simultaneous adaptability at both levels ensures that the system can maintain performance and efficiency even as workloads and resource availability fluctuate in real time.

2.3 eBPF

As discussed in the previous section, reconfiguration in a database system can occur at both the DBMS and OS layers. To enable adaptability at the OS level, we require

a mechanism that operates directly within the kernel. Unlike user-space solutions, OS mechanisms provide global visibility across all processes, efficiency by avoiding expensive context switches, and transparency to the application. One such mechanism is Extended Berkeley Packet Filter (eBPF), a framework that allows safe execution of sandboxed programs in privileged environments [eBP]. Custom programs can be loaded and executed within the operating system kernel without modifying the source code or loading kernel modules.

eBPF programs are attached to well-defined events and are triggered automatically once those events occur. A built-in verifier enforces safety, while helper functions provide controlled interaction with the kernel state. In this way, eBPF extends or modifies the kernel behavior in a safe, programmable, and dynamic manner.

Popular use cases include extending the functionality in areas such as networking, security, and observability ([Cil], [IOV], [Aqu]).

2.3.1 Helper Functions

eBPF programs cannot call arbitrary functions but instead rely on a set of predefined helper functions exposed by the runtime. Interaction with the kernel state is achieved through helpers providing various functionalities. By restricting access to only those functions, eBPF preserves safety without compromising flexibility.

2.3.2 uBPF

User-space Berkeley Packet Filter (uBPF) is a user-space implementation of the eBPF virtual machine (VM). It provides a library for executing eBPF bytecode in user space [Net].

2.4 Unikernels (Operating Systems for the cloud)

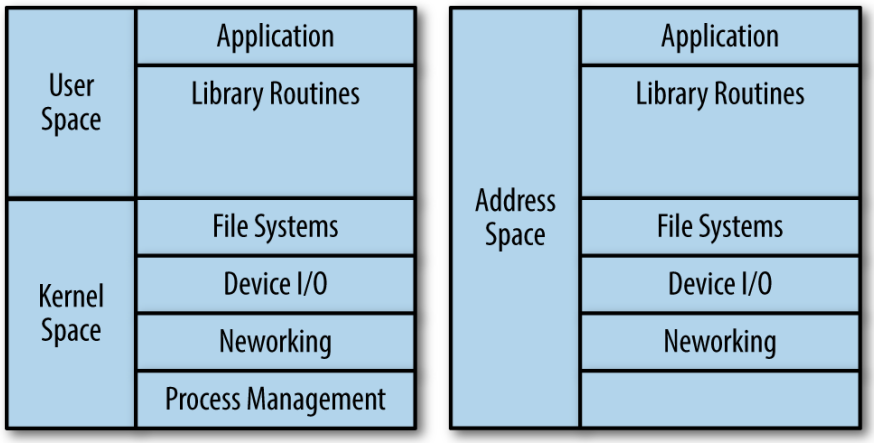
Despite reconfigurability, the challenge of bulky, resource-intensive OSes remains. In the context of could-native DBMSs, it is crucial for efficiency to provide a lightweight, highly specialized execution environment. libOSes achieve this by exposing OS functionality in the form of a library that can be linked directly with the application. This removes the traditional boundary between user and kernel space, allowing the database system to run in a single address space with only the necessary components included.

Unikernels are specialised, single-address-space machine images that leverage library operating systems [com]. They are constructed by linking together an application with only those OS libraries it needs. This results in a VM image produced at compile

time that contains a minimal environment, tailored to the application. By omitting redundant features, unikernels achieve extreme lightweight-ness.

Inside the unikernel, all code runs in a single address space, eliminating traditional user-kernel separation. This tight integration between the OS and the application simplifies communication and reduces overhead (Figure 2.1b). That differs from using general-purpose operating systems, where a strict separation between user processes and the privileged kernel exists, and communication occurs exclusively through system calls (Figure 2.1a).

Unikernels stand out with their small sizes, fast boot times, and security, making them ideal for the needs of the cloud.



(a) Normal Application Stack (b) Unikernel Application Stack

Figure 2.1: Comparison between a traditional OS application stack and a unikernel stack [Pav16]

2.4.1 OSv

OSv [Sysa], originally developed by Cloudius Systems and now maintained as an open-source project. Following the unikernel model, OSv is tailored specifically for the cloud [Sysb], with the objective of running a single unmodified Linux application per image. It is well-suited especially for programs that compile to native machine code (e.g., C, C++, Rust). At the same time, OSv also supports managed runtimes, enabling the execution of applications built on environments such as Java Virtual Machine (JVM), Python, and Node.js.

In this work, we embed uBPF into OSv, where in-kernel eBPF is not supported. By

doing so, we enable the execution of eBPF programs at predefined hookpoints in the DBMS/OS stack, enabling runtime modification of behavior.

3 Overview

3.1 System Overview

We propose a system that enables safe and dynamic reconfiguration of unikernel-based DBMSs using a lightweight, eBPF-based extension mechanism. The high-level view of the system architecture is shown in Figure 3.1. The system consists of the following components:

- **Manager:** receives extensions from authorized admins or clients, performs validation and verification, and is then responsible for uploading the programs into the unikernel.
- **Unikernel:** image that bundles the application and the Runtime.
  - **DBMS:** exposes hookpoints in its source code where logic can be injected. When those hookpoints are reached during execution, the corresponding Runtime functionality is triggered.
  - **Runtime:** acts as the core OS mechanism for executing extensions. It loads the verified bytecode, exposes a Helper Application Programming Interface (API), and runs the extension logic inside a sandboxed, isolated environment.

3.2 Design Goals

Several design goals guide the system:

- **Live adaptability:** Logic can be added, updated, or disabled at runtime without rebooting or stopping the system.
- **Lightweight-ness:** Dynamic update support must not introduce significant overhead nor compromise the lightweight-ness of the system, and the entire mechanism must remain minimal.
- **Portability:** The design should be generalizable. The system avoids platform-specific assumptions and should be compatible with different database management systems.

- **Performance:** For the system to make sense, the throughput of the Database should be maximized and the latency minimized. Hook invocations should be efficient enough to be placed on hot paths, and the overall performance should not degrade significantly after introducing the extension support.
- **Safety:** All reconfigurations must be verified and sandboxed. Each extension runs in isolation, and safety checks prevent faulty code from executing invalid, risky operations. Only a limited set of predefined helper functions is exposed to the hooks.

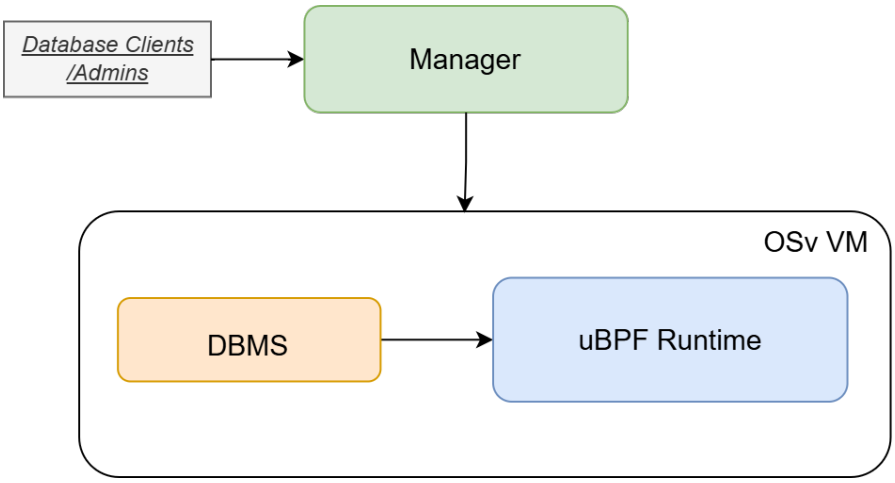


Figure 3.1: Overview of the System's Components

4 Design

The system is designed to evolve dynamically at runtime, without sacrificing performance, safety, or simplicity. This chapter thoroughly explores the system's design choices and main components.

4.1 System Design

We propose an architecture in which the application (in this case, a DBMS) is built together with a specialized runtime component for dynamic reconfiguration into a single-address-space unikernel image. This design enables runtime reconfiguration of the database management system by injecting small, sandboxed programs at well-defined hookpoints in the application source code. A dedicated API is provided for communication between the DBMS, the Runtime, and the extensions.

4.2 System Components

4.2.1 DBMS

Inside the unikernel image, the DBMS is instrumented with carefully chosen hookpoints at crucial decision points, where runtime adaptation can meaningfully influence the behavior. Each hookpoint is defined by a unique name and exposes a minimal context structure, restricting the hook's view of the database system.

When the DBMS reaches a hookpoint, it prepares the necessary context and calls the runtime to execute the hook. The return value is interpreted according to the implementation and should be documented alongside it. If no extension is specified, the system will execute a default policy.

Standard use-cases include buffer-manager eviction decisions, prefetching decisions, and caching policies.

4.2.2 Runtime

The Runtime provides an execution engine for the extensions. Its responsibilities are:

- Accepting bytecode programs received from the manager.
- Exposing a restricted set of helper functions that expand the hook programs' functionality, following the least-privilege principle.
- Preparing an execution environment for each hook and registering relevant helpers.
- Ensuring safety by running each extension in an isolated environment.

After verification, when it is invoked, the Runtime executes the bytecode, passing the context provided by the DBMS as input. Extensions can only interact with the host through the predefined helper functions. Those functions should be carefully selected to avoid exposing sensitive information or privileged functionality.

The system also supports live reconfiguration. If an extension is updated dynamically, the Runtime detects the modification and reloads the new logic. As a result, the updated bytecode is executed the next time the hookpoint is reached - all without rebooting or restarting the unikernel.

4.2.3 Interfaces

The communication between the different components of the system and the extensions is explicit and narrow by design.

- **Hookpoints:** Each hookpoint is identified by its name, fixed function signature and return type, defined by the Application Binary Interface (ABI). On invocation, the DBMS passes a context structure, exposing internal state, and passes it to the extension. It then interprets the return value according to its own logic, while the Manager is responsible for enforcing compatibility with this ABI when admitting new extensions into the system.
- **Helper API:** Extensions interact with the system exclusively through the Helper API, which provides a list of accessible functions, each identified by a numeric ID. The Runtime exposes only the subset of helpers required by the given hook, minimizing the available surface area.
- **Runtime API:** The Runtime provides a narrow management API for the DBMS. It is used at boot time to register the hookpoints and later to load and execute the extensions.

4.3 System's Workflow

The diagram in Figure 4.1 illustrates the system's workflow from extension submission to execution.

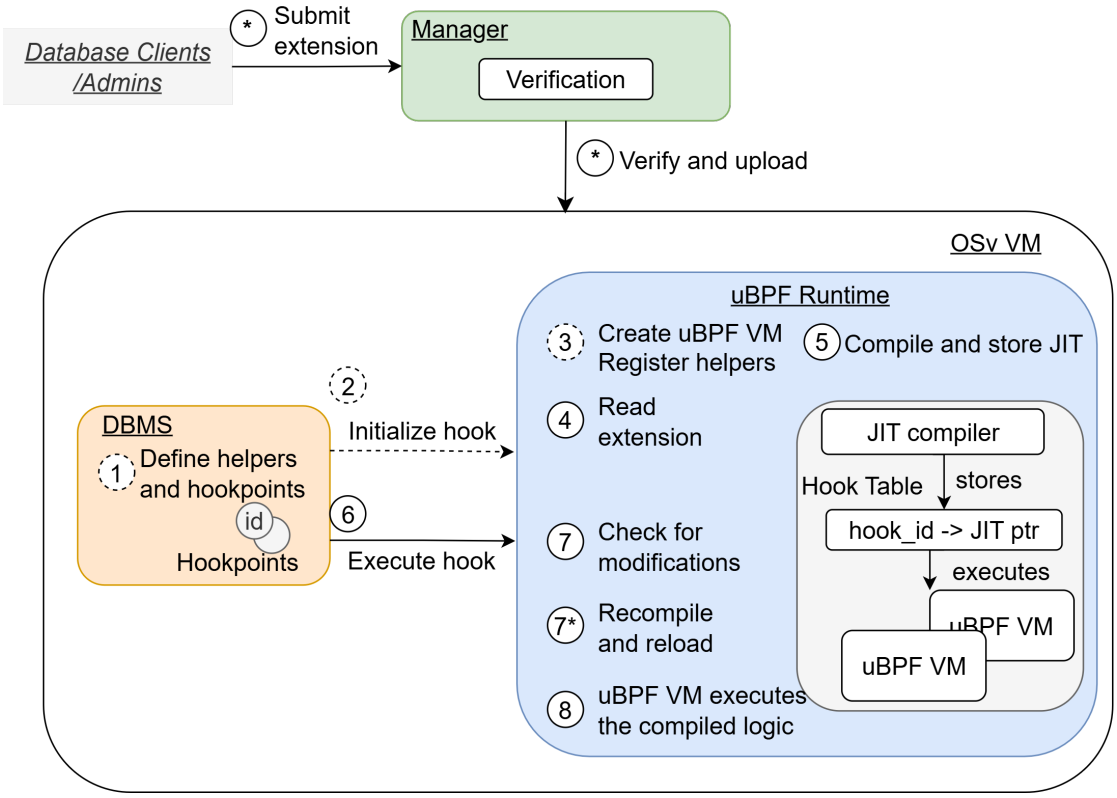


Figure 4.1: System Workflow

- **Submission and Verification (outside of the unikernel)**
  The lifecycle begins outside of the unikernel when a database client or an admin submits an extension package to the Manager. The Manager authenticates the request, validates the package, and uploads it to the VM, making it accessible to the rest of the system's components. This ensures that neither the DBMS nor the Runtime ever handles untrusted code.

- **Boot-Time Registration**
  Before startup, the DBMS declares places in its code where logic can be injected at runtime and specifies a set of helpers that the policies are allowed to call. At

startup, the application invokes the Runtime to execute an initialization routine for each hookpoint. In response, the Runtime creates a dedicated uBPF VM instance for each hook and registers the listed helpers under fixed indices. At this point, the hook is registered and the isolated environment is prepared. Until a verified extension becomes available, the DBMS will continue executing its default behavior.

- **Extension Loading**
  Once a verified extension is submitted, the Runtime reads and compiles the bytecode. Successful compilation yields a callable pointer that is stored in a per-extension key-value map to enable faster and easier execution at a later point in time. If compilation fails, the system simply returns to the old policy.

- **Normal Execution**
  During normal execution of the application logic, when a hookpoint is reached, the DBMS prepares the necessary context structure for the hook and makes a single call to the Runtime. The Runtime executes the extension inside of the sandboxed environment of the dedicated uBPF VM using the precompiled pointer. Any error or violation is reported back, making the DBMS fall back to its baseline implementation.

- **Live Updates**
  The live update support is enabled by monitoring extensions for updates. Whenever a new version of a hook is submitted at runtime, the Manager validates and publishes it. The Runtime detects the modification (or is explicitly instructed via an interface), unloads the corresponding uBPF VM, and reloads the new bytecode by following the same load-and-compile path. This is done without rebooting or interrupting the system.

The combination of out-of-band verification, boot-time registration, and sandboxed execution ensures that extensions can be safely loaded, replaced, and executed at runtime. At the same time, the communication between the system's components is kept minimal, reducing the overhead of function calls and preserving the DBMS performance.

5 Implementation

This chapter presents the system's implementation choices. We describe the key modifications made to OSv and the Database System component DuckDB to support the eBPF-based execution mechanism and explore how the live reconfiguration support is implemented.

5.1 Implementation Overview

Our implementation extends the OSv unikernel by integrating the uBPF library and instrumenting dynamic hookpoints in the DBMS DuckDB. Below we summarize the main changes, repositories and commits used.

- **OSv:** The base of the system is the OSv unikernel [Sysb]. We use the commit 4f0e835 and add a new module under *modules/ubpf/* to host the uBPF runtime.
- **uBPF Runtime:** We implement our Runtime component leveraging the uBPF library [Net], commit 7ce99c7, and integrate it into OSv as a standalone module.
- **DBMS component/ DuckDB:** DuckDB is a high-performance, analytical database system[25] and is used here as the DBMS component for benchmarking and evaluation. It is integrated in OSv under *benchmarks/duckdb/*. We instrument DuckDB, commit ca7ab7c, with hookpoints at selected functions.

5.2 Data Structures

5.2.1 Hook record

We define a small record function, to represent each hook and to store important information about the extension after it has been compiled. The record structure is shown in Listing 5.1.

```
typedef struct {
    char name[32];
```

```
    struct ubpf_vm *vm;
    ubpf_jit_fn fn;
    char prog_path[256];
    char entry_symbol[64];
} compiled_hook;
```

Listing 5.1: uBPF hook record

The structure consists of:

- `char name[32]`: A name field that uniquely identifies the hook.
- `struct ubpf_vm *vm`: Pointer to the uBPF VM instance created for this exact hook. This VM provides the environment where the extension is executed.
- `ubpf_jit_fn fn`: A JIT-compiled function pointer with a uniform ABI that defines the calling convention:

  ```
  typedef uint64_t (*ubpf_jit_fn)(void *ctx, size_t ctx_size);
  ```

  The caller passes a pointer to a context buffer with its size and receives a 64-bit result.
- `char prog_path[256]`: Absolute or relative path to the ELF object (.o file) used for loading or reloading of the hook.
- `char entry_symbol[64]`: Symbol name of the program's entry point (main function name) within the ELF.

Hook records are stored in a fixed-size array for simplicity.

5.2.2 Helper Function Descriptor

We implement another record (see Listing 5.2) to describe each helper function. The descriptor is used to register helpers in the uBPF VM during setup so that the loader can resolve helper calls and bind them to numeric IDs.

```
typedef struct {
    const char *name;
    void *fn;
    int index;
```

```
} ubpf_helper_descriptor;
```

Listing 5.2: Helper Function Descriptor

The fields of the record are as follows:

- `const char *name`: Symbolic helper name that is used by the ELF loader to resolve the call to the helper from the BPF program. Names must be kept unique.
- `void *fn`: Native function pointer implementing the helper. The runtime doesn't enforce a fixed ABI, so the helpers may have different signatures and return types. Therefore, calling conventions must be documented.
- `int index`: The numeric, unique helper ID. The loader will rewrite calls by name to this index at load time.

Helpers are stored in an array, which is passed to the Runtime.

5.3 uBPF Runtime

The Runtime is responsible for the execution of extensions and is implemented using uBPF. It provides an API for helper registration, loading and compiling BPF programs, and invocation of hooks at well-defined points.

Both the Runtime and the DBMS include the shared header (`#include "ubpf_runtime.h"`), which defines the function signatures, to enforce consistent interfaces.

5.3.1 Initialization

We define a function for hook initialization with the following signature:

```
int initialize_hook(const char *hook_name,
    const char *prog_path, const char *
    entry_symbol);
```

This call uses the uBPF library functions `ubpf_create()` and `ubpf_register()` to create a dedicated uBPF VM and register helpers by iterating the helper descriptor table:

```
for (size_t i = 0; i < num_helpers; ++i) {
    ubpf_register(vm, my_helpers[i].index,
        my_helpers[i].name, my_helpers[i].fn);
}
```

Each hook has its own VM, simplifying lifecycle management (e.g., reloading, destroying) and isolating failures.

5.3.2 Load and Compile

After helper registration, `initialize_hook` continues to the load and compile path.

5.3.2.1 ELF Loading

Programs are read from Executable and Linkable Format (ELF) object files (.o files). The function from the uBPF library:

```
ubpf_load_elf_ex(struct ubpf_vm* vm, const void*
    elf, size_t elf_len, const char*
   main_section_name, char** errmsg);
```

is invoked to load the program into the VM and resolve calls to helpers by binding them to their corresponding registered implementations.

If the loading fails, the runtime aborts and destroys the VM.

5.3.2.2 JIT Compilation

After a successful load, the Runtime invokes `ubpf_compile(struct ubpf_vm* vm, char ** errmsg)`, provided by uBPF, to Just-In-Time (JIT) compile the bytecode to native machine code and obtain a JIT function pointer with the uniform signature: `uint64_t (*)(void *ctx, size_t ctx_size)`

The compiled function together with the hook's metadata is then stored in a compact `compiled_hook` record. All records are then inserted into a fixed-size array for easier access, making the hook IDs stable array indices. The path and entry symbol fields (see Listing 5.1) allow the same hook to be rebuilt or reloaded later.

5.3.3 Hook Invocation

We implement a function for runtime invocation of the Runtime component via:

```
int call_hook(const char* hook_name, void *ctx,
    size_t ctx_size, uint64_t *result);
```

This function resolves the hook by name, uses the JIT function pointer from the record of the hook, and directly calls it with the provided context and size. The context layout is part of the hook's calling convention and is documented alongside it. On successful execution, it stores the returned value in `result` and returns 0.

If the name of the hook is invalid, it returns -1.

5.4 DuckDB

To evaluate the proposed reconfiguration mechanism in a realistic setting, we integrate DuckDB into the system. In order to enable interaction with the Runtime, we instrument DuckDB by introducing hookpoints into its source code.

To run the application with hook support, it has to be compiled with the CMake option `ENABLE_HOOKS=ON` which defines a preprocessor macro `HOOKS`. The DBMS implements two execution paths: one with hook support enclosed within `#ifdef HOOKS` blocks, and another that contains only the baseline behavior.

DuckDB links against the Runtime by including the `ubpf_runtime.h` header. At the start of the process, each hook is initialized with its default extension. The initialization is explicit: for example,

```
#ifdef HOOKS
      initialize_hook("hash", "/hooks/hash_hook.o",
      "hash_main");
      initialize_hook("unpin", "/hooks/unpin_hook.o",
      "unpin_main");
//initialize all other extensions that will be called
#endif
```

Invoking a hook at each hookpoint follows the same pattern:

1. Build a context structure.
2. Call the Runtime by passing the context and size by invoking the `ubpf_create` method.
3. Interpret the returned 64-bit value as documented and steer the code based on it.

The example below shows a "hash" hook that returns the hashed value:

```
hash_t Hash(string_t val) {
#ifdef HOOKS
    hash_hook_ctx ctx = { \* prepare the context structure *\ };
    uint64_t result;
    call_hook("hash", &ctx, sizeof(ctx), &result);
    if(result>0){
        return result;
    }
// if the hook is missing or fails, fall through to the
    baseline
```

```
#endif
// if HOOKS is not defined, continue with default
    implementation
```

We define the context structure once in a shared header file (`hook_ctx.h`) and is used both by the DBMS and the BPF programs. For example `hash_hook_ctx` is defined as follows:

```
struct hash_hook_ctx{
    const uint8_t *data;
    uint64_t len;
};
```

This results in a single source of truth for the context definition and avoids a mismatch between the caller and the extension.

5.5 Extensions and Helper Functions

As described earlier, we introduce hookpoints into DuckDB's source code, each associated with a default hook implementation. To ensure fair evaluation, the hooks replicate the behavior of the DuckDB's original functions they replace. This requires careful definition of helper functions that expose the necessary internal functionality and state from the system to the hooks. Hookpoints are places at performance-critical execution paths to ensure that their impact on performance can be meaningfully measured.

In Table 5.1 we list the implemented hookpoints, where they are located, and their default behavior. The last column reports the number of helper functions defined for the hook. In our case, we register 23 helpers, most of which expose internal DBMS operations (e.g, managing scheduling queues, controlling buffer access and eviction, and performing atomic memory updates) rather than general-purpose utilities. This grants BPF extensions controlled access to DuckDB internals.

Authors of BPF extensions must declare the needed helpers and call them accordingly. A mismatch between the program's declaration and the helper's prototype will lead to a runtime error.

An extension (BPF program) is a C file that implements one hook's behavior. It is compiled to an ELF object (.o), that uBPF can load and is compiled, via the command:

```
clang -O2 -target bpf -c <HOOK_SRC>.c -o <HOOK_OBJ>.o
```

where `<HOOK_SRC>` is the path to the hook's C source and `<HOOK_OBJ>` is the desired output object. Extensions can only use the context passed by the DBMS and call the helper functions that the host registers into the uBPF VM. Below, we provide a thorough

Hookpoint function	# Helpers
Hash	1
ExecuteForever	9
ScheduleTask	1
Load	2
UpdateUsedMemory	3
AllocateNewBlock	2
Unpin	5

Table 5.1: DuckDB hookpoints: function, file, semantics, and number of helpers declared for that hook.

guide on introducing hook support.

Recipe for writing extensions:

1. Include the shared header that defines the hook's context structure.
2. Declare all the helpers that you are going to need.
3. Define the entry function with the name configured when calling `initialize_hook`. The signature must match the JIT entry type.

Defining helper functions (host side): Helpers are functions that hooks may call and are registered into each VM during initialization. An example helper function and helper array definition:

```
// ubpf_helpers.cc (host side)
#include "ubpf_runtime.h"
#include <stdint.h>
uint64_t bpf_print(char* a) {/* implement helper
    functionality*/}
uint64_t bpf_memcpy(void *ctx, size_t ctx_len) {/*
    implement helper functionality */}
ubpf_helper_descriptor my_helpers[] = {
    {"bpf_print", (void *) bpf_print, 1 },
    {"bpf_memcpy", (void*) bpf_memcpy, 2},
} // descriptor table used by register_helpers
```

```
size_t num_helpers = sizeof(my_helpers) / sizeof(
    my_helpers[0]);
```

Helpers that expose the DBMS' internals: The helpers run in the same address space as the DBMS, so they can use database functions to provide internal functionality. However, this requires consciousness about object access and function visibility.

If a helper needs to call a non-static DBMS method, it must receive a handle to the owning object from the hook, which should be passed via the hook's context. In practice, we pass pointers of type `uint64_t` and then cast back on the host side. This keeps the BPF side simple. The pattern is:

1. DBMS: pass a pointer to the object as `uint64_t` scalar
2. BPF: pass the arguments to the helper, including the pointer
3. Helper: cast the pointer to the appropriate object (perform null checks) and then call the method

Accessing private/protected members: The needed functions and attributes are not always public, in which case there are three options on how to proceed:

1. Setters and Getters: Implement accessor functions for private variables.
2. Helper wrapper methods: Expose helper functions from the DBMS that perform checks and call private routines from the inside.
3. Friendship: Declare the BPF helper function (or a class) as a *friend* of the DBMS class. This gives access to private and protected members.

Invoking helper functions (hook side): The helper functions that are going to be called from the hook should be declared in the BPF program like this:

```
// hook.c (hook side)
#include <stdint.h>
static uint64_t (*bpf_print)(char* a) = (void*)1;
static uint64_t (*bpf_memcpy)(void *ctx, size_t ctx_len
    ) = (void*)2;
```

The system does not enforce a fixed ABI for the helpers: helpers may expose arbitrary C prototypes. As a result, the BPF programs must ensure that the functions are declared and called exactly as they are documented. Once declared, the program can invoke helpers as normal functions:

```
// hook.c (hook side)
uint64_t result = bpf_memcpy(&args, sizeof(args));
// invoke helper with the appropriate arguments
```

5.6 OSv Integration

OSv composes images from "modules". Each module defines the components to be built, the files to be copied into the file system, and the corresponding run configurations. The Runtime and the DBMS are defined as separate modules and installed together into one OSv image. Both modules contain a Makefile which will be automatically built with the *make module* command [Clo15].

5.6.1 Manifest files

Modules declare files to be uploaded to the OSv VM using a *usr.manifest* file [Clo15]. The structure of a manifest file looks like:

```
[manifest]
/usr/file1: ${MODULE_DIR}/file1
/usr/file2: ${MODULE_DIR}/file2
```

In our implementation, the DBMS and the Runtime manifest files should include the extensions object files and the Runtime library. As a result, those files will be placed in the root of the OSv VM.

5.6.2 module.py

By defining a *module.py* file, the module declares a run configuration and can declare a dependency on other modules. In DuckDB the following *module.py* is created:

```
from osv.modules import api
api.require('ubpf')
default = api.run('/duckdb')
```

When the *duckdb* module is built, the *ubpf* module is built first, and its *usr.manifest* is pulled into the image. When the VM starts, OSv launches */duckdb*: the default boot command, defined by the composing module.

5.6.3 Linking the Runtime into the DBMS

DuckDB's build links against the uBPF runtime as a shared library. The uBPF runtime is integrated into the DBMS at build time using an imported, prebuilt CMake target. The top-level Makefile supplies two paths into CMake: *UBPF_INCLUDE_DIR* (the uBPF headers) and *UBPF_LIBRARY* (the shared object, *ubpf.so*).

The CMake materializes these paths as an imported target:

- `add_library(ubpf SHARED IMPORTED GLOBAL)` declares a target named *ubpf* that is not built in the project, but is available for linking.

- ```
  set_target_properties(ubpf PROPERTIES
      IMPORTED_LOCATION "${UBPF_LIBRARY}"
      INTERFACE_INCLUDE_DIRECTORIES "${UBPF_INCLUDE_DIR}")
  ```

  points the linker to the exact location of the .so file and publishes the uBPF header directories to any target that links to *ubpf*.

- All DBMS binaries then link against *ubpf* in the usual way (e.g. `target_link_libraries (duckdb ubpf)`), establishing a runtime dependency.

Using a shared library keeps the DBMS binary smaller and allows the Runtime to be updated independently.

Because the Runtime is a shared library, the .so file must be available in the OSv image. We therefore need to use a *usr.manifest* file to copy the shared object into the root filesystem of the OSv VM so that the dynamic loader can resolve it at startup.

5.6.4 Hook Instrumentation

Hook support is a build-time option. The CMake option `ENABLE_HOOKS` is a build-time switch that decides if DuckDB should compile the hook paths. When enabled, CMake defines a preprocessor macro `HOOKS` and the code under `#ifdef HOOKS` gets compiled. When disabled, those paths are left out.

To set the option, the OSv image should be built with `CMAKE_ARGS="-DENABLE_HOOKS=ON"`. It then propagates this flag down to the CMake via DuckDB's Makefile. CMake picks it up to define the macro.

5.7 Live Reconfiguration

Updating hooks without rebooting the VM can be achieved by starting the image with *Virtio-fs*, a shared filesystem. It mounts a host directory inside of the OSv guest, so
  ```

the system can treat the hook objects as ordinary files whose contents may change at runtime.

6 Evaluation

The following chapter aims to evaluate whether the proposed system design addresses the following central research questions:

1. Can the system provide live reconfiguration support while preserving the performance of the database?
2. Is the design lightweight, introducing minimal memory overhead relative to the baseline?
3. Does the implemented mechanism improve reconfiguration times compared to a statically configured system that requires a reboot to apply changes?

The goal of these evaluations is to demonstrate that the system enables dynamic extensibility without introducing significant overhead.

6.1 Environment

The measurements and benchmarks are conducted on a machine with the following specifications:

- CPU: AMD EPYC 9654P, 96 cores / 192 threads, base frequency 1.5 GHz (up to 3.7 GHz), with 384 MB L3 cache, 96 MB L2 cache, and 3 MB L1 data + 3 MB L1 instruction cache.
- Memory: 752GiB of RAM @ 4800 MT/s

The system under evaluation is the extended OSv unikernel with integrated uBPF Runtime and DuckDB, compared against the baseline: OSv without the Runtime, only the database system. Each OSv instance is run inside a QEMU (version 8.1.2) virtual machine.

6.2 Performance

Live reconfiguration support is only valuable if it can be achieved without compromising the DBMS performance. Evaluating performance for cloud-native systems is

crucial to ensure that the system will maintain its efficiency and continue to meet service-level objectives. The following benchmarks aim to validate that the proposed design preserves low-latency query execution, scales with the addition of extensions, and introduces only insignificant overhead compared to the baseline.

6.2.1 Metrics

We focus on the following metrics for the performance evaluation:

- Query latency (*real_s*): the total runtime for each query.
- Hook invocations: the number of hook executions triggered during query processing.

The metrics are collected by repeatedly executing the queries to measure stability. At the end, the mean across multiple runs is computed to reduce noise.

6.2.2 Methodology

We use the industry-standard TPC-H benchmark (decision-support queries with high complexity) [Tra] consisting of 21 queries to evaluate the performance.

The evaluation is automated using a benchmarking script that:

- Builds the system image (with or without uBPF).
- Executes the TPC-H workload at scale factor 20, repeatedly 10 times.
- Collects runtime metrics by parsing DuckDB's built-in timing output. In addition to that, it also tracks the statistics reported by the uBPF Runtime, which has been instrumented to collect and output information about the number of hook calls.
- Aggregates and averages the results into CSV files.

6.2.3 Real Time per Query

To interpret performance differences, we consider the hook activity. Figure 6.1 shows the average number of hook invocations triggered per query under TPC-H at scale factor 20. The distribution is highly skewed: some queries (Q10, Q15, Q21) trigger millions of hook calls, while others only tens of thousands. A logarithmic scale is used to better capture these differences. The results indicate that depending on the query structure and the placement of hookpoints, the hook mechanism is utilized very differently across queries.

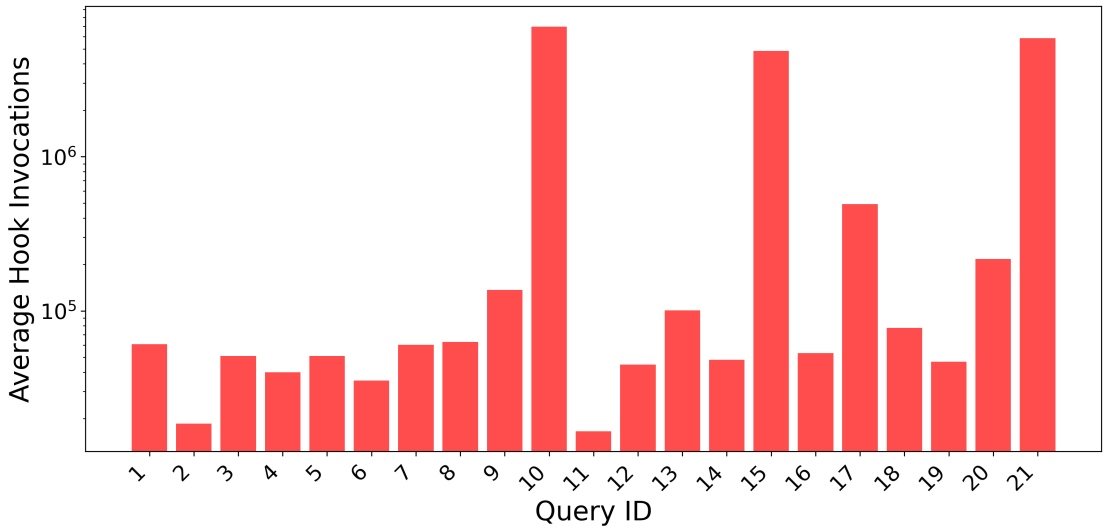


Figure 6.1: Hook Invocations per Query

6.2.4 Result

Figure 6.4 compares per-query runtimes between the baseline OSv system and the uBPF-enabled system. For most queries, the runtime difference between the two configurations is negligible, with overheads well below 5%. The largest differences correspond to the highest number of hook invocations (as shown in Figure 6.1). Even in those stress cases, the overhead remains modest relative to the overall runtime.

RQ1 takeaway: The results demonstrate that live reconfiguration support can be integrated with minimal performance impact.

6.3 Memory Footprint

Another key design goal of the system is lightweight-ness, emphasizing minimizing memory overhead. We therefore compare the OSv image library sizes between the baseline (without uBPF) and the uBPF-enabled configuration.

6.3.1 Methodology

OSv images are dynamically linked at runtime: required application libraries are included in the root file system via the *usr.manifest* file. The relevant libraries are:

- *libosv.so:* provides the OSv core runtime and system functionality.

- *libduckdb.so:* the database system library, (either the baseline version vs. the version instrumented with hook support).
- *libubpf.so:* the shared library that provides the uBPF Runtime.

It is sufficient to compare the sizes of those libraries, since they dominate the overall image size and represent the components that differ between the two configurations.

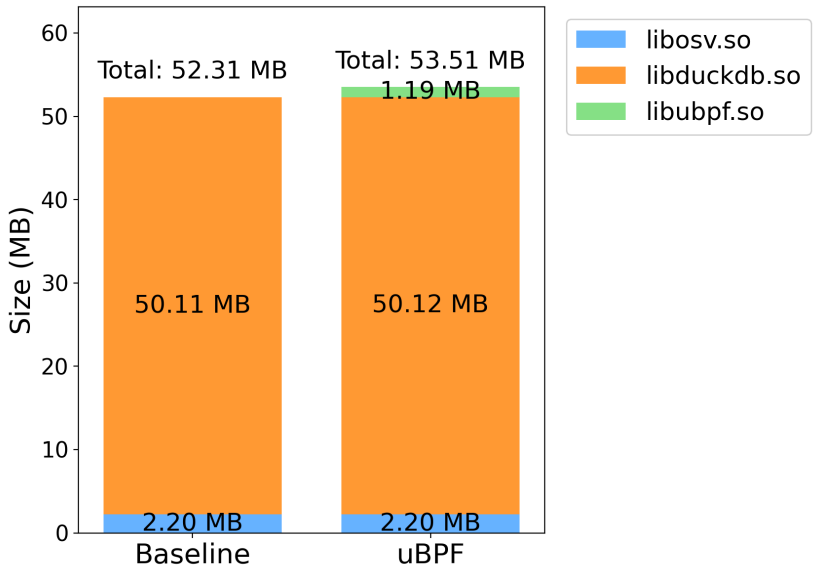


Figure 6.2: Library Sizes per Build

6.3.2 Results

Figure 6.2 presents the library sizes for both configurations. The baseline system consists only of the *libosv.so* and *libduckdb.so* library, totaling to **52.31MB**. Adding the uBPF Runtime introduces *libubpf.so* (1.19MB) and implementation of hookpoints and helper functions increases the DuckDB library size by 0.01MB, resulting in a total of **53.51MB**.
In addition, the OSv root file system includes the object files (.o) of the extensions. These are minimal, averaging 1KB per extension in our implementation.

RQ2 takeaway: Overall, we can see that enabling live reconfiguration introduces insignificant memory overhead, and maintains the leightweight-ness of the unikernel design. By preserving a reduced execution environment and limiting resource consumption, the system ensures suitability for cloud-native applications.

6.4 Reconfiguration time

Reconfiguration is the core feature of our system, so we quantify how much faster live reloading is compared to rebooting the unikernel.

6.4.1 Methodology

- Baseline (reboot): In order to reconfigure the baseline system, we need to perform a plain reboot of OSv with the new image (assuming that it is already built at the time of rebooting). OSv reports its own boot latency at startup.
- Live reconfiguration (reload): When reconfiguring the application using uBPF, we need to reload the corresponding BPF extension in-process. We time only the reload path - unloading the previous bytecode from the VM, loading and JIT compiling the new bytecode - without assuming any particular upload mechanism (e.g., API, shared file system (FS), etc.). The elapsed time is measured with a monotonic wall clock. We evaluate the reload time for all hooks and compute the average.

  We perform 10 independent runs and report the mean across all trials.

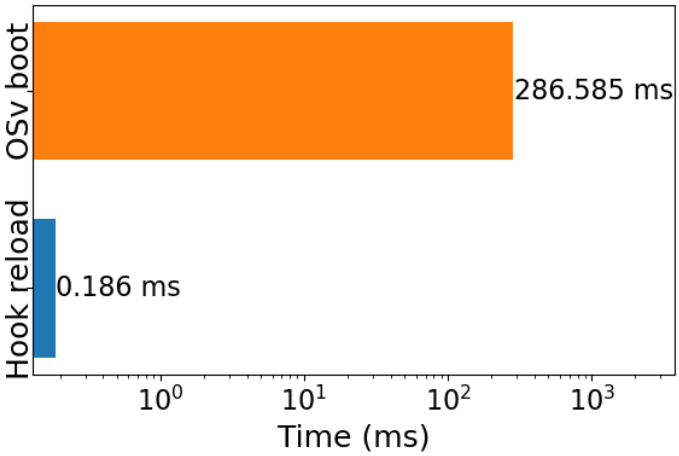


Figure 6.3: OSv Boot Time vs Hook Reload Time

6.4.2 Results

Figure 6.3 shows the results on a logarithmic scale. Across 10 runs, OSv takes on average **286.585 ms** to reboot, while the reload path takes **0.186 ms**.

RQ3 takeaway: The uBPF mechanism substantially outperforms static updates, enabling efficient, low-latency dynamic reconfiguration support. The minimal overhead and almost instant reconfiguration with zero downtime align well with the requirements of the cloud, where rapid elasticity and high availability are crucial.

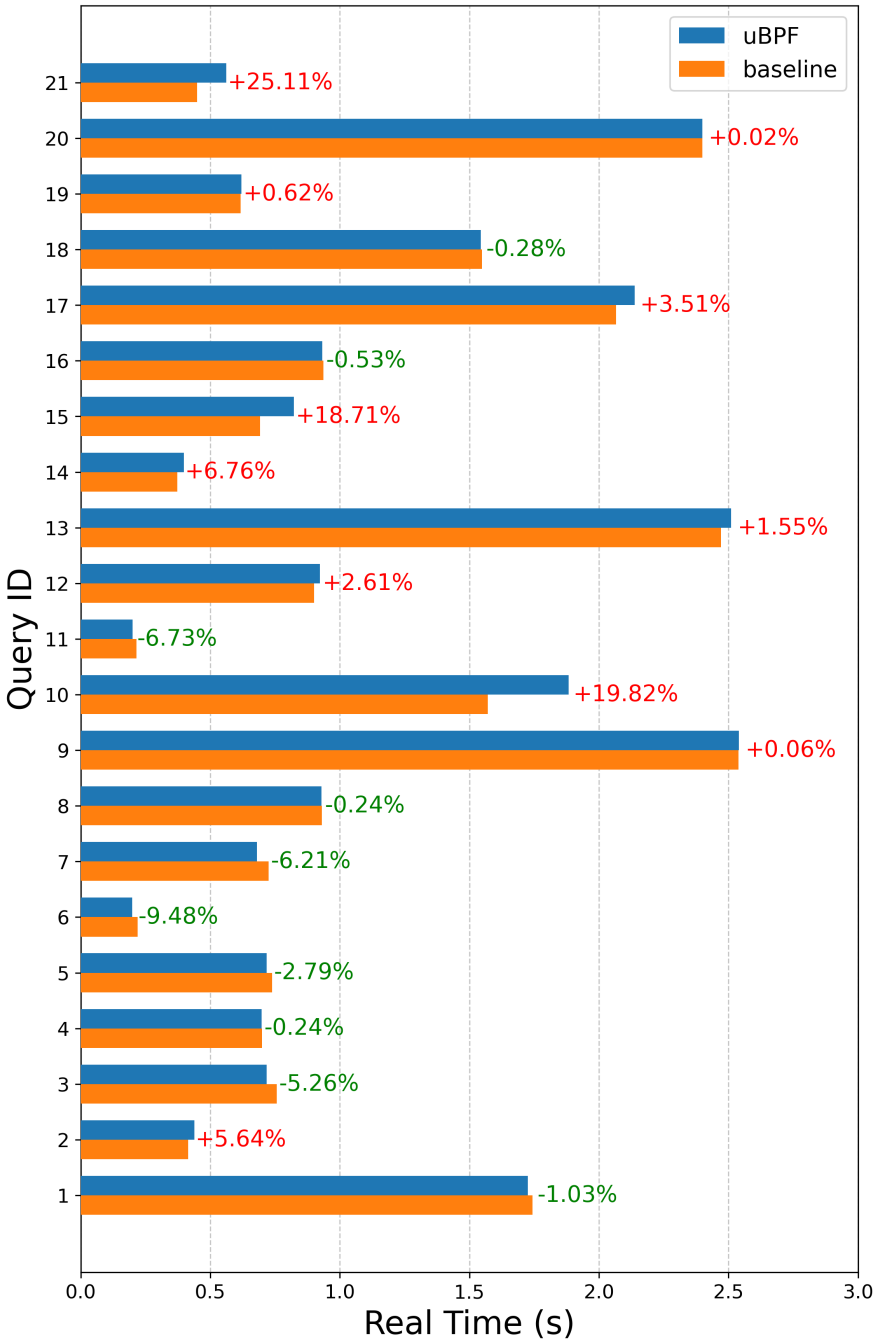


Figure 6.4: Real Time (s) per Query (with % difference labels)

7 Related Work

7.1 OS/DBMS Co-Design

The idea of aligning DBMS and OS design has a long history. Stonebraker (1981) [Sto81] argues that the DBMS should be engineered with explicit awareness of the OS it runs on. When general-purpose OS services are a poor fit, DBMS developers end up re-implementing missing functionality in user space - forming a small "user-level OS". The paper shows that many standard OS services are either too slow or ill-suited for the DBMS needs, motivating a minimal, specialized OS and tighter OS/DBMS co-design.

Building on this critique, subsequent work explores how to formalize cooperation between the two layers and focuses on providing a clearer communication interface. COD [Gic+13] achieves tighter OS/DBMS integration by combining the database system's knowledge about its own resource demands with the OS's global view of the hardware and co-running applications. By rethinking the DBMS/OS interface and introducing an OS level policy engine for deployment and placement, COD demonstrates that policy coordination across the DBMS/OS boundary improves performance and predictability compared to black-box OS scheduling.

A more radical approach towards OS/DBMS co-design includes redesigning the operating system stack itself. DBOS [Ski+21] is a DBMS-oriented Operating System, specialized for distributed applications. The operating system is re-designed by placing a multi-node, transactional DBMS at the center, as the control plane. It proposes a four-layer architecture (microkernel, DBMS, OS services, user space) and implements core OS services as SQL queries over relational tables. The "everything-is-a-table" model demonstrates that DB-backed OS services can achieve competitive performance while enhancing analytics and simplifying service implementation.

Virtualization has also been used to adapt OS interfaces to database systems' demands. LIBDBOS [Zho+25] achieves compatibility with minimal host-kernel modification by introducing a lightweight guest kernel with two DB-oriented virtual-memory mechanisms. The result is a privileged kernel bypass model in which the DBMS runs with elevated privilege inside the guest and co-designs critical VM subsystems. By granting the DBMS fine-grained control, LIBDBOS reports significant latency and efficiency gains over traditional designs.

Our system takes a more incremental and dynamic approach to OS/DBMS co-

design. Rather than replacing the OS (DBOS) or introducing a new virtualization layer (LIBDBOS), it enables online reconfiguration of both OS and DBMS behavior. By leveraging unikernels and lightweight hookpoints inside the DBMS, we achieve database-aware adaptability without the complexity of redesigning the OS stack. We focus specifically on runtime flexibility in cloud deployments where workloads and hardware configurations change frequently.

7.2 Operating System Extensibility

While DBMS/OS co-design improves efficiency and inflexibility via tighter cross-layer integration, a parallel line of work addresses the same issue by evolving the Operating System functionality itself. Extensible kernels allow the OS to evolve in response to application demands. There are many approaches to modify the kernel in order to improve performance, differing in the technology used and the philosophy behind extensibility [SS96].

SPIN [Ber+95] is an extensible operating system, allowing for a safe modification to the OS's interface and implementation, by loading type-safe extensions into the kernel address space. SPIN's design combines co-location by executing extensions inside of the virtual kernel address space, enforced modularity via interfaces, logical protection domains through namespaces, and dynamic call binding on events. This model achieves extensibility without sacrificing safety and performance.

Instead of implementing a new OS architecture, systems like SLIC [Gho+98] extend existing commodity OSes by inserting extension code. SLIC uses kernel-level interposition, dynamically intercepts kernel interfaces, and redirects them to trusted extensions. This work demonstrates how static interposition can be used to evolve existing systems.

More recently, eBPF has emerged as a powerful kernel extensibility runtime, letting verified programs be loaded into the OS [Gba+24]. By embedding a virtual machine runtime and verifier inside the kernel, eBPF ensures safety while allowing modifications of kernel behavior at designated hooks, without bypassing or replacing the kernel. This model enables fine-grained, dynamic extensibility in monolithic kernels like Linux and Windows.

Unlike approaches that require new operating environments and kernel structures, our system adopts the eBPF execution model but relocates it into a unikernel, providing lightweight and specialized runtime adaptability.

7.3 Database System Extensibility

While OS extensibility adapts the kernel to applications' needs, many database systems pursue a complementary route: extending the DBMS engine itself. Kim et al. [Kim+] survey DBMS extensibility across multiple systems such as DuckDB, PostgreSQL and MySQL, proposing a taxonomy of extension types and design dimensions. This work highlights practical pitfalls - extension incompatibilities, unexpected failures, and correctness issues - highlighting the challenge of balancing safety and extension flexibility.

The Starburst Database System [Sch+86] is an early example of extensible relational DBMS design. This system adapts to new applications and technologies by letting trusted developers implement new types and methods invoked from within the DBMS through well-defined interfaces, without a complete redesign.

However, most production DBMSs, including Starburst, rely on static extension mechanisms - new functionality is compiled, loaded, and deployed via controlled release cycles. They expand DBMS functionality but leave the OS layer unchanged. In contrast to these static mechanisms, we focus on live reconfiguration of system behavior across both the DBMS and OS without restarts.

7.4 Cloud-native DBMS

Beyond extending the DBMS functionality, the rise of cloud computing and the heterogeneous and dynamic environment of the cloud have created the need to rethink the design of cloud-native database systems. Dong et al. [Don+24b] synthesize and evaluate trends across new techniques, designs, and architectural choices and discuss their trade-offs.

PolarDB [Wan+23] is a cloud-native Hybrid Transactions and Analytics Processing (HTAP) database system, built around five design goals: transparency, competitive OLAP performance, minimal perturbation of OLTP, high data freshness, and resource elasticity. It adopts a separated compute-storage architecture model, with multiple computation nodes, isolating analytical from transaction processing queries. PolarDB integrates an in-memory column index storage for analytics, while supporting row-based storage engines optimized for transactions. While PolarDB focuses on advancing the internal database architecture and workload isolation, we target runtime adaptation of the system's behavior.

Finally, Leis and Dietrich [LD24] argue that cloud-native DBMSs should extend beyond internal optimization and should also adapt the OS functionality. They propose customizing the OS kernel by leveraging unikernels for co-designed high-performance

cloud-native DBMS, reducing complexity, removing system call costs, and providing direct access to hardware primitives. A minimal kernel can cut overheads and offer a better-suited API for DBMS needs. Our system builds on this vision, leveraging the capabilities of the OSv unikernel.

Our work lies at the intersection of three strands: DBMS/OS co-design, runtime OS programmability, and DBMS extensibility in cloud-native environments. Prior work delivers static specialization (Starbust, COD), kernel extensibility in general-purpose OSes (SPIN, SLIC,eBPF), and radical redesigns (DBOS, LIBDBOS). Building on this foundation, our work shifts focus to runtime reconfiguration by extending the OSv unikernel with a uBPF Runtime and instrumenting the DBMS with hookpoints. This design enables dynamic adaptation of policies across the OS/DBMS boundary, combining the safety and flexibility of the eBPF model with the lightweight specialization of unikernels, ultimately providing a practical path towards extensible cloud-native DBMS deployments.

8 Summary and Conclusion

This work introduces a lightweight mechanism for safe, online reconfiguration at the OS/DBMS boundary. We embedded a tiny uBPF-based runtime inside the OSv unikernel and exposed well-defined hookpoints in the database system. Extensions are JIT-compiled, executed in an isolated, sandboxed environment, and hot-swappable at runtime without rebuilds or reboots to ensure high performance, safety, and efficiency. The design exposes a uniform ABI and a minimal API and helper surface for cross-layer communication, enabling DBMSs to define narrow, dynamically reconfigurable hookpoints.

We evaluated the approach along footprint, performance overhead, and reconfiguration latency. The results show that adaptive behavior in a minimal VM image is both feasible and useful: clients and admins can specialize behavior online while retaining the small footprint, isolation, and fast-boot properties of unikernels.

Looking ahead, we plan to broaden the hook surface to support richer decisions and add carefully constrained helpers to enable more expressive policies. The system can be extended to distributed settings, coordinating policies across VMs, integrating with disaggregated storage, with attention to consistency and tenant interference. Finally, observability can be added to enable SLO-driven automation and diagnosis in production-like deployments.

Overall, this work demonstrates a practical path to extensible, cloud-native DBMS deployments: combine a minimal unikernel, an embeddable DBMS engine, and a safe uBPF-style Runtime component to make core execution policies dynamically reconfigurable. The result is a system that preserves the efficiency and lightweight-ness of unikernels and the safe eBPF execution model while unlocking millisecond-scale adaptability at runtime.

The source code for this work is available at:
https://github.com/hristinagrig/BachelorThesis

Abbreviations

libOSes library operating systems

OS Operating System

eBPF Extended Berkeley Packet Filter

uBPF User-space Berkeley Packet Filter

DBMS Database Management System

DBMSs Database Management Systems

API Application Programming Interface

ABI Application Binary Interface

VM virtual machine

JIT Just-In-Time

ELF Executable and Linkable Format

FS file system

HTAP Hybrid Transactions and Analytics Processing

List of Figures

List of Tables

Bibliography

[25] *DuckDB*. Accessed: 2025-08-19. 2025.

[Al 13] W. Al Shehri. "Cloud database database as a service." In: *International Journal of Database Management Systems* 5.2 (2013), p. 1.

[Amand] Amazon Web Services. *What is Cloud Native?* AWS Website. Accessed: 2025-08-13. n.d.

[Ant+19] P. Antonopoulos, A. Budovski, C. Diaconu, A. Hernandez Saenz, J. Hu, H. Kodavalla, D. Kossmann, S. Lingam, U. F. Minhas, N. Prakash, et al. "Socrates: The new sql server in the cloud." In: *Proceedings of the 2019 International Conference on Management of Data*. 2019, pp. 1743–1756.

[Aqu] Aqua Security. *Tracee: Runtime Security and Observability using eBPF*. `https://github.com/aquasecurity/tracee`. Accessed: 2025-08-26.

[AS13] M. Alam and K. A. Shakil. "Cloud database management system architecture." In: *UACEE International Journal of Computer Science and its Applications* 3.1 (2013), pp. 27–31.

[Ber+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. "Extensibility safety and performance in the SPIN operating system." In: *Proceedings of the fifteenth ACM symposium on Operating systems principles*. 1995, pp. 267–283.

[Car+86] M. J. Carey, D. J. DeWitt, J. Richardson, and E. Shekita. *Object and file management in the EXODUS extensible database system*. Tech. rep. University of Wisconsin-Madison Department of Computer Sciences, 1986.

[Cil] Cilium Project. *Cilium: eBPF-based Networking, Security, and Observability*. `https://github.com/cilium/cilium`. Accessed: 2025-08-26.

[Clo15] Cloudius Systems (OSv Wiki). *OSv modules*. GitHub Wiki. Accessed: 2025-08-15. May 2015.

[com] U. community. *What are unikernels?* `http://unikernel.org/`. Accessed: 2025-08-11.

[Dag+16] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, et al. "The snowflake elastic data warehouse." In: *Proceedings of the 2016 International Conference on Management of Data*. 2016, pp. 215–226.

[Don+24a] H. Dong, C. Zhang, G. Li, and H. Zhang. "Cloud-Native Databases: A Survey." In: *IEEE Transactions on Knowledge and Data Engineering* 36.12 (2024), pp. 7772–7791. DOI: 10.1109/TKDE.2024.3397508.

[Don+24b] H. Dong, C. Zhang, G. Li, and H. Zhang. "Cloud-Native Databases: A Survey." In: *IEEE Transactions on Knowledge and Data Engineering* 36.12 (2024), pp. 7772–7791. DOI: 10.1109/TKDE.2024.3397508.

[eBP] eBPF Foundation. *What is eBPF?* https://ebpf.io/what-is-ebpf/. Accessed: 2025-08-11.

[Elh+22] M. Elhemali, N. Gallagher, N. Gordon, J. Idziorek, R. Krog, C. Lazier, E. Mo, A. Mritunjai, S. Perianayagam, T. Rath, et al. "Amazon {DynamoDB}: A scalable, predictably performant, and fully managed {NoSQL} database service." In: *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 2022, pp. 1037–1048.

[Gba+24] B. Gbadamosi, L. Leonardi, T. Pulls, T. Høiland-Jørgensen, S. Ferlin-Reiter, S. Sorce, and A. Brunström. "The eBPF runtime in the linux kernel." In: *arXiv preprint arXiv:2410.00026* (2024).

[Gho+98] D. P. Ghormley, S. H. Rodrigues, D. Petrou, and T. E. Anderson. "SLIC: An Extensibility System for Commodity Operating Systems." In: *USENIX Annual Technical Conference*. Vol. 98. 1998.

[Gic+13] J. Giceva, T.-I. Salomie, A. Schüpbach, G. Alonso, and T. Roscoe. "COD: Database/Operating System Co-Design." In: *CIDR*. 2013.

[Gup+15] A. Gupta, D. Agarwal, D. Tan, J. Kulesza, R. Pathak, S. Stefani, and V. Srinivasan. "Amazon redshift and the case for simpler data warehouses." In: *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015, pp. 1917–1923.

[Gut+05] R. H. Guting, V. Almeida, D. Ansorge, T. Behr, Z. Ding, T. Hose, F. Hoffmann, M. Spiekermann, and U. Telle. "Secondo: An extensible dbms platform for research prototyping and teaching." In: *21st International Conference on Data Engineering (ICDE'05)*. IEEE. 2005, pp. 1115–1116.

[Ima18] T. Imada. "Mirageos unikernel with network acceleration for iot cloud environments." In: *Proceedings of the 2018 2nd International Conference on Cloud and Big Data Computing*. 2018, pp. 1–5.

[IOV] IOVisor Project. *BCC: BPF Compiler Collection*. `https://github.com/iovisor/bcc`. Accessed: 2025-08-26.

[Kim+] A. Kim, M. Slot, D. G. Andersen, and A. Pavlo. "Anarchy in the Database: A Survey and Evaluation of Database Management System Extensibility." In: ().

[Kue+21] S. Kuenzer, V.-A. Bădoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, Ş. Teodorescu, C. Răducanu, et al. "Unikraft: fast, specialized unikernels the easy way." In: *Proceedings of the Sixteenth European Conference on Computer Systems*. 2021, pp. 376–394.

[LD24] V. Leis and C. Dietrich. "Cloud-native database systems and unikernels: Reimagining os abstractions for modern hardware." In: *Proceedings of the VLDB Endowment* 17.8 (2024), pp. 2115–2122.

[Li+25] F. Li, X. Zhou, P. Cai, R. Zhang, G. Huang, and X. Liu. *Cloud Native Database: Principle and Practice*. Springer, 2025.

[Net] B. S. Networks. *uBPF*. `https://github.com/iovisor/ubpf`. Accessed: 2025-08-11.

[Pav16] R. Pavlicek. *Unikernels: Beyond Containers to the Next Generation of Cloud*. O'Reilly Media, 2016. ISBN: 9781491959244.

[Sch+86] P. Schwarz, W. Chang, J. C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. "Extensibility in the Starburst database system." In: *Proceedings on the 1986 international workshop on Object-oriented database systems*. 1986, pp. 85–92.

[Ski+21] A. Skiadopoulos, Q. Li, P. Kraft, K. Kaffes, D. Hong, S. Mathew, D. Bestor, M. Cafarella, V. Gadepally, G. Graefe, et al. "DBOS: A dbms-oriented operating system." In: (2021).

[SMM14] M. N. Sadiku, S. M. Musa, and O. D. Momoh. "Cloud computing: opportunities and challenges." In: *IEEE potentials* 33.1 (2014), pp. 34–36.

[SS25] S. Susnjara and I. Smalley. *What is cloud computing?* IBM. Feb. 2025.

[SS96] C. Small and M. I. Seltzer. "A Comparison of OS Extension Technologies." In: *USENIX Annual Technical Conference*. 1996, pp. 41–54.

[Sto81] M. Stonebraker. "Operating system support for database management." In: *Communications of the ACM* 24.7 (1981), pp. 412–418.

[Sysa] C. Systems. *OSv*. `https://osv.io/`. Online; accessed 2025-08-19.

[Sysb] C. Systems. *OSv*. `https://github.com/cloudius-systems/osv`. Accessed: 2025-08-11.

[Tra] Transaction Processing Performance Council. *TPC-H Version 2 and Version 3*. `https://www.tpc.org/tpch/`. Accessed: 2025-08-18.

[Ver+17] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao. "Amazon aurora: Design considerations for high throughput cloud-native relational databases." In: *Proceedings of the 2017 ACM International Conference on Management of Data*. 2017, pp. 1041–1052.

[Wan+23] J. Wang, T. Li, H. Song, X. Yang, W. Zhou, F. Li, B. Yan, Q. Wu, Y. Liang, C. Ying, et al. "Polardb-imci: A cloud-native htap database system at alibaba." In: *Proceedings of the ACM on Management of Data* 1.2 (2023), pp. 1–25.

[Zho+25] X. Zhou, V. Leis, J. Hu, X. Yu, and M. Stonebraker. "Practical db-os co-design with privileged kernel bypass." In: *Proceedings of the ACM on Management of Data* 3.1 (2025), pp. 1–27.