Linköping University | Department of Computer and Information Science
Master's thesis, 30 ECTS | Computer Science
2026 | LIU-IDA/LITH-EX-A--26/092--SE

Dynamic Software Updating Shared Libraries in a Multi-Core High-Availability System

– A cross comparison of multiple techniques

Dynamisk Programvaruuppdatering av Delade Bibliotek i ett Flerkärnigt Högtillgängligt System

Roberto Wilnerzon

Supervisor : Máté Földiák
Examiner : Filip Strömbäck

External supervisor : Stefan Sundkvist

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida `http://www.ep.liu.se/`.

Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: `http://www.ep.liu.se/`.

© Roberto Wilnerzon

Abstract

This thesis aims to design, implement, and cross-compare software capable of updating parts of a running system, without requiring a restart of the host application. Dynamic Software Updating (DSU) enables such runtime modifications. High-availability systems aim to remain continuously in service, which makes dynamic updates particularly valuable for applying changes with minimal service interruption in such systems. In academia, DSU has been explored in general-purpose computing where prior work largely focuses on abstract mechanisms rather than application-specific ones. In this work, we take an application-specific perspective, using DSU to update program functions within a running Radio Access Network (RAN) application.

The scientific novelty of this work lies in applying established DSU techniques to a uniquely constrained RAN application. Restructuring the application to support DSU introduced some pros and cons, prompting an investigation into how different DSU strategies affect downtime, performance, memory usage and system stability. By implementing four update solutions and evaluating them under conditions that resemble a worst-case packet transmission workload, the thesis quantifies the costs of enabling runtime updates in a high-availability RAN system.

Keywords: DSU, Shared libraries, Parallel Execution, Dynamic linking, High availability

Acknowledgments

I would like to express my sincere gratitude to everyone who supported me throughout the work on this thesis and for the valuable resources Ericsson has provided for this project.

My deepest appreciation goes to Stefan Sundkvist at Ericsson, my supervisor, for his guidance, expertise, and continuous support during the project. I am equally grateful to Máté Földiák, my academic supervisor, for his valuable feedback, insightful discussions, and patience throughout the thesis work. I would also like to thank Filip Strömbäck whose expertise in this research area and constructive input contributed to the quality of this thesis. Finally, I wish to thank my friends and family for their encouragement during this period.

Contents

Contents v

List of Figures vi

List of Tables vii

Abbreviations viii

1 Introduction 1
- 1.1 Motivation . . . 2
- 1.2 Aim . . . 2
- 1.3 Research Questions . . . 2
- 1.4 Delimitations . . . 3

2 Theory 4
- 2.1 The Conceptual Model . . . 4
- 2.2 Dynamic Linking Background . . . 5
- 2.3 Challenges . . . 8
- 2.4 Existing Techniques . . . 12
- 2.5 Alternative Techniques . . . 14

3 Method 16
- 3.1 Pre-study . . . 16
- 3.2 The DPDK Testbench . . . 16
- 3.3 DSU Solutions for Library Updating . . . 19
- 3.4 Benchmark Profiling . . . 28
- 3.5 Isolated Microbenchmarks . . . 31

4 Results 33
- 4.1 Execution Setup . . . 33
- 4.2 RQ1: Comparison Against Program Restart . . . 34
- 4.3 RQ2: Steady-State Indirection Overhead . . . 37
- 4.4 RQ3: Service Impact of DSU in a RAN System . . . 39

5 Discussion 41
- 5.1 Results . . . 41
- 5.2 Method . . . 43
- 5.3 The Work in a Wider Context . . . 44

6 Conclusion 45
- 6.1 Future Work . . . 46

Bibliography 47

List of Figures

1.1 Simplified illustration of a 5G Radio Access Network (RAN) allowing User Equipment (UE) to connect to the core network. . . . 2

2.1 Packet processing dispatcher pipeline model used in the application. . . . 5
2.2 Overview of a C program build. . . . 7
2.3 Overview of a shared library build. . . . 7
2.4 Overview of lazy binding with PLT. . . . 8
2.5 Three approaches to redirecting execution to a new module. . . . 9
2.6 Example of an update leading to a memory access to uninitialized data. . . . 10
2.7 A function call at the machine level. . . . 11
2.8 Timeline of a dynamic software update. . . . 12
2.9 Example of automatically profiled update points constrained by con-freeness [14]. 14

3.1 Testbench architecture. . . . 17
3.2 The interaction between the workers and the controller. . . . 18
3.3 Dynamic linking routine performed at runtime when a DSU is invoked. . . . 19
3.4 Controller and worker behavior in the QALL solution. . . . 21
3.5 Controller and worker behavior in the QMOD solution. . . . 23
3.6 Continuation 26
3.7 Controller and worker behavior in the FLAG solution. . . . 27
3.8 Example of packet-level ingress, enqueue, egress, and loss rates over time used for system-level performance analysis. . . . 30
3.9 Macro-expanded approach for accurate microbenchmarks. . . . 31

4.1 Downtime measurements aggregated in histograms for each solution. . . . 36
4.2 Latency distribution for DSU solutions. . . . 38
4.3 DPDK testbench output showing packet-level ingress, enqueue, egress, and loss rates. The dashed vertical line marks the point in time when the dynamic update is performed. . . . 40

List of Tables

2.1 Overview of a C program build and execution pipeline on Linux. 6

4.1 Cross comparison of the prototyped DSU solutions. 35

Abbreviations

ABI Application Binary Interface

API Application Programming Interface

CPU Central Processing Unit

DPDK Data Plane Development Kit

DSU Dynamic Software Update

DU Distributed Unit

ELF Executable and Linkable Format

PDCP Packet Data Convergence Protocol

PIC Position-Independent Code

RAN Radio Access Network

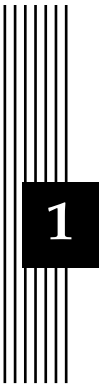
RAM Random Access Memory

RX Receiver

TX Transmitter

UE User Equipment

5G Fifth Generation of Cellular Network Technology



1 Introduction

Imagine a system that millions of people rely on every second, like the network that keeps phones connected to the Internet. Software updates for such systems are regularly needed, but every update risks interrupting the seamless experience users take for granted. In environments such as 5G networks, where connectivity is expected to be continuous, even a brief disruption can be unacceptable.

Updates are warranted to fix bugs, add new functionality, and comply with evolving standards. Yet they are a disruptive operation, especially in systems with high availability requirements. A conventional update involves a complete restart of the system, even if only a small part of the system is to be updated. This raises the question of whether such disruptions can be avoided. In this thesis, we explore how functions of an in-service system can be updated at runtime.

Dynamic Software Updating (DSU) is a technique for updating a program at runtime without restarting it. Conventional updates occur at build time, when new code is incorporated to the program before execution. In contrast, integrating new code into a process presents fundamentally different challenges. The update must be introduced into a predefined process address space, full of interacting components, without causing disruption. Some key concerns are correct post-update behavior, not unloading active code, managing dependencies, and resolving dynamic symbols.

Many programs use libraries, either statically or dynamically linked into the program. Dynamically linked libraries, subsequently referred to as shared libraries, are not embedded directly into the executable, but instead dynamically linked in at runtime. This raises the question of whether we can replace one version of a shared library with another at runtime. The modular design of shared libraries makes this seem feasible, as they can be rebuilt and loaded into memory independently of the main program. Building and loading shared libraries are both well-supported by existing mechanisms and allow the program to interact through a well-defined external interface.

Taking these properties into account, a system can be restructured for DSU by moving the functions intended for runtime updates from the executable into shared libraries. However, this restructuring introduces both benefits and drawbacks. It enables DSU, but it also degrades the original system through performance and memory overheads. These trade-offs prompted the investigation presented in this thesis.

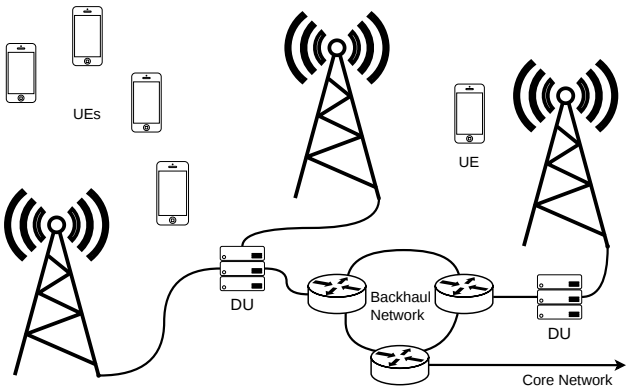


Figure 1.1: Simplified illustration of a 5G Radio Access Network (RAN) allowing User Equipment (UE) to connect to the core network.

1.1 Motivation

Despite extensive research in Dynamic Software Updating (DSU) [1, 2], many state-of-the-art software systems that could benefit from in-service updates still do not employ these techniques to improve their availability. High availability systems often use countermeasures such as redundant hardware or replicated services to hide downtime, but these solutions introduce significant operational and financial costs. In contrast DSU offers an alternative approach by enabling updates without service interruption. It has the potential to reduce infrastructure costs, enable faster patching, and simplify deployment.

1.2 Aim

The aim of this thesis is to implement and evaluate DSU techniques in the context of a planned update of shared libraries in a multi-core system. Specifically, this work seeks to achieve near-zero downtime when updating program functions of an in-service Radio Access Network application, characterized by a 99.999% availability requirement, allowing for about five minutes of downtime per year. Our aim is to investigate DSU techniques by prototyping some solutions based on established principles, identifying practical trade-offs, and presenting concrete metrics through a cross-comparison.

Figure 1.1 shows the 5G network, consisting of radio base stations and other network elements. This represents the system to which we apply DSU, with particular focus on the Distributed Units (DUs), which are servers responsible for key RAN functions and the targets for applying software updates. However, in this thesis a simulated environment that mimics the RAN will be used, further described in Section 3.2.

1.3 Research Questions

RQ1 How do different DSU approaches compare to a program restart in terms of downtime, memory overhead, and runtime performance?

RQ2 During steady-state operation, what performance degradation results from using indirect function calls instead of direct function calls?

RQ3 How do DSU mechanisms influence the service of a RAN-like system? Can updates be applied while the system is actively serving traffic?

1.4 Delimitations

This section outlines the scope boundaries imposed on this work. Rather than pursuing a general-purpose DSU solution, a number of limitations were adopted to focus the thesis on tractable challenges and the development of multiple prototypes.

Most notably, *no generic updates to the main program* will be done. Dynamic updates in this context only focus on changing parts of the program that reside in shared libraries, rather than supporting larger updates of the program that is using those libraries. This simplifies control-flow migration, since execution can simply continue as usual after an update without concerns about pending returns having invalid return addresses.

Generally, programs that use libraries do not depend on the version-specific details of the exported functions, as libraries tend to be implemented such that the observable behavior remains consistent across different versions, even if the internal implementation differs. If the interface between program and shared library remains consistent, existing call sites in the program are stable and can continue to invoke updated functions without requiring modifications. Thus, this thesis assumes *call-site compatibility* across shared library versions, meaning that function signatures and calling conventions between program and shared library must remain unchanged between different versions. Additionally, *no internal library state will be preserved* between different versions.

The system performs *no immediate updates*, meaning that it does not modify any function that is executing at the moment an update is requested. Instead, any function that is already running is allowed to continue until it naturally completes, even if it is long-running or takes substantial time to complete. In practice, updates can be applied even while a function is running, with the update taking effect the next time that function is invoked.

Together, these constraints reduce the challenge of control-flow migration to a matter of code pointer redirection. This is sufficient to demonstrate the core feasibility of dynamic updates in shared libraries, without requiring the system to handle the far more complex challenges typically associated with DSU. We acknowledge that these constraints inherently restrict the kinds of updates we can support, as changes that modify data layouts or introduce new interfaces fall outside our scope and would require more intrusive update strategies.

Rather than attempting to build a fully general DSU framework, this work establishes minimum viable solutions that can be used to evaluate whether DSU is practically useful in this setting. The broader flexibility required for updating arbitrary program components is left for future work. Nevertheless, the ability to modify the internal implementation of functions remains valuable.

2 Theory

This chapter establishes the theoretical basis necessary to understand the working context of this thesis and the design decisions behind dynamic updates in general. A considerable amount of literature on Dynamic Software Updates (DSU) has accumulated over the past several decades [1, 2], with works dating back to 1976 [3]. In this thesis a selection of that literature is reviewed, focusing on core issues, established techniques, and alternative approaches within the field of DSU. Section 2.1 describes the real-world application model, that we try to mimic with simulations. Section 2.2 reviews how a program is built and executed. Section 2.3 introduces key challenges. Section 2.4 presents our predecessors work, while Section 2.5 outlines industry technologies and related approaches.

2.1 The Conceptual Model

The overarching Radio Access Network (RAN) pipeline, seen in Figure 2.1, consists of multiple worker processes, called *workers*, with high demands on performance and resilience. The workers are identical, but input-dependent branching causes them to perform different work. Each worker executes a set of library functions to handle tasks such as sequence numbering (assigning per-packet numbers for ordering and loss detection), data compression, and encryption. The tasks they are responsible for are conceptually similar to tasks performed by Packet Data Convergence Protocol (PDCP) entities as specified in 3GPPTS 38.323 [4].

Received packets are passed to a dispatcher that classifies each packet and routes it into different event-driven *flows*. In practice, these flows correspond to different processing paths. For example when packets are sent uplink (UE→RAN→Core Network) or inversely downlink (Core Network→RAN→UE). When a packet arrives, the dispatcher triggers an appropriate handler and places the packet into a flow-specific event queue. Workers then dequeue packets in bursts from the available queues, allowing multiple workers to process a flow while still respecting requirements of each flow. Flow-aware logic ensures that packets belonging to the same flow are handled appropriately. Some flows are processed strictly in order by a single worker, others are distributed across workers but reassembled in order afterward, and stateless flows can be processed fully in parallel. This architecture maximizes throughput while preserving correctness for ordered protocols like TCP.

The pipeline achieves multi-core parallelism through multiple single-threaded workers, each on a dedicated core. Queues are made available to workers as needed so that each

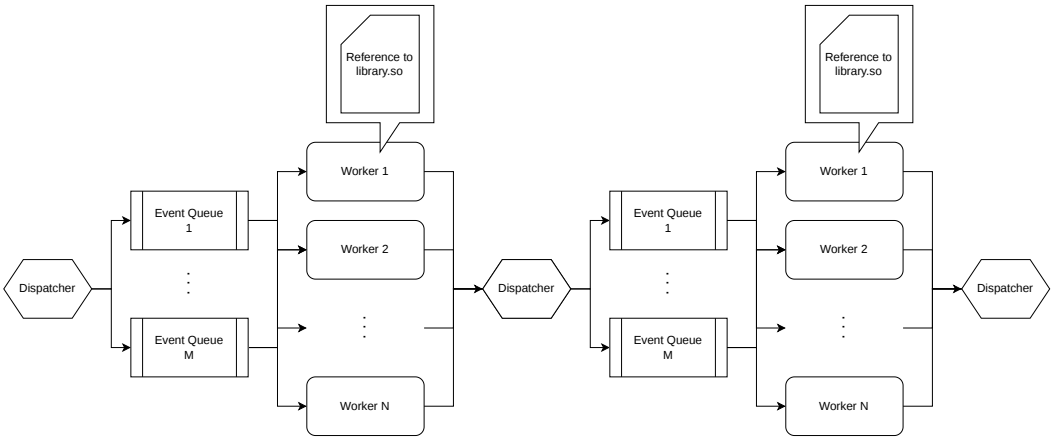


Figure 2.1: Packet processing dispatcher pipeline model used in the application.

worker can make progress on the overall workload. Although the specifics of the dispatcher are not central to this thesis. Aside from a tiny shared memory region, the workers are completely modular and function independently of each other. *Downtime* in this context is therefore defined as the update-induced pause measured from the time a worker stops processing until it resumes operation.

From a systems perspective, the RAN encompasses both radio sites and the associated front-, mid-, and backhaul connections, meaning workers live at a variety of locations. The pipeline functions as a component within this larger application, operating wherever packet processing is required across the network.

2.2 Dynamic Linking Background

Software development begins with source code, the human-readable representation of a program. Source code expresses the program's logic, structure, and behavior, but it cannot be executed directly by hardware. In this section, we use C programs as an example and focus specifically on native software environments. The native compilation pipeline used by C is broadly representative of how compiled languages are processed on Linux systems, making it a suitable model for discussing low-level execution and update mechanisms.

Table 2.1 outlines how a program is progressively built into a machine-ready form. It summarizes the four build-time stages that convert source code into an executable program, followed by two stages for program execution handled by the operating system. During compilation, the toolchain performs tasks such as rewriting code into more efficient forms (applies optimization), deciding how to use CPU registers (register allocation), and recording symbol information for later. Because these optimizations and register allocations are done at build time, challenges may arise when trying to modify the program at runtime, as further discussed in Section 2.3.

The build-time portion of the pipeline is carried out by the preprocessor, compiler, assembler, and linker. Together, these components transform a program through a sequence of representations: source code (.c, .h) $\rightarrow$ intermediate (.i) $\rightarrow$ assembly (.s) $\rightarrow$ object file (.o) $\rightarrow$ a binary. In this context, the outputted binary of the build-time stages may be a standalone executable or a shared library (.so).

Once an executable is launched it becomes a process through two mechanisms. First, the loader, running in kernel space, loads the executable into memory, sets up the address space of the process, loads the dynamic linker, and transfers control to the dynamic linker's entry point. Thereafter, the dynamic linker, running in user space, can load shared libraries into

Table 2.1: Overview of a C program build and execution pipeline on Linux.

Stage	Input	Output	Operation
Preprocessing	main.c + main.h	main.i	Macro expansion, header inclusion, comment removal
Compiling	main.i	main.s	Semantic analysis, apply optimizations, register allocation
Assembling	main.s	main.o	Build a symbol table, record unsolved symbols (a.k.a. relocation entries)
Linking	main.o + dependencies	prog	Resolve symbols, combine object files, apply more optimizations
Loading	prog	Process shell	Load the executable into memory and transfer control to dynamic linker
Dynamic linking	Process shell + lib.so	Process	Load all required shared libraries and perform relocations

memory, resolve remaining relocation entries, and transfer control to the program's entry point [5]. The process remains a process shell until the dynamic linker finishes.

In this context, a process shell refers to the partially constructed process created by the loader after mapping the executable and the dynamic linker into memory. At this stage, shared libraries have not yet been loaded and relocations have not been applied. The process exists and has an allocated address space, but it is not yet fully linked or ready to execute user code. The dynamic linker finalizes this shell by loading dependencies, resolving symbols, and finalizing the runtime environment.

Note that linking performed by the linker (`ld`) at build time is different from dynamic linking performed by the dynamic linker (`ld-linux.so`) at runtime. It is the dynamic linker that enables DSU of shared libraries in a running process.

Symbols and relocations

A symbol is a named identifier that represents something in a program, usually a function or global variable. Symbols enable the system to keep track of what exists and where it is located. Before a program is assigned real memory addresses, symbols act as placeholders that let separately compiled pieces of code refer to each other. The symbolic references are later replaced with actual addresses [6]. *Symbol resolution* is done by finding which object defines a symbol. Performing a *relocation* is done by binding a symbolic reference to its actual memory address. This binding can occur in two ways:

- **Lazy binding**, which defers symbol resolution until a symbol is first used. Modern systems typically adopt this behavior to avoid unnecessary work relocating unused symbols, since programs may not actually call all functions provided by a library object.
- **Immediate binding**, where symbols are resolved immediately. When a program then runs, all symbols required during execution will already be resolved and available.

Figure 2.2 illustrates the differing scopes visible to the assembler and the linker. It also shows that a shared library can be given to the linker at build time to provide the system with metadata to resolve symbols from that library and generate relocation entries. Build-time symbol work is largely a bookkeeping process. The assembler identifies symbol definitions and references, embeds a symbol table into the object file, and generates relocation entries for unsolved symbols whose addresses cannot be determined at compile time. The linker then combines input files into a binary, performs symbol resolution and relocations across all input files, and defers any unresolved symbols to be fixed by the dynamic linker when shared libraries are actually loaded into memory [5].

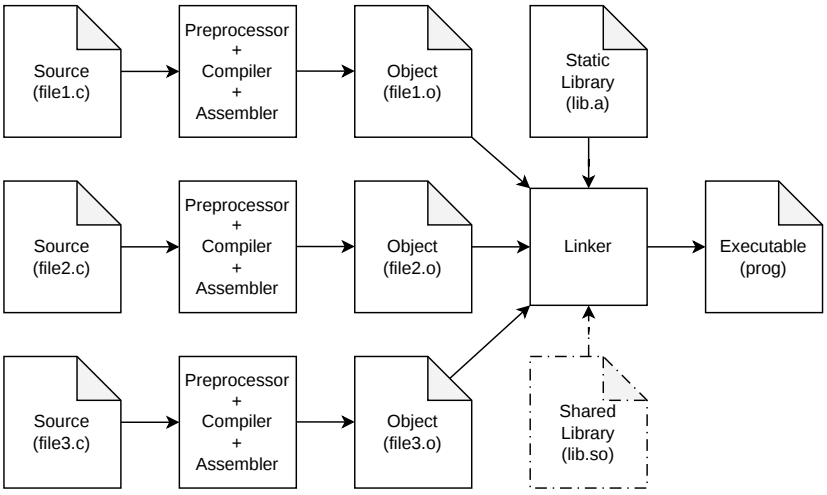


Figure 2.2: Overview of a C program build.

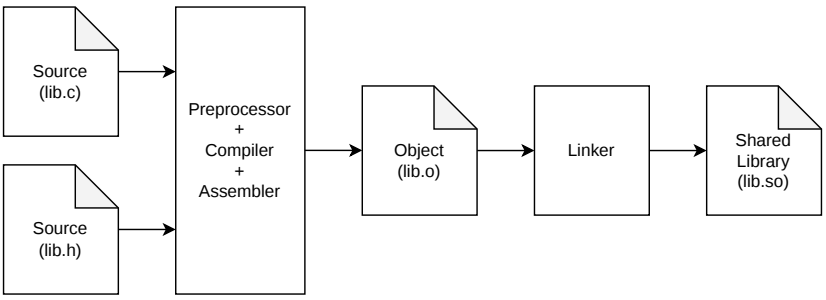


Figure 2.3: Overview of a shared library build.

Figure 2.3 illustrates that an object file must be processed by the linker to produce a valid shared object file, even when it is not combining additional dependencies. The linker finalizes the output by producing an Executable and Linkable Format (ELF) file, applying position-independent code (PIC) requirements [7, p. 42], and generating some metadata. As shared libraries are compiled as PIC, they may be loaded at arbitrary memory locations.

Dynamic linking with GOT/PLT

Because it is impossible to predict where a binary and its libraries will be loaded[1], the dynamic linker is used to resolve the addresses of externally defined functions. This resolution relies on pointers, i.e. variables that store absolute memory addresses. A function pointer is simply a pointer whose value is the address of a function's entry point, allowing the program to call that function indirectly through the stored address.

In ELF-based systems a Global Offset Table (GOT) and Procedure Linkage Table (PLT) is used to support dynamic linking. The GOT acts as a centralized indirection layer that stores addresses of dynamic symbols, i.e. pointers. A single GOT is constructed by the linker during the final link step of building an ELF executable or shared library. The table is instantiated once and can not be extended at runtime. The dynamic linker populates the GOT with function addresses, global variables, and other relocated pointers as it processes the program's relocation entries. The executable and each shared library it uses maintain their own local GOT, so that each module can record pointers to its own dependencies.

[1] in part due to address space layout randomization

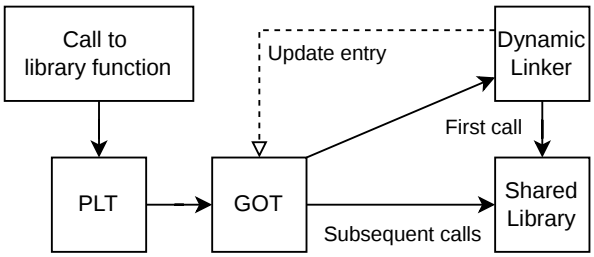


Figure 2.4: Overview of lazy binding with PLT.

The PLT is another indirection layer, used specifically for invoking external functions provided by shared libraries. Each entry in the table consists of a small code stub corresponding to a single external function. The compiler and linker rewrite external function calls so that call sites target these PLT stubs rather than the unresolved functions, providing a stable call interface. The PLT is also used to support lazy binding. Figure 2.4 illustrates this process; On the first invocation of a function, the PLT stub reads the GOT entry, sees it is unresolved, and jumps into the dynamic linker triggering lazy resolution. Subsequent calls bypass the dynamic linker and perform an indirect jump to the target.

Provided that a binary contains the relevant metadata of external symbol dependencies, the initial GOT and PLT can be constructed. This ensures that the runtime environment has a stable view of all external symbol dependencies at startup. Because both the tables are fixed size, no new entries can be added later. Technically, function pointer swaps could be done via the GOT at runtime, but doing so complicates the unloading of shared libraries.

2.3 Challenges

This section outlines the key challenges associated with dynamic updates, with a particular focus on systems-programming in C.

Update flexibility

Update flexibility is the degree to which dynamic update mechanisms can accommodate various changes while ensuring that the post-update behavior remains correct. An Application Programming Interface (API) consists of constants, data structures, functions, and other language-level constructs that a library exposes to enable external code to interact with its functionality. For a library, changes to exported functions or data structures create a problem where existing referrers continue to access its items under outdated assumptions about signatures or memory layouts. Thus, updating only the library without simultaneously updating the external components that depend on it can create a mismatch between different versions. Another mismatch arises if an API update requires changing the low-level calling convention. For example, by requiring additional arguments being passed in a function call. In such cases, both the updated API and all external components invoking it must agree on the same calling convention. Otherwise, callers will supply arguments incorrectly which ultimately leads to undefined behavior or possibly program crashes.

A practical Dynamic Software Updating (DSU) tool would ideally allow developers to change anything, including data structures and function signatures between program versions. Then after an update, assert the validity of the runtime state which includes checking that call sites remain compatible with updated return types, number of arguments, and arguments types. Though, as noted in Section 1.4, our approach avoids this complexity by limiting the range of updates and assuming call-site compatibility.

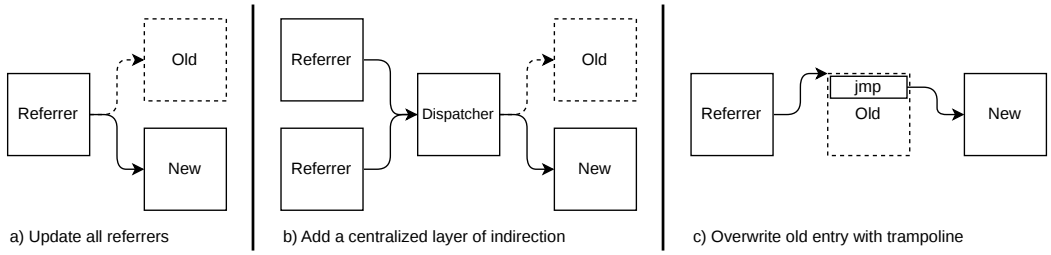


Figure 2.5: Three approaches to redirecting execution to a new module.

Performance overhead

Adapting a system to enable DSU capabilities may introduce penalties such as extra work performed continuously throughout execution, which we denote *performance overhead*. This extra work degrades performance compared to a equivalent non-DSU system and can negatively impact throughput. Performance overhead may result from update-related mechanisms such as conditional update checks, synchronization between threads, loading function pointers before use, and bookkeeping tasks. Overhead is also introduced by restructuring the program for DSU, particularly when relocating functionality into shared libraries.

Performance overhead arises when one shifts from direct function calls to indirect function calls through dynamic function pointers. To maintain clarity, this thesis distinguishes *calling* as a direct function call and *invoking* as an indirect function call performed through a function pointer. This shift not only adds an extra pointer-lookup step but also prevents compiler optimizations like inlining, speculative optimization, and inter-procedural analysis [6]. These lost optimizations are compiler-level penalties, as the compiler can no longer reason about the target and must generate more conservative code. Pipeline stalls may also be introduced because the CPU must load the function-pointer value before it can determine the next program counter, leaving empty slots in the CPU pipeline [8].

Adding a layer of *indirection* introduce one additional control-flow jump before reaching the actual target function, which adds latency to each function call and can cause micro-architectural penalties. Because the CPU resolves these jumps using its branch predictor, incorrect instruction paths may be predicted when the jump target changes. Each misprediction forces the processor pipeline to be flushed and subsequently refilled. In addition, indirection may cause instruction-cache misses when the target resides in a cold or distant code region, requiring the CPU to fetch instructions from slower memory instead of the cache [8]. Collectively, all of these effects ultimately lead to additional clock-cycle costs at runtime each time a function is invoked.

Figure 2.5 displays three common approaches used in DSU systems to redirect references from an old module to its updated version. Together, these approaches represent the main design space for redirecting execution during dynamic updates, each offering different trade-offs in performance, complexity, and update flexibility.

a) Updates all referrers directly by locating every call site or stored reference to the old module and rewriting each to target the new one.

b) Introduces a centralized indirection layer at compilation, such as a global function pointer, through which all invocations are routed. Updates then require modifying only this single indirection point instead of the entire program.

c) Applies binary patching at runtime by overwriting the entry point of the old function with a trampoline that redirects execution to the new module. The trampoline is a minimal sequence of instructions, typically a single `jmp`.

Old version

```
2 void init() {
3     pos.y = 2;
4 }
5 void foo() {
6     pos.x = 1;
7     pos.x += 1;
8 }
```

New version

```
2 void init() {
3     pos.x = 1;
4     pos.y = 2;
5 }
6 void foo() {
7     pos.x += 1;
8 }
```

```
1 struct T { int x; int y; } pos;
9 ...
10 int main() {
11     init(); // Initializes only y
12     // Update occurs here
13     foo(); // Assumes x and y are initialized
14 }
```

Figure 2.6: Example of an update leading to a memory access to uninitialized data.

Memory overhead

Memory overhead arises for example by adding code or new APIs to support the DSU, allowing multiple versions of the updated code to co-exist, or in the case of Figure 2.5 c) where old deprecated function bodies stay in memory after their final use and cannot be unloaded. Shared libraries are loaded into RAM, and typically an old library should be unloaded after switching to a new version. Managing this process of performing the eventual cleanup adds yet another update-related cost.

Proper memory management is necessary when reclaiming unused objects in a multi-threaded system. Improper handling can lead to multiple deallocations of the same memory region and dangling pointers with use-after-free errors. Further, if obsolete code remain in RAM after an update, the application's memory usage increases unnecessarily. Seifzadeh et al. [1] claim that several existing DSU tools either ignore code cleanup or do not measure the amount of RAM they use.

Control flow migration

Control flow refers to the order in which the instructions of a program are executed. A valid DSU solution must determine both where updates are allowed to occur in the program and how to resume execution after an update. Figure 2.6 shows a scenario when initialization is moved earlier in control flow, making the in-flight operation access state that was supposed to be initialized previously but was not. Note that neither `init()` or `foo()` needs to be active when the update takes place for this update to be problematic.

At the machine level, program execution is driven by the Program Counter (PC), a special register that holds the address of the next instruction to execute. Each instruction advances the PC to the next instruction, except for control-flow instructions such as the x86 `call`, `jmp`, and `ret`, that instead redirect it. Calling a function pushes a new stack frame to the call stack, containing a return address, the arguments, and local variables. The PC is then set to the entry point of the callee and the function becomes *active* and remains so until its frame is popped from the stack. Note that the function itself may issue further calls, each pushing additional frames onto the stack. When a function completes, a return instruction restores the PC from the saved return address and pops the corresponding frame, resuming execution exactly at the return address This strict discipline ensures that control flow is well defined and that the runtime state reflects the nested structure of active function calls [8].

```
1 int foo(int a, int b) { return a & b }
2 int main() { int x = foo(1,2); }
```

Pseudo x86_64 ABI

```
00...00400550 <main>:
400550: be 01 00 00 00      mov    $0x1,%esi
400555: bf 02 00 00 00      mov    $0x2,%edi
40055a: e8 e0 ff ff ff      call   400554 <foo>
40055f: 89 44 24 XX         mov    %eax,[rsp + x_offset]
400563: c3                  ret

00...00400564 <foo>:
400564: 89 f8               mov    %esi,%eax
400566: 21 f0               and    %edi,%eax
400568: c3                  ret
```

Figure 2.7: A function call at the machine level.

An Application Binary Interface (ABI) defines how compiled code interact at the machine level, including data layout, symbol naming, register usage, and linking rules. A calling convention is the part of the ABI that specifies how function calls are performed, including how arguments are passed (in registers or on the stack), how return values are delivered, and which registers must be preserved across calls. In Figure 2.7, the compiler has generated machine code where `reg1` and `reg2` are used to pass arguments, and `eax` is used to return the result. The program is reliant on this contract being stable. If `foo` were changed to require additional arguments, the existing call site in `main` would have no space to accommodate the extra mov instructions, and the caller's stack frame and register usage would no longer match the callee's expectations. This is one reason why call-site compatibility is critical and why changing it is complicated.

A library can safely be updated if its rebuilt from scratch and interacted with only through well-defined entry points, handled by the dynamic loader [5]. But, altering the main program is significantly more complex, considering that compiler optimizations may reorder instructions in ways that make the program highly sensitive to layout changes, and that in-flight stack frames created before an update may contain return addresses invalidated by the update. Redirecting control flow back into those addresses would immediately corrupt execution. Which is why, as stated in Section 1.4, no generic updates of the main program will be done; updates are restricted to dynamically loaded libraries, ensuring that all active frames remain valid and that control-flow integrity is preserved.

Dynamic updates can be categorized broadly into *delayed* and *immediate* update types [1]. These differ in when and how they interact with the program's control flow. Delayed updates postpone the update until the program is in a tractable state, reducing the risk that in-flight operations references code or state that the update invalidates. The update mechanisms may begin executing immediately, but the observable effects of the update are delayed. For example, a modified function is not replaced mid-execution but only on its next invocation.

Immediate updates on the other hand are applied as soon as the update is requested, forcing changes to take effect even if the affected objects are in use. This makes them have comparatively less *update latency*, which refers to the time interval between the moment an update is requested and the moment the update begins to take effect (see Figure 2.8).

Formal proofs of correct post-update behavior vary significantly between these categories. Such proofs are often far simpler for delayed update types, where the system can rely on well-defined control-flow boundaries. In contrast, proving correctness for immediate updates is considerably more complex and may not be feasible for all tools [9].

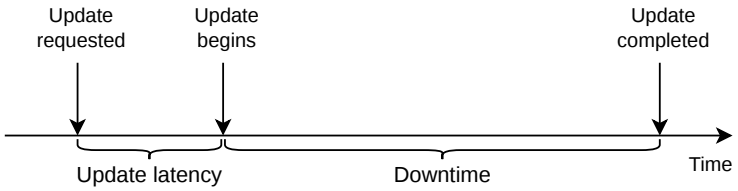


Figure 2.8: Timeline of a dynamic software update.

Multi-core support

Supporting dynamic updates in multi-core, multi-threaded programs requires careful coordination of concurrency and shared memory. When multiple threads execute in parallel, an update must ensure that the program remains consistent across versions. This typically involves reasoning about application modularity, shared state, and threads.

Modularity determines how responsibilities are separated, how predictable component boundaries are, and how easily an update can be confined to a well-defined region of the program. Modular systems reduce the risk that an update to one component accidentally affects another that is executing simultaneously. In this thesis, updates are applied to shared libraries, which are modular components with clear bounds.

An atomic operation is indivisible and runs without interruption. In multi-threaded systems, atomicity is necessary to correctly read or write to shared data, since it prevents other threads from observing intermediate or inconsistent states. Within the context of DSU, atomicity ensures that updates appear to take effect simultaneously with respect to all threads.

To handle shared state in multi-threaded systems some tools [10, 11] adopt conservative strategies such as thread blocking to ensure updates are done correctly. This approach makes all but one thread sleep at update points. The remaining thread then performs the update while ensuring that no other thread is executing modified code. This approach simplifies correctness reasoning but restricts when updates may occur, which can increase update latency.

Others explore more flexible mechanisms. Polus [12] introduces a thread-tracking model that monitors which functions each thread is executing. By maintaining per-thread metadata Polus can determine when it is safe to update individual functions without requiring thread-blocking. This allows updates to proceed with less disruption, but at the cost of additional performance overhead and more bookkeeping.

2.4 Existing Techniques

The broad model

One approach to updates is to apply a broad update, changing both the definitions and every referrer at the same time. This is generally sensible, since moving from one program version to another requires all dependent files to be updated together, so types and interfaces remain consistent. Large updates like this are commonly applied by a transformer function [11, 13, 14] or similarly a transformer class [15, 16, 17]. These components specify how runtime state is migrated and may initialize new data with default values, using automatically generated transformations when possible and programmer-supplied logic when structural changes demand it.

However, doing this is only possible if all referrers can be changed at the same time. Additionally, it forces potentially large synchronous updates, which can cause long pauses and violate availability requirements.

Quiescent-based transitions

Quiescence refers to a state of inactivity. A function is active if there exists at least one stack frame for that function on the call stack of any thread, the function is *quiescent* if it is not active. Quiescence provides a well-defined moment to apply a dynamic update, where no thread is executing the code being replaced. Tools like Ksplice [18], gDSU [19], and many others leverage quiescence and thread synchronizations as fundamental safety features during transitions. Complex data migration is not needed for functions in a quiescent state, as no execution state or local variables are on the stack. This simplifies both applying updates and reasoning about post-update correctness.

As quiescence is a strict constraint, difficulties arise when attempting to update functions that never naturally leave the stack [20]. Such cases include long-running loops or state machines that execute indefinitely thereby never reaching quiescence. Admittedly, attempts have been made to mitigate this problem. Ginseng [13] uses *loop extraction*, involving a transformation that turns loops into callable functions, thus enabling loop logic to be updated on the next iteration. Although, updates can still take a long time as truly long-running functions cannot be updated before a quiescent state is reached.

The incremental model

Another approach is to break down a larger update into incremental steps. This approach requires old and new code to co-exist for a period of time, since the system can choose to update individual parts of the system to their new versions while others remain on the old version [21, 12]. This allows the system to transition invocations to the new version incrementally, enabling partially applied updates without requiring the entire program to switch versions at once.

Stub-based transitions

Stub-based transitions are used to manage calls from existing, not-yet-updated callers to an updated function. In this context, a stub function is an indirection layer that matches the signature of the function being updated. Existing referrers can then be redirected to invoke the stub without modification. During a transition, stub functions are interposed between existing referrers and updated definitions to perform translations, preserve type compatibility, and control the transition [14, 21]. By forwarding arguments to either the old or the new definition and then returning values to the referrer a stub function can effectively act as an intermediary. To avoid performance overhead, a DSU system can let the stub update its referrers to point directly to the new definition if the transitional work is completed, thereby removing the indirection [21].

Function pointer swaps

Function pointer swaps (also known as hot swapping) redirect execution from an old function to a new one by updating a function pointer. Ginseng [11, 13], Hicks and Nettles [21], PSU [22], and many others transform direct function calls into invocations using global function pointers. During runtime, function calls are resolved using these global function pointers. Updating the program then becomes a matter of swapping these pointers to reference the new implementation.

This type of indirection layer differs from stub-based indirection because this is introduced at compile time rather than inserted dynamically at runtime. Instead of adding a stub function to handle control flow migration and type changes, the system may use a lightweight global function pointer that all invocations pass through, allowing updates to be applied simply by redirecting this centralized pointer like in Figure 2.5 b).

```
1 typedef struct {int x; int y;} pair;
2
3 void foo(int* x) {
4 //  {} No Dependencies
5     *x=1;
6 //  {}
7 }
8
9 int main() {
10 //  {}
11     pair P = {3,2};
12 //  {P,foo} Block updates
13     foo(&P.x);
14 //  {P} Can update foo here
15     P = {0,2};
16 //  {} Can update both here
17 }
```

Figure 2.9: Example of automatically profiled update points constrained by con-freeness [14].

Update points

In this method, the program relies on reaching *update points* before modifications can be made. An update point is a predefined location in the program where update logic is inserted. Update points can either be specified by a developer or automatically by a tool. These points typically specify where long-running functions can be updated during execution, eliminating the need for structural refactoring such as loop extraction. However, using them requires foresight either through annotations or automatic profiling. Figure 2.9 illustrates update points found automatically by analyzing variable dependencies. The example demonstrates a simple form of con-freeness [14], meaning that updates are only permitted at program locations where no live variables depend on the objects being updated.

Tools like Ginseng [13], UpStare [10], and Kitsune [23] leverage the developer's capability to assign update points instead, and in an empirical study Hayden et al. [24] found manually placed update points to be more effective than automatic tools detecting locations constrained by quiescence or con-freeness.

2.5 Alternative Techniques

Several alternative techniques address the challenges of dynamic software updating through fundamentally different ways. These techniques do not directly translate to low-level C programs, but they demonstrate alternative design spaces for achieving low-downtime dynamic updates in similar environments.

Industry technology like Erlang [25] already provide some built-in support for hot code swapping by structuring applications around isolated processes that communicate via message passing. This design avoids many shared state problems, which greatly simplifies dynamic updates and reduces the risk of inconsistent program state during transitions.

Native code (like C) executes without supervision, so DSU must respect strict constraints such as ABI compatibility and cannot rely on runtime assistance for state management. *Supervised* environments (like Java and Erlang) execute under a managed runtime that provides services such as garbage collection, type checking, and controlled memory access, which mitigates many challenges faced by native DSU systems. More comprehensive supervised DSU systems can take advantage of Just-In-Time (JIT) compilation [15, 16, 22]. In such systems, the

adaptive JIT compiler can regenerate machine code whenever an updated function is called, ensuring that subsequent calls use the new version.

Validation with MVE

Multi-version execution (MVE) is an approach that improves availability and confidence in post-update correctness, by executing parallel instances of a program and comparing their behavior. MVE systems ensure that each instance gets the same inputs and checks that the produced outputs are equivalent. Tools like MX [26] and MVEdsua [27] let one instance act as the leader interacting with the operating system, while the rest follow its execution by replaying system calls and validating that their behavior matches. Divergences can indicate bugs in the updated code, errors in the DSU transformation, or behavioral changes. Dynamic updates are applied on a forked process, causing memory overhead, while the leader continues serving requests, to eliminate downtime. If a follower later diverges unexpectedly, the system can simply discard it and continue running the leader. Expected divergences, like new features, can be tolerated by adding rules and exceptions in the MVE system.

Stack rewriting

By rewriting return addresses and reconstructing stack frames, systems can migrate both control flow and local state across versions, effectively eliminating update latency. ActiveDSU [9] and UpStare [10] rewrite the call stack directly to execute transitions, allowing for updates to be carried out instantly with no synchronization. However, this comes at the cost of increased complexity since stack layouts differ across architectures and compilers, correctness is difficult to guarantee, and subtle errors can easily lead to crashes or corrupted program state.

Binary patching

Binary patching (also known as hot patching) works by rewriting instructions to redirect control flow and inserting trampolines at runtime. Binary patching modifies the loaded program's machine code directly. The mechanisms can add DSU capabilities onto existing programs without prior preparation. This enables updates to legacy programs that were not designed with DSU in mind [12, 18]. However, this flexibility comes with substantial challenges when applying an update. Many systems disallow updates to legacy code unless the program has been explicitly prepared for DSU, which often require significant refactoring.

3 Method

This chapter presents the methods used in this study, explaining how the work was carried out and establishing the foundation for interpreting the results. Different DSU solutions were implemented and here we present the experimental setup used to study the behavior of these. Section 3.2 describes how the conceptual model was simulated using a Data Plane Development Kit (DPDK) packet transferring testbench. In Section 3.3, our implementations are defined. In Section 3.4, the methodology for taking measurements in a realistic environment are described. In Section 3.5, we outline how microbenchmarks were taken.

3.1 Pre-study

The first part of this thesis involved an investigation of current research in dynamic software updating (DSU). This pre-study established the foundational knowledge and understanding required to carry out the subsequent implementation and evaluation. Prior work has explored a wide spectrum of techniques, ranging from low-level function-pointer swaps [13, 22] to state transformation frameworks at the virtual machine level [16]. These studies provided insight into the trade-offs between update flexibility, control flow complexity, and DSU-related overhead.

The pre-study also examined how DSU systems can be evaluated. Existing research commonly measures update latency, runtime overheads, and update success rates under varying workloads. Many systems also measure post-update correctness through sophisticated evaluation strategies such as controlled fault-injection, regression testing, or by verifying that updates don't break logical rules. In this thesis, evaluation is limited to checking that the program behaves as expected and that the intended outputs are produced.

3.2 The DPDK Testbench

Data Plane Development Kit (DPDK) [28] is a framework that enables reproducible packet-flow simulations for networking experiments. In this study, it is used to simulate the conceptual model and generate controlled packet workloads for evaluating DSU behavior. In Section 2.1 the conceptual model was described, which this testbench aims to simulate to obtain measurements in a realistic environment. To mimic the conceptual model while enabling controlled experimentation, the simulated system is structured around a dedicated

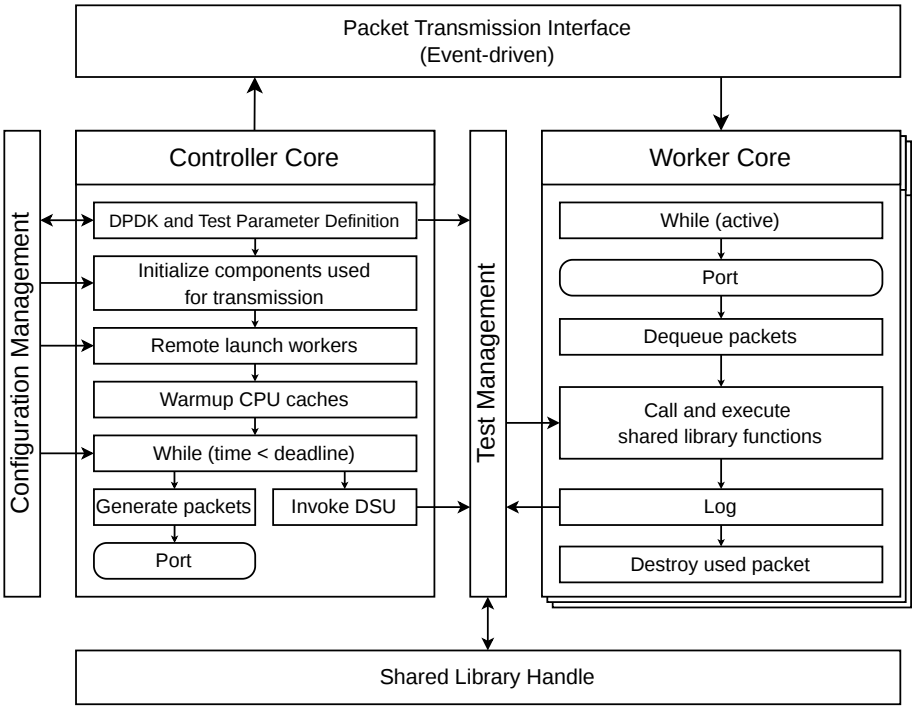


Figure 3.1: Testbench architecture.

CPU core, referred to as the controller, which is responsible for generating artificial packets and dispatching them into flows. These packets are then served to the workers via a packet transmission interface, which operates independently of the DSU subsystem and is therefore not central to the mechanisms under study. Each worker listens for events on its respective port, retrieving packets in bursts as they become available for processing.

The controller core manages the worker cores and is able to remotely launch workers, query their execution state (idle or active), and set a flag that causes a specific worker to exit its hot loop. The controller is also responsible for orchestrating the entire experiment flow, including processing user input, automatically triggering DSU experiments based on that input, and coordinating the system's overall operation.

Figure 3.1 illustrates the testbench architecture. The controller core interacts with a configuration management to set test parameters that determine the duration of each test, declare the interval between DSU attempts, decide the number of worker cores to activate, and setup various miscellaneous DPDK flags (e.g. `no-telemetry`). Note that the controller executes a cache-warmup phase to prime all CPU caches, even worker core caches, before measurements begin. This removes cold-start effects and leads to more consistent test results of steady-state behavior.

Figure 3.2 presents a pseudocode representation of the interaction between the worker and the controller. First, the controller performs all essential initialization inside the init function, which parses the terminal arguments provided through argc/argv and configures the runtime environment. As part of this setup, it allocates a shared memory region for communication between the worker and the controller, using a `struct` called `S`. Once the environment has been configured, the controller spawns the worker cores by remotely launching the start function on each target core. After all workers are active, the controller enters its while loop, where it continuously generates packets and ports them to the worker cores.

Each activated worker core can then dequeue the packets and begin some processing using the functions from the shared library. The workers run inside a busy-waiting polling loop,

Controller core

```
1  int main(argc, argv) {
2    init(argc, argv);
3    struct foo_args S = {...};
4    ...
5    /* Remote launch threads */
6    foreach core_id
7      launch(foo, &S, core_id);
8    ...
9    while(time() < deadline){
10     generate_packets(&eventQ);
11     if (deadline_dsu < time()){
12       perform_dsu();
13       increment(deadline_dsu);
14     }
15   }
16 }
```

Worker core(s)

```
1  int foo(*S){
2    ... // Declare variables etc.
3    while(S->active) {
4      pkts = dequeue(&eventQ);
5      foreach pkt in pkts {
6        /* Some processing using
              the library */
7        lib_foo1(pkt);
8        lib_foo2(...);
9        ...
10       log_testbench_metrics();
11       destroy(pkt);
12     }
13   }
14 }
```

Figure 3.2: The interaction between the workers and the controller.

never yielding to the OS scheduler and remaining fully dedicated to packet processing. Each worker repeatedly dequeues available packets and processes them using application-specific functions, similar to how `lib_foo1` is used, before updating any relevant testbench metrics and destroying the packet. This hot loop continues indefinitely, or until the controller decides to terminate the test. To stop the workers cleanly, the controller simply flips the `S->active` flag. Since each worker checks this flag on every iteration of its polling loop, it may take some time before they exit. Once a worker has exited it transitions into an idle execution state.

The design leverages architectural modularity to avoid complex synchronization between workers. Because each worker operates within its own isolated execution context on a dedicated core, workers can be updated independently. This enables parallelization of dynamic updates, as each worker can modify its relevant thread-local state without interfering with other workers.

As can be seen at lines 9–14 in Figure 3.2, the controller executes a timed loop in which it generates packets and upon reaching the DSU deadline invokes `perform_dsu()` to initiate an update. After triggering the update, it advances `deadline_dsu` to schedule the next one. How update operations are applied is determined by the DSU solution in use, and the exact procedure varies between solutions. Despite this, all solutions share the same abstraction: a delayed-update mechanism implemented through function-pointer swaps.

Updates were done using the POSIX dynamic-linking API (`dlfcn`) [29], which exposes mechanisms for interacting directly with the dynamic linker. This interface allows the system to manually load and unload shared libraries, resolve symbol addresses, retrieve function pointers, and manage recursive dependencies under explicit program control. Crucially, it enables shared libraries to be manually loaded and relocated at a time of choice. By obtaining function pointers at runtime, the system can directly control which version of a function is invoked, allowing it to substitute or update implementations on demand.

Figure 3.3 details how a typical update procedure was done in the `perform_dsu` function. First, a path to the targeted library must be provided, this identifies the new version that the system should switch to. The function `dlopen()` attempts to load the new shared library with immediate binding and returns a handle representing the loaded object if successful. Using this handle, the system then queries each required symbol through `dlsym()`, retrieving the corresponding function pointers from the new shared library. After all symbols have been resolved, the code checks `dlerror()` to ensure that no lookup failed. If any step

```
int perform_dsu(const char* lib_path) {
    void* handle = dlopen(lib_path, IMMEDIATE_BIND);
    if (handle == NULL) {
        printf(stderr, "dlopen: %s\n", dlerror());
        return 1; // Discard update
    }
    dlerror(); // Clears error buffer
    fn_type1 foo1 = (fn_type1)dlsym(handle, "foo1");
    fn_type2 foo2 = (fn_type2)dlsym(handle, "foo2");
    ... // Cast and assign values from new library
    fn_typeN fooN = (fn_typeN)dlsym(handle, "fooN");

    const char* e = dlerror(); // Detects any dlsym error
    if (e != NULL) {
        printf(stderr, "dlsym: %s\n", e);
        return 2; // Discard update
    }
    // At this point all relocations are valid
    // Proceed with storing and using the update artifacts
    return 0;
}
```

Figure 3.3: Dynamic linking routine performed at runtime when a DSU is invoked.

of the process failed, the update attempt was discarded and no changes were applied. This means that the library is closed, no state is modified, and execution continues using the existing implementation. Because failed attempts have negligible side effects the controller can simply retry the update at a later time.

From an implementation standpoint, several technical considerations merit attention. The dynamic linker maintains an internal reference count for each loaded shared object, ensuring that a shared library is not actually deallocated until `dlclose()` has been called on it as many times as `dlopen()` has succeeded on it. If the same shared library is opened multiple times, the same object handle is returned. Consequently, repeated calls to `dlopen()` on the same library path do not create multiple independent instances but return the same object handle while incrementing its reference count. This behavior guarantees consistency in symbol resolution but also requires disciplined resource management to avoid unintended retention of obsolete library versions.

In our design, clarity and explicitness are achieved by storing update artifacts, that is both the object handle and the resolved function pointers, in ordinary program variables, making the active implementation visible and controllable at the source-code level. This enables a lightweight, custom indirection mechanism that serves a similar purpose to the PLT/GOT, but is implemented within our system and tailored to our needs.

3.3 DSU Solutions for Library Updating

This section presents the DSU solutions developed during this thesis, outlining their names, design principles, and concrete mechanisms. Each solution reflects a different balance between update latency, service downtime, implementation complexity, memory and performance overhead, and overall reliability. This forms the foundation for the evaluation presented in the following chapters.

Quiescence all-at-once (QALL)

This update strategy relies on shared memory communication between controller and workers to enforce quiescence. As shown in Figure 3.4, each worker is adapted to store thread local variables for update artifacts. These thread-local variables ensure that each worker core invokes its own copy of the function pointer, avoiding cross-core interference and eliminating the need for global locks.

Worker-side behavior

At steady-state, each worker executes inside the start() function, which has received a pointer to the shared `active_flag`. The worker repeatedly executes its hot loop, performing packet processing or other workload-specific tasks. At each iteration boundary, the worker reads the `active_flag`. As long as the flag remains set, the worker continues processing and invokes the current implementation through `foo_ptr()`. When the flag is flipped the worker observes the change at the next iteration boundary and cleanly exits its hot loop. This provides a quiescent point where no worker is executing code from the shared library, allowing updates to be performed without risking inconsistent state.

The function `release_handle()` checks whether the worker currently holds a library handle and, if so, attempts to release it using `dlclose()`. This step is performed before installing a new version to ensure that no stale references remain and to avoid the memory overhead associated with retaining multiple loaded instances of the same shared library. The update is applied by `perform_dsu()`. This function loads the new shared library using immediate binding, resolves the required symbol using `dlsym()`, and installs the new function pointer and handle into the worker's thread-local storage. If any part of this process fails, the function returns an error code and does not alter any thread-local variables.

Controller-side behavior

The controller orchestrates the entire update cycle. It begins by allocating the shared `active_flag` and performing the necessary initialization for the testbench environment. Once the system is prepared, The controller launches functions on each worker core using `rte_eal_remote_launch(fn_ptr, args, core_id)` [28]. When the update deadline is reached, the controller clears the `active_flag`. Each worker observes the change at its iteration boundary and cleanly exits its hot loop. The controller waits for each worker to become idle using an execution blocking call like `rte_eal_wait_lcore()` [28], which can additionally return the value produced by the worker's last executed function.

Once all worker are quiescent, the controller invokes `release_handle()` and `perform_dsu()` in sequence, blocking after each invocation to ensure that every worker completes the corresponding step before proceeding. If any worker reports an error, the update attempt is discarded and the system continues using the previous implementation. If all workers succeed, the controller sets the `active_flag` again and relaunches the `start` function on each worker. The workers resume their hot loops and can process using the newly installed function pointer. The controller then advances the update deadline and continues normal operation.

This design ensures that updates occur only when all workers are quiescent, each worker installs the new implementation independently, failed update attempts have negligible side effects, and that the system achieves zero library coexistence time, thereby avoiding the memory overhead associated with retaining multiple loaded instances of the same shared library. Although it has a notable drawback: the update latency and downtime depends on how long it takes the slowest worker to reach its loop boundary. A timing skew can arise when one worker exits its loop immediately and becomes idle, while another worker completes one final iteration. In this case, the downtime experienced by the first worker is entirely determined by how quickly the slowest worker leaves its hot loop.

WORKER.C

```
thread_local handle;
thread_local foo_ptr;

int release_handle() {
  if (handle == NULL) { return 1; } // No handle to release
  dlclose(handle); // May fail
  if (dlerror) { return 2; }
  return 0;
}
int perform_dsu(lib_path) {
  new_handle = dlopen(lib_path, IMMEDIATE_BIND); // May fail
  if (dlerror) { return 1; }
  new_foo_ptr = dlsym(handle, "foo"); // May fail
  if (dlerror) { return 2; }
  handle = new_handle;
  foo_ptr = new_foo_ptr;
  return 0;
}
int start(*worker, *active_flag){
  ...
  while(active_flag == 1) {
    ...
    foo_ptr(...); // Processing using library
} }
```

CONTROLLER.C

```
int main(argc, argv) {
  active = malloc(...);
  struct worker_state workers[MAX_WORKERS];
  ... /* Initialize, set default values, etc. */
  foreach (i, core_id)
    remote_launch(start,(&workers[i], &active]), core_id);
  while(time() < deadline) {
    ... /* Generate packets */
    if (deadline_dsu < time()) {
      active = 0;
      /* Wait until all workers finish their job */
      foreach core_id {
          remote_launch(release_handle, NULL, core_id);
      } foreach core_id {
          result = wait_lcore(core_id);
          assert(result == 0); // Handle was released
      } foreach core_id {
          remote_launch(perform_dsu, LIB_PATH, core_id);
      } foreach core_id {
          assert(wait_lcore(core_id) == 0); // DSU success
      }
      active = 1;
      foreach (i, core_id)
        remote_launch(start,(&workers[i], &active]), core_id);
      increment(deadline_dsu);
} } }
```

Figure 3.4: Controller and worker behavior in the QALL solution.

Quiescence modular (QMOD)

The modular quiescence approach to updating the workers is very similar to QALL. The main difference is that the single `active_flag` is replaced with per-worker flags. As shown in Figure 3.5, these flags are stored in `worker_state` structures. The `workers` array holds one state object per worker, and the appropriate entry is passed to each worker for update coordination. In the testbench, the state objects are also used to configure workers and track supplementary metrics such as packet latency and transmitted packet counts. These fields are irrelevant to the DSU mechanism and omitted from figures containing the structures.

Worker-side behavior

Each worker core executes the `start` function, invokes the shared library function through a local pointer, and checks the `state->active` flag at each iteration. This design allows the controller to force each worker into quiescence independently rather than halting all workers at once. As a result, update latency becomes dependent on the responsiveness of the individual worker rather than the slowest worker in the system, significantly reducing downtime and modularizing update timing across workers.

Inside `perform_dsu()`, the worker loads the new library, resolves the updated symbol, releases its old handle if one exists, and installs the new update artifacts into thread-local storage. If any step fails, the worker reports an error and leaves its existing implementation unchanged.

Controller-side behavior

Once the controller determines that an update should occur, it clears the `active` flag for a specific worker, waits for that worker to exit its hot loop, and then launches `perform_dsu()` on that worker. If the update completes successfully, the controller restarts the worker by invoking `start()` again with the same state structure.

QMOD introduces a preemptive dynamic linking step performed by the controller at line 11. Before initiating quiescence on any worker, the controller opens the new library to trigger symbol resolution and relocation work ahead of time, so that this work is done by the controller instead of a worker. After all workers have attempted their updates, the controller releases this temporary handle using `dlclose()`, so that no additional memory overhead persists. This optimization reduces worker downtime and shifts the cost of dynamic linking away from the critical path.

A potential improvement like the preloading optimization can be done in the post-update phase. Instead of having the last worker perform the dynamic linking work associated with unloading the old library, the controller could do it instead. By storing a second temporary handle for the old library handle and invoking `dlclose()` on it after each worker has unloaded its own equivalent handle, the dynamic linker is forced to perform the associated teardown work on the controller's core. This shifts additional dynamic linking costs off the worker's critical path and further reduces per-worker downtime during DSU.

WORKER.C

```
thread_local handle;
thread_local foo_ptr;

/* Returns non-zero on dynamic linker error */
int perform_dsu(lib_path) {
  new_handle = dlopen(lib_path, IMMEDIATE_BIND); // May fail
  new_foo_ptr = dlsym(handle, "foo"); // May fail
  if (handle != NULL) {
    dlclose(handle); // May fail
  }
  handle = new_handle;
  foo_ptr = new_foo_ptr;
  return 0;
}

int start(*worker){
  ...
  while(worker->active == 1) {
    ...
    foo_ptr(...); // Processing using library
  }
}
```

CONTROLLER.C

```
int main(argc, argv) {
  struct worker_state {
    bool active; // New member
    ...
  } workers[MAX_WORKERS];
  ...
  while(time() < deadline) {
    ... /* Generate packets */
    if (deadline_dsu < time()) {
      temp = dlopen(LIB_PATH, IMMEDIATE_BIND);
      foreach (i, core_id) {
        workers[i].active = 0;
        wait_lcore(core_id) // Let worker finish its job
        remote_launch(perform_dsu, LIB_PATH, core_id);
        assert(wait_lcore(core_id) == 0); // DSU success
        workers[i].active = 1;
        remote_launch(start, &workers[i], core_id);
      }
      dlclose(temp); // Release preload handle
      increment(deadline_dsu);
    }
  }
}
```

Figure 3.5: Controller and worker behavior in the QMOD solution.

Global function pointer table (GFPT)

The GFPT approach allows workers to switch library versions without stopping, using a shared global table of function pointers. Each worker repeatedly reads the current version index and function pointer from this table using atomic operations, ensuring that updates published by the controller are observed in a consistent order [30]. Workers transition to using the new library version as soon as the controller publishes it, eliminating the need for explicit quiescence during the update itself.

Worker-side behavior

Figure 3.6 shows how each iteration of the worker's hot loop atomically reads the current version index and function pointer, from the table `T` using `ACQUIRE` semantics. This guarantees that if the controller has published a new function pointer using a `RELEASE` store, the worker will observe the updated pointer *after* the version index has been updated. Workers record the version index they observed, allowing the controller to track which version each worker is currently executing. The ordering of these atomic loads is important since workers should never see a new version index paired with an old function pointer.

The workers invoke the function `foo_ptr` through the freshly loaded pointer, ensuring that all processing uses the new implementation. This design introduces a small per-iteration performance overhead due to the atomic loads, but in exchange provides zero downtime. Workers never need to stop or enter a quiescent state to adopt the new version, each worker simply transitions to the new library on its next loop iteration. Because atomic operations are indivisible they ensure that a pointer is read as one consistent value, i.e. a fully written 64-bit function pointer.

Controller-side behavior

The controller is responsible for installing new library versions, publishing updated function pointers to the global table, and safely reclaiming old library instances. The array `handles[2]` is used to keep track of the shared library instances used by the GFPT mechanism. Each slot is used to store an active library handle and during cleanup when old instances are reclaimed. This design makes it so at most one old version and one new version can coexist at any time.

The update procedure begins in `perform_dsu()`, where the controller loads the new shared library using immediate binding. If any error occurs, the function returns an error code and no state is modified. Once the symbol is successfully resolved, the controller computes the next version index by flipping the current one (`1-T.version_index`).

Before publishing an update, the controller verifies that the target slot in the `handles` array is unused. If the slot is already occupied, the update is aborted to prevent overwriting a handle that may still be in use. Otherwise, the controller stores the new handle in the appropriate slot and publishes the new function pointer *then* the version index using a `RELEASE` atomic store. This ordering guarantees that worker that observe the new index will also observe the new function pointer.

A successful update sets a flag indicating that two library versions now coexist, and the controller advances the DSU deadline. To reclaim memory, the controller invokes `cleanup_dsu()` whenever two library versions coexist. The routine reads the current version index with an `ACQUIRE` load and checks each worker's last observed version. If any worker is still using the old library it cannot be unloaded. Once all workers have transitioned to the new version, the controller identifies the old index, verifies that a handle exists, and attempts to unload the library using `dlclose()`. If successful, the corresponding handle slot is cleared so that no stale reference remain.

CONTROLLER.C

```
// Global table for DSU symbols
struct function_pointer_table T; // Definition

/* Returns non-zero on dynamic linker error */
int perform_dsu(lib_path, handles) {
  new_handle = dlopen(lib_path, IMMEDIATE_BIND); // May fail
  new_foo_ptr = dlsym(handle, "foo"); // May fail
  new_index = 1-T.version_index;
  if (handles[new_index] != NULL) { return 3; }
  handles[new_index] = new_handle;
  atomic_store(&T.foo_ptr, new_foo_ptr, RELEASE);
  atomic_store(&T.version_index, new_index, RELEASE);
  return 0;
}

int cleanup_dsu(workers, handles) {
  current = atomic_load(&T.version_index, ACQUIRE);
  for (int i=0; i<MAX_WORKERS, i++)
    if (workers[i].version_index != current)
      return 1; // Worker still using old library, cannot unload
  old = 1-current;
  if (handles[old] == NULL) { return 2; } // No handle to release
  dlclose(handles[old]); // May fail; abort and return non-zero
  handles[old] = NULL;
  return 0;
}

int main(argc, argv) {
  void *handles[2]; // Loaded library handles
  struct worker_state {
    int worker_version; // New member
    bool active;
    ...
  } workers[MAX_WORKERS];
  ...
  bool coexisting_libraries = false;
  while(time() < deadline) {
    ...
    if (deadline_dsu < time()) {
      assert(perform_dsu(LIB_PATH, handles) == 0); // DSU success
      coexisting_libraries = true;
      increment(deadline_dsu);
    }
    if (coexisting_libraries) {
      if (cleanup_dsu(workers, handles) == 0) // Cleanup success
        coexisting_libraries = false;
    }
  }
}
```

WORKER.C

```
1 extern struct function_pointer_table T; // Declaration
2
3 int start(*worker){
4   ...
5   while(worker->active) {
6     ...
7     worker->worker_version = atomic_load(&T.version, ACQUIRE);
8     foo_ptr = atomic_load(&T.foo_ptr, ACQUIRE);
9     foo_ptr(...); // Processing using library
10  }
11 }
```

Figure 3.6: Controller and worker behavior in the GFPT solution.

Flag induced update (FLAG)

The FLAG approach coordinates dynamic software updates using shared memory for a flag and a path variable. Unlike QALL and QMOD, workers are not stopped but instead updated when the flag is raised. In contrast to GFPT, the update execution is decentralized and parallelized across workers. This design minimizes controller involvement.

Worker-side behavior

Figure 3.7 outlines the concrete mechanisms of this solution. Each worker maintains two thread-local variables: a handle to the currently loaded shared library and a function pointer to the active implementation of `foo`. The worker repeatedly executes its hot loop while its `active` flag remains set. At the beginning of each iteration, the worker checks its `dsu_flag`. If the flag is raised, the worker invokes `perform_dsu()` to update itself. This design introduces a negligible per-iteration performance overhead due to the `if` check on the flag, which costs essentially nothing when the branch is correctly predicted.

Inside `perform_dsu()`, the worker loads the new library specified by its own `lib_path`, resolves the updated symbol, and unloads the previous library via `dlclose()` if an old handle exists. After installing the new handle and function pointer, the worker clears its `dsu_flag`, ensuring that the update is performed exactly once. The worker then resumes normal processing, invoking the newly installed function pointer on subsequent iterations. This design allows all workers to update independently and in parallel, with no need for controller-driven quiescence or restart. Update latency therefore depends solely on how quickly each worker reaches the flag check in its loop and downtime is dependent on how long `perform_dsu()` takes to successfully execute.

Controller-side behavior

The controller periodically evaluates whether a dynamic update should occur based on the DSU deadline. When the deadline is reached, it iterates over all workers and sets each worker's dsu flag to 1. This signals the workers to perform their updates the next time they reach the flag check. The controller does not wait for workers to finish updating, does not coordinate quiescence, and does not perform any linking work itself. Instead, all update activity is delegated to the workers. Because workers update themselves asynchronously, the FLAG mechanism achieves minimal controller overhead.

WORKER.C

```
thread_local handle;
thread_local foo_ptr;

/* Returns non-zero on dynamic linker error */
int perform_dsu(*worker) {
  new_handle = dlopen(worker->lib_path, IMMEDIATE_BIND); // May
      fail
  new_foo_ptr = dlsym(handle, "foo"); // May fail
  if (handle != NULL) {
    dlclose(handle); // May fail
  }
  handle = new_handle;
  foo_ptr = new_foo_ptr;
  worker->dsu_flag = 0;
  return 0;
}

int start(*worker){
  ...
  while(worker->active == 1) {
    if (worker->dsu_flag == 1)
      perform_dsu(worker);
    ...
    foo_ptr(...); // Processing using library
  }
}
```

CONTROLLER.C

```
int main(argc, argv) {
  struct worker_state {
    bool active;
    bool dsu_flag; // New member
    char* lib_path; // New member
    ...
  } workers[MAX_WORKERS];
  ...
  while(time() < deadline) {
    ...
    if (deadline_dsu < time()) {
      for (i=0; i<MAX_WORKERS, i++) {
        workers[i].dsu_flag = 1;
      }
      increment(deadline_dsu);
    }
  }
}
```

Figure 3.7: Controller and worker behavior in the FLAG solution.

3.4 Benchmark Profiling

This section explains how measurements were collected for the various metrics used to cross-compare solutions. All experiments in this section were conducted within the DPDK testbench environment.

Program restart

Updating the RAN application can also be done by restarting the program, which is why a normal program restart was measured to establish a baseline for comparison with DSU downtime. The program restart represents a lower bound for downtime in the absence of any DSU mechanisms. To this end, a cold start of the DPDK testbench was done, measured from process creation to normal termination. To ensure that the measurement isolates process teardown, initialization, and setup costs of a typical DPDK program, the testbench was configured to be non-generative, thereby bypassing all packet transmission work. A DPDK program start cannot be meaningfully warmed up, because a restart requires a complete reinitialization of hugepages and the environment abstraction layer [28].

The timed interval was measured using two Linux tools. First, `strace` [31] was used to record system call activity and associated kernel-space delays during process startup and dynamic linking. Although it introduces some overhead, it remains suitable for capturing teardown and initialization behavior. Second, `perf stat` [32] was used to record whole-program behavior and measure execution time with high accuracy. Together, these measurements produce the total cost of a conventional restart, enabling a direct comparison with DSU solutions.

The `perf` profiler is a Linux performance-monitoring tool designed to record a large collection of hardware and kernel performance counters over the lifetime of a program. Unlike periodic-polling based profilers, it uses hardware counter accumulation to provide precise totals for a variety of metrics including CPU cycles used, instructions completed, cache misses, and branch misses. Although not all of its metrics were needed for the restart measurement, `perf` was still an important tool for analyzing whole-program behavior during the thesis work.

Downtime and update latency

The update-induced *downtime* is measured from the time a worker stops processing until it resumes operation. *Update latency* is the time interval between the moment an update is requested and the moment the update begins to take effect (recall Figure 2.8). To obtain values for the downtime and update latency, the system was put through repeated dynamic updates. Each time an update occurred a sample was taken for the respective value.

Let t_0 denote the time when the DSU is requested, t_1 the time when the entry point of `perform_dsu` is reached, t_2 the time when the worker is redirected away from its hot loop, and t_3 the time when the worker resumes normal processing using the new library:

$$\text{Update latency} := t_1 - t_0, \qquad \text{Downtime} := t_3 - t_2.$$

These values were measured with high-resolution timing obtained using the processor's Time Stamp Counter (TSC), a 64-bit hardware counter. Reading the current TSC value was done through the low-overhead Read Time Stamp Counter (RDTSC) instruction. Differences between subsequent timestamp readings were used to compute the intervals.

All this data was collected in the DPDK testbench at appropriate locations depending on the specific solution in use. Because TSC reports raw clock cycle counts, all measurements were converted to milliseconds before analysis. This was done by dividing them by the CPU frequency. Additionally, the tools `objdump` [33] and the GNU Debugger [34] were used to

confirm that the measured regions corresponded to the expected machine-code sequences and that we actually measure what we intended to measure.

In cases where instruction serialization was required, to ensure that measurements reflected the intended code regions and were not affected by compiler reordering or out-of-order execution, the `rte_rdtsc_precise()` function from `rte_cycles` [28] was used. This function internally applies fencing, a mechanism that forces the CPU to complete all prior instructions before executing the timestamp read. Fencing prevents the CPU's instruction pipeline from moving the RDTSC instruction earlier or later than intended, thereby ensuring accurate timing boundaries.

Performance Overhead

The performance overhead introduced by the DSU mechanisms was evaluated using two complementary approaches: (1) micro-architectural timing measurements collected within the workers, and (2) performance analysis based on packet-level logs.

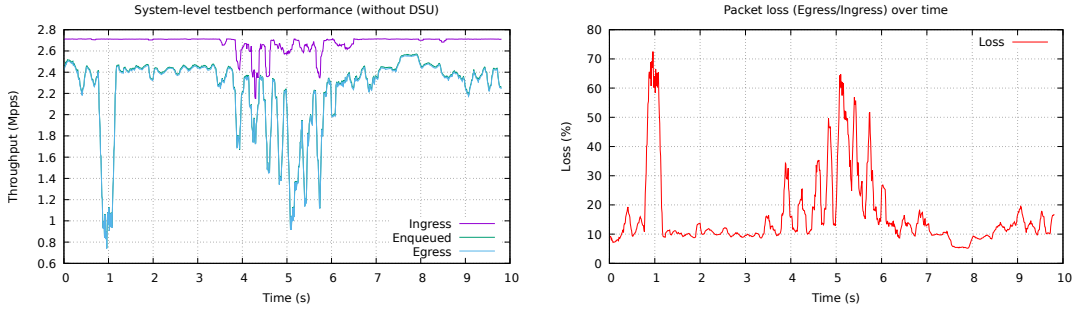
Timing measurements were collected using `perf stat` and RDTSC-based timers to capture potential micro-architectural degradations such as increased function call latency, instruction-cache misses, or disruptions in the CPU pipeline. By comparing cycle-level timings in the hot loop, before and after restructuring the program for DSU, it is possible to quantify the steady-state performance overhead. In this context, the performance overhead was measure per function invocation, meaning we quantify how much more it costs to call a function in the DSU system compared to the non-DSU baseline system.

The DPDK testbench logged metrics such as Ingress, Enqueued, Egress and Loss of packets under worst-case transit conditions. *Ingress* refers to the total number of packets received by the system at the input interface, representing the raw workload presented to the pipeline. *Enqueued* is the amount of packets that were successfully placed into the worker queues for further processing, reflecting the system's ability to buffer incoming traffic. *Egress* captures the number of packets that successfully exited the system after processing. Finally, *Loss* represents the percentage of packets that were dropped at any stage.

These counters can be used to infer whether a DSU operation introduces observable degradation in throughput. Because packet generation was made identical across experiments, the workload in the workers was consistent between DSU-enabled and non-DSU runs of the testbench. Figure 3.8 shows data collected from a baseline testbench run under normal operation, with each data point representing packet statistics aggregated over a 10 ms window. The ingress and egress rates exhibit considerable noise, suggesting instability in packet generation and processing. The enqueued rate closely matches the egress rate, making them effectively indistinguishable, and Figure 3.8b shows the packet-loss ratio, computed as $\frac{egress}{ingress}$ at each data point. Some packet loss was expected, since experiments were done in a worst-case transit scenario in which the system receives more traffic than it can process ($\approx$ 2.7 Mpps), resulting in unavoidable packet drops. These statistics allow us to determine whether DSU activity causes measurable deviations in packet handling performance, for example by checking whether the system ever reaches sustained 100% packet loss.

Memory overhead

We define *memory overhead* as the increase in proportional set size (PSS), i.e. memory consumption, introduced by the DSU system relative to the baseline program, measured at steady state after initialization effects have subsided. Memory overhead may arise by adding code or new APIs to support the DSU. This was assessed using a combination of analytical reasoning and experimental measurements. From an analytical standpoint, one expected source of overhead arises when multiple versions of a shared library coexist in memory. When multiple versions of a shared library remain loaded simultaneously the program's memory footprint increases proportionally to the size of those libraries. Note that all our



(a) System behavior during a 10-second baseline DPDK testbench run.

(b) Packet loss profile for the corresponding testbench run.

Figure 3.8: Example of packet-level ingress, enqueue, egress, and loss rates over time used for system-level performance analysis.

DSU solutions explicitly call `dlclose()` on outdated libraries, so any additional memory usage from coexisting versions is short-lived. Our primary focus is on the steady-state memory overhead introduced by restructuring the program for DSU, rather than the short-lived overhead during the update phase.

Memory usage was measured using periodic polling, invoking `smem` [35] once per second. The tool `smem` reports memory consumption using metrics such as Unique Set Size (USS) and Proportional Set Size (PSS), that account for shared pages and private allocations. These metrics provide another viewpoint of process memory usage than the traditional Resident Set Size (RSS) metric that counts everything the process maps, even if most of it is shared.

A structured comparison of how memory usage differs between the original system and the system restructured to support DSU was done by taking measurements for a non-DSU-adapted testbench and the corresponding DSU-adapted testbench. This comparative analysis makes it possible to isolate the steady-state memory overhead.

Reliability

Reliability in this context refers to two separate properties: (1) the absence of crashes during dynamic updates, and (2) the preservation of correct program behavior after each update. Both aspects were assessed through analytical reasoning and stress testing.

To measure how rare update failures were, the system was put through repeated dynamic updates under accelerated conditions. This was done simultaneously as the downtime and update latency measurements. Each time a DSU was applied the system alternated between using e.g. `lib_v0` and `lib_v1`, loading the new library and unloading the previous one on every update cycle. Correct post-update behavior was verified by having functions in the two library versions return distinct values (1 in `lib_v1` and 0 in `lib_v0`), allowing the main program to check at runtime whether the active library corresponds to the expected version. This provided a simple and deterministic way to confirm whether the intended code is used after an update or not.

These experiments served as a empiric test, where we specifically looked for any errors or crashes. In this context, errors include both dynamic-linking errors and incorrect post-update behavior. While this stress test cannot provide a decisive guarantee of correctness, it serves for exposing how rare update failures are.

```
#define CALL256(x) x; x; ... x;
int foo() { return 0; }
void test() {
    ...
    for (int i=0; i < NR_SAMPLES; i++) {
        T0[i] = rte_rdtsc_precise();
        CALL256(foo())
        T1[i] = rte_rdtsc_precise();
        CALL256(shared_library_foo())
        T2[i] = rte_rdtsc_precise();
    }
    // Direct cost:   (T1-T0)/256
    // Indirect cost: (T2-T1)/256
}
```

Figure 3.9: Macro-expanded approach for accurate microbenchmarks.

3.5 Isolated Microbenchmarks

To better understand the performance overhead introduced by indirection and DSU mechanisms, we conducted isolated microbenchmarks comparing various function call mechanisms. This was done by timing the latency of calling a function that was called was intentionally kept minimal to isolate the cost of the calling mechanisms. Furthermore, our environment had to amortize the overhead of the timestamp reads to enable single-digit clock cycle accuracy. This was accomplished by using a macro `CALL256`, that issues 256 consecutive calls per measurement.

Initially, an environment was constructed that ensured the presence of 256 explicit `callq` instructions in the compiled binary. Inspecting the binary with `objdump` [33] confirmed the presence of all these call sites as separate instructions, despite using compiler optimizations. Each `callq` instruction requires 5 bytes, so the total code footprint would be $256 \cdot 5 = 1.28$ kB, which must fit in the L1 instruction cache of the CPU to ensure that cache capacity does not distort the measurements. For most modern CPUs this is well within the available cache budget, making 256 call sites a suitable choice for avoiding distortions while still amortizing the timestamp read.

However, because each call site appeared on a new line, the CPU's branch predictor could not exploit repetition, leading to guaranteed mispredictions on every call. To avoid this effect, the microbenchmark was improved by wrapping the measurement structure in a loop, allowing the predictor to stabilize after the first iteration. Looping eliminates misprediction penalties and produces cleaner, more representative timing data, while also facilitating repeated sampling under identical conditions. Immediate binding was used when relocating `shared_library_foo()` to ensure that symbol resolution occurred upfront, preventing lazy binding overhead from influencing the timing results. Figure 3.9 illustrates how the final benchmark was setup.

The cost of indirection was measured under two configurations: (1) invocations routed through local function pointers, and (2) invocations resolved via the PLT/GOT mechanism. The cost $\frac{(T_2-T_1)-(T_1-T_0)}{256}$ in Figure 3.9 captures the indirection cost; that is, the performance overhead introduced by replacing direct calls with indirect calls. This overhead is common to all our DSU approaches and exists before even introducing any DSU-specific logic.

The cost of DSU mechanisms were measured under two configurations: (3) conditional branch checks, and (4) atomic loads followed by indirect calls. These behaviors were incorporated directly into the macro so that each expanded call site contained the desired mechanism. To benchmark conditional checks, the macro invocation `CALL256(foo())` was rewrit-

ten as `CALL256(if(cond){} foo())`. This block invokes `foo` unconditionally while still forcing the CPU to execute the branch instruction. This allows us to measure whether the branch is effectively predicted and pipelined. To benchmark the cost of performing an atomic load before a function call, the macro invocation `CALL256(foo())` was rewritten as `CALL256(bar = atomic_load(...); bar())`.

4 Results

In this chapter, we present the results of our study and evaluate them. We use these findings to answer the three research questions introduced in Section 1.3:

RQ1 How do different DSU approaches compare to a program restart in terms of downtime, memory overhead, and runtime performance?

RQ2 During steady-state operation, what performance degradation results from using indirect function calls instead of direct function calls?

RQ3 How do DSU mechanisms influence the service of a RAN-like system? Can updates be applied while the system is actively serving traffic?

4.1 Execution Setup

As described in Section 3.4, data was collected to evaluate downtime, update latency, performance overhead, memory overhead, and reliability. Most measurements were collected within the DPDK testbench described in Section 3.2. The testbench consists of a dedicated controller core and a configurable number of worker cores. The controller recorded global packet-level metrics, while each worker core captured its own timing-related measurements relevant to the DSU mechanism under test.

Packet-generation parameters were kept identical across all experiments so that the baseline and DSU-enabled runs were directly comparable. All measurements were taken during steady-state execution, unless explicitly stated otherwise. This ensured that initialization effects and warm-up behavior had already subsided before data collection began.

The experiments were conducted on a Linux Red Hat 8 system on an AMD EPYC 9354 processor using the x86-64 instruction set. The software stack consisted of DPDK 22.03 [28], the GCC 8.5.0 compiler, and the C17 language standard. A total of four dedicated CPU cores were allocated to the testbench, with one core acting as the controller and the remaining cores serving as workers.

To ensure comparability across experiments, the packet generator was configured to produce a stable workload of approximately 2.7 million packets per second (MPPs), and this rate was kept invariant throughout all measurements.

4.2 RQ1: Comparison Against Program Restart

Motivation

Comparing DSU mechanisms against a conventional program restart is essential to evaluate their practical usefulness in performance-critical systems. A program restart represents the baseline update strategy, where the entire process is terminated and reinitialized. This approach is simple and robust but inherently introduces service interruption due to process teardown, reloading, and initialization. Interruptions can violate availability requirements, making downtime a critical constraint. DSU aims to significantly reduce downtime and comparing DSU solutions to a restart establishes whether DSU offers any meaningful improvement. In addition, this comparison highlights the trade-offs involved, which can be used to evaluate whether DSU is suitable for deployment.

Data presentation

Following the methods described in Section 3.4, a program restart was measured and three key metrics were collected: downtime, memory overhead, and runtime performance.

To collect downtime samples, each DSU solution was put through repeated updates, resulting in 30,000 update cycles per solution. The downtime samples were plotted using histograms to visualize their distributions, with a logarithmic y-axis to highlight samples that would otherwise be visually suppressed. Histogram binning was performed such that 100 bins of fixed width were plotted, where the x-axis represents downtime intervals and the y-axis the sample count. All histograms were plotted using a fixed and identical axis range. To reduce the influence of noisy outliers 90% trimmed statistics were reported where applicable, excluding the upper and lower 5% of samples.

Results

This section presents a cross-comparison of our four DSU solutions and a base program restart across the metrics: update latency, service downtime, memory overhead, performance overhead, and overall reliability. Table 4.1 summarizes the results.

Update latency

Update latency reflects how quickly a system reacts once an update is requested. Among our solutions, GFPT achieves the lowest latency (less than 0.01 ms), because workers branch into the update logic immediately after the request. FLAG also performs well (0.02 ms), with latency determined by the duration of a worker's iteration loop. In contrast, QALL and QMOD exhibit moderate latencies of 0.11 ms and 0.15 ms respectively, reflecting the time required for workers to reach their quiescent point and synchronize. Although the values for QALL, QMOD, and FLAG are seemingly small, they will increase if the frequency of polling decreases. The baseline restart has no meaningful latency metric.

Downtime

Downtime is the interval during which a worker is hindered from processing packets. This metric shows the most dramatic differences between implementations. A baseline restart incurs 310 – 430 ms of downtime, dominated by process creation and initialization. Measurements from `strace` [31] show that 60 – 90 ms is spent in system calls during process creation and loading. The remaining time is attributed to the initialization, as reported by `perf` [32].

QALL shows a wide range of measurements (0.48 – 69.8 ms), but its trimmed distribution (0.59 – 0.80 ms) and median (0.65 ms) reveal that most updates complete within a millisecond. QMOD performs significantly better, with trimmed downtime between 0.07 – 0.10 ms and

Table 4.1: Cross comparison of the prototyped DSU solutions.

	BASE	**QALL**	**QMOD**	**GFPT**	**FLAG**
Update latency (ms) median	–	0.10	0.15	< 0.01	0.02
Downtime (ms) min/max	310 – 430	0.48 – 69.8	0.04 – 30.8	0	0.00 – 38.3
Downtime (ms) 90% trimmed min/max	–	0.59 – 0.80	0.07 – 0.10	0	0.05 – 0.63
Downtime (ms) median	–	0.65	0.09	0	0.44
Downtime (ms) mean	–	0.85	0.12	0	0.59
Memory overhead (kB)	0	300 – 350	300 – 350	300 – 350	300 – 350
Performance overhead min/max (clk)	0	2 – 10	2 – 10	2 – 212	2 – 11
Performance overhead 90% min/max (clk)	0	2 – 2	2 – 2	9 – 9	2 – 3
Performance overhead median (clk)	0	2	2	9	2
Performance overhead mean (clk)	0	1.93	1.93	9.15	2.31
DSU errors detected (count)	–	0	0	0	0

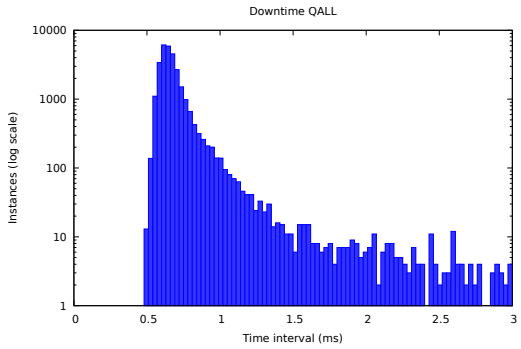
a median of 0.09 ms, confirming that modular quiescence reduces synchronization delays. GFPT eliminates downtime entirely, as workers simply switch to the new function pointers atomically and never stops. FLAG displays a two mode pattern with trimmed downtime between 0.05 – 0.63 ms. These values are all acceptable for typical RAN applications and would allow at least $4000 < \frac{5 \text{ min}}{69.8 \text{ ms}}$ dynamic updates to be made to an in-service RAN application per year without violating availability constraints.

QALL exhibits moderate downtime, with no samples being found below 0.48 milliseconds (ms). The data reveals that 90% of the distribution is concentrated in the range 0.59 – 0.80 ms. This pattern aligns with the design of QALL, since all workers must reach a quiescent update point before the update can proceed. The downtime therefore corresponds to the time required for the slowest worker to complete its final iteration.

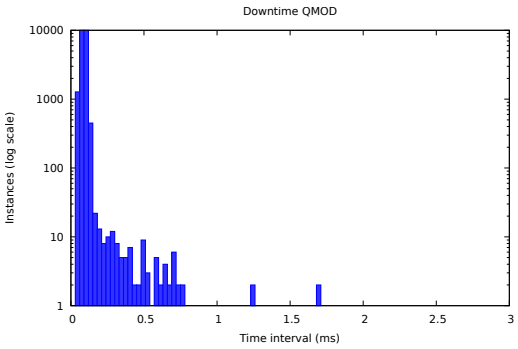
Figure 4.1b reveals that QMOD achieves significantly lower downtime than QALL, with 90% of the distribution concentrated within 0.07 – 0.10 ms. The data shows a sharp drop-off, indicating that downtime samples more than one standard deviation ($\sigma = 0.05$) above the mean are very rare. This confirms that QMOD's more selective quiescence mechanism reduces downtime. Compared to QALL, the distribution is both narrower and shifted left, reflecting the reduced synchronization wait time.

Figure 4.1c stands out because GFPT exhibits effectively zero downtime. This behavior is expected since GFPT allows workers to do packet processing without interruption. Since no update work is performed by the worker, the update becomes instantaneous from the worker's perspective and the histogram collapses into a single bin at 0 ms.

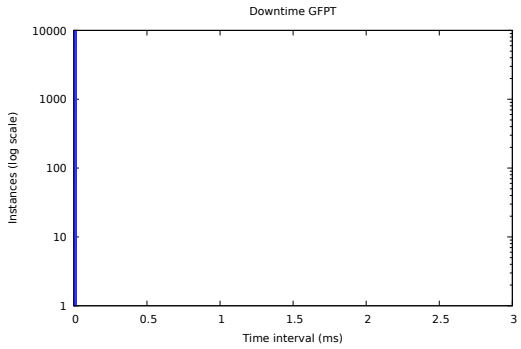
Figure 4.1d shows that FLAG has two modes of downtime, one at 0.02 ms and another larger mode clustering around 0.44 ms. FLAG's downtime depends on how quickly workers complete their `perform_dsu` routine. The first worker to reach `perform_dsu` performs some dynamic-linking work, which on average accounts for 0.50 ms of the update time. This worker, and any workers that arrive early and must wait for the linking to finish, are blocked from doing packet processing, resulting in the higher mode. Workers that reach the update



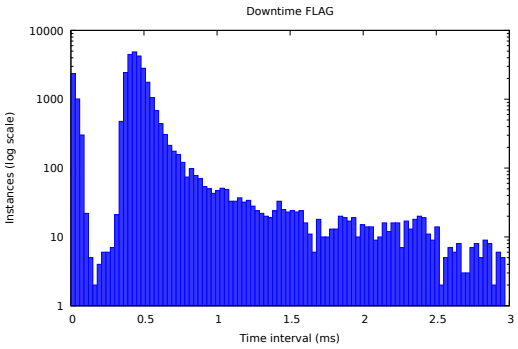
(a) Downtime distribution for QALL.



(b) Downtime distribution for QMOD.



(c) Downtime distribution for GFPT.



(d) Downtime distribution for FLAG.

Figure 4.1: Downtime measurements aggregated in histograms for each solution.

point after the linking work has completed finish their updates quickly, which produces the lower mode. Ideally, most workers should experience the lower mode of downtime. However, because each worker loop iteration completes quickly, the linking work is rarely finished by the time workers reach their update point, causing most workers to not experience the lower mode.

Memory overhead

Memory overhead reflects the increase in memory consumption. Based on `smem` measurements, the baseline has a proportional set size (PSS) of approximately 14.23 MB at the initialization phase and 18.68 MB at steady state, while the equivalent DSU systems reached approximately 14.56 MB at initialization and 18.98 – 19.03 MB at steady state. The PSS difference when comparing the DSU system with the baseline at steady state thus corresponds to an overhead of roughly 300 – 350 kB. This overhead is identical across QALL, QMOD, GFPT, and FLAG, and it is evident that our solutions (Section 3.3) are small in terms of code size. Thus, the specific DSU mechanism in use have negligible effect on memory overhead, which mainly stems from using the POSIX dynamic-linking API [29].

Performance overhead

Performance overhead reflects the extra work performed continuously throughout execution. The baseline program incurs no overhead. QALL, QMOD, and FLAG all show similar behaviors with a median of 2 cycles and similar min-max ranges cycles. These costs correspond to a single indirect branch, which modern CPUs handle efficiently. GFPT, however, is an outlier. Its median overhead is 9 cycles, but the maximum can reach 212 cycles due to the

additional atomic load. For a function call on an extremely hot path this rare outlier may cause problems.

Reliability

Throughout the 30,000 sampled update cycles per mechanism, all DSU solutions consistently reported zero errors, crashes, and incorrect behaviors. This indicates that the update protocols, synchronization logic, and pointer-swapping mechanisms are stable under repeated use and do not introduce correctness issues.

4.3 RQ2: Steady-State Indirection Overhead

Motivation

Indirection is a fundamental building block, as it enables both external functions to be called and execution to be redirected at runtime without modifying call sites. However, it introduces additional work in the program's hot path, as described in Section 2.3. Because this work is incurred on every function invocation, even small overheads can accumulate and impact throughput, which is why quantifying this performance overhead is essential.

Data presentation

To quantify the steady-state cost of indirection and DSU mechanisms, isolated microbenchmarks were conducted as described in Section 3.5. These measurements isolate the cost of function call mechanisms. The evaluated configurations were:

(1) Local function pointer indirection

(2) PLT/GOT-based indirection

(3) Atomic load + (1)

(4) Conditional branch check + (1)

To collect performance overhead samples, a test was ran for each configuration the benchmarks were executed, resulting in 100,000 samples per configuration. The samples were plotted using histograms to visualize their distributions, with a logarithmic y-axis to highlight samples that would otherwise be visually suppressed. Histogram binning was performed such that 21 bins of fixed width were plotted, where the x-axis represents clock cycles and the y-axis the sample count. All histograms were plotted using a fixed and identical axis range.

Results

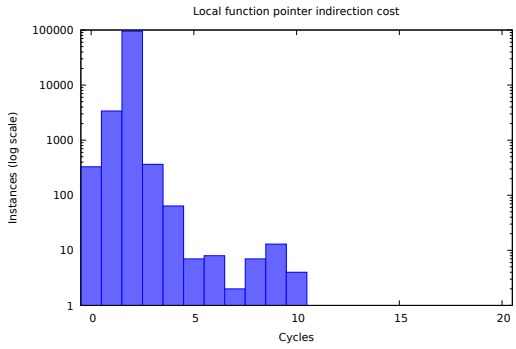
Figure 4.2a shows how much the cost of invoking a function through a local function pointer exceeds the cost of a direct function call. The histogram reveals a tight cluster around the minimum call cost, with the mode corresponding to 2 clock cycles. The data indicates that a highly accurate and optimized path has been used in the CPU pipeline. The branch predictor reliably chooses the correct path, and the instruction cache always hits.

Similarly, Figure 4.2b shows the cost associated with function invocations through the PLT/GOT indirection mechanism compared to direct function calls. Here, the caller branches to a PLT stub, which then loads the final function address from the GOT entry. This 2-step indirection introduces an additional memory load. The shift from Fig. 4.2a to Fig. 4.2b captures the additional penalty of the GOT pointer lookup. The measurements shows this as the mode shifts to 4 clock cycles, doubling the minimum cost observed in the previous case.

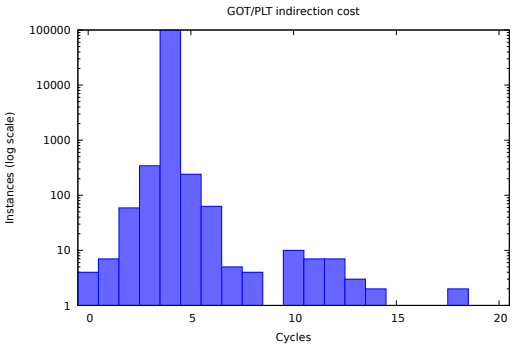
Note that Figure 4.2a effectively represents the performance overhead for QMOD and QALL, as neither solution introduces any additional work inside the worker's hot loop. Consequently, the main source of performance overhead in these arises from the transition to using indirection rather than direct calls.

Figure 4.2c illustrates a noticeable contrast to the other solutions, as it has the highest performance overhead of 9 clock cycles. Recall that GFPT relies on atomic loads, which introduce additional memory loads, with a cost of approximately six to seven clock cycles per atomic load. Our experiment tested an exaggerated scenario by pairing every function invocation with a atomic load. The worst case effect is thus visible in the histogram, where the entire distribution is shifted 7 cycles to the right compared to Figure 4.2a.

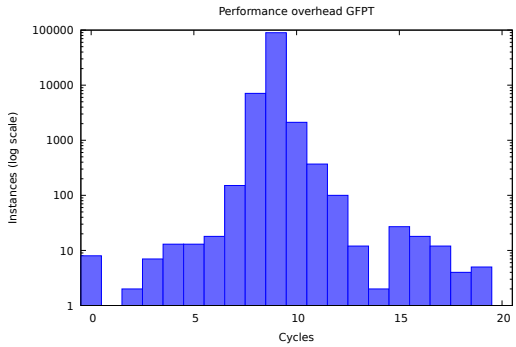
Figure 4.2d shows that FLAG has a similar cost to QALL and QMOD, with the mode at 2 clock cycles. Recall that the added overhead of FLAG consists of a single conditional branch check, which modern CPUs can pipeline so effectively that its cost is too small to measure in our setup. This is reflected in the histogram, 82124 samples fall at 2 cycles for Figure 4.2d and 95596 samples for Figure 4.2a. This is why FLAG has a performance overhead on par with QALL and QMOD.



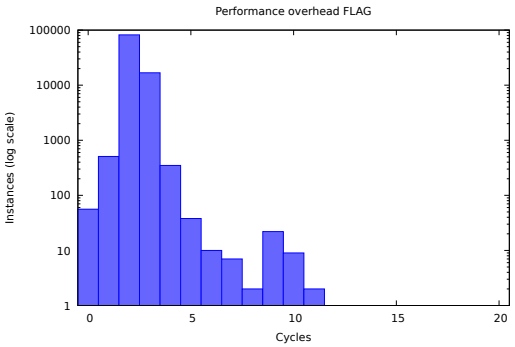
(a) Latency distribution for local function pointers.



(b) Latency distribution for PLT/GOT indirection



(c) Latency distribution for the GFPT solution.



(d) Latency distribution for the FLAG solution.

Figure 4.2: Latency distribution for DSU solutions.

4.4 RQ3: Service Impact of DSU in a RAN System

Motivation

Dynamic Software Update (DSU) mechanisms can influence end-to-end service behavior in a RAN-like system by reducing throughput or causing temporary buffer buildup during an update. Understanding these effects is essential for determining how DSU can be deployed without compromising service-level objectives.

Data presentation

To evaluate how DSU mechanisms affect service behavior, we examined their interaction with the RAN-like workload executed in our DPDK testbench. As described in Section 3.4, packet-level metrics (ingress, enqueued, egress, loss) and microarchitectural timing data was used to assess whether DSU activity introduced observable service degradation. Figure 3.8 illustrated the packet-level metrics using a 10-second baseline run of the DPDK testbench without any updates. This baseline presents the expected steady-state behavior of the system. The same measurement setup was used and graphed for our 4 DSU-enabled testbenches.

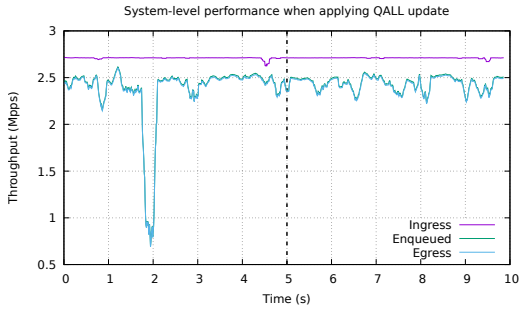
Results

The results show that all evaluated DSU solutions can be applied without noticeably disturbing service at the system level. When combined with the earlier measurements of downtime and performance overhead, these observations reveal the practical constraints and architectural choices that are important to consider when deploying DSU in a RAN-like environment.

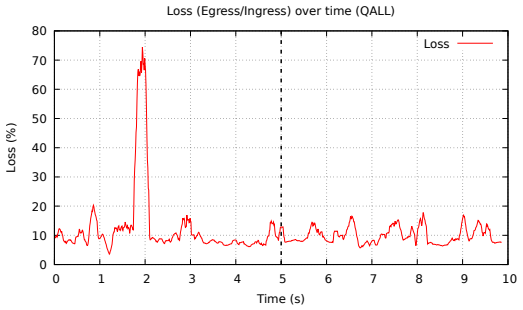
The results in Figure 4.3 demonstrate that there is no noticeable difference in system-level behavior between DSU-enabled runs and the baseline Figure 3.8. Although one might expect a disturbance at the 5-second mark when the dynamic update is triggered, no such spike appears, indicating that the dynamic update can be applied without disrupting the system.

This data suggests that workers are able to maintain their processing capacity without introducing backlog or instability in the pipeline. This implies that DSU does not introduce sufficient delays to cause buffer overflow or packet drops, even under sustained load. The lack of disturbance can be explained by the structure of the packet processing pipeline. The RAN design relies on burst processing and queue buffering, which allows short interruptions in worker execution to be absorbed without propagating.

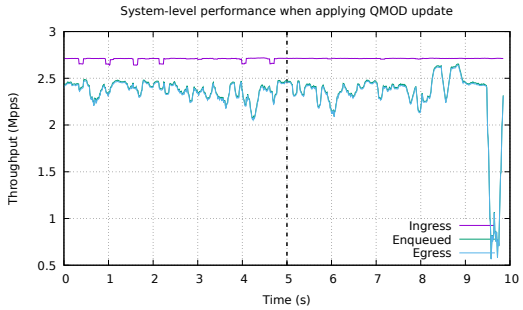
Even mechanisms that temporarily pause workers, such as QALL, QMOD, and FLAG, do not produce observable degradation in throughput. Their downtime is sufficiently short relative to the buffering capacity of the system, allowing normal operation to resume before any measurable effect emerges.



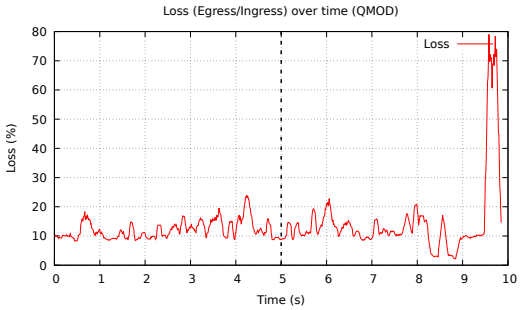
(a) Throughput profile (QALL)



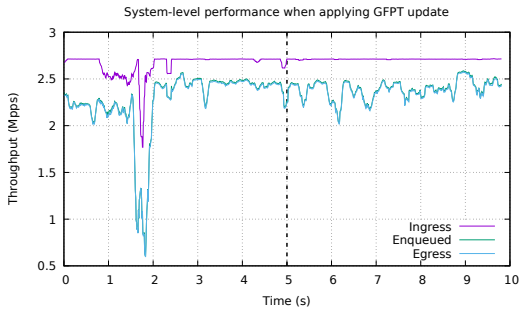
(b) Loss profile (QALL)



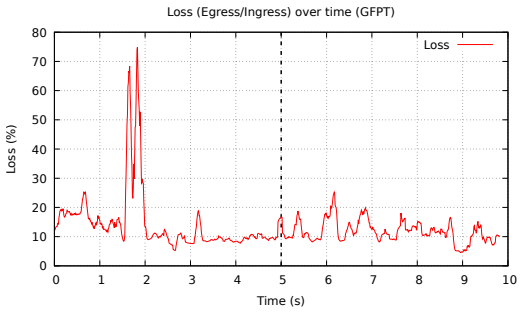
(c) Throughput profile (QMOD)



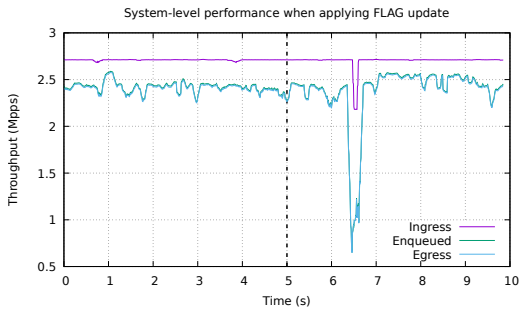
(d) Loss profile (QMOD)



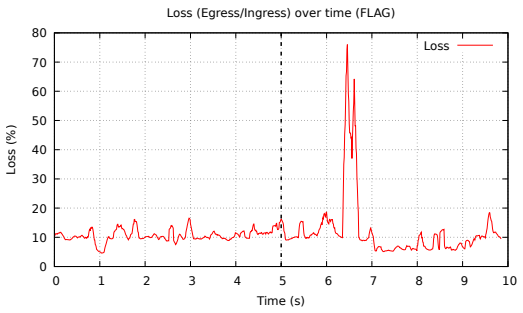
(e) Throughput profile (GFPT)



(f) Loss profile (GFPT)



(g) Throughput profile (FLAG)



(h) Loss profile (FLAG)

Figure 4.3: DPDK testbench output showing packet-level ingress, enqueue, egress, and loss rates. The dashed vertical line marks the point in time when the dynamic update is performed.

5 Discussion

This chapter discusses the results presented in Chapter 4, evaluates the choice of methods from Chapter 3, and reflects on the broader implications of this work. We also discuss design choices that should be considered to meet availability and performance requirements.

5.1 Results

Downtime

The most significant result of this thesis is the large reduction in downtime achieved by all DSU solutions compared to a conventional program update. While a normal restart of the non-DSU adapted DPDK testbench incurs 310 – 430 ms of downtime, all DSU solutions reduce this by several orders of magnitude.

QALL exhibits the highest downtime due to its reliance on total quiescence. Because all workers must stop before the update can proceed, the downtime is dominated by the slowest worker. This behavior aligns with the theory presented in Section 2.4, where synchronization between cores can be a key limiting factor when using quiescence.

QMOD and FLAG improves upon this by introducing per-worker update mechanisms. Whether through forced quiescence or an explicit conditional check, these methods aim to achieve updates that are applied in a modular manner. Pausing each worker one at a time reduces coordination overhead. This not only increases our control during updates but significantly reduces both the average downtime and the variance. In practice, one could also implement logic to perform the update when a worker's packet-receiving queue is empty, effectively using the CPU when it would otherwise have been burning CPU cycles by polling for packets.

GFPT achieves zero downtime by applying updates in such a way that workers are allowed to execute without interruption. The mechanisms, inspired by the GOT, use shared data between workers and the controller to look up function locations. At the time of an update, workers finish any ongoing processing using the old library functions and at their next iteration they load in fresh pointers and perform subsequent processing using the new library functions. This approach proved highly effective for our system with our delimitations, and could even be improved by performing atomic loads less frequently. However, this solution is less expandable than the others because it relies heavily on call-site compatibility.

In GFPT, all workers continuously read from a shared function-pointer table at very high frequency. Our results show that this did not lead to measurable contention in the evaluated system. However, in systems with many more cores or more complex pointer structures, contention on shared cache lines could become a limiting factor.

Performance

The results show that the transition from direct function calls to indirect calls though function pointers introduces a small but measurable performance overhead which is affects all our proposed solutions. The use of a custom indirection mechanism in DSU for indirection, rather than relying on the standard PLT/GOT system, can be motivated if finer control and reduced performance overhead is desire.

The theoretical expectations presented in Section 2.3, where indirection was identified as a source of pipeline stalls, extra pointer-lookup steps, and reduced compiler optimizations, are likely correct. The relatively small overhead observed suggests that carefully designed indirection makes DSU fit for use without significantly impacting program performance. If a system is rebuilt to use shared libraries like us, exporting functions with very small bodies should be avoided and looping should be refactored into the dedicated library functions to minimize the overhead associated with frequent function invocations.

The GFPT solution shows a substantially higher maximal performance overhead compared to the other solutions. This can be attributed to the use of atomic load operations in the worker loop, which introduces additional memory ordering costs. The operations not only constrain compiler optimizations but may also lead to CPU stalls from ordering constraints. This explains the higher cost and the observed shift in Figure 4.2c. It is worth noting that our setup reflects a worst-case pattern, as an atomic load is performed for every function call. In practical systems, atomic loads can occur far less frequently than once per function call.

Another point considered was if atomic loads caused memory congestion in the multi-core system. But this conjecture was ill-founded as an atomic load with acquire semantics is still just a load and our results reflected that read-sharing was not a problem.

The FLAG solution introduces only a simple conditional check in the worker loop, which results in almost no measurable cost because it can be efficiently predicted and pipelined by most modern CPUs. As a results, its performance overhead is nearly identical to QALL and QMOD. The importance of minimizing extra work in the worker's hot loop is critical for Radio Access Network (RAN) systems, which is why this solution was made in an attempt to achieve lower performance overhead than GFPT while simultaneously improving on QMOD by not having to quiesce the worker.

Reliability

From an analytical standpoint, most of the delimitations outlined in Section 1.4 function more as enablers rather than restrictions. By not having to account for ABI changes, state preservation across updates, and call-site incompatibilities, the complexity of each update is greatly reduced. This makes it feasible to reason about the reliability of the system and to validate it without requiring more elaborate techniques. Consequently, no crashes or incorrect behaviors were observed for all solutions presented in Section 3.3.

Although the 30,000-update experiment provides evidence of stability, it does not guarantee complete reliability. Executing updates tens of thousands of times gives confidence that the basic hot-swap mechanisms are stable under repeated use. However, it is important to recognize that this reliability came at the cost of flexibility. The solutions are robust precisely because they avoid the hardest DSU problems.

Architectural impact

The results of this work demonstrate that DSU can substantially reduce the architectural and operational complexity of RAN systems by eliminating the need for replicated services dedicated to takeover during planned updates. By enabling in-place code updates without interrupting execution, DSU removes the orchestration overhead traditionally required to coordinate traffic redirection to nearby radio units with overlapping coverage area. Our minimal-viable implementations show that application-specific DSU can be practical and an effective mechanism for maintaining service continuity. Consequently, DSU offers a compelling alternative to redundancy-driven update strategies, for a cleaner future in RAN software life cycles grounded in continuous operation.

5.2 Method

Combining a realistic testbench with isolated microbenchmarks proved effective for evaluating DSU behavior in a controlled yet representative environment. The DPDK testbench mimics real-world behavior of a RAN worker pipeline, ensuring that results reflect cache behavior, bursty traffic, and tight polling loops. Additionally, TSC-based measurements allowed sufficient precision in timing. Although, several reflections are worth noting:

Limitations

First of all, the system-level data and figures outlining ingress and egress were hard to compare with the baseline. Figures 3.8 and 4.3 provides only limited insight and show no clear separation between solutions or executions. Although all mechanisms process the same workload, system noise introduces fluctuations that obscure meaningful differences and make the figures difficult to interpret.

Furthermore, the CPU and operating system were not configured for low-noise measurements. No cores were isolated to focus exclusively on the testbench program, and CPU frequency scaling remained enabled. These factors allow the OS scheduler and power-management mechanisms to introduce variability in execution timing, adding noise that can mask subtle performance differences between DSU mechanisms.

The workload used in this thesis is synthetic, although DPDK accurately simulates high-rate packet flow and multi-core processing, it does not capture the full complexity of a real RAN stack. The evaluation reflects packet-processing behavior but not the broader timing-critical interactions present in production RAN systems.

All measurements were taken on instruction set architecture x86_64. ARM-based RAN systems may exhibit different behavior, especially regarding atomic operations, register usage and cache hierarchies.

Workers may observe the update flag in short succession. In such cases, the worker responsible for performing the dynamic linking work must complete that task before the others can proceed. The remaining workers cannot update until this linking step finishes, which introduces additional waiting time. This behavior explains why the ideal case, where most workers experience the lower mode of downtime, is not reflected in the results. Instead, more workers tend to fall into the higher-downtime mode. A major improvement to the FLAG solution would be achieved by decoupling dynamic linking work in the same manner as QMOD. In hindsight, this was an oversight on our part.

Supplementary remarks

Because updates are invoked asynchronously, inconsistencies may arise. For example, when the goal is to update an in-use encryption function. Consider that a large payload is being encrypted, naturally it needs to be split up in multiple packets. Then, an update triggered mid-stream can lead to workers applying one encryption algorithm to the first few packets

while another algorithm is applied to the remaining packets. Implications like this is not central to the mechanisms under study, since we destroy packets after use, and will therefore be deferred for consideration in future work.

5.3 The Work in a Wider Context

The findings of this thesis demonstrate that application-specific DSU mechanisms can be highly effective in performance-critical systems such as RAN. Unlike general-purpose DSU frameworks, which must handle arbitrary updates, the solutions developed here exploit the structure of the RAN pipeline: workers are independent, workloads are repetitive and predictable, call-site compatibility is ensured, and state is minimal or externalized. These properties make RAN systems particularly well-suited for lightweight DSU mechanisms.

More broadly, this work contributes to the ongoing discussion in DSU research about the trade-off between flexibility and performance. Many DSU frameworks aim for maximal flexibility, but this often comes at the cost of overhead, complexity, and reduced predictability of post-update correctness. In contrast, this thesis shows that application-specific DSU can achieve near-zero downtime with negligible overhead, provided that update flexibility is intentionally constrained. This suggests a potential direction for future RAN software in which DSU mechanisms are tailored to the architecture and workload of the system rather than relying on conventional program update strategies.

6 Conclusion

This thesis set out to investigate whether Dynamic Software Updating (DSU) can be applied effectively to shared libraries in a multi-core, high-availability environment resembling a Radio Access Network (RAN). By designing, implementing, and evaluating four application-specific DSU solutions (QALL, QMOD, GFPT, and FLAG) this work provides a concrete assessment of trade-offs involved in applying DSU to performance-critical systems. These solutions are minimal viable products. Each solution represents a different point in the design space:

- **QALL** is simple and reliable with zero memory overhead from co-existing libraries but slower than all others.
- **QMOD** offers balance with more control, lower downtime, and retains minimal performance overhead.
- **GFPT** provides perfect availability at the cost of some performance overhead.
- **FLAG** is lightweight with less synchronization.

While the work demonstrates the feasibility of DSU for shared libraries under controlled conditions, it also has clear boundaries. The mechanisms evaluated here do not support state migration, ABI changes, or updates to the main program, all of which remain open challenges. Nevertheless, the ability to update internal library implementations safely and with near-zero downtime represents a meaningful step toward more flexible and resilient RAN software.

Dynamic updates reduce downtime by several orders of magnitude compared to conventional updates. Memory overhead is modest and dominated by the dynamic linking infrastructure. Runtime performance is only minimally affected, with the primary cost incurred by the loss of direct function calls. Overall, DSU seem to provide a substantial availability improvement with negligible trade-offs.

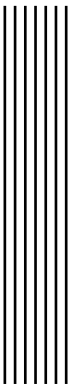
6.1 Future Work

This thesis demonstrated that lightweight DSU mechanisms are effective for hot-swapping shared-library implementations. However, as soon as more complex updates are required, the DSU capabilities of this system will need to be extended.

One possible direction for future work on this system is support for changing function signatures and data structures. Such modifications are frequently required and existing techniques like stub functions could enable controlled type and argument evolution while preserving call-site correctness.

Another promising area is architecture-specific considerations. While our mechanisms were designed to be general, instruction-set architecture details (e.g. x86 versus ARM) influence calling conventions, memory models, and atomic operations. These factors affect both correctness and performance of DSU mechanisms and would need deeper investigation for production-grade portability.

A further challenge is support for updating the main program, which this thesis explicitly excluded. The main program typically owns initialization logic, global state, and long-lived resources while being packaged into a fixed, pre-allocated process address space. Once the binary is compiled and linked, new functionality cannot simply be added at runtime, making it significantly harder to update than a shared library.



Bibliography

[1] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. "A survey of dynamic software updating". In: *Journal of Software: Evolution and Process* 25.5 (2013), pp. 535–568. DOI: 10.1002/smr.1556. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.1556.

[2] Babiker Hussien Ahmed, Sai Peck Lee, Moon Ting Su, and Abubakar Zakari. "Dynamic software updating: a systematic mapping study". In: *IET Software* 14.5 (2020), pp. 468–481. DOI: 10.1049/iet-sen.2019.0201. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-sen.2019.0201.

[3] Robert S Fabry. "How to design a system in which modules can be changed on the fly". In: *Proceedings of the 2nd International Conference on Software Engineering*. 1976, pp. 470–476. URL: https://dl.acm.org/doi/10.5555/800253.807720.

[4] 3rd Generation Partnership Project. *Technical Specification Group Radio Access Network, Packet Data Convergence Protocol (PDCP) Release 19*. Dec. 2025. URL: https://www.3gpp.org/ftp/Specs/archive/38_series/38.323/38323-j10.zip.

[5] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. 9th ed. Hoboken, NJ: Wiley, 2014. ISBN: 9781118093757.

[6] Randy Allen and Ken Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. 1st ed. Morgan Kaufmann, 2002. ISBN: 9781493303540.

[7] x86-psABIs Maintainers. *x86-64 System V Application Binary Interface*. https://gitlab.com/x86-psABIs/x86-64-ABI. 2025.

[8] William Stallings. *Computer Organization and Architecture: Designing for Performance*. 11th ed. Boston: Pearson, 2019. ISBN: 9780135205129.

[9] Filip Strömbäck and Dániel Varró. "Active DSU: Dynamic Software Updates for Active Functions". In: *Proceedings of the 2024 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. 2024, pp. 26–37. DOI: 10.1145/3689492.3690046. URL: https://dl.acm.org/doi/10.1145/3689492.3690046.

[10] Kristis Makris and Rida A. Bazzi. "Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction". In: *Proceedings of the 2009 USENIX Annual Technical Conference*. 2009, p. 31. URL: https://www.usenix.org/conference/usenix-09/immediate-multi-threaded-dynamic-software-updates-using-stack-reconstruction.

[11] Iulian Neamtiu and Michael Hicks. "Safe and timely updates to multi-threaded programs". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009, pp. 13–24. DOI: 10.1145/1542476.1542479. URL: https://dl.acm.org/doi/10.1145/1542476.1542479.

[12] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. "Polus: A powerful live updating system". In: *Proceedings of the 29th International Conference on Software Engineering*. 2007, pp. 271–281. DOI: 10.1109/ICSE.2007.65. URL: https://ieeexplore.ieee.org/document/4222589.

[13] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. "Practical dynamic software updating for C". In: *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2006, pp. 72–83. DOI: 10.1145/1133981.1133991. URL: https://dl.acm.org/doi/10.1145/1133981.1133991.

[14] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. "Mutatis mutandis: Safe and predictable dynamic software updating". In: *ACM Transactions on Programming Languages and Systems* 29.4 (2007), pp. 183–194. DOI: 10.1145/1047659.1040321. URL: https://dl.acm.org/doi/10.1145/1047659.1040321.

[15] Tianxiao Gu, Chun Cao, Chang Xu, Xiaoxing Ma, Linghao Zhang, and Jian Lü. "Low-disruptive dynamic updating of Java applications". In: *Information and Software Technology* 56.9 (2014), pp. 1086–1098. DOI: 10.1016/j.infsof.2014.04.003. URL: https://www.sciencedirect.com/science/article/abs/pii/S0950584914000846.

[16] Suriya Subramanian, Michael Hicks, and Kathryn S McKinley. "Dynamic software updates: a VM-centric approach". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009, pp. 1–12. DOI: 10.1145/1543135.1542478. URL: https://dl.acm.org/doi/10.1145/1543135.1542478.

[17] Dong Kwan Kim, Eli Tilevich, and Calvin J Ribbens. "Dynamic software updates for parallel high-performance applications". In: *Concurrency and Computation: Practice and Experience* 23.4 (2011), pp. 415–434. DOI: 10.1002/cpe.1663. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1663.

[18] Jeff Arnold and M. Frans Kaashoek. "Ksplice: automatic rebootless kernel updates". In: *Proceedings of the 4th ACM European Conference on Computer Systems*. New York, NY, USA, 2009, pp. 187–198. DOI: 10.1145/1519065.1519085. URL: https://dl.acm.org/doi/10.1145/1519065.1519085.

[19] Pablo Tesone, Guillermo Polito, Noury Bouraqadi, Stéphane Ducasse, and Luc Fabresse. "Dynamic Software Update from Development to Production". In: *Journal of Object Technology* 17.1 (2018), pp. 1–36. DOI: 10.5381/jot.2018.17.1.a2. URL: http://www.jot.fm/contents/issue_2018_01/article2.html.

[20] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. "Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates". In: *IEEE Transactions on Software Engineering* 33.12 (2007), pp. 856–868. DOI: 10.1109/TSE.2007.70733. URL: https://ieeexplore.ieee.org/document/4359466.

[21] Michael Hicks and Scott Nettles. “Dynamic software updating”. In: *ACM Transactions on Programming Languages and Systems* 27.6 (2005), pp. 1049–1096. DOI: 10.1145/1108970.1108971. URL: https://dl.acm.org/doi/10.1145/1108970.1108971.

[22] Marcus Karpoff, Jose Nelson Amaral, Kai-Ting Amy Wang, Rayson Ho, and Brice Dobry. “PSU: A Framework for Dynamic Software Updates in Multi-threaded C-Language Programs”. In: *Proceedings of the 32nd International Symposium on Computer Architecture and High Performance Computing*. 2020, pp. 233–240. DOI: 10.1109/SBAC-PAD49847.2020.00040. URL: https://ieeexplore.ieee.org/document/9235074.

[23] Christopher M. Hayden, Karla Saur, Edward K. Smith, Michael Hicks, and Jeffrey S. Foster. “Kitsune: Efficient, General-Purpose Dynamic Software Updating for C”. In: *ACM Transactions on Programming Languages and Systems* 36.4 (2014), pp. 1–38. DOI: 10.1145/2629460. URL: https://dl.acm.org/doi/10.1145/2629460.

[24] Christopher M. Hayden, Edward K. Smith, Eric A. Hardisty, Michael Hicks, and Jeffrey S. Foster. “Evaluating Dynamic Software Update Safety Using Systematic Testing”. In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1340–1354. DOI: 10.1109/TSE.2011.101. URL: https://ieeexplore.ieee.org/document/6035725.

[25] Joe Armstrong. “Erlang”. In: *Communications of the ACM* 53.9 (2010), pp. 68–75. DOI: 10.1145/1810891.1810910. URL: https://dl.acm.org/doi/10.1145/1810891.1810910.

[26] Petr Hosek and Cristian Cadar. “Safe software updates via multi-version execution”. In: *Proceedings of the 35th International Conference on Software Engineering*. 2013, pp. 612–621. DOI: 10.1109/ICSE.2013.6606607. URL: https://ieeexplore.ieee.org/document/6606607.

[27] Luis Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. “Mvedsua: Higher availability dynamic software updates via multi-version execution”. In: *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 573–585. DOI: 10.1145/3297858.3304063. URL: https://dl.acm.org/doi/10.1145/3297858.3304063.

[28] DPDK Project. *Data Plane Development Kit Version 22.03*. https://www.dpdk.org. Accessed: 2026-05-01. URL: https://doc.dpdk.org/api-22.03.

[29] IEEE and The Open Group. *dlfcn.h - The Open Group Base Specifications Issue 8*. https://publications.opengroup.org. Accessed: 2026-05-01. URL: https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/dlfcn.h.html.

[30] IEEE and The Open Group. *stdatomic.h - The Open Group Base Specifications Issue 8*. https://publications.opengroup.org. Accessed: 2026-05-01. URL: https://pubs.opengroup.org/onlinepubs/9799919799/basedefs/stdatomic.h.html.

[31] The strace Developers. *strace: Linux System Call Tracer*. https://strace.io. Accessed: 2026-05-01. URL: https://man7.org/linux/man-pages/man1/strace.1.html.

[32] The Linux Kernel Developers. *perf: Linux Profiling with Performance Counters*. https://perf.wiki.kernel.org. Accessed: 2026-05-01. URL: https://man7.org/linux/man-pages/man1/perf-stat.1.html.

[33] The GNU Binutils Developers. *objdump: Display Information from Object Files*. https://sourceware.org/binutils/docs/binutils/objdump.html. Accessed: 2026-05-01. URL: https://man7.org/linux/man-pages/man1/objdump.1.html.

[34] The GNU Project. *GDB: The GNU Project Debugger*. https://sourceware.org/gdb/. Accessed: 2026-05-01. URL: https://man7.org/linux/man-pages/man1/gdb.1.html.

[35] Matt Mackall. *smem: Memory Reporting Tool*. https://www.selenic.com/smem/. Accessed: 2026-05-01. 2026. URL: https://man7.org/linux/man-pages/man8/smem.8.html.