박사학위논문
Ph.D. Dissertation

바이트 단위 영속성의 원리

Principles of Byte-Addressable Persistency

2024

조경민 (Cho, Kyeongmin)

한국과학기술원

Korea Advanced Institute of Science and Technology

박사학위논문

바이트 단위 영속성의 원리

2024

조경민

한국과학기술원

전산학부

바이트 단위 영속성의 원리

조 경 민

위 논문은 한국과학기술원 박사학위논문으로
학위논문 심사위원회의 심사를 통과하였음

2024년 6월 5일

심사위원장 강 지 훈 (인)

심 사 위 원 권 영 진 (인)

심 사 위 원 허 기 홍 (인)

심 사 위 원 허 재 혁 (인)

심 사 위 원 허 충 길 (인)

Principles of Byte-Addressable Persistency

Kyeongmin Cho

Advisor: Jeehoon Kang

A dissertation submitted to the faculty of
Korea Advanced Institute of Science and Technology in
partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer Science

Daejeon, Korea
June 5, 2024

Approved by

Jeehoon Kang
Professor of School of Computing

The study was conducted in accordance with Code of Research Ethics[1].

[1] Declaration of Ethical Conduct in Research: I, as a graduate student of Korea Advanced Institute of Science and Technology, hereby declare that I have not committed any act that may damage the credibility of my research. This includes, but is not limited to, falsification, thesis written by someone else, distortion of research findings, and plagiarism. I confirm that my thesis contains honest conclusions based on my own careful research under the guidance of my advisor.

DCS 조경민. 바이트 단위 영속성의 원리. 전산학부. 2024년. 136+vii 쪽. 지도교수: 강지훈. (영문 논문)
Kyeongmin Cho. Principles of Byte-Addressable Persistency. School of Computing. 2024. 136+vii pages. Advisor: Jeehoon Kang. (Text in English)

초 록

영속성 메모리는 DRAM의 성능과 SSD의 비휘발성을 결합한 저장 기술이다. 데이터를 바이트 단위로 읽고 저장할 수 있다는 특징을 지니며 이를 통해 저장 시스템의 성능을 향상시킬 수 있다. 그러나 영속성 메모리에서 프로그램을 안전하고 효율적으로 설계하기 위해선 예기치 않은 크래시로 인해 발생하는 다음 두 가지 문제를 해결해야 한다: (1) 명령어 수준에서는 CPU 명령어 순서 바뀜 현상을 올바르게 파악해야 하고 (2) 객체 수준에서는 데이터 손실을 방지하면서 높은 성능을 얻기 위해 자료구조를 신중하게 설계해야 한다.

이 논문은 바이트 단위 영속성을 위한 프로그래밍 원리를 엄밀한 형식으로 제공하여 이러한 문제를 해결한다. 첫째, CPU 명령어 순서 바뀜 현상을 파악하기 위해 주요 아키텍처인 Intel-x86과 Armv8에 대해 하드웨어 영속성 실행의미 모델인 $\text{Px86}_{\text{view}}$과 $\text{PArmv8}_{\text{view}}$를 제공한다. 이를 기반으로 개발한 검사기를 통해 대표적인 영속성 프로그램들의 올바름을 검증한다. 둘째, 데이터 손실 없는 고성능 영속성 자료구조를 설계하기 위한 프로그래밍 프레임워크인 Memento를 제공한다. Memento를 기반으로 설계된 영속성 자료구조는 임의의 크래시에도 안전함이 증명되었고, 기존의 수동 최적화된 영속성 자료구조와 유사한 성능을 보인다.

핵심 낱말 영속성 메모리, 크래시 일관성, 엄밀한 실행의미, 느슨한 영속성, 프로그래밍 모델, 영속성 자료구조

Abstract

Persistent memory (PM) is an emerging storage technology that combines the performance of DRAM with the durability of SSD, offering the benefits of both. A key feature of PM is its byte-addressable persistency, enabling data to be efficiently loaded and stored with byte-level granularity. This capability is crucial for enhancing performance by adapting various data structures and algorithms for DRAM to storage devices. However, designing these approaches in PM presents significant challenges due to the potential for crashes, particularly in addressing two types of persistency challenges: (1) at the instruction level, managing CPU instruction reordering that complicates the understanding of persistent programs; and (2) at the object level, carefully designing and composing data structures to prevent data loss while maintaining high performance.

This dissertation presents systematic programming principles to tackle these challenges through *formal* abstractions for byte-addressable persistency, advancing the reliability and performance of PM systems. (1) To address the challenge of managing CPU instruction reordering, we present $\text{Px86}_{\text{view}}$ and $\text{PArmv8}_{\text{view}}$: hardware persistency semantic models that formally standardize instruction descriptions for major architectures, Intel-x86 and Armv8, in a unified manner using the notion of views. Based on these models, we develop a model checker to verify the correctness of several representative persistent programs. (2) To address the challenge of designing crash-consistent and high-performance data structures, we present Memento: a programming framework for PM that ensures generally applicable to various data structures and algorithms. We prove that data structures designed using Memento are crash-consistent and perform comparably to existing hand-tuned alternatives.

Keywords persistent memory, crash consistency, formal semantics, weak persistency, programming model, durable data structure

Contents

List of Tables

List of Figures

Time,
is what keeps everything from happening at once.

— Ray Cummings

Chapter 1. Introduction

Persistent memory (PM) technology, such as Samsung's recently announced CMM-H [Samsung 2024] and Kioxia's XL-FLASH [Kioxia 2024], simultaneously provides (1) low-latency, high-throughput, and fine-grained data transfer capabilities as DRAM does; and (2) durable and high-capacity storage as SSD does. As such, PM has the potential to radically change the way we build fault-tolerant systems by optimizing traditional and distributed file systems [Kwon et al. 2017; Kadekodi et al. 2021; Chen et al. 2021; Lu et al. 2017; Kim et al. 2021a; Xu and Swanson 2016], transaction processing systems for high-velocity real-time data [Meehan et al. 2015], distributed stream processing systems [Wang et al. 2021], and stateful applications organized as a pipeline of cloud serverless functions interacting with cloud storage systems [Setty et al. 2016; Zhang et al. 2020].

A key characteristic of PM is its fine-grained persistency, which enables read and write operations to directly access individual bytes of persistent data. For example, Intel's PMDK, an open-source PM library, utilizes `mmap()` to map PM to virtual memory, supporting direct memory access [Intel 2023]. Similarly, Samsung's SMDK employs the CXL.mem interface [Samsung 2022] to achieve the same objective. Moreover, both Intel Optane Persistent Memory and Samsung CMM-H's CXL.mem interface transfer data at cache line granularity [Blankenship 2020].

Leveraging the fine-grained access to PM, data structures or algorithms that are traditionally implemented for DRAM can be directly applied to PM, which sets PM apart from traditional coarse-grained (i.e., block-based) storage. For instance, lock-free data structures that employ fine-grained synchronization for concurrency control is a potential candidate for PM-based transaction processing systems. This is because persistent data can be directly modified using atomic operations such as compare-and-swap (CAS) or fetch-and-add, available in modern CPUs [Intel 2024a; Arm 2020]. Moreover, PM can indeed benefit from lock-free mechanisms in two ways: efficiency and resilience. Specifically, (1) Lock-free approaches offer enhanced efficiency by providing greater potential for parallelizing workloads, spreading memory accesses across numerous contention points [David et al. 2018]. For example, the time and space overhead associated with logging—stemming from the central contention point at the tip and the recording of all intermediate changes—can be avoided. (2) They enhance resilience by simplifying crash consistency. The single commit point concept in typical lock-free algorithms maintains consistent data states at every moment, obviating the need for supplementary crash consistency mechanisms, as long as updates to the data structures are promptly flushed to PM. Thanks to these benefits, building lock-free storage systems in PM is a potential alternative to the traditional approach of using locks and logging-based transactions.

Problem However, building PM programs is challenging due to the risk of *crash*es. Crashes can occur for various reasons, such as power failure, hardware failure, or software bugs. In the context of PM, crashes are particularly problematic because caches and registers are volatile, whereas PM is persistent. This discrepancy can lead to inconsistent states in the event of a crash for two reasons: (1) the state of PM may not include all previous writes to the data structure that were still in the cache and not yet flushed into PM; and (2) the state of PM may contain only a portion of the data structure that was being updated at the time of the crash.

To ensure the consistency of PM programs in the event of crashes, developers must meticulously design crash-consistent algorithms. These algorithms must ensure that the program's state remains consistent despite crashes. However, the necessity for crash consistency introduces additional complexity and typically incurs performance overhead. This is because implementing recovery mechanisms, such as logging, checkpoints, or transactions, inherently imposes performance penalties on the system.

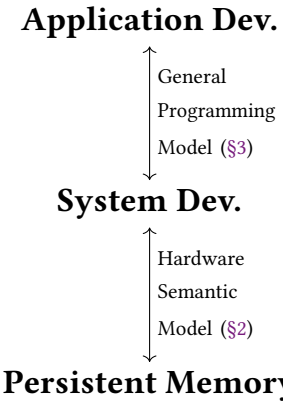


Figure 1.1: Our challenges for building byte-addressable storage systems.

Therefore, to fully harness the potential of PM, we require programming principles that ensure the correctness of programs in the presence of crashes without compromising performance. These challenges encompass various layers, from PM hardware to application development, aiming to develop a comprehensive understanding of the opportunities that PM presents, as illustrated in Fig. 1.1. This dissertation is structured in two main parts, which are briefly summarized below.

Chapter 2: Hardware Semantic Models As a foundational step, we develop hardware semantic models to describe program behaviors in the context of PM. Having such models is essential for system developers to reason about program behaviors effectively and develop reliable programs on the hardware.

However, accurately describing program behaviors on PM is challenging due to the *reordering* of persistency instructions. Specifically, the effects of store instructions may reach PM in an out-of-order fashion in the event of a system crash, a phenomenon known as *relaxed* persistency. To maintain the intended invariants set by developers in a program, it is crucial to properly control this reordering. Fortunately, two major architectures, Intel-x86 and Armv8, provide a flush instruction that acts as a persistency fence to prevent reordering. However, flush is an expensive instruction and should be used only when absolutely necessary.

Thus, we require a semantic model that accurately depicts the hardware's persistency reordering behavior. Recent work has proposed several *persistency models* for mainstream architectures such as Intel-x86 and Armv8, describing the order in which writes are propagated to NVM. However, these models have several limitations; most notably, they either lack operational models or do not support persistent synchronization patterns.

In this chapter, we address the gap by revamping the existing persistency models. Inspired by recent advancements in *view*-based semantics [Kang et al. 2017; Lee et al. 2020; Pulte et al. 2019], we first introduce a *unified operational style* for describing persistency through views. This includes the development of view-based operational persistency models: $\text{Px86}_{\text{view}}$ for Intel-x86 and $\text{PArmv8}_{\text{view}}$ for Armv8, with $\text{PArmv8}_{\text{view}}$ representing the *first* operational model for Armv8 persistency. Next, we propose a *unified axiomatic style* for describing hardware persistency, which enables us to recast and improve the existing axiomatic models of Intel-x86 and Armv8 persistency. We demonstrate that our axiomatic models are equivalent to the authoritative semantics reviewed by Intel and Arm engineers. We prove that each axiomatic hardware persistency model is equivalent to its operational counterpart. Finally, we develop a persistent model checking algorithm and tool, which we use to verify several representative examples.

This chapter heavily builds on the work and writing presented in the following paper:
[Cho et al. 2021b] **Kyeongmin Cho**, Sung-Hwan Lee, Azalea Raad, Jeehoon Kang. *Revamping Hardware*

Persistency Models: View-Based and Axiomatic Persistency Models for Intel-X86 and Armv8. **PLDI 2021.**

Chapter 3: A General Programming Model Despite the well-defined system layer with libraries for PM programming, a new programming approach is required to build *crash-safe* high-level applications. Designing persistent objects that can be composed in a crash-consistent manner is not straightforward. For instance, even if two persistent data structures are crash-consistent, data loss can occur if their operations are sequentially composed. To prevent this, persistent data structures must adhere to a sufficiently robust correctness criterion.

One of the most widely used correctness criteria for persistent concurrent data structures is *detectable recoverability* [Friedman et al. 2018], which ensures both thread safety (correctness in non-crashing concurrent executions) and crash consistency (correctness in crashing executions). However, existing approaches to designing detectably recoverable concurrent data structures are either constrained to simple algorithms or incur high runtime overheads.

In this chapter, we present Memento: a *general* and *high-performance* programming framework for detectably recoverable concurrent data structures in persistent memory (PM). To ensure general applicability across various data structures, Memento supports primitive operations such as checkpoint and compare-and-swap, along with their composition using control constructs. To achieve high performance, Memento employs a timestamp-based recovery strategy that requires fewer writes and flushes to PM compared to existing approaches. We formally prove that Memento ensures detectable recoverability in the event of crashes. To demonstrate Memento, we implement a lock-free stack, list, queue, and hash table, as well as a combining queue, all of which detectably recover from random crashes in stress tests and perform comparably to existing hand-tuned persistent data structures, both with and without detectable recoverability.

This chapter heavily builds on the work and writing presented in the following paper:
[Cho et al. 2023b] **Kyeongmin Cho**, Seungmin Jeon, Azalea Raad, Jeehoon Kang. *Memento: A Framework for Detectable Recoverability in Persistent Memory.* **PLDI 2023.**

Outline The remainder of this dissertation is structured as follows. §2 presents hardware semantic models for persistency in Intel-x86 and Armv8 architectures. §3 presents a general programming model for detectable recoverability in PM. Finally, §4 concludes this dissertation with a summary of our contributions and future work. Additionally, §A and §B provide appendices for §2 and §3, respectively, offering supplementary details including full definitions and proofs.

Chapter 2. Hardware Semantic Models

2.1 Introduction

As discussed in §1, it is widely believed that PM will eventually supplant volatile memory [Pelley et al. 2014], allowing efficient access to persistent data. This belief is backed by industrial support. Specifically, the two major architectures, Intel-x86 and Armv8 which together account for almost 100% of the desktop and mobile market, have extended their official specifications to support persistent programming [Arm 2020; Intel 2024a]. Intel has further released open-source PM libraries such as PMDK [Intel 2024d], and manufactured its own line of PM, Optane DC persistent memory [Intel 2024b], with an extended academic study evaluating its performance [Izraelevitz et al. 2019]. PM is therefore expected to innovate high performance transactional systems [Volos et al. 2011; Liu et al. 2017; Lu et al. 2016; Kolli et al. 2016; Beadle et al. 2020; Hwang et al. 2015] and large-scale memory systems [Oukid et al. 2017; Shan et al. 2017; Lu et al. 2017].

However, building correct transactional systems over persistent memory is difficult in part due to *relaxed* persistency: writes to PM locations may not be persisted to memory in the program order due to micro-architectural optimizations such as out-of-order execution, store buffering, or caching protocols. For instance, consider the programs below:

$$\begin{array}{l}(a)\ \mathsf{Data} \coloneqq 42\\(b)\ \mathsf{Commit} \coloneqq 1\end{array}\quad (\textsc{CommitWeak})\qquad \begin{array}{l}(a)\ \mathsf{Data} \coloneqq 42\\(b)\ \texttt{flush Data}\\(c)\ \mathsf{Commit} \coloneqq 1\end{array}\quad (\textsc{Commit1})$$

Hereafter we assume all program variables in our examples are locations in PM[1] initialized to 0; variable reads and writes are architecture-level load and store instructions, e.g. `mov` on Intel-x86 and `ldr`, `str` in Armv8; and that `flush` represents a persistency fence, e.g. `clflush` on Intel-x86 and `dc.cvap`; `dsb.sy` on Armv8.[2]

In both examples we aim to establish the invariant $I \triangleq \mathsf{Commit}{=}1 \Rightarrow \mathsf{Data}{=}42$ even in case of an unexpected crash. In the case of CommitWeak without a persistency fence, we fail to establish I over mainstream architectures such as Intel-x86 and Armv8: the two stores may persist to PM out of order, thereby allowing $\mathsf{Commit}{=}1, \mathsf{Data}{=}0$ upon crash recovery. By contrast, in the case of Commit1 the persistency fence at b ensures that the two stores persist in the intended (program) order, thereby establishing the invariant I. Micro-architecturally, `flush` Data blocks until the previous store on Data at a is persisted to PM, thus ensuring that the store at c always persists after that of a. As such, persistency fences are expensive and should be used sparingly.

Relaxed persistency is further complicated in multi-threaded settings. Consider the following program with two threads:

$$\begin{array}{l}(a)\ \mathsf{Data} \coloneqq 42\end{array}\ \Big\|\ \begin{array}{ll}(b) & \texttt{if (Data != 0) \{}\\(c) & \quad \texttt{flush Data}\\(d) & \quad \mathsf{Commit} \coloneqq 1\ \texttt{\}}\end{array}\quad (\textsc{Commit2})$$

This example differs from Commit1 in that Data and Commit are written to by different threads. Once again, if the fence at c were removed, the desired invariant I would no longer hold: although the store on Data at a may be propagated (made visible) to the the right thread through cache coherence protocols, it may not be persisted to PM prior to the crash. As before, the fence at c ensures that the store at a (which was propagated to the right thread before c) persists to PM before the store at d, thus establishing I.

[1] As in [Raad et al. 2019b], we assume all locations are durable locations in PM.

[2] Armv8 recently introduced the `dc.cvadp` instruction that, unlike `dc.cvap`, guarantees persistence even in case of battery/hardware failures [Arm 2020]. We focus on `dc.cvap` in this dissertation, but most discussions also apply to `dc.cvadp`.

Note that during normal (non-crashing) executions, under both Intel-x86 and Armv8 no thread can observe the undesirable behavior Commit=1, Data=0 even without the fence at c, underlining the difference between the *consistency order* (the order in which stores are propagated across threads) and the *persistency order* (the order in which stores are persisted to PM). In general, relaxed concurrency models constrain the consistency order, while relaxed persistency models additionally constrain the persistency order, further compounding the complexity of relaxed concurrency.

In order to facilitate correct persistent programming with efficient use of persistency fences, existing work includes several persistency models [Pelley et al. 2014; Chakrabarti et al. 2014; Kolli et al. 2017; Gogte et al. 2018; Raad and Vafeiadis 2018; Raad et al. 2019a,b; Khyzha and Lahav 2021]. However, as we discuss below, these models have several shortcomings.

Problem To our knowledge, no existing persistency model (except for PTSO_{syn} [Khyzha and Lahav 2021], discussed shortly below) satisfies all of the following properties simultaneously:

(A) **Describing mainstream architectures or languages**: For a persistency model to be widely used and applied, it should describe the persistency behavior of mainstream hardware/software platforms such as readily available architectures, e.g. Intel-x86 [Intel 2024a] and Armv8 [Arm 2020], and ubiquitous languages, e.g. C/C++, over which several persistent libraries are implemented [Intel 2024d; Hwang et al. 2015]. Moreover, the model should be sufficiently relaxed that the behaviors observable on existing platforms are also allowed by the model. Otherwise, invariants that hold according to the model would be invalidated by executions on such platforms, rendering the model unsound for reasoning.

(B) **Supporting persistent synchronization patterns**: A persistency model should support common synchronization patterns used in practical implementations of persistent objects, e.g. transactions or file systems. For instance, a model should prohibit undesirable behaviors, e.g. Commit=1, Data=0 in Commit1 and Commit2 that capture the essence of practical implementations of transactional systems. In particular, the model should be sufficiently strict that unobservable behaviors on existing platforms are also forbidden by the model. Otherwise, admitting unobservable behaviors in the model makes it impossible to reason about such patterns. Moreover, a model should serve as an objective correctness criteria for new, more efficient designs of persistent objects, and in doing so, guide such new designs.

(C) **Operational**: An *operational* persistency model is desirable in that it enables stepping through an execution for debugging purposes. Moreover, operational models are more suitable for building high-level reasoning techniques such as program logics. By contrast, axiomatic models constrain the admitted behaviors through a set of axioms over full executions, making them undesirable for step-by-step reasoning (e.g. as in program logics).

The models of Pelley et al. [2014]; Raad and Vafeiadis [2018]; Kolli et al. [2017]; Gogte et al. [2018]; Chakrabarti et al. [2014] do not satisfy (A). Specifically, Pelley et al. [2014]; Kolli et al. [2017]; Gogte et al. [2018]; Chakrabarti et al. [2014] present language-level persistency models put forward as academic *proposals*, and are not supported by mainstream programming languages. Similarly, Raad and Vafeiadis [2018] propose a hardware persistency model, PTSO, by integrating buffered epoch persistency [Pelley et al. 2014] with the TSO architecture of x86/SPARC [Sewell et al. 2010]. However, PTSO is not supported by mainstream architectures of Intel-x86 and Armv8.

The PArmv8 model [Raad et al. 2019a, "PARMv8"] describes the persistency semantics of the Armv8 architecture, but is not operational (C). Moreover, as we discuss shortly in §2.2, the PArmv8 model is too weak

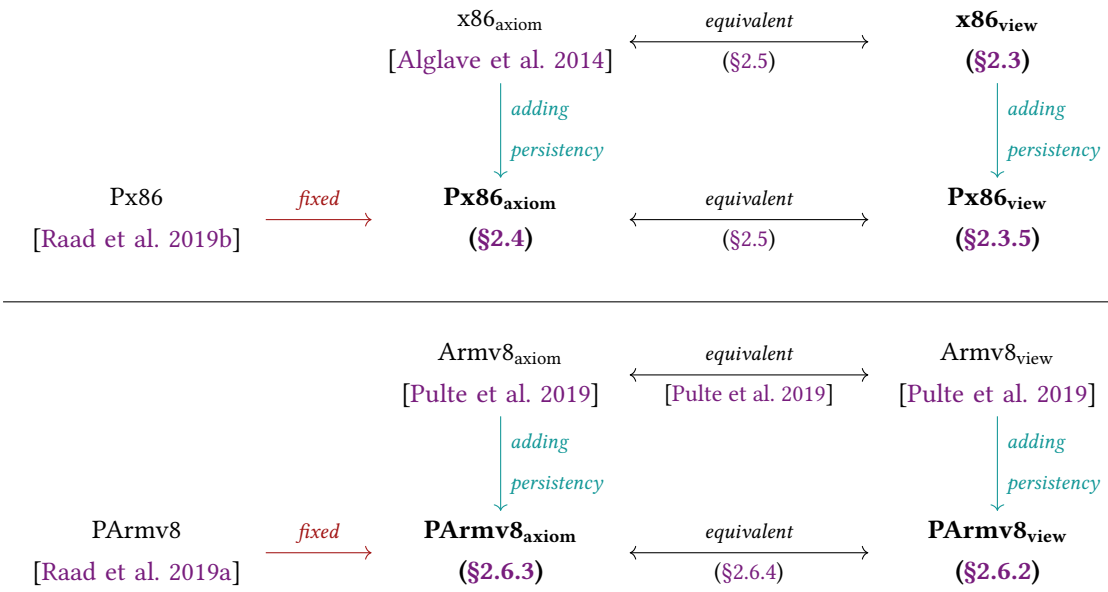


Figure 2.1: Relationship among Intel-x86 and Armv8 models.

in that it violates multi-copy atomicity. Similarly, the Px86 model [Raad et al. 2019b] describes the persistency semantics of the Intel-x86 architecture operationally and axiomatically.[3] However, as we discuss shortly, Px86 is too relaxed and does not always support persistent synchronization patterns in the presence of I/O (B).

Khyzha and Lahav [2021] developed the PTSO_{syn} model for Intel-x86 that fixes the Px86 problem regarding I/O and satisfies (A)–(C). However, they do not discuss this problem as they have a different motivation, i.e. presenting a model that better matches the developers' intuition [Khyzha and Lahav 2021, §1]. We discuss PTSO_{syn} in more detail later in §2.8.

Our Solution, Contributions and Outline We propose a *unified operational style* for describing relaxed persistency using *views*, and develop view-based persistency models of Intel-x86/Armv8 that satisfy all three (A)–(C) properties. In doing so, we develop the *first* operational model for Armv8 persistency. Our operational models highlight 2 flaws in the existing (axiomatic) persistency models of Intel-x86 and Armv8. To remedy this, we develop a *unified axiomatic style* for persistency, adapt the the existing Intel-x86/Armv8 persistency models to our unified style, and repair their flaws.

The remainder of this chapter is organized as follows:

- We discuss the shortcomings of the existing persistency models of Intel-x86/Armv8 and present an intuitive account of our solution as view-based models (§2.2).
- We develop x86_{view}, a new view-based model for Intel-x86 concurrency (§2.3).
- We develop $\text{Px86}_{\text{view}}$ (§2.3.5) and $\text{PArmv8}_{\text{view}}$ (§2.6.2), respectively extending the x86_{view} and $\text{Armv8}_{\text{view}}$ [Pulte et al. 2019] models to account for persistency.
- We present $\text{Px86}_{\text{axiom}}$ (§2.4) and $\text{PArmv8}_{\text{axiom}}$ (§2.6.3), our axiomatic models of Intel-x86 and Armv8 persistency that simplify and repair the state-of-the-art models of the respective architectures [Raad et al. 2019b,a]. We

[3] In [Raad et al. 2019b], the authors introduce two persistency models for Intel-x86: Px86_{man} which formalizes the ambiguous and under-specified behavior described in the Intel reference manual [Intel 2024a], and Px86_{sim} which simplifies and strengthens Px86_{man} to capture the architectural intent envisaged by Intel engineers. In this dissertation we focus on the Px86_{sim} model and simply refer to it as Px86.

prove that our axiomatic models are equivalent to the authoritative semantics reviewed by Intel and Arm engineers, modulo our proposed fixes (§2.4.4 and §2.6.3). Our proposed fix in PArmv8$_{\text{axiom}}$ has been reviewed by Arm engineers.

- We prove that Px86$_{\text{view}}$ and PArmv8$_{\text{view}}$ are equivalent to Px86$_{\text{axiom}}$ and PArmv8$_{\text{axiom}}$, respectively. The equivalence proof is mechanized in Coq (§2.5 and §2.6.4).
- We develop a stateless model checker for persistency and use it to verify several representative examples under PArmv8$_{\text{view}}$ (§2.7). We conclude with related and future work (§2.8).

We present an overview of the concurrency and persistency models we present in this chapter in Fig. 2.1, summarizing their relationship with existing models in the literature.

2.2 Overview

We discuss the shortcomings of Px86 and PArmv8 as regards to (B) (§2.2.1 and §2.2.2). We then present an intuitive account of our key idea to provide a persistency model that satisfies all three desired properties in (A)–(C) simultaneously (§2.2.3).

2.2.1 The Px86 Model and Synchronous Flushes

The Px86 model [Raad et al. 2019b] is too weak in that its instruction for propagating stores to PM behaves *asynchronously*: executing `clflush` under Px86 does not block execution, and merely guarantees that the pending stores on the given location will be persisted to PM at some future point. For instance, if the generic `flush` Data instruction in Commit1 is replaced with its Intel-x86 analogue, `clflush` Data, then once `clflush` Data is executed, there is no guarantee under Px86 that the earlier stores on Data (including that at a) are persisted to PM; rather (1) these stores will be persisted to PM at some future point; and (2) they will be persisted to PM *before all future stores* (including that at c). In other words, the persistency ordering guarantees of `clflush` in (2) allows us to establish the desired invariant I, even though the effect of `clflush` Data may not immediately take place.

The asynchronous behavior of `clflush` is observable in the presence of *external operations* as they narrow down possible crash points through additional observations. For instance, consider the variant of Commit1 below where we replace the store to Commit with an analogous I/O operation that writes "commit" to `file` on disk:

$$\begin{array}{l|l} (a)\ \mathsf{Data} \coloneqq 42 & (d)\ \texttt{if (Flag != 0) \{} \\ (b)\ \texttt{flush Data} & (e)\quad \texttt{log(file,"commit")\}} \quad \textsc{(CommitE)} \\ (c)\ \mathsf{Flag} \coloneqq 1 & \end{array}$$

Let us write C to denote that `file` contains `"commit"`. Under Px86 it is possible to observe the post-crash state $S : \mathsf{Data}{=}0 \wedge C$; i.e. when the I/O operation is executed, the asynchronous effect of `clflush` may not have taken place yet.

As such, to support persistency synchronization patterns such as CommitE in the presence of external operations under Px86, we must strengthen Px86 by modeling the behavior of `clflush` *synchronously*. Let us write SPx86 for a strengthening of Px86 in which `clflush` instructions are executed synchronously, i.e. they block until all pending stores on the location are persisted to PM. In the absence of external operations such as I/O or network messages, the asynchronous behavior of `clflush` cannot be observed, i.e. SPx86 is indistinguishable from Px86. By contrast, in the presence of external operations, only SPx86 satisfies an invariant analogous to that

of Commit1: $C \Rightarrow$ Data=42; i.e. once the I/O operation is executed, the synchronous effect of `clflush` must have taken place and S cannot be observed.

2.2.2 The PArmv8 Model and Multi-Copy Atomicity

The PArmv8 model [Raad et al. 2019a] is too weak in that it violates the principles of *multi-copy atomicity* (MCA) which ensures that a write by one thread is made visible to all other threads simultaneously. Although Armv8 was originally non-MCA, it was recently simplified to observe MCA [Pulte et al. 2017]. However, the persistency extension of Armv8 in PArmv8 violates MCA by allowing the following behavior regarding persistency:

(a) Y $\coloneqq 1$	$\|\|$	(f) X $\coloneqq 1$
(b) `dsb.sy`	$\|\|$	(g) `dsb.sy`
(c) `flushopt x`	$\|\|$	(h) `flushopt y` (FlushMCA)
(d) `dsb.sy`	$\|\|$	(i) `dsb.sy`
(e) Z $\coloneqq 1$	$\|\|$	(j) W $\coloneqq 1$

Executing `flushopt X` persists all pending stores on the same cache line as X *asynchronously*.[4] Moreover, if `flushopt X` is followed by a data synchronization barrier, `dsb.sy`, its effects take place *synchronously*; i.e. executing `dsb.sy` awaits the completion of all earlier `flushopt` by the same thread.

We argue that MCA should preclude the post-crash state $S : \mathsf{Z} = \mathsf{W} = 1 \ \wedge \ \mathsf{X} = \mathsf{Y} = 0$. First, to observe $\mathsf{Z} = \mathsf{W} = 1$ after a crash, the two threads should have fully executed to the end. Second, to observe $\mathsf{Y} = 0$ after a crash, (a) should not have been made visible to (h) prior to the crash, and thus (h) must be ordered before (a). Third, (g) must be ordered before (h) and (a) before (b) because (g) and (b) are fences. Transitively, (g) must be ordered before (h), (a), and then (b). As such, (f) should be visible to (c), thus ensuring $\mathsf{X} = 1$ after the crash and precluding the behavior in S.[5]

To ensure MCA for persistency, we must thus strengthen PArmv8 by enforcing an order between a flush (e.g. c) and a write on the same location that is not persisted by the flush (e.g. f). Let us write SPArmv8 for such a strengthening of PArmv8. Under SPArmv8, if $\mathsf{X} = 0$ after a crash, then (c) is ordered before (f); (a) is ordered before (h); $\mathsf{Y} = 1$ is persisted to the PM; and thus S cannot be observed.

Upon discussing FlushMCA with engineers at Arm, they confirmed that this non-MCA behavior is indeed prohibited and our proposal in §2.6.3 is the correct interpretation of Arm architecture reference manual [Arm 2020].

2.2.3 Our Solution: View-Based Operational Models

We present view-based operational models for the relaxed persistency behavior of Intel-x86/Armv8 architectures that satisfy all three properties in (A)–(C). We build our model over the view-based model of Armv8/RISC-V relaxed-memory *concurrency* [Pulte et al. 2019]. Intuitively, view-based models [Kang et al. 2017; Lee et al. 2020; Pulte et al. 2019] combine two key ideas: (1) recording the entire *store history* in the memory and allowing threads to read old values; and (2) imposing ordering constraints with per-thread *views* representing the set of stores propagated to each thread and thus constraining the outcomes of future loads and stores by a thread. Here, we further introduce the notion of *persistency views* for each location l, denoting the set of stores on l that have persisted to PM and thus will survive a crash.

[4] For the sake of uniformity with our respective Intel-x86 models, we write `flushopt X` in lieu of the Armv8 instruction `dc cvap X`.

[5] The reader may have noted that this behavior is forbidden even if the `dsb.sy` at (b) and (g) are replaced with the weaker `dmb.sy`. We opt for `dsb.sy` to simplify the example by using only one kind of fence.

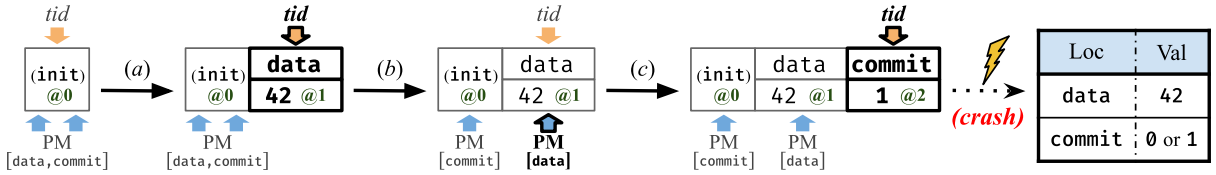


Figure 2.2: A view-based execution of Commit1.

We next illustrate these ideas through a view-based execution of Commit1 in Fig. 2.2, comprising a single thread tid. At each execution stage, the store history is recorded in memory as an indexed (timestamped, e.g. @1) list of stores; the view of tid records (the timestamp of) the latest store propagated to tid (the tid-labelled arrow); and the persistency view of each location l records (the timestamp of) the latest store on l that has persisted to PM (the PM[l]-labelled arrows).

The initial memory is $M = []$, denoting the empty history (no stores have executed), depicted as `init` at timestamp 0 (@0); the tid view is $v = @0$ (no stores have propagated to tid); and the persistency view of each location l is $v_{\text{PM}}[l] = @0$ (no stores on l have persisted to PM). Subsequently:

(a) Executing Data $:=$ 42 appends its store to memory ($M = [\langle \text{Data} := 42\rangle @1]$), and advances the tid view ($v = @1$): the store is executed by and thus propagated to tid.

(b) Executing `flush` Data joins the persistency view of Data with the tid view ($v_{\text{PM}}[\text{Data}] = @1$), ensuring that the latest Data store propagated to tid is persisted to PM.

(c) Analogously, executing Commit $:=$ 1 yields $v = @2$ and $M = [\langle \text{Data} := 42\rangle @1, \langle \text{Commit} := 1\rangle @2]$.

The post-crash outcomes (PM contents) are then determined by the persistency views. Concretely, after a crash each PM location l may contain a value written by a store whose timestamp is at least $v_{\text{PM}}[l]$. For instance, if a crash occurs after executing `flush` Data, then in the post-crash state $v_{\text{PM}}[\text{Data}] = @1$ and thus Data $= 42@1$; i.e. Data $:=$ 42 must have persisted to PM, establishing invariant I.

We next describe an execution of Commit2, where tid_1 and tid_2 denote the left and right threads, respectively. Initially, the memory is $M = []$; the persistency view is $v_{\text{PM}} = \lambda l.@0$; and the tid_i view is $v_i = @0$ (for $i \in \{1, 2\}$). Then:

(a) Executing Data $:=$ 42 yields M=$[\langle \text{Data} := 42\rangle @1]$, v_1=$@1$.

(b) Thread tid_2 may then load Data $= 42$ as its view timestamp ($v_2 = @0$) is less than @1 of Data $:=$ 42. After loading Data $= 42$, the tid_2 view is joined with @1: $v_2 = @1$.

(c) Executing `flush` Data yields $v_{\text{PM}}[\text{Data}] = @1$.

(d) Executing Commit $:=$ 1 results in $M = [\langle \text{Data} := 42\rangle @1, \langle \text{Commit} := 1\rangle @2]$ and $v_2 = @2$.

As with Commit1, the invariant I holds in case of a crash.

Our models indeed satisfy all desired properties. (A) Our models capture the persistency behavior of the mainstream Armv8 and Intel-x86 architectures. Specifically, we prove that our models are equivalent (modulo fixes) to the axiomatic models of Raad et al. [2019a,b] reviewed by Intel/Arm engineers. Our equivalence proof is mechanized in Coq [Coq 2024] and is publicly available [Cho et al. 2021a]. (B) Our models support persistent synchronization patterns such as those of Commit1 and Commit2. (C) Our models are operational as with the existing family of view-based models [Kang et al. 2017; Pulte et al. 2019]. Furthermore, to support reasoning about programs over our models, we develop a stateless model checking algorithm and tool for persistency verification,

$$
\begin{array}{rcl}
p & ::= & s_1 \mid\mid \ldots \mid\mid s_n \\
s \in \mathrm{Stmt} & ::= & \texttt{skip} \mid s_1; s_2 \mid \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \mid \texttt{while } (e)\ s \mid r := e \\
& \mid & r := \texttt{pload}(e) \mid \texttt{store } [e_1]\ e_2 \mid r := \texttt{rmw } rop\ [e] \\
& \mid & \texttt{fence}_f \mid \texttt{flush } e \mid \texttt{flushopt } e \\
rop \in \mathrm{Rmw} & ::= & \texttt{fetch-op } op\ e \mid \texttt{cas } e_1\ e_2 \\
f \in \mathrm{F} & ::= & \texttt{sfence} \mid \texttt{mfence} \\
e \in \mathrm{Expr} & ::= & v \mid r \mid (e_1\ op\ e_2) \qquad op \in \mathrm{O} ::= + \mid - \mid \ldots \\
v \in \mathrm{Val} & = & \mathbb{Z} \qquad r \in \mathrm{VReg} = \mathbb{N} \qquad l \in \mathrm{PLoc} = \mathrm{Val}
\end{array}
$$

Figure 2.3: Intel-x86 concurrency and persistency language.

and use it to verify several representative examples under PArmv8$_{\text{view}}$.[6] Our model checking tool and verified examples are open-source and publicly available [Cho et al. 2021a].

2.3 Px86$_{\text{view}}$: A View-Based Model for Intel-x86 Persistency

We develop Px86$_{\text{view}}$, a view-based model for Intel-x86 persistency. We present a simple language for Intel-x86 concurrency and persistency (§2.3.1) used throughout this section. We develop x86$_{\text{view}}$, a new Intel-x86 concurrency model we use as a baseline (without persistency) and its two key ideas: store histories (§2.3.2) and views (§2.3.3); we describe how we support read-modify-writes (§2.3.4). We then extend x86$_{\text{view}}$ with persistency and develop Px86$_{\text{view}}$ (§2.3.5).

2.3.1 Language for Intel-x86 Persistency

To keep our presentation concrete, we use the language in Fig. 2.3 for Intel-x86 concurrency and persistency. A program p consists of concurrent statements run by distinct threads. A statement s is given by the standard 'while' language over register machines with concurrent memory instructions. The instruction $r := \texttt{pload}(e)$ reads from the (PM) location denoted by expression e and returns it in register r. The $\texttt{store } [e_1]\ e_2$ reads the value denoted by e_2 and stores it at the location denoted by e_1. Analogously, $r := \texttt{rmw } rop\ [e]$ evaluates the expressions rop, performs an RMW ('read-modify-write') operation (e.g. 'compare-and-swap') on the location denoted by e, and returns its old value in r. Finally, $\texttt{fence}_f$ issues a memory 'fence' such as `sfence` or `mfence`; and `flush` e and `flushopt` e persist to PM the pending stores on the cache line containing the location given by e.

2.3.2 The x86$_{\text{view}}$ Model

We present x86$_{\text{view}}$, our view-based operational model for Intel-x86 concurrency, in Fig. 2.4. Our design of x86$_{\text{view}}$ is inspired by Armv8$_{\text{view}}$, a view-based concurrency model of Armv8 [Pulte et al. 2019]. As Intel-x86 concurrency is simpler than that of Armv8, we develop x86$_{\text{view}}$ by removing certain Armv8$_{\text{view}}$ features. Here, we highlight the interesting aspects of Intel-x86 and refer the reader to §A.1 for the full details.

[6] As a proof of concept, we focus on model checking only Armv8 persistency. This is sufficient to showcase the feasibility of model checking for hardware persistency since Armv8 is more complex than Intel-x86 with a bigger search space. We believe it is straightforward to adapt our approach to Intel-x86 persistency, especially given our unified semantic style for persistency.

$$\langle \vec{T}, M \rangle \in \text{Machine} \triangleq (\text{TId} \to \text{Thread}) \times \text{Memory}$$
$$tid \in \text{TId} \triangleq \mathbb{N} \qquad T \in \text{Thread} \triangleq \text{Stmt} \times \text{TState}$$
$$M \in \text{Memory} \triangleq \text{list Msg} \qquad w \in \text{Msg} \triangleq \langle \mathsf{loc} : \text{PLoc}; \mathsf{val} : \text{Val}; \mathsf{tid} : \text{TId} \rangle$$
$$\langle l := v \rangle_{tid} \triangleq \langle \mathsf{loc} = l; \mathsf{val} = v; \mathsf{tid} = tid \rangle \qquad t \in \text{Time} \triangleq \mathbb{N} \qquad v \in \mathbb{V} \triangleq \text{Time}$$
$$ts \in \text{TState} \triangleq \left\langle \begin{array}{l} \sigma : \text{VReg} \to \text{Val}; \\ \mathsf{coh} : \text{PLoc} \to \mathbb{V}; \quad \mathsf{v_{rNew}} : \mathbb{V}; \end{array} \right\rangle$$

(INIT)
$$\frac{p = s_1 \mid\mid \ldots \mid\mid s_n}{\text{init}(p, \langle \lambda tid.\ \langle s_{tid}, \langle \sigma = \lambda\_.\ 0; \mathsf{coh} = \lambda\_.\ @0; \mathsf{v_{rNew}} = @0 \rangle \rangle, [] \rangle)}$$

(MACHINE)
$$\frac{\vec{T}[tid], M \to_{tid} T', M'}{\langle \vec{T}, M \rangle \to \langle \vec{T}[tid \mapsto T'], M' \rangle}$$

(NOT-OVERWRITTEN)
$$\frac{\forall t \in (v_2, v_1].\ M[t].\mathsf{loc} \neq l}{v_1 \sqsubseteq_{M,l} v_2}$$

(STORE)
$$\frac{l = [\![e_1]\!]_{ts.\sigma} \quad v = [\![e_2]\!]_{ts.\sigma} \quad ts.\mathsf{coh}[l] \sqsubseteq t \quad \sqcup_l ts.\mathsf{coh}[l] \sqsubseteq t \quad t = |M| + 1 \quad M' = M +\!\!+ [\langle l := v \rangle_{tid} @ t] \quad ts' = ts[\mathsf{coh}[l] \mapsto t]}{(\texttt{store}\ [e_1]\ e_2, ts), M \to_{tid} (\texttt{skip}, ts'), M'}$$

(LOAD)
$$\frac{l = [\![e]\!]_{ts.\sigma} \quad M[t] = \langle l := v \rangle \quad ts.\mathsf{coh}[l] \sqsubseteq t \quad ts.\mathsf{v_{rNew}} \sqsubseteq_{M,l} t \quad ts' = ts[\sigma[r] \mapsto v, \mathsf{coh}[l] \mapsto t, \mathsf{v_{rNew}} \mapsto_{\sqcup} t \neq ts.\mathsf{coh}[l]\ ?\ t]}{(r := \texttt{pload}(e), ts), M \to_{tid} (\texttt{skip}, ts'), M}$$

(MFENCE)
$$\frac{ts' = ts[\mathsf{v_{rNew}} \mapsto_{\sqcup} \sqcup_l ts.\mathsf{coh}[l]]}{(\texttt{mfence}, ts), M \to_{tid} (\texttt{skip}, ts'), M}$$

Figure 2.4: States and transitions of x86_{view} (excerpt).

States We represent a machine as a pair $\langle \vec{T}, M \rangle$, comprising a thread map $\vec{T}$ and a memory M. A thread map associates each thread with a statement and a thread state. A thread state $ts \in \text{TState}$ consists of a register map, σ, assigning values to registers, and per-thread 'views' (described in §2.3.3). A memory is a list of messages; a message is a triple $\langle l := v \rangle_{tid}$ comprising a memory location (l), a value stored (v), and the id (tid) of the thread storing it. We write $\langle l := v \rangle_{tid} @ t$ to denote that $\langle l := v \rangle_{tid}$ is issued at timestamp (index) t, starting from index @1. For simplicity, we assume a memory contains the initial message $\langle l := 0 \rangle @ 0$ for each l.

Transitions of x86_{view} In the initial state for a program p **(INIT)**, thread statements are those in p; the register maps are $\lambda\_.0$; the views are @0; and the memory is empty ($[]$).

The transitions for control flow and assignment are standard (omitted). The **(MACHINE)** transition of x86_{view} models thread interleaving as in sequential consistency (SC) [Lamport 1979].

Nevertheless, x86_{view} allows relaxed (weaker than SC) behaviors since it records the entire history of stores in its memory as a list of messages, and allows threads to read stale values. Ignoring the colored premises (described later), when executing a store **(STORE)**, a thread determines the location l and the value v, and appends a new message $\langle l := v \rangle$ to the memory. Analogously, when executing $r := \texttt{pload}(e)$ **(LOAD)**, a thread determines the location l, chooses a message $\langle l := v \rangle @ t$ from the memory, and assigns v to r in the register map. Crucially, the chosen message need not be the latest one, thus allowing a stale value to be read. However, the chosen message should not have been overwritten **(NOT-OVERWRITTEN)** from the thread's point of view. We describe the remaining transitions shortly.

Table 2.1: Informal description of concurrency views.

View	*Past*	*Future*
$\mathsf{coh}[l]$	Upper bound of past reads and writes on l	Lower bound of future reads and writes on l
$\mathsf{V}_{\mathsf{rNew}}$	Upper bound of past updates; upper bound of external reads (from other threads)	Lower bound of future reads

Store Buffering Recording stores as messages allows store buffering, a representative relaxed behavior of Intel-x86:

$$\begin{array}{l} (a)\ \mathsf{X} := 1 \\ (b)\ r_1 := \mathsf{y}\ //= 0 \end{array} \Bigg\| \begin{array}{l} (c)\ \mathsf{Y} := 1 \\ (d)\ r_2 := \mathsf{x}\ //= 0 \end{array} \quad \text{(SB)}$$

While the relaxed outcome $r_1 = r_2 = 0$ is prohibited under SC, it is allowed under Intel-x86 and may arise in x86_{view} by: (a) writing $\langle \mathsf{X} := 1 \rangle_{tid_1} @1$; (b) reading $\langle \mathsf{Y} := 0 \rangle @0$; (c) writing $\langle \mathsf{Y} := 1 \rangle_{tid_2} @2$; and most importantly, (d) reading the old value $\langle \mathsf{X} := 0 \rangle @0$ that is overwritten by (a).

2.3.3 Concurrency Views

The model described thus far is too weak in that it allows behaviors prohibited under Intel-x86. We next describe how we strengthen the model to forbid such behaviors through *views*, as summarized in Table 2.1.

Coherence Intel-x86 orders loads and stores on the same location in a single thread as illustrated below:

$$\begin{array}{lll} (a)\ \mathsf{X} := 20 & //= @2 \\ (b)\ \mathsf{X} := 10 & //\neq @1 \\ (c)\ r_1 := \mathsf{X} & //\neq @1 \end{array} \Bigg\| \begin{array}{lll} (d)\ r_2 := \mathsf{Y} & //= @4 \\ (e)\ \mathsf{Y} := 30 & //\neq @3 \\ (f)\ r_3 := \mathsf{Y} & //\neq @3 \end{array} \Bigg\| \cdots \ \text{(Coh)}$$

The first thread issues $\langle \mathsf{X} := 20 \rangle @2$, and then writes to X again and reads from it. Coherence orders (a) before (b) and (c) since they access the same location, thus forbidding them from accessing earlier timestamps, e.g. $@1$. Similarly, the second thread reads the message $\langle \mathsf{Y} := 40 \rangle @4$, and then writes to Y and reads from it again. Coherence orders (d) before (e) and (f) as they access the same location, thus forbidding them from accessing earlier timestamps, e.g. $@3$.[7]

To enforce coherence, we introduce *coherence views* that record past thread behaviors and simultaneously constrain future thread behaviors. Specifically, for each location l, a thread state ts records a coherence view in $ts.\mathsf{coh}[l]$ as a timestamps (initialised to $@0$) representing an index in memory. The $ts.\mathsf{coh}[l]$ represents the maximum (latest) timestamp observed for l by the thread; moreover, it forbids the thread from accessing messages of l with earlier timestamps than $ts.\mathsf{coh}[l]$. Put formally, in **(load)** and **(store)** we additionally require $ts.\mathsf{coh}[l] \sqsubseteq t$ in the premise and update $ts'.\mathsf{coh}[l] \mapsto t$ in the conclusion.

These changes indeed forbid the undesirable behavior in Coh: (a) updates $ts.\mathsf{coh}[\mathsf{X}]$ to $@2$, forbidding (b) and (c) from accessing $@1$. Similarly, (d) updates $ts.\mathsf{coh}[\mathsf{Y}]$ to $@4$, forbidding (e) and (f) from accessing $@3$.

[7] Indeed, coherence between an access and a write is already enforced through **(store)** : stores always append messages to the end of memory. Nevertheless, we explicitly order them with views to achieve (i) uniformity with other coherence orders and (ii) correspondence with $\text{Armv8}_{\text{view}}$, where stores may add messages in places other than the end of memory.

Message Passing In addition to coherence, Intel-x86 orders certain accesses on different locations via 'message passing':

$$\begin{array}{l|l} (a)\ \mathsf{Data} := 42 & (c)\ r_1 := \mathsf{Flag} \quad //=1 \\ (b)\ \mathsf{Flag} := 1 & (d)\ r_2 := \mathtt{data} \quad //\neq 0 \end{array} \quad \text{(MP)}$$

If the right thread reads 1 is from Flag, then it should read 42 from Data as (a) is ordered before (d) as follows:

- (a) **before** (b)**:** A load or store is ordered before later stores. To enforce this, in **(STORE)** we additionally require $\bigsqcup_l ts.\mathsf{coh}[l] \sqsubseteq t$ in the premise.[8]
- (b) **before** (c)**:** A store is ordered before loads that read from it ("message passing"). This is already enforced as the store message read by the load is issued before it.
- (c) **before** (d)**:** A load is ordered before a later load. To enforce this, we introduce the *new-read view*. Specifically, a thread state ts includes a 'new-read' view, $ts.\mathsf{v}_{\mathsf{rNew}}$, recording the maximum (latest) view previously read by the thread. Moreover, it forbids the thread's future loads (on any location) from reading messages that are overwritten by $ts.\mathsf{v}_{\mathsf{rNew}}$. Put formally, in **(LOAD)** we require $ts.\mathsf{v}_{\mathsf{rNew}} \sqsubseteq_{M,l} t$ (i.e. t is not overwritten by $ts.\mathsf{v}_{\mathsf{rNew}}$ in M as far as l is concerned; see **(NOT-OVERWRITTEN)** for details) in the premise and $ts'.\mathsf{v}_{\mathsf{rNew}} \mapsto_{\sqcup} t$ (shorthand for $ts'.\mathsf{v}_{\mathsf{rNew}} = ts.\mathsf{v}_{\mathsf{rNew}} \sqcup t$) in the conclusion.

These changes ensure 'message passing' in MP: (a) the left thread issues $\langle \mathsf{Data} := 42 \rangle @1$, updating $ts_1.\mathsf{coh}[\mathsf{Data}]$ to $@1$; and (b) issues $\langle \mathsf{Flag} := 1 \rangle @2$, updating $ts_1.\mathsf{coh}[\mathsf{Flag}]$ to $@2$; (c) the right thread reads $\langle \mathsf{Flag} := 1 \rangle @2$, updating $ts_2.\mathsf{coh}[\mathsf{Flag}]$ and $ts_2.\mathsf{v}_{\mathsf{rNew}}$ to $@2$; (d) it then cannot read $\langle \mathsf{Data} := 0 \rangle @0$ as $ts_2.\mathsf{v}_{\mathsf{rNew}} = @2 \not\sqsubseteq_{M,\mathsf{Data}} @0$.

Store Buffering with Fences As shown in SB, Intel-x86 may reorder a store and a later load on different locations. If necessary, one can prevent this by inserting fences:

$$\begin{array}{l|l} (a)\ \mathsf{X} := 1 & (d)\ \mathsf{Y} := 1 \\ (b)\ \mathtt{mfence} & (e)\ \mathtt{mfence} \\ (c)\ r_1 := \mathsf{y}\ //=0 & (f)\ r_2 := \mathsf{x}\ //\neq 0 \end{array} \quad \text{(SBFence)}$$

To model this, in the conclusion of **(MFENCE)** we join $ts.\mathsf{v}_{\mathsf{rNew}}$ with $\bigsqcup_l ts.\mathsf{coh}[l]$, thus forbidding store buffering. Without loss of generality, assume $M{=}[\langle \mathsf{X} := 1 \rangle @1, \langle \mathsf{Y} := 1 \rangle @2]$. The right thread then (d) issues $\langle \mathsf{Y} := 1 \rangle @2$, updating $ts_2.\mathsf{coh}[\mathsf{Y}]$ to $@2$; (e) executes `mfence`, updating $ts_2.\mathsf{v}_{\mathsf{rNew}}$ to $@2$; and (f) cannot read $\langle \mathsf{X} := 0 \rangle @0$ as $ts_2.\mathsf{v}_{\mathsf{rNew}}{=}@2 \not\sqsubseteq_{M,\mathsf{X}} @0$.

Forwarding By strengthening $\mathrm{x86}_{\mathrm{view}}$ we have precluded forbidden Intel-x86 behaviors. However, $\mathrm{x86}_{\mathrm{view}}$ is now too strong and must be weakened to allow store forwarding:

$$\begin{array}{l|l} (a)\ \mathsf{X} := 1 & (d)\ \mathsf{Y} := 1 \\ (b)\ r_1 := \mathsf{X}\ //=1 & (e)\ r_3 := \mathsf{Y}\ //=1 \\ (c)\ r_2 := \mathsf{Y}\ //=0 & (f)\ r_4 := \mathsf{X}\ //=0 \end{array} \quad \text{(SBFwd)}$$

While (b) and (c) are ordered, (a) and (c) are not because (b) is forwarded from (a) in the same thread, thus allowing the reordering of (a) after (b) and (c). To model this, in **(LOAD)** the new-read view is joined with the read message's timestamp only if it is written by a different thread. This is denoted by the conditional

[8] The astute reader may have noticed that this condition is stronger than the coherence requirement $ts.\mathsf{coh}[l] \sqsubseteq t$ and thus makes it redundant. Nevertheless, we explicit include the two conditions to emphasize the two requirements, namely coherence and ordering.

(RMW-FAIL)

$$\frac{l = [\![e]\!]_{ts.\sigma} \quad M[t] = \langle l := v \rangle \quad [\![rop]\!]_{ts.\sigma}(v, \bot) \quad ts.\mathsf{coh}[l] \sqsubseteq t \quad ts.\mathsf{v_{rNew}} \sqsubseteq_{M,l} t \quad ts' = ts[\sigma[r] \mapsto v, \mathsf{coh}[l] \mapsto t, \mathsf{v_{rNew}} \mapsto_{\sqcup} t \neq ts.\mathsf{coh}[l] \ ?\ t]}{(r := \texttt{rmw}\ rop\ [e], ts), M \to_{tid} (\texttt{skip}, ts'), M}$$

(RMW)

$$\frac{l = [\![e]\!]_{ts.\sigma} \quad M[t_1] = \langle l := v_1 \rangle \quad [\![rop]\!]_{ts.\sigma}(v_1, v_2) \quad t_2 = |M| + 1 \quad M' = M +\!\!+ [\langle l := v_2 \rangle_{tid} @ t_2] \quad t_2 - 1 \sqsubseteq_{M,l} t_1 \quad ts.\mathsf{coh}[l] \sqsubseteq t_1, t_2 \quad ts.\mathsf{v_{rNew}} \sqsubseteq_{M,l} t_1 \quad \sqcup_l ts.\mathsf{coh}[l] \sqsubseteq t_2 \quad ts' = ts[\sigma[r] \mapsto v_1, \mathsf{coh}[l] \mapsto t_2, \mathsf{v_{rNew}} \mapsto_{\sqcup} \sqcup_l ts.\mathsf{coh}[l] \sqcup t_2]}{(r := \texttt{rmw}\ rop\ [e], ts), M \to_{tid} (\texttt{skip}, ts'), M'}$$

Figure 2.5: RMW transitions of x86_{view}.

notation $ts'.\mathsf{v_{rNew}} \mapsto_{\sqcup} t \neq ts.\mathsf{coh}[l] \ ?\ t$, stating that if $t \neq ts.\mathsf{coh}[l]$, then $ts'.\mathsf{v_{rNew}} \mapsto_{\sqcup} t$; otherwise, $ts'.\mathsf{v_{rNew}}$ is left unchanged. These changes then admit the behavior in SBFwd. Without loss of generality, assume $M = [\langle \mathsf{X} := 1 \rangle @ 1, \langle \mathsf{Y} := 1 \rangle @ 2]$. The right thread ($d$) writes $\langle \mathsf{Y} := 1 \rangle @ 2$, updating $ts_2.\mathsf{coh}[\mathsf{Y}]$ to @2; (e) reads $\langle \mathsf{Y} := 1 \rangle @ 2$, without updating $ts_2.\mathsf{v_{rNew}}$ thanks to forwarding; and (f) reads $\langle \mathsf{X} := 0 \rangle @ 0$ as $ts_2.\mathsf{v_{rNew}} = @0 \sqsubseteq_{M,\mathsf{X}} @0$.

2.3.4 Supporting Read-Modify-Writes (RMW)

The RMW transitions (Fig. 2.5) are obtained by combining the transitions of loads, stores and mfences. A failed RMW **(RMW-FAIL)** degenerates to a load;[9] if an RMW fails, then $[\![rop]\!]_{ts.\sigma}(v_1, \bot)$ holds (e.g. $[\![\mathsf{cas}\ 4\ 5]\!]_{rmap}(3, \bot)$ but not $[\![\mathsf{cas}\ 4\ 5]\!]_{rmap}(4, \bot)$[10]). A successful RMW **(RMW)** atomically reads from and writes to a location; if an RMW succeeds, then $[\![rop]\!]_{ts.\sigma}(v_1, v_2)$ holds (e.g. $[\![\mathsf{cas}\ 3\ 5]\!]_{rmap}(3, 5)$ or $[\![\texttt{fetch-add}\ 1]\!]_{rmap}(4, 5)$). Moreover, atomicity requires that there be no intervening messages on the same location between those read and written by the RMW; i.e. $t_2 - 1 \sqsubseteq_{M,l} t_1$. Lastly, as with mfences, we join $ts.\mathsf{v_{rNew}}$ with $\bigsqcup_l ts.\mathsf{coh}'[l]$.

As we show in §2.5, our x86_{view} model is *equivalent* to the authoritative axiomatic model reviewed by Intel engineers.

2.3.5 Persistency Views

We next develop $\text{Px86}_{\text{view}}$ by extending x86_{view} with persistency. As discussed in §2.2.3, the key idea is *persistency views*, determining persisted messages as summarized in Table 2.2.

Synchronous Flush As shown in Fig. 2.6, in order to model the behaviour of `flush` instructions synchronously, we extend a thread state ts with a persistency view, $ts.\mathsf{v_{pCommit}}$. For each location l, the $ts.\mathsf{v_{pCommit}}[l]$ denotes the maximum view (timestamp) of the messages on l that have persisted to PM. Executing a `flush` **(FLUSH)** determines the location l, and for each location l' on the same cache line as l, joins $ts.\mathsf{v_{pCommit}}[l']$ with the maximum coherence view v, thus persisting those messages of l' propagated to the thread (i.e. all earlier writes on l'). (The *asynchronous persistency view*, $ts.\mathsf{v_{pAsync}}[l']$, will be described shortly.) After a crash **(CRASH)**, the

[9] The semantics of failed RMWs in Intel-x86 is not fully agreed upon in the literature. Our model assumes a failed RMW to degenerate to a load; an alternative model may additionally assume that failed RMWs execute a memory fence. Nevertheless, we can straightforwardly adapt our model to support this by extending **(RMW-FAIL)** with the effects of **(MFENCE)**.

[10] Here we assume compare-and-swaps are strong: they do not fail spuriously. In our Coq formalization, we also support weak compare-and-swaps.

Table 2.2: Informal description of persistency views.

View	*Past*	*Future*
$\mathsf{v}_{\mathsf{pReady}}$	Upper bound of past external reads (from other threads)	Lower bound of messages to be asynchronously flushed by future `flushopt`
$\mathsf{v}_{\mathsf{pAsync}}[l]$	Upper bound of past `flush`/`flushopt` on the same cache line as l	Lower bound of messages on l to be persisted by future fences/updates
$\mathsf{v}_{\mathsf{pCommit}}[l]$	Upper bound of past (1) `flush` l'; and (2) `flushopt` l' followed by fences/updates, where l' is on the same cache line as l	Lower bound of persisted messages on l to survive a crash

contents of PM, SM ('sequential memory'), satisfy the following condition for each location l: $SM[l]$ holds the value of some message on l whose timestamp t is not overwritten by any thread's persistency view on l.

This indeed establishes the invariant $I \triangleq \mathsf{Commit}{=}1 \Rightarrow \mathsf{Data}{=}42$ for Commit2. After executing the left thread, $M = [\langle \mathsf{Data} := 42\rangle@1]$. The right thread ($b$) reads $\langle \mathsf{Data} := 42\rangle@1$, updating $ts_2.\mathsf{coh}[\mathsf{Data}]$ and $ts_2.\mathsf{v}_{\mathsf{rNew}}$ to $@1$; (c) persists the message $ts_2.\mathsf{coh}[\mathsf{Data}] = @1$, updating $ts_2.\mathsf{v}_{\mathsf{pCommit}}[\mathsf{Data}]$ to $@1$; and (d) writes $\langle \mathsf{Commit} := 1\rangle@2$. After a crash, if $\langle \mathsf{Commit} := 1\rangle@2$ has persisted, then (d) must have been executed; therefore $ts_2.\mathsf{v}_{\mathsf{pCommit}}[\mathsf{Data}] = @1$ and $\mathsf{Data} = 42$.

Asynchronous Flush `flushopt` is a weaker variant of `flush` that may be reordered after certain instructions, and thus its execution may be delayed until a later fence/RMW. This may improve performance when persisting multiple locations:

(a) Data1 := 42	(c) `if (Data2 != 0) {`
(b) Data2 := 7	(d) `flushopt Data1`
	(e) `flushopt Data2` (CommitOpt)
	(f) `sfence`
	(g) Commit := 1 }

Similarly to Commit2, the invariant $I' \triangleq \mathsf{Commit}{=}1 \Rightarrow \mathsf{Data1}{=}42 \wedge \mathsf{Data2}{=}7$ always holds. The `sfence` (f) awaits the completion of both (d) and (e), reducing I/O latency.

To model `flushopt` instructions, we extend a thread state ts with (1) $ts.\mathsf{v}_{\mathsf{pReady}}$, denoting the view to be persisted asynchronously at a subsequent `flushopt`; and (2) $ts.\mathsf{v}_{\mathsf{pAsync}}[l]$, denoting the maximum view of messages on l that have been persisted asynchronously.

The additional transitions of $\text{Px86}_{\text{view}}$ are given in Fig. 2.6. Executing `flushopt` l **(FLUSHOPT)** or `flush` l **(FLUSH)** joins, for each location l' on the same cache line as l, $ts.\mathsf{v}_{\mathsf{pAsync}}[l']$ with $ts.\mathsf{v}_{\mathsf{pReady}}$ and the maximum coherence view, v, of the cache line. Executing a fence in **(MFENCE)** and **(SFENCE)**, or a successful RMW in **(RMW)**, joins $ts.\mathsf{v}_{\mathsf{pCommit}}$ with $ts.\mathsf{v}_{\mathsf{pAsync}}$ and $ts.\mathsf{v}_{\mathsf{pReady}}$ with $\bigsqcup_l ts.\mathsf{coh}[l]$. Executing a load or a failed RMW in **(LOAD)** and **(RMW-FAIL)** joins $ts.\mathsf{v}_{\mathsf{pReady}}$ with the read message's timestamp unless forwarded.

This allows us to establish I' for CommitOpt. Without loss of generality, let $M = [\langle \mathsf{Data1} := 42\rangle@1, \langle \mathsf{Data2} := 7\rangle@2]$. The right thread then (c) reads $\langle \mathsf{Data2} := 7\rangle@2$, updating $ts_2.\mathsf{coh}[\mathsf{Data2}]$, $ts_2.\mathsf{v}_{\mathsf{rNew}}$ and $ts_2.\mathsf{v}_{\mathsf{pReady}}$ to $@2$; (d, e) asynchronously persists $ts_2.\mathsf{v}_{\mathsf{pReady}} = @2$ to Data1 and Data2,

$$ts \in \mathrm{TState} \triangleq \left\langle \ldots;\quad \mathsf{v_{pReady}} : \mathbb{V};\quad \mathsf{v_{pAsync}}, \mathsf{v_{pCommit}} : \mathrm{PLoc} \to \mathbb{V};\right\rangle$$

(FLUSH)

$$\frac{l = [\![e]\!]_{ts.\sigma} \quad v = \sqcup_{l'}\, ts.\mathsf{coh}[l'] \qquad ts' = ts[\mathsf{v_{pAsync}} \mapsto_\sqcup \lambda l'.\ cl(l,l')\ ?\ v, \mathsf{v_{pCommit}} \mapsto_\sqcup \lambda l'.\ cl(l,l')\ ?\ v]}{(\mathtt{flush}\ e, ts), M \to_{tid} (\mathtt{skip}, ts'), M}$$

(FLUSHOPT)

$$\frac{l = [\![e]\!]_{ts.\sigma} \quad v = \sqcup_{l'}\ cl(l,l')\ ?\ ts.\mathsf{coh}[l'] \qquad ts' = ts[\mathsf{v_{pAsync}} \mapsto_\sqcup \lambda l'.\ cl(l,l')\ ?\ (v \sqcup ts.\mathsf{v_{pReady}})]}{(\mathtt{flushopt}\ e, ts), M \to_{tid} (\mathtt{skip}, ts'), M}$$

(SFENCE)

$$\frac{ts' = ts[\mathsf{v_{pReady}} \mapsto_\sqcup \sqcup_l\, ts.\mathsf{coh}[l], \mathsf{v_{pCommit}} \mapsto_\sqcup ts.\mathsf{v_{pAsync}}]}{(\mathtt{sfence}, ts), M \to_{tid} (\mathtt{skip}, ts'), M}$$

(LOAD)

$$\frac{\cdots\ ts' = ts[\cdots, \mathsf{v_{pReady}} \mapsto_\sqcup t \neq ts.\mathsf{coh}[l]\ ?\ t]}{(r := \mathtt{pload}(e), ts), M \to_{tid} (\mathtt{skip}, ts'), M}$$

(CRASH)

$$\frac{\forall l.\ \exists t.\ M[t] = \langle l := SM[l]\rangle\ \wedge\ \forall (\_, ts) \in \vec{T}.\ ts.\mathsf{v_{pCommit}}[l] \sqsubseteq_{M,l} t}{\langle \vec{T}, M\rangle \to_{\mathsf{crash}} SM}$$

Figure 2.6: States and transitions of $\mathrm{Px86}_{\mathrm{view}}$ where the highlighted rule denotes the *extension* of **LOAD** transition from Fig. 2.4 as shown; the premises of **MFENCE**, **RMW** and **RMW-FAIL** are analogously extended and omitted here.

updating $ts_2.\mathsf{v_{pAsync}}[\mathsf{Data1}]$, $ts_2.\mathsf{v_{pAsync}}[\mathsf{Data2}]$ to @2; (f) awaits the completion of (d) and (e), updating $ts_2.\mathsf{v_{pCommit}}[\mathsf{Data1}]$ and $ts_2.\mathsf{v_{pCommit}}[\mathsf{Data2}]$ to @2; (g) writes $\langle \mathsf{Commit} := 1\rangle$@3. After a crash, if $\langle \mathsf{Commit} := 1\rangle$@3 is persisted, then (g) must have been executed; $ts_2.\mathsf{v_{pCommit}} = [\mathsf{Data1} \mapsto @2, \mathsf{Data2} \mapsto @2]$; and thus $\mathsf{Data1} = 42$ and $\mathsf{Data2} = 7$.

The resulting model, $\mathrm{Px86}_{\mathrm{view}}$, is proven equivalent to the authoritative axiomatic model reviewed by Intel engineers [Raad et al. 2019b] (modulo the fix discussed in §2.2.1 – see §2.5).

2.4 Fixing and Simplifying the Px86 Model

We present $\mathrm{Px86}_{\mathrm{axiom}}$, a new axiomatic model for Intel-x86 persistency that simplifies Px86 [Raad et al. 2019b] and fixes its flaws discussed in §2.2.1. We present a short background on axiomatic models (§2.4.1); describe the baseline axiomatic model for Intel-x86 concurrency (§2.4.2); extend it to persistency and present $\mathrm{Px86}_{\mathrm{axiom}}$ (§2.4.3); and compare $\mathrm{Px86}_{\mathrm{axiom}}$ with Px86, proving their equivalence modulo our fixes in $\mathrm{Px86}_{\mathrm{axiom}}$ (§2.4.4).

2.4.1 Background on Axiomatic Models

Executions and Events In the literature of axiomatic (a.k.a. declarative) memory models, the traces of shared memory accesses of a program are represented as a set of *executions*, where each execution G is a graph comprising: (i) a set of events (graph nodes); and (ii) a number of relations on events (graph edges). We typically use a, b and e to range over events. Each event captures the execution of a primitive command (e.g. a load) and is a triple of the form $e{=}(n, \mathsf{tid}, l)$, where $n{\in}\mathbb{N}$ is the (unique) *event identifier*; $\mathsf{tid}{\in}\mathrm{TId}$ identifies the executing thread; and $l{\in}\mathrm{Lab}$ is the event *label*. Event labels are defined by the underlying memory model; for Intel-x86 a label l may be (1) $(\mathtt{R}, x, v)$ for reading (loading) value v from location x; (2) $(\mathtt{W}, x, v)$ for writing (storing) value v to location

$$\text{obs} = \text{co} \cup \text{rfe} \cup \text{fre}$$

$$\text{dob} = ([W \cup U \cup R]; \text{po}; [W \cup U \cup R]) \setminus (W \times R)$$

$$\text{bob} = [W \cup U \cup R]; \text{po}; [MF]; \text{po}; [W \cup U \cup R]$$

$$\text{ob} = \text{obs} \cup \text{dob} \cup \text{bob}$$

$(\text{rf}; \text{po}^?)$ irreflexive (CO-RW)

$(\text{fr}; \text{po})$ irreflexive (CO-WR)

ob acyclic (EXTERNAL)

Figure 2.7: The $\text{x86}_{\text{axiom}}$ model [Alglave et al. 2014].

x; (3) $(\texttt{U}, x, v, v')$ for a successful update (RMW) modifying x to v' when its value matches v; (4) $\texttt{MF}$ for executing an `mfence`. The functions `loc`, `rval` and `wval` respectively project the location, the read value and the written value of a label, where applicable. For instance, $\texttt{loc}(l){=}x$ and $\texttt{wval}(l){=}v$ for $l{=}(\texttt{W}, x, v)$. The functions `thrd` and `lab` respectively project the thread identifier and the label of an event.

Notation Given a relation r on a set A, we write $\text{r}^?$ and r^+ for the reflexive and transitive closures of r, respectively. We write r^{-1} for the inverse of r; $[A]$ for the identity relation on A, i.e. $\{(a, a) \mid a \in A\}$; and A_x for $\{a \in A \mid \texttt{loc}(a){=}x\}$. We write ri for the internal subset of r (on events of the same thread), i.e. $\text{ri} \triangleq \{(a, b) \in \text{r} \mid \texttt{thrd}(a){=}\texttt{thrd}(b)\}$; and re for the external subset of r (on events of different threads). Finally, we write $\text{r}_1; \text{r}_2$ for the relational composition of r_1 and r_2, i.e. $\{(a, b) \mid \exists c.\ (a, c) \in \text{r}_1 \wedge (c, b) \in \text{r}_2\}$.

DEFINITION 2.4.1 (EXECUTIONS). *An* execution, *G, is a tuple of the form $(E, \text{po}, \text{rf}, \text{co})$, where:*

- *E is a set of events, including a set of* initialisation *events, $I \subseteq E$, comprising a single write event with label $(\texttt{W}, x, 0)$ for each $x \in \text{PLoc}$. The set of* read *events in E is: $R \triangleq \{e \in E \mid \exists x, v.\ \texttt{lab}(e){=}(\texttt{R}, x, v)\}$; the sets of* writes *($W$),* RMW *($U$) and* memory fence *($MF$) events are analogous.*
- *$\text{po} \subseteq E \times E$ denotes the* 'program-order' relation, *defined as a disjoint union of strict total orders, each ordering the events of one thread, together with $I \times (E \setminus I)$ that orders initialisation events before all others.*
- *$\text{rf} \subseteq (W \cup U) \times (R \cup U)$ denotes the* 'reads-from' relation *on events of the same location with matching values; i.e. $(a, b) \in \text{rf} \Rightarrow \texttt{loc}(a){=}\texttt{loc}(b) \wedge \texttt{wval}(a){=}\texttt{rval}(b)$. Moreover, rf is total and functional on its range, i.e. every read/update is related to exactly one write/update. A read/update may be rf-related to an initialisation write.*
- *$\text{co} \subseteq E \times E$ is the* 'coherence-order', *defined as the disjoint union of relations $\{\text{co}_x\}_{x \in \text{PLoc}}$, such that each co_x is a strict total order on $W_x \cup U_x$ and $I_x \times ((W_x \cup U_x) \setminus I) \subseteq \text{co}_x$.*

In the context of an execution graph $(E, \text{po}, \text{rf}, \text{co})$, we define the *'from-reads' relation* as $\text{fr} \triangleq \text{rf}^{-1}; \text{co}$. Note that in this initial stage, executions are unrestricted: there are few constraints on rf and co. Such restrictions are determined by the set of model-specific *consistent* executions. We next define execution consistency for several models.

$$
\begin{aligned}
& \text{(axioms of x86}_{\text{axiom}}\text{ (Fig. 2.7))} \\
\text{fob} \;&=\; [W \cup U \cup R]; \text{po}; [FL] \\
&\cup\; ([U \cup R] \cup ([W]; \text{po}; [MF \cup SF])); \text{po}; [FO] \\
&\cup\; [W]; (\text{po}; [FL])^{?}; (\text{po} \cap \text{CL}); [FO] \\
\text{ob} \;&=\; \text{obs} \cup \text{dob} \cup \text{bob} \cup \text{fob} \cup \text{pf} \cup \text{fp} && (\textit{redefined}) \\
\text{pf} \;&\subseteq\; (\text{obs} \cup \text{dob} \cup \text{bob} \cup \text{fob} \cup \text{fp})^{+} && (\textsc{pf-min}) \\
P \;&=\; dom(\text{pf}; ([FL] \cup ([FO]; \text{po}; [MF \cup SF \cup U]))) \\
\forall l.\, \exists w.\, SM(l) = \texttt{wval}(w) \;\wedge\; &(P \times \{w\}) \cap \text{Loc} \subseteq \text{co}^{?} && (\textsc{persist})
\end{aligned}
$$

Figure 2.8: The Px86$_{\text{axiom}}$ model.

2.4.2 The x86$_{\text{axiom}}$ Model [Alglave et al. 2014]

As the baseline axiomatic model for Intel-x86, we use that of Alglave et. al. [Alglave et al. 2014], presented in Fig. 2.7, which we refer to as x86$_{\text{axiom}}$.[11] We choose x86$_{\text{axiom}}$ as the baseline as it is stylistically similar with Armv8$_{\text{axiom}}$ [Pulte et al. 2019], thus allowing a more uniform treatment of Intel-x86 and Armv8 persistency.

The co-rw ('coherence-read-write') axiom requires that loads not read from later stores; co-wr ('coherence-write-read') ensures that loads do not read values overwritten by earlier stores; and external, ensures that externally visible events can be linearized with respect to the *'ordered-before'* relation (ob). The existence of such a globally-agreed order of events makes x86$_{\text{axiom}}$ multi-copy-atomic.

The ob relation enforces the order (a, b) if: (1) a is a store overwritten by b (co); (2) a is a store read by b in a different thread (rfe); (3) a reads a value overwritten by b in a different thread (fre); (4) a, b are accesses by the same thread and $(a, b) \notin W \times R$ (dob); or (5) a, b are accesses by the same thread and are separated by a fence (bob).

Coherence between two writes (resp. two reads) is derived from the axioms. Specifically, $\text{co} \cup ([W \cup U]; \text{po}; [W \cup U]) \subseteq \text{ob}$ and acyclicity of ob ensure irreflexivity of co; po. Similarly, $(\text{fre}; \text{rfe}) \cup (\text{fri}; \text{rfi}) \cup ([U \cup R]; \text{po}; [U \cup R]) \subseteq \text{ob}$ and acyclicity of ob ensure irreflexivity of fr; rf; po.

2.4.3 The Px86$_{\text{axiom}}$ Model

We extend x86$_{\text{axiom}}$ with persistency semantics and develop the Px86$_{\text{axiom}}$ model as presented in Fig. 2.8. We first define:

- FL and FO: the set of synchronous flush (`flush`) and asynchronous flush (`flushopt`) events, respectively;
- SF: the set of `sfence` events;
- $\text{pf} \subseteq (W \cup U) \times (FL \cup FO)$: the *'persists-from'* relation, relating each flush to the co-latest store for each location persisted by the flush. This is analogous to the rf relation; however, while rf relates a load to a *single* store, pf may relate a flush to multiple stores (one for each location) on the same cache line.
- $\text{fp} \triangleq \text{pf}^{-1}; \text{co}$: the *'from-persists'* relation (analogous to fr), relating a flush to co-later stores (*cf.* $\text{fr} \triangleq \text{rf}^{-1}; \text{co}$).

[11] For clarity, we rename the relations and axioms in [Alglave et al. 2014] to highlight its similarity with the axiomatic model for Armv8 concurrency [Pulte et al. 2019].

The ob relation is extended with fob ('flush-ordered-before'), ordering earlier events and a later flush as per the Intel manual [Intel 2024a]. Furthermore, pf and fp are included in ob for the same reason rf and fr are; i.e. because Intel-x86 is multi-copy atomic. P denotes the set of writes that must be persisted, i.e. those writes that are persisted by a synchronous (FL) or an asynchronous flush (FO) followed by a fence ($MF \cup SF \cup U$).[12] The persist axiom states that in case of a crash, the persisted value (in PM) of each location l in $SM[l]$ should not be coherence-before the writes in P. For simplicity, the PF-MIN axiom ensures that P is minimal, i.e. a flush persists only those writes that are strictly ordered before it. However, this minimality axiom is optional (Theorem 2.4.2).

LEMMA 2.4.2. *A behavior is allowed under $Px86_{axiom}$ with axiom PF-MIN iff it is allowed under $Px86_{axiom}$ without PF-MIN.*

PROOF. The proof is given in §A.3. □

2.4.4 Comparing $Px86_{axiom}$ to Px86 in [Raad et al. 2019b]

Fix Our $Px86_{axiom}$ model indeed fixes the Px86 shortcomings described in §2.2.1. In particular, as discussed in §2.2.1, we first strengthen Px86 to SPx86 by additionally requiring that flush instructions behave synchronously – see Fig. A.8 and Fig. A.9 for the definitions of Px86 and SPx86.[13] In Theorem 2.4.3 below we then prove that $Px86_{axiom}$ and SPx86 are equivalent.

THEOREM 2.4.3. *A behavior is allowed under SPx86 iff it is allowed under $Px86_{axiom}$.*

PROOF. The proof is given in §A.4. □

The Px86 and SPx86 models are based on the axiomatic Intel-x86 model known as TSO [Owens et al. 2009; Sewell et al. 2010], henceforth referred to as $x86_{man}$ (given in Fig. A.8). As such, in order to prove Theorem 2.4.3 we first show that $x86_{man}$ and $x86_{axiom}$ are equivalent. In particular, existing equivalence results between $x86_{man}$ and $x86_{axiom}$ cover loads and stores only and not RMWs and fences [Alglave 2012]. We extend this result for the first time to cover RMWs and fences in Theorem 2.4.4 below.

THEOREM 2.4.4. *A behavior is allowed under $x86_{man}$ iff it is allowed under $x86_{axiom}$.*

PROOF. The proof is given in §A.4.2. □

Simplification Our $Px86_{axiom}$ model is simpler than Px86 in [Raad et al. 2019b] in the following aspects:

- While tso ('total store order'), nvo ('non-volatile order'), and P ('persisted stores') components of Px86 are *existentially quantified*, thus increasing non-determinism, the analogous ob and P in $Px86_{axiom}$ are *constructed*.
- While the conditions for intra-thread, inter-thread, and CPU-PM communications are intertwined in Px86, they are separated and constrained by distinct axioms in $Px86_{axiom}$: intra-thread ones by co-rw and co-wr, inter-thread ones by external and CPU-PM ones by persist. To achieve this, $Px86_{axiom}$ orders fewer flush events than the Intel reference manual [Intel 2024a] does; e.g. unlike the manual, $Px86_{axiom}$ does not order FL before R.

[12] One may expect an asynchronous flush to complete also when the thread terminates. But this is defined neither in the Intel manual [Intel 2024a] nor in its libraries [Intel 2024d]. We thus assume an asynchronous flush *not* to be completed when a thread terminates. However, we can easily change this by appending $TERM$ to $MF \cup SF \cup U$, where $TERM$ denotes thread termination. Analogously, we can adapt $Px86_{view}$ in §2.3 to account for terminated threads.

[13] For clarity, we adapted Px86 from [Raad et al. 2019b] to match our style.

- $\text{Px86}_{\text{axiom}}$ may optionally require the minimality of pf, which is beneficial for e.g. reducing the search space significantly for stateless model checking. By contrast, Px86 does not require a similar minimality in tso.

As we show in §2.5, the constructive and succinct nature of $\text{Px86}_{\text{axiom}}$ and its stylistic similarity to the axiomatic Armv8 model [Pulte et al. 2019] make it easier to prove its equivalence to $\text{Px86}_{\text{view}}$.

2.5 Equivalence of $\text{Px86}_{\text{view}}$ and $\text{Px86}_{\text{axiom}}$

To evaluate the fidelity of $\text{Px86}_{\text{view}}$, we show that it is equivalent to $\text{Px86}_{\text{axiom}}$. To do this, we first prove the equivalence of x86_{view} and $\text{x86}_{\text{axiom}}$ by adapting the equivalence proof of the view-based and axiomatic models for Armv8 concurrency [Pulte et al. 2019], and then generalize it to Intel-x86 persistency. All theorems in this section are mechanized in Coq [Cho et al. 2021a].

Equivalence of x86_{view} and $\text{x86}_{\text{axiom}}$ In order to reuse the existing equivalence proof of the view-based and axiomatic models for Armv8 concurrency [Pulte et al. 2019] maximally, we appeal to a new model, x86_{prom}, the *promising* view-based model for Intel-x86 concurrency, as the bridge between x86_{view} and $\text{x86}_{\text{axiom}}$. Compared to x86_{view}, x86_{prom} additionally allows 'promises', modeling speculative writes (see §2.6.2). Specifically, we employ the following proof strategy:

(1) We prove that x86_{view} and x86_{prom} are equivalent and that promises do not enable additional behaviors as their effect is cancelled out by concurrency views (Theorem 2.5.1).

(2) We prove that x86_{prom} and $\text{x86}_{\text{axiom}}$ are equivalent by adapting the analogous equivalence proof for Armv8 concurrency [Pulte et al. 2019] as x86_{prom} and $\text{x86}_{\text{axiom}}$ respectively have the same style as the (view-based) $\text{Armv8}_{\text{view}}$ and (axiomatic) $\text{Armv8}_{\text{axiom}}$ models of Armv8 concurrency.

Combining the two steps we then establish the desired equivalence in Theorem 2.5.2.

LEMMA 2.5.1. *A behavior is allowed under $x86_{prom}$ iff it is allowed under $x86_{view}$.*

THEOREM 2.5.2. *A behavior is allowed under $x86_{view}$ iff it is allowed under $x86_{axiom}$.*

Equivalence of $\text{Px86}_{\text{view}}$ and $\text{Px86}_{\text{axiom}}$ We next extend Theorem 2.5.2 to Intel-x86 persistency (Theorem 2.5.3). To do this, we relate each view of an x86_{view} execution to a set of events in the corresponding $\text{x86}_{\text{axiom}}$ execution; similarly for the persistency views in $\text{Px86}_{\text{view}}$. For example, the $\mathsf{v}_{\mathsf{pCommit}}$ view of a thread state is related to the set P of persisted writes in the corresponding $\text{Px86}_{\text{axiom}}$ execution. This then allows us to prove the equivalence of $\text{Px86}_{\text{view}}$ and $\text{Px86}_{\text{axiom}}$.

THEOREM 2.5.3. *A behavior is allowed under $Px86_{axiom}$ iff it is allowed under $Px86_{view}$.*

2.6 View-Based and Axiomatic Models for Armv8 Persistency

In §2.3-§2.5, we presented view-based and axiomatic models for Intel-x86 persistency and proved their equivalence. We next do the same for Armv8. As Intel-x86 and Armv8 persistency are highly similar, we focus on their differences (§2.6.1; see §A.2 for the full details). We then present the view-based Armv8 persistency model (§2.6.2), fix and simplify the axiomatic model for Armv8 persistency due to [Raad et al. 2019a] as discussed in §2.2.2 (§2.6.3), and finally prove the equivalence of our view-based and axiomatic models (§2.6.4).

$$
\begin{array}{rcll}
 & & \ldots \text{ (the language for Intel-x86 in Fig. 2.3)} & \\
s \in \mathrm{Stmt} & ::= & \cdots & \textit{statement} \\
 & | & r \coloneqq \mathtt{load}_{xcl,rk}\ [e] & \textit{load} \\
 & | & r_{\mathrm{succ}} \coloneqq \mathtt{store}_{xcl,wk}\ [e_1]\ e_2 & \textit{store} \\
 & | & \mathtt{isb} \mid \mathtt{dmb}.f \mid \mathtt{dsb}.f & \textit{fence} \\
 & | & \mathtt{flushopt}\ e & \textit{flush} \\
f \in \mathrm{F} & ::= & \mathtt{ld} \mid \mathtt{st} \mid \mathtt{sy} & \textit{order}
\end{array}
$$

$$
\begin{array}{rcll}
xcl \in \mathbb{B} & ::= & \mathtt{false} \mid \mathtt{true} & \textit{exclusivity} \\
rk \in \mathrm{RK} & ::= & \mathsf{pln} \mid \mathsf{wacq} \mid \mathsf{acq} & \textit{read kind} \\
wk \in \mathrm{WK} & ::= & \mathsf{pln} \mid \mathsf{wrel} \mid \mathsf{rel} & \textit{write kind}
\end{array}
$$

Figure 2.9: The Armv8 concurrency/persistency language.

2.6.1 Armv8 versus Intel-x86 Persistency

We present the Armv8 language in Fig. 2.9, which is similar to that for Intel-x86 (Fig. 2.3), modulo the following:

Ordering: Armv8 ordering constraints are weaker and more elaborate than those of Intel-x86. Specifically, Armv8 loads and stores are annotated with *access ordering* constraints (rk or wk in Fig. 2.9). Moreover, Armv8 fences are more diverse: `isb` orders loads and later dependent accesses; $\mathtt{dmb}.f$ orders accesses according to the ordering constraint f (see Fig. 2.9); and `dsb.sy` additionally awaits the completion of pending flush instructions.

Exclusivity: Unlike Intel-x86, Armv8 supports exclusive *load-link* and *store-conditional* instructions [Jensen et al. 1987] that (if successful) prohibit intervening stores between the load and store. Exclusive instructions are more primitive than RMWs: RMWs can be implemented via exclusive instructions but not vice versa.[14] As such, loads and stores are annotated with *exclusivity* tags (xcl in Fig. 2.9).

Flush: All Armv8 flushes are asynchronous (`flushopt`).

As we describe shortly, these differences are largely orthogonal to modeling persistency, except for the relaxed ordering of writes. Specifically, Armv8 allows (unlike Intel-x86) speculative execution of writes, interacting with PM in an interesting way. To see this, we review the relaxed 'load buffering' behavior of Armv8 due to speculative writes:

$$
\begin{array}{l|\!|l}
(a)\ r_1 := \mathsf{Y}\ //= 1 & (c)\ r_2 := \mathsf{X}\ //= 1 \\
(b)\ \mathsf{X} := 1 & (d)\ \mathsf{Y} := 1
\end{array}
\qquad \text{(LB)}
$$

As Armv8 does not order a read and a subsequent write, (a) and (b) may be reordered; similarly for (c) and (d). As such, Armv8 allows an execution where (b), (d), (a), and (c) are executed in order, thus allowing the $r_1 = r_2 = 1$ behavior.

[14] While Armv8.1 also supports RMWs, they are currently missing in $\mathrm{Armv8}_{\mathrm{view}}$ and $\mathrm{Armv8}_{\mathrm{axiom}}$ [Pulte et al. 2019]. Accordingly, we do not extend them to support RMWs as this is orthogonal to our objectives here.

2.6.2 PArmv8$_{\text{view}}$: View-Based Armv8 Persistency

As with Px86$_{\text{view}}$, the view-based Armv8 persistency model, PArmv8$_{\text{view}}$, follows the same interleaving model over the history of stores. However, PArmv8$_{\text{view}}$ differs from Px86$_{\text{view}}$ in that (1) its views are more elaborate; and (2) it introduces *promises* to model speculative writes.

Views To model the ordering constraints and exclusivity of Armv8, the PArmv8$_{\text{view}}$ thread state in Fig. A.5 has additional view components compared to x86$_{\text{view}}$ in Fig. 2.4. These additional components are those of Armv8$_{\text{view}}$ [Pulte et al. 2019]; i.e. the PArmv8$_{\text{view}}$ thread state is that of Armv8$_{\text{view}}$ extended with persistency views ($\mathsf{v}_{\mathsf{pReady}}$, $\mathsf{v}_{\mathsf{pAsync}}$ and $\mathsf{v}_{\mathsf{pCommit}}$ in §2.3.5).

Promises The additional views, however, are not sufficient to model LB: without further instrumentation, the model remains interleaving, where either (a) or (c) is executed first, reading the initial value 0.

To model speculative writes, Armv8$_{\text{view}}$ [Pulte et al. 2019] introduces the notion of a *promise*: a message that may be speculatively added to the memory (or *promised*) without executing a store, provided that the promised message is later substantiated (or *fulfilled*) by executing a corresponding store. Put formally, a thread state ts contains the set $ts.\mathsf{prom}$ of the message ids that are promised by the thread but not yet fulfilled.

Using promises, we can model the LB behavior as follows, where tid_1 and tid_2 denote the left and right threads, respectively: (b-prom) tid_1 promises $\langle \mathsf{X} := 1\rangle_{tid_1}@1$ with $ts_1.\mathsf{prom} = \{@1\}$; (c) tid_2 reads $\langle \mathsf{X} := 1\rangle_{tid_1}@1$, updating $ts_2.\mathsf{coh}[\mathsf{X}]$ and $ts_2.\mathsf{v}_{\mathsf{rOld}}$ ('old-read view'[15]) to @1; (d) tid_2 writes $\langle \mathsf{Y} := 1\rangle_{tid_2}@2$, updating $ts_2.\mathsf{coh}[\mathsf{Y}]$ and $ts_2.\mathsf{v}_{\mathsf{wOld}}$ ('old-write view') to @2; (a) tid_1 reads $\langle \mathsf{Y} := 1\rangle_{tid_2}@2$, updating $ts_1.\mathsf{coh}[\mathsf{Y}]$ and $ts_1.\mathsf{v}_{\mathsf{rOld}}$ to @2; and (b-fulfill) tid_1 fulfills $\langle \mathsf{X} := 1\rangle_{tid_1}@1$, yielding $ts_1.\mathsf{prom}{=}\emptyset$ and $ts_1.\mathsf{v}_{\mathsf{wOld}}{=}@1$. Effectively, the write (b) is speculatively executed before the read (a) is executed.

To ensure that all speculations are substantiated, we require that a thread state's prom set be empty at the end of an execution; otherwise, the execution is deemed invalid.

Promises and Persistency The promises in PArmv8$_{\text{view}}$ similarly model speculative writes. Indeed, promises are largely orthogonal to persistency, except in the case of a crash. Specifically, in case of a crash in the presence of unfulfilled promises, we must determine the PM contents.

On the one hand, one may argue that unfulfilled promises should persist (remain in PM) as they have been made visible to other threads. To see this, consider Commit2 and suppose that the left thread promises $\langle \mathsf{Data} := 42\rangle@1$ which is yet unfulfilled, the right thread reads $\langle \mathsf{Data} := 42\rangle@1$ and writes $\langle \mathsf{Commit} := 1\rangle@2$, and then a crash occurs. If upon recovery $\langle \mathsf{Commit} := 1\rangle@2$ has persisted, then $\langle \mathsf{Data} := 42\rangle@1$ (which is an unfulfilled promise) should have also persisted.

On the other hand, one may argue that unfulfilled promises should not persist as they are not substantiated by a store. For example, suppose that the left thread in Commit2 promises to write $\langle \mathsf{Data} := 23\rangle@1$ without fulfilling it, and then it crashes. The promised write then should not persist as it is unsubstantiated; i.e. otherwise 23 appears *out-of-thin-air*.

To resolve this dilemma, we allow an execution to crash only if it has no unfulfilled promises. This then admits only the desired behaviors in Commit2: the execution cannot crash if either $\langle \mathsf{Data} := 42\rangle@1$ or $\langle \mathsf{Data} := 23\rangle@1$ is promised and not yet fulfilled. At first glance, this may seem restrictive as micro-architecturally an execution may crash even in the presence of uncommitted speculative writes. However, when this is the case, executing the remaining instructions to commit speculative writes does not constrain the PM contents. Moreover, we formally justify our design by proving that PArmv8$_{\text{view}}$ and PArmv8$_{\text{axiom}}$ are equivalent (see §2.6.4).

[15] While reads update $\mathsf{v}_{\mathsf{rNew}}$ in x86$_{\text{view}}$, they update $\mathsf{v}_{\mathsf{rOld}}$ in Armv8$_{\text{view}}$. We refer the reader to §A.2 for more details.

$$
\begin{aligned}
&\text{(axioms of Armv8}_{\text{axiom}}\text{ [Pulte et al. 2019] Fig. A.10)} \\
\text{fob} &= [W \cup R]; \text{po}; [\texttt{dmb.sy} \cup \texttt{dsb.sy}]; \text{po}; [FO] \\
&\cup\ [W \cup R]; (\text{po} \cap \text{CL}); [FO] \\
\text{ob} &= \text{obs} \cup \text{dob} \cup \text{aob} \cup \text{bob} \cup \text{fob} \cup \text{pf} \cup \text{fp} && (\textit{redefined}) \\
\text{pf} &\subseteq (\text{obs} \cup \text{dob} \cup \text{aob} \cup \text{bob} \cup \text{fob} \cup \text{fp})^{+} && (\textsc{pf-min}) \\
P &= dom(\text{pf}; [FO]; \text{po}; [\texttt{dsb.sy}]) \\
\forall l.\ \exists w.\ SM(l) = \texttt{wval}(w) \ \wedge\ &(P \times \{w\}) \cap \text{Loc} \subseteq \text{co}^{?} && (\textsc{persist})(\textsc{persist})
\end{aligned}
$$

Figure 2.10: The PArmv8$_{\text{axiom}}$ model.

2.6.3 PArmv8$_{\text{axiom}}$: Fixing and Simplifying PArmv8

We use the model of Pulte et al. [2019] as the baseline axiomatic model for Armv8 concurrency, presented as Armv8$_{\text{axiom}}$ in [Pulte et al. 2019, Appendix D].[16] The Armv8$_{\text{axiom}}$ model is equivalent to the authoritative axiomatic model in [Pulte et al. 2017] which is reviewed by Arm engineers. Note that Armv8$_{\text{axiom}}$ has the same style as x86$_{\text{axiom}}$ in Fig. 2.7, except that: (1) all coherence constraints are captured by a single axiom (INTERNAL) since (CO-WW) and (CO-RR) no longer follow from the other axioms;[17] (2) the ob component of Armv8$_{\text{axiom}}$ is more elaborate, modeling the weak ordering constraints of Armv8; and (3) Armv8$_{\text{axiom}}$ has an additional axiom (ATOMIC) that ensures the exclusivity of load-link/store-conditional instructions.

We next define an axiomatic model for Armv8 persistency, PArmv8$_{\text{axiom}}$ in Fig. 2.10, by extending Armv8$_{\text{axiom}}$ with persistency in the same style as Px86$_{\text{axiom}}$. The key differences from Px86$_{\text{axiom}}$ are that: (1) flush instructions impose different ordering constraints; (2) PArmv8$_{\text{axiom}}$ has no strong flush instructions; and (3) optimized flush instructions are guaranteed to commit only upon executing `dsb.sy` fences.

The PF-MIN axiom is optional as in Px86$_{\text{axiom}}$ (Theorem 2.4.2).

Fix Our PArmv8$_{\text{axiom}}$ model fixes the PArmv8 problem discussed in §2.2.2. Put formally, we prove the equivalence of PArmv8$_{\text{axiom}}$ and SPArmv8 which denotes strengthening PArmv8 by extending ob with pf and fp.

THEOREM 2.6.1. *A behavior is allowed under SPArmv8 iff it is allowed under PArmv8$_{axiom}$.*

PROOF. The proof is given in §A.5. □

2.6.4 Equivalence of PArmv8$_{\text{view}}$ and PArmv8$_{\text{axiom}}$

Finally, we prove that PArmv8$_{\text{axiom}}$ and PArmv8$_{\text{view}}$ are equivalent by generalizing the analogous concurrency result in [Pulte et al. 2019, Theorem 6.1] (showing that Armv8$_{\text{axiom}}$ and Armv8$_{\text{view}}$ are equivalent) and extending it with persistency.

THEOREM 2.6.2. *A behavior is allowed under PArmv8$_{axiom}$ iff it is allowed under PArmv8$_{view}$.*

PROOF. The proof is mechanized in [Cho et al. 2021a]. □

[16] We refactor the relations in [Pulte et al. 2019] to replace dmb with dmb ∪ dsb. The latter is a straightforward extension as dsb is strictly stronger than dmb [Arm 2020].

[17] We could replace INTERNAL with irreflexivity of po; (co ∪ rf ∪ fr ∪ fr; rf) for uniformity with x86$_{\text{axiom}}$. We forwent this to use Armv8$_{\text{axiom}}$ [Pulte et al. 2019] as is.

2.7 Model Checking Persistency Patterns

We develop a stateless model checker for PArmv8$_{\text{view}}$ by generalizing and extending the Armv8$_{\text{view}}$ model checking framework in [Pulte et al. 2019] to support persistency and account for crashes (§2.7.1). We use our model checker to verify representative persistent synchronization examples, including the AtomicPersists example [Raad et al. 2020] that emulates a persistent transaction §A.6. Our model checking tool and verified examples are open source and publicly available [Cho et al. 2021a].

2.7.1 Model Checking Tool

Model Checking Tool for Armv8$_{\text{view}}$ We first briefly review the baseline model checking tool for Armv8$_{\text{view}}$ [Pulte et al. 2019], which is a part of RMEM [Armstrong et al. 2019]. The tool consists of two parts: the executable model for sequential semantics of Armv8 ISA written in Sail [Armstrong et al. 2019]; and the executable memory model for concurrency written in Lem [Mulligan et al. 2014]. The former is adopted from [Pulte et al. 2017], and the latter is split into two modes: the "promise-mode" which approximately enumerates the reachable final memories; and the "non-promise-mode" that checks if each potentially reachable final memory is actually reachable by thread executions to the end without promises. The two-mode execution is sound for the Armv8$_{\text{view}}$ model: a reachable state in Armv8$_{\text{view}}$ is also reachable by first promising to write all messages and then fulfilling the promises by executing the threads [Pulte et al. 2019, Theorem 7.1].

Extension for PArmv8$_{\text{view}}$ We extend the model checking tool for Armv8$_{\text{view}}$ as follows: (1) we add persistency instructions to the executable model for sequential semantics in Sail; (2) we add persistency views to the executable memory model for Armv8$_{\text{view}}$ in Lem; (3) we enumerate not only final but also intermediate reachable memories in the promise-mode; and (4) we allow each thread's execution to stop amidst the non-promise-mode; and (5) we enumerate all post-crash states from the reachable states of intermediate memories and persistency views.

The performance of the resulting model checking algorithm for PArmv8$_{\text{view}}$ is similar to that for Armv8$_{\text{view}}$ because $(1), (2), (4), (5)$ introduce only a constant-factor overhead; and the number of intermediate memories in (3) is usually dominated by that of final memories.

2.8 Discussion

2.8.1 Related Work and Impact

Hardware Persistency Models Existing literature includes several works on formalising and testing hardware persistency models [Pelley et al. 2014; Raad et al. 2019b,a; Joshi et al. 2015; Condit et al. 2009; Liu et al. 2019; Khyzha and Lahav 2021]. As discussed in detail in §2.2–2.6, the works of Raad et al. [2019b,a] are closest to ours. Pelley et al. [2014] propose several persistency models including epoch persistency; however, these models have not been adopted by mainstream architectures as of yet. Joshi et al. [2015]; Condit et al. [2009] describe epoch persistency under x86-TSO [Sewell et al. 2010]. Liu et al. [2019] develop the PMTest testing framework for finding persistency bugs in software running over hardware models. Izraelevitz et al. [2016a] give a formal semantics of epoch persistency under release consistency [Gharachorloo et al. 1990]. As discussed in §2.1, the PTSO model of Raad and Vafeiadis [2018] formalises epoch persistency under x86$_{\text{man}}$ (TSO) as a *proposal* for Intel-x86. However, PTSO is rather different from the existing Intel-x86 persistency model in [Intel 2024a] in that it does not support the fine-grained Intel primitives for *selectively* persisting cache lines (`flush` and `flushopt`), and instead proposes coarse-grained instructions (for persisting *all* locations at once) that do not exist in Intel-x86.

Khyzha and Lahav [2021] recently developed the PTSO_{syn} model for Intel-x86 that satisfies the three properties of (A)–(C) discussed in §2.1. In particular, PTSO_{syn} supports persistent synchronization patterns even in the presence of I/O (B) as it also models `flush` instructions synchronously like $\text{Px86}_{\text{view}}$ (§2.3.5). However, this problem of asynchronous modeling of `flush` regarding I/O is not discussed in the paper.

Intel recently introduced Optane Persistent Memory 200 Series [Intel 2024c] that feature Enhanced Asynchronous DRAM Refresh (eADR), which treats processor caches as persistent (rather than volatile) by automatically flushing cache data to PM in case of a crash. When eADR is available, a store is guaranteed to persist when made visible to other threads (e.g. after executing an `mfence`/`sfence`, but not `clflush`/`clflushopt`). Nevertheless, we believe that our contributions still stand for the following reasons. First, to ensure backwards compatibility, programs must support persistency in the absence of eADR. That is, a *correct* PM program must defensively check whether eADR is enabled, and if not insert appropriate `clflush` or `clflushopt` instructions per our models. Second, eADR may increase runtime cost. For example, to flush cache data to PM when a crash occurs, eADR must drain more power with higher voltage level or larger capacity, the impact of which on power consumption has not been thoroughly analyzed as of yet. The increased power consumption may affect embedded systems worse, and to our knowledge, Arm currently has no plans for supporting an eADR-like feature in Armv8.

Software Persistency Models The literature on software persistency is more limited [Chakrabarti et al. 2014; Kolli et al. 2017; Gogte et al. 2018]. Kolli et al. [2017] propose *acquire-release persistency*, an analogue to release-acquire consistency in C/C++. Gogte et al. [2018] propose *synchronisation-free regions* (regions delimited by synchronisation operations or system calls). Although both approaches enjoy good performance, their semantic models are rather fine-grained, paving the way towards more coarse-grained transactional models [Intel 2024d; Kolli et al. 2016; Tavakkol et al. 2018; Shu et al. 2018; Avni et al. 2015; Raad et al. 2019a].

Verification There are several works on implementing and verifying algorithms that operate on PM. Friedman et al. [2018] developed persistent queue implementations using Intel-x86 persist instructions (e.g. `flush`). Similarly, Zuriel et al. [2019] developed persistent set implementations using Intel-x86 persist instructions. Derrick et al. [2019] provided a formal correctness proof of the implementation in [Zuriel et al. 2019]. All three of [Derrick et al. 2019; Zuriel et al. 2019; Friedman et al. 2018] assume that the underlying concurrency model is sequential consistency [Lamport 1979], rather than x86_{man} (TSO). Raad et al. [2020] developed a persistent program logic for verifying programs under the Px86 model. Kokologiannakis et al. [2021] formalised the consistency and persistency semantics of the Linux ext4 filesystem, and developed a model-checking algorithm and tool for verifying the consistency and persistency behaviors of ext4 applications such as text editors

Our work had concrete impacts on (1) a view-based program logic, Pierogi, for Owicki-Gries reasoning about Intel-x86 persistency [Bila et al. 2022]; and (2) a view-based concurrent separation logic, Spirea, for verifying programs under a weak persistent memory model [Vindum and Birkedal 2023]. Both approaches build upon the notion of persistency view introduced in this work for reasoning about persistent programs.

2.8.2 Future Work

We plan to build on this work in several ways. First, we will empirically validate the proposed models w.r.t. PM hardware using custom SoC (ASIC or FPGA) that captures the traffic between CPU and PM, as proposed also in [Raad et al. 2019b]. Second, we will explore language-level persistency by researching persistency extensions of high-level languages such as C/C++. This will liberate programmers from understanding hardware-specific persistency guarantees and make persistent programming more accessible. Third, we will first specify existing persistent libraries such as PMDK [Intel 2024d] and then use our model checker (§2.7) to verify their

implementations against our specifications. Lastly, in the spirit of persistency semantics defining the order in which writes are propagated to PM in DIMM slots, we will study the semantics in the presence of accelerators (e.g. CXL [Consortium 2024b] and CCIX [Consortium 2024a]), defining the order in which writes are propagated to accelerators in PCIe slots or other peripheral interconnects.

Chapter 3. A General Programming Model

3.1 Introduction

A key building block in PM for such optimizations are concurrent data structures that ensure that the underlying DS is both *thread-safe* (i.e. it behaves correctly when accessed by concurrent threads racing to manipulate the DS) and *crash-consistent* (i.e. it is restored into a consistent state upon recovery from a crash, e.g. a power failure). The thread-safety of the underlying DS is ensured by using a suitable *concurrency control* mechanism, e.g. transactional memory (TM), locking, or lock-free techniques using fine-grained synchronization primitives (e.g. CAS instructions). Compared to TM- or locking-based DS implementations, *lock-free* data structures have the following two advantages. (1) They have a greater potential to parallelize workloads than others by distributing memory accesses across a multitude of contention points [David et al. 2018]. For instance, logging imposes significant overhead both in time (due to the concentrated contention point at the tip) and in space (because all intermediate changes are recorded). As such, lock-free queues and hash tables [Fatourou and Kallimanis 2012, 2011; Hendler et al. 2010; Goodman et al. 1989] outperform lock- and TM-based ones in PM. (2) Lock-free algorithms ensure that the DS is in a consistent state *at all times*, thereby eliminating the need for additional mechanisms to ensure crash consistency, so long as the updates on the DS are flushed to PM in a timely manner.

As such, persistent lock-free DSs have drawn significant attention in the literature, including persistent lock-free stacks [Attiya et al. 2019], queues [Friedman et al. 2018], lists [Attiya et al. 2022; Zuriel et al. 2019], hash tables [Chen et al. 2020; Zuriel et al. 2019; Nam et al. 2019], and trees [Attiya et al. 2022], as well as general techniques for transforming volatile (in-DRAM) lock-free DSs to persistent (in-PM) DSs [Lee et al. 2019; Friedman et al. 2020, 2021; Izraelevitz et al. 2016b].

One of the most widely accepted correctness criteria for persistent lock-free DSs (and concurrent DSs in general) is *durable linearizability* (DL) [Izraelevitz et al. 2016b]. A multi-threaded execution (where the operations of concurrent threads can arbitrarily interleave) is *linearizable* if each operation appears to execute and take effect atomically (without being interleaved by operations in other threads) at some point, called the *linearization point*, between its invocation and response [Herlihy and Wing 1990]. DL is an extension of linearizability to the PM setting and additionally offers crash consistency guarantees. Specifically, a multi-threaded execution that possibly spans multiple crashes satisfies DL if it is linearizable when ignoring the crash events. In particular, operations completed before a crash should be persisted across the crash, and if there are operations whose executions are interrupted by the crash, then the DS should be *recovered* to a consistent state after the crash. DL is indeed satisfied by most existing persistent concurrent DSs, except for those DSs that intentionally trade durability for performance [Friedman et al. 2018].

However, DL is insufficient for *composing* persistent DSs with one another [Friedman et al. 2018]. For instance, consider a banking DS comprising a *savings* account, S, and a *current* account, C, where amount a is withdrawn from S and, if successful, deposited into C:

1: $succ \coloneqq$ WITHDRAW(S, a); **if** $succ$ **then** DEPOSIT(C, a);

Even if both DSs underlying C and S each individually satisfy DL, the whole banking DS does not: the amount a withdrawn from S can be lost if a crash occurs before it is deposited into C. What is needed for the correctness of this composition is the stronger *detectable recoverability* (or *detectability* in short) [Friedman et al. 2018]. Under

detectability, after a crash a user can (1) detect if an operation was not invoked, interrupted by the crash, or completed before the crash; (2) resume the execution of an interrupted operation; and (3) retrieve the correct output for completed operations. If S and C were detectable, then one could detect the value withdrawn from S and whether it was deposited into C in case of a crash, and resume the interrupted operation.

Note that existing *persistent TM* (PTM) systems such as those of Memaripour et al. [2017]; Krishnan et al. [2020] provide such detectability guarantees. Specifically, code wrapped within a persistent transaction t is executed both atomically (i.e. it is thread-safe) and failure-atomically (i.e. either all or none of the effects of t take place in case of a crash, and thus t is detectable). Nevertheless, PTM systems have two limitations that make them unsuitable for implementing persistent concurrent DSs. First, the code wrapped within a PTM (or TM for that matter) is typically required to comprise simple memory read and write operations, rather than arbitrary operations associated with DSs. Second, even if one could enclose arbitrary DS operations within a PTM transaction, combining PTM and a concurrent DS is not straightforward: a PTM (as with TM) system provides its own concurrency control mechanism (e.g. via locking), clashing with and defeating the purpose of the already in place concurrency control mechanism of a concurrent DS. As such, PTM systems are not immediately suitable for implementing detectable concurrent DSs in PM.

Challenges Our aim here is to devise a technique for implementing detectable concurrent DSs in PM in such a way that is both *generally applicable* (i.e. it can be applied to implement an arbitrary DS rather than tailored towards a specific DS, e.g. a queue) and *highly performant*.

This, however, is far from straightforward. Specifically, as we discuss below, although several detectable concurrent DSs have been proposed in the literature [Friedman et al. 2018; Li and Golab 2021; Rusanovsky et al. 2021; Attiya et al. 2018; Ben-David et al. 2019; Attiya et al. 2022], to the best of our knowledge, each is either limited to simple algorithms or suffers from high runtime overhead.

- ***General Applicability***: Many of the existing detectable concurrent DSs are hand-tuned and manually reason about crash consistency and detectability. Friedman et al. [2018]; Li and Golab [2021] present detectable lock-free queues. Rusanovsky et al. [2021]; Fatourou et al. [2022] present a general combiner to construct persistent combining DSs, but it can recover only the last invocation of each operation. As such, in an execution of the banking example where S is withdrawn two times before a crash, we cannot distinguish whether the crash happened during the first or the second invocation of Withdraw. Attiya et al. [2018] present a detectable compare-and-swap (CAS) operation on PM locations as a general primitive operation for pointer-based DSs. However, the applicability of their CAS to concurrent DSs has not been established. Attiya et al. [2022] present a transformation from concurrent DSs in DRAM into those in PM with detectability, but this requires the operations to be strictly splittable into two phases: load-only *gather* and CAS-only *update*. Such a requirement is satisfied by data structures such as linked-lists [Harris 2001], but not by more sophisticated ones such as the Michael-Scott queue [Michael and Scott 1996] or hash tables [Shalev and Shavit 2006; Chen et al. 2020] that perform loads and CASes in an interleaved manner. Ben-David et al. [2019] present a more general transformation, but theirs requires the operations to follow specific patterns such as the *normalized* form [Timnat and Petrank 2014] for efficient transformation and makes a simplifying assumption that is not satisfied by real-world systems (see §3.7).

- ***High Performance***: While the overhead of detectability is modest or negligible for hand-tuned DSs [Friedman et al. 2018; Li and Golab 2021; Rusanovsky et al. 2021; Attiya et al. 2022], it is significant for the transformation of Ben-David et al. [2019] for two reasons. First, an object supporting a detectable CAS consumes $O(T)$ space in PM where T is the number of threads, prohibiting its use for space-efficient DSs such as hash tables and

trees. More significantly, the detectable CAS object of Attiya et al. [2018] consumes $O(T^2)$ space in PM. Second, the transformed program writes and flushes to PM rather frequently (see §3.7 for details).

Contributions and Outline To address the above challenges, we present Memento: the first general programming framework for high-performance, detectable, concurrent DSs in PM.[1] To this end, we generalize Ramalingam and Vaswani [2013]'s type system that statically ensures the detectable recovery of programs in a simple core language. In contrast to the prior work, Memento's type system additionally supports control constructs such as conditionals, loops, and function calls for general programming, and the CAS primitive operation for concurrent programming in PM. Our type system ensures programs to be deterministically replayed after a crash so that well-typed programs are detectably recoverable when simply re-executed *from the beginning* after a crash. As such, our type system substantially *reduces* the complexity of designing detectable DS in PM to that of designing volatile DS. Unlike most hand-tuned persistent DSs that require challenging-to-develop and reason-about DS-specific recovery code, our framework solely requires a program to conform to our type system, thereby *eliminating* the need for DS-specific recovery code! As example, we adapt several volatile concurrent DSs to well-typed programs and automatically derive detectable concurrent DSs. Specifically, we make the following contributions:

- In §3.2, we describe how to design programs that are deterministically replayed after a crash. We do so using two primitive operations, detectable checkpoint and CAS, by composing them with usual control constructs such as sequential composition, conditionals, and loops.
- In §3.3, we design a core language for persistent programming and its associated type system for deterministic replay, and prove that well-typed programs are detectably recoverable.
- In §3.4, we present an implementation of our core language in the Intel-x86 Optane DCPMM architecture. Our construction is not tightly coupled with Intel-x86 so that it can be adapted to other PM architectures like Samsung's CMM-H in a straightforward manner.
- In §3.5, we adapt several volatile, concurrent DSs to satisfy our type system, automatically deriving detectable concurrent DSs in PM. These include a lock-free linked-list [Harris 2001], Treiber stack [Treiber 1986], Michael-Scott queue [Michael and Scott 1996], a combining queue, and Clevel hash table [Chen et al. 2020]. In doing so, we capture the optimizations of hand-tuned persistent concurrent DSs with additional primitives and type derivation rules (§B.1 and §B.2), and support safe memory reclamation even in the presence of crashes.
- In §3.6, we evaluate the detectability and performance of our CAS and automatically derived concurrent DSs in PM. They successfully recover from random thread and system crashes in stress tests, respectively (§3.6.1); and perform comparably with the existing hand-tuned persistent DSs with and without detectability (§3.6.2).

In §3.7, we conclude with related and future work. Our implementation and experimental results are open-sourced and available as supplementary material [Cho et al. 2023a].

3.2 Designing Detectable Programs with Deterministic Replay

Memento achieves detectability by deterministically replaying programs after a crash. Before presenting our type system that statically ensures deterministic replay of programs in §3.3, we first describe our key idea,

[1] We use the word "concurrent" to emphasize Memento's general applicability, but the framework applies not only to lock-free or lock-based concurrent CSs but also to sequential DSs.

Algorithm 1 Transfer from a savings account to a current account with mementos

```
1: function TRANSFER(savings, current, amount, mid)                ▷ mid: memento id
2:     let succ := WITHDRAW(savings, amount, mid.withdraw);
3:     if succ then DEPOSIT(current, amount, mid.desposit)
4: end function
```

which is recording the progress and result of a program using a *memento*, a thread-private log stored in PM (hence the framework name), in a compositional manner.

3.2.1 Ensuring Deterministic Replay of Composed Operations

Composition Consider the TRANSFER function of our banking example (§3.1) shown in Algorithm 1: it attempts to withdraw $amount$ from $savings$ (L2), and if successful, it deposits the same amount into $current$ (L3). The code without highlighted parts is correct on volatile memory but not recoverable on PM in case of crashes. To ensure deterministic replay of TRANSFER, it suffices to ensure those of its sub-operations WITHDRAW and DEPOSIT using sub-mementos mid.withdraw and mid.deposit, respectively. Regardless of whether the execution of a function f is finished or interrupted at crash time, thanks to its memento the post-crash re-execution of f will return the same result or resume from the interrupted program point, respectively. For instance, if the pre-crash execution crashes at L2, the post-crash re-execution resumes WITHDRAW thanks to its deterministic replay. On the other hand, if the pre-crash execution crashes during DEPOSIT at L3, the post-crash re-execution produces the same result $succ$ from WITHDRAW, takes the same branch, and resumes DEPOSIT. In general, the deterministic replay property is preserved by sequential composition and conditionals.

Checkpoint Primitive As a general-purpose primitive operation, our framework provides a detectable *checkpoint* operation that records the result of a read-only expression:

```
1: v := chkpt(λ.e, mid)                                            ▷ e: read-only
```

Here, e is a read-only expression whose result may change across crashes due to, e.g. concurrent modifications to PM. The checkpoint operation first checks if a value is recorded in the memento mid, and if so it returns its value; otherwise, it executes e, records its result in the memento mid and returns the result. The checkpoint operation is detectable: even though it may partially execute e multiple times across crashes (hence the requirement for e to be read-only), it produces a unique result that is recorded in the memento across crashes and assigns this unique result to v.

A PM allocation is considered read-only as its effect is thread-local and becomes visible to other threads only after the address is published to shared memory. It is safe to leak PM allocations during crashes, as the underlying memory allocator is assumed to trace garbage after a crash.

Checkpoint operation is already proposed in prior work [Ben-David et al. 2019], but we generalize their implementation with *timestamps* (see §3.2.2 for details). We will present our design in §3.4.2.

Compare-and-Swap Primitive As another general-purpose primitive operation for concurrent programming, our framework provides a detectable, persistent *compare-and-swap* (CAS) operation:[2]

```
1: r := pcas(loc, v_old, v_new, mid)
```

This operation compares the current value of loc against v_{old}, and if the values match it updates it to v_{new}; otherwise the value of loc is unchanged. The return value $r \in \mathbb{B} \times \mathrm{Val}$ is a pair comprising a boolean flag

[2]Here we omit memory orderings [McKenney 2005]—e.g. release or acquire—but we annotate the most efficient and yet correct orderings in our implementation.

Algorithm 2 Insertion on the Harris concurrent sorted linked-list

```
1: function INSERT(head, val, mid)
2:     loop
3:         let (prev, next, blk) := chkpt(λ.e_pnb, mid.pnb);        ▷ timestamp: 30 | 80
4:         let succ := pcas(prev.next, next, blk, mid.cas);          ▷ timestamp: 20 | 90
5:         if succ then return
6:     end loop
7: end function
   e_pnb ≜ (p, n) := FIND(head, val); b := palloc(⟨val : val; next : n⟩); return (p, n, b)
```

reflecting whether the update was successful, and the original value held in loc. The operation guarantees that the result r is deterministic so long as the arguments are also deterministic. In particular, if a pcas were unsuccessful before a crash, its failure would be recorded in its memento mid and thus the post-crash execution would also fail by inspecting mid.

Note that deterministic replay cannot be achieved using plain CAS operations: in case of a crash, one loses such information as whether the plain CAS was performed, and if it was successful or not. The pcas requires additional synchronization in PM. Recognizing its general applicability, Attiya et al. [2018]; Ben-David et al. [2019] have proposed alternative implementations, but they consume $O(T^2)$ and $O(T)$ PM space for each location, respectively, where T is the number of threads. By contrast, our implementation (§3.4.3) uses only 8 PM bytes for each location.

3.2.2 Supporting Simple Loops with Timestamps

The banking example uses a unique sub-memento for each sub-operation, making it easier to ensure a deterministic replay of composed operations. While feasible for simple programs, the unique memento assumption does not apply to complex programs with loops as the sub-mementos are reused across different loop iterations. To support loops, our framework employs *timestamps*.

Example: Concurrent Linked-List Consider the INSERT operation on the concurrent sorted linked-list by Harris [2001] in Algorithm 2. For brevity, we omit the implementation of the function FIND$(head, val)$ (traversing the list from $head$ to find val) and the deallocation of non-inserted blocks (see §3.5.1 for the implementation). As before, the code without highlighted parts is correct for volatile memory: it searches for adjacent blocks, $prev$ and $next$, between which val is inserted while preserving the sorted order and allocates a new block, blk, that contains val and points to $next$ (L3); performs a CAS on $prev.next$ from $next$ to blk (L4); and keeps trying until successful (L5).

Challenge: Reused Memento Adding the highlighted parts (replacing cas with pcas at L4), programmers can ensure the deterministic replay of the loop body. However, it is insufficient to correctly recover from crashes after loop iterations as they reuse mementos. Consider an execution that crashes right after L3 in the *second* loop iteration. After the crash, mid.pnb contains the result of e_{pnb} in the *second* iteration, while mid.cas contains the result of the CAS in the *first* iteration. As such, it is necessary to distinguish the results of sub-operations from different iterations for correct recovery; otherwise, a post-crash execution would mix the sub-operation results.

To address the challenge of loops and more generally of complex control flow, the prior work performs additional writes and following flushes to PM to record the operation progress. Specifically, Attiya et al. [2018]; Li and Golab [2021] additionally reset memento-like "operation descriptors" by writing sentinel values to PM; and Ben-David et al. [2019] further checkpoint the program counter in PM. However, these additional writes and

Algorithm 3 Resizing the Clevel hash table [Chen et al. 2020] (simplified)

```
1: function ResizeMoveArray(from, to, mid)
2:     loop
3:         let i := chkpt(λ.φ(0, i + 1), mid.i);          ▷ timestamp: 30 | 80
4:         if i ≥ |from| then return
5:         ResizeMoveEntry(from, i, to, mid.entry)        ▷ timestamp: 20 | 90
6:     end loop
7: end function
```

flushes to PM incur a significant performance overhead for high-contention workloads with heavy use of loops (see §3.6 for details).

Solution: Timestamp To distinguish between the sub-operation results of different iterations efficiently (and record the operation progress more generally), our framework uses *timestamps*. A timestamp is a counter that increases monotonically during executions and across crashes.[3] Specifically, each primitive detectable sub-operation additionally records in its sub-memento the timestamp at which it completes. In the above scenario, the sub-operations may record timestamps of 10 and 20 in mid.pnb and mid.cas in the first loop iteration, respectively, and then overwrite the timestamp of 30 in mid.pnb in the next iteration.

In the post-crash execution, our framework first observes that timestamp 30 in mid.pnb and then 20 in mid.cas, which is *not* monotonically increasing with the control flow. That is, the checkpoint at L3 was performed in the last iteration before the crash, but the pcas at L4 was not. As such, the post-crash execution may resume at L4 and re-execute pcas.

Regardless of the program point at which the execution crashes, the post-crash execution can deterministically replay the last iteration before the crash. Suppose the timestamps recorded in mid.pnb and mid.cas were 80 and 90, respectively. Then the post-crash execution replays the last iteration by observing the monotonically increasing timestamps (80 at L3 and 90 at L4) and retrieves the recorded results. Thereafter, it will either successfully return or try again (L5).

Unlike prior approaches [Attiya et al. 2018; Ben-David et al. 2019], our approach does not incur additional writes and flushes to PM.[4] On the one hand, our primitive operations, checkpoint and CAS, record an operation's timestamp and result atomically at once. On the other hand, our framework does not require additional writes and flushes for loops and other control constructs.

3.2.3 Supporting Loop-Carried Dependence by Checkpointing Dependent Variables

In the presence of loop-carried dependence, timestamps alone do not guarantee deterministic replay because dependent variable values may be lost in case of a crash. As such, our framework further requires programmers to checkpoint the dependent variables for each iteration.

Example: Clevel Hash Table Consider the ResizeMoveArray operation on the Clevel hash table [Chen et al. 2020] presented in Algorithm 3. When resizing the hash table, every entry in the array of an old level, *from*, is moved to the array of a new level, *to*. To do this, the operation iterates over *from* (L3) and invokes the sub-operation ResizeMoveEntry for each entry index i (for brevity we omit ResizeMoveEntry). To reveal loop-carried dependence explicitly, we represent the code in the Static Single Assignment (SSA) form [Cytron

[3] Intel-x86 does not natively support such a timestamp with strong properties, but we develop such a counter in §3.4.2.

[4] Since our approach reduces the number of writes as well as that of flushes, it has performance advantages over the prior approaches in a wide range of PM platforms including Intel eADR [Intel 2021].

et al. 1989, 1991].[5] In the SSA form, loop-dependent variables are defined as a *ϕ-node* at the beginning of the loop. A ϕ-node of the form $v = \phi(v_0, v_1)$ assign v_0 (resp. v_1) to v if it is the first (resp. a later) iteration. In our example of Algorithm 3, i gets 0 in the first iteration and $i + 1$ in the later iterations at L3.

Challenge: Dependent Variable With the highlighted part, especially invoking the sub-operation with an additional memento argument mid.entry at L5, our framework ensures the deterministic replay of the loop body. However, the loop-dependent variable i makes it challenging to correctly recover from crashes because the framework needs to restore the value of i in the last iteration.

Solution: Checkpoint at the Loop Head To address the challenge above, our framework requires programmers to checkpoint dependent variables, e.g. i, at the loop head. In the post-crash execution, the checkpoint operation retrieves the i value in the last iteration and, moreover, *delimits* the last iteration. For instance, suppose that L3 and L5 record timestamps 30 and 20, respectively. Then the last iteration began at timestamp 30 and the post-crash execution should re-execute L5. Similarly, if L3 and L5 respectively record timestamps 80 and 90, then the last iteration began at timestamp 80 and the post-crash execution should retrieve the sub-operation result recorded in mid.entry at L5.

In the presence of multiple dependent variables, our framework requires programmers to merge them all into a single tuple or struct and checkpoint it at once. Otherwise, dependent variables of two consecutive iterations can be mixed. For instance, suppose there were two dependent variables, x and y, and they were individually checkpointed. If only x were checkpointed at the loop head and then the thread crashes, then the post-crash execution retrieves the value of x from the last iteration and that of y from the previous iteration, violating the recovery correctness.

3.3 Type System for Detectability

We next formalize the key idea presented in §3.2. We design a core language for PM (§3.3.1) and a type system for deterministic replay (§3.3.2), and prove that typed programs are detectable (§3.3.3).

3.3.1 Core Language

We present the syntax and semantics of our core language for PM in Fig. 3.1. We discuss the implementation of our language later in §3.4, and give its semantics in the technical appendix (§B.4).

A program, p, consists of a function environment, δ, and a list of statements, $\overrightarrow{s_{tid}}$, for each thread tid. An assignment statement, $r := e$ where $r \in \mathrm{VReg}$ is a register id and $e \in \mathrm{Expr}$ is a pure expression, evaluates e to a value in $\mathrm{Val} \subseteq \mathrm{Expr}$ and assigns it to r. An expression is either a constant, register, arithmetic/boolean operation, tuple/union introduction/elimination, *memento id*, empty expression (ϵ) or concatenation ($e.lab$, see below). A value is an irreducible expression without variables. A load statement, $r := \texttt{pload}(e)$, evaluates e as a PM location, $l \in \mathrm{PLoc} \triangleq \mathbb{N}$, in the *shared* memory, loads the value of l and writes it to r. For simplicity, we classify PM locations into shared and thread-local ones so that we can use the former as concurrent DS memory blocks and the latter as mementos. An allocation, $r := \texttt{palloc}(e)$, initializes a fresh PM location in the shared memory with the value evaluated from e and writes the location to r.

A conditional statement, $\texttt{if}\ (e)\ \overrightarrow{s_t}\ \overrightarrow{s_f}$, reduces either to $\overrightarrow{s_t}$ or to $\overrightarrow{s_f}$ depending on the value evaluated from e. Loops reveal loop-carried dependence explicitly in the style of the SSA form (§3.2.3). Specifically, $\texttt{loop}\ r\ e\ \overrightarrow{s}$

[5] The SSA form can represent a much more general class of control flow-dependent variables than loop-dependent variables [Cytron et al. 1989, 1991]. Although we present this example in SSA form, we do not require SSA in our implementation.

$$
\begin{array}{rcll}
p & ::= & [\delta]\ \overrightarrow{s_1} \mid\mid \ldots \mid\mid \overrightarrow{s_n} & \textit{program} \\
s \in \mathrm{Stmt} & ::= & r \coloneqq e \mid r \coloneqq \mathtt{pload}(e) \mid r \coloneqq \mathtt{palloc}(e) & \textit{assignment \& PM} \\
& \mid & \mathtt{if}\ (e)\ \overrightarrow{s_\mathsf{t}}\ \overrightarrow{s_\mathsf{f}} \mid \mathtt{loop}\ r\ e\ \overrightarrow{s} \mid \mathtt{continue}\ e \mid \mathtt{break} & \textit{control constructs} \\
& \mid & r \coloneqq f(\overrightarrow{e}) \mid \mathtt{return}\ e & \textit{function call/return} \\
& \mid & r \coloneqq \mathtt{chkpt}(\overrightarrow{s}, e_\mathsf{mid}) \mid r \coloneqq \mathtt{pcas}(e_\mathsf{loc}, e_\mathsf{old}, e_\mathsf{new}, e_\mathsf{mid}) & \textit{detectable op.} \\
e \in \mathrm{Expr} & ::= & () \mid z \mid b \mid \mathit{mid} \mid r \mid (e_1\ \mathit{op}\ e_2) \mid e.i \mid (e_1, e_2) \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e & \textit{pure expr.} \\
& \mid & \mathsf{match}\ e\ \{\ \mathsf{inl}\ e_\mathsf{l} \Rightarrow e_\mathsf{l}'\ ,\ \mathsf{inr}\ e_\mathsf{r} \Rightarrow e_\mathsf{r}'\} \mid \epsilon \mid e.\mathit{lab} \mid\ \ldots & \\
v \in \mathrm{Val} & ::= & () \mid z \mid b \mid \mathit{mid} \mid (v_1, v_2) \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v & \textit{value}
\end{array}
$$

$$
z \in \mathbb{Z} \qquad b \in \mathbb{B} \qquad \mathit{op} \in \mathrm{Op} \qquad r \in \mathrm{VReg} \triangleq \mathbb{N} \qquad \sigma \in \mathrm{VRegMap} \triangleq \mathrm{VReg} \rightharpoonup \mathrm{Val}
$$

$$
f \in \mathrm{FnId} \qquad \delta \in \mathrm{Env} \triangleq \mathrm{FnId} \rightharpoonup (\overrightarrow{\mathrm{VReg}} \times \overrightarrow{\mathrm{Stmt}}) \qquad l \in \mathrm{PLoc} \triangleq \mathbb{N} \qquad \mathit{lab} \in \mathrm{Label} \qquad \mathit{mid} \in \overrightarrow{\mathrm{Label}}
$$

$$
t \in \mathrm{Time} \triangleq \mathbb{N} \qquad \mathit{tid} \in \mathrm{TId} \triangleq \mathbb{N} \qquad \mathit{mmts} \in \mathrm{Mmts} \triangleq \overrightarrow{\mathrm{Label}} \to \langle \mathsf{val} : \mathrm{Val}; \mathsf{time} : \mathrm{Time} \rangle
$$

$$
c \in \mathrm{Cont} \qquad \mathit{ts} \in \mathrm{TState} \triangleq \langle \mathsf{regs} : \mathrm{VRegMap}; \mathsf{time} : \mathrm{Time} \rangle \qquad T \in \mathrm{Thread} \triangleq \overrightarrow{\mathrm{Stmt}} \times \overrightarrow{\mathrm{Cont}} \times \mathrm{TState} \times \mathrm{Mmts}
$$

$$
\mathit{ev} \in \mathrm{Event} ::= \mathsf{R}(l, v) \mid \mathsf{U}(l, v_{old}, v_{new}) \qquad \mathit{tr} \in \overrightarrow{\mathrm{Event}} \qquad \mathsf{mem} \in \mathrm{Mem} \qquad M \in \mathrm{Machine} \triangleq \overrightarrow{\mathrm{Thread}} \times \mathrm{Mem}
$$

$$
\boxed{\overrightarrow{s_1}, \overrightarrow{c_1}, \mathit{ts}_1, \mathit{mmts}_1 \xrightarrow{\mathit{tr}}_{\delta} \overrightarrow{s_2}, \overrightarrow{c_2}, \mathit{ts}_2, \mathit{mmts}_2} \qquad \boxed{\mathsf{mem}_1 \xrightarrow{\mathit{tr}} \mathsf{mem}_2} \qquad \boxed{M_1 \xrightarrow{\mathit{tr}}_{p} M_2}
$$

(MACHINE-STEP)

$$
\frac{\mathcal{T}_1[\mathit{tid}] = (\overrightarrow{s_1}, \overrightarrow{c_1}, \mathit{ts}_1, \mathit{mmts}_1) \quad \mathcal{T}_2 = \mathcal{T}_1[\mathit{tid} \mapsto (\overrightarrow{s_2}, \overrightarrow{c_2}, \mathit{ts}_2, \mathit{mmts}_2)] \quad \overrightarrow{s_1}, \overrightarrow{c_1}, \mathit{ts}_1, \mathit{mmts}_1 \xrightarrow{\mathit{tr}}_{p.\delta} \overrightarrow{s_2}, \overrightarrow{c_2}, \mathit{ts}_2, \mathit{mmts}_2 \quad \mathsf{mem}_1 \xrightarrow{\mathit{tr}} \mathsf{mem}_2}{(\mathcal{T}_1, \mathsf{mem}_1) \xrightarrow{\mathit{tr}|_{\mathsf{U}}}_{p} (\mathcal{T}_2, \mathsf{mem}_2)}
$$

(MACHINE-CRASH)

$$
\frac{\mathcal{T}_1[\mathit{tid}] = (\overrightarrow{s_1}, \overrightarrow{c_1}, \mathit{ts}_1, \mathit{mmts}_1) \quad \mathcal{T}_2 = \mathcal{T}_1[\mathit{tid} \mapsto (\overrightarrow{p.s_{\mathit{tid}}}, [], \mathit{ts}_{\mathsf{init}}, \mathit{mmts}_1)]}{(\mathcal{T}_1, \mathsf{mem}) \xrightarrow{[]}_{p} (\mathcal{T}_2, \mathsf{mem})}
$$

Figure 3.1: The syntax and semantics of our core PM language (excerpt).

(1) evaluates the initial value from e and assigns it to the dependent variable r; (2) executes the body $\overrightarrow{s}$; (3) in doing so, if `continue` e is executed, then the (merged) loop-carried dependent value evaluated from e is assigned to r, and $\overrightarrow{s}$ is re-executed for the next iteration; and (4) if `break` is executed, the loop terminates. A function call, $r \coloneqq f(\overrightarrow{e})$, evaluates the arguments $\overrightarrow{e}$, finds the function id f in the program's function environment δ with $\delta(f) = (\overrightarrow{\mathit{prms}}, \overrightarrow{s_f}) \in \overrightarrow{\mathrm{VReg}} \times \overrightarrow{\mathrm{Stmt}}$, and executes the function body $\overrightarrow{s_f}$ with a fresh variable context assigning the evaluated arguments to $\overrightarrow{\mathit{prms}}$. If `return` e is executed, then the control goes back to the caller and the return value evaluated from e is assigned to r.

We treat primitive detectable operations as language constructs and implement them on Intel-x86 later in §3.4. Primitive detectable operations comprise chkpt and pcas. A detectable checkpoint, $r \coloneqq \mathtt{chkpt}(\overrightarrow{s}, e_\mathsf{mid})$, evaluates $\overrightarrow{s}$ as if it is a function body, but using the same variable context as the operation's caller as a variable-capturing closure. A detectable CAS, $r \coloneqq \mathtt{pcas}(e_\mathsf{l}, e_\mathsf{o}, e_\mathsf{n}, e_\mathsf{mid})$, evaluates the expressions respectively to v_l, v_o, and v_n, attempts to update the PM location v_l from v_o to v_n atomically, and writes whether it succeeded to r. For both chkpt and pcas, their results and timestamps are checkpointed at the thread's sub-memento (located in its private PM) identified by the memento id (mid) evaluated from e_mid.

A thread consists of statements ($\overrightarrow{s}$), loop and function continuations ($\overrightarrow{c}$, definition omitted), a volatile state (ts), and a persistent memento (mmts). Continuations are pushed (resp. popped) for loop and call (resp. break and return) statements, respectively. A thread state, ts, consists of a register file (ts.regs) and the thread's last observed timestamp (ts.time, see §3.2.2). To maintain its invariant, ts.time is initialized with zero at thread initialization point (see **MACHINE-CRASH**), and incremented when a primitive operation is executed or replayed.

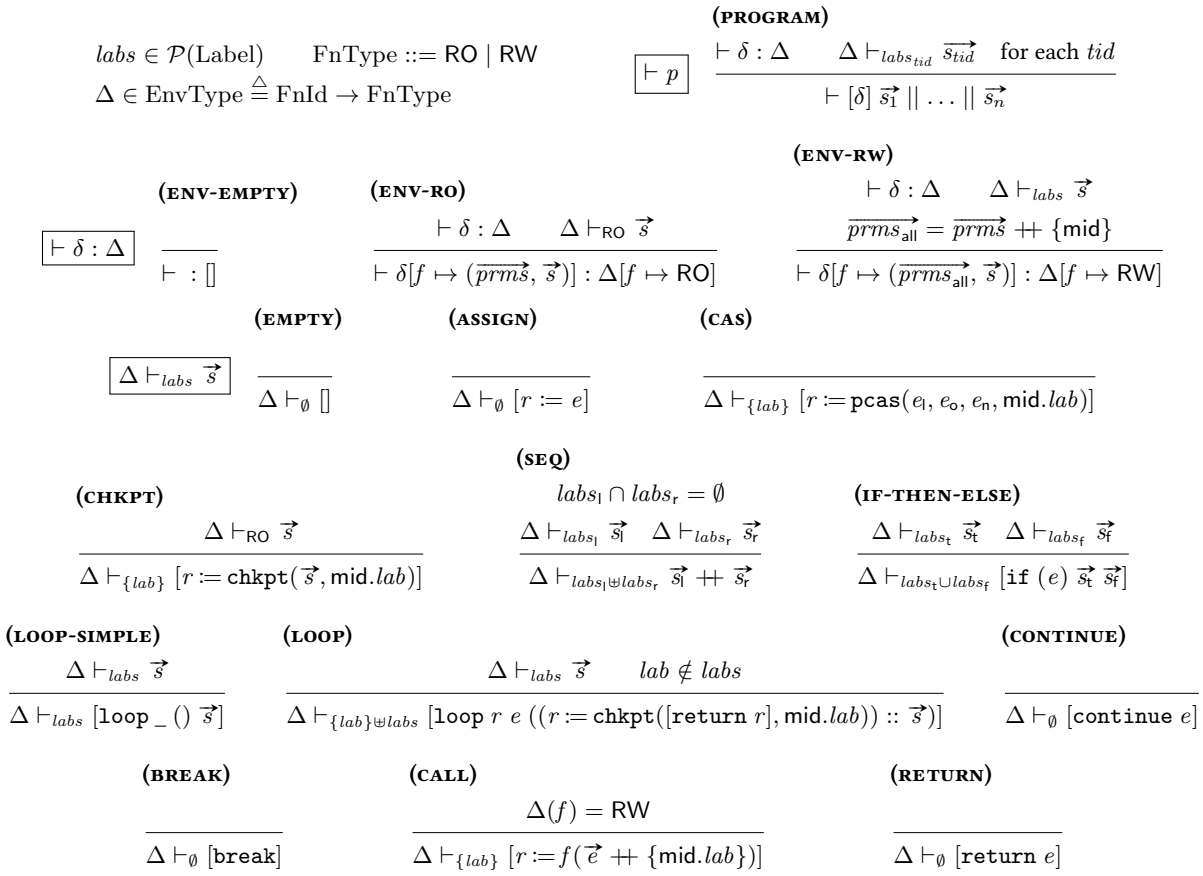


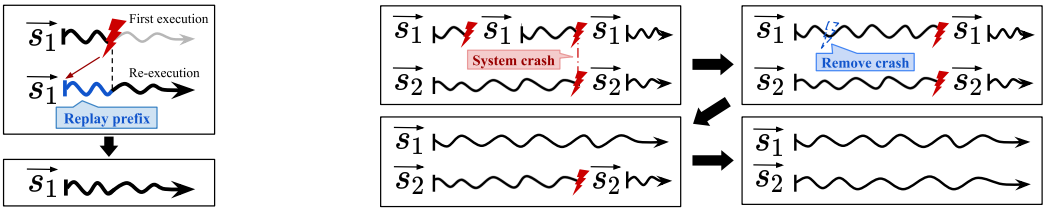
Figure 3.2: Type system (excerpt).

When executing a primitive operation op, we compare $ts.\mathsf{time}$ with the timestamp t_{mmt} checkpointed in the memento of op. If $ts.\mathsf{time} < t_{\mathsf{mmt}}$, then op was executed before the crash, and thus we simply update $ts.\mathsf{time}$ to t_{mmt}; otherwise, the replay is over and we execute op and update $ts.\mathsf{time}$ to a new timestamp. A memento is a map from memento ids (lists of labels) to primitive mementos that record values and timestamps; e.g. the id list.pnb denotes the primitive memento used at L3 in Algorithm 2. In our implementation, we statically reason about the structure and size of the memento for each operation with types. Lastly, a machine, M, consists of a list of threads ($\mathcal{T}$) and a memory (mem).

A judgement of the form $\vec{s_1}, \vec{c_1}, ts_1, mmts_1 \xrightarrow{tr}_{\delta} \vec{s_2}, \vec{c_2}, ts_2, mmts_2$ denotes a *thread transition* for environment δ, emitting a *trace* tr. A trace is a list of *events*; an event is either a read ($\mathsf{R}(l, v)$, reading v from shared PM location l) or an update ($\mathsf{U}(l, v_{old}, v_{new})$, atomically updating l from v_{old} to v_{new}). For read events, the values read from the shared memory are constrained not by thread transitions but by *memory transitions* of the form $\mathsf{mem}_1 \xrightarrow{tr} \mathsf{mem}_2$. Two transitions are combined into a *machine transition* of the form $M_1 \xrightarrow{tr}_p M_2$ for program p. The **MACHINE-STEP** rule states that a thread may execute a step tr, transitioning the memory with the same trace tr, emitting only updates externally ($tr|_{\mathsf{U}}$); the **MACHINE-CRASH** states that a thread may crash and re-execute the initial statements with an empty continuation, initial thread state, and the preserved memento.

3.3.2 Type System

We present our type system for detectable operations with deterministic replay in Fig. 3.2. The **PROGRAM** rule states that a program is typed if its function environment and each thread's statements are typed. A judgement of



(a) Deterministic Replay

(b) Removing Crashes with Deterministic Replay

Figure 3.3: Proving detectability by gradually removing crashes.

the form $\vdash \delta : \Delta$ denotes that for each function id f, the function $\delta(f)$ is detectable with type $\Delta(f) \in \mathrm{FnType}$. A function type is either RO, meaning the function only reads from shared PM locations and does not access mementos at all; or RW, meaning the function reads and writes to shared PM locations and accesses only those mementos prefixed by mid given as its last argument. The **ENV-EMPTY** rule states that the empty function environment is typed; **ENV-RO** adds a read-only function to the environment[6]; and **ENV-RW** adds a read-write function with the last parameter being the memento id mid. The judgement $\Delta \vdash_{labs} \vec{s}$ in the premise of **ENV-RW** states that for any function environment (δ) with type Δ, the execution of $\vec{s}$ satisfies the interpretation of RW while using only those sub-mementos prefixed by mid.lab for some $lab \in labs$.

For read-write functions, **EMPTY** states that the empty statement list is typed for any function environment type (Δ) using no mementos ($\emptyset$); and **ASSIGN**, **CONTINUE**, **BREAK** and **RETURN** state that so are assignment, continue, break, and return statements for all sub-expressions as they are pure.[7] The **SEQ** composes lists of statements so long as they use disjoint mementos ($labs_\mathsf{l} \cap labs_\mathsf{r} = \emptyset$) and sequential composition uses their disjoint union ($labs_\mathsf{l} \uplus labs_\mathsf{r}$). The **IF-THEN-ELSE** composes a conditional branch without requiring disjointness as only one branch is executed (see §3.2.1).

The **CAS** rule states that a pcas is typed against the memento label it uses (lab); the **CHKPT** behaves analogously so long as the checkpoint body ($\vec{s}$) is read-only. We require the body's result to be immediately checkpointed before being assigned to a register for deterministic replay. For instance, consider an execution of Algorithm 2 where among $prev$, $next$ and blk obtained at L3only $prev$ is checkpointed before a crash. The post-crash execution then re-calculates new values, $(prev', next', blk')$, and uses the old $prev$ from the memento but the new values $next'$, blk', mixing the results of different executions across crashes. This leads to a bug: as list traversal is non-deterministic, $prev$ and $next'$ may not be adjacent to each other, breaking the list invariant.

The **LOOP-SIMPLE** states that a loop without loop-carried dependence is typed if its body is ($\vec{s}$). Here, the loop-dependent variable "_" means it is written to nowhere, or equivalently, there are no dependent variables (§3.2.2). The **LOOP** states that a loop is typed if so is its body, its dependent variable (r) is checkpointed at the loop head, and the checkpoint and body use disjoint memento labels (§3.2.3). The **CALL** states that an RW function call is typed against the memento label it uses.

3.3.3 Detectability of Typed Programs

We sketch the proof of the detectability of typed programs and give the full proof in §B.6. Unlike the prior work [Friedman et al. 2018; Attiya et al. 2018], we formulate detectability in terms of behaviour refinement. For a program, p, we say event trace tr is a *behaviour* of p, written $tr \in B^{\lightning}(p)$, if there exists M such that $init(p) \xrightarrow{tr}{}^*_p M$, where $init(p)$ is the initial machine of p and $\xrightarrow{tr}{}^*_p$ is the reflexive transitive closure of the machine

[6] For brevity, we omit the definition of the read-only $\vdash_{\mathsf{RO}}$ judgement as it is straightforward (see §B.5 for the its definition).

[7] We do not establish the usual soundness result with our type system; e.g. while we can derive $\Delta \vdash_\emptyset [r_1 := r_2]$ for any Δ, r_1 and r_2, the $r_1 := r_2$ may get stuck as r_2 is a free variable. Though it is straightforward to adapt our system for soundness, we forgo this as this is not our aim and our type system is sufficient for our main goal: detectability by deterministic replay.

transition $\xrightarrow{tr}_p$ with concatenated event traces. A tr is a *crash-free behaviour* of p, written $tr \in B(p)$ if it is a behaviour from a crash-free machine execution using only **MACHINE-STEP**. We then prove the following theorem.

THEOREM 3.3.1 (DETECTABILITY). *Given a program p, if $\vdash p$ holds, then $B^{\lightning}(p) \subseteq B(p)$.*

This theorem ensures failure transparency in that crashes do not introduce additional behaviours; that is, this theorem ensures the detectable recoverability of typed programs.

We prove this theorem by gradually transforming an arbitrary execution of p into one without crashes while preserving the behaviour, as illustrated in Fig. 3.3. We exploit the fact that each thread interacts with the other components only via event traces: as long as event traces are preserved, we can *locally merge* a thread's consecutive executions across crashes into one without crashes. Subsequently, the resulting machine execution would produce the same behaviour as before with fewer crashes. Going forward, we will get a crash-free execution with the same behaviour.

Deterministic Replay We formulate the ability to locally merge thread executions in Theorem 3.3.2. We assume that a thread executes the statements $\vec{s}$ twice, before and after a crash. As such, the statements, continuations, and volatile thread state are initialized and the memento ($mmts_\omega$) is preserved. There then is an execution without crashes that results in the same memento ($\underline{mmts_\omega}$) while emitting an event trace (tr_x) that *refines* the original event trace ($tr +\!\!+ \underline{tr}$): we can reach tr_x from $tr +\!\!+ \underline{tr}$ by removing some read events. Trace refinement is sufficient to replace thread executions in a machine execution while preserving its behaviour, because machine transitions ignore read events and memory transitions are closed under trace refinement.

DEFINITION 3.3.2 (DETERMINISTIC REPLAY). *Let δ be a function environment and $\vec{s}$ be a list of statements. We say $\vec{s}$ is deterministically replayed for δ, denoted by* $\mathsf{DR}(\delta, \vec{s})$*, if the following holds:*

$$\begin{aligned}
&\forall tr, \underline{tr}, \vec{s_\omega}, \underline{\vec{s_\omega}}, \vec{c_\omega}, \underline{\vec{c_\omega}}, ts, ts_\omega, \underline{ts_\omega}, mmts, mmts_\omega, \underline{mmts_\omega}.\\
&\quad \vec{s}, [], ts, mmts \xrightarrow{tr}{}^*_\delta \vec{s_\omega}, \vec{c_\omega}, ts_\omega, mmts_\omega \longrightarrow \vec{s}, [], ts, mmts_\omega \xrightarrow{tr}{}^*_\delta \underline{\vec{s_\omega}}, \underline{\vec{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega} \longrightarrow\\
&\exists tr_x, \vec{s_x}, \vec{c_x}, ts_x.\ \vec{s}, [], ts, mmts \xrightarrow{tr_x}{}^*_\delta \vec{s_x}, \vec{c_x}, ts_x, \underline{mmts_\omega} \wedge tr_x \sim tr +\!\!+ \underline{tr}\ .
\end{aligned}$$

LEMMA 3.3.3. *Let δ be an environment, Δ be an environment type, $\vec{s}$ be a list of statements, and $labs$ be a set of labels. If we have $\vdash \delta : \Delta$ and $\Delta \vdash_{labs} \vec{s}$, then* $\mathsf{DR}(\delta, \vec{s})$.

This lemma states that typed statements are deterministically replayed. We prove it by strong induction on the derivations of $\vdash \delta : \Delta$ and $\Delta \vdash_{labs} \vec{s}$, formalizing the arguments presented in §3.2.

Erasure In the absence of crashes, a program p behaves equivalently to the *erasure* of p, written $erase(p)$, intuitively corresponding to removing the highlighted parts in §3.2. In particular, memento parameters and arguments are removed, checkpoint operations are removed, and `pcas` operations are replaced with plain `cas` operations. We thus obtain the following theorem.

THEOREM 3.3.4 (ERASURE). *Given a program p, If $\vdash p$ holds, then $B^{\lightning}(p) \subseteq B(erase(p))$.*

The theorem effectively reduces the complexity of designing detectable and persistent DS to that of designing volatile DS (already well-studied) and adapting volatile DS to our type system (straightforward). In particular, programmers no longer need to write challenging-to-develop and reason-about DS-specific recovery code, which is required by most hand-tuned persistent DSs. This way, we will straightforwardly design a wide variety of high-performance detectable DSs in §3.5.

3.4 Implementation of the Core Language

To show the feasibility and practicality of our core language in §3.3, we implement it on Intel-x86.

3.4.1 Framework

PM Primitive We use the App Direct mode of Intel-x86 Optane DCPMM to access PM locations with byte addressability via load, store, and CAS instructions. We use **clwb** instructions to ensure a write to a PM location is persisted: a store or CAS to a PM cache line cl is guaranteed to be persisted if followed by **clwb** cl and then **sfence**, **mfence**, or a successful CAS. We refer the reader to Cho et al. [2021b]; Raad et al. [2019b] for the formal semantics of **clwb**. We use Ralloc [Cai et al. 2020] for PM allocation and our modified version of Crossbeam [Developers 2019] for safe memory reclamation of shared PM locations (see §3.5.1 for more details on reclamation).

Crash Handler To emulate MACHINE-CRASH, we install a *crash handler* that continuously observes and handles crashes. (1) When a thread crashes, which may happen due to signals but not widely considered in the literature [Attiya et al. 2018; Ben-David et al. 2019; Attiya et al. 2022], the handler creates a new thread that executes the original thread's initial statements. It also initializes the thread state (ts), e.g. setting ts.time to zero, and runtime resources such as reclamation handle (see §3.5.1 for more details). (2) When the whole system crashes, the post-crash execution first executes the handler, which then initializes the system state *as if* every thread experiences just a thread crash instead of the system crash. Specifically, the handler performs Ralloc's garbage collection, initializes volatile data used by primitive operations (see §3.4.2 and §3.4.3 for details), and revives the threads.

Timestamp The core language assumes a consistent clock for multiple threads across crashes. We design such a clock on Intel-x86 using the `rdtscp` instruction generating hardware timestamps. The hardware clock is consistent for a single thread: *strictly increasing* and *serializing* in that `rdtscp` followed by **lfence** is not reordered with the surrounding instructions [Intel 2024a].

However, Kashyap et al. [2018] observe that the clock is *not* consistent for multiple threads across crashes as follows. (1) The clock is reset to zero when the machine is rebooted after a crash. (2) The clock has an inter-core skew due to misaligned delivery of the `RESET` signal at the system boot. As such, even if an `rdtscp` instruction *happens before* [Owens et al. 2009] another in a different thread, their timestamps may not be ordered. Still, the skew is *invariant*: constant regardless of dynamic frequency and voltage scaling. For the core language, we address these caveats as follows.

For reset, the crash handler calibrates the clock at the system boot. Specifically, it ❶ calculates the maximum timestamp checkpointed in all mementos, t_{max}; ❷ generates the current timestamp, t_{init}, using `rdtscp`; and ❸ adds offset $(-t_{\text{init}} + t_{\text{max}})$ to all timestamps generated by `rdtscp`. The calibrated timestamps are then always larger than those checkpointed before the system boot.

For skew, we relax the synchronization criteria of the clock. We follow Kashyap et al. [2018] to measure the maximum pair-wise inter-core skew, O_g. We then make the following observation.

OBSERVATION 1 (WEAK GLOBAL SYNCHRONIZATION). *Suppose a and b are* `rdtscp; lfence` *instruction sequences. If either $a \xrightarrow{po} b$ (single-thread program order) or $a \xrightarrow{hb} \textbf{wait}(O_g) \xrightarrow{hb} b$ (multi-thread happens-before), then a's timestamp is less than b's.*

Here, **wait**(O_g) is a spin loop to provide a sufficient margin for the clock skew. The single-thread program order and multi-thread happens-before order conditions are used in the implementation of checkpoint (§3.4.2) and CAS (§3.4.3) operations, respectively.

Algorithm 4 Detectable checkpoint

```
1:  function chkpt(s⃗, mid)
2:      t_0 := LOAD_PLN(mmts[mid][0].time)
3:      t_1 := LOAD_PLN(mmts[mid][1].time)
4:      t_mmt := (t_0 < t_1) ? t_1 : t_0
5:      (st, lt) := (t_0 < t_1) ? (0, 1) : (1, 0)
6:      if t_mmt > ts.time then
7:          ts.time := t_mmt
8:          v_r := LOAD_PLN(mmts[mid][lt].val)
9:          return v_r
10:     end if
11:     v_r := exec s⃗
12:     STORE_PLN(mmts[mid][st].val, v_r)
13:     if SIZEOF(mmt) > CLSIZE then
14:         flushopt mmts[mid][st].val; sfence
15:     end if
16:     t := rdtscp
17:     STORE_PLN(mmts[mid][st].time, t)
18:     flushopt mmts[mid][st].time; sfence
19:     ts.time := t
20:     return v_r
21: end function
```

The single-thread program order sufficiently separates two `rdtscp` instructions even if the thread is context-switched in-between. Even if the thread switches to a core with a negative timestamp offset, its effect, bounded by O_g (60ns at the maximum in our evaluation), is subsumed by the context switch latency (2-5µs at the minimum [Microsoft 2023; Blandy 2022]). Similarly, for the multi-thread happens-before condtition, O_g sufficiently separates two `rdtscp` instructions regardless of their executed cores because O_g is the maximum *inter-core* skew.

3.4.2 Detectable Checkpoint

We implement the `chkpt` operation of the core language (§3.3.1) on Intel-x86. Following Ben-David et al. [2019], we ensure the atomicity of `chkpt` (i.e. one never observes a partially checkpointed value) by double buffering: while a buffer is being written, the other buffer holds a valid value. Moreover, we record timestamps and values in PM to deterministically replay control flow (§3.2.2).

We present our implementation in Algorithm 4. To atomically update a timestamped value in the *abstract* memento (§3.3.1), its *concrete* implementation uses two timestamped values, *stale* and *latest*. The algorithm then ❶ compares the given memento's two timestamps (st and lt) to distinguish stale from latest (L2-L5); ❷ if the memento's timestamp (t_{mmt}) is greater than the thread's replaying timestamp (ts.time), then the operation was already performed before the crash. In this case, ts.time is incremented to t_{mmt} first, and then the pre-crash result is replayed by simply returning the old returned value (L6-10); ❸ write the result of the given statements to the memento's stale buffer (L12); ❹ flush the stale buffer, unless the memento fits in a cache line so that the buffer is anyway flushed at L18, following van Renen et al. [2020]'s optimization technique (L14); and ❺ update the stale timestamp to the current timestamp (L17), flush it (L18), update ts.time (L19), and return the result (L20). Here, "**flushopt** l" is a shorthand for performing **clwb** cl on all cache lines cl that spanning location l.

3.4.3 Detectable Compare-and-Swap

We implement the `pcas` operation of our core language (§3.3.1) on Intel-x86. Following Attiya et al. [2018]; Ben-David et al. [2019], our `pcas` on location l comprises three phases: *locking* l with an architecture-provided plain CAS, *committing* the operation with PM writes, and *unlocking* l with another plain CAS. If a thread observes a locked location, it *helps* the ongoing operation to guarantee lock freedom. When helping, it is crucial to notify such a fact to the helped thread to ensure deterministic replay; otherwise, in the case that a thread performs a locking CAS, crashes, and gets helped, then the thread would incorrectly perform the same CAS (that is already performed by the helper) again in the post-crash execution. While the helping mechanism of Attiya et al. [2018] (resp. Ben-David et al. [2019]) requires an array of $O(T^2)$ (resp. $O(T)$) sequence numbers in PM for each location

(where T is the number of threads), we reduce the space consumption in PM to only 8 bytes per location. The key idea is comparing timestamps as for loops (§3.2.2).

Components An 8-byte location consists of 1-bit *parity* for helping, 1-bit *helping flag* to prevent ABA, 8-bit thread id (0 reserved for the `pcas` algorithm and 1-255 usable), 54-bit address annotated with user tag (64TB with 8-bit tag or 256GB with 16-bit tag)[8]. The tag is reserved for users to annotate arbitrary bits to pointer values for correctness [Harris 2001] or optimization [Chen et al. 2020]. We assume the ENCODE and DECODE functions respectively convert a ⟨parity, thread id, offset⟩ tuple to a location and vice versa.

As with the `chkpt` operation, `pcas` ensures atomicity by double buffering, storing two copies of a value and an annotated timestamp in its implementation of primitive memento. A 62-bit timestamp generated from `rdtscp` (sufficient for about 47 years without overflow) is annotated with a 1-bit *parity* and a 1-bit *success* flag, forming 8 bytes in total. We assume ENCODET and DECODET convert a ⟨parity, success flag, timestamp⟩ tuple into an annotated timestamp and vice versa.

For helping, our framework tracks several timestamps in DRAM and PM. The ts.`cas` timestamp in DRAM records the parity-annotated timestamps of each thread's last CAS operation across crashes, while the global arrays $HELP[2][T]$ in PM record the timestamp of the last helping for each parity and thread, written by the helpers. Our framework maintains the invariant that the thread ts's CAS was helped if ts.`cas` is less than $HELP[p][ts]$ for some appropriate parity p (see below).

The crash handler initializes ts.`cas` with the maximum timestamp checkpointed in `pcas` primitive mementos when a thread crashes, and uses $HELP$ to calculate $t_{\max}$ for clock calibration when the system crashes (§3.4.1).

Load We present our `pload` and `pcas` implementation in Algorithm 5. As `pcas` acquires a lock by temporarily tagging parity, success flag, and thread id to the location value in PM, we also implement `pload` that helps the ongoing `pcas` to release the lock, ensuring it reads a value persisted in PM. Specifically, $\textsc{Load}_{\textsc{pln}}$(L1) performs an architecture-provided plain load and invokes HELP (see below for details on helping). As such, both operations are oblivious to tags: their input and output location values are tagged with zero.

CAS: Normal Execution The `pcas` operation (L5) begins by identifying the stale and latest values in the memento (L9). It then performs two main tasks: (1) determining whether the CAS operation was completed or crashed while executing previously with the latest values in the memento, and if so, returning the previous result value (L12-27); (2) if not, executing an actual CAS operation (L28-52). For easier understanding, we describes the second task first.

The CAS operation ❶ tries to lock the location by performing a plain CAS to the new value annotated with the next parity ($\neg p_{\mathsf{own}}$) and the thread id (tid, L30-34); ❷ if unsuccessful, it finishes the operation after updating ts.`time` and persisting the failure to the memento (L36-41); ❸ ensures the operation is committed by flushing the plain CAS (L43); and ❹ completes the operation after updating ts.`time` (L44) and ts.`cas` (for the next CAS operation) (L46), persisting the success to the memento (L48), attempting to unlock the location by atomically clearing annotations (L50), and (regardless of the result) ensuring the writes to the memento are flushed (L51).

CAS: Replay To demonstrate that the execution of `pcas` is deterministically replayed, we first define the following events of a pre-crash execution. *Commit* is the flush of the first plain CAS at L30. Note that this event does not coincide with the flush instruction at L43, as a write can be voluntarily flushed before requested. *Checkpoint* is the flush of memento writes at L39 and L47. *Unlock* is the flush of the second plain CAS at L50.

[8] If 64 or more bits are necessary, a 118-bit integer supporting detectable CAS can be constructed from 128-bit machine words and double-word machine CAS operations.

Algorithm 5 Load and Detectable CAS for Location Values

```
1:  function pload(loc)                                  ▷ for location values
2:      cur := LOAD_PLN(loc)
3:      return HELP(loc, cur)
4:  end function

5:  function pcas(loc, old, new, mid)
6:      t_0 := LOAD_PLN(mmts[mid][0].time)
7:      t_1 := LOAD_PLN(mmts[mid][1].time)
8:      t_mmt := (t_0 < t_1) ? t_1 : t_0
9:      (st, lt) := (t_0 < t_1) ? (0, 1) : (1, 0)
10:     pst_mmt := LOAD_PLN(mmts[mid][lt].time)
11:     (par_mmt, suc_mmt, t_mmt) := DECODET(pst_mmt)
12:     if t_mmt > ts.time then
13:         ts.time := t_mmt
14:         if suc_mmt then return (true, old)
15:         v_r := LOAD_PLN(mmts[mid][lt].val)
16:         return (false, v_r)
17:     end if
18:     _ := pload(loc)
19:     (par_own, _, t_own) := DECODET(ts.cas)
20:     t_help := LOAD_PLN(HELP[¬par_own][tid])
21:     if t_own < t_help then
22:         ts.time := t_help
23:         pst_suc := ENCODET(¬par_own, true, t_help)
24:         STORE_PLN(mmts[mid][st].time, pst_suc)
25:         flushopt mmts[mid][st].time; sfence
26:         return (true, old)
27:     end if
28:     old' := ENCODE(EVEN, false, 0, old)
29:     new' := ENCODE(¬par_own, false, tid, new)
30:     r_1 := CAS_PLN(loc, old', new')
31:     t := rdtscp; lfence
32:     if r_1 is (ERR cur) then
33:         cur := HELP(loc, cur)
34:         if cur = old then goto 30
35:         ts.time := t
36:         pst_fail := ENCODET(EVEN, false, t)
37:         ts.cas := pst_fail
38:         STORE_PLN(mmts[mid][st].val, cur)
39:         STORE_PLN(mmts[mid][st].time, pst_fail)
40:         flushopt mmts[mid][st]; sfence
41:         return (false, cur)
42:     end if
43:     flushopt loc; sfence
44:     ts.time := t
45:     pst_suc := ENCODET(¬par_own, true, t)
46:     ts.cas := pst_suc
47:     STORE_PLN(mmts[mid][st].time, pst_suc)
48:     flushopt mmts[mid][st].time
49:     new'' := ENCODE(EVEN, false, 0, new)
50:     r_2 := CAS_PLN(loc, new', new'')
51:     if r_2 is ERR then sfence
52:     return (true, old)
53: end function
```

Based on the timing of a crash, the memory state that can be observed during post-crash execution can be categorized as follows:

($\lightning_1$) *Before commit*: the latest timestamp in the memento (t_{mmt}) is less than or equal to[9] the thread's last observed timestamp ($ts.\mathsf{time}$).

($\lightning_2$) *Between commit and checkpoint*: t_{mmt} is still less than or equal to $ts.\mathsf{time}$. The location (loc) can have one of two states: ($\lightning_{2a}$) loc is still locked by the thread; or ($\lightning_{2b}$) loc is not locked by the thread as it is unlocked by another thread's helping.

($\lightning_3$) *After checkpoint*: t_{mmt} is greater than $ts.\mathsf{time}$.

The replay algorithm (L12-27) exhaustively covers all the crash cases mentioned above. After decoding the memento's annotated timestamp (L11), it compares t_{mmt} and $ts.\mathsf{time}$. If t_{mmt} is greater than $ts.\mathsf{time}$ (corresponding to $\lightning_3$), the pre-crash execution is replayed: it updates $ts.\mathsf{time}$ and if pcas was successful, returns true and old (L14); otherwise returns false and the value stored in the memento (L16). If t_{mmt} is less than or equal to $ts.\mathsf{time}$, it helps the location's ongoing pcas if it exists (L18), which transitions the sub-case $\lightning_{2a}$ to $\lightning_{2b}$. To distinguish

[9] If the memento function is within a loop, it is possible for the timestamp of the memento and the thread's last observed timestamp to be equal.

Algorithm 6 Help for Detectable CAS

```
1:  function Help(loc, old)
2:      (par_old, dsc_old, tid_old, o_old) := decode(old)
3:      if tid_old = 0 then return o_old
4:      wait(O_g)
5:      t := rdtscp; lfence
6:      wait(O_g)
7:      cur := Load_PLN(loc)
8:      if old ≠ cur then old := cur; goto 2
9:      flushopt loc
10:     r_dsc := RegisterDesc(loc, old)
11:     if r_dsc is (Ok cur) then
12:         (par_old, _, tid_old, o_old) := decode(cur)
13:     else
14:         old := Load_PLN(loc); goto 2
15:     end if
16:     t_help := Load_PLN(HELP[par_old][tid_old])
17:     if t ≤ t_help then
18:         old := Load_PLN(loc); goto 2
19:     end if
20:     r := CAS_PLN(HELP[par_old][tid_old], t_help, t)
21:     if r is Err then
22:         old := Load_PLN(loc); goto 2
23:     end if
24:     flushopt HELP[par_old][tid_old]
25:     old' := encode(Even, false, 0, o_old)
26:     if CAS_PLN(loc, old, old') is (Err cur) then
27:         old := cur; goto 2
28:     end if
29:     flushopt loc; sfence
30:     return o_old
31: end function

32: function RegisterDesc(loc, old)
33:     (_, dsc_old, tid_old, seq_old) := decode(old)
34:     if dsc_old is true then
35:         goto 45
36:     end if
37:     seq_next := Load_PLN(DESC[tid].seq) + 1
38:     Store_PLN(DESC[tid].seq, seq_next)
39:     Store_PLN(DESC[tid].new, old)
40:     flushopt DESC[tid]
41:     desc' := encode(Even, true, tid, seq_next)
42:     CAS_PLN(loc, old, desc')
43:     old := Load_PLN(loc)
44:     (_, dsc_old, tid_old, seq_old) := decode(old)
45:     new := Load_PLN(DESC[tid_old].new)
46:     if dsc_old ∧ seq_old == DESC[tid_old].seq then
47:         return Ok new
48:     end if
49:     return Err
50: end function
```

between cases ↯$_1$ and ↯$_{2b}$, the last timestamp increased by helper and the timestamp of the thread's last CAS operation should be compared. To this end, it decodes ts.cas, retrieves the parity and timestamp (t_{own}), and loads the helping timestamp (t_{help}) using the opposite parity (see below for details of parity and timestamp on helping). If t_{help} is greater than t_{own} (corresponding to ↯$_{2b}$), it detects (from the invariant of ts.cas and $HELP$) that the last CAS operation actually succeeded and finalizes the operation (L21-27). Otherwise (corresponding to ↯$_1$), it proceeds to the normal execution (L28-52).

Helping We present our Help implementation in Algorithm 6. For lock-freedom, a thread may invoke Help(loc, old) (L1) for loc's ongoing pcas operation to be flushed, unlock it and to return an unlocked (i.e. untagged) location value. It ❶ returns the given value old read from loc if is already unlocked (L3); ❷ waits for O_g, reads the current timestamp (t), and waits for O_g again to make t synchronized across other threads (L4-L6, see Observation 1); ❸ loads a value, say cur, from loc again, and if $old \neq cur$, then retries from L2 (L8); ❹ ensures the ongoing operation is committed by flushing loc (L9); ❺ registers the helping descriptor flag to prevent ABA (L10-15); ❻ loads $HELP[par_{\mathsf{old}}][tid_{\mathsf{old}}]$, the last CAS help's timestamp for the parity and thread id annotated in old, and if it is bigger than t, retries the operation (L16-19); ❼ performs a plain CAS and flush on $HELP[par_{\mathsf{old}}][tid_{\mathsf{old}}]$ to atomically increase it to t, and if unsuccessful, retries the operation because the CAS has been already helped (L20-24); ❽ tries to unlock the location with a plain CAS and a flush, and if unsuccessful, retries the operation (L25-29); and ❾ returns the unlocked location (L30).

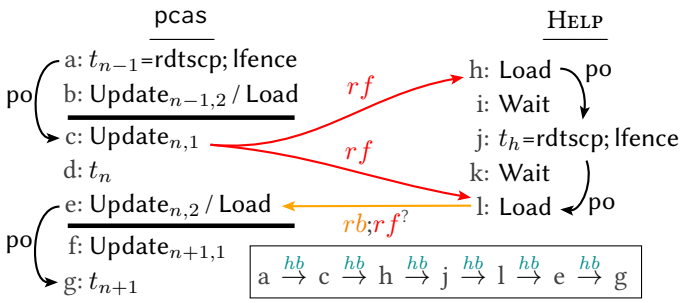


Figure 3.4: Synchronization of pcas() and Help().

For deterministic replay, we show that Help() updates $HELP$ for a re-execution of pcas to enter the branch at L21 if and only if the previous execution of pcas crashed between *commit* and *checkpoint* of success ($\lightning_2$). To this end, it is sufficient to prove the following.

Lemma 3.4.1. *Let p_n and t_n denote the parity and timestamp of tid's n^{th} pcas invocation. The sequence $\{p_i\}$ then alternates between even and odd numbers, and the sequence $\{t_i\}$ is strictly increasing. Then $t_{n-1} < HELP[p_n][tid]$ if and only if either the n^{th} or a later CAS with parity p_n was helped.*

Proof. Suppose a Help operation generates a timestamp t_h at L5 and tries to help the second plain CAS of thread tid's n-th CAS invocation, as illustrated in Fig. 3.4. Here, we depict the plain CASes and timestamp generations of tid's $(n-1)^{\text{st}}$ to $(n+1)^{\text{st}}$ CAS invocations, and loads and timestamp generations of a Help invocation, where Update$_{n,i}$ represents the i^{th} plain CAS of tid's n^{th} CAS, and Load represents a load from a location. Then we have the following properties from Observation 1: (1) $t_{n-1} < t_h$ from $a \xrightarrow{po} c \xrightarrow{rf} h \xrightarrow{po} i \xrightarrow{po} j$; and (2) $t_h < t_{n+1}$ from $j \xrightarrow{po} k \xrightarrow{po} l \xrightarrow{rb;rf^?} e \xrightarrow{po} g$, where po is the program order; rf is the reads-from relation from each write to its readers; rb is the reads-before relation from each read to the later writes; $rb;rf^?$ is the reads-before relation possibly followed by a reads-from relation; and all relations constitute the *happens-before* relation hb in the x86-TSO memory model (see Owens et al. [2009] for more details).

Recall that Help persists Update$_{n,1}$ (L9), atomically increases $HELP[p_n][tid]$ to t_h (L20-L24), and helps Update$_{n,2}$ (L25-L29). If thread tid's n^{th} CAS was helped, then we have $t_{n-1} < t_h \leq HELP[p_n][tid]$ due to property **(1)**. Conversely, if $t_{n-1} < HELP[p_n][tid]$, then it cannot be the result of a help for $(n-2)^{\text{nd}}$ or earlier CASes or those with parity $\neg p_n$ due to property **(2)**. □

Preventing ABA To see why an ABA problem may happen, consider the following scenario for a location, say l:

(1) Thread A is performing pcas. It locks l, atomically changing the value from v_1 to v_2 with the first plain CAS (L30 in Algorithm 5).

(2) Thread B is performing Help(). It is about to help T_1's operation, read a timestamp, and sleep for a while (L5 in Algorithm 6).

(3) Thread A finishes pcas by itself, and performs another pcas from v_2 to v_2. Again, it locks l, atomically changing the value from v_2 to v_2 with the first plain CAS (L30 in Algorithm 5).

(4) Thread B validates its helping operation by reading v_2 at L7 in Algorithm 6. Since the old and new values are the same as v_2, and Thread B thinks it can help Thread A's pcas.

(5) Thread A is crashed, and Thread B *wrongly* helps Thread A's operation by recording an *old* timestamp (generated before the sleep) to $HELP$.

(6) Even though Thread A's operation was helped, Thread A thinks that it failed because an old timestamp is recorded in $HELP$.

We prevent such an ABA problem with sequence numbers in helping descriptors. More specifically, we introduce a shared PM array, $DESC$, that records helper's information. For each tid, $DESC[tid]$ records the helper tid's information consisting of the new location value to write as the result of helping, annotated with tid and parity; and a unique sequence number. Instead of helping the second plain CAS directly at L26 in Algorithm 6, the helper announces its intention to help by atomically updating the location to a tuple consisting of the helping bit, helper's tid, and helping's sequence number. Once an intention was announced, it can be served by any helpers. Thanks to the unique sequence number, now helpers cannot be confused with two `pcas` operations that atomically updates to the same value, thus preventing the ABA problem.

3.5 Implementation of Concurrent Data Structures

As primitive detectable operations, we implement ***Chkpt-mmt***: `chkpt` (§3.4.2); ***CAS-mmt***: `pcas` (§3.4.3); ***Indel-mmt***: *insertion/deletion* for atomic locations that performs fewer flushes than `pcas`. These primitives capture the essence of optimization in Friedman et al. [2018]; Li and Golab [2021]'s hand-tuned detectable Michael-Scott queues (MSQs) [Michael and Scott 1996] (see §B.1 for details). Accordingly, we extend the core language to support additional primitive operations, including ***Vol-mmt***: a volatile location for cached values requiring no flushes (see §B.2.1 for details); and ***Comb-mmt***: an adaptation of Fatourou et al. [2022]'s general *combiner* for persistent DSs to our framework. While the original combiner is detectable, it only supports a *single* invocation of each operation by each thread, e.g. the following statements are not detectably recoverable:

1: $v_1 \coloneqq \textsc{Dequeue}(q)$; $\quad v_2 \coloneqq \textsc{Dequeue}(q)$; $\quad \textsc{Enqueue}(q, v_1 + v_2)$

If an execution crashes while performing Dequeue, we cannot detect whether it was for v_1 or v_2. In contrast, we distinguish the two invocations by distinct sub-mementos.

Using the primitives and our type system, we implement the following detectable, persistent DSs: ***List-mmt***: CAS-based lock-free linked-list; ***TreiberS-mmt***: CAS-based Treiber stack [Treiber 1986]; ***MSQ-mmt-O0***: CAS-based MSQ; ***MSQ-mmt-O1***: MSQ based on *Indel-mmt* and *Vol-mmt*; ***CombQ-mmt***: combining queue based on *Comb-mmt*; and ***Clevel-mmt***: CAS-based lock-free resizing hash table of Chen et al. [2020][10], which we optimize with an advanced type rule, **loop-try** (see §B.2.2 for details). Theorem 3.3.4 guarantees the detectability of these implementations. In addition, we implement ***MSQ-mmt-O2***: a variant of *MSQ-mmt-O1* with an *invariant-based optimization*, which reduces PM flushes based on the invariant that certain location values are always persisted (see §B.2.3 for details).

3.5.1 Safe Memory Reclamation

All pointer-based lock-free DSs in DRAM should deal with the problem of safe memory reclamation (SMR). For instance, Treiber's stack [Treiber 1986] is basically a linked-list of elements with the head being the stack top, where a thread detaches the head block while the other threads may hold a local pointer to the same block. Due to the local pointer, the reclamation of the detached block should be *deferred* until every thread no longer holds such local pointers. Often, the SMR problem is systematically handled with *reclamation schemes* such as hazard pointers (HP) [Michael 2004] and epoch-based reclamation (EBR) [Fraser 2004].

Prior works on persistent lock-free DSs in PM also handle the SMR problem with reclamation schemes: Friedman et al. [2018] use HP; and DSS [Li and Golab 2021] and (nb)Montage [Wen et al. 2021; Cai et al. 2021]

[10] We use a bug-fixed version due to Chen et al. [2022].

use EBR. However, the prior work does not discuss how to ensure durable linearizability or detectability in the presence of memory reclamation in details. In fact, we discover and fix a use-after-free bug in the queues of Friedman et al. [2018]; Li and Golab [2021] in case of crashes due to a lack of flush. In this section, we present four concrete guidelines for safe memory reclamation of PM.

Introducing the Retire Primitive Operation SMR schemes provide the *retire* function that receives a detached block and reclaims it when every thread no longer references it. We extend the core language to support such a retire function, $\texttt{retire}(e)$, and the type system to admit the following rule:

$$\textsc{(retire)}\qquad \frac{}{\Delta \vdash_{\emptyset} [\texttt{retire}(e)]}$$

Here, e is an expression that evaluates to a PM location.

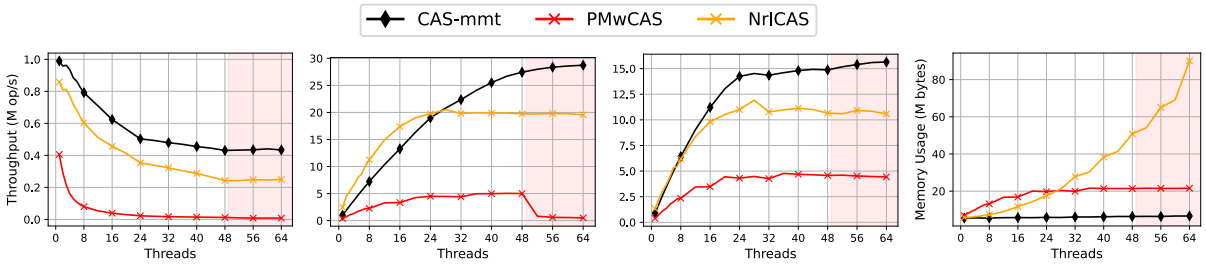
Clearing Local Pointers in Mementos on PM Before reclamation, we should clear local pointers not only in program variables on DRAM but also in mementos on PM. Otherwise, blocks may be reclaimed, the thread crashes, and then the operation retrieves and dereferences its memento's local pointers, invoking use-after-free error. We prevent this error by clearing mementos just before the end of a *critical section*. More specifically, we (1) wrap a thread's operation on its memento within a critical section; and (2) at the end of a critical section, clear the memento by overwriting null to all of its PM locations. For clearing's crash consistency, we install a per-thread, 1-bit "clearing" flag, which is toggled and flushed at the beginning and the end of the clearing. Should a crash happens, the crash handler first checks whether the flag is set, and if so, resumes clearing.

Flushing Location before Retirement The queues of Friedman et al. [2018]; Li and Golab [2021] have a use-after-free bug caused by the lack of a flush before retirement. Their delete operations detach a block, say blk, from a location, say loc, of the DS and retire it without flushing loc. Then the code may incur a use-after-free error in the following execution scenario: (1) blk is reclaimed; (2) the system crashes; (3) loc was *not* persisted and it still points to blk; and (4) the post-crash execution dereferences blk which is already reclaimed.

A straightforward fix would insert a flush between the CAS and the retirement, but we observe that such a flush is detrimental to the performance. We mitigate the slowdown by exploiting the following property of EBR: if a memory block is retired in a critical section, then it is never reclaimed in the same critical section. Now instead of flushing loc, we enforce the code to *defer* the flush of loc and actually perform such flushes in batch at the end of a critical section. (Even if an execution is crashed without persisting loc, we can restore loc's new value after the crash by traversing the DS.) Batching is beneficial for two reasons: (1) we can merge multiple flushes; and (2) we can asynchronously finish a critical section so that the flushes are performed not in the critical path. In our evaluation, we observe that the deferred flushes incur no noticeable runtime overhead.

Allowing Double Retirement A straightforward application of SMR schemes may incur double-free error in case of a thread crash due to the thread's inability to detect whether a block is retired. Suppose a thread crashes right after retiring a block. Then the post-crash recovery execution would re-retire the same block, because it cannot detect whether the block was already retired, causing typical SMR schemes to free the block twice.

We prevent this error by (1) retaining the critical section of a crashed thread and reviving it for the post-crash recovery execution; and (2) relaxing the retirement condition by allowing double retirements in a single critical section. For the former, we propose a new API for retrieving the critical section by thread id; for the latter, we install a buffer of retired blocks and deduplicate it at the end of a critical section.



(a) Throughput (1 location) (b) Throughput (1K locations) (c) Throughput (1M locations) (d) Memory usage (1M locations)

Figure 3.5: Multi-threaded throughput of detectable CASes.

3.6 Evaluation

We evaluate the detectable recoverability (§3.6.1) and performance (§3.6.2) of our detectable CAS, list, queues and hash table. We implement our framework and DSs in Rust nightly-2022-05-26 [Team 2019] and build them with release mode. We use a machine running Ubuntu 20.04 and Linux 5.4 with dual-socket Intel Xeon Gold 6248R (3.0GHz, 24 cores, 48 threads) and an Intel Optane DCPMM (100 Series, 256GB). We pin all threads to a single socket to keep all DCPMM traffic within the same NUMA node. For brevity, we present only key results here; see §B.3 for the full results.

3.6.1 Detectability

We evaluate the detectability of two distinct crash scenarios: thread crashes and system crashes. Thread crashes present a more non-deterministic and challenging aspect to address in comparison to system crashes. Conversely, system crashes provide an opportunity to examine if data is accurately retained in persistent memory, thereby enabling the detection of missing flush bugs in weak persistency memory models [Cho et al. 2021b].

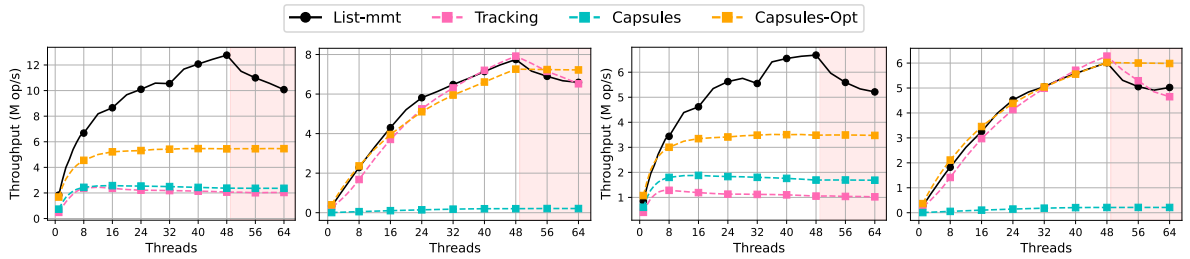
To perform stress test under thread crashes, we randomly crash an arbitrary thread. To crash a specific thread, we use the `tgkill` system call to send the `SIGUSR1` signal to the thread and let its signal handler abort its execution. To the best of our knowledge, this is the first general stress test for thread crashes carried out for detectable, persistent DSs. For the integration test of CAS and each DS, **we observe no test failures for 100K runs with thread crashes**.

Provoking an actual system crash in a controlled and efficient manner is challenging within conventional systems. Instead, we perform stress test under *simulated* system crashes by running model-checking tools, Yashme [Gorjiara et al. 2022b] and PSan [Gorjiara et al. 2022a], in the "random" mode, which does not enumerate all possible executions and thus possibly fails to detect existing bugs. We use the random mode to avoid state explosion. For the integration test of CAS and each DS, **we observe no test failures for 1K runs with simulated system crashes.**

3.6.2 Performance

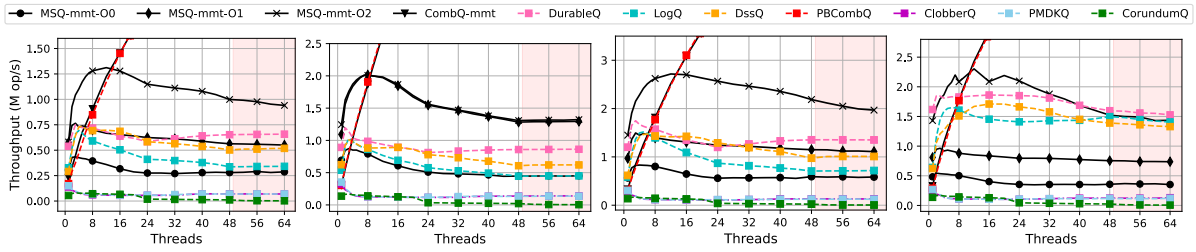
Unless specified otherwise, we measure the throughput for a varying number of threads: 1 to 8 and the multiples of 4 from 12 to 64; we report the average throughput of 5 runs, each for 10 seconds.

CAS Fig. 3.5 presents the throughput and memory usage of our *CAS-mmt*; ***PMwCAS***: detectable multi-word CAS by Wang et al. [2018]; and ***NrlCAS***: detectable CAS by Attiya et al. [2018]. We reimplement *PMwCAS* in



(a) Read intensive (20 key range) (b) Read intensive (2K key range) (c) Update intensive (20 key range) (d) Update intensive (2K key range)

Figure 3.6: Multi-threaded throughput of persistent lists.



(a) Enqueue-dequeue (b) Enqueue-20% (c) Enqueue-50% (d) Enqueue-80%

Figure 3.7: Multi-threaded throughput of persistent queues.

Rust to use the same allocator as the other CASes; we implement *NrlCAS* in Rust because its source code is not publicly available. Over-subscription over 48 threads is indicated as shaded regions. (1) When multiple threads perform CASes randomly on a varying number of locations (Fig. 3.5a, Fig. 3.5b, Fig. 3.5c), *CAS-mmt* exhibits higher throughput than the others for every thread count, except for *NrlCAS* with low thread counts for 1K locations. (2) When multiple threads perform CASes randomly on 1M locations (Fig. 3.5d), *NrlCAS* indeed consumes $O(T^2)$ PM locations, where T is the number of threads. (3) *PMwCAS* exhibits lower throughput than reported by Wang et al. [2018], because PM was not generally available at the time of writing and they experimented with DRAM. Also, *PMwCAS* generally exhibits lower throughput than single-word CASes because it supports multi-word CAS.

List Fig. 3.6 illustrates the throughput of *List-mmt*; ***Capsule***: detectable linked-list by Ben-David et al. [2019]; and ***Capsule-Opt***: optimized detectable linked-list and ***Tracking***: detectable linked-list by Attiya et al. [2022]. We use the DS implementation and evaluation workloads of Attiya et al. [2022]: from a random initial list, read-intensive workloads perform inserts, deletes and finds for 15%, 15%, and 70% times; and update-intensive workloads perform them for 35%, 35%, and 30% times. (1) For small key ranges, *List-mmt* significantly outperforms the others thanks to fewer flushes to PM of timestamp-based replay (Fig. 3.6a, Fig. 3.6c); and (2) for large key ranges, all lists are saturated at almost the same performance because search dominates the cost (Fig. 3.6b, Fig. 3.6d).

Queue We compare the throughput of our queues; ***DurableQ***: undetectable durable MSQ by Friedman et al. [2018]; ***LogQ***: detectable MSQ by Friedman et al. [2018]; ***DssQ***: detectable MSQ by Li and Golab [2021]; ***PBcombQ***: detectable combining queue by Fatourou et al. [2022]; ***ClobberQ***: transaction-based queue in Clobber-NVM [Xu et al. 2021]; ***PMDKQ***: transaction-based queue in PMDK [Intel 2024d]; and ***CorundumQ***: transaction-based queue in Corundum [Hoseinzadeh and Swanson 2021]. We reimplement *DurableQ* and *LogQ* in Rust for a use-after-free bug (§3.5.1); reimplement *PBcombQ* in Rust because it does not implement detectable recovery and uses a custom

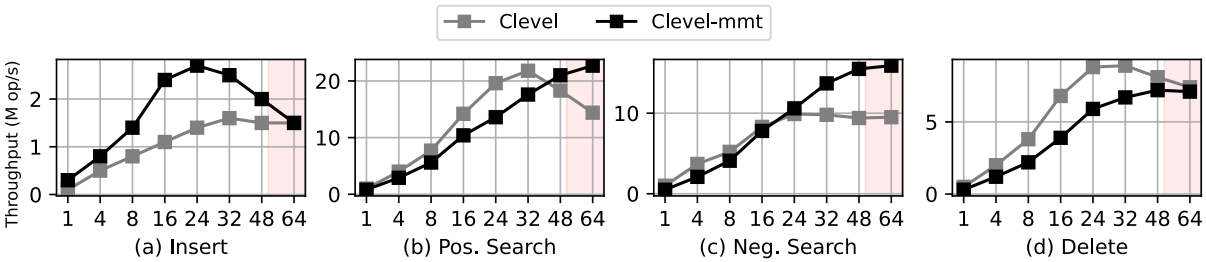


Figure 3.8: Multi-threaded throughput of hash tables for uniform distributions.

allocator; and implement *DssQ* in Rust because its source code is not publicly available.

Fig. 3.7 shows the throughput of queues for four workloads: ***enqueue-dequeue***: each operation enqueues an item and then immediately dequeues an item; ***enqueue-X%*** (with X=20, 50 or 80): each operation enqueues (or dequeues) an item for the probability of X%. We initialize the queues with 10M items to prevent excessive empty dequeues. (1) Transaction-based queues are noticeably slower than MSQs and combining queues. (2) Combining queues significantly outperform MSQs at high thread counts, in line with observations by Fatourou et al. [2022]. Thus, we cut the graphs to focus on MSQs rather than combining queues. (3) Although not shown in the graphs, it's worth noting that *CombQ-mmt* incurs a slight overhead over *PBcombQ*, especially for dequeue operations, because the latter saves a flush by assuming that a thread does not invoke an operation multiple times (see above). (4) *MSQ-mmt-O2* outperforms hand-tuned persistent MSQs with and without detectability thanks to fewer flushes to PM of timestamp-based deterministic replay (§3.2.2, see also §3.7). In addition, *DurableQ*'s dequeue incurs PM block allocation to store the return value. (5) *MSQ-mmt-O1* performs comparably with hand-tuned MSQs for dequeue-heavy workloads but not for enqueue-heavy workloads, because without an invariant-based optimization, its enqueue performs two plain CASes. (6) *MSQ-mmt-O0* is outperformed by hand-tuned MSQs due to its CAS-based dequeue flushing the head pointer and invalidating its cache line for every thread.

Hash Table We compare the throughput of *Clevel-mmt* with the original, undetectable ***Clevel*** [Chen et al. 2020] using the PiBench benchmark [Lersch et al. 2019] specifically designed for PM hash tables. We employ the *Clevel* implementation and evaluation workloads from a PM hash table evaluation paper [Hu et al. 2021], which consists of seven workloads (insert, positive and negative search, delete, and write-heavy, balanced, and read-heavy) and two key distributions: uniform and skewed (80% of accesses target 20% of keys); see §B.3 for full details.

Fig. 3.8 illustrates multi-threaded throughput of hash tables under uniform key distributions. The results for the skewed distribution are similar. (1) *Clevel-mmt* exhibits a slight overhead over *Clevel* for positive search queries because *Clevel-mmt*'s load operations checks if the location value is locked and it should help concurrent pcas (§3.4.3). (2) *Clevel-mmt* exhibits a noticeable overhead over *Clevel* for delete queries because *Clevel-mmt*'s delete operations perform two plain CASes for detectability. (3) *Clevel-mmt* outperforms *Clevel* for insert queries, and *Clevel* does not scale well over 24 threads. The main reason for this is that the PMDK allocator used by *Clevel* does not perform well for allocation and thread counts above the core count. While the comparison is not apple-to-apple, we can at least deduce that *Clevel-mmt*'s detectability introduces only modest overhead for most combinations of thread counts, workloads, and key distributions.

3.7 Related and Future Work

Detectable Lock-Free DSs in PM Attiya et al. [2022] propose transforming lock-free DRAM-based DSs into

PM-based ones by persistently tracking an operation's progress and necessary completion information in its operation descriptor in PM. They assume each DS operation on the DS can be split into load-only gather and CAS-only update phases. However, this efficient approach is limited to specific operations that can be split in this manner and cannot handle complex operations with interleaved loads, CASes, or control constructs such as conditional branches and loops. Additionally, their approach performs a PM flush to reset an operation descriptor before reuse, while Memento directly overwrites mementos without resetting, utilizing timestamps.

Ben-David et al. [2019] checkpoint program points and local variables to record an operation's progress and result. However, their approach has two limitations. First, it makes unrealistic system assumptions to recover the execution context from the checkpointed values correctly, such as the persistence of the OS page table and maintaining the same virtual address space upon recovery. These assumptions are not satisfied by Linux, which is typically used for PM deployments. Moreover, their method requires the number of each stack frame's persisted local variables to be less than a machine word's bitwidth to atomically update the validity of the local variables, limiting its applicability to complex operations in file systems and DBMS. Second, their approach must checkpoint program points around CASes and after branches, causing noticeable performance overhead, especially in write-heavy workloads, as shown in §3.6.

Friedman et al. [2018] and Li and Golab [2021] present detectable MSQs in PM, but both have a bug on reclamation (§3.5.1) and perform slower than our MSQ due to an additional flush (§3.6).

Rusanovsky et al. [2021] and Fatourou et al. [2022] present hand-tuned persistent combining DSs based on a general combiner. However, their DSs only support a single invocation for each operation (§3.5): their DSs use a *fixed* per-thread PM storage to track the progress of a thread's operation, and in our experience of implementing *CombQ-mmt*, storing the results of multiple invocations requires a sizeable restructuring of the algorithms. Furthermore, their methods require additional DS logic, requiring deep understanding: e.g. the combining queue of Fatourou et al. [2022] has extra synchronization that prevents dequeuing of elements that are enqueued but not yet persisted. By contrast, our type system applies to general programs with control constructs (Fig. 3.2) and automatically guarantees the detectability of well-typed programs (Theorem 3.3.1).

Undetectable Lock-Free DSs in PM Friedman et al. [2018] present an undetectable lock-free MSQ in PM. Our detectable MSQ outperforms theirs because their dequeue operation allocates a PM block to store the return value (§3.6). Various hash tables [Nam et al. 2019; Zuo et al. 2018; Chen et al. 2020; Zuriel et al. 2019; Lu et al. 2020; Lee et al. 2019] and trees [Arulraj et al. 2018; Kim et al. 2021b] in PM have been proposed in the literature. In this dissertation, we convert the Clevel [Chen et al. 2020] hash table to a detectable one as a case study because it is lock-free. Converting the others to detectable DSs is an interesting direction for future work.

Transformation of DSs from DRAM to PM Izraelevitz et al. [2016b] present a universal construction of lock-free DSs in PM, but the constructed DSs are generally slow [Friedman et al. 2020, 2021]. Lee et al. [2019] propose a *RECIPE* to convert indexes from DRAM to PM and Kim et al. [2021b] propose the *Packed Asynchronous Concurrency* guideline to construct high-performance persistent DSs in PM, but their approaches are abstract, high-level, and not immediately applicable to DSs in general. By contrast, our rules of composition provide a more concrete guideline at the code level.

NVTraverse [Friedman et al. 2020] is a systematic transformation of persistent DSs, exploiting an observation that most operations comprise two phases: read-only traversal (which does not require flushes) and critical modification. Mirror [Friedman et al. 2021] is a more general and efficient transformation that replicates DSs in PM and DRAM, significantly improving read performance. FliT [Wei et al. 2022] is a persistent DS library based on a transformation utilizing dirty cache line tracking. However, none of these works support the transformation

of detectable DSs.

Detectability Friedman et al. [2018]; Li and Golab [2021]; Attiya et al. [2018, 2022] define detectability as *thread*'s ability to detect the DS operation's progress of a pre-crash execution and resume thereafter. We formalize this property as a deterministic replay of thread executions (Theorem 3.3.2) and instead define detectability as failure transparency of *machine*'s behaviours under crashes, and generally prove the detectability of well-typed programs (Theorem 3.3.1).

PM Platforms Memento applies not only to Intel Optane persistent memory [Intel 2024b] (with and without eADR [Intel 2021]) but also to other PM platforms, such as Samsung's CMM-H [Samsung 2024], because they all provide the following features that Memento relies on: direct access via mmap and fine-grained data transfer. Intel's PMDK [Intel 2024d] maps persistent memory to virtual memory via mmap to support direct memory access [Intel 2023], while Samsung's SMDK supports the CXL.mem interface [Samsung 2022] that serves the same purpose. Furthermore, both Intel Optane persistent memory and Samsung CMM-H's CXL.mem interface transfer data at the cache line granularity [Blankenship 2020].

Future Work (1) We will design larger objects (e.g. file systems and storage engines) in Memento (see §4.2.1). (2) In doing so, we will adapt existing hand-tuned detectable concurrent DSs and persistent transactional memory (PTM) systems [Memaripour et al. 2017; Krishnan et al. 2020] to Memento to compose them into larger objects. (3) We will formalize our type system and verify the detectability of well-typed programs in logics for PM [Raad et al. 2020; Vindum and Birkedal 2023]. (4) We will reason about the invariant-based optimizations to verify *MSQ-mmt-O2* by composing the type-based automatic verification of *MSQ-mmt-O1* and manual verification of the invariant-based optimization.

Chapter 4. Conclusion

4.1 Summary

As shown in Fig. 4.1, this dissertation presents principles of byte-addressable persistency, particularly in the context of PM. It focuses on two primary contributions: the development of hardware semantic models (§2) and a general programming model for detectably recoverable concurrent data structures (§3).

Hardware Semantic Models To provide a formal foundation for addressing the complexities of relaxed persistency, we introduced hardware semantic models, $\text{Px86}_{\text{view}}$ (§2.3) and $\text{PArmv8}_{\text{view}}$ (§2.6.2), for the Intel-x86 and Armv8 architectures, respectively. $\text{Px86}_{\text{view}}$ and $\text{PArmv8}_{\text{view}}$ were developed using view-based semantics, providing a unified operational style for describing persistency. Additionally, we developed axiomatic models, $\text{Px86}_{\text{axiom}}$ (§2.4.4) and $\text{PArmv8}_{\text{axiom}}$ (§2.6.3), and in doing so, identified and fixed bugs in the existing models. For Px86 [Raad et al. 2019b], we ensured that flush instructions behave synchronously, preventing potential post-crash inconsistencies when external operations are involved (§2.2.1). For PArmv8 [Raad et al. 2019a], we addressed the non-MCA behavior by enforcing an order between a flush and a subsequent write on the same location (§2.2.2). We formally proved the equivalence between the view-based and fixed axiomatic models for both architectures (§2.5 and §2.6.4), ensuring the soundness of our models. Furthermore, we developed a stateless model checker for persistency by adapting $\text{PArmv8}_{\text{view}}$ to RMEM [Armstrong et al. 2019] and used it to verify several representative examples (§2.7), demonstrating the practical applicability of our models.

A General Programming Model To ensure crash-consistent and efficient programming in PM, we introduced a general programming model for detectably recoverable concurrent data structures, Memento. Memento supports primitive operations like detectable checkpoint (§3.4.2) and CAS (§3.4.3), integrated with standard control constructs such as sequential composition, conditionals, and loops (§3.2). Memento is underpinned by a core language and type system specifically tailored for PM, and we verified the soundness of Memento's type system, demonstrating that well-typed programs are detectably recoverable (§3.3). Furthermore, we utilized Memento to transform volatile concurrent data structures into their detectable counterparts suitable for PM (§3.5), which include a lock-free linked-list, Treiber stack, Michael-Scott queue, combining queue, and Clevel hash table. Our evaluations demonstrated that Memento's primitive operations and data structures based on Memento can recover effectively from random thread and system crashes (§3.6.1), performing comparably to existing hand-tuned persistent data structures (§3.6.2). These results highlight the robustness and efficiency of our general programming model, providing a solid foundation for future development in PM systems.

4.2 Future Work

We believe that our contributions will serve as a stepping stone for future research in PM systems, providing a formal foundation for byte-addressable persistency. As shown in Fig. 4.1, we propose specific tasks for the future works based on this dissertation: the development of a lock-free file system as a large-scale application of Memento (§4.2.1), and the formal verification of primitive operations presented in Memento (§4.2.2).

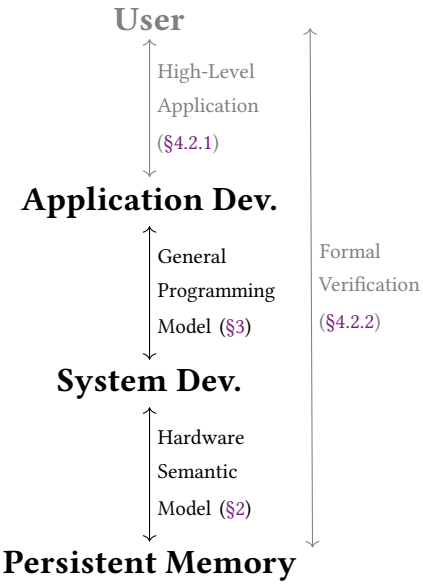


Figure 4.1: Our contributions (in black) and future work (in gray).

4.2.1 A Lock-Free File System as a High-Level Application

Our first future work is to develop the *first* lock-free file system on top of PM as a high-level application. This will serve as a proof of concept, showcasing the potential performance gains and crash safety that lock-free concurrency can offer in PM-based storage systems. Although several file systems in PM have already been presented [Xu and Swanson 2016; Kwon et al. 2017; Chen et al. 2021], to the best of our knowledge, none of them exploit the potential of lock-free concurrency. By developing a lock-free file system, we can demonstrate the potential performance of lock-free storage systems, which has not yet been fully realized.

Memento can be used to develop a lock-free file system, since it is expressive enough to support high-level applications by supporting control constructs for general programming, as well as the detectable CAS primitive operation for lock-free programming in PM. In addition to Memento's expressiveness, we can develop a PM-based crash-safe application in a correct-by-construction way, thanks to its verified type system.

To achieve this, we set the following tasks: (1) we design the high-level of file system based on Memento; and (2) devise a more robust safe memory reclamation strategy which is a key component of lock-free programs [Michael 2004] for the file system. By evaluating the performance of the file system in comparison with existing PM-based file systems, We expect this work will not only further showcase the versatility and effectiveness of the Memento framework in the context of high-level applications, but also impact the design of future PM-based lock-free storage systems.

4.2.2 Formal Verification of Memento Primitive Operations

To complete end-to-end verification from PM to high-level applications, we formlly define hardware semantic models (§2), then validate the soundness of type system for lock-free programs in PM (§3), and finally, thanks to Memento, build a high-level application in a correct-by-construction way (§4.2.1). Our second future work is to enhance the verification process of Memento for greater reliability.

Although we have demonstrated the soundness of Memento's type system, there remain challenges in enhancing the verification process for greater reliability. For example, the primitive operations of Memento, `chkpt` and `pcas`, have not yet been verified for correctness. For high-level verification of the type system, the

original proof assumes that the implementations of primitive operations refine the corresponding specifications. To make the proof more reliable, this assumption should be replaced with a formal proof, particularly when reasoning about shared memory operations based on underlying weak concurrency and persistency.

To address the issue, we plan to design a program logic for verifying the correctness of primitive operations, based on our formal semantic model, $\text{Px86}_{\text{view}}$. To this end, we set the following tasks: (1) extend the Iris framework [Iris 2024], a higher-order concurrent separation logic framework, to incorporate weak persistency; and building upon this foundation, (2) prove that the primitive operations refine the corresponding semantics described in §B.4. For all the proofs included in this verification, we intend to mechanize the entire proof using the Coq [Coq 2024] proof assistant.

Appendices

Chapter A. Appendices of §2

A.1 Full Model of $\text{Px86}_{\text{view}}$

A.1.1 Language

Fig. A.1 presents the language for Intel-x86 concurrency and persistency. The language is explained in §2.3.1.

A.1.2 States and Transitions

Fig. A.2 presents the states of $\text{Px86}_{\text{view}}$ and x86_{view}.

Fig. A.3 presents the transitions of $\text{Px86}_{\text{view}}$ and x86_{view}. We get this figure by putting Fig. 2.4, Fig. 2.5, and Fig. 2.6 in one figure and simplifying redundant rules.

A.2 Full Model of $\text{PArmv8}_{\text{view}}$

A.2.1 Language

Fig. A.4 presents the language for Armv8 concurrency and persistency. It is similar to that for Intel-x86 (Fig. A.1), the differences being:

- Loads and stores are annotated with *access ordering* ("rk" or "wk", resp.) and *exclusivity* ("xcl").
- Fences are more diverse: `isb` orders loads and succeeding dependent accesses; `dmb.`f orders accesses according to the ordering constraint f; and `dsb.`f additionally waits for the pending flush instructions to finish.
- Armv8 has only asynchronous flush `flushopt` (`dc.cvap`).

A.2.2 States and Transitions for $\text{Armv8}_{\text{view}}$ [Pulte et al. 2019]

We review $\text{Armv8}_{\text{view}}$ due to [Pulte et al. 2019]: a view-based model for Armv8 concurrency. We first explain the key differences from x86_{view} including the semantics of the above-mentioned features; we refer to [Pulte et al. 2019] for the full detail. In doing so we gradually introduce the components of a thread state presented in Fig. A.5. Then we present the full model.

Relaxed Access Ordering Unlike Intel-x86, Armv8 does not order accesses unless specified otherwise. To capture the relaxedness, we introduce the following view components to thread states: (1) $\mathsf{v}_{\mathsf{rOld}}$ that represents the maximum timestamp previously read by the thread; (2) $\mathsf{v}_{\mathsf{wOld}}$ that represents the maximum timestamp previously written by the thread; (3) $\mathsf{v}_{\mathsf{rNew}}$ that forbids the thread's future reads from accessing messages overwritten by itself; and (4) $\mathsf{v}_{\mathsf{wNew}}$ that forbids the thread's future writes from writing messages earlier than itself. Recall that $\mathsf{v}_{\mathsf{rNew}}$ in x86_{view} also represents the maximum timestamp previously read by the thread; its role is split to $\mathsf{v}_{\mathsf{rOld}}$ and $\mathsf{v}_{\mathsf{rNew}}$ in $\text{Armv8}_{\text{view}}$. Thus, e.g. a read does not automatically constrain the succeeding reads.

There are two ways to order the accesses: fence and access ordering. Fence joins the "old" views ($\mathsf{v}_{\mathsf{rOld}}$ and $\mathsf{v}_{\mathsf{wOld}}$) to the "new" views ($\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$), thereby forcing the messages read or written by the thread in the past, to be earlier than those read or written in the future. Specifically, `dmb.ld` joins $\mathsf{v}_{\mathsf{rOld}}$ to each of $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$; `dmb.st`, $\mathsf{v}_{\mathsf{wOld}}$ to $\mathsf{v}_{\mathsf{wNew}}$; and `dmb.st`, $\mathsf{v}_{\mathsf{rOld}} \sqcup \mathsf{v}_{\mathsf{wOld}}$ to each of $\mathsf{v}_{\mathsf{rNew}}$ and $\mathsf{v}_{\mathsf{wNew}}$.

$$
\begin{array}{rcll}
p & ::= & s_1 \mid\mid \ldots \mid\mid s_n & \textit{program} \\
s \in \mathrm{Stmt} & ::= & \texttt{skip} & \textit{statement} \\
& \mid & s_1; s_2 \mid \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \mid \texttt{while } (e)\ s & \textit{control} \\
& \mid & r := e & \textit{assign} \\
& \mid & r := \texttt{pload}(e) & \textit{load} \\
& \mid & \texttt{store } [e_1]\ e_2 & \textit{store} \\
& \mid & r := \texttt{rmw } rop\ [e] & \textit{update} \\
& \mid & \texttt{fence}_f & \textit{fence} \\
& \mid & \texttt{flush } e \mid \texttt{flushopt } e & \textit{flush} \\
rop \in \mathrm{Rmw} & ::= & \texttt{fetch-op } op\ e \mid \texttt{cas } e_1\ e_2 & \textit{rmw op.} \\
f \in \mathrm{F} & ::= & \texttt{sfence} \mid \texttt{mfence} & \textit{fence} \\
e \in \mathrm{Expr} & ::= & v \mid r \mid (e_1\ op\ e_2) & \textit{pure expr.} \\
op \in \mathrm{O} & ::= & + \mid - \mid \ldots & \textit{arith op.} \\
v \in \mathrm{Val} & = & \mathbb{Z} & \textit{value} \\
r \in \mathrm{VReg} & = & \mathbb{N} & \textit{register} \\
l \in \mathrm{PLoc} & = & \mathrm{Val} & \textit{location}
\end{array}
$$

Figure A.1: Intel-x86 concurrency and persistency language.

$$
\begin{array}{c}
\langle \vec{T}, M \rangle \in \mathrm{Machine} \triangleq (\mathrm{TId} \to \mathrm{Thread}) \times \mathrm{Memory} \\
tid \in \mathrm{TId} \triangleq \mathbb{N} \qquad T \in \mathrm{Thread} \triangleq \mathrm{Stmt} \times \mathrm{TState} \\
M \in \mathrm{Memory} \triangleq \mathrm{list\ Msg} \qquad w \in \mathrm{Msg} \triangleq \langle \mathsf{loc} : \mathrm{PLoc}; \mathsf{val} : \mathrm{Val}; \mathsf{tid} : \mathrm{TId} \rangle \\
\langle l := v \rangle_{tid} \triangleq \langle \mathsf{loc} = l; \mathsf{val} = v; \mathsf{tid} = tid \rangle \qquad t \in \mathrm{Time} \triangleq \mathbb{N} \qquad v \in \mathbb{V} \triangleq \mathrm{Time} \\
ts \in \mathrm{TState} \triangleq \left\langle \begin{array}{l} \sigma : \mathrm{VReg} \to \mathrm{Val}; \\ \mathsf{coh} : \mathrm{PLoc} \to \mathbb{V}; \quad \mathsf{v_{rNew}} : \mathbb{V}; \\ \mathsf{v_{pReady}} : \mathbb{V}; \quad \mathsf{v_{pAsync}}, \mathsf{v_{pCommit}} : \mathrm{PLoc} \to \mathbb{V}; \end{array} \right\rangle
\end{array}
$$

Figure A.2: States of $\mathrm{Px86}_{\mathrm{view}}$ (and that of $\mathrm{x86}_{\mathrm{view}}$ if the highlighted area is removed).

Access ordering works similarly by joining new views. When a load is marked as wacq or acq it is ordered with succeeding accesses, and when a store is marked as wrel and rel it is ordered with preceding accesses. Also, a rel store and a succeeding acq are also ordered. For this purpose, $\mathrm{Armv8}_{\mathrm{view}}$ introduces the $\mathsf{v_{Rel}}$ view that represents the maximum timestamp of previous rel stores.

Exclusive Instruction instead of RMW Armv8 supports exclusive *load-link* and *store-conditional* [Jensen et al. 1987] that guarantee, when successful, there are no intervening stores between the load and the store. The return value of an exclusive store ("r_{succ}") indicates if the instruction is successful. Exclusive instructions are more primitive than RMWs in that an RMW can be implemented on top of them.[1]

The semantics of exclusive loads and stores are similar to that for RMWs except that the information on the "linked" load is stored in a thread state's *exclusivity bank* ($ts.\mathsf{xclb}$). Also, Armv8 forbids forwarding of exclusive

[1]While Armv8.1 also supports RMW instructions, they are currently missing in $\mathrm{Armv8}_{\mathrm{view}}$ [Pulte et al. 2019]. We do not generalize $\mathrm{Armv8}_{\mathrm{view}}$ to support RMW instructions because it is orthogonal to our purpose.

$$c\ ?\ v_1 : v_2 \triangleq \text{if } c \text{ then } v_1 \text{ else } v_2 \qquad c\ ?\ v \triangleq c\ ?\ v : 0 \qquad v_1 \sqcup v_2 \triangleq \max(v_1, v_2)$$

(INIT)

$$\frac{p = s_1 \mid\mid \ldots \mid\mid s_n}{\text{init}(p, \langle \lambda tid.\ \langle s_{tid}, \langle \sigma = \lambda\_.\ 0; \mathsf{coh} = \lambda\_.\ @0; \mathsf{v_{rNew}} = @0; \mathsf{v_{pReady}} = @0; \mathsf{v_{pAsync}}, \mathsf{v_{pCommit}} = \lambda\_.\ @0; \rangle\rangle, []\rangle)}$$

(MACHINE)

$$\frac{\vec{T}[tid], M \rightarrow_{tid} T', M'}{\langle \vec{T}, M\rangle \rightarrow \langle \vec{T}[tid \mapsto T'], M'\rangle}$$

(SKIP)

$$\frac{}{(\texttt{skip}; s, ts), M \rightarrow_{tid} (s, ts), M}$$

(ASSIGN)

$$\frac{ts' = ts[\sigma[r] \mapsto [\![e]\!]_{ts.\sigma}]}{(r := e, ts), M \rightarrow_{tid} (\texttt{skip}, ts'), M}$$

(BRANCH)

$$\frac{[\![e]\!]_{ts.\sigma} = v}{(\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2, ts), M \rightarrow_{tid} (v \neq 0\ ?\ s_1 : s_2, ts), M}$$

(WHILE)

$$\frac{s' = \textbf{if } e \textbf{ then } (s; \texttt{while } (e)\ s) \textbf{ else } \texttt{skip}}{(\texttt{while } (e)\ s, ts), M \rightarrow_{tid} (s', ts'), M}$$

(SEQ)

$$\frac{(s_1, ts), M \rightarrow_{tid} (s_1', ts'), M'}{(s_1; s_2, ts), M \rightarrow_{tid} (s_1'; s_2, ts'), M'}$$

(NOT-OVERWRITTEN)

$$\frac{\forall t \in (v_2, v_1].\ M[t].\mathsf{loc} \neq l}{v_1 \sqsubseteq_{M,l} v_2}$$

(LOAD)

$$\frac{\begin{array}{l} l = [\![e]\!]_{ts.\sigma} \qquad M[t] = \langle l := v\rangle \\ ts.\mathsf{coh}[l] \sqsubseteq t \qquad ts.\mathsf{v_{rNew}} \sqsubseteq_{M,l} t \\ v = t \neq ts.\mathsf{coh}[l]\ ?\ t \\ ts' = ts \begin{bmatrix} \sigma[r] \mapsto v, \\ \mathsf{coh}[l] \mapsto t, \\ \mathsf{v_{rNew}} \mapsto_\sqcup v, \\ \mathsf{v_{pReady}} \mapsto_\sqcup v \end{bmatrix} \end{array}}{(r := \texttt{pload}(e), ts), M \rightarrow_{tid} (\texttt{skip}, ts'), M}$$

(STORE)

$$\frac{\begin{array}{l} l = [\![e_1]\!]_{ts.\sigma} \qquad v = [\![e_2]\!]_{ts.\sigma} \\ t = |M| + 1 \qquad M' = M +\!\!+ [\langle l := v\rangle_{tid} @ t] \\ ts' = ts \left[\mathsf{coh}[l] \mapsto t\right] \end{array}}{(\texttt{store } [e_1]\ e_2, ts), M \rightarrow_{tid} (\texttt{skip}, ts'), M'}$$

(RMW)

$$\frac{\begin{array}{l} [\![rop]\!]_{ts.\sigma}(v_1, v_2) \\ l = [\![e]\!]_{ts.\sigma} \qquad M[t_1] = \langle l := v_1\rangle \\ t_2 = |M| + 1 \qquad t_2 - 1 \sqsubseteq_{M,l} t_1 \\ M' = M +\!\!+ [\langle l := v_2\rangle_{tid} @ t_2] \\ ts' = ts \begin{bmatrix} \sigma[r] \mapsto v_1, \\ \mathsf{coh}[l] \mapsto t_2, \\ \mathsf{v_{rNew}} \mapsto_\sqcup t_2, \\ \mathsf{v_{pReady}} \mapsto_\sqcup t_2, \\ \mathsf{v_{pCommit}} \mapsto_\sqcup ts.\mathsf{v_{pAsync}} \end{bmatrix} \end{array}}{(r := \texttt{rmw } rop\ [e], ts), M \rightarrow_{tid} (\texttt{skip}, ts'), M'}$$

(RMW-FAIL)

$$\frac{\begin{array}{l} [\![rop]\!]_{ts.\sigma}(v, \bot) \\ l = [\![e]\!]_{ts.\sigma} \qquad M[t] = \langle l := v\rangle \\ ts.\mathsf{coh}[l] \sqsubseteq t \qquad ts.\mathsf{v_{rNew}} \sqsubseteq_{M,l} t \\ v = t \neq ts.\mathsf{coh}[l]\ ?\ t \\ ts' = ts \begin{bmatrix} \sigma[r] \mapsto v, \\ \mathsf{coh}[l] \mapsto t, \\ \mathsf{v_{rNew}} \mapsto_\sqcup v, \\ \mathsf{v_{pReady}} \mapsto_\sqcup v \end{bmatrix} \end{array}}{(r := \texttt{rmw } rop\ [e], ts), M \rightarrow_{tid} (\texttt{skip}, ts'), M}$$

(MFENCE)

$$\frac{ts' = ts \begin{bmatrix} \mathsf{v_{rNew}} \mapsto_\sqcup \sqcup_l ts.\mathsf{coh}[l], \\ \mathsf{v_{pReady}} \mapsto_\sqcup \sqcup_l ts.\mathsf{coh}[l], \\ \mathsf{v_{pCommit}} \mapsto_\sqcup ts.\mathsf{v_{pAsync}} \end{bmatrix}}{(\texttt{mfence}, ts), M \rightarrow_{tid} (\texttt{skip}, ts'), M}$$

(SFENCE)

$$\frac{ts' = ts \begin{bmatrix} \mathsf{v_{pReady}} \mapsto_\sqcup \sqcup_l ts.\mathsf{coh}[l], \\ \mathsf{v_{pCommit}} \mapsto_\sqcup ts.\mathsf{v_{pAsync}} \end{bmatrix}}{(\texttt{sfence}, ts), M \rightarrow_{tid} (\texttt{skip}, ts'), M}$$

(FLUSH)

$$\frac{\begin{array}{l} l = [\![e]\!]_{ts.\sigma} \qquad v = \sqcup_{l'} ts.\mathsf{coh}[l'] \\ ts' = ts \begin{bmatrix} \mathsf{v_{pAsync}} \mapsto_\sqcup \lambda l'.\ cl(l, l')\ ?\ v, \\ \mathsf{v_{pCommit}} \mapsto_\sqcup \lambda l'.\ cl(l, l')\ ?\ v \end{bmatrix} \end{array}}{(\texttt{flush } e, ts), M \rightarrow_{tid} (\texttt{skip}, ts'), M}$$

(FLUSHOPT)

$$\frac{\begin{array}{l} l = [\![e]\!]_{ts.\sigma} \qquad v = \sqcup_{l'} cl(l, l')\ ?\ ts.\mathsf{coh}[l'] \\ ts' = ts \left[\mathsf{v_{pAsync}} \mapsto_\sqcup \lambda l'.\ cl(l, l')\ ?\ (v \sqcup ts.\mathsf{v_{pReady}})\right] \end{array}}{(\texttt{flushopt } e, ts), M \rightarrow_{tid} (\texttt{skip}, ts'), M}$$

(CRASH)

$$\frac{\forall l.\ \exists t.\ M[t] = \langle l := SM[l]\rangle\ \wedge\ \forall (\_, ts) \in \vec{T}.\ ts.\mathsf{v_{pCommit}}[l] \sqsubseteq_{M,l} t}{\langle \vec{T}, M\rangle \rightarrow_{\mathsf{crash}} SM}$$

Figure A.3: Transitions of $\text{Px86}_{\text{view}}$ (and those of x86_{view} if the highlighted area is removed).

$$
\begin{array}{rcll}
 & & \ldots \text{ (based on the language for Intel-x86 in Fig. A.1)} & \\
s \in \mathrm{Stmt} & ::= & \cdots & \textit{statement} \\
 & | & r \coloneqq \mathtt{load}_{xcl,rk}\ [e] & \textit{load} \\
 & | & r_{\mathrm{succ}} \coloneqq \mathtt{store}_{xcl,wk}\ [e_1]\ e_2 & \textit{store} \\
 & | & \mathtt{isb} \mid \mathtt{dmb}.f \mid \mathtt{dsb}.f & \textit{fence} \\
 & | & \mathtt{flushopt}\ e & \textit{flush} \\
f \in \mathrm{F} & ::= & \mathtt{ld} \mid \mathtt{st} \mid \mathtt{sy} & \textit{order}
\end{array}
$$

$$
\begin{array}{rcll}
xcl \in \mathbb{B} & ::= & \mathtt{false} \mid \mathtt{true} & \textit{exclusivity} \\
rk \in \mathrm{RK} & ::= & \mathsf{pln} \mid \mathsf{wacq} \mid \mathsf{acq} & \textit{read kind} \\
wk \in \mathrm{WK} & ::= & \mathsf{pln} \mid \mathsf{wrel} \mid \mathsf{rel} & \textit{write kind}
\end{array}
$$

Figure A.4: Armv8 concurrency and persistency language.

$$
\begin{array}{l}
\ldots \text{ (based on the states for Intel-x86 in Fig. A.2)} \\
ts \in \mathrm{TState} \triangleq \left\langle \begin{array}{l}
\sigma : \mathrm{VReg} \to \mathrm{Val} \times \mathbb{V}\,; \quad \mathsf{prom} : \mathrm{set\ Time}\,; \\
\mathsf{coh} : \mathrm{PLoc} \to \mathbb{V}\,; \\
\mathsf{v_{rOld}}, \mathsf{v_{wOld}}, \mathsf{v_{rNew}}, \mathsf{v_{wNew}}, \mathsf{v_{Rel}}, \mathsf{v_{CAP}} : \mathbb{V}\,; \\
\mathsf{fwdb} : \mathrm{PLoc} \to \langle \mathsf{time} : \mathrm{Time}; \mathsf{view} : \mathbb{V}; \mathsf{mem} : \mathbb{B} \rangle; \\
\mathsf{xclb} : \mathrm{option}\ \langle \mathsf{time} : \mathrm{Time}; \mathsf{view} : \mathbb{V} \rangle \\
\mathsf{v_{pReady}} : \mathbb{V}\,; \quad \mathsf{v_{pAsync}}, \mathsf{v_{pCommit}} : \mathrm{PLoc} \to \mathbb{V}\,;
\end{array} \right\rangle
\end{array}
$$

Figure A.5: States of PArmv8$_{\mathrm{view}}$ (and that of Armv8$_{\mathrm{view}}$ [Pulte et al. 2019] if the highlighted area is removed).

stores to wacq loads. To model this, we introduce *forward bank* (ts.fwdb) that describes forwardable stores.

Dependency Unlike Intel-x86, Armv8 tracks control, address, and data dependency to order a load and succeeding dependent instructions. To this end, a thread state ($ts.\sigma$) not only contains a value for each register but also a view that represents the dependency carried by the register. When a register is used as branch condition or address, its view is joined to the thread's *control-address-program-order* view ($ts.\mathsf{v_{CAP}}$) that constrains future stores. It is also joined to $\mathsf{v_{rNew}}$ when an `isb` fence is executed, thus starting to constrain future loads as well.

Full Model Fig. A.6 and Fig. A.7 present the transitions of Armv8$_{\mathrm{view}}$.

A.2.3 States and Transitions for PArmv8$_{\mathrm{view}}$

Fig. A.6 and Fig. A.7 also present the transitions of PArmv8$_{\mathrm{view}}$. The differences between PArmv8$_{\mathrm{view}}$ and Armv8$_{\mathrm{view}}$, which are highlighted in the figures, are largely the same with those between Px86$_{\mathrm{view}}$ and x86$_{\mathrm{view}}$.

A.3 Proof of the Optionality of (PF-MIN) in Px86$_{\mathrm{axiom}}$ and PArmv8$_{\mathrm{axiom}}$

We prove that the **(PF-MIN)** axiom is optional in Px86$_{\mathrm{axiom}}$. More specifically, a behavior is allowed under Px86$_{\mathrm{axiom}}$ with **(PF-MIN)** iff it is allowed under Px86$_{\mathrm{axiom}}$ without **(PF-MIN)**.

Proof of Theorem 2.4.2. ($\Rightarrow$) Obvious from the fact that the axioms are weakened.
($\Leftarrow$) Suppose a behavior satisfies the old axioms.

$$v@v \triangleq \langle v, v\rangle : \mathrm{Val} \times \mathbb{V}$$

(INIT)

$$\frac{p = s_1 \mid\mid \ldots \mid\mid s_n}{\mathrm{init}(p, \langle \lambda tid.\ \langle s_{tid}, \left\langle \begin{array}{l} \sigma = \lambda\_.\ 0@0; \quad \mathsf{coh} = \lambda\_.\ @0; \quad \mathsf{v_{rOld}}, \mathsf{v_{wOld}}, \mathsf{v_{rNew}}, \mathsf{v_{wNew}}, \mathsf{v_{CAP}}, \mathsf{v_{Rel}} = @0; \\ \mathsf{fwdb} = \lambda\_.\ \langle \mathsf{time} = @0; \mathsf{view} = @0; \mathsf{mem} = \mathtt{false}\rangle; \quad \mathsf{xclb} = \mathit{none} \\ \mathsf{v_{pReady}} = @0; \quad \mathsf{v_{pAsync}}, \mathsf{v_{pCommit}} = \lambda\_.\ @0; \end{array} \right\rangle, []\rangle)}$$

(MACHINE)

$$\frac{\vec{T}[tid], M \to_{tid} T', M' \qquad \langle T', M'\rangle \text{ certified}}{\langle \vec{T}, M\rangle \to \langle \vec{T}[tid \mapsto T'], M'\rangle}$$

$$\langle T, M\rangle \text{ certified} \triangleq \exists T', M'.\ \langle T, M\rangle \xrightarrow{\mathrm{seq}}{}^{*}_{tid} \langle T', M'\rangle \wedge T'.\mathsf{prom} = \{\}$$

(SEQ-EXEC)

$$\frac{T, M \to_{tid} T'}{\langle T, M\rangle \xrightarrow{\mathrm{seq}}_{tid} \langle T', M\rangle}$$

(SEQ-WRITE)

$$\frac{\langle T, M\rangle \xrightarrow{tid}_{a} t\ \langle T', M'\rangle \qquad T', M' \xrightarrow{tid}_{a} t\ T''}{\langle T, M\rangle \xrightarrow{\mathrm{seq}}_{tid} \langle T'', M'\rangle}$$

(EXECUTE)

$$\frac{T, M \to_{tid} T'}{\langle T, M\rangle \to_{tid} \langle T', M\rangle}$$

(PROMISE)

$$\frac{w.\mathsf{tid} = tid \quad t = |M| + 1 \quad M' = M +\!\!+ [w] \quad ts' = ts[\mathsf{prom} \mapsto ts.\mathsf{prom} \cup \{t\}]}{\langle (s, ts), M\rangle \xrightarrow{tid}_{a} t\ \langle (s, ts'), M'\rangle}$$

(CRASH)

$$\frac{\forall (\_, ts) \in \vec{T}.\ ts.\mathsf{prom} = \{\} \qquad \forall l.\ \exists t.\ M[t] = \langle l := SM[l]\rangle \ \wedge\ \forall (\_, ts) \in \vec{T}.\ ts.\mathsf{v_{pCommit}}[l] \sqsubseteq_{M,l} t}{\langle \vec{T}, M\rangle \to_{\mathsf{crash}} SM}$$

Figure A.6: Machine and thread steps of PArmv8$_{\mathrm{view}}$ (and those of Armv8$_{\mathrm{view}}$ [Pulte et al. 2019] if the highlighted area is removed).

We may assume:

$$\mathrm{ob}_{w,f} = \mathrm{obs} \cup \mathrm{dob} \cup \mathrm{bob} \cup \mathrm{fob} \cup (\mathrm{pf} \setminus \{(w, f)\}) \cup \mathrm{fp}$$
$$\forall (w, f) \in \mathrm{pf}, (w, f) \in \mathrm{ob}^{+}_{w,f} \qquad (\textsc{pf-min}_0)$$

Suppose otherwise, and let $(w, f) \in \mathrm{pf} \setminus \mathrm{ob}^{+}_{w,f}$. Let w' be the store event just before w w.r.t. co, and $\mathrm{pf}' = \mathrm{pf} \cup \{(w', f)\} \setminus \{(w, f)\}$. Then pf' also satisfies the old axioms. In particular, if there is a cycle of $\mathrm{ob}' = \mathrm{ob}_{w,f} \cup \{(w', f), (f, w)\} \subseteq \mathrm{ob} \cup \{(f, w)\}$, then it should contain the edge (f, w) and thus $(w, f) \in (\mathrm{ob}_{w,f} \cup \{(w', f)\})^{+}$. If such a trace contains (w', f), then we have $(w, w') \in \mathrm{ob}^{+}_{w,f}$, contradicting the acyclicity of ob. Thus we have $(w, f) \in \mathrm{ob}^{+}_{w,f}$, contradicting the assumption. Now by repeatedly moving pf "backwards" w.r.t. co, we get a behavior that satisfies ($\textsc{pf-min}_0$).

Let $\mathrm{ob}_0 = \mathrm{obs} \cup \mathrm{dob} \cup \mathrm{bob} \cup \mathrm{fob} \cup \mathrm{fp}$ and we prove $\mathrm{pf} \subseteq {\mathrm{ob}_0}^{+}$. Suppose otherwise. We linearize events w.r.t. ob, and to each event e, give the linearization index $L(e)$, and let $(w, f) \in \mathrm{pf} \setminus {\mathrm{ob}_0}^{+}$ be such a pair of events with the minimum number of $L(f) - L(w)$. Since $(w, f) \in \mathrm{ob}^{+}_{w,f}$, such a trace contains an edge in $\mathrm{pf} \setminus \{(w, f)\}$. By the minimality of $L(f) - L(w)$, such an edge is also contained in ${\mathrm{ob}_0}^{+}$ and thus $(w, f) \in {\mathrm{ob}_0}^{+}$, contradicting the assumption. □

The same proof applies also to PArmv8$_{\mathrm{axiom}}$.

$[\![(-)_1]\!]_{(-)_2} : \mathit{Expr} \to (\mathrm{VReg} \to \mathrm{Val} \times \mathbb{V}) \to \mathrm{Val} \times \mathbb{V}$

$$[\![v]\!]_m \triangleq v@0 \qquad [\![r]\!]_m \triangleq m(r) \qquad [\![e_1 \ \mathsf{op}\ e_2]\!]_m \triangleq (v_1\ [\![\mathsf{op}]\!]\ v_2)@(v_1 \sqcup v_2) \text{ with } [\![e_1]\!]_m = v_1@v_1, [\![e_2]\!]_m = v_2@v_2$$

$$\mathrm{read}(M,l,t) : \mathrm{option\ Val} \triangleq \text{ if } t = 0 \text{ then } \mathsf{v}_{\mathrm{init}} \text{ else if } M[t].\mathsf{loc} = l \text{ then } M[t].\mathsf{val} \text{ else } \mathit{none}$$

$$\text{read-view}(rk, f, t) \triangleq \text{ if } (f.\mathsf{time} = t \wedge (f.\mathsf{mem} \Rightarrow rk \sqsubseteq \mathsf{pln})) \text{ then } f.\mathsf{view} \text{ else } t$$

$$\mathrm{atomic}(M, l, tid, t_{\mathrm{r}}, t_{\mathrm{w}}) \triangleq M(t_{\mathrm{r}}).\mathsf{loc} = l \implies \forall t'.\ (t_{\mathrm{r}} < t' < t_{\mathrm{w}} \wedge M[t'].\mathsf{loc} = l) \implies M[t'].\mathsf{tid} = tid$$

(EXCLUSIVE-FAILURE)

$$\frac{xcl = \mathtt{true} \qquad ts' = ts[\sigma[r] \mapsto v, \mathsf{xclb} \mapsto \mathit{none}]}{(r_{\mathrm{succ}} := \mathtt{store}_{xcl,wk}[e_1]\, e_2, ts), M \to_{tid} (\mathtt{skip}, ts')}$$

(LOAD)

$$\frac{\begin{array}{l} l@v_{\mathrm{addr}} = [\![e]\!]_{ts.\sigma} \\ \mathrm{read}(M,l,t) = v \\ v_{\mathrm{pre}} = v_{\mathrm{addr}} \sqcup ts.\mathsf{v_{rNew}} \sqcup (rk \sqsupseteq \mathsf{acq}\ ?\ ts.\mathsf{v_{Rel}}) \\ \forall t'.\ t < t' \leq (v_{\mathrm{pre}} \sqcup ts.\mathsf{coh}(l)) \implies M[t'].\mathsf{loc} \neq l \\ v_{\mathrm{post}} = v_{\mathrm{pre}} \sqcup \text{read-view}(rk, ts.\mathsf{fwdb}(l), t) \\ ts' = ts \left[\begin{array}{l} \sigma(r) \mapsto v@v_{\mathrm{post}}, \\ \mathsf{coh}(l) \mapsto\sqcup\ v_{\mathrm{post}}, \\ \mathsf{v_{rOld}} \mapsto\sqcup\ v_{\mathrm{post}}, \\ \mathsf{v_{rNew}} \mapsto\sqcup\ rk \sqsupseteq \mathsf{wacq}\ ?\ v_{\mathrm{post}}, \\ \mathsf{v_{wNew}} \mapsto\sqcup\ rk \sqsupseteq \mathsf{wacq}\ ?\ v_{\mathrm{post}}, \\ \mathsf{v_{CAP}} \mapsto\sqcup\ v_{\mathrm{addr}}, \\ \mathsf{xclb} \mapsto xcl\ ?\ \langle \mathsf{time} = t; \mathsf{view} = v_{\mathrm{post}} \rangle : ts.\mathsf{xclb} \end{array}\right] \end{array}}{(r := \mathtt{load}_{xcl,rk}\ [e], ts), M \to_{tid} (\mathtt{skip}, ts')}$$

(FULFIL)

$$\frac{\begin{array}{l} [\![e_1]\!]_{ts.\sigma} = l@v_{\mathrm{addr}} \qquad [\![e_2]\!]_{ts.\sigma} = v@v_{\mathrm{data}} \\ xcl \implies ts.\mathsf{xclb} \neq \mathit{none} \wedge \mathrm{atomic}(M,l,tid,ts.\mathsf{xclb.time},t) \\ t \in ts.\mathsf{prom} \qquad M[t] = \langle l := v \rangle_{tid} \\ v_{\mathrm{pre}} = v_{\mathrm{addr}} \sqcup v_{\mathrm{data}} \sqcup ts.\mathsf{v_{wNew}} \sqcup ts.\mathsf{v_{CAP}} \sqcup \\ \qquad (wk \sqsupseteq \mathsf{wrel}\ ?\ (ts.\mathsf{v_{rOld}} \sqcup ts.\mathsf{v_{wOld}})) \\ (v_{\mathrm{pre}} \sqcup ts.\mathsf{coh}(l)) < t \\ v_{\mathrm{post}} = t \qquad v_{\mathrm{succ}} = \bot \\ ts' = ts \left[\begin{array}{l} \mathsf{prom} \mapsto ts.\mathsf{prom} \setminus \{t\}, \\ \sigma(r_{\mathrm{succ}}) \mapsto xcl\ ?\ \mathsf{v}_{\mathrm{succ}}@v_{\mathrm{succ}} : ts.\sigma(r_{\mathrm{succ}}), \\ \mathsf{coh}(l) \mapsto\sqcup\ v_{\mathrm{post}}, \\ \mathsf{v_{wOld}} \mapsto\sqcup\ v_{\mathrm{post}}, \\ \mathsf{v_{CAP}} \mapsto\sqcup\ v_{\mathrm{addr}}, \\ \mathsf{v_{Rel}} \mapsto\sqcup\ wk \sqsupseteq \mathsf{rel}\ ?\ v_{\mathrm{post}}, \\ \mathsf{fwdb}(l) \mapsto \langle \mathsf{time} = t; \mathsf{view} = v_{\mathrm{addr}} \sqcup v_{\mathrm{data}}; \mathsf{mem} = xcl \rangle \\ \mathsf{xclb} \mapsto xcl\ ?\ \mathit{none} : ts.\mathsf{xclb} \end{array}\right] \end{array}}{(r_{\mathrm{succ}} := \mathtt{store}_{xcl,wk}\ [e_1]\ e_2, ts), M \xrightarrow{tid}_a t\ (\mathtt{skip}, ts')}$$

(DMB)

$$\frac{ts' = ts \left[\begin{array}{l} \mathsf{v_{rNew}} \mapsto\sqcup\ (f = \mathsf{sy} \vee f = \mathtt{ld}\ ?\ ts.\mathsf{v_{rOld}}) \sqcup (f = \mathsf{sy}\ ?\ ts.\mathsf{v_{wOld}}), \\ \mathsf{v_{wNew}} \mapsto\sqcup\ (f = \mathsf{sy} \vee f = \mathtt{ld}\ ?\ ts.\mathsf{v_{rOld}}) \sqcup (f = \mathsf{sy} \vee f = \mathtt{st}\ ?\ ts.\mathsf{v_{wOld}}), \\ \mathsf{v_{pReady}} \mapsto\sqcup\ f = \mathsf{sy}\ ?\ (ts.\mathsf{v_{rOld}} \sqcup ts.\mathsf{v_{wOld}}) \end{array}\right]}{(\mathtt{dmb}.f, ts), M \to_{tid} (\mathtt{skip}, ts')}$$

(DSB)

$$\frac{ts' = ts \left[\begin{array}{l} \mathsf{v_{rNew}} \mapsto\sqcup\ (f = \mathsf{sy} \vee f = \mathtt{ld}\ ?\ ts.\mathsf{v_{rOld}}) \sqcup (f = \mathsf{sy}\ ?\ ts.\mathsf{v_{wOld}}), \\ \mathsf{v_{wNew}} \mapsto\sqcup\ (f = \mathsf{sy} \vee f = \mathtt{ld}\ ?\ ts.\mathsf{v_{rOld}}) \sqcup (f = \mathsf{sy} \vee f = \mathtt{st}\ ?\ ts.\mathsf{v_{wOld}}), \\ \mathsf{v_{pReady}} \mapsto\sqcup\ f = \mathsf{sy}\ ?\ (ts.\mathsf{v_{rOld}} \sqcup ts.\mathsf{v_{wOld}}), \\ \mathsf{v_{pCommit}} \mapsto\sqcup\ ts.\mathsf{v_{pAsync}} \end{array}\right]}{(\mathtt{dsb}.f, ts), M \to_{tid} (\mathtt{skip}, ts')}$$

(ISB)

$$\frac{ts' = ts\ [\mathsf{v_{rNew}} \mapsto\sqcup\ ts.\mathsf{v_{CAP}}]}{(\mathtt{isb}, ts), M \to_{tid} (\mathtt{skip}, ts')}$$

(ASSIGN)

$$\frac{ts' = ts[\sigma(r) \mapsto [\![e]\!]_{ts.\sigma}]}{(r := e, ts), M \to_{tid} (\mathtt{skip}, ts')}$$

(BRANCH)

$$\frac{[\![e]\!]_{ts.\sigma} = v@v \qquad ts' = ts[\mathsf{v_{CAP}} \mapsto\sqcup\ v]}{(\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2, ts), M \to_{tid} (v \neq 0\ ?\ s_1 : s_2, ts')}$$

(SKIP)

$$\frac{}{(\mathtt{skip}; s, ts), M \to_{tid} (s, ts)}$$

(SEQ)

$$\frac{(s_1, ts), M \to_{tid} (s_1', ts')}{(s_1; s_2, ts), M \to_{tid} (s_1'; s_2, ts')}$$

(WHILE)

$$\frac{s' = \mathbf{if}\ e\ \mathbf{then}\ (s; \mathtt{while}\ (e)\ s)\ \mathbf{else}\ \mathtt{skip}}{(\mathtt{while}\ (e)\ s, ts), M \to_{tid} (s', ts')}$$

(FLUSHOPT)

$$\frac{\begin{array}{l} l@\_ = [\![e]\!]_{ts.\sigma} \qquad v = \sqcup_{l'}\ cl(l,l')\ ?\ ts.\mathsf{coh}[l'] \\ ts' = ts \left[\mathsf{v_{pAsync}} \mapsto\sqcup\ \lambda l'.\ cl(l,l')\ ?\ (v \sqcup ts.\mathsf{v_{pReady}})\right] \end{array}}{(\mathtt{flushopt}\ e, ts), M \to_{tid} (\mathtt{skip}, ts')}$$

Figure A.7: Thread-local steps of PArmv8$_{\mathrm{view}}$ (and that of Armv8$_{\mathrm{view}}$ [Pulte et al. 2019] if the highlighted area is removed).

$$\text{tso is transitive and irreflexive} \tag{TSO-STRICT}$$

$$\text{tso is total on } E \setminus R \tag{TSO-TOTAL}$$

$$\text{co} \subseteq \text{tso} \tag{TSO-CO}$$

$$\text{rf} \subseteq \text{tso} \cup \text{po} \tag{TSO-RF1}$$

$$\forall x \in \text{Loc}, \forall (w, r) \in \text{rf}_x, \forall w' \in W_x \cup U_x,$$
$$(w', r) \in \text{tso} \cup \text{po} \implies (w, w') \notin \text{tso} \tag{TSO-RF2}$$

$$([W \cup U \cup R]; \text{po}; [W \cup U \cup R]) \setminus (W \times R) \subseteq \text{tso} \tag{TSO-PO}$$

$$([E]; \text{po}; [MF]) \cup ([MF]; \text{po}; [E]) \subseteq \text{tso} \tag{TSO-MF}$$

Figure A.8: The x86_{man} model [Raad et al. 2019b, Definition 4].

A.4 Proof of the Equivalence of SPx86 and $\text{Px86}_{\text{axiom}}$

A.4.1 SPx86

Fig. A.8 presents x86_{man}, the authoritative axiomatic model of Intel-x86 concurrency due to [Owens et al. 2009; Sewell et al. 2010] that is reviewed by Intel engineers. Fig. A.9 presents the (S)Px86 model [Raad et al. 2019b]. Px86 and SPx86 are the same except that SPx86 additionally enforces (SYNC-FL). While not explicit in [Raad et al. 2019b], we require the (PERSIST) axiom that governs the contents of PM after crash.

A.4.2 Equivalence of x86_{man} and $\text{x86}_{\text{axiom}}$

Proof of Theorem 2.4.4. We prove the equivalence by continuously transforming x86_{man} into an equivalent one until reaching $\text{x86}_{\text{axiom}}$.

(1) We replace (TSO-RF1) with a simpler condition:

$$\begin{aligned} & \text{rf} \subseteq \text{tso} \cup \text{po} && \text{(TSO-RF1)} \\ \Leftrightarrow\ & \text{rf} \setminus \text{po} \subseteq \text{tso} \\ \Leftrightarrow\ & (\text{rfe} \subseteq \text{tso}) \ \wedge \ (\text{rfi} \setminus \text{po} \subseteq \text{tso}) \\ \Leftrightarrow\ & (\text{rfe} \subseteq \text{tso}) \ \wedge \ (\text{rf}; \text{po}^? \text{ irreflexive}) \ , \end{aligned}$$

where the forward direction of the last equivalence holds because otherwise, if $(w, r) \in \text{rf}$ and $(r, w) \in \text{po}^?$, then $(w, r) \in \text{rfi} \setminus \text{po} \subseteq \text{tso}$ and $(r, w) \in [U \cup R]; \text{po}^?; [W \cup U] \subseteq \text{tso}^?$ by (TSO-PO), contradicting the irreflexivity of tso.

(2) We replace (TSO-RF2) with a simpler condition:

$$\begin{aligned} & \forall x \in \text{Loc}, \forall (w, r) \in \text{rf}_x, \forall w' \in W_x \cup U_x, (w', r) \in \text{tso} \cup \text{po} \implies (w, w') \notin \text{tso} && \text{(TSO-RF2)} \\ \Leftrightarrow\ & \text{rf}^{-1}; (\text{tso} \cap \text{Loc}); [W \cup U]; (\text{tso} \cup \text{po}) \text{ irreflexive} \\ \Leftrightarrow\ & \text{rf}^{-1}; \text{co}; (\text{tso} \cup \text{po}) \text{ irreflexive} \\ \Leftrightarrow\ & (\text{fr}; \text{tso} \text{ irreflexive}) \ \wedge \ (\text{fr}; \text{po} \text{ irreflexive}) \ , \end{aligned}$$

$$
\begin{array}{rll}
 & \text{(axioms of x86}_{\text{man}}\text{ (Fig. A.8))} & \\
\mathsf{tso} & \supseteq ([E];\mathrm{po};[SF]) \cup ([SF];\mathrm{po};[E \setminus R]) & \text{(X86-TSO-FL)} \\
 & \cup\ [W \cup U \cup FL];\mathrm{po};[FL] & \\
 & \cup\ [FL];\mathrm{po};[W \cup U \cup FL] & \\
 & \cup\ [FL];(\mathrm{po} \cap \mathrm{CL});[FO] & \\
 & \cup\ [FO];(\mathrm{po} \cap \mathrm{CL});[FL] & \\
 & \cup\ ([U];\mathrm{po};[FO]) \cup ([FO];\mathrm{po};[U]) & \\
 & \cup\ [W];(\mathrm{po} \cap \mathrm{CL});[FO] & \\
 & \cup\ [R];\mathrm{po};[FL \cup FO] & \\
\mathsf{nvo} & \supseteq \mathsf{tso} \cap \mathrm{Loc} & \text{(NVO-ORDER)} \\
 & \cup\ [W \cup U];(\mathsf{tso} \cap \mathrm{CL});[FL \cup FO] & \\
 & \cup\ [FL \cup FO];\mathsf{tso};[W \cup U \cup FL \cup FO] & \\
 & \mathsf{nvo} \text{ is total on } W \cup U \cup FL \cup FO & \text{(NVO-TOTAL)} \\
P & \supseteq dom(\mathsf{nvo};[P]) & \text{(NVO-P)} \\
P & \supseteq dom([FL] \cup ([FO];\mathrm{po};[MF \cup SF \cup U])) & \boxed{\text{(SYNC-FL)}} \\
\forall l.\ \exists w.\ SM(l) = \mathtt{wval}(w)\ \wedge\ (P \times \{w\}) \cap \mathrm{Loc} \subseteq \mathsf{co}^? & & \boxed{\text{(PERSIST)}}
\end{array}
$$

Figure A.9: (S)Px86 [Raad et al. 2019b].

where the second equivalence holds from:

$$
\begin{aligned}
& [W \cup U];(\mathsf{tso} \cap \mathrm{Loc});[W \cup U] \\
&= \mathsf{tso} \cap (\mathsf{co} \cup \mathsf{co}^{-1} \cup [W \cup U]) \\
&= \mathsf{co} \cup (\mathsf{tso} \cap (\mathsf{co}^{-1} \cup [W \cup U])) \\
&= \mathsf{co}\ ,
\end{aligned}
$$

since $(\mathsf{co}^{-1} \cup [W \cup U]) \subseteq \mathsf{tso}^{-1?}$ and tso is acyclic.

(3) We simplify (fr; tso irreflexive) to fre $\subseteq$ tso.

($\Leftarrow$) Since fr; po is irreflexive, we have fri $\subseteq [U \cup R];\mathrm{po}^?;[W \cup U] \subseteq \mathsf{tso}^?$, and thus fr $\subseteq \mathsf{tso}^?$ and fr; tso is irreflexive from the acyclicity of tso.

($\Rightarrow$) Without loss of generality, assume tso is minimal such a relation that satisfies all the axioms. Let $\mathsf{tso}' = (\mathsf{tso} \cup \mathsf{fre})^+$. Then tso' satisfies all the other axioms except for (fr; tso irreflexive), and in addition, fre $\subseteq \mathsf{tso}'$.

It remains to prove the acyclicity of tso'. Assume otherwise and let c be a cycle of tso $\cup$ fre. Then c should be tso^+, fre^+, or $(\mathsf{fre}^+;\mathsf{tso}^+)^+$. But the first case is impossible from the acyclicity of tso; the second case is impossible because then c is co^+ and thus tso^+; and the third case is also impossible as follows. Since $\mathsf{fre}^+;\mathsf{tso}^+ \subseteq \mathsf{fre};\mathsf{co}?;\mathsf{tso}^+ \subseteq \mathsf{fre};\mathsf{tso}$, c is also $(\mathsf{fre};\mathsf{tso})^+$. Since fr $\subseteq [U \cup R] \times [W \cup U]$, c is also $(\mathsf{fre};\mathsf{tso};[U \cup R])^+$. By the minimality of tso, we have $\mathsf{tso};[U \cup R] \subseteq \mathsf{tso}^?;(\mathsf{co} \cup \mathsf{rfe} \cup [MF];\mathrm{po});[U \cup R]$ and c is also $(\mathsf{tso}^?;(\mathsf{co} \cup \mathsf{rfe} \cup [MF];\mathrm{po});\mathsf{fre})^+$. Without loss of generality, assume c is minimal such a cycle. If there is an edge of $\mathsf{tso}^?;(\mathsf{co} \cup \mathsf{rfe});\mathsf{fre}$, then it is also $\mathsf{tso}^?;\mathsf{co}^+ \subseteq \mathsf{tso}$ so that it can be merged into the next edge, contradicting the minimality of c. Thus c is also $([MF];\mathrm{po};\mathsf{fre};\mathsf{tso}^?)^+$. If c has two distinct vertices,

say v_1, v_2, then either $(v_1, v_2) \in \mathrm{tso}$ or $(v_2, v_1) \in \mathrm{tso}$, and either the trace from v_1 to v_2 or that from v_2 to v_1 can be merged into the previous edge, contradicting the minimality of c. Thus we have a vertex v such that $(v, v) \in [MF]; \mathrm{po}; \mathrm{fre}; \mathrm{tso}^?$, contradicting the irreflexivity of $\mathrm{fr}; \mathrm{tso}$.

(4) We replace (TSO-MF) with $[E]; \mathrm{po}; [MF]; \mathrm{po}; [E] \subseteq \mathrm{tso}$ and (TSO-TOTAL) with (tso is total on $E \setminus (R \cup MF)$).

($\Rightarrow$) Obvious from the fact that the condition is weakened.

($\Leftarrow$) Let tso be minimal such a relation that satisfies the new axioms. Due to the minimality, tso does not relate MF. Let $\mathrm{tso}' = (\mathrm{tso} \cup ([E]; \mathrm{po}; [MF]) \cup ([MF]; \mathrm{po}; [E]))^+$. We prove tso' is acyclic. Suppose otherwise and let c be a cycle of tso'. Since MF is not related in tso, c is also a cycle of $(\mathrm{tso} \cup ([E]; \mathrm{po}; [MF]; \mathrm{po}; [E]))$. Since $[E]; \mathrm{po}; [MF]; \mathrm{po}; [E] \subseteq \mathrm{tso}$, c is also a cycle of tso, contradicting the assumption. Now let tso'' be an acyclic superset of tso' in which MF is linearized. Then tso'' satisfies (TSO-MF), (TSO-TOTAL), and all the other old axioms.

(5) We remove tso. Specifically, we replace the following axioms with the acyclicity of ob:

- tso is transitive and irreflexive;
- tso is total on $E \setminus (R \cup MF)$; and
- $\mathrm{ob}^+ \subseteq \mathrm{tso}$.

We reached $\mathrm{x86}_{\mathrm{axiom}}$, concluding the proof. □

A.4.3 Equivalence of SPx86 and $\mathrm{Px86}_{\mathrm{axiom}}$

PROOF OF THEOREM 2.4.3. We prove the equivalence by continuously transforming SPx86 into an equivalent one until reaching $\mathrm{Px86}_{\mathrm{axiom}}$.

(1) We name each component of tso and remove fences from the relations. Specifically, (i) we replace $\mathrm{x86}_{\mathrm{man}}$ with $\mathrm{x86}_{\mathrm{axiom}}$; and (ii) replace (x86-TSO-FL) with:

$$\begin{aligned}
\mathrm{fob} \;=\; & [E]; \mathrm{po}; [MF]; \mathrm{po}; [E] \\
\cup\; & [E]; \mathrm{po}; [SF]; \mathrm{po}; [E \setminus R] \\
\cup\; & ([W \cup U \cup FL]; \mathrm{po}; [FL]) \cup ([FL]; \mathrm{po}; [W \cup U \cup FL]) \\
\cup\; & ([FL]; (\mathrm{po} \cap \mathrm{CL}); [FO]) \cup ([FO]; (\mathrm{po} \cap \mathrm{CL}); [FL]) \\
\cup\; & ([U]; \mathrm{po}; [FO]) \cup ([FO]; \mathrm{po}; [U]) \\
\cup\; & [W]; (\mathrm{po} \cap \mathrm{CL}); [FO] \\
\cup\; & [R]; \mathrm{po}; [FL \cup FO] \\
\mathrm{ob} \;=\; & \mathrm{obs} \cup \mathrm{dob} \cup \mathrm{bob} \cup \mathrm{fob} \\
& \mathrm{tso} \supseteq \mathrm{ob}^+ .
\end{aligned}$$

The old and new axioms are equivalent similarly with the proof of Theorem 2.4.4.

(2) We introduce to ob new relations, pf ("persist-from") and fp ("from-persist"), where pf is meant to relate a flush instruction to the co-maximal store that is flushed at that instruction; and fp is defined as $\mathrm{pf}^{-1}; \mathrm{co}$. Like rf for load instructions, pf relates a flush instruction to at most one store instruction for each location, the key difference being that pf relates a flush instruction to a store for each location in the same cache line. Specifically, we change the axioms as follows:

$$\mathrm{ob} \;=\; \mathrm{obs} \cup \mathrm{dob} \cup \mathrm{bob} \cup \mathrm{fob} \cup \mathrm{pf} \cup \mathrm{fp}$$

($\Leftarrow$) Obvious from the fact that the axioms are strengthened.

($\Rightarrow$) Suppose a behavior satisfies the old axioms. Let $\mathrm{pf}_0 = ((W \cup U) \times (FL \cup FO)) \cap \mathrm{CL} \cap \mathrm{tso}$, $\mathrm{pf} = \mathrm{pf}_0 \setminus (\mathrm{co}; \mathrm{pf}_0)$, and $\mathrm{fp} = \mathrm{pf}^{-1}; \mathrm{co}$. Then $\mathrm{pf} \cup \mathrm{fp} \subseteq \mathrm{tso}$ by construction. Thus the behavior satisfies the new axioms as well.

(3) We simplify (NVO-ORDER) into:

$$\begin{aligned} \mathrm{nvo} &\supseteq \mathrm{co} && \text{(NVO-CO)} \\ &\cup \mathrm{pf} \cup \mathrm{fp} && \text{(NVO-FOBS)} \\ &\cup [FL \cup FO]; \mathrm{tso}; [FL \cup FO] && \text{(NVO-FL)} \end{aligned}$$

(4) From now on, let $PFO = dom([FO]; \mathrm{po}; [MF \cup SF \cup U])$.

(5) We replace (NVO-ORDER), (NVO-TOTAL), (NVO-P), and (NVO-PERS) with the following axiom:

$$\begin{aligned} \mathrm{per} &= \mathrm{pf}; \mathrm{tso}^?; [FL \cup PFO] && \text{(PER)} \\ P &\supseteq dom(\mathrm{per}) \\ P &\supseteq dom(\mathrm{co}; [P]) \\ P &\subseteq W \cup U \end{aligned}$$

($\Rightarrow$) Suppose a behavior satisfies the old axioms. Let $P' = P \cap (W \cup U)$. Thanks to (NVO-PERS), (NVO-CO), and (NVO-FL), we have $dom(\mathrm{per}) \subseteq P'$. Thanks to (NVO-CO), we have $dom(\mathrm{co}; [P']) \subseteq P'$. Since only writes (in $W \cup U$) affect the contents of PM, the same behavior is also allowed in the new axioms.

($\Leftarrow$) Suppose a behavior satisfies the new axioms. nvo_1 be the RHS of (NVO-ORDER), $\mathrm{nvo}_2 = (dom(\mathrm{fp}^?; P) \cup FL \cup PFO) \times ((W \cup U) \setminus P)$, and $\mathrm{nvo}_c = \mathrm{nvo}_1 \cup \mathrm{nvo}_2$. We prove nvo_c is acyclic. If there is such a cycle, then it should be a cycle of nvo_1, nvo_2, or $\mathrm{nvo}_1^+; \mathrm{nvo}_2^+$. But the first case is impossible because $\mathrm{nvo}_1 \subseteq \mathrm{tso}$ and tso is acyclic; the second case is impossible because its domain and codomain are disjoint; and the third case is also impossible as follows. If there is a cycle of $\mathrm{nvo}_1^+; \mathrm{nvo}_2^+$, then there is an edge $(a, b) \in \mathrm{nvo}_1^+ \cap \mathrm{nvo}_2^{-1} = [(W \cup U) \setminus P]; \mathrm{nvo}_1^+; [dom(\mathrm{fp}^?; P) \cup FL \cup PFO]$. Without loss of generality, we assume (a, b) is minimal such an arc of nvo_1^+.

(i) Suppose (a, b) has no intermediate vertices, i.e. $(a, b) \in \mathrm{nvo}_1$. By case analysis on $[(W \cup U) \setminus P]; \mathrm{nvo}_1; [dom(\mathrm{fp}^?; P) \cup FL \cup PFO]$, we can derive contradiction.

(ii) Suppose (a, b) has an intermediate vertex, say c, of nvo_1^+. Then $c \notin dom(\mathrm{nvo}_2) \cup codom(\mathrm{nvo}_2)$ by the minimality, and thus $c \in FO \setminus PFO \setminus dom(\mathrm{fp}; P)$ and:

$$\begin{aligned} (a, b) \in\ & [(W \cup U) \setminus P]; \mathrm{pf}; [FO \setminus PFO \setminus dom(\mathrm{fp}; P)]; \mathrm{tso}^?; [FO \setminus PFO \setminus dom(\mathrm{fp}; P)]; \\ & ((\mathrm{fp}; [P]) \cup (\mathrm{tso}; [FL \cup PFO])) \\ =\ & [(W \cup U) \setminus P]; \mathrm{pf}; [FO \setminus PFO \setminus dom(\mathrm{fp}; P)]; \mathrm{tso}; [FL \cup PFO] \end{aligned}$$

Then we have $a \in dom(\mathrm{per}) \subseteq P$, contradicting the assumption.

Now let nvo be an acyclic superset of nvo_c in which $W \cup U \cup FL \cup FO$ is linearized, and $P' = dom(\mathrm{nvo}^?; [P \cup FL \cup PFO])$. Then nvo and P' satisfy the original axioms: (NVO-PERS) by the definition of P'; (NVO-CO), (NVO-FOBS), and (NVO-FL) by $\mathrm{nvo} \supseteq \mathrm{nvo}_1$; (NVO-TOTAL) by linearization; and (NVO-P) by the definition of P'.

It remains to prove $P' \cap (W \cup U) = P$.

($\supseteq$) By the definition of P'.

($\subseteq$) Suppose $w \in P' \cap (W \cup U)$. Then there exists e such that $(w, e) \in \mathrm{nvo}^?; [P \cup FL \cup PFO]$. If $w \notin P$, then $(e, w) \in \mathrm{nvo}_2$ and thus $(w, w) \in \mathrm{nvo}^+$, contradicting the acyclicity of nvo. Thus $w \in P$.

(6) We replace (PER), $P \supseteq dom(\mathrm{per})$, $P \supseteq dom(\mathrm{co}; [P])$, and $P \subseteq W \cup U$ with the following axiom:

$$\begin{aligned} \mathrm{per} &= \mathrm{pf}; \mathrm{tso}^?; [FL \cup PFO] && \text{(PER)} \\ P &= dom(\mathrm{per}) \end{aligned}$$

($\Rightarrow$) Obvious from the fact that the axioms are weakened.

($\Leftarrow$) Let $P' = dom(\mathrm{co}^?; P)$. Then P' and SM satisfies all the old axioms.

(7) We replace (PER) with the following axiom:

$$\mathrm{per} = \mathrm{pf}; ([FL] \cup ([FO]; \mathrm{ob}^*; [FL \cup PFO])) \qquad \text{(PER)}$$

($\Rightarrow$) Obvious from the fact that the axioms are weakened.

($\Leftarrow$) Let $PFO' = dom([FO]; \mathrm{ob}^*; [FL \cup PFO])$ and $Y = (FL \cup PFO') \times (FO \setminus PFO')$. We prove $\mathrm{ob} \cup Y$ is acyclic. If there is such a cycle, then it should be a cycle of ob, Y, or $\mathrm{ob}^+; Y$. But the first case is impossible because $\mathrm{ob} \subseteq \mathrm{tso}$ and tso is acyclic; the second case is impossible because its domain and codomain are disjoint; and the third case is also impossible as follows. If there is a cycle of $\mathrm{ob}^+; Y$, then there is an edge $(a, b) \in \mathrm{ob}^+ \cap Y^{-1} = [FO \setminus PFO']; \mathrm{ob}^+; [FL \cup PFO']$. Then $a \in PFO'$, contradicting the assumption.

Now let tso' be a linearization of $\mathrm{ob} \cup Y$. Then if $(a, b) \in [FL \cup FO]; \mathrm{tso}'; [FL \cup PFO]$, then we have $a \in FL \cup PFO'$ because otherwise $(b, a) \in Y \subseteq \mathrm{tso}'$, contradicting the acyclicity of tso'. Thus we have:

$$\begin{aligned} \mathrm{per}' &= \mathrm{pf}; \mathrm{tso}'^?; [FL \cup PFO] \\ &\subseteq \mathrm{pf}; [FL \cup PFO'] \\ &= \mathrm{per} \end{aligned}$$

As a result, tso', per', P', and SM satisfy the old axioms.

(8) We replace the irreflexivity of tso with the acyclicity of ob, and remove tso which is no longer used.

(9) We replace (PER) with the following axiom:

$$\mathrm{per} = \mathrm{pf}; [FL \cup PFO] \qquad \text{(PER)}$$

($\Rightarrow$) Obvious from the fact that the axioms are weakened.

($\Leftarrow$) Let $\mathrm{co}_{imm} = \mathrm{co} \setminus (\mathrm{co}; \mathrm{co})$, which represents the "immediate" coherence order.

We may assume:

$$\forall w, w', f.\ (w, w') \in \mathrm{co}_{imm},\ (w, f) \in \mathrm{pf},\ f \in FO \setminus PFO \implies (f, w') \in (\mathrm{obs} \cup \mathrm{dob} \cup \mathrm{bob} \cup \mathrm{fob} \cup \mathrm{pf} \cup (\mathrm{fp} \setminus \{f\} \times W))^+ .$$

Intuitively, it means $f \in FO \setminus PFO$ persists the co-latest posible store event. Suppose w, w', f do not satisfy the above statement. Let $\mathrm{pf}' = \mathrm{pf} \setminus \{(w, f)\} \cup \{(w', f)\}$. Then pf' satisfies the new axioms. In particular, if there is a cycle of ob', then it should contain the edge $(w', f) \in \mathrm{pf}'$, $(f, w') \in \mathrm{ob}'^+$, and thus (f, w') satisfies the statement. Furthermore, $P' = P$ because $f \notin FL \cup PFO$. Now by repeatedly moving pf "forwards", we get an execution in which the above statement is satisfied.

Let $FFO = dom([FO]; (\text{po} \cap \text{CL}); [FL])$ and $Y = (FL \cup PFO) \times (FO \setminus PFO \setminus FFO)$. We prove $\text{ob} \cup (FL \cup PFO) \times (FO \setminus PFO \setminus FFO)$ is acyclic. If there is such a cycle, then it should be a cycle of ob, $(FL \cup PFO) \times (FO \setminus PFO \setminus FFO)$, or $\text{ob}^+; (FL \cup PFO) \times (FO \setminus PFO \setminus FFO)$. But the first case is impossible because ob is acyclic; the second case is impossible because its domain and codomain are disjoint; and the third case is also impossible as follows. If there is such a cycle, then there is an edge $(a, b) \in [FO \setminus PFO \setminus FFO]; \text{ob}^+; [FL \cup PFO]$. By the assumption, we have $(a, b) \in [FO \setminus PFO \setminus FFO]; (\text{obs} \cup \text{dob} \cup \text{bob} \cup \text{fob} \cup \text{pf}); \text{co}^?; \text{ob}^*; [FL \cup PFO]$. By case analysis on $[FO \setminus PFO \setminus FFO]; (\text{obs} \cup \text{dob} \cup \text{bob} \cup \text{fob} \cup \text{pf})$, we can derive contradiction.

If $(a, b) \in [FO]; \text{ob}^*; [FL \cup PFO]$, then $a \in PFO \cup FFO$ because otherwise $(b, a) \in Y$, contradicting the acyclicity of $\text{ob} \cup Y$. Also, if $(a, b) \in \text{pf}; [FO]; (\text{po} \cap \text{CL}); [FL]$, then $(a, b) \in \text{co}^?; \text{pf}$ because otherwise $(b, a) \in \text{fp} \subseteq \text{ob}$, contradicting the acyclicity of ob. Thus we have:

$$\begin{aligned}
dom(\text{per}') &= dom(\text{pf}; ([FL] \cup ([FO]; \text{ob}^*; [FL \cup PFO]))) \\
&\subseteq dom(\text{pf}; [FL \cup PFO \cup FFO]) \\
&= dom(\text{pf}; [FL \cup PFO]) \cup dom(\text{pf}; [FFO]) \\
&= dom(\text{per}) \cup dom(\text{pf}; [FO]; (\text{po} \cap \text{CL}); [FL]) \\
&\subseteq dom(\text{per}) \cup dom(\text{co}^?; \text{pf}; [FL]) \\
&\subseteq dom(\text{co}^?; \text{per})
\end{aligned}$$

As a result, tso', per', P', and SM satisfy the old axioms.

(10) We additionally require the following axiom:

$$\begin{aligned}
\text{ob}_0 &= \text{obs} \cup \text{dob} \cup \text{bob} \cup \text{fob} \cup \text{fp} \\
\text{pf} &\subseteq \text{ob}_0{}^+ \qquad \text{(PF-MIN)}
\end{aligned}$$

The proof is the same with that of Theorem 2.4.2 in §A.3.

(11) We refactor fob as follows:

$$\begin{aligned}
\text{fob} &= [FL]; \text{po}; ([W \cup U \cup FL] \cup ([MF \cup SF]; \text{po}; [FO])) \\
&\cup [FO]; \text{po}; ([U] \cup ([MF \cup SF]; \text{po}; [W \cup FL \cup FO])) \\
&\cup [FL]; (\text{po} \cap \text{CL}); [FO] \\
&\cup [FO]; (\text{po} \cap \text{CL}); [FL] \\
&\cup [W \cup U \cup R]; \text{po}; [FL] \\
&\cup ([U \cup R] \cup ([W]; \text{po}; [MF \cup SF])); \text{po}; [FO] \\
&\cup [W]; (\text{po}; [FL])^?; (\text{po} \cap \text{CL}); [FO]
\end{aligned}$$

Notice that $[W]; \text{po}; [FL]; (\text{po} \cap \text{CL}); [FO]$ is added to fob, but it was already in fob^+ and thus the addition does not change semantics.

(12) We remove those edges starting at $FL \cup FO$:

$$\begin{aligned}
\text{fob} &= [W \cup U \cup R]; \text{po}; [FL] \\
&\cup ([U \cup R] \cup ([W]; \text{po}; [MF \cup SF])); \text{po}; [FO] \\
&\cup [W]; (\text{po}; [FL])^?; (\text{po} \cap \text{CL}); [FO]
\end{aligned}$$

Let $Y = [FL \cup FO]; \text{fob}$. Then the old relation $\text{fob} = \text{fob}' \cup Y$.

$$
\begin{aligned}
\mathrm{obs} &= \mathrm{co} \cup \mathrm{rfe} \cup \mathrm{fr} \\
\mathrm{dob} &= (\mathrm{addr} \cup \mathrm{data}); \mathrm{rfi}^? \\
&\cup\ (\mathrm{ctrl} \cup (\mathrm{addr}; \mathrm{po})); ([W] \cup ([\texttt{isb}]; po; [R])) \\
\mathrm{aob} &= [\mathrm{range}(\mathrm{rmw})]; \mathrm{rfi}; [AQ \cup AQpc] \\
\mathrm{bob} &= [R]; \mathrm{po}; [\texttt{dmb.ld} \cup \texttt{dsb.ld}]; \mathrm{po}; [R \cup W] \\
&\cup\ [W]; \mathrm{po}; [\texttt{dmb.st} \cup \texttt{dsb.st}]; \mathrm{po}; [W] \\
&\cup\ [R \cup W]; \mathrm{po}; [\texttt{dmb.sy} \cup \texttt{dsb.sy}]; \mathrm{po}; [R \cup W] \\
&\cup\ [RL]; \mathrm{po}; [AQ] \\
&\cup\ [AQ \cup AQpc]; \mathrm{po}; [W \cup R] \\
&\cup\ [W \cup R]; \mathrm{po}; [RL \cup RLpc] \\
\mathrm{ob} &= \mathrm{obs} \cup \mathrm{dob} \cup \mathrm{aob} \cup \mathrm{bob} \\
&((\mathrm{po} \cap \mathrm{Loc}) \cup \mathrm{co} \cup \mathrm{rf} \cup \mathrm{fr})\ \text{acyclic} && \text{(INTERNAL)} \\
&\mathrm{ob}\ \text{acyclic} && \text{(EXTERNAL)} \\
&\mathrm{rmw} \cap (\mathrm{fre}; \mathrm{coe})\ \text{empty} && \text{(ATOMIC)}
\end{aligned}
$$

Figure A.10: The $\mathrm{Armv8}_{\mathrm{axiom}}$ model [Pulte et al. 2019, Appendix D].

We first prove $\mathrm{ob_0}'; Y \subseteq \mathrm{ob_0}'$ by case analysis. We also prove:

$$
\begin{aligned}
\mathrm{ob}^+ &\subseteq \mathrm{ob_0}^+ && \text{(by PF-MIN)} \\
&= (\mathrm{ob_0}' \cup Y)^+ \\
&= \mathrm{ob_0}'^+ \cup (Y^+; \mathrm{ob_0}'^*)
\end{aligned}
$$

Now we show equivalence.

($\Rightarrow$) Since $\mathrm{ob}' \subseteq \mathrm{ob}$, it is sufficient to prove, for (PF-MIN), $[W \cup U]; \mathrm{ob}^+; [FL \cup FO] \subseteq [W \cup U]; \mathrm{ob}'^+; [FL \cup FO]$. It follows from $[W \cup U]; \mathrm{ob}^+ \subseteq [W \cup U]; (\mathrm{ob_0}'^+ \cup (Y^+; \mathrm{ob_0}'^*)) = [W \cup U]; \mathrm{ob_0}'^+$.

($\Leftarrow$) It is sufficient to prove the acyclicity of ob, or the irreflexivity of ob^+. Since $\mathrm{ob}^+ \subseteq \mathrm{ob_0}'^+ \cup (Y^+; \mathrm{ob_0}'^*)$, it is sufficient to prove the irreflexivity of $\mathrm{ob_0}'^+$ and that of $Y^+; \mathrm{ob_0}'^*$. The former is immediate. To prove the latter, suppose otherwise. Then $\mathrm{ob_0}'^*; Y^+ = Y^+ \cup \mathrm{ob_0}'^+$ is also not irreflexive. But it contradicts the fact that both Y^+ and $\mathrm{ob_0}'^+$ are irreflexive.

We reached $\mathrm{Px86}_{\mathrm{axiom}}$, concluding the proof. □

A.5 Proof of the Equivalence of SPArmv8 and $\mathrm{PArmv8}_{\mathrm{axiom}}$

A.5.1 SPArmv8

Fig. A.10 presents the $\mathrm{Armv8}_{\mathrm{view}}$ model [Pulte et al. 2019], and Fig. A.11 presents the (S)PArmv8 model [Raad et al. 2019a]. PArmv8 and SPArmv8 are the same except that SPArmv8's ob additionally includes fp. While not explicit in [Raad et al. 2019a], we require the (PERSIST) axiom that governs the contents of PM after crash.

$$
\begin{array}{rll}
 & \text{(axioms of Armv8}_{\text{axiom}}\text{ (Fig. A.10))} & \\
\mathrm{fob} & = (\mathrm{po}^{?}; [\mathtt{dmb.sy} \cup \mathtt{dsb.sy}]; \mathrm{po}^{?}) \setminus id & \text{(ARM-OB-BAR)} \\
 & \cup\ [W \cup R]; (\mathrm{po} \cap \mathrm{CL}); [FO] & \text{(ARM-W-WB)} \\
 & \cup\ [FO]; (\mathrm{po} \cap \mathrm{CL}); [FO] & \text{(ARM-WB-WB)} \\
\mathrm{ob} & = \mathrm{obs} \cup \mathrm{dob} \cup \mathrm{aob} \cup \mathrm{bob} \cup \mathrm{fob} \boxed{\cup\ \mathrm{fp}} & \textit{(redefined)} \\
\mathrm{nvo} & \supseteq [FO]; \mathrm{ob}^{+}; [\mathtt{dsb.sy}]; \mathrm{ob}^{+}; [W \cup FO] & \text{(NVO-WB-D)} \\
 & \cup\ [W]; (\mathrm{ob}^{+} \cap \mathrm{CL}); [FO] & \text{(NVO-W-WB)} \\
 & \cup\ \mathrm{co} & \text{(NVO-CO)} \\
 & \mathrm{nvo} \text{ is total on } W \cup FO & \text{(NVO-TOTAL)} \\
P & \supseteq dom([FO]; \mathrm{ob}^{+}; [\mathtt{dsb.sy}]) & \text{(NVO-PERS)} \\
P & \supseteq dom(\mathrm{nvo}; [P]) & \text{(NVO-P)} \\
\forall l.\ \exists w.\ SM(l) = \mathtt{wval}(w)\ \wedge\ (P \times \{w\}) \cap \mathrm{Loc} \subseteq \mathrm{co}^{?} & & \boxed{\text{(PERSIST)}}
\end{array}
$$

Figure A.11: (S)PArmv8 [Raad et al. 2019a].

A.5.2 Equivalence of SPArmv8 and PArmv8$_{\text{axiom}}$

Proof of Theorem 2.6.1. We prove by continuously transforming SPArmv8 into an equivalent one until reaching PArmv8$_{\text{axiom}}$.

(1) We replace (NVO-WB-D) and (NVO-PERS) with:

$$
\begin{array}{rll}
\mathrm{nvo} & \supseteq [FO]; \mathrm{po}; [\mathtt{dsb.sy}]; \mathrm{ob}^{+}; [W \cup FO] & \text{(NVO-WB-D)} \\
P & \supseteq dom([FO]; \mathrm{po}; [\mathtt{dsb.sy}]) & \text{(NVO-PERS)}
\end{array}
$$

It is sufficient to prove that $[FO]; \mathrm{ob}^{+}; [\mathtt{dsb.sy}] = [FO]; \mathrm{po}; [\mathtt{dsb.sy}]$.

($\supseteq$) By $[FO]; \mathrm{po}; [\mathtt{dsb.sy}] \subseteq \mathrm{fob} \subseteq \mathrm{ob}$.

($\subseteq$) We have $[FO]; \mathrm{ob} = [FO]; \mathrm{fob} \subseteq ([FO]; \mathrm{po}; [\mathtt{dmb.sy} \cup \mathtt{dsb.sy}]; \mathrm{po}^{?}) \cup ([FO]; (\mathrm{po} \cap \mathrm{CL}); [FO])$. Then by induction, we have $[FO]; \mathrm{ob}^{+}; [\mathtt{dsb.sy}] \subseteq [FO]; \mathrm{po}; [\mathtt{dsb.sy}]$.

(2) We remove fences from the relations. Specifically, we replace (ARM-OB-BAR) and (NVO-WB-D) with:

$$
\begin{array}{rl}
\mathrm{fob} & = [FO]; \mathrm{po}; [\mathtt{dmb.sy} \cup \mathtt{dsb.sy}]; \mathrm{po}; [W \cup R \cup FO] \\
 & \cup\ [FO]; (\mathrm{po} \cap \mathrm{CL}); [FO] \\
 & \cup\ [W \cup R]; \mathrm{po}; [\mathtt{dmb.sy} \cup \mathtt{dsb.sy}]; \mathrm{po}; [FO] \\
 & \cup\ [W \cup R]; (\mathrm{po} \cap \mathrm{CL}); [FO] \\
\mathrm{nvo} & \supseteq [FO]; \mathrm{po}; [\mathtt{dsb.sy}]; \mathrm{po}; \mathrm{ob}^{*}; [W \cup FO]
\end{array}
$$

The old and new axioms are equivalent. The proof is similar to that of Theorem 2.4.4.

(3) We replace (NVO-WB-D), (NVO-W-WB), (NVO-CO), (NVO-TOTAL), (NVO-PERS), and (NVO-P) with the following

axiom:

$$\begin{aligned} \mathrm{per} &= [W]; (\mathrm{ob}^+ \cap \mathrm{CL}); [FO]; \mathrm{po}; [\mathtt{dsb.sy}] && \text{(PER)} \\ P &\supseteq dom(\mathrm{per}) \\ P &\supseteq dom(\mathrm{co}; [P]) \\ P &\subseteq W \end{aligned}$$

($\Rightarrow$) Suppose a behavior satisfies the old axioms. Let $P' = P \cap W$. Thanks to (NVO-PERS) and (NVO-W-WB), we have $dom(\mathrm{per}) \subseteq P'$. Thanks to (NVO-CO), we have $dom(\mathrm{co}; [P]) \subseteq P$. Since only writes (in W) affect the contents of PM, the same behavior is also allowed in the new axioms.

($\Leftarrow$) Suppose a behavior satisfies the new axioms. Let nvo_1 be the union of RHS'es of (NVO-WB-D), (NVO-W-WB), and (NVO-CO); $\mathrm{nvo}_2 = (P \cup dom([FO]; \mathrm{po}; [\mathtt{dsb.sy}])) \times (W \setminus P)$; and $\mathrm{nvo}_c = \mathrm{nvo}_1 \cup \mathrm{nvo}_2$. We prove nvo_c is acyclic. If there is such a cycle, then it should be a cycle of nvo_1, nvo_2, or $\mathrm{nvo}_1^+; \mathrm{nvo}_2^+$. But the first case is impossible because $\mathrm{nvo}_1 \subseteq \mathrm{ob}^+$ and ob is acyclic; the second case is impossible because its domain and codomain are disjoint; and the third case is also impossible as follows. If there is a cycle of $\mathrm{nvo}_1^+; \mathrm{nvo}_2^+$, then there is an edge $(a, b) \in \mathrm{nvo}_1^+ \cap \mathrm{nvo}_2^{-1} = [W \setminus P]; \mathrm{nvo}_1^+; [P \cup dom([FO]; \mathrm{po}; [\mathtt{dsb.sy}])]$. Without loss of generality, we assume (a, b) is minimal such an arc of nvo_1^+. (i) Suppose (a, b) has no intermediate vertices, i.e. $(a, b) \in \mathrm{nvo}_1$. By case analysis on $[W \setminus P]; \mathrm{nvo}_1; [P \cup dom([FO]; \mathrm{po}; [\mathtt{dsb.sy}])]$, we can derive contradiction. (ii) Suppose (a, b) has an intermediate vertex, say c, of nvo_1^+. Then $c \notin dom(\mathrm{nvo}_2) \cup codom(\mathrm{nvo}_2)$ by the minimality, and thus $c \in FO \setminus dom(\mathrm{po}; [\mathtt{dsb.sy}])$. As a result, $(a, b) \in [W]; (\mathrm{ob}^+ \cap \mathrm{CL}); ([FO]; \mathrm{po}; [\mathtt{dsb.sy}]; \mathrm{po}; \mathrm{ob}^*)^+$. Then we have $a \in dom(\mathrm{per}) \subseteq P$, contradicting the assumption.

Now let nvo be an acyclic superset of nvo_c in which $W \cup FO$ is linearized, and $P' = dom(\mathrm{nvo}^?; ([P] \cup ([FO]; \mathrm{po}; [\mathtt{dsb.sy}])))$. Then nvo and P' satisfy the original axioms: (NVO-PERS) by the definition of P'; (NVO-WB-D), (NVO-W-WB), and (NVO-CO) by $\mathrm{nvo} \supseteq \mathrm{nvo}_1$; (NVO-TOTAL) by linearization; and (NVO-P) by the definition of P'.

It remains to prove $P' \cap W = P$. ($\supseteq$) By the definition of P'. ($\subseteq$) Suppose $w \in P' \cap W$. Then there exists e such that $(w, e) \in \mathrm{nvo}^?; ([P] \cup ([FO]; \mathrm{po}; [\mathtt{dsb.sy}]))$. If $w \notin P$, then $(e, w) \in \mathrm{nvo}_2$ and thus $(w, w) \in \mathrm{nvo}^+$, contradicting the acyclicity of nvo. Thus $w \in P$.

(4) We replace (PER), $P \supseteq dom(\mathrm{per})$, $P \supseteq dom(\mathrm{co}; [P])$, and $P \subseteq W$ with the following axiom:

$$\begin{aligned} \mathrm{per} &= [W]; (\mathrm{ob}^+ \cap \mathrm{CL}); [FO]; \mathrm{po}; [\mathtt{dmb.sy}] && \text{(PER)} \\ P &= dom(\mathrm{per}) \end{aligned}$$

($\Rightarrow$) Obvious from the fact that the axioms are weakened.

($\Leftarrow$) Let $P' = dom(\mathrm{co}^?; P)$. Then P' and SM satisfies all the old axioms.

(5) We additionally require the following axiom:

$$\mathrm{pf} \subseteq \mathrm{ob}^+ \qquad \text{(PF-MIN)}$$

The proof is almost the same with that of Theorem 2.4.2 in §A.3.

(6) We replace (PER) with the following axiom:

$$\mathrm{per} = \mathrm{pf}; [FO]; \mathrm{po}; [\mathtt{dmb.sy}] \qquad \text{(PER)}$$

It is sufficient to prove that $[W]; (\mathrm{ob}^+ \cap \mathrm{CL}); [FO] = \mathrm{co}^?; \mathrm{pf}; [FO]$.

($\supseteq$) By (PF-MIN).

($\subseteq$) Suppose $(w, f) \in [W]; \mathrm{ob}^+ \cap \mathrm{CL}; [FO]$ but $(w, f) \notin \mathrm{co}^?; \mathrm{pf}; [FO]$. Then $(f, w) \in \mathrm{fp} \subseteq \mathrm{ob}$ by definition. Then $(w, w) \in \mathrm{ob}^+$, contradicting the acyclicity of ob.

(7) We remove those edges starting at FO:

$$\begin{aligned}\mathrm{fob} \;=\; & [W \cup R]; \mathrm{po}; [\texttt{dmb.sy} \cup \texttt{dsb.sy}]; \mathrm{po}; [FO] \\ \cup\; & [W \cup R]; (\mathrm{po} \cap \mathrm{CL}); [FO]\end{aligned}$$

Let $Y = [FO]; \mathrm{fob}$. Then the old relation $\mathrm{fob} = \mathrm{fob}' \cup Y$. We first prove $\mathrm{ob_0}'; Y \subseteq \mathrm{ob_0}'$ by case analysis. Then the rest of the proof of this step is the same with that of the last step in Theorem 2.4.3.

We reached $\text{PArmv8}_{\text{axiom}}$, concluding the proof. □

A.6 Verified Examples

We use our model checking tool (§2.7) to verify several representative persistent synchronization examples. All examples are verified within one second.

First, all examples presented in this dissertation (except for CommitE) are verified. Specifically, the following examples are verified without modification[2]:

- CommitWeak satisfies $I \triangleq (\mathsf{Data}{=}0 \vee \mathsf{Data}{=}42) \wedge (\mathsf{Commit}{=}0 \vee \mathsf{Commit}{=}1)$.
- Commit1 satisfies $I \triangleq \mathsf{Commit}{=}1 \Rightarrow \mathsf{Data}{=}42$.
- Commit2 satisfies $I \triangleq \mathsf{Commit}{=}1 \Rightarrow \mathsf{Data}{=}42$.
- FlushMCA satisfies $I \triangleq \neg(\mathsf{Z}{=}\mathsf{W}{=}1 \wedge \mathsf{X}{=}\mathsf{Y}{=}0)$.

The CommitOpt example satisfies its invariant only under x86, but not under Armv8. We adapted this example to Armv8 as follows and verified:

```
(a) Data1 := 42   ||  (d) if (Data2 != 0) {
(b) dmb.sy        ||  (e)   dmb.sy
(c) Data2 := 7    ||  (f)   flushopt Data1
                  ||  (g)   flushopt Data2
                  ||  (e)   dsb.sy
                  ||  (h)   Commit := 1 }
```
(CommitOptArm)

- CommitOptArm satisfies $I \triangleq \mathsf{Commit}{=}1 \Rightarrow \mathsf{Data1}{=}42 \wedge \mathsf{Data2}{=}7$.

The CommitE example involving I/O is not verified yet by the model checker because it currently does not support I/O instructions. We believe it is straightforward to support them as done in [Kang et al. 2017] and verify the example.

In addition, the following examples are verified:

```
(a) Data := 42                    (a) Data := 42  ||  (b) if (Data != 0) {
(b) flushopt Data (CommitWeakOpt)                 ||  (c)     flushopt Data  (Commit2Opt)
(c) Commit := 1                                   ||  (d)     Commit := 1 }
```

[2]Unless otherwise noted, loads and stores are plain (i.e. LDR and STR).

```
(a) X := 1   ||  (d) r1 := Y
(b) dmb.sy   ||  (e) dmb.sy
(c) Y := 1   ||  (f) flushopt X (FOB)
             ||  (g) dsb.sy
             ||  (h) Z := r1
```

```
(a) r1 := WeakCAS_acq(Lock,0,1)   ||  (j) r2 := WeakCAS_acq(Lock,0,1)
(b) if (r1 == 0) {                ||  (k) if (r2 == 0) {
(c)     X := 1                    ||  (l)     if (X == 1) {
(d)     Y := 1                    ||  (m)         Z := 1
(e)     flushopt X                ||  (n)         flushopt Z        (AtomicPersists)
(f)     dsb.sy                    ||  (o)         dsb.sy }
(g)     flushopt Y                ||  (p)     Lock :=_rel 0 }
(h)     dsb.sy                    ||
(i)     Lock :=_rel 0 }           ||
```

The AtomicPersists example, adopted from [Raad et al. 2020, Example 3] modulo architectural differences, models persistent transaction. Here, $\mathsf{Lock} :=_{\mathsf{rel}} 0$ is a release store (STLR), and $\mathsf{WeakCAS}_{\mathsf{acq}}(\mathtt{l}, \mathtt{v}_1, \mathtt{v}_2)$ tries to compare-and-swap Lock from $\mathtt{v}_1$ to $\mathtt{v}_2$ with the acquire ordering, returning 0 if successful or 1 otherwise. $\mathsf{WeakCAS}_{\mathsf{acq}}$ is implemented roughly as follows:

```
(a) r1 := load_{true,acq} [l] // LDAXR
(b) if (r1 != v1) {
(c)     return 1 }                          (WeakCAS)
(d) r2 := store_{true,pln} [l] v2 // STXR
(e) return r2
```

- CommitWeakOpt satisfies $I \triangleq (\mathsf{Data}{=}0 \vee \mathsf{Data}{=}42) \wedge (\mathsf{Commit}{=}0 \vee \mathsf{Commit}{=}1)$.
- Commit2Opt satisfies $I \triangleq (\mathsf{Data}{=}0 \vee \mathsf{Data}{=}42) \wedge (\mathsf{Commit}{=}0 \vee \mathsf{Commit}{=}1)$.
- FOB satisfies $I \triangleq \mathsf{Z}{=}1 \Rightarrow \mathsf{X}{=}1$.
- AtomicPersists satisfies $I \triangleq \mathsf{Z}{=}1 \Rightarrow \mathsf{X}{=}1 \wedge \mathsf{Y}{=}1$.

Chapter B. Appendices of §3

B.1 Detectably Recoverable Insertion and Deletion

While CAS is a general base operation for pointer-based DSs, we observe that the performance of DSs implemented with detectable CAS (§3.4.3) are sometimes worse than that of hand-tuned detectable DSs, e.g., *MSQ-mmt-O0* is slower than hand-tuned detectable MSQs (§3.6.2). The primary reason is that general detectable CASes like `pcas()` performs plain CASes to the same location twice and flushes the location between them, incurring an high contention on the location among multiple threads.

As an optimization, we design a more efficient atomic pointer location object supporting detectable *insert*—atomically replacing the Null pointer—and *delete*—atomically detaching a valid memory block from DSs—as primitive operations. To this end, we extend the core language to support these operations. The key idea, inspired by an optimization of Friedman et al. [2018]; Li and Golab [2021]'s hand-tuned detectable MSQs, is distributing the contention of multiple threads into multiple memory blocks, significantly relieving contention on any single location. Such an optimization, however, requires non-trivial synchronization among multiple memory blocks, thus limiting its application to only insert and delete operations. While less general than CAS, insert and delete operations still support a wide variety of DSs, e.g. we can implement Michael-Scott queue (MSQ) [Michael and Scott 1996] with insert and delete operations.

B.1.1 API

We introduce operations of the following API:

1: $\underline{v_1} \coloneqq \texttt{insert}(loc, \underline{new}, \underline{ds})$ ▷ v_1 is either Ok or (Err cur)
2: $\underline{v_2} \coloneqq \texttt{delete}(\underline{loc}, \underline{old}, \underline{new})$ ▷ v_2 is either Ok or (Err cur)
3: $v_3 \coloneqq \texttt{pload}_{\texttt{opt}}(loc)$ ▷ considered read-only

The `insert` function atomically updates loc from Null to new, and `delete` atomically updates loc from old to new under the assumption that old is then unlinked from its DS (hence the name). The two functions returns Ok if successful; otherwise, returns (Err cur) where cur is the current value of loc. Since the two functions utilize tagged pointers [Wikipedia 2022] to synchronize with each other, we also provide a non-memento function $\texttt{pload}_{\texttt{opt}}$ that reads a untagged value from a location loc designated for `insert` and `delete` operations.

The API of these operations are different from that of detectable checkpoint and CAS as follows:

- While `insert` and `delete` are memento functions, they do not receive a memento id because all necessary information is checkpointed in the DS ds's memory blocks.
- The `insert` operation's loc parameter does not need to be the same across crashes. Thus, we underline the other parameters (that should be the same across crashes) but not loc of `insert`. This can be used to reduce PM flushes by saving a checkpoint for the location. To exploit this fact, we extend the type system to distinguish *stable* variables (that is same across crashes) and *unstable* variables (that may not be the same across crashes), and introduce a rule for unstable variable definition (omitted in §B.5).

B.1.2 Components and Assumptions

We present the components for insertion and deletion and their assumptions on the DS.

Location A location is a 64-bit architecture word, which we split into four categories: 1-bit *persist* (or *dirty*) flag for the *link-and-persist* technique [Wang et al. 2018; David et al. 2018], 8 bits reserved for future purposes, 10-bit user tag, and 45-bit offset. We enforce the invariant for the persist bit that the pointer value is persisted whenever the bit is cleared. Such an invariant is cooperatively maintained by location's three operations: load, insertion, and deletion. The load operation ensures that the returned pointer value is always persisted in the location. We assume the ENCODE and DECODE functions convert a tuple of persist bit and offset with user tag into an 64-bit word and the other way around.

Memory Block We assume each memory block has a dedicated 64-bit architecture word, which we call `repl`, that describes the memory block that replaces itself as an atomic location's next pointer value. We split 64 bits into two categories: 9-bit thread id, 10-bit user tag, and 45-bit offset. Similarly to detectable CAS (§3.4.3), the thread id 0 is reserved for those blocks that are not replaced yet. We assume the ENCODER and DECODER functions convert a tuple of thread id and offset with user tag into an 64-bit word and the other way around.

Traversal As we will see, we checkpoint the insertion of a memory block to a DS in the inserted node's `repl` field, and for efficiency purposes, we flush such checkpoint to PM only when the block is deleted. Should the system crash, we detect whether a non-checkpointed memory block was inserted before the crash by checking if the block is still in the DS (see §B.1.3 for details). To this end, we require the ability to traverse all the memory blocks in a DS. For instance, an MSQ is traversable from its head by recursively chasing each node's next pointer.

Memento The memento of insertion and deletion operations consist of a single timestamp at which an operation was *failed* for the last time.

Replay Flag We assume the per-thread variable $REPLAY$ indicates whether the thread is replaying a pre-crash execution ($REPLAY = \textsc{True}$) or executing new operations ($REPLAY = \textsc{False}$). Formally, $REPLAY$ is true if and only if the thread's execution does not yet observe a memento with the timestamp larger than $ts.\mathsf{time}$ (e.g., **PCAS-SUCC**).

B.1.3 Normal Execution

Algorithm 7 presents the load, insertion, and deletion algorithms.

Load The $\mathtt{pload}_{\mathtt{opt}}$ operation, which we adopt from David et al. [2018]; Wang et al. [2018], ensures the returned pointer values are always persisted by ❶ performing an architecture-provided plain load and decodes the pointer value (L2-L3); ❷ if its persist bit is cleared—which implies the pointer value is persisted, then returning the pointer value (L4); otherwise, ❸ retrying for a while to read a pointer value without the persist bit being set (L5-L10); and if it fails, ❹ flushing loc itself (L11); ❺ trying to clear the persist bit of loc by performing a CAS, and if it fails, retrying from the beginning (L12-L14); and ❻ returning the pointer value with the persist bit being cleared (L15).

Insertion The `insert` operation receives a location (loc), a new pointer value (new), the enclosing DS (ds), and the memento id `mid`; and atomically updates loc from NULL to new in a persistent manner (L17). `insert` first checks if it is executed in the replay mode (see §B.1.4 for details on replay); otherwise, it ❶ atomically updates loc's pointer value from the NULL pointer to the given pointer with the persist bit being set (L30); if it fails, records such a fact to the memento and reports the failure (L36); ❷ flushes loc (L38); and ❸ tries to atomically clear loc's

Algorithm 7 Load, Insertion, Deletion, and Helping Load

```
1:  function pload_opt(loc)
2:      old := LOAD_PLN(loc)
3:      (p_old, o_old) := DECODE(old)
4:      if ¬p_old then return o_old
5:      t := rdtsc
6:      cur := LOAD_PLN(loc)
7:      (p_cur, o_cur) := DECODE(cur)
8:      if ¬p_cur then return o_cur
9:      if old ≠ cur then old := cur; goto 3
10:     if rdtsc < t + PATIENCE then goto 6
11:     flushopt loc
12:     old' := ENCODE(False, o_old)
13:     r := CAS_PLN(loc, old, old')
14:     if r is (Err cur) then old := cur; goto 3
15:     return old'
16: end function

17: function insert(loc, new, ds, mid)
18:     if REPLAY then
19:         b1 := Contains(ds, new)
20:         b2 := (LOAD_PLN(new.repl) ≠ Null)
21:         if b1 ∨ b2 then return Ok
22:         t_mmt := LOAD_PLN(mmts[mid].time)
23:         if ts.time < t_mmt then
24:             STORE_PLN(ts.time, t_mmt)
25:             return Err
26:         end if
27:         STORE_PLN(REPLAY, false)
28:     end if
29:     new' := ENCODE(True, new)
30:     r := CAS_PLN(loc, Null, new')
31:     if r is Err then
32:         t := rdtscp
33:         STORE_PLN(mmts[mid].time, t)
34:         flushopt mmts[mid].time; sfence
35:         STORE_PLN(ts.time, t)
36:         return Err
37:     end if
38:     flushopt loc
39:     new'' := ENCODE(False, new)
40:     CAS_PLN(loc, new', new'')
41:     return Ok
42: end function

43: function delete(loc, old, new, mid)
44:     if REPLAY then
45:         next := LOAD_PLN(old.repl)
46:         (tid_new, o_new) := DECODER(next)
47:         if tid = tid_new then
48:             goto 67
49:         end if
50:         t_mmt := LOAD_PLN(mmts[mid].time)
51:         if ts.time < t_mmt then
52:             STORE_PLN(ts.time, t_mmt)
53:             return Err
54:         end if
55:         STORE_PLN(REPLAY, false)
56:     end if
57:     new' := ENCODER(tid, new)
58:     r := CAS_PLN(old.repl, Null, new')
59:     if r is (Err cur) then
60:         t := rdtscp
61:         STORE_PLN(mmts[mid].time, t)
62:         flushopt mmts[mid].time; sfence
63:         STORE_PLN(ts.time, t)
64:         _ := Help(loc, cur)
65:         return Err
66:     end if
67:     flushopt old.repl
68:     CAS_PLN(loc, old, new)
69:     DeferFlush(loc); Retire(old)
70:     return old
71: end function

72: function Help(loc, old)
73:     if old = Null then return (Ok old)
74:     new := LOAD_PLN(old.repl)
75:     (tid_new, o_new) := DECODER(old)
76:     if o_new = Null then return (Ok old)
77:     t := rdtsc
78:     cur := LOAD_PLN(loc)
79:     if cur ≠ old then return (Ok cur)
80:     if rdtsc < t + PATIENCE then goto 77
81:     flushopt old.repl
82:     r := CAS_PLN(loc, old, o_new)
83:     return (r is (Err e)) ? (Ok e) : (Ok o_new)
84: end function
```

persist bit (L39-L40). It is okay to let the second CAS fail (L40) because it means a concurrent operation should have persisted loc and cleared the persist bit.

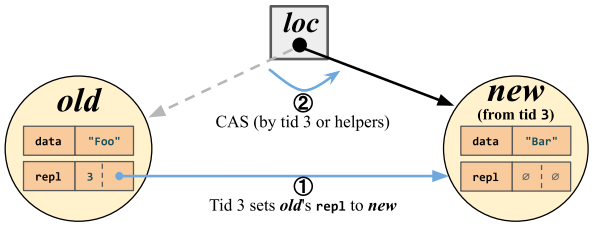


Figure B.1: Delete operation steps.

Deletion The `delete` operation, illustrated in Fig. B.1, receives a location (loc), an old pointer value to a valid memory block (old), an new pointer value (new), and the memento id (`mid`); and atomically updates loc from old to new in a persistent manner (L43). `delete` first checks if it is executed in the replay mode; otherwise, it ❶ tries to atomically install new annotated with the current thread id to old's `repl` field (L57-L58); if it fails, helps the completion of concurrent delete operations to guarantee lock freedom, records such a fact to the memento, and reports the failure (L65, see §B.1.5 for details on helping); ❷ flushes old's `repl` field (L67); ❸ tries to replace loc's value from old to new, persists loc in a *deferred* manner, and *retires* old (L68-L69); and ❹ returns old (L70). Here, we retire old so that it will be freed once it is no longer accessible from the other threads using safe memory reclamation schemes such as hazard pointers [Michael 2004] and epoch-based reclamation [Fraser 2004]. We also ensure loc is flushed at least before old is freed using DeferFlush so that loc points to a valid memory block even in case of crashes (§3.5.1).

A delete operation is committed when the CAS on old's `repl` (L58) is persisted, while its effects are applied to loc later (L68). It is safe for concurrent operations to see an old value of loc even after a new value is committed, as they either would fail or can linearize before the deletion.

B.1.4 Replay

Insertion The replay execution of insertion needs to distinguish the cases when the pre-crash execution is interrupted before or after persisting the first CAS (L30). To this end, we use Attiya et al. [2019]'s *direct tracking* approach: in a replay execution, a block has been inserted to a DS if and only if (1) the block is still contained in the DS (L19); or (2) the block's `repl` field is populated, which means it is already deleted (L20). If it is not the case, the replay execution reads the timestamp checkpointed in the memento, and if it is more recent than the thread's latest observed timestamp, replays the failure (L23). Otherwise, the operation is being freshly executed, so exits the replay mode and continues (L27).

Deletion The replay execution of deletion needs to distinguish the cases when the pre-crash execution is interrupted ($\lightning_1$) after reporting a failure; or ($\lightning_2$) before successfully persisting the first CAS (L58); or ($\lightning_3$) after that. To this end, the replay execution of deletion ① loads and decodes the old's `repl`, and if its tid is the current thread id, then resumes from the normal execution's step ❷ (L48); ② reads the timestamp checkpointed in the memento, and if it is more recent than the thread's latest observed timestamp, replays the failure (L51); and ② otherwise, the operation is being freshly executed, so exits the replay mode and continues the normal execution (L55). For each case, the post-crash replay execution correctly resumes the operation as follows:

($\lightning_1$) Since the operation performed nothing to old's `repl`, it goes to L51 and replays the failure.

($\lightning_2$) Since the operation performed nothing to old's `repl`, it goes to L55, exits the replay mode, and continues the operation.

Algorithm 8 Michael-Scott Queue's Enqueue Operation with Volatile Cache Optimization

```
1:  function Enqueue(q, val, mid)
2:      let blk := chkpt(λ.e_blk, mid.blk);
3:      loop
4:          let tail := Load_vol(q.tail);
5:          let next := Load(tail.next);
6:          if next = Null then
7:              CAS_vol(q.tail, tail, next);
8:              continue
9:          end if
10:         let succ := insert(tail.next, blk, q, mid.cas);
11:         if succ then
12:             if ¬REPLAY then CAS_vol(q.tail, tail, blk);
13:             return
14:         end if
15:     end loop
16: end function
    e_blk ≜ blk := palloc(⟨val : val ; next : Null⟩); blk
```

($\lightning_3$) Since new should contain the current thread id, it goes to L67 and correctly resumes from the normal execution's step ❷.

B.1.5 Helping

A delete operation may help an ongoing concurrent delete operation's second CAS. Such a help is essential for lock freedom because the concurrent operation may be committed—have successfully performed the first CAS—while its effects have not been applied to loc yet. The Help operation receives a location (loc) and an old pointer value (old) and returns a settled value of loc by possibly helping the second CAS of ongoing deletions (L72) as follows: it ❶ returns old if it is the settled value Null (L73); ❷ loads and decodes old.`repl` as new (L74-L75); ❸ returns old if `repl` is Null and thus old is settled (L76); ❹ tries to read a pointer value for a while (L77-L80); ❺ flushes old.`repl` (L81); ❻ tries to update loc from old to new, and regardless of the result, returns the current value of loc (L82-L83).

B.2 Extending Memento Framework with Advanced Optimizations

We introduce advanced primitive operations and type derivation rules used for the implementation of *MSQ-mmt-O1*, *MSQ-mmt-O2*, and *Clevel-mmt* (§3.5).

B.2.1 Volatile Cache Optimization

Example: Michael-Scott Queue To motivate volatile cache optimization, consider Michael-Scott's queue (MSQ) presented in Algorithm 8. For volatile memory, its enqueue operation proceeds as follows:

(1) It allocates a new block with the given value (L2).

(2) It dereferences the queue's tail pointer and its next block (L4, L5).

(3) If $next$ is occupied, then $tail$ is stale, so it tries to advance it and retries the operation (L6).

(4) Tries to append the new block by a CAS (L10).

(5) If successful, advances the $tail$ pointer and returns (L11). Otherwise, retries the operation.

Observation We may transform MSQ for volatile memory to persistent memory by checkpointing all variables including $tail$ and $next$. However, the evaluation result for *MSQ-mmt-O0* shows that such an algorithm is slower than hand-tuned detectable versions of MSQ (§3.6). The primary reason is that, as Friedman et al. [2018]; Li and Golab [2021] observed, $tail$ does not need to be persistent across crashes because the tail pointer is necessary just to satisfy a simple invariant: it should be reachable from the head pointer. Even if the tail pointer's value is lost due to a crash, we can easily recover a value that satisfies the invariant, e.g., we can re-initialize the tail pointer with the head pointer. We capture this idea with a general *volatile cache* optimization, e.g., placing $tail$ in DRAM so that we do not need to persist its writes.

Primitive Operations To this end, we can introduce the following operations:

(1) $vl \coloneqq \textsc{Alloc}_{\text{VOL}}(\vec{s})$: it allocates a *volatile location*, $vl \in \text{VLoc}$, that is semantically distinguished from PM location (PLoc). The allocation is annotated with statements, $\vec{s}$, that is executed to initialize the value at vl not only for the first allocation but also for crashes. To capture this in the semantics, machine transitions admit a step that randomly picks a volatile location and re-initialize it with the provided statements.

(2) $\textsc{Load}_{\text{VOL}}(vl)$: loads from the volatile location vl.

(3) $\text{CAS}_{\text{VOL}}(vl, v_1, v_2)$: tries to atomically update vl from v_1 to v_2.

As discussed above, we consider $q.\mathsf{tail}$ a volatile location and initialize it with statements, $\vec{s}$, that returns the queue's head pointer. As such, the tail pointer's invariant—reachable from the head pointer—is always maintained even after crashes. Also, we use volatile location operations accordingly.

Type Derivation Rules In the presence of volatile locations, the type system can be generalized as follows.

- ***Variable Context***: The type system presented in Fig. 3.2 assumes all variables are *stable*: their values are preserved even after crashes. However, e.g., the queue example's $tail$ is unstable in that its value can be changed after crashes. As such, we should distinguish those variables that are stable and those that are unstable (precise definition omitted).
- ***Volatile Parameters***: The type system allows deterministically replayed functions to have volatile parameters, e.g., loc of Insert (§B.1).
- ***Unstable Statements***: The type system allows unstable read-write statements on volatile locations (e.g., volatile CASes), but only under the condition that these statements use only those values that are produced in pre-crash executions and retrieved from mementos. For instance, the first volatile CAS's arguments are $q.\mathsf{tail}$, $tail$, and $next$, which is directly from the operation's argument ($q.\mathsf{tail}$) or freshly calculated in the latest execution ($tail$ and $next$). For another instance, even though one of the second volatile CAS's arguments is $\underline{blk}$, which may be produced in pre-crash executions and retrieved from mementos, but we execute the CAS only under the condition $\neg REPLAY$, which means the operation is not replayed and thus freshly executed. As such, even $\underline{blk}$'s value is freshly calculated.

The last condition is crucial not to overwrite volatile locations with stale checkpointed values. For instance, suppose blk was already inserted to the queue before crashes and a post-crash execution reaches L12. If it were

successfully executing the CAS and writing blk to the volatile location q.tail, then it may break the tail pointer's invariant because the queue's head pointer may have already passed blk in its linked-list. We prevent such an error by executing the CAS only in the normal execution.

Volatile locations should be used with caution—hence an advanced technique—because we assume they can be re-initialized at arbitrary moments as a machine step. As such, they should be used only as a *cache* of PM with a well-defined invariant (hence the name of the optimization).

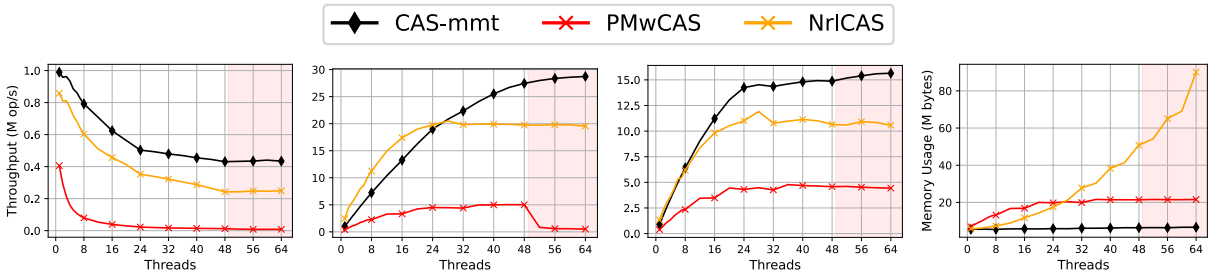
Future Work We believe our treatment of the volatile cache optimization subsumes the existing use cases of data placement in volatile locations (especially DRAM). We leave as future work recasting persistent index structures [Oukid et al. 2016; Friedman et al. 2018; Li and Golab 2021] and those DSs that *mirror* data in both DRAM and PM [Zuriel et al. 2019; Friedman et al. 2021] to the Memento framework.

B.2.2 Try Loop Optimization

In porting Clevel [Chen et al. 2020] to our framework, we discover a pattern, which we call *try loop*, that is supported with the **loop** rule, but only inefficiently. In a try loop, we iterate over elements and repeatedly execute a memento function until it returns a successful result. There is no preference on the elements and a single successful execution of the function for any element suffices. For instance, Clevel is based on the open hashing scheme in which you may insert a value (memento function) to one of multiple slots (elements). If such a pattern were represented with the composition rules in Fig. 3.2, the **loop** rule would checkpoint the index for each iteration, significantly degrading the performance. To optimize such a pattern, we additionally allow the following pattern with a new type derivation rule, **loop-try** (definition omitted, a straightforward translation of the definition of TryLoop):

```
1:  function TryLoop(i_init, next, arg, f, mid)
2:      if REPLAY then
3:          if ChkptPeek(mid.fail) is (Ok ()) then return Err
4:          if ChkptPeek(mid.arg) is (Ok arg) and f(arg, mid.f, True) is (Ok res) then return (Ok res)
5:          Store_pln(REPLAY, false)
6:      end if
7:      loop
8:          let i := φ(i_init, next(i)); let arg := arg(i);
9:          if arg is (Ok arg) then
10:             Checkpoint(arg, mid.arg);
11:             if f(arg, mid.f) is (Ok res) then return (Ok res)
12:         else
13:             Checkpoint((), mid.fail); return Err
14:         end if
15:     end loop
16: end function
```

In essence, for each index i, the function tries to perform $f(arg(i))$, and if successful, returns the result (L11); and if the iteration is over ($arg(i) = \text{Err}$), returns the failure (L13). For detectable recovery, we (1) checkpoint the failure (L13) and replay it in the recovery mode with ChkptPeek function that returns the checkpointed value (L3); (2) checkpoint the last argument (L10) and recover it in the recovery mode (L4); (3) execute the memento f (L11) and replay it in the recovery mode (L4); and (4) in the recovery mode, after trying to recover the last execution, begin from the initial index (i_{init}) as if it is normal.



(a) Throughput (1 location) (b) Throughput (1K locations) (c) Throughput (1M locations) (d) Memory usage (1M locations)

Figure B.2: Multi-threaded throughput of detectable CASes.

B.2.3 Invariant-Based Optimization

As the last optimization, we exploit invariants to optimize *MSQ-mmt-O1* to *MSQ-mmt-O2*. More specifically, we follow Friedman et al. [2018]; Li and Golab [2021] to rely on the invariant that all links from the head pointer to the tail pointer are already persisted.

By exploiting this invariant, we can reduce the number of CASes from two to one for MSQ's enqueue operation. Without the invariant, we employ the *link-and-persist* technique [Wang et al. 2018; David et al. 2018] to guarantee that readers can read only persisted pointer values. However, this technique requires two CASes: the first with marking the *dirty* bit and the second to remove the mark. The two CASes significantly degrade the performance because, after the first CAS, the target cacheline is flushed to the PM hardware so that the second CAS should bring the same cacheline from the PM hardware again. In contrast, with the invariant, we can ensure the same property as follows:

(1) The enqueue operations writes pointer values without marking the dirty bit using only a single CAS.

(2) The dequeue operations persist pointer values before using them. But if a pointer value is before the tail pointer, the invariant guarantees that the pointer value is already persisted. Only when the pointer value is beyond the tail pointer, the dequeue operations persist it.

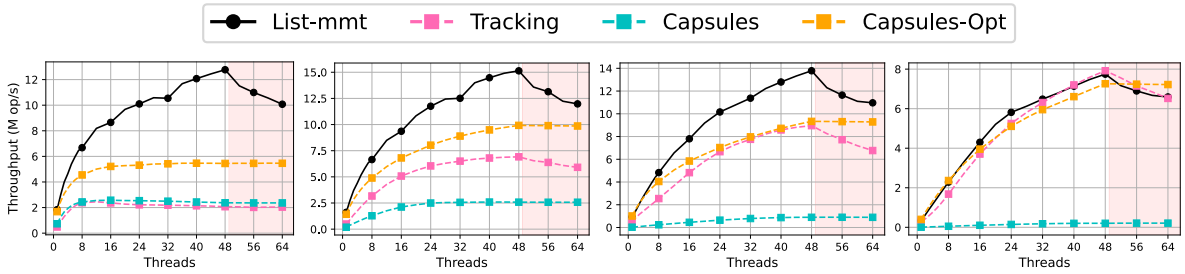
This optimization, however, is not currently captured in our type system: the optimization is based on an invariant on the runtime information, which is beyond the capability of the static type system. But we can incrementally reason about the safety of this invariant-based optimization (at least informally) on top of *MSQ-mmt-O1* whose detectability is statically reasoned about in our type system. Formalizing such an incremental reasoning is left as future work (§3.7).

B.3 Full Evaluation Results

In this section, we report the full experimental results. For a detailed analysis of the results, see §3.6; for the code and script used for the experiments, see the supplementary materials [Cho et al. 2023a].

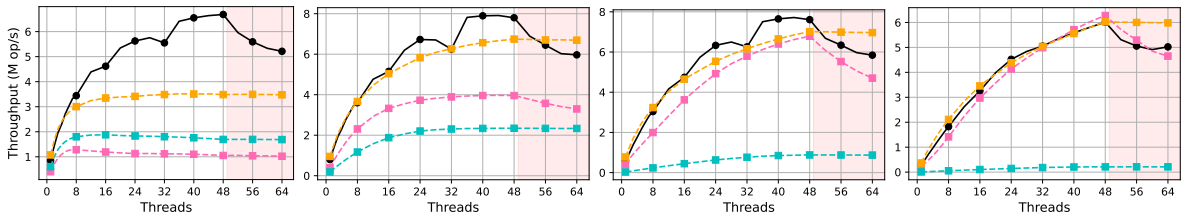
Performance of Detectably CAS Fig. B.2 is the same with Fig. 3.5. See §3.6.2 for an analysis.

Performance of Detectable List Fig. B.3 and Fig. B.4 illustrate the performance of lists for read- and update-intensive workloads, respectively. They collectively subsume Fig. 3.6. See §3.6.2 for an analysis.



(a) 20 key range (b) 100 key range (c) 500 key range (d) 2K key range

Figure B.3: Multi-threaded throughput of persistent lists for read intensive.



(a) 20 key range (b) 100 key range (c) 500 key range (d) 2K key range

Figure B.4: Multi-threaded throughput of persistent lists for update intensive.

Performance of Detectable Queue Fig. B.5 is the same with Fig. 3.7. See §3.6.2 for an analysis.

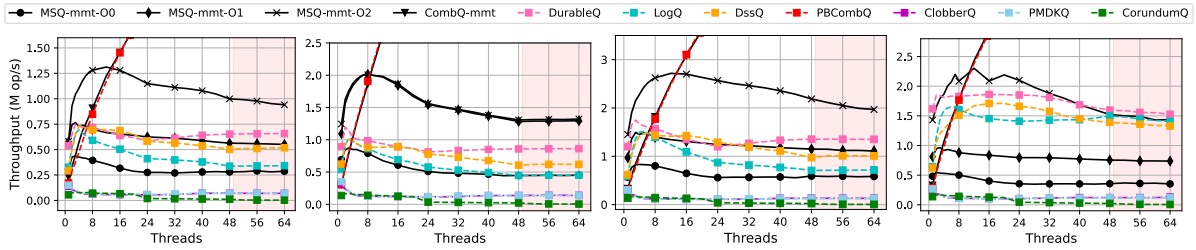
Performance of Detectable Hash Table For hash tables, we use the evaluation workloads of a PM hash table evaluation paper [Hu et al. 2021]: "We stress test each hash table with individual operations (insert, positive and negative search, and delete) and mixed workloads. Negative search means searching keys that do not exist." "We initialize hash tables with a capacity that can accommodate 16M key-value pairs." "To measure insert-only performance, we insert 200M records into an empty hash table directly. To measure the performance of the search and delete operation and the mixed workloads, we first initialize the hash table with 200M items (loading phase), then execute 200M operations to perform the measurements (measuring phase)." "We run the experiments with workloads using uniform distribution and skewed distribution (self-similar with a factor of 0.2, which means 80% of accesses focus on 20% of keys)." "We consider fixed-length (8 bytes) keys and values."

Fig. B.6 and Fig. B.7 illustrate the performance of hash tables for uniform and skewed distributions. Fig. B.6 subsumes Fig. 3.8. *Clevel* exhibits a segmentation fault for the balanced workload with 32 threads (hence the blank). See §3.6.2 for an analysis.

B.4 Full Core Language Semantics

The definitions presented in this section subsume those presented in §3.3.1. For a detailed discussion, see §3.3.1. Fig. B.8 presents the core language syntax. Fig. B.9 presents the states used in the core language semantics. Fig. B.10 defines the machine transitions. Fig. B.11 defines the memory transitions. Fig. B.12 defines the thread transitions for non-memento steps. Fig. B.13 defines the thread transitions for memento steps. Here, the shaded area represents persistent data written to PM.

Note that in **LOOP**, we *copy* the register map, ts_1.regs, for the loop body (ts_2.regs) loop continuation (in $\overrightarrow{c_2}$). We define semantics in this way to ensure a loop body does not modify the variables defined outside of the loop,



(a) Enqueue-dequeue (b) Enqueue-20% (c) Enqueue-50% (d) Enqueue-80%

Figure B.5: Multi-threaded throughput of persistent queues.

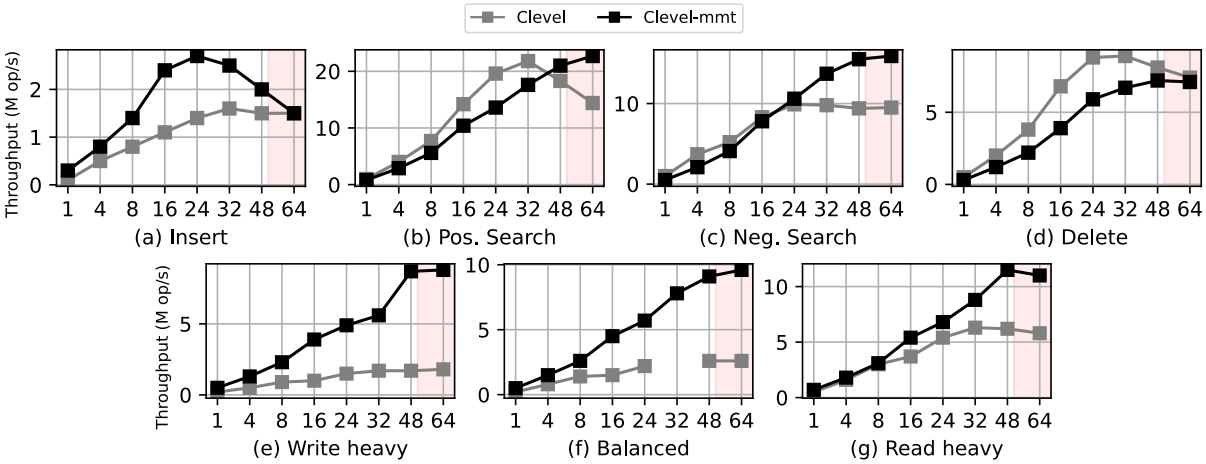


Figure B.6: Multi-threaded throughput of hash tables for uniform distribution.

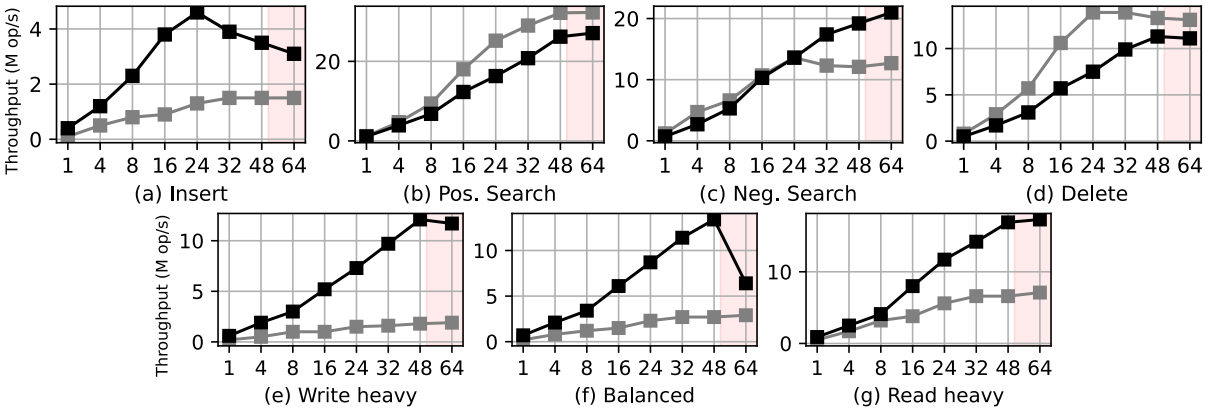


Figure B.7: Multi-threaded throughput of hash tables for self similar distribution with factor 0.2.

in order to simplify our our theoretical development. This design does not fundamentally limits programmability: given a program, you can perform SSA transformation to ensure that a loop body does not modify the variables defined outside of the loop. Furthermore, we require a loop to explicitly continue with the `continue` e instruction to proceed to the next iteration. That is required to guarantee the loop-dependent variable is properly set.

B.5 Full Type System

Fig. B.14 presents the type system. This definition subsumes Fig. 3.2 and additionally defines read-only

$$
\begin{array}{rcll}
p & ::= & [\delta]\ \vec{s_1} \mid\mid \ldots \mid\mid \vec{s_n} & \textit{program} \\
s \in \mathrm{Stmt} & ::= & r \coloneqq e \mid r \coloneqq \mathtt{pload}(e) \mid r \coloneqq \mathtt{palloc}(e) & \textit{assignment \& PM} \\
& \mid & \mathtt{if}\ (e)\ \vec{s_\mathsf{t}}\ \vec{s_\mathsf{f}} \mid \mathtt{loop}\ r\ e\ \vec{s} \mid \mathtt{continue}\ e \mid \mathtt{break} & \textit{control constructs} \\
& \mid & r \coloneqq f(\vec{e}) \mid \mathtt{return}\ e & \textit{function call/return} \\
& \mid & r \coloneqq \mathtt{chkpt}(\vec{s}, e_{\mathsf{mid}}) \mid r \coloneqq \mathtt{pcas}(e_{\mathsf{loc}}, e_{\mathsf{old}}, e_{\mathsf{new}}, e_{\mathsf{mid}}) & \textit{detectable op.} \\
e \in \mathrm{Expr} & ::= & () \mid z \mid b \mid \mathit{mid} \mid r \mid (e_1\ \mathit{op}\ e_2) \mid e.i \mid (e_1, e_2) \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e & \textit{pure expr.} \\
& \mid & \mathsf{match}\ e\ \{\ \mathsf{inl}\ e_\mathsf{l} \Rightarrow e_\mathsf{l}'\ ,\ \mathsf{inr}\ e_\mathsf{r} \Rightarrow e_\mathsf{r}'\} \mid \epsilon \mid e.\mathit{lab} \mid\ \ldots & \\
v \in \mathrm{Val} & ::= & () \mid z \mid b \mid \mathit{mid} \mid (v_1, v_2) \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v & \textit{value}
\end{array}
$$

$$
z \in \mathbb{Z} \qquad b \in \mathbb{B} \qquad \mathit{op} \in \mathrm{Op} \qquad r \in \mathrm{VReg} \triangleq \mathbb{N} \qquad l \in \mathrm{PLoc} \triangleq \mathbb{N}
$$

$$
f \in \mathrm{FnId} \qquad \delta \in \mathrm{Env} \triangleq \mathrm{FnId} \rightharpoonup (\overrightarrow{\mathrm{VReg}} \times \overrightarrow{\mathrm{Stmt}}) \qquad \mathit{lab} \in \mathrm{Label} \qquad \mathit{mid} \in \overrightarrow{\mathrm{Label}}
$$

Figure B.8: Core language syntax.

$$
\begin{array}{rcll}
c \in \mathrm{Cont} & ::= & \mathtt{loopCont}(\sigma, r, \vec{s_\mathsf{b}}, \overrightarrow{s_\mathsf{cont}}) & \textit{loop context} \\
& \mid & \mathtt{fnCont}(\sigma, r, \vec{s}) & \textit{function context} \\
& \mid & \mathtt{chkptCont}(\sigma, r, \vec{s}, \mathit{mid}) & \textit{checkpoint context}
\end{array}
$$

$$
t \in \mathrm{Time} \triangleq \mathbb{N} \qquad \mathit{tid} \in \mathrm{TId} \triangleq \mathbb{N}
$$

$$
\sigma \in \mathrm{VRegMap} \triangleq \mathrm{VReg} \rightharpoonup \mathrm{Val} \qquad \mathit{ts} \in \mathrm{TState} \triangleq \langle \mathsf{regs} : \mathrm{VRegMap}; \mathsf{time} : \mathrm{Time} \rangle
$$

$$
\mathit{mmts} \in \mathrm{Mmts} \triangleq \overrightarrow{\mathrm{Label}} \to \langle \mathsf{val} : \mathrm{Val}; \mathsf{time} : \mathrm{Time} \rangle
$$

$$
T \in \mathrm{Thread} \triangleq \overrightarrow{\mathrm{Stmt}} \times \overrightarrow{\mathrm{Cont}} \times \mathrm{TState} \times \mathrm{Mmts}
$$

$$
\mathit{ev} \in \mathrm{Event} ::= \mathsf{R}(l, v) \mid \mathsf{U}(l, v_{\mathsf{old}}, v_{\mathsf{new}}) \qquad \mathit{tr} \in \overrightarrow{\mathrm{Event}}
$$

$$
\mathsf{mem} \in \mathrm{Mem} \triangleq \mathrm{PLoc} \to \mathrm{Val} \qquad M \in \mathrm{Machine} \triangleq \overrightarrow{\mathrm{Thread}} \times \mathrm{Mem}
$$

Figure B.9: Core language semantics: states.

statement judgment of the form $\Delta \vdash_{\mathsf{RO}} \vec{s}$. For a detailed discussion, see §3.3.2.

B.6 Proof of the Detectability Theorem

We prove the detectability theorem (Theorem 3.3.1).

B.6.1 Transitions

Definition B.6.1 (Reflexive Transitive Closure with Concatenated Traces). *Let $\overset{tr}{\hookrightarrow}$ be a relation over a trace. We define $\overset{tr}{\hookrightarrow}^*$ be the reflexive transitive closure of $\overset{tr}{\hookrightarrow}$ with concatenated traces.*

For instance, if $A \overset{tr_1}{\hookrightarrow} B$ and $B \overset{tr_2}{\hookrightarrow} C$, then we have $A \overset{tr}{\hookrightarrow}^* C$ where $tr = tr_1 +\!\!+ tr_2$.

$$ts_{\mathsf{init}} \triangleq \langle \mathsf{regs} = \{\mathsf{mid} \mapsto []\}; \mathsf{time} = 0\rangle$$

$$mmts_{\mathsf{init}} \triangleq \{mid \mapsto \langle \mathrm{Val} = (); \mathsf{time} = 0\rangle \mid mid \in \overrightarrow{\mathrm{Label}}\}$$

$$\mathsf{mem}_{\mathsf{init}} \triangleq \{l \mapsto () \mid l \in \mathrm{PLoc}\}$$

$$init([\delta]\ \vec{s_1} \mid\mid \ldots \mid\mid \vec{s_n}) \triangleq \langle \lambda tid \in [1..n].\langle \overrightarrow{s_{tid}}, [], ts_{\mathsf{init}}, mmts_{\mathsf{init}}\rangle, \mathsf{mem}_{\mathsf{init}}\rangle$$

$\boxed{M_1 \xrightarrow{tr}_p M_2}$

(MACHINE-STEP)

$$\frac{\begin{array}{c}\mathcal{T}_1[tid] = (\vec{s_1}, \vec{c_1}, ts_1, mmts_1) \\ \mathcal{T}_2 = \mathcal{T}_1[tid \mapsto (\vec{s_2}, \vec{c_2}, ts_2, mmts_2)] \\ \vec{s_1}, \vec{c_1}, ts_1, mmts_1 \xrightarrow{tr}_{p.\delta} \vec{s_2}, \vec{c_2}, ts_2, mmts_2 \\ \mathsf{mem}_1 \xrightarrow{tr} \mathsf{mem}_2\end{array}}{(\mathcal{T}_1, \mathsf{mem}_1) \xrightarrow{tr|_{\mathsf{U}}}_p (\mathcal{T}_2, \mathsf{mem}_2)}$$

(MACHINE-CRASH)

$$\frac{\begin{array}{c}\mathcal{T}_1[tid] = (\vec{s_1}, \vec{c_1}, ts_1, mmts_1) \\ \mathcal{T}_2 = \mathcal{T}_1[tid \mapsto (\overrightarrow{p.s_{tid}}, [], ts_{\mathsf{init}}, mmts_1)]\end{array}}{(\mathcal{T}_1, \mathsf{mem}) \xrightarrow{[]}_p (\mathcal{T}_2, \mathsf{mem})}$$

Figure B.10: Core language semantics: machine transitions.

$\boxed{\mathsf{mem}_1 \xrightarrow{tr} \mathsf{mem}_2}$

(READ)

$$\frac{\mathsf{mem}[l] = v \qquad ev = \mathsf{R}(l, v)}{\mathsf{mem} \xrightarrow{[ev]} \mathsf{mem}}$$

(UPDATE)

$$\frac{ev = \mathsf{U}(l, v_{\mathsf{old}}, v_{\mathsf{new}}) \qquad \mathsf{mem}_1[l] = v_{\mathsf{old}} \qquad \mathsf{mem}_2 = \mathsf{mem}_1\left[l \mapsto v_{\mathsf{new}}\right]}{\mathsf{mem}_1 \xrightarrow{[ev]} \mathsf{mem}_2}$$

Figure B.11: Core language semantics: memory transitions.

Definition B.6.2 (Transitive Closure with Concatenated Traces). *Let $\overset{tr}{\hookrightarrow}$ be a relation over a trace. We define $\overset{tr}{\hookrightarrow}^{+}$ be the transitive closure of $\overset{tr}{\hookrightarrow}$ with concatenated traces.*

B.6.2 Lifting

Definition B.6.3 (Sequence with Continuations). *We define sequence of statements with continuations, $(\vec{s}, \vec{c}) +\!\!+ \vec{s_\alpha}$, as follows:*

$$\texttt{loopCont}(\sigma, r, \vec{s_\mathsf{b}}, \vec{s}) +\!\!+ \vec{s_\alpha} \triangleq \texttt{loopCont}(\sigma, r, \vec{s_\mathsf{b}}, \vec{s} +\!\!+ \vec{s_\alpha})$$

$$\texttt{fnCont}(\sigma, r, \vec{s}) +\!\!+ \vec{s_\alpha} \triangleq \texttt{fnCont}(\sigma, r, \vec{s} +\!\!+ \vec{s_\alpha})$$

$$\texttt{chkptCont}(\sigma, r, mid, \vec{s}) +\!\!+ \vec{s_\alpha} \triangleq \texttt{chkptCont}(\sigma, r, mid, \vec{s} +\!\!+ \vec{s_\alpha})$$

$$(\vec{s}, []) +\!\!+ \vec{s_\alpha} \triangleq (\vec{s} +\!\!+ \vec{s_\alpha}, [])$$

$$(\vec{s}, \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ [c_{\mathsf{base}}]) +\!\!+ \vec{s_\alpha} \triangleq (\vec{s}, \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ [c_{\mathsf{base}} +\!\!+ \vec{s_\alpha}])$$

Lemma B.6.4 (Sequence Lifting). *For all $\delta, tr, \vec{s_1}, \vec{s_2}, \vec{s}, ts_1 ts_2, mmts_1, mmts_2$, we have:*

$$\begin{aligned} & \vec{s_1}, \vec{c_1}, ts_1, mmts_1 \xrightarrow{tr}_\delta{}^{*} \vec{s_2}, \vec{c_2}, ts_2, mmts_2 \\ \implies\ & \exists \overrightarrow{s_{m1}}, \overrightarrow{s_{m2}}, \overrightarrow{c_{m1}}, \overrightarrow{c_{m2}}. \\ & \overrightarrow{s_{m1}}, \overrightarrow{c_{m1}}, ts_1, mmts_1 \xrightarrow{tr}_\delta{}^{*} \overrightarrow{s_{m2}}, \overrightarrow{c_{m2}}, ts_2, mmts_2 \\ \wedge\ & (\overrightarrow{s_{m1}}, \overrightarrow{c_{m1}}) = (\vec{s_1}, \vec{c_1}) +\!\!+ \vec{s} \\ \wedge\ & (\overrightarrow{s_{m2}}, \overrightarrow{c_{m2}}) = (\vec{s_2}, \vec{c_2}) +\!\!+ \vec{s}\ . \end{aligned}$$

$$\mathsf{Loops}(\overrightarrow{c_{\mathsf{loops}}}) \triangleq \forall c \in \overrightarrow{c_{\mathsf{loops}}}.\ \exists \sigma, r, \overrightarrow{s_{\mathsf{b}}}, \overrightarrow{s_{\mathsf{cont}}}.\ c = \mathtt{loopCont}(\sigma, r, \overrightarrow{s_{\mathsf{b}}}, \overrightarrow{s_{\mathsf{cont}}})$$

$$\boxed{\overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1 \xrightarrow{[ev]}_\delta \overrightarrow{s_2}, \overrightarrow{c_2}, ts_2, mmts_2}$$

(ASSIGN)

$$\frac{ts_1.\mathsf{regs}(e) = v \qquad ts_2 = ts_1[\mathsf{regs} \mapsto ts_1.\mathsf{regs}[r \mapsto v]]}{(r := e) :: \overrightarrow{s}, \overrightarrow{c}, ts_1, mmts \xrightarrow{[]}_\delta \overrightarrow{s}, \overrightarrow{c}, ts_2, mmts}$$

(PLOAD)

$$\frac{ts_1.\mathsf{regs}(e) = l \qquad ev = \mathsf{R}(l, v) \qquad ts_2 = ts_1[\mathsf{regs} \mapsto ts_1.\mathsf{regs}[r \mapsto v]]}{(r := \mathtt{pload}(e)) :: \overrightarrow{s}, \overrightarrow{c}, ts_1, mmts \xrightarrow{[ev]}_\delta \overrightarrow{s}, \overrightarrow{c}, ts_2, mmts}$$

(PALLOC)

$$\frac{ts_1.\mathsf{regs}(e) = v \qquad ev = \mathsf{R}(l, v) \qquad ts_2 = ts_1[\mathsf{regs} \mapsto ts_1.\mathsf{regs}[r \mapsto l]]}{(r := \mathtt{palloc}(e)) :: \overrightarrow{s}, \overrightarrow{c}, ts_1, mmts \xrightarrow{[ev]}_\delta \overrightarrow{s}, \overrightarrow{c}, ts_2, mmts}$$

(BRANCH)

$$\frac{ts.\mathsf{regs}(e) = v \qquad \overrightarrow{s_{\mathsf{d}}} = \text{if } v \text{ then } \overrightarrow{s_{\mathsf{t}}} \text{ else } \overrightarrow{s_{\mathsf{f}}}}{(\mathtt{if}\ (e)\ \overrightarrow{s_{\mathsf{t}}}\ \overrightarrow{s_{\mathsf{f}}}) :: \overrightarrow{s}, \overrightarrow{c}, ts, mmts \xrightarrow{[]}_\delta \overrightarrow{s_{\mathsf{d}}} +\!\!+ \overrightarrow{s}, \overrightarrow{c}, ts, mmts}$$

(LOOP)

$$\frac{ts_1.\mathsf{regs}(e) = v \qquad ts_2 = ts_1\left[\mathsf{regs} \mapsto ts_1.\mathsf{regs}[r \mapsto v]\right] \qquad \overrightarrow{c_2} = \mathtt{loopCont}(ts.\mathsf{regs}, r, \overrightarrow{s}, \overrightarrow{s_{\mathsf{cont}}}) :: \overrightarrow{c_1}}{(\mathtt{loop}\ r\ e\ \overrightarrow{s}) :: \overrightarrow{s_{\mathsf{cont}}}, \overrightarrow{c_1}, ts_1, mmts \xrightarrow{[]}_\delta \overrightarrow{s}, \overrightarrow{c_2}, ts_2, mmts}$$

(CONTINUE)

$$\frac{ts_1.\mathsf{regs}(e) = v \qquad ts_2 = ts_1\left[\mathsf{regs} \mapsto \sigma[r \mapsto v]\right] \qquad \overrightarrow{c} = \mathtt{loopCont}(\sigma, r, \overrightarrow{s_{\mathsf{b}}}, \overrightarrow{s_{\mathsf{cont}}}) :: \overrightarrow{c_{\mathsf{rem}}}}{(\mathtt{continue}\ e) :: \overrightarrow{s}, \overrightarrow{c}, ts_1, mmts \xrightarrow{[]}_\delta \overrightarrow{s_{\mathsf{b}}}, \overrightarrow{c}, ts_2, mmts}$$

(BREAK)

$$\frac{ts_2 = ts_1\left[\mathsf{regs} \mapsto \sigma\right] \qquad \overrightarrow{c_1} = \mathtt{loopCont}(\sigma, r, \overrightarrow{s_{\mathsf{b}}}, \overrightarrow{s_{\mathsf{cont}}}) :: \overrightarrow{c_2}}{(\mathtt{break}) :: \overrightarrow{s}, \overrightarrow{c_1}, ts_1, mmts \xrightarrow{[]}_\delta \overrightarrow{s_{\mathsf{cont}}}, \overrightarrow{c_2}, ts_2, mmts}$$

(CALL)

$$\frac{ts.\mathsf{regs}(\overrightarrow{e}) = \overrightarrow{v} \qquad \delta(f) = (\overrightarrow{prms}, \overrightarrow{s_f}) \qquad ts_2 = ts_1\left[\mathsf{regs} \mapsto \sqcup_i [prms_i \mapsto v_i]\right] \qquad \overrightarrow{c_2} = \mathtt{fnCont}(ts_1.\mathsf{regs}, r, \overrightarrow{s}) :: \overrightarrow{c_1}}{(r := f(\overrightarrow{e})) :: \overrightarrow{s}, \overrightarrow{c_1}, ts_1, mmts \xrightarrow{[]}_\delta \overrightarrow{s_f}, \overrightarrow{c_2}, ts_2, mmts}$$

(RETURN)

$$\frac{ts.\mathsf{regs}(e) = v \qquad ts_2 = ts_1\left[\mathsf{regs} \mapsto \sigma[r \mapsto v]\right] \qquad \overrightarrow{c_1} = \overrightarrow{c_{\mathsf{loops}}} +\!\!+ [\mathtt{fnCont}(\sigma, r, \overrightarrow{s_2})] +\!\!+ \overrightarrow{c_2} \qquad \mathsf{Loops}(\overrightarrow{c_{\mathsf{loops}}})}{(\mathtt{return}\ e) :: \overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts \xrightarrow{[]}_\delta \overrightarrow{s_2}, \overrightarrow{c_2}, ts_2, mmts}$$

Figure B.12: Core language semantics: thread transitions (non-memento steps).

PROOF SKETCH. Straightforward by induction on the transitions. □

LEMMA B.6.5 (CONTINUATION LIFTING). *For all* $\delta, tr, \overrightarrow{s_1}, \overrightarrow{s_2}, \overrightarrow{c_1}, \overrightarrow{c_2}, \overrightarrow{c_\alpha}, ts_1\, ts_2, mmts_1, mmts_2$*, we have:*

$$\overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1 \xrightarrow{tr}{}^*_\delta \overrightarrow{s_2}, \overrightarrow{c_2}, ts_2, mmts_2$$
$$\implies \overrightarrow{s_1}, \overrightarrow{c_1} +\!\!+ \overrightarrow{c_\alpha}, ts_1, mmts_1 \xrightarrow{tr/\overrightarrow{c_\alpha}}{}^*_\delta \overrightarrow{s_2}, \overrightarrow{c_2} +\!\!+ \overrightarrow{c_\alpha}, ts_2, mmts_2\ .$$

PROOF SKETCH. Straightforward from the definition. □

DEFINITION B.6.6 (MEMENTO ID EXPANSION). *For* $mid_{\mathsf{sfx}} \in \overrightarrow{\mathrm{Label}}$ *and* $labs \subseteq \mathrm{Label}$*, we define the* memento id expansion, $\mu(mid_{\mathsf{pfx}}, labs)$*, as follows:*

$$\mu(mid_{\mathsf{pfx}}, labs) \triangleq \{mid \in \overrightarrow{\mathrm{Label}} \mid \exists lab \in labs, mid_{\mathsf{sfx}}.\ mid = mid_{\mathsf{pfx}} +\!\!+ [lab] +\!\!+ mid_{\mathsf{sfx}}\}\ .$$

(CHKPT-CALL)

$$\frac{ts.\mathsf{regs}(e_{\mathsf{mid}}) = mid \quad mmts[mid] = \left\langle \begin{array}{l} \mathsf{val} \mapsto v_{\mathsf{mmt}}, \\ \mathsf{time} \mapsto t_{\mathsf{mmt}} \end{array} \right\rangle \quad t_{\mathsf{mmt}} \leq ts.\mathsf{time} \quad \vec{c_2} = \mathtt{chkptCont}(ts.\mathsf{regs}, r, \vec{s}, mid) :: \vec{c_1}}{(r \coloneqq \mathtt{chkpt}(\vec{s_c}, e_{\mathsf{mid}})) :: \vec{s}, \vec{c_1}, ts, mmts \xrightarrow{[]}_{\delta} \vec{s_c}, \vec{c_2}, ts, mmts}$$

(CHKPT-RETURN)

$$\frac{\vec{c_1} = \overrightarrow{c_{\mathsf{loops}}} +\!\!+ [\mathtt{chkptCont}(\sigma, r, \vec{s}, mid)] +\!\!+ \vec{c_2} \quad \mathsf{Loops}(\overrightarrow{c_{\mathsf{loops}}}) \quad ts_1.\mathsf{time} < t \quad v_r = ts_1.\mathsf{regs}(e) \quad ts_2 = \left\langle \begin{array}{l} \mathsf{regs} \mapsto \sigma[r \mapsto v_r], \\ \mathsf{time} \mapsto t \end{array} \right\rangle \quad mmts_2 = mmts_1 \left[mid \mapsto \left\langle \begin{array}{l} \mathsf{val} \mapsto v_r, \\ \mathsf{time} \mapsto t \end{array} \right\rangle \right]}{(\mathtt{return}\ e) :: \overrightarrow{s_{\mathsf{rem}}}, \vec{c_1}, ts_1, mmts_1 \xrightarrow{[]}_{\delta} \vec{s}, \vec{c_2}, ts_2, mmts_2}$$

(CHKPT-REPLAY)

$$\frac{ts.\mathsf{regs}(e_{\mathsf{mid}}) = mid \quad mmts[mid] = \left\langle \begin{array}{l} \mathsf{val} \mapsto v_{\mathsf{mmt}}, \\ \mathsf{time} \mapsto t_{\mathsf{mmt}} \end{array} \right\rangle \quad ts_1.\mathsf{time} < t_{\mathsf{mmt}} \quad ts_2 = \left\langle \begin{array}{l} \mathsf{regs} \mapsto ts_1.\mathsf{regs}[r \mapsto v_{\mathsf{mmt}}], \\ \mathsf{time} \mapsto t_{\mathsf{mmt}} \end{array} \right\rangle}{(r \coloneqq \mathtt{chkpt}(\_, e_{\mathsf{mid}})) :: \vec{s}, \vec{c}, ts_1, mmts \xrightarrow{[]}_{\delta} \vec{s}, \vec{c}, ts_2, mmts}$$

(PCAS-SUCC)

$$\frac{\begin{array}{c} ts_1.\mathsf{regs}(e_{\mathsf{loc}}) = l \quad ts_1.\mathsf{regs}(e_{\mathsf{old}}) = v_{\mathsf{old}} \quad ts_1.\mathsf{regs}(e_{\mathsf{new}}) = v_{\mathsf{new}} \quad ts_1.\mathsf{regs}(e_{\mathsf{mid}}) = mid \\ ev = \mathsf{U}(l, v_{\mathsf{old}}, v_{\mathsf{new}}) \quad v_r = (\textsc{true}, v_{\mathsf{old}}) \quad mmts_1[mid] = \left\langle \begin{array}{l} \mathsf{val} \mapsto v_{\mathsf{mmt}}, \\ \mathsf{time} \mapsto t_{\mathsf{mmt}} \end{array} \right\rangle \\ t_{\mathsf{mmt}} \leq ts_1.\mathsf{time} < t \quad ts_2 = \left\langle \begin{array}{l} \mathsf{regs} \mapsto ts_1.\mathsf{regs}[r \mapsto v_r], \\ \mathsf{time} \mapsto t \end{array} \right\rangle \quad mmts_2 = mmts_1 \left[mid \mapsto \left\langle \begin{array}{l} \mathsf{val} \mapsto v_r, \\ \mathsf{time} \mapsto t \end{array} \right\rangle \right] \end{array}}{(r \coloneqq \mathtt{pcas}(e_{\mathsf{loc}}, e_{\mathsf{old}}, e_{\mathsf{new}}, e_{\mathsf{mid}})) :: \vec{s}, \vec{c}, ts_1, mmts_1 \xrightarrow{[ev]}_{\delta} \vec{s}, \vec{c}, ts_2, mmts_2}$$

(PCAS-FAIL)

$$\frac{\begin{array}{c} ts_1.\mathsf{regs}(e_{\mathsf{loc}}) = l \quad ts_1.\mathsf{regs}(e_{\mathsf{old}}) = v_{\mathsf{old}} \quad ts_1.\mathsf{regs}(e_{\mathsf{mid}}) = mid \\ ev = \mathsf{R}(l, v) \quad v \neq v_{\mathsf{old}} \quad v_r = (\textsc{false}, v) \quad mmts_1[mid] = \left\langle \begin{array}{l} \mathsf{val} \mapsto v_{\mathsf{mmt}}, \\ \mathsf{time} \mapsto t_{\mathsf{mmt}} \end{array} \right\rangle \\ t_{\mathsf{mmt}} \leq ts_1.\mathsf{time} < t \quad ts_2 = \left\langle \begin{array}{l} \mathsf{regs} \mapsto ts_1.\mathsf{regs}[r \mapsto v_r], \\ \mathsf{time} \mapsto t \end{array} \right\rangle \quad mmts_2 = mmts_1 \left[mid \mapsto \left\langle \begin{array}{l} \mathsf{val} \mapsto v_r, \\ \mathsf{time} \mapsto t \end{array} \right\rangle \right] \end{array}}{(r \coloneqq \mathtt{pcas}(e_{\mathsf{loc}}, e_{\mathsf{old}}, e_{\mathsf{new}}, e_{\mathsf{mid}})) :: \vec{s}, \vec{c}, ts_1, mmts_1 \xrightarrow{[ev]}_{\delta} \vec{s}, \vec{c}, ts_2, mmts_2}$$

(PCAS-REPLAY)

$$\frac{ts_1.\mathsf{regs}(e_{\mathsf{mid}}) = mid \quad mmts[mid] = \left\langle \begin{array}{l} \mathsf{val} \mapsto v_{\mathsf{mmt}}, \\ \mathsf{time} \mapsto t_{\mathsf{mmt}} \end{array} \right\rangle \quad ts_1.\mathsf{time} < t_{\mathsf{mmt}} \quad ts_2 = ts_1 \left[\begin{array}{l} \mathsf{regs} \mapsto ts_1.\mathsf{regs}[r \mapsto v_{\mathsf{mmt}}], \\ \mathsf{time} \mapsto t_{\mathsf{mmt}} \end{array} \right]}{(r \coloneqq \mathtt{pcas}(e_{\mathsf{loc}}, e_{\mathsf{old}}, e_{\mathsf{new}}, e_{\mathsf{mid}})) :: \vec{s}, \vec{c}, ts_1, mmts \xrightarrow{[]}_{\delta} \vec{s}, \vec{c}, ts_2, mmts}$$

Figure B.13: Core language semantics: thread transitions (memento steps).

LEMMA B.6.7 (MEMENTO LIFTING). *For all* $\Delta, \delta, labs, tr, \vec{s}, \vec{s_\omega}, ts, ts_\omega, \vec{c_\omega}, mmts, mmts_\omega,$

$labs \in \mathcal{P}(\text{Label})$ $\quad$ $\text{FnType} ::= \mathsf{RO} \mid \mathsf{RW}$

$\Delta \in \text{EnvType} \triangleq \text{FnId} \to \text{FnType}$

$\boxed{\vdash p}$

(PROGRAM)

$$\frac{\vdash \delta : \Delta \qquad \Delta \vdash_{labs_{tid}} \overrightarrow{s_{tid}} \quad \text{for each } tid}{\vdash [\delta]\ \overrightarrow{s_1} \mid\mid \ldots \mid\mid \overrightarrow{s_n}}$$

$\boxed{\vdash \delta : \Delta}$

(ENV-EMPTY)

$$\frac{}{\vdash \ : []}$$

(ENV-RO)

$$\frac{\vdash \delta : \Delta \qquad \Delta \vdash_{\mathsf{RO}} \overrightarrow{s}}{\vdash \delta[f \mapsto (\overrightarrow{prms}, \overrightarrow{s})] : \Delta[f \mapsto \mathsf{RO}]}$$

(ENV-RW)

$$\frac{\vdash \delta : \Delta \qquad \Delta \vdash_{labs} \overrightarrow{s} \qquad \overrightarrow{prms_{\mathsf{all}}} = \overrightarrow{prms} +\!\!+ \{\mathsf{mid}\}}{\vdash \delta[f \mapsto (\overrightarrow{prms_{\mathsf{all}}}, \overrightarrow{s})] : \Delta[f \mapsto \mathsf{RW}]}$$

$\boxed{\Delta \vdash_{labs} \overrightarrow{s}}$

(EMPTY)

$$\frac{}{\Delta \vdash_{\emptyset} []}$$

(ASSIGN)

$$\frac{}{\Delta \vdash_{\emptyset} [r := e]}$$

(CAS)

$$\frac{}{\Delta \vdash_{\{lab\}} [r := \mathtt{pcas}(e_{\mathsf{l}}, e_{\mathsf{o}}, e_{\mathsf{n}}, \mathsf{mid}.lab)]}$$

(CHKPT)

$$\frac{\Delta \vdash_{\mathsf{RO}} \overrightarrow{s}}{\Delta \vdash_{\{lab\}} [r := \mathtt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)]}$$

(SEQ)

$$\frac{labs_{\mathsf{l}} \cap labs_{\mathsf{r}} = \emptyset \qquad \Delta \vdash_{labs_{\mathsf{l}}} \overrightarrow{s_{\mathsf{l}}} \quad \Delta \vdash_{labs_{\mathsf{r}}} \overrightarrow{s_{\mathsf{r}}}}{\Delta \vdash_{labs_{\mathsf{l}} \uplus labs_{\mathsf{r}}} \overrightarrow{s_{\mathsf{l}}} +\!\!+ \overrightarrow{s_{\mathsf{r}}}}$$

(IF-THEN-ELSE)

$$\frac{\Delta \vdash_{labs_{\mathsf{t}}} \overrightarrow{s_{\mathsf{t}}} \quad \Delta \vdash_{labs_{\mathsf{f}}} \overrightarrow{s_{\mathsf{f}}}}{\Delta \vdash_{labs_{\mathsf{t}} \cup labs_{\mathsf{f}}} [\mathtt{if}\ (e)\ \overrightarrow{s_{\mathsf{t}}}\ \overrightarrow{s_{\mathsf{f}}}]}$$

(LOOP-SIMPLE)

$$\frac{\Delta \vdash_{labs} \overrightarrow{s}}{\Delta \vdash_{labs} [\mathtt{loop}\ \_\ ()\ \overrightarrow{s}]}$$

(LOOP)

$$\frac{\Delta \vdash_{labs} \overrightarrow{s} \qquad lab \notin labs}{\Delta \vdash_{\{lab\} \uplus labs} [\mathtt{loop}\ r\ e\ ((r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})]}$$

(CONTINUE)

$$\frac{}{\Delta \vdash_{\emptyset} [\mathtt{continue}\ e]}$$

(BREAK)

$$\frac{}{\Delta \vdash_{\emptyset} [\mathtt{break}]}$$

(CALL)

$$\frac{\Delta(f) = \mathsf{RW}}{\Delta \vdash_{\{lab\}} [r := f(\overrightarrow{e} +\!\!+ \{\mathsf{mid}.lab\})]}$$

(RETURN)

$$\frac{}{\Delta \vdash_{\emptyset} [\mathtt{return}\ e]}$$

$\boxed{\Delta \vdash_{\mathsf{RO}} \overrightarrow{s}}$

(EMPTY)

$$\frac{}{\Delta \vdash_{\mathsf{RO}} []}$$

(ASSIGN)

$$\frac{}{\Delta \vdash_{\mathsf{RO}} [r := e]}$$

(LOAD)

$$\frac{}{\Delta \vdash_{\mathsf{RO}} [r := \mathtt{pload}(e)]}$$

(ALLOC)

$$\frac{}{\Delta \vdash_{\mathsf{RO}} [r := \mathtt{palloc}(e)]}$$

(LOOP)

$$\frac{\Delta \vdash_{\mathsf{RO}} \overrightarrow{s}}{\Delta \vdash_{\mathsf{RO}} [\mathtt{loop}\ r\ e\ \overrightarrow{s}]}$$

(CONTINUE)

$$\frac{}{\Delta \vdash_{\mathsf{RO}} [\mathtt{continue}\ e]}$$

(BREAK)

$$\frac{}{\Delta \vdash_{\mathsf{RO}} [\mathtt{break}]}$$

(CALL)

$$\frac{\Delta(f) = \mathsf{RO}}{\Delta \vdash_{\mathsf{RO}} [r := f(\overrightarrow{e})]}$$

(RETURN)

$$\frac{}{\Delta \vdash_{\mathsf{RO}} [\mathtt{return}\ e]}$$

(SEQ)

$$\frac{\Delta \vdash_{\mathsf{RO}} \overrightarrow{s_{\mathsf{l}}} \quad \Delta \vdash_{\mathsf{RO}} \overrightarrow{s_{\mathsf{r}}}}{\Delta \vdash_{\mathsf{RO}} \overrightarrow{s_{\mathsf{l}}} +\!\!+ \overrightarrow{s_{\mathsf{r}}}}$$

(IF-THEN-ELSE)

$$\frac{\Delta \vdash_{\mathsf{RO}} \overrightarrow{s_{\mathsf{t}}} \quad \Delta \vdash_{\mathsf{RO}} \overrightarrow{s_{\mathsf{f}}}}{\Delta \vdash_{\mathsf{RO}} [\mathtt{if}\ (e)\ \overrightarrow{s_{\mathsf{t}}}\ \overrightarrow{s_{\mathsf{f}}}]}$$

Figure B.14: Type system.

Let $mid_{\mathsf{pfx}} = ts.\mathsf{regs}(\mathsf{mid})$ *and* $mids = \mu(mid_{\mathsf{pfx}}, labs)$.

We have:

$$
\begin{aligned}
& \Delta \vdash_{labs} \overrightarrow{s} \\
\Longrightarrow\ & \overrightarrow{s}, [], ts, mmts \xrightarrow{tr}_{\delta}{}^{*} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \\
\Longrightarrow\ & \overrightarrow{s}, [], ts, mmts|_{mids} \xrightarrow{tr}_{\delta}{}^{*} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega|_{mids} \\
& \wedge\ mmts|_{mids^{\mathsf{c}}} = mmts_\omega|_{mids^{\mathsf{c}}} \\
& \wedge\ \forall mmts_\alpha, \overrightarrow{s}, [], ts, mmts|_{mids} \uplus mmts_\alpha|_{mids^{\mathsf{c}}} \xrightarrow{tr}_{\delta}{}^{*} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega|_{mids} \uplus mmts_\alpha|_{mids^{\mathsf{c}}} .
\end{aligned}
$$

Proof Sketch. We define the type system in such a way that typed statements only access those mementos referenced by $labs$. Such an intention is formalized in this lemma. □

B.6.3 Control Construct Cases

Lemma B.6.8 (Sequence Cases). *For all* $\delta, tr, \overrightarrow{s_{\mathsf{l}}}, \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{s_{\mathsf{c}}}, \overrightarrow{s_\omega}, \overrightarrow{c}, \overrightarrow{c_{\mathsf{c}}}, \overrightarrow{c_\omega}, ts, ts_\omega, mmts, mmts_\omega, e$,
We have:

$$
\begin{aligned}
& \overrightarrow{s_{\mathsf{l}}} +\!\!+ \overrightarrow{s_{\mathsf{r}}}, [], ts, mmts \xrightarrow{tr}_{\delta}{}^{*} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \\
\Longrightarrow\ & \textbf{SEQ-LEFT-ONGOING} : (\exists \overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}. \\
& \quad \overrightarrow{s_{\mathsf{l}}}, [], ts, mmts \xrightarrow{tr}_{\delta}{}^{*} \overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}, ts_\omega, mmts_\omega \ \wedge\ (\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) = (\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}) +\!\!+ \overrightarrow{s_{\mathsf{r}}} \ \wedge\ (\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}) \neq ([], [])) \\
\vee\ & \textbf{SEQ-LEFT-DONE} : (\exists tr_1, tr_2, ts_1, mmts_1. \ tr = tr_1 +\!\!+ tr_2 \ \wedge \\
& \quad \overrightarrow{s_{\mathsf{l}}}, [], ts, mmts \xrightarrow{tr_1}_{\delta}{}^{*} [], [], ts_1, mmts_1 \ \wedge\ \overrightarrow{s_{\mathsf{r}}}, [], ts_1, mmts_1 \xrightarrow{tr_2}_{\delta}{}^{*} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega) .
\end{aligned}
$$

Proof Sketch. The lemma says that an execution of a sequential composition of statements, $\overrightarrow{s_{\mathsf{l}}}$ and $\overrightarrow{s_{\mathsf{r}}}$, either does not finish the execution of $\overrightarrow{s_{\mathsf{l}}}$ or finishes $\overrightarrow{s_{\mathsf{l}}}$ and continues on $\overrightarrow{s_{\mathsf{r}}}$. Straightforward induction on the structure of $\overrightarrow{s_{\mathsf{l}}}$. □

Definition B.6.9 (Thread Transition with Base Continuations). *We define thread transition with base continuations,* $\overrightarrow{c_\alpha}$*, as follows:*

$$
\begin{aligned}
& \overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1 \xrightarrow{tr/\overrightarrow{c_\alpha}}_{\delta} \overrightarrow{s_2}, \overrightarrow{c_2}, ts_2, mmts_2 \\
\triangleq\ & \overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1 \xrightarrow{tr}_{\delta} \overrightarrow{s_2}, \overrightarrow{c_2}, ts_2, mmts_2 \ \wedge\ (\exists \overrightarrow{c_{\mathsf{pfx}}}. \ \overrightarrow{c_2} = \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ \overrightarrow{c_\alpha}) .
\end{aligned}
$$

As a special case, we have $\xrightarrow{tr}_{\delta} \ = \ \xrightarrow{tr/[]}_{\delta}$.

Lemma B.6.10 (Loop Cases). *For any* $tr, \overrightarrow{s}, ts, mmts, \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega, \sigma, r$,
Let $\overrightarrow{c} = [\texttt{loopCont}(\sigma, r, \overrightarrow{s}, [])]$.
We have:

$$
\begin{aligned}
& \overrightarrow{s}, \overrightarrow{c}, ts, mmts \xrightarrow{tr}_{\delta}{}^{*} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \\
\Longrightarrow\ & \textbf{LOOP-ONGOING} : (\overrightarrow{s}, \overrightarrow{c}, ts, mmts \xrightarrow{tr/\overrightarrow{c}}_{\delta}{}^{*} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega) \\
\vee\ & \textbf{LOOP-DONE} : (\overrightarrow{s}, \overrightarrow{c}, ts, mmts \xrightarrow{tr/\overrightarrow{c}}_{\delta}{}^{*} (\texttt{break}) :: \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c}, ts_{\mathsf{r}}, mmts_\omega \ \wedge \\
& \quad \overrightarrow{s_\omega} = [] \ \wedge\ \overrightarrow{c_\omega} = [] \ \wedge\ ts_\omega = \langle \mathsf{regs} : \sigma; \mathsf{time} : ts_{\mathsf{r}}.\mathsf{time} \rangle) .
\end{aligned}
$$

Proof Sketch. Straightforward by induction on the transitions. □

Lemma B.6.11 (First Loop Iteration). *For any* $tr, \overrightarrow{s}, ts, mmts, \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega, \sigma, r$,
Let $\overrightarrow{c} = [\texttt{loopCont}(\sigma, r, \overrightarrow{s}, [])]$.
We have:

$$
\begin{aligned}
& \overrightarrow{s}, \overrightarrow{c}, ts, mmts \xrightarrow{tr/\overrightarrow{c}}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \\
\implies & \textbf{FIRST-ONGOING} : (\exists \overrightarrow{c_{\mathsf{pfx}}}. \overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ \overrightarrow{c} \ \wedge \ \overrightarrow{s}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}, ts_\omega, mmts_\omega) \\
\vee \ & \textbf{FIRST-DONE} : (\exists \overrightarrow{s_1}, ts_1, mmts_1, tr_1, tr_2, e. \\
& \quad tr = tr_1 +\!\!+ tr_2 \ \wedge \\
& \quad \overrightarrow{s}, [], ts, mmts \xrightarrow{tr_1}{}^{*}_{\delta} (\texttt{continue}\ e) :: \overrightarrow{s_1}, [], ts_1, mmts_1 \ \wedge \\
& \quad (\texttt{continue}\ e) :: \overrightarrow{s_1}, \overrightarrow{c}, ts_1, mmts_1 \xrightarrow{tr_2/\overrightarrow{c}}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega) \ .
\end{aligned}
$$

Proof Sketch. Straightforward by induction on the transitions. □

Lemma B.6.12 (Last Loop Iteration). *For any* $tr, \overrightarrow{s}, ts, mmts, \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega, \sigma, r$,
Let $\overrightarrow{c} = [\texttt{loopCont}(\sigma, r, \overrightarrow{s}, [])]$.
We have:

$$
\begin{aligned}
& \overrightarrow{s}, \overrightarrow{c}, ts, mmts \xrightarrow{tr/\overrightarrow{c}}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \\
\implies & \exists ts_1, mmts_1, \overrightarrow{c_{\mathsf{pfx}}}, tr_1, tr_2. \\
& \quad tr = tr_1 +\!\!+ tr_2 \ \wedge \\
& \quad (\textbf{LAST-FIRST} : ((ts, mmts) = (ts_1, mmts_1) \ \wedge \ tr_1 = []) \vee \\
& \quad \ \textbf{LAST-CONT} : (\exists e, \overrightarrow{s_{\mathsf{r}}}, ts_{\mathsf{r}}. \ \overrightarrow{s}, \overrightarrow{c}, ts, mmts \xrightarrow{tr_1/\overrightarrow{c}}{}^{*}_{\delta} (\texttt{continue}\ e) :: \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c}, ts_{\mathsf{r}}, mmts_1 \ \wedge \\
& \quad \ ts_1 = \langle \mathsf{regs} : \sigma[r \mapsto ts_{\mathsf{r}}.\mathsf{regs}(e)]; \mathsf{time} : ts_{\mathsf{r}}.\mathsf{time} \rangle)) \ \wedge \\
& \quad \overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ \overrightarrow{c} \ \wedge \ \overrightarrow{s}, [], ts_1, mmts_1 \xrightarrow{tr_2}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}, ts_\omega, mmts_\omega \ .
\end{aligned}
$$

Proof Sketch. Straightforward by induction on the transitions. □

Lemma B.6.13 (Function and Checkpoint Cases). *For any* $\delta, tr, \overrightarrow{s}, \overrightarrow{s_\omega}, \overrightarrow{c}, \overrightarrow{c_\omega}, ts, ts_\omega, mmts, mmts_\omega$,
Let $\overrightarrow{c} = [c_{\mathsf{hd}}]$.
We have:

$$
\begin{aligned}
& ((\exists \sigma, r, mid. \ c_{\mathsf{hd}} = \texttt{chkptCont}(\sigma, r, [], mid)) \ \vee \ c_{\mathsf{hd}} = \texttt{fnCont}(\sigma, r, [])) \\
\implies & \overrightarrow{s}, \overrightarrow{c}, ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \\
\implies & \textbf{CALL-ONGOING} : (\exists \overrightarrow{c_{\mathsf{pfx}}}. \ \overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ \overrightarrow{c} \ \wedge \ \overrightarrow{s}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}, ts_\omega, mmts_\omega) \ \vee \\
& \quad \textbf{CALL-DONE} : (\exists \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c_{\mathsf{r}}}, ts_{\mathsf{r}}, mmts_{\mathsf{r}}, e. \\
& \quad \ \overrightarrow{s}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} (\texttt{return}\ e) :: \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c_{\mathsf{r}}}, ts_{\mathsf{r}}, mmts_{\mathsf{r}} \ \wedge \\
& \quad \ (\texttt{return}\ e) :: \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c_{\mathsf{r}}} +\!\!+ \overrightarrow{c}, ts_{\mathsf{r}}, mmts_{\mathsf{r}} \xrightarrow{[]}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \ \wedge \\
& \quad \ \overrightarrow{s_\omega} = [] \ \wedge \ \overrightarrow{c_\omega} = []) \ .
\end{aligned}
$$

Proof Sketch. Straightforward by induction on the transitions. □

$$\boxed{tr_1 \sim tr_2} \qquad \frac{}{[] \sim []}\ \textsc{(refine-empty)} \qquad \frac{tr_1 \sim tr_2}{tr_1 +\!\!+ [ev] \sim tr_2 +\!\!+ [ev]}\ \textsc{(refine-both)} \qquad \frac{tr_1 \sim tr_2}{tr_1 \sim tr_2 +\!\!+ [\mathsf{R}(l, v)]}\ \textsc{(refine-read)}$$

Figure B.15: Refinement rule of two traces.

B.6.4 Semantics and Type System Properties

Lemma B.6.14 (Monotonically Increasing Time). *For any* $\delta, \overrightarrow{s}, tr, \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts, ts_\omega, mmts, mmts_\omega$, *We have:*

$$\begin{aligned} & \overrightarrow{s}, \overrightarrow{c}, ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \\ \Longrightarrow\ & ts.\mathsf{time} \leq ts_\omega.\mathsf{time}\ . \end{aligned}$$

Proof Sketch. Straightforward by induction on the transitions. □

Lemma B.6.15 (Function Environment Lookup). *For any* $\delta, \Delta, labs, \overrightarrow{s}$, *We have:*

$$\begin{aligned} & \vdash \delta : \Delta \\ \Longrightarrow\ & f \in dom(\Delta) \\ \Longrightarrow\ & \exists \overrightarrow{prms}, \overrightarrow{s_f}.\ \delta(f) = (\overrightarrow{prms}, \overrightarrow{s_f})\ . \end{aligned}$$

Proof Sketch. Straightforward by induction on the derivation of $\vdash \delta : \Delta$. □

Definition B.6.16 (Stop). *We define:*

$$\begin{aligned} \mathsf{STOP}(\overrightarrow{s}, \overrightarrow{c}) \triangleq\ & (\overrightarrow{s} = [] \wedge \overrightarrow{c} = []) \\ \vee\ & (\exists \overrightarrow{s_{\mathsf{rem}}}.\ \overrightarrow{s} = (\texttt{break}) :: \overrightarrow{s_{\mathsf{rem}}} \wedge \overrightarrow{c} = []) \\ \vee\ & (\exists \overrightarrow{s_{\mathsf{rem}}}, e.\ \overrightarrow{s} = (\texttt{continue}\ e) :: \overrightarrow{s_{\mathsf{rem}}} \wedge \overrightarrow{c} = []) \\ \vee\ & (\exists \overrightarrow{s_{\mathsf{rem}}}, e.\ \overrightarrow{s} = (\texttt{return}\ e) :: \overrightarrow{s_{\mathsf{rem}}} \wedge \mathsf{Loops}(\overrightarrow{c}))\ . \end{aligned}$$

Lemma B.6.17 (Stop means No Step). *For any* $tr, \overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1, \overrightarrow{s_2}, \overrightarrow{c_2}, ts_2, mmts_2$, *We have:*

$$\begin{aligned} & \mathsf{STOP}(\overrightarrow{s_1}, \overrightarrow{c_1}) \\ \Longrightarrow\ & \overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1 \rightarrow_\delta \overrightarrow{s_2}, \overrightarrow{c_2}, ts_2, mmts_2 \\ \Longrightarrow\ & False\ . \end{aligned}$$

Proof Sketch. Straightforward from the definition. □

B.6.5 Proof of the Detectability Theorem

Definition B.6.18 (Behavior). *Let* p *be a program. Let* $B^{\lightning}(p) \triangleq \{tr \mid \exists M.\ init(p) \xrightarrow{tr}{}^{*}_{p} M\}$ *be the set of* behaviors *of* p. *Let* $B(p) \triangleq \{tr \mid \exists M.\ init(p) \overset{tr}{\Rightarrow}{}^{*}_{p} M\}$ *be the set of* crash-free behaviors *of* p, *where* $\overset{tr}{\Rightarrow}_p$ *is the relation of* **machine-step**.

Definition B.6.19 (Trace Refinement). *Trace tr_1 refines tr_2, denoted by $tr_1 \sim tr_2$, if we can reach tr_1 from tr_2 by only removing read events. Formally, trace refinement is defined in Fig. B.15.*

Lemma B.6.20 (Crash-Free Interleaving). *Let $p = [\delta]\ \vec{s_1} \mid\mid \ldots \mid\mid \vec{s_n}$ be a program and b be a behavior. Then $b \in B(p)$ if and only if:*

$$
\begin{aligned}
& \exists tr, \mathsf{mem}, \{tr_i\}_i, \{\overrightarrow{s_{i,\omega}}\}_i, \{\overrightarrow{c_{i,\omega}}\}_i, \{ts_{i,\omega}\}_i, \{mmts_{i,\omega}\}_i. \\
& \quad \mathsf{mem}_{\mathsf{init}} \xrightarrow{tr}{}^{*} \mathsf{mem} \\
\wedge\ & tr \text{ is an interleaving of } tr_1, \cdots, tr_n \\
\wedge\ & \forall i.\ \vec{s_i}, [], ts_{\mathsf{init}}, mmts_{\mathsf{init}} \xrightarrow{tr_i}{}^{*}_{\delta} \overrightarrow{s_{i,\omega}}, \overrightarrow{c_{i,\omega}}, ts_{i,\omega}, mmts_{i,\omega} \\
\wedge\ & b \sim tr\ .
\end{aligned}
$$

Proof Sketch. Essentially, this lemma holds because thread transitions are communicating with the other threads and memory only via traces. □

Lemma B.6.21 (Interleaving). *Let $p = [\delta]\ \vec{s_1} \mid\mid \ldots \mid\mid \vec{s_n}$ be a program and b be a behavior. Then $b \in B^{\lightning}(p)$ if and only if:*

$$
\begin{aligned}
& \exists tr, \mathsf{mem}, \{tr_i\}_i, \{\overrightarrow{s_{i,\omega}}\}_i, \{\overrightarrow{c_{i,\omega}}\}_i, \{ts_{i,\omega}\}_i, \{mmts_{i,\omega}\}_i. \\
& \quad \mathsf{mem}_{\mathsf{init}} \xrightarrow{tr}{}^{*} \mathsf{mem} \\
\wedge\ & tr \text{ is an interleaving of } tr_1, \cdots, tr_n \\
\wedge\ & \forall i.\ \vec{s_i}, [], ts_{\mathsf{init}}, mmts_{\mathsf{init}} \xrightarrow{tr_i}{}^{\lightning *}_{\delta} \overrightarrow{s_{i,\omega}}, \overrightarrow{c_{i,\omega}}, ts_{i,\omega}, mmts_{i,\omega} \\
\wedge\ & b \sim tr\ ,
\end{aligned}
$$

where $\xrightarrow{tr}{}^{\lightning}_{\delta}$ is defined as the union of $\xrightarrow{tr}_{\delta}$ and the thread crash step that initializes s, c, and ts as described in **machine-crash**.

Proof Sketch. Essentially the same with Theorem B.6.20. □

Definition B.6.22 (Deterministic Replay). *For a function environment $\delta \in \mathrm{Env}$ and a list of statements $\vec{s} \in \overrightarrow{\mathrm{Stmt}}$, $\vec{s}$ is deterministically replayed for δ, denoted by $\mathsf{DR}(\delta, \vec{s})$, if the following holds:*

$$
\begin{aligned}
& \forall tr, \underline{tr}, \overrightarrow{s_\omega}, \underline{\overrightarrow{s_\omega}}, \overrightarrow{c_\omega}, \underline{\overrightarrow{c_\omega}}, ts, ts_\omega, \underline{ts_\omega}, mmts, mmts_\omega, \underline{mmts_\omega}. \\
& \vec{s}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \longrightarrow \vec{s}, [], ts, mmts_\omega \xrightarrow{tr}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega} \longrightarrow \\
& \exists tr_x, \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x.\ \vec{s}, [], ts, mmts \xrightarrow{tr_x}{}^{*}_{\delta} \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x, \underline{mmts_\omega} \wedge tr_x \sim tr +\!\!+ \underline{tr} \wedge \\
& (\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_x} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_x} \wedge ts_\omega = ts_x \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}) \wedge \\
& (\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_x} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_x} \wedge \underline{ts_\omega} = ts_x)\ .
\end{aligned}
$$

Lemma B.6.23 (Determistic Replay of Read-Write Statements). *For any $\delta, \Delta, labs, \vec{s}$, if we have $\vdash \delta : \Delta$ and $\Delta \vdash_{labs} \vec{s}$, then $\mathsf{DR}(\delta, \vec{s})$.*

We defer its proof to §B.6.6.

Theorem B.6.24 (Detectability, Restatement of Theorem 3.3.1). *Given a program p, if $\vdash p$ holds, then $B^{\lightning}(p) \subseteq B(p)$.*

Proof. Suppose $b \in B^{\lightning}(p)$. From Theorem B.6.21, we have:

$$\begin{aligned}
&\exists tr, \mathsf{mem}, \{tr_i\}_i, \{\overrightarrow{s_{i,\omega}}\}_i, \{\overrightarrow{c_{i,\omega}}\}_i, \{ts_{i,\omega}\}_i, \{mmts_{i,\omega}\}_i.\\
\wedge\ & \mathsf{mem}_{\mathsf{init}} \xrightarrow{tr}{}^{*} \mathsf{mem}\\
\wedge\ & tr \text{ is an interleaving of } tr_1, \cdots, tr_n\\
\wedge\ & \forall i.\ \overrightarrow{s_i}, [], ts_{\mathsf{init}}, mmts_{\mathsf{init}} \xrightarrow{tr_i}{}^{\lightning *}_{\delta} \overrightarrow{s_{i,\omega}}, \overrightarrow{c_{i,\omega}}, ts_{i,\omega}, mmts_{i,\omega}\\
\wedge\ & b \sim tr\ .
\end{aligned}$$

Pick any i whose execution, $\overrightarrow{s_i}, [], ts_{\mathsf{init}}, mmts_{\mathsf{init}} \xrightarrow{tr_i}{}^{\lightning *}_{\delta} \overrightarrow{s_{i,\omega}}, \overrightarrow{c_{i,\omega}}, ts_{i,\omega}, mmts_{i,\omega}$, involves crash steps. We apply Theorem B.6.23 to obtain a transition, say $\overrightarrow{s_i}, [], ts_{\mathsf{init}}, mmts_{\mathsf{init}} \xrightarrow{tr'_i}{}^{\lightning *}_{\delta} \overrightarrow{s_{i,\omega'}}, \overrightarrow{c_{i,\omega'}}, ts_{i,\omega'}, mmts_{i,\omega'}$, with fewer crash steps and $tr'_i \sim tr$. By inductively performing these steps, we obtain:

$$\begin{aligned}
&\exists \{tr'_i\}_i,\\
&\forall i.\ \overrightarrow{s_i}, [], ts_{\mathsf{init}}, mmts_{\mathsf{init}} \xrightarrow{tr'_i}{}^{*}_{\delta} \overrightarrow{s_{i,\omega}}, \overrightarrow{c_{i,\omega}}, ts_{i,\omega}, mmts_{i,\omega}\\
\wedge\ & \forall i.\ tr'_i \sim tr_i\ .
\end{aligned}$$

Then we interleave $\{tr'_i\}$ to tr' in such a say that $tr' \sim tr$. Since memory transition is closed under trace refinement, we have $\mathsf{mem}_{\mathsf{init}} \xrightarrow{tr'}{}^{*} \mathsf{mem}$. From b has no read events, we also have $b \sim tr'$.

We conclude this proof by applying Theorem B.6.21 for tr' and $\{tr'_i\}$ to obtain $b \in B(p)$. □

B.6.6 Proof of the Deterministic Replay Lemma

Lemma B.6.25 (Read-Only Statements). *For any $\delta, \Delta, \overrightarrow{s}, tr, \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts, ts_\omega, mmts, mmts_\omega$, We have:*

$$\begin{aligned}
&\vdash \delta : \Delta\\
\Longrightarrow\ & \Delta \vdash_{\mathsf{RO}} \overrightarrow{s}\\
\Longrightarrow\ & \overrightarrow{s}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega\\
\Longrightarrow\ & [] \sim tr\ \wedge\ mmts = mmts_\omega\ .
\end{aligned}$$

Proof Sketch. Straightforward by induction on the derivation of $\Delta \vdash_{\mathsf{RO}} \overrightarrow{s}$. □

Now we prove Theorem B.6.23: deterministic replay of well-typed read-write statements. By induction on the derivation of $\vdash \delta : \Delta$, it is sufficient to prove the following lemma (with an additional premise on Δ):

Lemma B.6.26 (Determistic Replay of Read-Write Statements, Inductively). *For any $\delta, \Delta, labs, \overrightarrow{s}$, We have:*

$$\begin{aligned}
&\vdash \delta : \Delta\\
\Longrightarrow\ & \Delta \vdash_{labs} \overrightarrow{s}\\
\Longrightarrow\ & (\forall f, \overrightarrow{prms}, \overrightarrow{s_f}.\ \Delta(f) = \mathsf{RW} \longrightarrow \delta(f) = (\overrightarrow{prms}, \overrightarrow{s_f}) \longrightarrow \mathsf{DR}(\delta, \overrightarrow{s_f}))\\
\Longrightarrow\ & \mathsf{DR}(\delta, \overrightarrow{s})\ .
\end{aligned}$$

Proof. We prove by induction on the derivation of $\Delta \vdash_{labs} \overrightarrow{s}$.

- For the case that there are empty transitions:

  By assumption, we have:

  ▷ FNJ: $\vdash \delta : \Delta$

  ▷ FNDR: $\forall f, \overrightarrow{prms}, \overrightarrow{s_f}.\ \Delta(f) = \mathsf{RW} \longrightarrow \delta(f) = (\overrightarrow{prms}, \overrightarrow{s_f}) \longrightarrow \mathsf{DR}(\delta, \overrightarrow{s_f})$

  ▷ EX1: $\overrightarrow{s}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

  ▷ EX2: $\overrightarrow{s}, [], ts, mmts_\omega \xRightarrow{tr}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  This is for the case that either EX1 or EX2 is empty transitions.

  We aim to prove the following goals: $\exists tr_x, \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x.$

  (1) $\overrightarrow{s}, [], ts, mmts \xrightarrow{tr_x}{}^{*}_{\delta} \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x, \underline{mmts_\omega}$

  (2) $tr_x \sim tr +\!\!+ \underline{tr}$

  (3) $\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_x} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_x} \wedge ts_\omega = ts_x \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}$

  (4) $\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_x} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_x} \wedge \underline{ts_\omega} = ts_x$

  - EX1 is empty:

    We have $\overrightarrow{s}, [], ts, mmts = \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \wedge tr = []$.

    We prove the goals with $tr_x = \underline{tr}, \overrightarrow{s_x} = \underline{\overrightarrow{s_\omega}}, \overrightarrow{c_x} = \underline{\overrightarrow{c_\omega}}, ts_x = \underline{ts_\omega}$ as follows:

    (1) $\overrightarrow{s}, [], ts, mmts \xRightarrow{tr}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

    (2) $\underline{tr} \sim [] +\!\!+ \underline{tr}$

    (3) If $\mathsf{STOP}(\overrightarrow{s}, [])$, $\overrightarrow{s}$ should be [], `continue` , `break` or `return` . By Theorem B.6.17, we have: $\underline{\overrightarrow{s_\omega}} = \underline{\overrightarrow{s_\omega}} \wedge \underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_\omega}} \wedge ts_\omega = \underline{ts_\omega} \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr} = []$

    (4) $\underline{\overrightarrow{s_\omega}} = \underline{\overrightarrow{s_\omega}} \wedge \underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_\omega}} \wedge \underline{ts_\omega} = \underline{ts_\omega}$

  - EX2 is empty:

    We have $\overrightarrow{s}, [], ts, mmts_\omega = \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega} \wedge \underline{tr} = []$.

    We prove the goals with $tr_x = tr, \overrightarrow{s_x} = \overrightarrow{s_\omega}, \overrightarrow{c_x} = \overrightarrow{c_\omega}, ts_x = ts_\omega$ as follows:

    (1) $\overrightarrow{s}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

    (2) $tr \sim tr +\!\!+ []$

    (3) $\overrightarrow{s_\omega} = \overrightarrow{s_\omega} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_\omega} \wedge ts_\omega = ts_\omega \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim []$

    (4) If $\mathsf{STOP}(\overrightarrow{s}, [])$, $\overrightarrow{s}$ should be [], `continue` , `break` or `return` . By Theorem B.6.17, we have: $\underline{\overrightarrow{s_\omega}} = \overrightarrow{s_\omega} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_\omega} \wedge \underline{ts_\omega} = ts_\omega$

  For the rest cases, we assume both of EX1 and EX2 are not empty.

- (**EMPTY**, **BREAK**, **CONTINUE**, **RETURN**):

  By assumption, we have:

  ▷ FNJ: $\vdash \delta : \Delta$

  ▷ FNDR: $\forall f, \overrightarrow{prms}, \overrightarrow{s_f}.\ \Delta(f) = \mathsf{RW} \longrightarrow \delta(f) = (\overrightarrow{prms}, \overrightarrow{s_f}) \longrightarrow \mathsf{DR}(\delta, \overrightarrow{s_f})$

▷ EX1: $\overrightarrow{s}, [], ts, mmts \xrightarrow{tr}{}^{+}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

▷ EX2: $\overrightarrow{s}, [], ts, mmts_\omega \xrightarrow{tr}{}^{+}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

For these cases, we have $\mathsf{STOP}(\overrightarrow{s}, [])$. This contradicts EX1 and EX2 by Theorem B.6.17.

- (**ASSIGN**):

  By assumption, we have:

  ▷ FNJ: $\vdash \delta : \Delta$

  ▷ FNDR: $\forall f, \overrightarrow{prms}, \overrightarrow{s_f}.\ \Delta(f) = \mathsf{RW} \longrightarrow \delta(f) = (\overrightarrow{prms}, \overrightarrow{s_f}) \longrightarrow \mathsf{DR}(\delta, \overrightarrow{s_f})$

  ▷ EX1: $[r := e], [], ts, mmts \xrightarrow{tr}{}^{+}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

  ▷ EX2: $[r := e], [], ts, mmts_\omega \xrightarrow{tr}{}^{+}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  We aim to prove the following goals: $\exists tr_x, \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x.$

  (1) $[r := e], [], ts, mmts \xrightarrow{tr_x}{}^{*}_{\delta} \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x, \underline{mmts_\omega}$

  (2) $tr_x \sim tr +\!\!+ \underline{tr}$

  (3) $\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_x} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_x} \wedge ts_\omega = ts_x \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}$

  (4) $\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_x} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_x} \wedge \underline{ts_\omega} = ts_x$

  From EX1 and EX2, there exists $\overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1, \underline{\overrightarrow{s_1}}, \underline{\overrightarrow{c_1}}, \underline{ts_1}, \underline{mmts_1}$ such that:

  ▷ $[r := e], [], ts, mmts \rightarrow_\delta \overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1 \xrightarrow{tr_1}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

  ▷ $[r := e], [], ts, mmts_\omega \rightarrow_\delta \underline{\overrightarrow{s_1}}, \underline{\overrightarrow{c_1}}, \underline{ts_1}, \underline{mmts_1} \xrightarrow{tr_1}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  Since the only possible first steps are **ASSIGN**, we have:

  ▷ $[r := e], [], ts, mmts \xrightarrow{[]}_\delta [], [], ts_1, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

  ▷ $[r := e], [], ts, mmts_\omega \xrightarrow{[]}_\delta [], [], \underline{ts_1}, mmts_\omega \xrightarrow{tr}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  Let $v = ts.\mathsf{regs}(e)$. We have: $ts_1 = ts[\mathsf{regs} \mapsto ts.\mathsf{regs}[r \mapsto v]] = \underline{ts_1}$.

  From $\mathsf{STOP}([], [])$ and Theorem B.6.17, we have:

  ▷ $([], [], ts_1, mmts) = (\overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega)$ and $tr = []$

  ▷ $([], [], \underline{ts_1}, mmts_\omega) = (\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega})$ and $\underline{tr} = []$

  We prove the goals with $tr_x = [], \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = ts_1$ as follows:

  (1) $[r := e], [], ts, mmts \xrightarrow{[]}{}^{*}_{\delta} [], [], ts_1, mmts$

  (2) $[] \sim [] +\!\!+ []$

  (3) $[] = [] \wedge [] = [] \wedge ts_1 = ts_1 \wedge mmts = mmts \wedge [] \sim []$

  (4) $[] = [] \wedge [] = [] \wedge \underline{ts_1} = ts_1$

- (**IF-THEN-ELSE**):

  By assumption, we have:

  ▷ FNJ: $\vdash \delta : \Delta$

  ▷ FNDR: $\forall f, \overrightarrow{prms}, \overrightarrow{s_f}.\ \Delta(f) = \mathsf{RW} \longrightarrow \delta(f) = (\overrightarrow{prms}, \overrightarrow{s_f}) \longrightarrow \mathsf{DR}(\delta, \overrightarrow{s_f})$

  ▷ EX1: $[\texttt{if}\ (e)\ \overrightarrow{s_\mathsf{t}}\ \overrightarrow{s_\mathsf{f}}], [], ts, mmts \xrightarrow{tr}{}^{+}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

  ▷ EX2: $[\texttt{if}\ (e)\ \overrightarrow{s_\mathsf{t}}\ \overrightarrow{s_\mathsf{f}}], [], ts, mmts_\omega \xrightarrow{tr}{}^{+}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  For this case, we have:

  ▷ T: $\Delta \vdash_{labs_\mathsf{t}} \overrightarrow{s_\mathsf{t}}$

  ▷ F: $\Delta \vdash_{labs_\mathsf{f}} \overrightarrow{s_\mathsf{f}}$

  ▷ IH_t: $\forall tr, \underline{tr}, \overrightarrow{s_\omega}, \underline{\overrightarrow{s_\omega}}, \overrightarrow{c_\omega}, \underline{\overrightarrow{c_\omega}}, ts, ts_\omega, \underline{ts_\omega}, mmts, mmts_\omega, \underline{mmts_\omega}.$
  $\overrightarrow{s_\mathsf{t}}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \longrightarrow$
  $\overrightarrow{s_\mathsf{t}}, [], ts, mmts_\omega \xrightarrow{tr}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega} \longrightarrow$
  $\exists tr_\mathsf{ht}, \overrightarrow{s_\mathsf{ht}}, \overrightarrow{c_\mathsf{ht}}, ts_\mathsf{ht}.$
  $\overrightarrow{s_\mathsf{t}}, [], ts, mmts \xrightarrow{tr_\mathsf{ht}}{}^{*}_{\delta} \overrightarrow{s_\mathsf{ht}}, \overrightarrow{c_\mathsf{ht}}, ts_\mathsf{ht}, \underline{mmts_\omega} \wedge$
  $tr_\mathsf{ht} \sim tr +\!\!+ \underline{tr} \wedge$
  $(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_\mathsf{ht}} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_\mathsf{ht}} \wedge ts_\omega = ts_\mathsf{ht} \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}) \wedge$
  $(\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_\mathsf{ht}} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_\mathsf{ht}} \wedge \underline{ts_\omega} = ts_\mathsf{ht})$

  ▷ IH_f: $\forall tr, \underline{tr}, \overrightarrow{s_\omega}, \underline{\overrightarrow{s_\omega}}, \overrightarrow{c_\omega}, \underline{\overrightarrow{c_\omega}}, ts, ts_\omega, \underline{ts_\omega}, mmts, mmts_\omega, \underline{mmts_\omega}.$
  $\overrightarrow{s_\mathsf{f}}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \longrightarrow$
  $\overrightarrow{s_\mathsf{f}}, [], ts, mmts_\omega \xrightarrow{tr}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega} \longrightarrow$
  $\exists tr_\mathsf{hf}, \overrightarrow{s_\mathsf{hf}}, \overrightarrow{c_\mathsf{hf}}, ts_\mathsf{hf}.$
  $\overrightarrow{s_\mathsf{f}}, [], ts, mmts \xrightarrow{tr_\mathsf{hf}}{}^{*}_{\delta} \overrightarrow{s_\mathsf{hf}}, \overrightarrow{c_\mathsf{hf}}, ts_\mathsf{hf}, \underline{mmts_\omega} \wedge$
  $tr_\mathsf{hf} \sim tr +\!\!+ \underline{tr} \wedge$
  $(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_\mathsf{hf}} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_\mathsf{hf}} \wedge ts_\omega = ts_\mathsf{hf} \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}) \wedge$
  $(\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_\mathsf{hf}} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_\mathsf{hf}} \wedge \underline{ts_\omega} = ts_\mathsf{hf})$

  We aim to prove the following goals: $\exists tr_x, \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x.$

  (1) $[\texttt{if}\ (e)\ \overrightarrow{s_\mathsf{t}}\ \overrightarrow{s_\mathsf{f}}], [], ts, mmts \xrightarrow{tr_x}{}^{*}_{\delta} \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x, \underline{mmts_\omega}$

  (2) $tr_x \sim tr +\!\!+ \underline{tr}$

  (3) $\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_x} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_x} \wedge ts_\omega = ts_x \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}$

  (4) $\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_x} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_x} \wedge \underline{ts_\omega} = ts_x$

  From EX1 and EX2, there exists $\overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1, \underline{\overrightarrow{s_1}}, \underline{\overrightarrow{c_1}}, \underline{ts_1}, \underline{mmts_1}$ such that:

  ▷ $[\texttt{if}\ (e)\ \overrightarrow{s_\mathsf{t}}\ \overrightarrow{s_\mathsf{f}}], [], ts, mmts \rightarrow_\delta \overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1 \xrightarrow{tr_1}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

  ▷ $[\texttt{if}\ (e)\ \overrightarrow{s_\mathsf{t}}\ \overrightarrow{s_\mathsf{f}}], [], ts, mmts_\omega \rightarrow_\delta \underline{\overrightarrow{s_1}}, \underline{\overrightarrow{c_1}}, \underline{ts_1}, \underline{mmts_1} \xrightarrow{tr_1}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  The only possible first steps are **BRANCH**. Without loss of generality, we assume $ts.\mathsf{regs}(e) = \texttt{true}$. Then we have:

▷ $[\texttt{if}\ (e)\ \overrightarrow{s_\mathsf{t}}\ \overrightarrow{s_\mathsf{f}}], [], ts, mmts \xrightarrow{[]}_\delta \overrightarrow{s_\mathsf{t}}, [], ts, mmts \xrightarrow{tr}{}^*_\delta \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

▷ $[\texttt{if}\ (e)\ \overrightarrow{s_\mathsf{t}}\ \overrightarrow{s_\mathsf{f}}], [], ts, mmts_\omega \xrightarrow{[]}_\delta \overrightarrow{s_\mathsf{t}}, [], ts, mmts_\omega \xrightarrow{tr}{}^*_\delta \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

From IH_t, we have: $\exists tr_\mathsf{ht}, \overrightarrow{s_\mathsf{ht}}, \overrightarrow{c_\mathsf{ht}}, ts_\mathsf{ht}.$
$\overrightarrow{s_\mathsf{t}}, [], ts, mmts \xrightarrow{tr_\mathsf{ht}}{}^*_\delta \overrightarrow{s_\mathsf{ht}}, \overrightarrow{c_\mathsf{ht}}, ts_\mathsf{ht}, \underline{mmts_\omega} \wedge$
$tr_\mathsf{ht} \sim tr +\!\!+ \underline{tr} \wedge$
$(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_\mathsf{ht}} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_\mathsf{ht}} \wedge ts_\omega = ts_\mathsf{ht} \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}) \wedge$
$(\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_\mathsf{ht}} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_\mathsf{ht}} \wedge \underline{ts_\omega} = ts_\mathsf{ht})$

We prove the goals with $tr_x = tr_\mathsf{ht}, \overrightarrow{s_x} = \overrightarrow{s_\mathsf{ht}}, \overrightarrow{c_x} = \overrightarrow{c_\mathsf{ht}}, ts_x = ts_\mathsf{ht}$ as follows:

(1) $[\texttt{if}\ (e)\ \overrightarrow{s_\mathsf{t}}\ \overrightarrow{s_\mathsf{f}}], [], ts, mmts \xrightarrow{tr_\mathsf{ht}}{}^*_\delta \overrightarrow{s_\mathsf{ht}}, \overrightarrow{c_\omega}, \overrightarrow{c_\mathsf{ht}}, ts_\mathsf{ht}, mmts_\omega$

(2) $tr_\mathsf{ht} \sim tr +\!\!+ []$

(3) $\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_\mathsf{ht}} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_\mathsf{ht}} \wedge ts_\omega = ts_\mathsf{ht} \wedge mmts = \underline{mmts_\omega} \wedge [] \sim \underline{tr}$

(4) $\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_\mathsf{ht}} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_\mathsf{ht}} \wedge \underline{ts_\omega} = ts_\mathsf{ht}$

- (**CAS**):

  By assumption, we have:

  ▷ FNJ: $\vdash \delta : \Delta$

  ▷ FNDR: $\forall f, \overrightarrow{prms}, \overrightarrow{s_f}.\ \Delta(f) = \mathsf{RW} \longrightarrow \delta(f) = (\overrightarrow{prms}, \overrightarrow{s_f}) \longrightarrow \mathsf{DR}(\delta, \overrightarrow{s_f})$

  ▷ EX1: $[r := \texttt{pcas}(e_\mathsf{loc}, e_\mathsf{old}, e_\mathsf{new}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{tr}{}^+_\delta \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

  ▷ EX2: $[r := \texttt{pcas}(e_\mathsf{loc}, e_\mathsf{old}, e_\mathsf{new}, \mathsf{mid}.lab)], [], ts, mmts_\omega \xrightarrow{tr}{}^+_\delta \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  We aim to prove the following goals: $\exists tr_x, \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x.$

  (1) $[r := \texttt{pcas}(e_\mathsf{loc}, e_\mathsf{old}, e_\mathsf{new}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{tr_x}{}^*_\delta \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x, \underline{mmts_\omega}$

  (2) $tr_x \sim tr +\!\!+ \underline{tr}$

  (3) $\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_x} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_x} \wedge ts_\omega = ts_x \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}$

  (4) $\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_x} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_x} \wedge \underline{ts_\omega} = ts_x$

  From EX1 and EX2, there exists $\overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1, \underline{\overrightarrow{s_1}}, \underline{\overrightarrow{c_1}}, \underline{ts_1}, \underline{mmts_1}$ such that:

  ▷ EXB1: $[r := \texttt{pcas}(e_\mathsf{loc}, e_\mathsf{old}, e_\mathsf{new}, \mathsf{mid}.lab)], [], ts, mmts \rightarrow_\delta \overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1 \xrightarrow{tr_1}{}^*_\delta \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

  ▷ EXB2: $[r := \texttt{pcas}(e_\mathsf{loc}, e_\mathsf{old}, e_\mathsf{new}, \mathsf{mid}.lab)], [], ts, mmts_\omega \rightarrow_\delta \underline{\overrightarrow{s_1}}, \underline{\overrightarrow{c_1}}, \underline{ts_1}, \underline{mmts_1} \xrightarrow{tr_1}{}^*_\delta \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  Let $mid = ts.\mathsf{regs}(\mathsf{mid}.lab)$.

  We do a case analysis on EXB1's transition $[r := \texttt{pcas}(e_\mathsf{loc}, e_\mathsf{old}, e_\mathsf{new}, \mathsf{mid}.lab)], [], ts, mmts \rightarrow_\delta \overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1$: **CAS-SUCC**, **CAS-FAIL**, or **CAS-REPLAY**.

- (**CAS-SUCC**):

  For this sub-case, we have:

  $[r := \mathsf{pcas}(e_{\mathsf{loc}}, e_{\mathsf{old}}, e_{\mathsf{new}}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[ev]}_\delta [], [], ts_1, mmts_1$
  $\xrightarrow{tr_1}_\delta^* \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$,

  where

  $ev = \mathsf{U}(l, v_{\mathsf{old}}, v_{\mathsf{new}}), ts.\mathsf{regs}(e_{\mathsf{loc}}) = l, ts.\mathsf{regs}(e_{\mathsf{old}}) = v_{\mathsf{old}}, ts.\mathsf{regs}(e_{\mathsf{new}}) = v_{\mathsf{new}}, tr = [ev] +\!\!+ tr_1$.

  From $\mathsf{STOP}([], [])$ and Theorem B.6.17, we have:

  $([], [], ts_1, mmts_1) = (\overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega)$ and $tr = [ev]$.

  From $ts.\mathsf{time} < mmts_1[mid].\mathsf{time}$, EX2 can first take only a **CAS-REPLAY** step. So we have:

  $[r := \mathsf{pcas}(e_{\mathsf{loc}}, e_{\mathsf{old}}, e_{\mathsf{new}}, \mathsf{mid}.lab)], [], ts, mmts_1 \xrightarrow{[]}_\delta [], [], \underline{ts_1}, mmts_1$
  $\xrightarrow{tr}_\delta^* \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

  From $\mathsf{STOP}([], [])$ and Theorem B.6.17, we have:

  $([], [], \underline{ts_1}, mmts_1) = (\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega})$ and $\underline{tr} = []$.

  From $ts_1.\mathsf{regs}(r) = mmts_1[mid].\mathsf{val} = \underline{ts_1}.\mathsf{regs}(r)$ and $ts_1.\mathsf{time} = mmts_1[mid].\mathsf{time} = \underline{ts_1}.\mathsf{time}$, we have:

  $ts_1 = \underline{ts_1}$.

  We prove the goals with $tr_x = [ev], \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = ts_1$ as follows:

  (1) $[r := \mathsf{pcas}(e_{\mathsf{loc}}, e_{\mathsf{old}}, e_{\mathsf{new}}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[ev]}_\delta^* [], [], ts_1, mmts_1$

  (2) $[ev] \sim [ev] +\!\!+ []$

  (3) $[] = [] \wedge [] = [] \wedge ts_1 = ts_1 \wedge mmts_1 = mmts_1 \wedge [] \sim []$

  (4) $[] = [] \wedge [] = [] \wedge ts_1 = ts_1$

- (**CAS-FAIL**):

  For this sub-case, we have:

  $[r := \mathsf{pcas}(e_{\mathsf{loc}}, e_{\mathsf{old}}, e_{\mathsf{new}}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[ev]}_\delta [], [], ts_1, mmts_1 \xrightarrow{tr_1}_\delta^* \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$,
  where $ev = \mathsf{R}(l, v), ts.\mathsf{regs}(e_{\mathsf{loc}}) = l, ts_1.\mathsf{regs}(r) = (\mathtt{false}, v), tr = [ev] +\!\!+ tr_1$.

  From $\mathsf{STOP}([], [])$ and Theorem B.6.17, we have:

  $([], [], ts_1, mmts_1) = (\overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega)$ and $tr = [ev]$.

  From $ts.\mathsf{time} < mmts_1(mid).\mathsf{time}$, EX2 can first take only a **CAS-REPLAY** step. So we have:

  $[r := \mathsf{pcas}(e_{\mathsf{loc}}, e_{\mathsf{old}}, e_{\mathsf{new}}, \mathsf{mid}.lab)], [], ts, mmts_1 \xrightarrow{[]}_\delta [], [], \underline{ts_1}, mmts_1 \xrightarrow{tr}_\delta^* \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

  From $\mathsf{STOP}([], [])$ and Theorem B.6.17, we have:

  $([], [], \underline{ts_1}, mmts_1) = (\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega})$ and $\underline{tr} = []$.

From $ts_1.\mathsf{regs}(r) = mmts_1[mid].\mathsf{val} = \underline{ts_1}.\mathsf{regs}(r)$ and $ts_1.\mathsf{time} = mmts_1[mid].\mathsf{time} = \underline{ts_1}.\mathsf{time}$, we have:

$ts_1 = \underline{ts_1}$.

We prove the goals with $tr_x = [ev], \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = ts_1$ as follows:

(1) $[r := \mathsf{pcas}(e_{\mathsf{loc}}, e_{\mathsf{old}}, e_{\mathsf{new}}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[ev]}{}^{*}_{\delta} [], [], ts_1, mmts_1$

(2) $[ev] \sim [ev] +\!\!+ []$

(3) $[] = [] \wedge [] = [] \wedge ts_1 = ts_1 \wedge mmts_1 = mmts_1 \wedge [] \sim []$

(4) $[] = [] \wedge [] = [] \wedge ts_1 = ts_1$

- (**CAS-REPLAY**):

  For this sub-case, we have:
  $[r := \mathsf{pcas}(e_{\mathsf{loc}}, e_{\mathsf{old}}, e_{\mathsf{new}}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[]}_{\delta} [], [], ts_1, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$.

  From $\mathsf{STOP}([], [])$ and Theorem B.6.17, we have:
  $([], [], ts_1, mmts) = (\overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega)$ and $tr = []$.

  From $mmts_\omega = mmts$ and $ts.\mathsf{time} < mmts[mid].\mathsf{time}$, $\underline{tr}$ can first take only a **CAS-REPLAY** step. So we have:
  $[r := \mathsf{pcas}(e_{\mathsf{loc}}, e_{\mathsf{old}}, e_{\mathsf{new}}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[]}_{\delta} [], [], \underline{ts_1}, mmts \xrightarrow{tr}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

  From $\mathsf{STOP}([], [])$ and Theorem B.6.17, we have:
  $([], [], \underline{ts_1}, mmts) = (\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega})$ and $\underline{tr} = []$.

  From $ts_1.\mathsf{regs}(r) = mmts[mid].\mathsf{val} = \underline{ts_1}.\mathsf{regs}(r)$ and $ts_1.\mathsf{time} = mmts[mid].\mathsf{time} = \underline{ts_1}.\mathsf{time}$, we have:
  $ts_1 = \underline{ts_1}$.

  We prove the goals with $tr_x = [], \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = ts_1$ as follows:

  (1) $[r := \mathsf{pcas}(e_{\mathsf{loc}}, e_{\mathsf{old}}, e_{\mathsf{new}}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[]}{}^{*}_{\delta} [], [], ts_1, mmts$

  (2) $[] \sim [] +\!\!+ []$

  (3) $[] = [] \wedge [] = [] \wedge ts_1 = ts_1 \wedge mmts_1 = mmts_1 \wedge [] \sim []$

  (4) $[] = [] \wedge [] = [] \wedge ts_1 = ts_1$

- (**LET-CHKPT**):

  By assumption, we have:

  ▷ FNJ: $\vdash \delta : \Delta$

  ▷ FNDR: $\forall f, \overrightarrow{prms}, \overrightarrow{s_f}.\ \Delta(f) = \mathsf{RW} \longrightarrow \delta(f) = (\overrightarrow{prms}, \overrightarrow{s_f}) \longrightarrow \mathsf{DR}(\delta, \overrightarrow{s_f})$

▷ EX1: $[r \coloneqq \texttt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{tr}_{\delta}{}^{+} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

▷ EX2: $[r \coloneqq \texttt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)], [], ts, mmts_\omega \xrightarrow{tr}_{\delta}{}^{+} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

For this case, we have:

▷ RO: $\Delta \vdash_{\mathsf{RO}} \overrightarrow{s}$

We aim to prove the following goals: $\exists tr_x, \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x.$

(1) $[r \coloneqq \texttt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{tr_x}_{\delta}{}^{*} \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x, \underline{mmts_\omega}$

(2) $tr_x \sim tr \mathbin{+\!\!+} \underline{tr}$

(3) $\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_x} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_x} \wedge ts_\omega = ts_x \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}$

(4) $\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_x} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_x} \wedge \underline{ts_\omega} = ts_x$

From EX1 and EX2, there exists $\overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1, \underline{\overrightarrow{s_1}}, \underline{\overrightarrow{c_1}}, \underline{ts_1}, \underline{mmts_1}$ such that:

▷ EXB1: $[r \coloneqq \texttt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)], [], ts, mmts \rightarrow_{\delta} \overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1 \xrightarrow{tr_1}_{\delta}{}^{*} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

▷ EXB2: $[r \coloneqq \texttt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)], [], ts, mmts_\omega \rightarrow_{\delta} \underline{\overrightarrow{s_1}}, \underline{\overrightarrow{c_1}}, \underline{ts_1}, \underline{mmts_1} \xrightarrow{tr_1}_{\delta}{}^{*} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

Let $mid = ts.\mathsf{regs}(\mathsf{mid}.lab)$.

We do a case analysis on EXB1's transition $[r \coloneqq \texttt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)], [], ts, mmts \rightarrow_{\delta} \overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1$: **CHKPT-CALL** or **CHKPT-REPLAY**.

○ (**CHKPT-CALL**):

For this sub-case, we have:
$[r \coloneqq \texttt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[]}_{\delta} \overrightarrow{s}, \overrightarrow{c_1}, ts_1, mmts \xrightarrow{tr}_{\delta}{}^{*} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$.

We apply Theorem B.6.13 to the later transitions and do case analysis on the lemma's conclusion:

· (**CALL-ONGOING**):

For this sub-case, we have:
$\exists \overrightarrow{c_{\mathsf{pfx}}}. \overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} \mathbin{+\!\!+} \overrightarrow{c_1} \wedge \overrightarrow{s}, [], ts_1, mmts \xrightarrow{tr}_{\delta}{}^{*} \overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}, ts_\omega, mmts_\omega$.

From RO and Theorem B.6.25, we have:
$[] \sim tr \wedge mmts = mmts_\omega$.

We prove the goals with $tr_x = \underline{tr}, \overrightarrow{s_x} = \underline{\overrightarrow{s_\omega}}, \overrightarrow{c_x} = \underline{\overrightarrow{c_\omega}}, ts_x = \underline{ts_\omega}$ as follows:

(1) $[r \coloneqq \texttt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{tr}_{\delta}{}^{*} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

(2) From $[] \sim tr$, we have: $\underline{tr} \sim tr \mathbin{+\!\!+} \underline{tr}$

(3) From $\overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} \mathbin{+\!\!+} \overrightarrow{c_1} = \overrightarrow{c_{\mathsf{pfx}}} \mathbin{+\!\!+} [\texttt{chkptCont}(\sigma, r, [], mid)]$, we have: $\neg\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega})$

(4) $\underline{\overrightarrow{s_\omega}} = \underline{\overrightarrow{s_\omega}} \wedge \underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_\omega}} \wedge \underline{ts_\omega} = \underline{ts_\omega}$

· (**CALL-DONE**):

For this sub-case, we have:
$\exists \overrightarrow{s_\mathsf{r}}, \overrightarrow{c_\mathsf{r}}, ts_\mathsf{r}, mmts_\mathsf{r}, e.$
$\overrightarrow{s}, [], ts, mmts \xrightarrow{tr_\mathsf{r}}_{\delta}{}^{*} (\texttt{return}\ e) :: \overrightarrow{s_\mathsf{r}}, \overrightarrow{c_\mathsf{r}}, ts_\mathsf{r}, mmts_\mathsf{r} \wedge$

$(\texttt{return}\ e) :: \overrightarrow{s_\mathsf{r}}, \overrightarrow{c_\mathsf{r}} +\!\!+ \overrightarrow{c_1}, ts_\mathsf{r}, mmts_\mathsf{r} \xrightarrow{[]}_\delta \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \ \wedge$
$\overrightarrow{s_\omega} = [] \ \wedge \ \overrightarrow{c_\omega} = [] \ .$

From $ts.\mathsf{time} \leq ts_\mathsf{r}.\mathsf{time} < mmts_\omega[mid].\mathsf{time}$ from Theorem B.6.14, EX2 can first take only a **CHKPT-REPLAY** step. So we have:
$[r := \mathtt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)], [], ts, mmts_\omega \xrightarrow{[]}_\delta [], [], \underline{ts_1}, mmts_\omega \xrightarrow{tr}{}^*_\delta \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega} \ .$

From $ts_\omega.\mathsf{regs}(r) = mmts_\omega[mid].\mathsf{val} = \underline{ts_1}.\mathsf{regs}(r)$ and $ts_\omega.\mathsf{time} = mmts_\omega[mid].\mathsf{time} = \underline{ts_1}.\mathsf{time}$, we have:
$ts_\omega = \underline{ts_1} \ .$

From $\mathsf{STOP}([], [])$ and Theorem B.6.17, we have:
$([], [], ts_\omega, mmts_\omega) = (\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega})$ and $\underline{tr} = []$.

We prove the goals with $tr_x = tr, \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = ts_\omega$ as follows:

(1) $[r := \mathtt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{tr}{}^*_\delta [], [], ts_\omega, mmts_\omega$

(2) $tr \sim tr +\!\!+ []$

(3) $[] = [] \ \wedge \ [] = [] \ \wedge \ ts_\omega = ts_\omega \ \wedge \ mmts_\omega = mmts_\omega \ \wedge \ [] \sim []$

(4) $[] = [] \ \wedge \ [] = [] \ \wedge \ ts_\omega = ts_\omega$

- **(CHKPT-REPLAY)**:

  For this sub-case, we have:
  $[r := \mathtt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[]}_\delta [], [], ts_1, mmts \xrightarrow{tr}{}^*_\delta \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \ .$

  From $\mathsf{STOP}([], [])$ an Theorem B.6.17, we have:
  $([], [], ts_1, mmts) = (\overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega)$ and $tr = [] \ .$

  From $mmts_\omega = mmts$ and $ts.\mathsf{time} < mmts[mid].\mathsf{time}$, EX2 can first take only a **CHKPT-REPLAY** step. So we have:
  $[r := \mathtt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[]}_\delta [], [], \underline{ts_1}, mmts \xrightarrow{tr}{}^*_\delta \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega} \ .$

  We have:
  $ts_1 = \langle \mathsf{regs} \mapsto ts.\mathsf{regs}[r \mapsto mmts[mid].\mathsf{val}]; \mathsf{time} \mapsto mmts[mid].\mathsf{time} \rangle = \underline{ts_1}.$

  From $\mathsf{STOP}([], [])$ and Theorem B.6.17, we have:
  $([], [], ts_1, mmts) = (\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega})$ and $\underline{tr} = [].$

  We prove the goals with $tr_x = [], \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = ts_1$ as follows:

  (1) $[r := \mathtt{chkpt}(\overrightarrow{s}, \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[]}{}^*_\delta [], [], ts_1, mmts$

  (2) $[] \sim [] +\!\!+ []$

  (3) $[] = [] \ \wedge \ [] = [] \ \wedge \ ts_1 = ts_1 \ \wedge \ mmts_1 = mmts_1 \ \wedge \ [] \sim []$

  (4) $[] = [] \ \wedge \ [] = [] \ \wedge \ ts_1 = ts_1$

- (**SEQ**):

  By assumption, we have:

  ▷ FNJ: $\vdash \delta : \Delta$

  ▷ FNDR: $\forall f, \overrightarrow{prms}, \overrightarrow{s_f}.\ \Delta(f) = \mathsf{RW} \longrightarrow \delta(f) = (\overrightarrow{prms}, \overrightarrow{s_f}) \longrightarrow \mathsf{DR}(\delta, \overrightarrow{s_f})$

  ▷ EX1: $\overrightarrow{s_\mathsf{l}} +\!\!+ \overrightarrow{s_\mathsf{r}}, [], ts, mmts \xrightarrow{tr}{}^{+}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

  ▷ EX2: $\overrightarrow{s_\mathsf{l}} +\!\!+ \overrightarrow{s_\mathsf{r}}, [], ts, mmts_\omega \xrightarrow{tr}{}^{+}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  For this case, we have:

  ▷ L: $\Delta \vdash_{labs_\mathsf{l}} \overrightarrow{s_\mathsf{l}}$

  ▷ R: $\Delta \vdash_{labs_\mathsf{r}} \overrightarrow{s_\mathsf{r}}$

  ▷ $DISJ$: $labs_\mathsf{l} \cap labs_\mathsf{r} = \emptyset$

  ▷ IH_l: $\forall tr, \underline{tr}, \overrightarrow{s_\omega}, \underline{\overrightarrow{s_\omega}}, \overrightarrow{c_\omega}, \underline{\overrightarrow{c_\omega}}, ts, ts_\omega, \underline{ts_\omega}, mmts, mmts_\omega, \underline{mmts_\omega}.$
  $\overrightarrow{s_\mathsf{l}}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \longrightarrow$
  $\overrightarrow{s_\mathsf{l}}, [], ts, mmts_\omega \xrightarrow{tr}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega} \longrightarrow$
  $\exists tr_\mathsf{hl}, \overrightarrow{s_\mathsf{hl}}, \overrightarrow{c_\mathsf{hl}}, ts_\mathsf{hl}.$
  $\overrightarrow{s_\mathsf{l}}, [], ts, mmts \xrightarrow{tr_\mathsf{hl}}{}^{*}_{\delta} \overrightarrow{s_\mathsf{hl}}, \overrightarrow{c_\mathsf{hl}}, ts_\mathsf{hl}, \underline{mmts_\omega} \wedge$
  $tr_\mathsf{hl} \sim tr +\!\!+ \underline{tr} \wedge$
  $(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_\mathsf{hl}} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_\mathsf{hl}} \wedge ts_\omega = ts_\mathsf{hl} \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}) \wedge$
  $(\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_\mathsf{hl}} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_\mathsf{hl}} \wedge \underline{ts_\omega} = ts_\mathsf{hl})$

  ▷ IH_r: $\forall tr, \underline{tr}, \overrightarrow{s_\omega}, \underline{\overrightarrow{s_\omega}}, \overrightarrow{c_\omega}, \underline{\overrightarrow{c_\omega}}, ts, ts_\omega, \underline{ts_\omega}, mmts, mmts_\omega, \underline{mmts_\omega}.$
  $\overrightarrow{s_\mathsf{r}}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \longrightarrow$
  $\overrightarrow{s_\mathsf{r}}, [], ts, mmts_\omega \xrightarrow{tr}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega} \longrightarrow$
  $\exists tr_\mathsf{hr}, \overrightarrow{s_\mathsf{hr}}, \overrightarrow{c_\mathsf{hr}}, ts_\mathsf{hr}.$
  $\overrightarrow{s_\mathsf{r}}, [], ts, mmts \xrightarrow{tr_\mathsf{hr}}{}^{*}_{\delta} \overrightarrow{s_\mathsf{hr}}, \overrightarrow{c_\mathsf{hr}}, ts_\mathsf{hr}, \underline{mmts_\omega} \wedge$
  $tr_\mathsf{hr} \sim tr +\!\!+ \underline{tr} \wedge$
  $(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_\mathsf{hr}} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_\mathsf{hr}} \wedge ts_\omega = ts_\mathsf{hr} \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}) \wedge$
  $(\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_\mathsf{hr}} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_\mathsf{hr}} \wedge \underline{ts_\omega} = ts_\mathsf{hr})$

  We aim to prove the following goals: $\exists tr_x, \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x.$

  (1) $\overrightarrow{s_\mathsf{l}} +\!\!+ \overrightarrow{s_\mathsf{r}}, [], ts, mmts \xrightarrow{tr_x}{}^{*}_{\delta} \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x, \underline{mmts_\omega}$

  (2) $tr_x \sim tr +\!\!+ \underline{tr}$

  (3) $\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_x} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_x} \wedge ts_\omega = ts_x \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}$

  (4) $\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_x} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_x} \wedge \underline{ts_\omega} = ts_x$

  Let $mid_\mathsf{pfx} = ts.\mathsf{regs}(\mathsf{mid})$,

  $mids_\mathsf{l} = \mu(mid_\mathsf{pfx}, labs_\mathsf{l})$, and

$mids_{\mathsf{r}} = \mu(mid_{\mathsf{pfx}}, labs_{\mathsf{r}})$.

We apply Theorem B.6.8 both to EX1 and EX2 and do case analysis on the lemma's conclusion:

- (**SEQ-LEFT-DONE** for EX1, **SEQ-LEFT-DONE** for EX2):

  For this sub-case, we have:

  - ▷ $\exists tr_1, tr_2, ts_1, mmts_1.\ tr = tr_1 +\!\!+ tr_2\ \wedge$
    $\vec{s_{\mathsf{l}}}, [], ts, mmts \xrightarrow{tr_1}{}^{*}_{\delta} [], [], ts_1, mmts_1\ \wedge\ \vec{s_{\mathsf{r}}}, [], ts_1, mmts_1 \xrightarrow{tr_2}{}^{*}_{\delta} \vec{s_\omega}, \vec{c_\omega}, ts_\omega, mmts_\omega$
  - ▷ $\exists \underline{tr_1}, \underline{tr_2}, \underline{ts_1}, \underline{mmts_1}.\ \underline{tr} = \underline{tr_1} +\!\!+ \underline{tr_2}\ \wedge$
    $\vec{s_{\mathsf{l}}}, [], ts, mmts_\omega \xrightarrow{\underline{tr_1}}{}^{*}_{\delta} [], [], \underline{ts_1}, \underline{mmts_1}\ \wedge\ \vec{s_{\mathsf{r}}}, [], \underline{ts_1}, \underline{mmts_1} \xrightarrow{\underline{tr_2}}{}^{*}_{\delta} \underline{\vec{s_\omega}}, \underline{\vec{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  From Theorem B.6.7, we have:

  - ▷ $mmts_1|_{mids_{\mathsf{r}}{}^{\mathsf{c}}} = mmts_\omega|_{mids_{\mathsf{r}}{}^{\mathsf{c}}}$
  - ▷ $mmts_\omega|_{mids_{\mathsf{l}}{}^{\mathsf{c}}} = \underline{mmts_1}|_{mids_{\mathsf{l}}{}^{\mathsf{c}}}$
  - ▷ $\underline{mmts_1}|_{mids_{\mathsf{r}}{}^{\mathsf{c}}} = \underline{mmts_\omega}|_{mids_{\mathsf{r}}{}^{\mathsf{c}}}$
  - ▷ $\vec{s_{\mathsf{l}}}, [], ts, mmts|_{mids_{\mathsf{l}}} \xrightarrow{tr_1}{}^{*}_{\delta} [], [], ts_1, mmts_1|_{mids_{\mathsf{l}}}$
  - ▷ $\vec{s_{\mathsf{l}}}, [], ts, mmts_\omega|_{mids_{\mathsf{l}}} \xrightarrow{tr_1}{}^{*}_{\delta} [], [], \underline{ts_1}, \underline{mmts_1}|_{mids_{\mathsf{l}}}$

  From DISJ and $mmts_1|_{mids_{\mathsf{r}}{}^{\mathsf{c}}} = mmts_\omega|_{mids_{\mathsf{r}}{}^{\mathsf{c}}}$, we have:
  $mmts_1|_{mids_{\mathsf{l}}} = mmts_\omega|_{mids_{\mathsf{l}}}$.

  From IH_{l}, we have:
  $\exists tr_{\mathsf{hl}}, \vec{s_{\mathsf{hl}}}, \vec{c_{\mathsf{hl}}}, ts_{\mathsf{hl}}.$
  $\vec{s_{\mathsf{l}}}, [], ts, mmts|_{mids_{\mathsf{l}}} \xrightarrow{tr_{\mathsf{hl}}}{}^{*}_{\delta} \vec{s_{\mathsf{hl}}}, \vec{c_{\mathsf{hl}}}, ts_{\mathsf{hl}}, \underline{mmts_1}|_{mids_{\mathsf{l}}}\ \wedge$
  $tr_{\mathsf{hl}} \sim tr_1 +\!\!+ \underline{tr_1}\ \wedge$
  $(\mathsf{STOP}([], [])\ \longrightarrow\ [] = \vec{s_{\mathsf{hl}}}\ \wedge\ [] = \vec{c_{\mathsf{hl}}}\ \wedge\ ts_1 = ts_{\mathsf{hl}}\ \wedge\ mmts_1|_{mids_{\mathsf{l}}} = \underline{mmts_1}|_{mids_{\mathsf{l}}}\ \wedge\ [] \sim \underline{tr_1})\ \wedge$
  $(\mathsf{STOP}([], [])\ \longrightarrow\ [] = \vec{s_{\mathsf{hl}}}\ \wedge\ [] = \vec{c_{\mathsf{hl}}}\ \wedge\ \underline{ts_1} = ts_{\mathsf{hl}})$.

  From $\mathsf{STOP}([], [])$, we have:
  $[] = \vec{s_{\mathsf{hl}}}\ \wedge\ [] = \vec{c_{\mathsf{hl}}}\ \wedge\ ts_1 = \underline{ts_1} = ts_{\mathsf{hl}}\ \wedge\ mmts_1|_{mids_{\mathsf{l}}} = \underline{mmts_1}|_{mids_{\mathsf{l}}}\ \wedge\ [] \sim \underline{tr_1}$.

  From $mmts_\omega|_{mids_{\mathsf{l}}} = mmts_1|_{mids_{\mathsf{l}}} = \underline{mmts_1}|_{mids_{\mathsf{l}}}$ and $mmts_\omega|_{mids_{\mathsf{l}}{}^{\mathsf{c}}} = \underline{mmts_1}|_{mids_{\mathsf{l}}{}^{\mathsf{c}}}$, we have:
  $mmts_\omega = \underline{mmts_1}$.

  From IH_{r}, we have:
  $\exists tr_{\mathsf{hr}}, \vec{s_{\mathsf{hr}}}, \vec{c_{\mathsf{hr}}}, ts_{\mathsf{hr}}.$
  $\vec{s_{\mathsf{r}}}, [], ts_1, mmts_1 \xrightarrow{tr_{\mathsf{hr}}}{}^{*}_{\delta} \vec{s_{\mathsf{hr}}}, \vec{c_{\mathsf{hr}}}, ts_{\mathsf{hr}}, \underline{mmts_\omega}\ \wedge$
  $tr_{\mathsf{hr}} \sim tr_2 +\!\!+ \underline{tr_2}\ \wedge$
  $(\mathsf{STOP}(\vec{s_\omega}, \vec{c_\omega})\ \longrightarrow\ \vec{s_\omega} = \vec{s_{\mathsf{hr}}}\ \wedge\ \vec{c_\omega} = \vec{c_{\mathsf{hr}}}\ \wedge\ ts_\omega = ts_{\mathsf{hr}}\ \wedge\ mmts_\omega = \underline{mmts_\omega}\ \wedge\ [] \sim \underline{tr_2})\ \wedge$
  $(\mathsf{STOP}(\underline{\vec{s_\omega}}, \underline{\vec{c_\omega}})\ \longrightarrow\ \underline{\vec{s_\omega}} = \vec{s_{\mathsf{hr}}}\ \wedge\ \underline{\vec{c_\omega}} = \vec{c_{\mathsf{hr}}}\ \wedge\ \underline{ts_\omega} = ts_{\mathsf{hr}})$.

  We prove the goals with $tr_x = tr_1 +\!\!+ tr_{\mathsf{hr}}, \vec{s_x} = \vec{s_{\mathsf{hr}}}, \vec{c_x} = \vec{c_{\mathsf{hr}}}, ts_x = ts_{\mathsf{hr}}$ as follows:

(1) From Theorem B.6.4, we have:
$\overrightarrow{s_{\mathsf{l}}} +\!\!+ \overrightarrow{s_{\mathsf{r}}}, [], ts, mmts \xrightarrow{tr_1}{}^{*}_{\delta} \overrightarrow{s_{\mathsf{r}}}, [], ts_1, mmts_1 \xrightarrow{tr_{\mathsf{hr}}}{}^{*}_{\delta} \overrightarrow{s_{\mathsf{hr}}}, \overrightarrow{c_{\mathsf{hr}}}, ts_{\mathsf{hr}}, \underline{mmts_\omega}$

(2) From $tr_{\mathsf{hr}} \sim tr_2 +\!\!+ \underline{tr_2}$, we have: $tr_1 +\!\!+ tr_{\mathsf{hr}} \sim tr +\!\!+ \underline{tr_2}$.
From $[] \sim \underline{tr_1}$, we have: $tr_1 +\!\!+ tr_{\mathsf{hr}} \sim tr +\!\!+ \underline{tr}$

(3) From $[] \sim \underline{tr_1}$, we have: $(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_{\mathsf{hr}}} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{hr}}} \wedge ts_\omega = ts_{\mathsf{hr}} \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr})$

(4) $\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_{\mathsf{hr}}} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_{\mathsf{hr}}} \wedge \underline{ts_\omega} = ts_{\mathsf{hr}}$

- (**SEQ-LEFT-DONE**, **SEQ-LEFT-ONGOING**):

For this sub-case, we have:

- $\exists tr_1, tr_2, ts_1, mmts_1.\ tr = tr_1 +\!\!+ tr_2 \wedge$
  $\overrightarrow{s_{\mathsf{l}}}, [], ts, mmts \xrightarrow{tr_1}{}^{*}_{\delta} [], [], ts_1, mmts_1 \wedge \overrightarrow{s_{\mathsf{r}}}, [], ts_1, mmts_1 \xrightarrow{tr_2}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$
- $\exists \underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}.\ \overrightarrow{s_{\mathsf{l}}}, [], ts, mmts_\omega \xrightarrow{tr}{}^{*}_{\delta} \underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}, \underline{ts_\omega}, \underline{mmts_\omega} \wedge (\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) = (\underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}) +\!\!+ \overrightarrow{s_{\mathsf{r}}} \wedge (\underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}) \neq ([], [])$

From Theorem B.6.7, we have:

- $mmts_1|_{mids_{\mathsf{r}}{}^{\mathsf{c}}} = mmts_\omega|_{mids_{\mathsf{r}}{}^{\mathsf{c}}}$
- $mmts_\omega|_{mids_{\mathsf{l}}{}^{\mathsf{c}}} = \underline{mmts_\omega}|_{mids_{\mathsf{l}}{}^{\mathsf{c}}}$
- $\overrightarrow{s_{\mathsf{l}}}, [], ts, mmts|_{mids_{\mathsf{l}}} \xrightarrow{tr_1}{}^{*}_{\delta} [], [], ts_1, mmts_1|_{mids_{\mathsf{l}}}$
- $\overrightarrow{s_{\mathsf{l}}}, [], ts, mmts_\omega|_{mids_{\mathsf{l}}} \xrightarrow{tr}{}^{*}_{\delta} \underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}, \underline{ts_\omega}, \underline{mmts_\omega}|_{mids_{\mathsf{l}}}$

From DISJ and $mmts_1|_{mids_{\mathsf{r}}{}^{\mathsf{c}}} = mmts_\omega|_{mids_{\mathsf{r}}{}^{\mathsf{c}}}$, we have:
$mmts_1|_{mids_{\mathsf{l}}} = mmts_\omega|_{mids_{\mathsf{l}}}$.

From IH_{l}, we have:
$\exists tr_{\mathsf{hl}}, \overrightarrow{s_{\mathsf{hl}}}, \overrightarrow{c_{\mathsf{hl}}}, ts_{\mathsf{hl}}.$
$\overrightarrow{s_{\mathsf{l}}}, [], ts, mmts|_{mids_{\mathsf{l}}} \xrightarrow{tr_{\mathsf{hl}}}{}^{*}_{\delta} \overrightarrow{s_{\mathsf{hl}}}, \overrightarrow{c_{\mathsf{hl}}}, ts_{\mathsf{hl}}, \underline{mmts_\omega}|_{mids_{\mathsf{l}}} \wedge$
$tr_{\mathsf{hl}} \sim tr_1 +\!\!+ \underline{tr} \wedge$
$(\mathsf{STOP}([], []) \longrightarrow [] = \overrightarrow{s_{\mathsf{hl}}} \wedge [] = \overrightarrow{c_{\mathsf{hl}}} \wedge ts_1 = ts_{\mathsf{hl}} \wedge mmts_1|_{mids_{\mathsf{l}}} = \underline{mmts_\omega}|_{mids_{\mathsf{l}}} \wedge [] \sim \underline{tr}) \wedge$
$(\mathsf{STOP}(\underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}) \longrightarrow \underline{\overrightarrow{s_{\mathsf{m}}}} = \overrightarrow{s_{\mathsf{hl}}} \wedge \underline{\overrightarrow{c_{\mathsf{m}}}} = \overrightarrow{c_{\mathsf{hl}}} \wedge \underline{ts_\omega} = ts_{\mathsf{hl}})$.

From $\mathsf{STOP}([], [])$, we have:
$[] = \overrightarrow{s_{\mathsf{hl}}} \wedge [] = \overrightarrow{c_{\mathsf{hl}}} \wedge ts_1 = ts_{\mathsf{hl}} \wedge mmts_1|_{mids_{\mathsf{l}}} = \underline{mmts_\omega}|_{mids_{\mathsf{l}}} \wedge [] \sim \underline{tr}$.

From $mmts_\omega|_{mids_{\mathsf{l}}} = mmts_1|_{mids_{\mathsf{l}}} = \underline{mmts_\omega}|_{mids_{\mathsf{l}}}$ and $mmts_\omega|_{mids_{\mathsf{l}}{}^{\mathsf{c}}} = \underline{mmts_\omega}|_{mids_{\mathsf{l}}{}^{\mathsf{c}}}$, we have:
$mmts_\omega = \underline{mmts_\omega}$.

From Theorem B.6.7, we have:
$\overrightarrow{s_{\mathsf{l}}}, [], ts, mmts \xrightarrow{tr_{\mathsf{hl}}}{}^{*}_{\delta} [], [], ts_1, mmts_1$.

We have $\neg\mathsf{STOP}(\underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}})$, because otherwise, from IH_{l}, we have $\underline{\overrightarrow{s_{\mathsf{m}}}} = \overrightarrow{s_{\mathsf{hl}}} = [] \wedge \underline{\overrightarrow{c_{\mathsf{m}}}} = \overrightarrow{c_{\mathsf{hl}}} = []$, contradicting the assumption that $(\underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}) \neq ([], [])$.

We prove the goals with $tr_x = tr, \overrightarrow{s_x} = \overrightarrow{s_\omega}, \overrightarrow{c_x} = \overrightarrow{c_\omega}, ts_x = ts_\omega$ as follows:

(1) $\overrightarrow{s_{\mathsf{l}}} +\!\!+ \overrightarrow{s_{\mathsf{r}}}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

(2) From $[] \sim \underline{tr}$, we have: $tr \sim tr +\!\!+ \underline{tr}$

(3) $\overrightarrow{s_\omega} = \overrightarrow{s_\omega} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_\omega} \wedge ts_\omega = ts_\omega \wedge mmts_\omega = mmts_\omega \wedge [] \sim \underline{tr}$

(4) Since $(\underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}) \neq ([], [])$ and $\neg\mathsf{STOP}(\underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}})$, we have: $\neg\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}})$

- (**SEQ-LEFT-ONGOING**, **SEQ-LEFT-DONE**):

  For this sub-case, we have:

  - $\exists \overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}.$
    $\overrightarrow{s_{\mathsf{l}}}, [], ts, mmts_\omega \xrightarrow{tr}{}^{*}_{\delta} \overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}, ts_\omega, mmts_\omega \wedge (\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) = (\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}) +\!\!+ \overrightarrow{s_{\mathsf{r}}} \wedge (\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}) \neq ([], [])$
  - $\exists \underline{tr_1}, \underline{tr_2}, \underline{ts_1}, \underline{mmts_1}.$
    $\underline{tr} = \underline{tr_1} +\!\!+ \underline{tr_2} \wedge \overrightarrow{s_{\mathsf{l}}}, [], ts, mmts_\omega \xrightarrow{tr_1}{}^{*}_{\delta} [], [], \underline{ts_1}, \underline{mmts_1} \wedge$
    $\overrightarrow{s_{\mathsf{r}}}, [], \underline{ts_1}, \underline{mmts_1} \xrightarrow{tr_2}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  From IH_{l}, we have:

  $\exists tr_{\mathsf{hl}}, \overrightarrow{s_{\mathsf{hl}}}, \overrightarrow{c_{\mathsf{hl}}}, ts_{\mathsf{hl}}.$
  $\overrightarrow{s_{\mathsf{l}}}, [], ts, mmts \xrightarrow{tr_{\mathsf{hl}}}{}^{*}_{\delta} \overrightarrow{s_{\mathsf{hl}}}, \overrightarrow{c_{\mathsf{hl}}}, ts_{\mathsf{hl}}, \underline{mmts_1} \wedge$
  $tr_{\mathsf{hl}} \sim tr +\!\!+ \underline{tr_1} \wedge$
  $(\mathsf{STOP}(\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}) \longrightarrow \overrightarrow{s_{\mathsf{m}}} = \overrightarrow{s_{\mathsf{hl}}} \wedge \overrightarrow{c_{\mathsf{m}}} = \overrightarrow{c_{\mathsf{hl}}} \wedge ts_\omega = ts_{\mathsf{hl}} \wedge mmts_\omega = \underline{mmts_1} \wedge [] \sim \underline{tr_1}) \wedge$
  $(\mathsf{STOP}([], []) \longrightarrow [] = \overrightarrow{s_{\mathsf{hl}}} \wedge [] = \overrightarrow{c_{\mathsf{hl}}} \wedge \underline{ts_1} = ts_{\mathsf{hl}})$.

  From $\mathsf{STOP}([], [])$, we have:
  $[] = \overrightarrow{s_{\mathsf{hl}}} \wedge [] = \overrightarrow{c_{\mathsf{hl}}} \wedge \underline{ts_1} = ts_{\mathsf{hl}}$.

  We have $\neg\mathsf{STOP}(\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}})$, because otherwise, from IH_{l}, we have $\overrightarrow{s_{\mathsf{m}}} = \overrightarrow{s_{\mathsf{hl}}} = [] \wedge \overrightarrow{c_{\mathsf{m}}} = \overrightarrow{c_{\mathsf{hl}}} = []$, contradicting the assumption that $(\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{s_{\mathsf{m}}}) \neq ([], [])$.

  We prove the goals with $tr_x = tr_{\mathsf{hl}} +\!\!+ \underline{tr_2}, \overrightarrow{s_x} = \underline{\overrightarrow{s_\omega}}, \overrightarrow{c_x} = \underline{\overrightarrow{c_\omega}}, ts_x = \underline{ts_\omega}$ as follows:

  (1) From $\overrightarrow{s_{\mathsf{l}}}, [], ts, mmts \xrightarrow{tr_{\mathsf{hl}}}{}^{*}_{\delta} [], [], \underline{ts_1}, \underline{mmts_1}$ and Theorem B.6.4, we have:
  $\overrightarrow{s_{\mathsf{l}}} +\!\!+ \overrightarrow{s_{\mathsf{r}}}, [], ts, mmts \xrightarrow{tr_{\mathsf{hl}}}{}^{*}_{\delta} \overrightarrow{s_{\mathsf{r}}}, [], \underline{ts_1}, \underline{mmts_1} \xrightarrow{tr_2}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  (2) From $tr_{\mathsf{hl}} \sim tr +\!\!+ \underline{tr_1}$, we have: $tr_{\mathsf{hl}} +\!\!+ \underline{tr_2} \sim tr +\!\!+ \underline{tr_1} +\!\!+ \underline{tr_2}$
  From $\underline{tr_1} +\!\!+ \underline{tr_2} = \underline{tr}$, we have: $tr_{\mathsf{hl}} +\!\!+ \underline{tr_2} \sim tr +\!\!+ \underline{tr}$

  (3) Since $(\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}) \neq ([], [])$ and $\neg\mathsf{STOP}(\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}})$, we have: $\neg\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega})$

  (4) $\underline{\overrightarrow{s_\omega}} = \underline{\overrightarrow{s_\omega}} \wedge \underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_\omega}} \wedge \underline{ts_\omega} = \underline{ts_\omega}$

- (**SEQ-LEFT-ONGOING**, **SEQ-LEFT-ONGOING**):

  For this case, we have:

  ▷ $\exists \overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}.\ \overrightarrow{s_{\mathsf{l}}}, [], ts, mmts_\omega \xrightarrow{tr}{}^*_\delta\ \overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}, ts_\omega, mmts_\omega \wedge (\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) = (\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}) +\!\!+ \overrightarrow{s_{\mathsf{r}}} \wedge (\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}) \neq ([], [])$

  ▷ $\exists \underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}.\ \overrightarrow{s_{\mathsf{l}}}, [], ts, mmts_\omega \xrightarrow{tr}{}^*_\delta\ \underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}, \underline{ts_\omega}, \underline{mmts_\omega} \wedge (\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) = (\underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}) +\!\!+ \overrightarrow{s_{\mathsf{r}}} \wedge (\underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}) \neq ([], [])$

  From IH_{l}, we have:

  $\exists tr_{\mathsf{hl}}, \overrightarrow{s_{\mathsf{hl}}}, \overrightarrow{c_{\mathsf{hl}}}, ts_{\mathsf{hl}}.$
  $\overrightarrow{s_{\mathsf{l}}}, [], ts, mmts \xrightarrow{tr_{\mathsf{hl}}}{}^*_\delta\ \overrightarrow{s_{\mathsf{hl}}}, \overrightarrow{c_{\mathsf{hl}}}, ts_{\mathsf{hl}}, \underline{mmts_\omega} \wedge$
  $tr_{\mathsf{hl}} \sim tr +\!\!+ \underline{tr} \wedge$
  $(\mathsf{STOP}(\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}) \longrightarrow \overrightarrow{s_{\mathsf{m}}} = \overrightarrow{s_{\mathsf{hl}}} \wedge \overrightarrow{c_{\mathsf{m}}} = \overrightarrow{c_{\mathsf{hl}}} \wedge ts_\omega = ts_{\mathsf{hl}} \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}) \wedge$
  $(\mathsf{STOP}(\underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}) \longrightarrow \underline{\overrightarrow{s_{\mathsf{m}}}} = \overrightarrow{s_{\mathsf{hl}}} \wedge \underline{\overrightarrow{c_{\mathsf{m}}}} = \overrightarrow{c_{\mathsf{hl}}} \wedge \underline{ts_\omega} = ts_{\mathsf{hl}})$.

  There exists $\overrightarrow{s_y}, \overrightarrow{c_y}$ such that $(\overrightarrow{s_y}, \overrightarrow{c_y}) = (\overrightarrow{s_{\mathsf{hl}}}, \overrightarrow{c_{\mathsf{hl}}}) +\!\!+ \overrightarrow{s_{\mathsf{r}}}$. Thus we have:
  $\overrightarrow{s_{\mathsf{l}}} +\!\!+ \overrightarrow{s_{\mathsf{r}}}, [], ts, mmts \xrightarrow{tr_{\mathsf{hl}}}{}^*_\delta\ \overrightarrow{s_y}, \overrightarrow{c_y}, ts_{\mathsf{hl}}, mmts_\omega$.

  We prove the goals with $tr_x = tr_{\mathsf{hl}}, \overrightarrow{s_x} = \overrightarrow{s_y}, \overrightarrow{c_x} = \overrightarrow{c_y}, ts_x = ts_{\mathsf{hl}}$ as follows:

  (1) $\overrightarrow{s_{\mathsf{l}}} +\!\!+ \overrightarrow{s_{\mathsf{r}}}, [], ts, mmts \xrightarrow{tr_{\mathsf{hl}}}{}^*_\delta\ \overrightarrow{s_y}, \overrightarrow{c_y}, ts_{\mathsf{hl}}, mmts_\omega$

  (2) $tr_{\mathsf{hl}} \sim tr +\!\!+ \underline{tr}$

  (3) Since $(\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}) \neq ([], [])$ and $(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) = (\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}}) +\!\!+ \overrightarrow{s_{\mathsf{r}}}$,
  we have: $\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \mathsf{STOP}(\overrightarrow{s_{\mathsf{m}}}, \overrightarrow{c_{\mathsf{m}}})$.
  Thus we have: $\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_y} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_y} \wedge ts_\omega = ts_{\mathsf{hl}} \wedge mmts = \underline{mmts_\omega} \wedge [] \sim \underline{tr}$

  (4) Since $(\underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}) \neq ([], [])$ and $(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) = (\underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}}) +\!\!+ \overrightarrow{s_{\mathsf{r}}}$,
  we have: $\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \mathsf{STOP}(\underline{\overrightarrow{s_{\mathsf{m}}}}, \underline{\overrightarrow{c_{\mathsf{m}}}})$.
  Thus we have: $\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_y} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_y} \wedge \underline{ts_\omega} = ts_{\mathsf{hl}}$

- (**CALL**):

  By assumption, we have:

  ▷ FNJ: $\vdash \delta : \Delta$

  ▷ FNDR: $\forall f, \overrightarrow{prms}, \overrightarrow{s_f}.\ \Delta(f) = \mathsf{RW} \longrightarrow \delta(f) = (\overrightarrow{prms}, \overrightarrow{s_f}) \longrightarrow \mathsf{DR}(\delta, \overrightarrow{s_f})$

  ▷ EX1: $[r := f(\overrightarrow{e} +\!\!+ \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{tr}{}^+_\delta\ \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$

  ▷ EX2: $[r := f(\overrightarrow{e} +\!\!+ \mathsf{mid}.lab)], [], ts, mmts_\omega \xrightarrow{tr}{}^+_\delta\ \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  For this case, we have:

  ▷ RW: $\Delta(f) = \mathsf{RW}$

  We aim to prove the following goals: $\exists tr_x, \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x$.

  (1) $[r := f(\overrightarrow{e} +\!\!+ \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{tr_x}{}^*_\delta\ \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x, \underline{mmts_\omega}$

  (2) $tr_x \sim tr +\!\!+ \underline{tr}$

(3) $\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_x} \wedge \overrightarrow{c_\omega} = \overrightarrow{c_x} \wedge ts_\omega = ts_x \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}$

(4) $\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_x} \wedge \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_x} \wedge \underline{ts_\omega} = ts_x$

From EX1 and EX2, there exists $\overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1, \underline{\overrightarrow{s_1}}, \underline{\overrightarrow{c_1}}, \underline{ts_1}, \underline{mmts_1}$ such that:

- ▷ $[r := f(\overrightarrow{e} +\!\!+ \mathsf{mid}.lab)], [], ts, mmts \rightarrow_\delta \overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1 \xrightarrow{tr_1}{}^*_\delta \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$
- ▷ $[r := f(\overrightarrow{e} +\!\!+ \mathsf{mid}.lab)], [], ts, mmts_\omega \rightarrow_\delta \underline{\overrightarrow{s_1}}, \underline{\overrightarrow{c_1}}, \underline{ts_1}, \underline{mmts_1} \xrightarrow{tr_1}{}^*_\delta \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

From FNJ, RW, and Theorem B.6.15, we have:

$f \in dom(\Delta)$ and $\exists \overrightarrow{prms}, \overrightarrow{s_f}.\ \delta(f) = (\overrightarrow{prms}, \overrightarrow{s_f})$.

Since the only possible first steps are **CALL**, we have:

- ▷ EXB1: $[r := f(\overrightarrow{e} +\!\!+ \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[]}_\delta \overrightarrow{s_f}, \overrightarrow{c_1}, ts_1, mmts \xrightarrow{tr}{}^*_\delta \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$
- ▷ EXB2: $[r := f(\overrightarrow{e} +\!\!+ \mathsf{mid}.lab)], [], ts, mmts_\omega \xrightarrow{[]}_\delta \overrightarrow{s_f}, \underline{\overrightarrow{c_1}}, \underline{ts_1}, mmts_\omega \xrightarrow{tr}{}^*_\delta \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

Let $\overrightarrow{v} = ts.\mathsf{regs}(\overrightarrow{e})$. We have:

- ▷ $\overrightarrow{c_1} = [\mathtt{fnCont}(ts.\mathsf{regs}, r, [])] = \underline{\overrightarrow{c_1}}$
- ▷ $ts_1 = ts\left[\mathsf{regs} \mapsto \sqcup_i [prms_i \mapsto v_i]\right] = \underline{ts_1}$

We apply Theorem B.6.13 to the later transitions of EXB1 and EXB2 and do case analysis on the lemma's conclusion:

- ○ (**CALL-ONGOING**, **CALL-ONGOING**):

  For this sub-case, we have:

  - ▷ $\exists \overrightarrow{c_{\mathsf{pfx}}}.\ \overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ \overrightarrow{c_1} \wedge \overrightarrow{s_f}, [], ts_1, mmts \xrightarrow{tr}{}^*_\delta \overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}, ts_\omega, mmts_\omega$
  - ▷ $\exists \underline{\overrightarrow{c_{\mathsf{pfx}}}}.\ \underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_{\mathsf{pfx}}}} +\!\!+ \overrightarrow{c_1} \wedge \overrightarrow{s_f}, [], ts_1, mmts_\omega \xrightarrow{tr}{}^*_\delta \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_{\mathsf{pfx}}}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  From FNDR, we have:

  $\exists tr_{\mathsf{hf}}, \overrightarrow{s_{\mathsf{hf}}}, \overrightarrow{c_{\mathsf{hf}}}, ts_{\mathsf{hf}}.$
  $\overrightarrow{s_f}, [], ts_1, mmts \xrightarrow{tr_{\mathsf{hf}}}{}^*_\delta \overrightarrow{s_{\mathsf{hf}}}, \overrightarrow{c_{\mathsf{hf}}}, ts_{\mathsf{hf}}, \underline{mmts_\omega} \wedge$
  $tr_{\mathsf{hf}} \sim tr +\!\!+ \underline{tr} \wedge$
  $(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_{\mathsf{hf}}} \wedge \overrightarrow{c_{\mathsf{pfx}}} = \overrightarrow{c_{\mathsf{hf}}} \wedge ts_\omega = ts_{\mathsf{hf}} \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}) \wedge$
  $(\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_{\mathsf{pfx}}}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_{\mathsf{hf}}} \wedge \underline{\overrightarrow{c_{\mathsf{pfx}}}} = \overrightarrow{c_{\mathsf{hf}}} \wedge \underline{ts_\omega} = ts_{\mathsf{hf}})$.

  From Theorem B.6.5, we have:
  $\overrightarrow{s_f}, \overrightarrow{c_1}, ts_1, mmts \xrightarrow{tr_{\mathsf{hf}}}{}^*_\delta \overrightarrow{s_{\mathsf{hf}}}, \overrightarrow{c_{\mathsf{hf}}} +\!\!+ \overrightarrow{c_1}, ts_{\mathsf{hf}}, \underline{mmts_\omega}$.

  We prove the goals with $tr_x = tr_{\mathsf{hf}}, \overrightarrow{s_x} = \overrightarrow{s_{\mathsf{hf}}}, \overrightarrow{c_x} = \overrightarrow{c_{\mathsf{hf}}} +\!\!+ \overrightarrow{c_1}, ts_x = ts_{\mathsf{hf}}$ as follows:

  (1) $[r := f(\overrightarrow{e} +\!\!+ \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[]}_\delta \overrightarrow{s_f}, \overrightarrow{c_1}, ts_1, mmts \xrightarrow{tr_{\mathsf{hf}}}{}^*_\delta \overrightarrow{s_{\mathsf{hf}}}, \overrightarrow{c_{\mathsf{hf}}} +\!\!+ \overrightarrow{c_1}, ts_{\mathsf{hf}}, \underline{mmts_\omega}$

  (2) $tr_{\mathsf{hf}} \sim tr +\!\!+ \underline{tr}$

  (3) From $\overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ [\mathtt{fnCont}(ts.\mathsf{regs}, r, [])]$, we have: $\neg\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega})$

  (4) From $\underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_{\mathsf{pfx}}}} +\!\!+ [\mathtt{fnCont}(ts.\mathsf{regs}, r, [])]$, we have: $\neg\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}})$

- (**CALL-ONGOING**, **CALL-DONE**):

  For this sub-case, we have:

  - $\triangleright\ \exists \overrightarrow{c_{\mathsf{pfx}}}.\ \overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ \overrightarrow{c_1}\ \wedge\ \overrightarrow{s_f}, [], ts_1, mmts\ \xrightarrow{tr}{}^{*}_{\delta}\ \overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}, ts_\omega, mmts_\omega$
  - $\triangleright\ \exists \overrightarrow{\underline{s_{\mathsf{r}}}}, \overrightarrow{\underline{c_{\mathsf{r}}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_{\mathsf{r}}}, \underline{e}.$
    $\overrightarrow{s_f}, [], ts_1, mmts_\omega\ \xrightarrow{tr_{\mathsf{r}}}{}^{*}_{\delta}\ (\texttt{return}\ \underline{e}) :: \overrightarrow{\underline{s_{\mathsf{r}}}}, \overrightarrow{\underline{c_{\mathsf{r}}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_{\mathsf{r}}}\ \wedge$
    $(\texttt{return}\ \underline{e}) :: \overrightarrow{\underline{s_{\mathsf{r}}}}, \overrightarrow{\underline{c_{\mathsf{r}}}} +\!\!+ \overrightarrow{c_1}, \underline{ts_{\mathsf{r}}}, \underline{mmts_{\mathsf{r}}}\ \xrightarrow{[]}_{\delta}\ \overrightarrow{\underline{s_\omega}}, \overrightarrow{\underline{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}\ \wedge$
    $\overrightarrow{\underline{s_\omega}} = []\ \wedge\ \overrightarrow{\underline{c_\omega}} = []$

  From FNDR, we have:

  $\exists tr_{\mathsf{hf}}, \overrightarrow{s_{\mathsf{hf}}}, \overrightarrow{c_{\mathsf{hf}}}, ts_{\mathsf{hf}}.$
  $\overrightarrow{s_f}, [], ts_1, mmts\ \xrightarrow{tr_{\mathsf{hf}}}{}^{*}_{\delta}\ \overrightarrow{s_{\mathsf{hf}}}, \overrightarrow{c_{\mathsf{hf}}}, ts_{\mathsf{hf}}, \underline{mmts_{\mathsf{r}}}\ \wedge$
  $tr_{\mathsf{hf}} \sim tr +\!\!+ \underline{tr}\ \wedge$
  $(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}})\ \longrightarrow\ \overrightarrow{s_\omega} = \overrightarrow{s_{\mathsf{hf}}}\ \wedge\ \overrightarrow{c_{\mathsf{pfx}}} = \overrightarrow{c_{\mathsf{hf}}}\ \wedge\ ts_\omega = ts_{\mathsf{hf}}\ \wedge\ mmts_\omega = \underline{mmts_{\mathsf{r}}}\ \wedge\ [] \sim \underline{tr})\ \wedge$
  $(\mathsf{STOP}((\texttt{return}\ \underline{e}) :: \overrightarrow{\underline{s_{\mathsf{r}}}}, \overrightarrow{\underline{c_{\mathsf{r}}}})\ \longrightarrow\ (\texttt{return}\ \underline{e}) :: \overrightarrow{\underline{s_{\mathsf{r}}}} = \overrightarrow{s_{\mathsf{hf}}}\ \wedge\ \overrightarrow{\underline{c_{\mathsf{r}}}} = \overrightarrow{c_{\mathsf{hf}}}\ \wedge\ \underline{ts_{\mathsf{r}}} = ts_{\mathsf{hf}})\,.$

  From $\mathsf{Loops}(\overrightarrow{\underline{c_{\mathsf{r}}}})$ and $\mathsf{STOP}((\texttt{return}\ \underline{e}) :: \overrightarrow{\underline{s_{\mathsf{r}}}}, \overrightarrow{\underline{c_{\mathsf{r}}}})$, we have:
  $(\texttt{return}\ \underline{e}) :: \overrightarrow{\underline{s_{\mathsf{r}}}} = \overrightarrow{s_{\mathsf{hf}}}\ \wedge\ \overrightarrow{\underline{c_{\mathsf{r}}}} = \overrightarrow{c_{\mathsf{hf}}}\ \wedge\ \underline{ts_{\mathsf{r}}} = ts_{\mathsf{hf}}\ .$

  From Theorem B.6.5, we have:
  $\overrightarrow{s_f}, \overrightarrow{c_1}, ts_1, mmts\ \xrightarrow{tr_{\mathsf{hf}}}{}^{*}_{\delta}\ (\texttt{return}\ \underline{e}) :: \overrightarrow{\underline{s_{\mathsf{r}}}}, \overrightarrow{\underline{c_{\mathsf{r}}}} +\!\!+ \overrightarrow{c_1}, \underline{ts_{\mathsf{r}}}, \underline{mmts_{\mathsf{r}}}\ .$

  We prove the goals with $tr_x = tr_{\mathsf{hf}}, \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = \underline{ts_\omega}$ as follows:

  (1) $[r := f(\overrightarrow{e} +\!\!+ \mathsf{mid}.lab)], [], ts, mmts\ \xrightarrow{[]}_{\delta}\ \overrightarrow{s_f}, \overrightarrow{c_1}, ts_1, mmts$
  $\xrightarrow{tr_{\mathsf{hf}}}{}^{*}_{\delta}\ (\texttt{return}\ \underline{e}) :: \overrightarrow{\underline{s_{\mathsf{r}}}}, \overrightarrow{\underline{c_{\mathsf{r}}}} +\!\!+ \overrightarrow{c_1}, \underline{ts_{\mathsf{r}}}, \underline{mmts_{\mathsf{r}}}\ \xrightarrow{[]}_{\delta}\ [], [], \underline{ts_\omega}, \underline{mmts_\omega}$

  (2) $tr_{\mathsf{hf}} \sim tr +\!\!+ \underline{tr}$

  (3) From $\overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ [\mathtt{fnCont}(ts.\mathsf{regs}, r, [])]$, we have: $\neg\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega})$

  (4) $[] = []\ \wedge\ [] = []\ \wedge\ \underline{ts_\omega} = \underline{ts_\omega}$

- (**CALL-DONE**, **CALL-ONGOING**):

  For this sub-case, we have:

  - $\triangleright\ \exists \overrightarrow{\underline{s_{\mathsf{r}}}}, \overrightarrow{\underline{c_{\mathsf{r}}}}, ts_{\mathsf{r}}, mmts_{\mathsf{r}}, e.$
    $\overrightarrow{s_f}, [], ts_1, mmts\ \xrightarrow{tr_{\mathsf{r}}}{}^{*}_{\delta}\ (\texttt{return}\ e) :: \overrightarrow{\underline{s_{\mathsf{r}}}}, \overrightarrow{\underline{c_{\mathsf{r}}}}, ts_{\mathsf{r}}, mmts_{\mathsf{r}}\ \wedge$
    $(\texttt{return}\ e) :: \overrightarrow{\underline{s_{\mathsf{r}}}}, \overrightarrow{\underline{c_{\mathsf{r}}}} +\!\!+ \overrightarrow{c_1}, ts_{\mathsf{r}}, mmts_{\mathsf{r}}\ \xrightarrow{[]}_{\delta}\ \overrightarrow{\underline{s_\omega}}, \overrightarrow{\underline{c_\omega}}, ts_\omega, mmts_\omega\ \wedge$
    $\overrightarrow{\underline{s_\omega}} = []\ \wedge\ \overrightarrow{\underline{c_\omega}} = []$
  - $\triangleright\ \exists \overrightarrow{\underline{c_{\mathsf{pfx}}}}.\ \overrightarrow{\underline{c_\omega}} = \overrightarrow{\underline{c_{\mathsf{pfx}}}} +\!\!+ \overrightarrow{c_1}\ \wedge\ \overrightarrow{s_f}, [], ts_1, mmts_\omega\ \xrightarrow{tr}{}^{*}_{\delta}\ \overrightarrow{\underline{s_\omega}}, \overrightarrow{\underline{c_{\mathsf{pfx}}}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  From FNDR, we have:

$\exists tr_{\mathsf{hf}}, \overrightarrow{s_{\mathsf{hf}}}, \overrightarrow{c_{\mathsf{hf}}}, ts_{\mathsf{hf}}.$
$\overrightarrow{s_f}, [], ts_1, mmts \xrightarrow{tr_{\mathsf{hf}}}{}^{*}_{\delta} \overrightarrow{s_{\mathsf{hf}}}, \overrightarrow{c_{\mathsf{hf}}}, ts_{\mathsf{hf}}, \underline{mmts_\omega} \wedge$
$tr_{\mathsf{hf}} \sim tr +\!\!+ \underline{tr} \wedge$
$(\mathsf{STOP}((\texttt{return}\ e) :: \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c_{\mathsf{r}}}) \longrightarrow$
$(\texttt{return}\ e) :: \overrightarrow{s_{\mathsf{r}}} = \overrightarrow{s_{\mathsf{hf}}} \wedge \overrightarrow{c_{\mathsf{r}}} = \overrightarrow{c_{\mathsf{hf}}} \wedge ts_{\mathsf{r}} = ts_{\mathsf{hf}} \wedge mmts_{\mathsf{r}} = \underline{mmts_\omega} \wedge [] \sim \underline{tr}) \wedge$
$(\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_{\mathsf{pfx}}}}) \longrightarrow \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_{\mathsf{hf}}} \wedge \underline{\overrightarrow{c_{\mathsf{pfx}}}} = \overrightarrow{c_{\mathsf{hf}}} \wedge \underline{ts_\omega} = ts_{\mathsf{hf}})$.

From $\mathsf{Loops}(\overrightarrow{c_{\mathsf{r}}})$ and $\mathsf{STOP}((\texttt{return}\ e) :: \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c_{\mathsf{r}}})$, we have:
$(\texttt{return}\ e) :: \overrightarrow{s_{\mathsf{r}}} = \overrightarrow{s_{\mathsf{hf}}} \wedge \overrightarrow{c_{\mathsf{r}}} = \overrightarrow{c_{\mathsf{hf}}} \wedge ts_{\mathsf{r}} = ts_{\mathsf{hf}} \wedge mmts_{\mathsf{r}} = \underline{mmts_\omega} \wedge [] \sim \underline{tr}$.

From Theorem B.6.5, we have:
$\overrightarrow{s_f}, \overrightarrow{c_1}, ts_1, mmts \xrightarrow{tr_{\mathsf{hf}}}{}^{*}_{\delta} (\texttt{return}\ e) :: \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c_{\mathsf{r}}} +\!\!+ \overrightarrow{c_1}, ts_{\mathsf{r}}, mmts_{\mathsf{r}}$.

From **RETURN** step, we have:
$mmts_{\mathsf{r}} = mmts_\omega$.

We prove the goals with $tr_x = tr, \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = ts_\omega$ as follows:

(1) $[r := f(\overrightarrow{e} +\!\!+ \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} [], [], ts_\omega, mmts_{\mathsf{r}}$
(2) From $[] \sim \underline{tr}$, we have: $tr \sim tr +\!\!+ \underline{tr}$
(3) $[] = [] \wedge [] = [] \wedge ts_\omega = ts_\omega \wedge mmts_{\mathsf{r}} = \underline{mmts_\omega} \wedge [] \sim \underline{tr}$
(4) From $\underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_{\mathsf{pfx}}}} +\!\!+ [\mathtt{fnCont}(ts.\mathsf{regs}, r, [])]$, we have: $\neg\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}})$

- (**CALL-DONE**, **CALL-DONE**):

  For this sub-case, we have:

  - $\exists \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c_{\mathsf{r}}}, ts_{\mathsf{r}}, mmts_{\mathsf{r}}, e.$
    $\overrightarrow{s_f}, [], ts_1, mmts \xrightarrow{tr_{\mathsf{r}}}{}^{*}_{\delta} (\texttt{return}\ e) :: \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c_{\mathsf{r}}}, ts_{\mathsf{r}}, mmts_{\mathsf{r}} \wedge$
    $(\texttt{return}\ e) :: \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c_{\mathsf{r}}} +\!\!+ \overrightarrow{c_1}, ts_{\mathsf{r}}, mmts_{\mathsf{r}} \xrightarrow{[]}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega \wedge$
    $\overrightarrow{s_\omega} = [] \wedge \overrightarrow{c_\omega} = []$
  - $\exists \underline{\overrightarrow{s_{\mathsf{r}}}}, \underline{\overrightarrow{c_{\mathsf{r}}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_{\mathsf{r}}}, \underline{e}.$
    $\overrightarrow{s_f}, [], ts_1, mmts_\omega \xrightarrow{tr_{\mathsf{r}}}{}^{*}_{\delta} (\texttt{return}\ \underline{e}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, \underline{\overrightarrow{c_{\mathsf{r}}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_{\mathsf{r}}} \wedge$
    $(\texttt{return}\ \underline{e}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, \underline{\overrightarrow{c_{\mathsf{r}}}} +\!\!+ \overrightarrow{c_1}, \underline{ts_{\mathsf{r}}}, \underline{mmts_{\mathsf{r}}} \xrightarrow{[]}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega} \wedge$
    $\underline{\overrightarrow{s_\omega}} = [] \wedge \underline{\overrightarrow{c_\omega}} = []$

  From FNDR, we have:
  $\exists tr_{\mathsf{hf}}, \overrightarrow{s_{\mathsf{hf}}}, \overrightarrow{c_{\mathsf{hf}}}, ts_{\mathsf{hf}}.$
  $\overrightarrow{s_f}, [], ts_1, mmts \xrightarrow{tr_{\mathsf{hf}}}{}^{*}_{\delta} \overrightarrow{s_{\mathsf{hf}}}, \overrightarrow{c_{\mathsf{hf}}}, ts_{\mathsf{hf}}, \underline{mmts_{\mathsf{r}}} \wedge$
  $tr_{\mathsf{hf}} \sim tr +\!\!+ \underline{tr} \wedge$
  $(\mathsf{STOP}((\texttt{return}\ e) :: \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c_{\mathsf{r}}}) \longrightarrow$
  $(\texttt{return}\ e) :: \overrightarrow{s_{\mathsf{r}}} = \overrightarrow{s_{\mathsf{hf}}} \wedge \overrightarrow{c_{\mathsf{r}}} = \overrightarrow{c_{\mathsf{hf}}} \wedge ts_{\mathsf{r}} = ts_{\mathsf{hf}} \wedge mmts_{\mathsf{r}} = \underline{mmts_{\mathsf{r}}} \wedge [] \sim \underline{tr}) \wedge$

$(\mathsf{STOP}((\texttt{return}\ \underline{e}) :: \underline{\vec{s_r}}, \underline{\vec{c_r}}) \longrightarrow (\texttt{return}\ \underline{e}) :: \underline{\vec{s_r}} = \overrightarrow{s_{hf}} \wedge \underline{\vec{c_r}} = \overrightarrow{c_{hf}} \wedge \underline{ts_r} = ts_{hf})$.

From $\mathsf{Loops}(\vec{c_r})$, $\mathsf{STOP}((\texttt{return}\ e) :: \vec{s_r}, \vec{c_r})$, $\mathsf{Loops}(\underline{\vec{c_r}})$ and $\mathsf{STOP}((\texttt{return}\ \underline{e}) :: \underline{\vec{s_r}}, \underline{\vec{c_r}})$, we have:
$(\texttt{return}\ \underline{e}) :: \underline{\vec{s_r}} = \overrightarrow{s_{hf}} \wedge \underline{\vec{c_r}} = \overrightarrow{c_{hf}} \wedge ts_r = \underline{ts_r} = ts_{hf} \wedge mmts_r = \underline{mmts_r} \wedge [] \sim \underline{tr}$.

From Theorem B.6.5, we have:
$\vec{s_f}, \vec{c_1}, ts_1, mmts \xrightarrow{tr_{hf}}{}^{*}_{\delta} (\texttt{return}\ \underline{e}) :: \underline{\vec{s_r}}, \underline{\vec{c_r}} +\!\!+ \vec{c_1}, ts_r, mmts_r$.

From **RETURN** step, we have:

- ▷ Let $v = ts_r.\mathsf{regs}(e)$. We have $ts_\omega = ts_r\,[\mathsf{regs} \mapsto ts.\mathsf{regs}\,[r \mapsto v]] = \underline{ts_\omega}$.
- ▷ $mmts_r = mmts_\omega = \underline{mmts_\omega}$

We prove the goals with $tr_x = tr_{hf}, \vec{s_x} = [], \vec{c_x} = [], ts_x = ts_\omega$ as follows:

(1) $[r := f(\vec{e} +\!\!+ \mathsf{mid}.lab)], [], ts, mmts \xrightarrow{[]}_{\delta} \vec{s_f}, \vec{c_1}, ts_1, mmts$
$\xrightarrow{tr_{hf}}{}^{*}_{\delta} (\texttt{return}\ \underline{e}) :: \underline{\vec{s_r}}, \underline{\vec{c_r}} +\!\!+ \vec{c_1}, ts_r, mmts_r \xrightarrow{[]}_{\delta} [], [], ts_\omega, mmts_r$

(2) $tr_{hf} \sim tr +\!\!+ \underline{tr}$

(3) $[] = [] \wedge [] = [] \wedge ts_\omega = ts_\omega \wedge mmts_r = \underline{mmts_r} \wedge [] \sim \underline{tr}$

(4) $[] = [] \wedge [] = [] \wedge ts_\omega = ts_\omega$

- (**LOOP-SIMPLE**):

  This case is a simplification of the next case for (**LOOP**).

- (**LOOP**):

  By assumption, we have:

  - ▷ FNJ: $\vdash \delta : \Delta$
  - ▷ FNDR: $\forall f, \overrightarrow{prms}, \vec{s_f}.\ \Delta(f) = \mathsf{RW} \longrightarrow \delta(f) = (\overrightarrow{prms}, \vec{s_f}) \longrightarrow \mathsf{DR}(\delta, \vec{s_f})$
  - ▷ EX1: $[\texttt{loop}\ r\ e\ ((r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \vec{s})], [], ts, mmts \xrightarrow{tr}{}^{+}_{\delta} \vec{s_\omega}, \vec{c_\omega}, ts_\omega, mmts_\omega$
  - ▷ EX2: $[\texttt{loop}\ r\ e\ ((r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \vec{s})], [], ts, mmts_\omega \xrightarrow{tr}{}^{+}_{\delta} \underline{\vec{s_\omega}}, \underline{\vec{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

  For this case, we have:

  - ▷ $BODY$: $\Delta \vdash_{labs} \vec{s}$
  - ▷ NIN: $lab \notin labs$
  - ▷ IH: $\forall tr, \underline{tr}, \vec{s_\omega}, \underline{\vec{s_\omega}}, \vec{c_\omega}, \underline{\vec{c_\omega}}, ts, ts_\omega, \underline{ts_\omega}, mmts, mmts_\omega, \underline{mmts_\omega}.$
    $\vec{s}, [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} \vec{s_\omega}, \vec{c_\omega}, ts_\omega, mmts_\omega \longrightarrow$
    $\vec{s}, [], ts, mmts_\omega \xrightarrow{tr}{}^{*}_{\delta} \underline{\vec{s_\omega}}, \underline{\vec{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega} \longrightarrow$
    $\exists tr_h, \vec{s_h}, \vec{c_h}, ts_h.$
    $\vec{s}, [], ts, mmts \xrightarrow{tr_h}{}^{*}_{\delta} \vec{s_h}, \vec{c_h}, ts_h, \underline{mmts_\omega} \wedge$

$tr_{\mathsf{h}} \sim tr +\!\!+ \underline{tr} \ \wedge$
$(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \ \longrightarrow \ \overrightarrow{s_\omega} = \overrightarrow{s_{\mathsf{h}}} \ \wedge \ \overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{h}}} \ \wedge \ ts_\omega = ts_{\mathsf{h}} \ \wedge \ mmts_\omega = \underline{mmts_\omega} \ \wedge \ [] \sim \underline{tr}) \ \wedge$
$(\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \ \longrightarrow \ \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_{\mathsf{h}}} \ \wedge \ \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_{\mathsf{h}}} \ \wedge \ \underline{ts_\omega} = ts_{\mathsf{h}})$

We aim to prove the following goals: $\exists tr_x, \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x.$

(1) $[\texttt{loop}\ r\ e\ ((r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts \ \xrightarrow{tr_x}{}^{*}_{\delta} \ \overrightarrow{s_x}, \overrightarrow{c_x}, ts_x, \underline{mmts_\omega}$

(2) $tr_x \sim tr +\!\!+ \underline{tr}$

(3) $\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega}) \ \longrightarrow \ \overrightarrow{s_\omega} = \overrightarrow{s_x} \ \wedge \ \overrightarrow{c_\omega} = \overrightarrow{c_x} \ \wedge \ ts_\omega = ts_x \ \wedge \ mmts_\omega = \underline{mmts_\omega} \ \wedge \ [] \sim \underline{tr}$

(4) $\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}) \ \longrightarrow \ \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_x} \ \wedge \ \underline{\overrightarrow{c_\omega}} = \overrightarrow{c_x} \ \wedge \ \underline{ts_\omega} = ts_x$

From EX1 and EX2, there exists $\overrightarrow{s_1}, \overrightarrow{c_1}, ts_1, mmts_1, \underline{\overrightarrow{s_1}}, \underline{\overrightarrow{c_1}}, \underline{ts_1}, \underline{mmts_1}$ such that:

- ▷ $[\texttt{loop}\ r\ e\ ((r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts$
  $\rightarrow_\delta \ \overrightarrow{s_{\mathsf{b}}}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{b}}, mmts_{\mathsf{b}} \ \xrightarrow{tr_{\mathsf{b}}}{}^{*}_{\delta} \ \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$
- ▷ $[\texttt{loop}\ r\ e\ ((r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts_\omega$
  $\rightarrow_\delta \ \underline{\overrightarrow{s_{\mathsf{b}}}}, \underline{\overrightarrow{c_{\mathsf{b}}}}, \underline{ts_{\mathsf{b}}}, \underline{mmts_{\mathsf{b}}} \ \xrightarrow{tr_{\mathsf{b}}}{}^{*}_{\delta} \ \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

Let $mid_{\mathsf{pfx}} = ts.\mathsf{regs}(\mathsf{mid})$ and $mids = \mu(mid_{\mathsf{pfx}}, labs)$.

Since the only possible first steps are **LOOP**, we have:

- ▷ EXB1: $[\texttt{loop}\ r\ e\ ((r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts$
  $\xrightarrow{[]}_\delta \ (r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{b}}, mmts \ \xrightarrow{tr}{}^{*}_{\delta} \ \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$
- ▷ EXB2: $[\texttt{loop}\ r\ e\ ((r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts_\omega$
  $\xrightarrow{[]}_\delta \ (r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \underline{\overrightarrow{c_{\mathsf{b}}}}, \underline{ts_{\mathsf{b}}}, mmts_\omega \ \xrightarrow{tr}{}^{*}_{\delta} \ \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

If EXB2's later transitions are empty, the proof is essentially the same with the case that EX2 itself is empty transitions (the entire proof's first case). From now on, we prove for the case that EXB2's later transitions are not empty.

Let $v = ts.\mathsf{regs}(e)$. We have:

- ▷ $\overrightarrow{c_{\mathsf{b}}} = [\texttt{loopCont}(ts.\mathsf{regs}, r, (r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [])] = \underline{\overrightarrow{c_{\mathsf{b}}}}$
- ▷ $ts_{\mathsf{b}} = ts\ \Big[\mathsf{regs} \mapsto ts.\mathsf{regs}[r \mapsto v]\Big] = \underline{ts_{\mathsf{b}}}$

We apply Theorem B.6.10 to EXB1's later transitions and do case analysis on the lemma's conclusion:

- ○ (**LOOP-DONE**):

  For this sub-case, we have:
  $(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{b}}, mmts \ \xrightarrow{tr/\overrightarrow{c_{\mathsf{b}}}}{}^{*}_{\delta} \ (\texttt{break}) :: \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{r}}, mmts_\omega \ \wedge$
  $\overrightarrow{s_\omega} = [] \ \wedge \ \overrightarrow{c_\omega} = [] \ \wedge \ ts_\omega = \langle \mathsf{regs} : ts.\mathsf{regs}; \mathsf{time} : ts_{\mathsf{r}}.\mathsf{time} \rangle$.

  We identify the first execution's last iteration by applying Theorem B.6.12 as follows:

$\exists ts_1, mmts_1, tr_1, tr_2.$
$tr = tr_1 +\!\!+ tr_2 \ \wedge$
$(((ts_{\mathsf{b}}, mmts) = (ts_1, mmts_1) \ \wedge \ tr_1 = []) \ \vee$
$(\exists e, \vec{s_{\mathsf{r}}}, ts_{\mathsf{r}}. \ (r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, \vec{c_{\mathsf{b}}}, ts_{\mathsf{b}}, mmts$
$\xrightarrow{tr_1/\vec{c_{\mathsf{b}}}}{}^{*}_{\delta} \ (\mathtt{continue}\ e) :: \vec{s_{\mathsf{r}}}, \vec{c_{\mathsf{b}}}, ts_{\mathsf{r}}, mmts_1 \ \wedge$
$ts_1 = \langle \mathsf{regs} : \sigma[r \mapsto ts_{\mathsf{r}}.\mathsf{regs}(e)]; \mathsf{time} : ts_{\mathsf{r}}.\mathsf{time}\rangle)) \ \wedge$
$\vec{c_{\mathsf{b}}} = [] +\!\!+ \vec{c_{\mathsf{b}}} \ \wedge$
$(r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, [], ts_1, mmts_1 \ \xrightarrow{tr_2}{}^{*}_{\delta} \ (\mathtt{break}) :: \vec{s_{\mathsf{r}}}, [], ts_{\mathsf{r}}, mmts_\omega \ .$

We do a case analysis on
$(r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, [], ts_1, mmts_1 \xrightarrow{tr_2}{}^{*}_{\delta} (\mathtt{break}) :: \vec{s_{\mathsf{r}}}, [], ts_{\mathsf{r}}, mmts_\omega$: **CHKPT-CALL** or **CHKPT-REPLAY**.

- (**CHKPT-CALL**):
  From the semantics, EX1 can take only the following steps:
  $(r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, [], ts_1, mmts_1$
  $\xrightarrow{[]}_{\delta} \ [\mathtt{return}\ r], \overrightarrow{c_{\mathsf{chk}}}, ts_1, mmts_1$
  $\xrightarrow{[]}_{\delta} \ \vec{s}, [], ts_{(1,1)}, mmts_{(1,1)}$
  $\xrightarrow{tr_2}{}^{*}_{\delta} (\mathtt{break}) :: \vec{s_{\mathsf{r}}}, [], ts_{\mathsf{r}}, mmts_\omega \ .$

  From Theorem B.6.7, we have:
  $mmts_{(1,1)}|_{mids^{\mathsf{c}}} = mmts_\omega|_{mids^{\mathsf{c}}} \ .$

  From NIN, we have:
  $mmts_{(1,1)}[mid] = mmts_\omega[mid] \ .$

  From Theorem B.6.14, we have:
  $ts_{\mathsf{b}}.\mathsf{time} \leq ts_1.\mathsf{time} \ .$

  From $ts_1.\mathsf{time} < mmts_{(1,1)}[mid].\mathsf{time} = mmts_\omega[mid].\mathsf{time}$, by **CHKPT-REPLAY**, we have:
  $(r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, \vec{c_{\mathsf{b}}}, ts_{\mathsf{b}}, mmts_\omega \ \xrightarrow{[]}_{\delta} \ \vec{s}, \vec{c_{\mathsf{b}}}, \underline{ts_1}, mmts_\omega \ .$

  From $ts_{(1,1)}.\mathsf{regs}(r) = mmts_{(1,1)}[mid].\mathsf{val} = mmts_\omega[mid].\mathsf{val} = \underline{ts_1}.\mathsf{regs}(r)$
  and $ts_{(1,1)}.\mathsf{time} = mmts_{(1,1)}[mid].\mathsf{time} = mmts_\omega[mid].\mathsf{time} = \underline{ts_1}.\mathsf{time}$, we have:
  $ts_{(1,1)} = \underline{ts_1} \ .$

  We apply Theorem B.6.10 to EXB2's later transitions,
  $(r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, \vec{c_{\mathsf{b}}}, ts_{\mathsf{b}}, mmts_\omega \ \xrightarrow{tr}{}^{*}_{\delta} \ \underline{\vec{s_\omega}}, \underline{\vec{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega} \ ,$
  and do case analysis on the lemma's conclusion:

  † (**LOOP-DONE**):
  For this sub-case, we have:
  $(r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, \vec{c_{\mathsf{b}}}, ts, mmts_\omega$
  $\xrightarrow{[]}_{\delta} \ \vec{s}, \vec{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_\omega$
  $\xrightarrow{tr/\vec{c_{\mathsf{b}}}}{}^{*}_{\delta} (\mathtt{break}) :: \underline{\vec{s_{\mathsf{r}}}}, \vec{c_{\mathsf{b}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_\omega} \ \wedge$

$\underline{\overrightarrow{s_\omega}} = [] \ \wedge \ \underline{\overrightarrow{c_\omega}} = [] \ \wedge \ \underline{ts_\omega} = \langle \mathsf{regs} : ts.\mathsf{regs}; \mathsf{time} : \underline{ts_\mathsf{r}}.\mathsf{time} \rangle$.

Again, we identify the second execution's first iteration by applying Theorem B.6.11 to
$(r \coloneqq \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_\mathsf{b}}, ts, mmts_\omega$
$\xrightarrow{tr/\overrightarrow{c_\mathsf{b}}}{}^{*}_{\delta}\ (\mathtt{break}) :: \underline{\overrightarrow{s_\mathsf{r}}}, \overrightarrow{c_\mathsf{b}}, \underline{ts_\mathsf{r}}, \underline{mmts_\omega}$.

‡ (**FIRST-DONE**):

For this sub-case, we have:
$\exists \underline{\overrightarrow{s_2}}, \underline{ts_2}, \underline{mmts_2}, \underline{tr_1}, \underline{tr_2}, e.\ \underline{tr} = \underline{tr_1} +\!\!+ \underline{tr_2} \ \wedge$
$(r \coloneqq \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts_\mathsf{b}, mmts_\omega$
$\xrightarrow{[]}_{\delta}\ \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega \ \xrightarrow{tr_1}{}^{*}_{\delta}\ (\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [], \underline{ts_2}, \underline{mmts_2} \ \wedge$
$(\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, \overrightarrow{c_\mathsf{b}}, \underline{ts_2}, \underline{mmts_2} \ \xrightarrow{tr_2/\overrightarrow{c_\mathsf{b}}}{}^{*}_{\delta}\ (\mathtt{break}) :: \underline{\overrightarrow{s_\mathsf{r}}}, \overrightarrow{c_\mathsf{b}}, \underline{ts_\mathsf{r}}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_\mathsf{h}, \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_{(1,1)} \ \xrightarrow{tr_\mathsf{h}}{}^{*}_{\delta}\ \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}, \underline{mmts_2} \ \wedge$
$tr_\mathsf{h} \sim tr_2 +\!\!+ \underline{tr_1} \ \wedge$
$(\mathsf{STOP}((\mathtt{break}) :: \overrightarrow{s_\mathsf{r}}, []) \longrightarrow$
$\quad (\mathtt{break}) :: \overrightarrow{s_\mathsf{r}} = \overrightarrow{s_\mathsf{h}} \ \wedge \ [] = \overrightarrow{c_\mathsf{h}} \ \wedge \ ts_\mathsf{r} = ts_\mathsf{h} \ \wedge \ mmts_\omega = \underline{mmts_2} \ \wedge \ [] \sim \underline{tr}) \ \wedge$
$(\mathsf{STOP}((\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, []) \longrightarrow (\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}} = \overrightarrow{s_\mathsf{h}} \ \wedge \ [] = \overrightarrow{c_\mathsf{h}} \ \wedge \ \underline{ts_2} = ts_\mathsf{h})$.

From $\mathsf{STOP}((\mathtt{break}) :: \overrightarrow{s_\mathsf{r}}, [])$ and $\mathsf{STOP}((\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [])$, we have:
$(\mathtt{break}) :: \overrightarrow{s_\omega} = \overrightarrow{s_\mathsf{h}} = (\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}$, which is a contradiction.

‡ (**FIRST-ONGOING**):

For this sub-case, we have:
$(r \coloneqq \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts, mmts_\omega \ \xrightarrow{[]}_{\delta}\ \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega$
$\xrightarrow{tr}{}^{*}_{\delta}\ (\mathtt{break}) :: \underline{\overrightarrow{s_\mathsf{r}}}, [], \underline{ts_\mathsf{r}}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_\mathsf{h}, \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_{(1,1)} \ \xrightarrow{tr_\mathsf{h}}{}^{*}_{\delta}\ \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}, \underline{mmts_\omega} \ \wedge$
$tr_\mathsf{h} \sim tr_2 +\!\!+ \underline{tr} \ \wedge$
$(\mathsf{STOP}((\mathtt{break}) :: \overrightarrow{s_\mathsf{r}}, []) \longrightarrow$
$\quad (\mathtt{break}) :: \overrightarrow{s_\mathsf{r}} = \overrightarrow{s_\mathsf{h}} \ \wedge \ [] = \overrightarrow{c_\mathsf{h}} \ \wedge \ ts_\mathsf{r} = ts_\mathsf{h} \ \wedge \ mmts_\omega = \underline{mmts_\omega} \ \wedge \ [] \sim \underline{tr}) \ \wedge$
$(\mathsf{STOP}((\mathtt{break}) :: \underline{\overrightarrow{s_\mathsf{r}}}, []) \longrightarrow (\mathtt{break}) :: \underline{\overrightarrow{s_\mathsf{r}}} = \overrightarrow{s_\mathsf{h}} \ \wedge \ [] = \overrightarrow{c_\mathsf{h}} \ \wedge \ \underline{ts_\mathsf{r}} = ts_\mathsf{h})$.

From $\mathsf{STOP}((\mathtt{break}) :: \overrightarrow{s_\mathsf{r}}, [])$ and $\mathsf{STOP}((\mathtt{break}) :: \underline{\overrightarrow{s_\mathsf{r}}}, [])$, we have:
$(\mathtt{break}) :: \overrightarrow{s_\mathsf{r}} = (\mathtt{break}) :: \underline{\overrightarrow{s_\mathsf{r}}} = \overrightarrow{s_\mathsf{h}}$
$\wedge \ [] = \overrightarrow{c_\mathsf{h}} \ \wedge \ ts_\mathsf{r} = \underline{ts_\mathsf{r}} = ts_\mathsf{h} \ \wedge \ mmts_\omega = \underline{mmts_\omega} \ \wedge \ [] \sim \underline{tr}$.

From $ts_\mathsf{r} = \underline{ts_\mathsf{r}}$, we have:
$ts_\omega = \underline{ts_\omega}$.

We prove the goals with $tr_x = tr, \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = ts_\omega$ as follows:

(1) $[\texttt{loop}\ r\ e\ ((r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts \xrightarrow{tr}{}^{*}_{\delta} (\texttt{break}) :: \overrightarrow{s_\mathsf{r}}, \overrightarrow{c_\mathsf{b}}, ts_\mathsf{r}, mmts_\omega \xrightarrow{[]}_{\delta} [], [], ts_\mathsf{r}, mmts_\omega$

(2) From $[] \sim \underline{tr}$, we have: $tr \sim tr +\!\!+ \underline{tr}$

(3) $[] = [] \wedge [] = [] \wedge ts_\omega = ts_\omega \wedge mmts_\omega = mmts_\omega \wedge [] \sim \underline{tr}$

(4) $[] = [] \wedge [] = [] \wedge ts_\omega = ts_\omega$

† (**LOOP-ONGOING**):

For this sub-case, we have:
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_\mathsf{b}}, ts_\mathsf{b}, mmts_\omega \xrightarrow{\underline{tr}/\overrightarrow{c_\mathsf{b}}}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

Again, we identify the second execution's first iteration by applying Theorem B.6.11 to
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_\mathsf{b}}, ts_\mathsf{b}, mmts_\omega \xrightarrow{\underline{tr}/\overrightarrow{c_\mathsf{b}}}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

‡ (**FIRST-DONE**):

For this sub-case, we have:
$\exists \underline{\overrightarrow{s_2}}, \underline{ts_2}, \underline{mmts_2}, \underline{tr_1}, \underline{tr_2}, e.\ \underline{tr} = \underline{tr_1} +\!\!+ \underline{tr_2} \wedge$
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts_\mathsf{b}, mmts_\omega$
$\xrightarrow{[]}_{\delta} \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega \xrightarrow{tr_1}{}^{*}_{\delta} (\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [], \underline{ts_2}, \underline{mmts_2} \wedge$
$(\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, \overrightarrow{c_\mathsf{b}}, \underline{ts_2}, \underline{mmts_2} \xrightarrow{\underline{tr_2}/\overrightarrow{c_\mathsf{b}}}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_\mathsf{h}, \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_{(1,1)} \xrightarrow{tr_\mathsf{h}}{}^{*}_{\delta} \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}, \underline{mmts_2} \wedge$
$tr_\mathsf{h} \sim tr_2 +\!\!+ \underline{tr_1} \wedge$
$(\mathsf{STOP}((\texttt{break}) :: \overrightarrow{s_\mathsf{r}}, []) \longrightarrow$
$(\texttt{break}) :: \overrightarrow{s_\mathsf{r}} = \overrightarrow{s_\mathsf{h}} \wedge [] = \overrightarrow{c_\mathsf{h}} \wedge ts_\mathsf{r} = ts_\mathsf{h} \wedge mmts_\omega = \underline{mmts_2} \wedge [] \sim \underline{tr}) \wedge$
$(\mathsf{STOP}((\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, []) \longrightarrow (\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}} = \overrightarrow{s_\mathsf{h}} \wedge [] = \overrightarrow{c_\mathsf{h}} \wedge \underline{ts_2} = ts_\mathsf{h})$.

From $\mathsf{STOP}((\texttt{break}) :: \overrightarrow{s_\mathsf{r}}, [])$ and $\mathsf{STOP}((\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [])$, we have:
$(\texttt{break}) :: \overrightarrow{s_\omega} = \overrightarrow{s_\mathsf{h}} = (\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}$, which is a contradiction.

‡ (**FIRST-ONGOING**):

For this sub-case, we have:
$\exists \underline{\overrightarrow{c_\mathsf{pfx}}}.\ \underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_\mathsf{pfx}}} +\!\!+ \overrightarrow{c_\mathsf{b}} \wedge$
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts, mmts_\omega$
$\xrightarrow{[]}_{\delta} \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega \xrightarrow{tr}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\mathsf{pfx}}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_\mathsf{h}, \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_{(1,1)} \xrightarrow{tr_\mathsf{h}}{}^{*}_{\delta} \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}, \underline{mmts_\omega} \wedge$
$tr_\mathsf{h} \sim tr_2 +\!\!+ \underline{tr} \wedge$

$(\mathsf{STOP}((\texttt{break}) :: \vec{s_\mathsf{r}}, []) \longrightarrow$
$\quad (\texttt{break}) :: \vec{s_\mathsf{r}} = \vec{s_\mathsf{h}} \wedge [] = \vec{c_\mathsf{h}} \wedge ts_\mathsf{r} = ts_\mathsf{h} \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}) \wedge$
$(\mathsf{STOP}(\underline{\vec{s_\omega}}, \underline{\vec{c_\mathsf{pfx}}}) \longrightarrow \underline{\vec{s_\omega}} = \vec{s_\mathsf{h}} \wedge \underline{\vec{c_\mathsf{pfx}}} = \vec{c_\mathsf{h}} \wedge \underline{ts_\omega} = ts_\mathsf{h})$.

From $\mathsf{STOP}((\texttt{break}) :: \vec{s_\mathsf{r}}, [])$, we have:
$(\texttt{break}) :: \vec{s_\mathsf{r}} = \vec{s_\mathsf{h}}$
$\wedge [] = \vec{c_\mathsf{h}} \wedge ts_\mathsf{r} = ts_\mathsf{h} \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}$.

We prove the goals with $tr_x = tr, \vec{s_x} = [], \vec{c_x} = [], ts_x = ts_\omega$ as follows:

(1) $[\texttt{loop}\ r\ e\ ((r := \mathsf{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \vec{s})], [], ts, mmts$
$\xrightarrow{tr}{}^*_\delta\ (\texttt{break}) :: \vec{s_\mathsf{r}}, \vec{c_\mathsf{b}}, ts_\mathsf{r}, mmts_\omega \xrightarrow{[]}_\delta [], [], ts_\mathsf{r}, mmts_\omega$

(2) From $[] \sim \underline{tr}$, we have: $tr \sim tr +\!\!+ \underline{tr}$

(3) $[] = [] \wedge [] = [] \wedge ts_\omega = ts_\omega \wedge mmts_\omega = mmts_\omega \wedge [] \sim \underline{tr}$

(4) From $\underline{\vec{c_\omega}} = \underline{\vec{c_\mathsf{pfx}}} +\!\!+ \vec{c_\mathsf{b}}$, we have: $\neg\mathsf{STOP}(\underline{\vec{s_\omega}},)$

· (**CHKPT-REPLAY**):
For this sub-case, we have:
$(r := \mathsf{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, [], ts_1, mmts_1$
$\xrightarrow{[]}_\delta \vec{s}, [], ts_{(1,1)}, mmts_1$
$\xrightarrow{tr_2}{}^*_\delta (\texttt{break}) :: \vec{s_\mathsf{r}}, [], ts_\mathsf{r}, mmts_\omega$.

From Theorem B.6.7, we have:
$mmts_1|_{mids^\mathsf{c}} = mmts_\omega|_{mids^\mathsf{c}}$.

From NIN, we have:
$mmts_1[mid] = mmts_\omega[mid]$.

From Theorem B.6.14, we have: $ts_\mathsf{b}.\mathsf{time} \leq ts_1.\mathsf{time}$.

From $ts_1.\mathsf{time} < mmts_1[mid].\mathsf{time} = mmts_\omega[mid].\mathsf{time}$, by **CHKPT-REPLAY**, we have:
$(r := \mathsf{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, \vec{c_\mathsf{b}}, ts_\mathsf{b}, mmts_\omega \xrightarrow{[]}_\delta \vec{s}, \vec{c_\mathsf{b}}, \underline{ts_1}, mmts_\omega$.

From $ts_{(1,1)}.\mathsf{regs}(r) = mmts_1[mid].\mathsf{val} = mmts_\omega[mid].\mathsf{val} = \underline{ts_1}.\mathsf{regs}(r)$
and $ts_{(1,1)}.\mathsf{time} = mmts_1[mid].\mathsf{time} = mmts_\omega[mid].\mathsf{time} = \underline{ts_1}.\mathsf{time}$, we have:
$ts_{(1,1)} = \underline{ts_1}$.

We apply Theorem B.6.10 to to EXB2's later transitions,
$(r := \mathsf{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, \vec{c_\mathsf{b}}, ts_\mathsf{b}, mmts_\omega \xrightarrow{tr}{}^*_\delta \underline{\vec{s_\omega}}, \underline{\vec{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$,
and do case analysis on the lemma's conclusion:

† (**LOOP-DONE**):
For this sub-case, we have:
$(r := \mathsf{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, \vec{c_\mathsf{b}}, ts, mmts_\omega$
$\xrightarrow{[]}_\delta \vec{s}, \vec{c_\mathsf{b}}, ts_{(1,1)}, mmts_\omega$

$\xrightarrow{tr/\vec{c_b}}{}^{*}_{\delta}\ (\texttt{break}) :: \vec{s_r}, \vec{c_b}, \underline{ts_r}, \underline{mmts_\omega}\ \wedge$
$\vec{s_\omega} = []\ \wedge\ \vec{c_\omega} = []\ \wedge\ \underline{ts_\omega} = \langle \mathsf{regs} : ts.\mathsf{regs}; \mathsf{time} : \underline{ts_r}.\mathsf{time} \rangle$.

Again, we identify the second execution's first iteration by applying Theorem B.6.11 to
$(r \coloneqq \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, \vec{c_b}, ts, mmts_\omega$
$\xrightarrow{tr/\vec{c_b}}{}^{*}_{\delta}\ (\texttt{break}) :: \vec{s_r}, \vec{c_b}, \underline{ts_r}, \underline{mmts_\omega}$.

‡ (**FIRST-DONE**):

For this sub-case, we have:
$\exists \underline{\vec{s_2}}, \underline{ts_2}, \underline{mmts_2}, \underline{tr_1}, \underline{tr_2}, e.\ \underline{tr} = \underline{tr_1} +\!\!+ \underline{tr_2}\ \wedge$
$(r \coloneqq \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, [], ts_b, mmts_\omega$
$\xrightarrow{[]}_{\delta}\ \vec{s}, [], ts_{(1,1)}, mmts_\omega\ \xrightarrow{tr_1}{}^{*}_{\delta}\ (\texttt{continue}\ e) :: \underline{\vec{s_2}}, [], \underline{ts_2}, \underline{mmts_2}\ \wedge$
$(\texttt{continue}\ e) :: \underline{\vec{s_2}}, \vec{c_b}, \underline{ts_2}, \underline{mmts_2}\ \xrightarrow{tr_2/\vec{c_b}}{}^{*}_{\delta}\ (\texttt{break}) :: \vec{s_r}, \vec{c_b}, \underline{ts_r}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_h, \vec{s_h}, \vec{c_h}, ts_h.$
$\vec{s}, [], ts_{(1,1)}, mmts_1\ \xrightarrow{tr_h}{}^{*}_{\delta}\ \vec{s_h}, \vec{c_h}, ts_h, \underline{mmts_2}\ \wedge$
$tr_h \sim tr_2 +\!\!+ \underline{tr_1}\ \wedge$
$(\mathsf{STOP}((\texttt{break}) :: \vec{s_r}, []) \longrightarrow$
$\quad (\texttt{break}) :: \vec{s_r} = \vec{s_h}\ \wedge\ [] = \vec{c_h}\ \wedge\ ts_r = ts_h\ \wedge\ mmts_\omega = \underline{mmts_2}\ \wedge\ [] \sim \underline{tr})\ \wedge$
$(\mathsf{STOP}((\texttt{continue}\ e) :: \underline{\vec{s_2}}, []) \longrightarrow (\texttt{continue}\ e) :: \underline{\vec{s_2}} = \vec{s_h}\ \wedge\ [] = \vec{c_h}\ \wedge\ \underline{ts_2} = ts_h)$.

From $\mathsf{STOP}((\texttt{break}) :: \vec{s_r}, [])$ and $\mathsf{STOP}((\texttt{continue}\ e) :: \underline{\vec{s_2}}, [])$, we have:
$(\texttt{break}) :: \vec{s_\omega} = \vec{s_h} = (\texttt{continue}\ e) :: \underline{\vec{s_2}}$, which is a contradiction.

‡ (**FIRST-ONGOING**):

For this sub-case, we have:
$(r \coloneqq \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \vec{s}, [], ts, mmts_\omega$
$\xrightarrow{[]}_{\delta}\ \vec{s}, [], ts_{(1,1)}, mmts_\omega\ \xrightarrow{tr}{}^{*}_{\delta}\ (\texttt{break}) :: \underline{\vec{s_r}}, [], \underline{ts_r}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_h, \vec{s_h}, \vec{c_h}, ts_h.$
$\vec{s}, [], ts_{(1,1)}, mmts_1\ \xrightarrow{tr_h}{}^{*}_{\delta}\ \vec{s_h}, \vec{c_h}, ts_h, \underline{mmts_\omega}\ \wedge$
$tr_h \sim tr_2 +\!\!+ \underline{tr}\ \wedge$
$(\mathsf{STOP}((\texttt{break}) :: \vec{s_r}, []) \longrightarrow$
$\quad (\texttt{break}) :: \vec{s_r} = \vec{s_h}\ \wedge\ [] = \vec{c_h}\ \wedge\ ts_r = ts_h\ \wedge\ mmts_\omega = \underline{mmts_\omega}\ \wedge\ [] \sim \underline{tr})\ \wedge$
$(\mathsf{STOP}((\texttt{break}) :: \underline{\vec{s_r}}, []) \longrightarrow (\texttt{break}) :: \underline{\vec{s_r}} = \vec{s_h}\ \wedge\ [] = \vec{c_h}\ \wedge\ \underline{ts_r} = ts_h)$.

From $\mathsf{STOP}((\texttt{break}) :: \vec{s_r}, [])$ and $\mathsf{STOP}((\texttt{break}) :: \underline{\vec{s_r}}, [])$, we have:
$(\texttt{break}) :: \vec{s_r} = (\texttt{break}) :: \underline{\vec{s_r}} = \vec{s_h}$
$\wedge\ [] = \vec{c_h}\ \wedge\ ts_r = \underline{ts_r} = ts_h\ \wedge\ mmts_\omega = \underline{mmts_\omega}\ \wedge\ [] \sim \underline{tr}$.

From $ts_r = \underline{ts_r}$, we have:
$ts_\omega = \underline{ts_\omega}$.

We prove the goals with $tr_x = tr, \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = ts_\omega$ as follows:

(1) $[\texttt{loop}\ r\ e\ ((r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts$
$\xrightarrow{tr}{}^{*}_{\delta}\ (\texttt{break}) :: \overrightarrow{s_\mathsf{r}}, \overrightarrow{c_\mathsf{b}}, ts_\mathsf{r}, mmts_\omega \xrightarrow{[]}_{\delta}\ [], [], ts_\mathsf{r}, mmts_\omega$

(2) From $[] \sim \underline{tr}$, we have: $tr \sim tr +\!\!+ \underline{tr}$

(3) $[] = [] \ \wedge\ [] = [] \ \wedge\ ts_\omega = ts_\omega \ \wedge\ mmts_\omega = mmts_\omega \ \wedge\ [] \sim \underline{tr}$

(4) $[] = [] \ \wedge\ [] = [] \ \wedge\ ts_\omega = ts_\omega$

† (**LOOP-ONGOING**):

For this sub-case, we have:
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_\mathsf{b}}, ts_\mathsf{b}, mmts_\omega \xrightarrow{\underline{tr}/\overrightarrow{c_\mathsf{b}}}{}^{*}_{\delta}\ \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

Again, we identify the second execution's first iteration by applying Theorem B.6.11 to
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_\mathsf{b}}, ts_\mathsf{b}, mmts_\omega \xrightarrow{\underline{tr}/\overrightarrow{c_\mathsf{b}}}{}^{*}_{\delta}\ \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

‡ (**FIRST-DONE**):

For this sub-case, we have:
$\exists \underline{\overrightarrow{s_2}}, \underline{ts_2}, \underline{mmts_2}, \underline{tr_1}, \underline{tr_2}, e.\ \underline{tr} = \underline{tr_1} +\!\!+ \underline{tr_2} \ \wedge$
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts_\mathsf{b}, mmts_\omega$
$\xrightarrow{[]}_{\delta}\ \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega \xrightarrow{tr_1}{}^{*}_{\delta}\ (\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [], \underline{ts_2}, \underline{mmts_2} \ \wedge$
$(\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, \overrightarrow{c_\mathsf{b}}, \underline{ts_2}, \underline{mmts_2} \xrightarrow{tr_2/\overrightarrow{c_\mathsf{b}}}{}^{*}_{\delta}\ \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_\mathsf{h}, \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_1 \xrightarrow{tr_\mathsf{h}}{}^{*}_{\delta}\ \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}, \underline{mmts_2} \ \wedge$
$tr_\mathsf{h} \sim tr_2 +\!\!+ \underline{tr_1} \ \wedge$
$(\mathsf{STOP}((\texttt{break}) :: \overrightarrow{s_\mathsf{r}}, []) \longrightarrow$
$\quad (\texttt{break}) :: \overrightarrow{s_\mathsf{r}} = \overrightarrow{s_\mathsf{h}} \ \wedge\ [] = \overrightarrow{c_\mathsf{h}} \ \wedge\ ts_\mathsf{r} = ts_\mathsf{h} \ \wedge\ mmts_\omega = \underline{mmts_2} \ \wedge\ [] \sim \underline{tr}) \ \wedge$
$(\mathsf{STOP}((\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, []) \longrightarrow (\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}} = \overrightarrow{s_\mathsf{h}} \ \wedge\ [] = \overrightarrow{c_\mathsf{h}} \ \wedge\ \underline{ts_2} = ts_\mathsf{h})$.

From $\mathsf{STOP}((\texttt{break}) :: \overrightarrow{s_\mathsf{r}}, [])$ and $\mathsf{STOP}((\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [])$, we have:
$(\texttt{break}) :: \overrightarrow{s_\omega} = \overrightarrow{s_\mathsf{h}} = (\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}$, which is a contradiction.

‡ (**FIRST-ONGOING**):

For this sub-case, we have:
$\exists \underline{\overrightarrow{c_\mathsf{pfx}}}.\ \underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_\mathsf{pfx}}} +\!\!+ \overrightarrow{c_\mathsf{b}} \ \wedge$
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts, mmts_\omega$
$\xrightarrow{[]}_{\delta}\ \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega \xrightarrow{tr}{}^{*}_{\delta}\ \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\mathsf{pfx}}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_\mathsf{h}, \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_1 \xrightarrow{tr_\mathsf{h}}{}^{*}_{\delta}\ \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}, \underline{mmts_\omega} \ \wedge$
$tr_\mathsf{h} \sim tr_2 +\!\!+ \underline{tr} \ \wedge$
$(\mathsf{STOP}((\texttt{break}) :: \overrightarrow{s_\mathsf{r}}, []) \longrightarrow$

$$(\texttt{break}) :: \overrightarrow{s_\mathsf{r}} = \overrightarrow{s_\mathsf{h}} \ \wedge \ [] = \overrightarrow{c_\mathsf{h}} \ \wedge \ ts_\mathsf{r} = ts_\mathsf{h} \ \wedge \ mmts_\omega = \underline{mmts_\omega} \ \wedge \ [] \sim \underline{tr}) \ \wedge$$
$$(\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\mathsf{pfx}}}) \ \longrightarrow \ \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_\mathsf{h}} \ \wedge \ \underline{\overrightarrow{c_\mathsf{pfx}}} = \overrightarrow{c_\mathsf{h}} \ \wedge \ \underline{ts_\omega} = ts_\mathsf{h}) \, .$$

From $\mathsf{STOP}((\texttt{break}) :: \overrightarrow{s_\mathsf{r}}, [])$, we have:
$(\texttt{break}) :: \overrightarrow{s_\mathsf{r}} = \overrightarrow{s_\mathsf{h}}$
$\wedge \ [] = \overrightarrow{c_\mathsf{h}} \ \wedge \ ts_\mathsf{r} = ts_\mathsf{h} \ \wedge \ mmts_\omega = \underline{mmts_\omega} \ \wedge \ [] \sim \underline{tr}$.

We prove the goals with $tr_x = tr, \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = ts_\omega$ as follows:

(1) $[\texttt{loop}\ r\ e\ ((r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts$
$\xrightarrow{tr}{}^*_\delta\ (\texttt{break}) :: \overrightarrow{s_\mathsf{r}}, \overrightarrow{c_\mathsf{b}}, ts_\mathsf{r}, mmts_\omega \xrightarrow{[]}_\delta [], [], ts_\mathsf{r}, mmts_\omega$

(2) From $[] \sim \underline{tr}$, we have: $tr \sim tr +\!\!+ \underline{tr}$

(3) $[] = [] \ \wedge \ [] = [] \ \wedge \ ts_\omega = ts_\omega \ \wedge \ mmts_\omega = mmts_\omega \ \wedge \ [] \sim \underline{tr}$

(4) From $\underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_\mathsf{pfx}}} +\!\!+ \overrightarrow{c_\mathsf{b}}$, we have: $\neg\mathsf{STOP}(\underline{\overrightarrow{s_\omega}},)$

- (**LOOP-ONGOING**):

  For this sub-case, we have:
  $(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_\mathsf{b}}, ts_\mathsf{b}, mmts \ \xrightarrow{tr/\overrightarrow{c_\mathsf{b}}}{}^*_\delta \ \overrightarrow{s_\omega}, \overrightarrow{c_\omega}, ts_\omega, mmts_\omega$.

  We identify the first execution's last iteration by applying Theorem B.6.12, we have (LAST-ITER):
  $\exists ts_1, mmts_1, tr_1, tr_2.$
  $tr = tr_1 +\!\!+ tr_2 \ \wedge$
  $(((ts_\mathsf{b}, mmts) = (ts_1, mmts_1) \ \wedge \ tr_1 = []) \ \vee$
  $(\exists e, \overrightarrow{s_\mathsf{r}}, ts_\mathsf{r}.\ (r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_\mathsf{b}}, ts_\mathsf{b}, mmts$
  $\xrightarrow{tr_1/\overrightarrow{c_\mathsf{b}}}{}^*_\delta\ (\texttt{continue}\ e) :: \overrightarrow{s_\mathsf{r}}, \overrightarrow{c_\mathsf{b}}, ts_\mathsf{r}, mmts_1 \ \wedge$
  $ts_1 = \langle \mathsf{regs} : \sigma[r \mapsto ts_\mathsf{r}.\mathsf{regs}(e)]; \mathsf{time} : ts_\mathsf{r}.\mathsf{time}\rangle))) \ \wedge$
  $\overrightarrow{c_\omega} = \overrightarrow{c_\mathsf{pfx}} +\!\!+ \overrightarrow{c_\mathsf{b}} \ \wedge$
  $(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts_1, mmts_1 \ \xrightarrow{tr_2}{}^*_\delta \ \overrightarrow{s_\omega}, \overrightarrow{c_\mathsf{pfx}}, ts_\omega, mmts_\omega$.

  We do a case analysis on the transitions $(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts_1, mmts_1$
  $\xrightarrow{tr_2}{}^*_\delta\ \overrightarrow{s_\omega}, \overrightarrow{c_\mathsf{pfx}}, ts_\omega, mmts_\omega$:

  (1) The transitions are empty;

  (2) The transitions are just **CHKPT-CALL**;

  (3) The transitions begin with **CHKPT-CALL** and then **CHKPT-RETURN**; or

  (4) The transitions begin with **CHKPT-REPLAY**.

  We prove for each case.

  - **(1), (2)** The transitions are empty or just **CHKPT-CALL**:

    In these cases, EXB1's last loop iteration does not finish its chkpt() operation for the dependent variable before the crash. As such, EXB2 recovers from EXB1's second last loop iteration, first by retrieving its dependent variable checkpointed in its memento.
    For these cases, we have $tr_2 = []$.

We do a case analysis on the part of LAST-ITER,

† N-ITER: $(ts_{\mathsf{b}}, mmts) = (ts_1, mmts_1) \ \wedge \ tr_1 = []$

† (N-1)-ITER: $\exists e, \overrightarrow{s_{\mathsf{r}}}, ts_{\mathsf{r}}.\ (r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{b}}, mmts$
$\xrightarrow{tr_1/\overrightarrow{c_{\mathsf{b}}}}{}^{*}_{\delta}\ (\texttt{continue}\ e) :: \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{r}}, mmts_1 \ \wedge$
$ts_1 = \langle \mathsf{regs} : \sigma[r \mapsto ts_{\mathsf{r}}.\mathsf{regs}(e)]; \mathsf{time} : ts_{\mathsf{r}}.\mathsf{time}\rangle$

For the N-ITER, the proof is essentially the same with the case that EX1 itself is empty transitions (the entire proof's first case).

For the (N-1)-ITER, we identify the first execution's second last iteration
by applying Theorem B.6.12 again to
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{b}}, mmts$
$\xrightarrow{tr_1/\overrightarrow{c_{\mathsf{b}}}}{}^{*}_{\delta}\ (\texttt{continue}\ e) :: \overrightarrow{s_{\mathsf{r}}}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{r}}, mmts_1.$
Then we have:
$\exists ts_0, mmts_0, tr_{(1,1)}, tr_{(1,2)}.$
$tr_1 = tr_{(1,1)} +\!\!+ tr_{(1,2)} \ \wedge$
$(((ts_{\mathsf{b}}, mmts) = (ts_0, mmts_0) \ \wedge \ tr_{(1,1)} = []) \ \vee$
$(\exists e0, \overrightarrow{s_{\mathsf{r}0}}, ts_{\mathsf{r}0}.\ (r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{b}}, mmts$
$\xrightarrow{tr_1/\overrightarrow{c_{\mathsf{b}}}}{}^{*}_{\delta}\ (\texttt{continue}\ e) :: \overrightarrow{s_{\mathsf{r}0}}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{r}0}, mmts_0 \ \wedge$
$ts_0 = \langle \mathsf{regs} : \sigma[r \mapsto ts_{\mathsf{r}0}.\mathsf{regs}(e)]; \mathsf{time} : ts_{\mathsf{r}0}.\mathsf{time}\rangle))) \ \wedge$
$\overrightarrow{c_{\mathsf{b}}} = [] +\!\!+ \overrightarrow{c_{\mathsf{b}}} \ \wedge$
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts_0, mmts_0$
$\xrightarrow{tr_{(1,2)}}{}^{*}_{\delta}\ (\texttt{continue}\ e) :: \overrightarrow{s_{\mathsf{r}}}, [], ts_{\mathsf{r}}, mmts_1\ .$

From above, we have:
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts_0, mmts_0$
$\xrightarrow{tr_{(1,2)}}{}^{*}_{\delta}\ (\texttt{continue}\ e) :: \overrightarrow{s_{\mathsf{r}}}, [], ts_{\mathsf{r}}, mmts_1\ \xrightarrow{tr_2}{}^{*}_{\delta}\ \overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}, ts_\omega, mmts_\omega\ .$

From here, the proof is essentially the same with the following cases for **(3)** and **(4)**.

· **(3)** The transitions begin with **CHKPT-CALL** and then **CHKPT-RETURN**:

For this sub-case, we have:
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts_1, mmts_1$
$\xrightarrow{[]}_{\delta}\ [\texttt{return}\ r], \overrightarrow{c_{(1,\mathsf{chk})}}, ts_1, mmts_1$
$\xrightarrow{[]}_{\delta}\ \overrightarrow{s}, [], ts_{(1,1)}, mmts_{(1,1)}$
$\xrightarrow{tr_2}{}^{*}_{\delta}\ \overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}, ts_\omega, mmts_\omega\ .$

From Theorem B.6.7, we have:
$mmts_{(1,1)}|_{mids^{\mathsf{c}}} = mmts_\omega|_{mids^{\mathsf{c}}}\ .$

From NIN, we have:
$mmts_{(1,1)}[mid] = mmts_\omega[mid]\ .$

From Theorem B.6.14, we have:
$ts_{\mathsf{b}}.\mathsf{time} \leq ts_1.\mathsf{time}$.

From $ts_1.\mathsf{time} < mmts_{(1,1)}[mid].\mathsf{time} = mmts_\omega[mid].\mathsf{time}$, by **CHKPT-REPLAY**, we have:
$(r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{b}}, mmts_\omega \xrightarrow{[]}_\delta \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_1}, mmts_\omega$.

From $ts_{(1,1)}.\mathsf{regs}(r) = mmts_{(1,1)}[mid].\mathsf{val} = mmts_\omega[mid].\mathsf{val} = \underline{ts_1}.\mathsf{regs}(r)$
and $ts_{(1,1)}.\mathsf{time} = mmts_{(1,1)}[mid].\mathsf{time} = mmts_\omega[mid].\mathsf{time} = \underline{ts_1}.\mathsf{time}$, we have:
$ts_{(1,1)} = \underline{ts_1}$.

We apply Theorem B.6.10 to EXB2's later transitions,
$(r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{b}}, mmts_\omega \xrightarrow{tr}{}^*_\delta \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$,
and do case analysis on the lemma's conclusion:

† (**LOOP-DONE**):
For this sub-case, we have:
$(r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts, mmts_\omega$
$\xrightarrow{[]}_\delta \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_\omega$
$\xrightarrow{tr/\overrightarrow{c_{\mathsf{b}}}}{}^*_\delta (\mathtt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_\omega} \ \wedge$
$\underline{\overrightarrow{s_\omega}} = [] \ \wedge \ \underline{\overrightarrow{c_\omega}} = [] \ \wedge \ \underline{ts_\omega} = \langle \mathsf{regs} : ts.\mathsf{regs}; \mathsf{time} : \underline{ts_{\mathsf{r}}}.\mathsf{time} \rangle$.

Again, we identify the second execution's first iteration by applying Theorem B.6.11 to
$(r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts, mmts_\omega$
$\xrightarrow{tr/\overrightarrow{c_{\mathsf{b}}}}{}^*_\delta (\mathtt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_\omega}$.

‡ (**FIRST-DONE**):
For this sub-case, we have:
$\exists \underline{\overrightarrow{s_2}}, \underline{ts_2}, \underline{mmts_2}, \underline{tr_1}, \underline{tr_2}, e.\ \underline{tr} = \underline{tr_1} +\!\!+ \underline{tr_2} \ \wedge$
$(r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts_{\mathsf{b}}, mmts_\omega$
$\xrightarrow{[]}_\delta \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega \xrightarrow{tr_1}{}^*_\delta (\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [], \underline{ts_2}, \underline{mmts_2} \ \wedge$
$(\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_2}, \underline{mmts_2} \xrightarrow{tr_2/\overrightarrow{c_{\mathsf{b}}}}{}^*_\delta (\mathtt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_{\mathsf{h}}, \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_{(1,1)} \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}, \underline{mmts_2} \ \wedge$
$tr_{\mathsf{h}} \sim tr_2 +\!\!+ \underline{tr_1} \ \wedge$
$(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_{\mathsf{h}}} \wedge \overrightarrow{c_{\mathsf{pfx}}} = \overrightarrow{c_{\mathsf{h}}} \wedge ts_\omega = ts_{\mathsf{h}} \wedge mmts_\omega = \underline{mmts_2} \wedge [] \sim \underline{tr}) \wedge$
$(\mathsf{STOP}((\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, []) \longrightarrow (\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}} = \overrightarrow{s_{\mathsf{h}}} \wedge [] = \overrightarrow{c_{\mathsf{h}}} \wedge \underline{ts_2} = ts_{\mathsf{h}})$.

From $\mathsf{STOP}((\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [])$, we have:
$(\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}} = \overrightarrow{s_{\mathsf{h}}} \ \wedge \ [] = \overrightarrow{c_{\mathsf{h}}} \ \wedge \ \underline{ts_2} = ts_{\mathsf{h}}$.

From Theorem B.6.5, we have:

$\overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_{(1,1)} \xrightarrow{tr_{\mathsf{h}}}{}^{*}_{\delta} (\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_2}, \underline{mmts_2}$.

We prove the goals with $tr_x = tr_1 +\!\!+ tr_{\mathsf{h}} +\!\!+ \underline{tr_2}, \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = \underline{ts_\omega}$ as follows:

(1) $[\texttt{loop}\ r\ e\ ((r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts$
$\xrightarrow{tr_1}{}^{*}_{\delta} \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_{(1,1)} \xrightarrow{tr_{\mathsf{h}}}{}^{*}_{\delta} (\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_2}, \underline{mmts_2}$
$\xrightarrow{tr_2}{}^{*}_{\delta} (\texttt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_\omega} \xrightarrow{[]}_{\delta} [], [], \underline{ts_\omega}, \underline{mmts_\omega}$

(2) From $tr = tr_1 +\!\!+ tr_2$ and $tr_{\mathsf{h}} \sim tr_2 +\!\!+ \underline{tr_1}$, we have: $tr_1 +\!\!+ tr_{\mathsf{h}} +\!\!+ \underline{tr_2} \sim tr +\!\!+ \underline{tr}$

(3) From $\overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}}$, we have: $\neg\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega})$

(4) $[] = [] \wedge [] = [] \wedge \underline{ts_\omega} = \underline{ts_\omega}$

‡ (**FIRST-ONGOING**):

For this sub-case, we have:
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts, mmts_\omega$
$\xrightarrow{[]}_{\delta} \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega \xrightarrow{tr}{}^{*}_{\delta} (\texttt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, [], \underline{ts_{\mathsf{r}}}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_{\mathsf{h}}, \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_{(1,1)} \xrightarrow{tr_{\mathsf{h}}}{}^{*}_{\delta} \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}, \underline{mmts_\omega} \wedge$
$tr_{\mathsf{h}} \sim tr_2 +\!\!+ \underline{tr} \wedge$
$(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_{\mathsf{h}}} \wedge \overrightarrow{c_{\mathsf{pfx}}} = \overrightarrow{c_{\mathsf{h}}} \wedge ts_\omega = ts_{\mathsf{h}} \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}) \wedge$
$(\mathsf{STOP}((\texttt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, []) \longrightarrow (\texttt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}} = \overrightarrow{s_{\mathsf{h}}} \wedge [] = \overrightarrow{c_{\mathsf{h}}} \wedge \underline{ts_{\mathsf{r}}} = ts_{\mathsf{h}})$.

From $\mathsf{STOP}((\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [])$, we have:
$(\texttt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}} = \overrightarrow{s_{\mathsf{h}}} \wedge [] = \overrightarrow{c_{\mathsf{h}}} \wedge \underline{ts_{\mathsf{r}}} = ts_{\mathsf{h}}$.

From Theorem B.6.5, we have:
$\overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_{(1,1)} \xrightarrow{tr_{\mathsf{h}}}{}^{*}_{\delta} (\texttt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_\omega}$.

We prove the goals with $tr_x = tr_1 +\!\!+ tr_{\mathsf{h}}, \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = \underline{ts_\omega}$ as follows:

(1) $[\texttt{loop}\ r\ e\ ((r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts$
$\xrightarrow{tr_1}{}^{*}_{\delta} \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_{(1,1)} \xrightarrow{tr_{\mathsf{h}}}{}^{*}_{\delta} (\texttt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_\omega} \xrightarrow{[]}_{\delta} [], [], \underline{ts_\omega}, \underline{mmts_\omega}$

(2) From $tr = tr_1 +\!\!+ tr_2$ and $tr_{\mathsf{h}} \sim tr_2 +\!\!+ \underline{tr}$, we have: $tr_1 +\!\!+ tr_{\mathsf{h}} +\!\!+ \underline{tr} \sim tr +\!\!+ \underline{tr}$

(3) From $\overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}}$, we have: $\neg\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega})$

(4) $[] = [] \wedge [] = [] \wedge \underline{ts_\omega} = \underline{ts_\omega}$

† (**LOOP-ONGOING**):

For this sub-case, we have:
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{b}}, mmts_\omega \xrightarrow{tr/\overrightarrow{c_{\mathsf{b}}}}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

Again, we identify the second execution's first iteration by applying Theorem B.6.11 to
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{b}}, mmts_\omega \xrightarrow{tr/\overrightarrow{c_{\mathsf{b}}}}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

‡ (**FIRST-DONE**):

For this sub-case, we have:

$\exists \overrightarrow{\underline{s_2}}, \underline{ts_2}, \underline{mmts_2}, \underline{tr_1}, \underline{tr_2}, e.\ \underline{tr} = \underline{tr_1} +\!\!+ \underline{tr_2} \ \wedge$
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts_{\mathsf{b}}, mmts_\omega$
$\xrightarrow{[]}_\delta \ \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega \ \xrightarrow{tr_1}{}^*_\delta \ (\texttt{continue}\ e) :: \overrightarrow{\underline{s_2}}, [], \underline{ts_2}, \underline{mmts_2} \ \wedge$
$(\texttt{continue}\ e) :: \overrightarrow{\underline{s_2}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_2}, \underline{mmts_2} \ \xrightarrow{tr_2/\overrightarrow{c_{\mathsf{b}}}}{}^*_\delta \ \overrightarrow{\underline{s_\omega}}, \overrightarrow{\underline{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_{\mathsf{h}}, \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_{(1,1)} \ \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \ \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}, \underline{mmts_2} \ \wedge$
$tr_{\mathsf{h}} \sim tr_2 +\!\!+ \underline{tr_1} \ \wedge$
$(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_{\mathsf{h}}} \wedge \overrightarrow{c_{\mathsf{pfx}}} = \overrightarrow{c_{\mathsf{h}}} \wedge ts_\omega = ts_{\mathsf{h}} \wedge mmts_\omega = \underline{mmts_2} \wedge [] \sim \underline{tr}) \wedge$
$(\mathsf{STOP}((\texttt{continue}\ e) :: \overrightarrow{\underline{s_2}}, []) \longrightarrow (\texttt{continue}\ e) :: \overrightarrow{\underline{s_2}} = \overrightarrow{s_{\mathsf{h}}} \wedge [] = \overrightarrow{c_{\mathsf{h}}} \wedge \underline{ts_2} = ts_{\mathsf{h}})$.

From $\mathsf{STOP}((\texttt{continue}\ e) :: \overrightarrow{\underline{s_2}}, [])$, we have:
$(\texttt{continue}\ e) :: \overrightarrow{\underline{s_2}} = \overrightarrow{s_{\mathsf{h}}} \wedge [] = \overrightarrow{c_{\mathsf{h}}} \wedge \underline{ts_2} = ts_{\mathsf{h}}$.

From Theorem B.6.5, we have:
$\overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_{(1,1)} \ \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \ (\texttt{continue}\ e) :: \overrightarrow{\underline{s_2}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_2}, \underline{mmts_2}$.

We prove the goals with $tr_x = tr_1 +\!\!+ tr_{\mathsf{h}} +\!\!+ \underline{tr_2}, \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = \underline{ts_\omega}$ as follows:

(1) $[\texttt{loop}\ r\ e\ ((r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts$
$\xrightarrow{tr_1}{}^*_\delta \ \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_{(1,1)} \ \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \ (\texttt{continue}\ e) :: \overrightarrow{\underline{s_2}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_2}, \underline{mmts_2}$
$\xrightarrow{tr_2}{}^*_\delta \ \overrightarrow{\underline{s_\omega}}, \overrightarrow{\underline{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

(2) From $tr = tr_1 +\!\!+ tr_2$ and $tr_{\mathsf{h}} \sim tr_2 +\!\!+ \underline{tr_1}$, we have: $tr_1 +\!\!+ tr_{\mathsf{h}} +\!\!+ \underline{tr_2} \sim tr +\!\!+ \underline{tr}$

(3) From $\overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}}$, we have: $\neg\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega})$

(4) For some $\overrightarrow{\underline{c_{\mathsf{pfx}}}}$, we have: $\overrightarrow{\underline{c_\omega}} = \overrightarrow{\underline{c_{\mathsf{pfx}}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}}$. Therefore, $\neg\mathsf{STOP}(\overrightarrow{\underline{s_\omega}}, \overrightarrow{\underline{c_\omega}})$

‡ (**FIRST-ONGOING**):

For this sub-case, we have:
$\exists \overrightarrow{\underline{c_{\mathsf{pfx}}}}.\ \overrightarrow{\underline{c_\omega}} = \overrightarrow{\underline{c_{\mathsf{pfx}}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}} \ \wedge$
$(r := \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts, mmts_\omega$
$\xrightarrow{[]}_\delta \ \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega \ \xrightarrow{tr}{}^*_\delta \ \overrightarrow{\underline{s_\omega}}, \overrightarrow{\underline{c_{\mathsf{pfx}}}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_{\mathsf{h}}, \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_{(1,1)} \ \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \ \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}, \underline{mmts_\omega} \ \wedge$
$tr_{\mathsf{h}} \sim tr_2 +\!\!+ \underline{tr} \ \wedge$
$(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_{\mathsf{h}}} \wedge \overrightarrow{c_{\mathsf{pfx}}} = \overrightarrow{c_{\mathsf{h}}} \wedge ts_\omega = ts_{\mathsf{h}} \wedge mmts_\omega = \underline{mmts_2} \wedge [] \sim \underline{tr}) \wedge$
$(\mathsf{STOP}(\overrightarrow{\underline{s_\omega}}, \overrightarrow{\underline{c_{\mathsf{pfx}}}}) \longrightarrow \overrightarrow{\underline{s_\omega}} = \overrightarrow{s_{\mathsf{h}}} \wedge \overrightarrow{\underline{c_{\mathsf{pfx}}}} = \overrightarrow{c_{\mathsf{h}}} \wedge \underline{ts_\omega} = ts_{\mathsf{h}})$.

From Theorem B.6.5, we have:
$\overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_{(1,1)} \ \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \ \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{h}}, \underline{mmts_\omega}$.

We prove the goals with $tr_x = tr_1 +\!\!+ tr_{\mathsf{h}}, \overrightarrow{s_x} = \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_x} = \overrightarrow{c_{\mathsf{h}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}}, ts_x = ts_{\mathsf{h}}$ as follows:

(1) $[\texttt{loop}\ r\ e\ ((r := \mathsf{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts \xrightarrow{tr_1}{}^{*}_{\delta} \overrightarrow{s}, \overrightarrow{c_b}, ts_{(1,1)}, mmts_{(1,1)} \xrightarrow{tr_h}{}^{*}_{\delta} \overrightarrow{s_h}, \overrightarrow{c_h} +\!\!+ \overrightarrow{c_b}, ts_h, \underline{mmts_\omega}$

(2) From $tr = tr_1 +\!\!+ tr_2$ and $tr_h \sim tr_2 +\!\!+ \underline{tr}$, we have: $tr_1 +\!\!+ tr_h +\!\!+ \underline{tr} \sim tr +\!\!+ \underline{tr}$

(3) From $\overrightarrow{c_\omega} = \overrightarrow{c_{pfx}} +\!\!+ \overrightarrow{c_b}$, we have: $\neg\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega})$

(4) From $\underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_{pfx}}} +\!\!+ \overrightarrow{c_b}$, we have: $\neg\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}})$

· **(4)** The transitions begin with **CHKPT-REPLAY**:

For this sub-case, we have:
$(r := \mathsf{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts_1, mmts_1 \xrightarrow{[]}_{\delta} \overrightarrow{s}, [], ts_{(1,1)}, mmts_1 \xrightarrow{tr_2}{}^{*}_{\delta} \overrightarrow{s_\omega}, \overrightarrow{c_{pfx}}, ts_\omega, mmts_\omega$.

From Theorem B.6.7, we have:
$mmts_1|_{mids^c} = mmts_\omega|_{mids^c}$.

From NIN, we have:
$mmts_1[mid] = mmts_\omega[mid]$.

From Theorem B.6.14, we have:
$ts_b.\mathsf{time} \leq ts_1.\mathsf{time}$.

From $ts_1.\mathsf{time} < mmts_1[mid].\mathsf{time} = mmts_\omega[mid].\mathsf{time}$, by **CHKPT-REPLAY**, we have:
$(r := \mathsf{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_b}, ts_b, mmts_\omega \xrightarrow{[]}_{\delta} \overrightarrow{s}, \overrightarrow{c_b}, \underline{ts_1}, mmts_\omega$.

From $ts_{(1,1)}.\mathsf{regs}(r) = mmts_1[mid].\mathsf{val} = mmts_\omega[mid].\mathsf{val} = \underline{ts_1}.\mathsf{regs}(r)$
and $ts_{(1,1)}.\mathsf{time} = mmts_1[mid].\mathsf{time} = mmts_\omega[mid].\mathsf{time} = \underline{ts_1}.\mathsf{time}$, we have:
$ts_{(1,1)} = \underline{ts_1}$.

We apply Theorem B.6.10 to EXB2's later transitions,
$(r := \mathsf{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_b}, ts_b, mmts_\omega \xrightarrow{tr}{}^{*}_{\delta} \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$,
and do case analysis on the lemma's conclusion:

† **(LOOP-DONE)**:
For this sub-case, we have:
$(r := \mathsf{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_b}, ts, mmts_\omega$
$\xrightarrow{[]}_{\delta} \overrightarrow{s}, \overrightarrow{c_b}, ts_{(1,1)}, mmts_\omega$
$\xrightarrow{tr/\overrightarrow{c_b}}{}^{*}_{\delta} (\texttt{break}) :: \underline{\overrightarrow{s_r}}, \overrightarrow{c_b}, \underline{ts_r}, \underline{mmts_\omega} \ \wedge$
$\underline{\overrightarrow{s_\omega}} = [] \ \wedge\ \underline{\overrightarrow{c_\omega}} = [] \ \wedge\ \underline{ts_\omega} = \langle \mathsf{regs} : ts.\mathsf{regs}; \mathsf{time} : \underline{ts_r}.\mathsf{time} \rangle$.

Again, we identify the second execution's first iteration by applying Theorem B.6.11 to
$(r := \mathsf{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_b}, ts, mmts_\omega$
$\xrightarrow{tr/\overrightarrow{c_b}}{}^{*}_{\delta} (\texttt{break}) :: \underline{\overrightarrow{s_r}}, \overrightarrow{c_b}, \underline{ts_r}, \underline{mmts_\omega}$.

‡ **(FIRST-DONE)**:
For this sub-case, we have:

$\exists \underline{\overrightarrow{s_2}}, \underline{ts_2}, \underline{mmts_2}, \underline{tr_1}, \underline{tr_2}, e.\ \underline{tr} = \underline{tr_1} +\!\!+ \underline{tr_2} \ \wedge$
$(r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts_{\mathsf{b}}, mmts_\omega$
$\xrightarrow{[]}_\delta \ \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega \ \xrightarrow{tr_1}{}^*_\delta \ (\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [], \underline{ts_2}, \underline{mmts_2} \ \wedge$
$(\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_2}, \underline{mmts_2} \ \xrightarrow{tr_2/\overrightarrow{c_{\mathsf{b}}}}{}^*_\delta \ (\mathtt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_\omega}\ .$

From IH, we have:
$\exists tr_{\mathsf{h}}, \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_1 \ \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \ \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}, \underline{mmts_2} \ \wedge$
$tr_{\mathsf{h}} \sim tr_2 +\!\!+ \underline{tr_1} \ \wedge$
$(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_{\mathsf{h}}} \wedge \overrightarrow{c_{\mathsf{pfx}}} = \overrightarrow{c_{\mathsf{h}}} \wedge ts_\omega = ts_{\mathsf{h}} \wedge mmts_\omega = \underline{mmts_2} \wedge [] \sim \underline{tr}) \wedge$
$(\mathsf{STOP}((\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, []) \longrightarrow (\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}} = \overrightarrow{s_{\mathsf{h}}} \wedge [] = \overrightarrow{c_{\mathsf{h}}} \wedge \underline{ts_2} = ts_{\mathsf{h}})\ .$

From $\mathsf{STOP}((\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [])$, we have:
$(\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}} = \overrightarrow{s_{\mathsf{h}}} \ \wedge \ [] = \overrightarrow{c_{\mathsf{h}}} \ \wedge \ \underline{ts_2} = ts_{\mathsf{h}}\ .$

From Theorem B.6.5, we have:
$\overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_1 \ \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \ (\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_2}, \underline{mmts_2}\ .$

We prove the goals with $tr_x = tr_1 +\!\!+ tr_{\mathsf{h}} +\!\!+ \underline{tr_2}, \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = \underline{ts_\omega}$ as follows:

(1) $[\mathtt{loop}\ r\ e\ ((r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts$
$\xrightarrow{tr_1}{}^*_\delta \ \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_1 \ \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \ (\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_2}, \underline{mmts_2}$
$\xrightarrow{tr_2}{}^*_\delta \ (\mathtt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_\omega} \ \xrightarrow{[]}_\delta \ [], [], \underline{ts_\omega}, \underline{mmts_\omega}$

(2) From $tr = tr_1 +\!\!+ tr_2$ and $tr_{\mathsf{h}} \sim tr_2 +\!\!+ \underline{tr_1}$, we have: $tr_1 +\!\!+ tr_{\mathsf{h}} +\!\!+ \underline{tr_2} \sim tr +\!\!+ \underline{tr}$

(3) From $\overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}}$, we have: $\neg\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega})$

(4) $[] = [] \ \wedge \ [] = [] \ \wedge \ \underline{ts_\omega} = \underline{ts_\omega}$

‡ (**FIRST-ONGOING**):

For this sub-case, we have:
$(r := \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts, mmts_\omega$
$\xrightarrow{[]}_\delta \ \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega \ \xrightarrow{tr}{}^*_\delta \ (\mathtt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, [], \underline{ts_{\mathsf{r}}}, \underline{mmts_\omega}\ .$

From IH, we have:
$\exists tr_{\mathsf{h}}, \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_1 \ \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \ \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}, \underline{mmts_\omega} \ \wedge$
$tr_{\mathsf{h}} \sim tr_2 +\!\!+ \underline{tr} \ \wedge$
$(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_{\mathsf{h}}} \wedge \overrightarrow{c_{\mathsf{pfx}}} = \overrightarrow{c_{\mathsf{h}}} \wedge ts_\omega = ts_{\mathsf{h}} \wedge mmts_\omega = \underline{mmts_\omega} \wedge [] \sim \underline{tr}) \wedge$
$(\mathsf{STOP}((\mathtt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, []) \longrightarrow (\mathtt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}} = \overrightarrow{s_{\mathsf{h}}} \wedge [] = \overrightarrow{c_{\mathsf{h}}} \wedge \underline{ts_{\mathsf{r}}} = ts_{\mathsf{h}})\ .$

From $\mathsf{STOP}((\mathtt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [])$, we have:
$(\mathtt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}} = \overrightarrow{s_{\mathsf{h}}} \ \wedge \ [] = \overrightarrow{c_{\mathsf{h}}} \ \wedge \ \underline{ts_{\mathsf{r}}} = ts_{\mathsf{h}}\ .$

From Theorem B.6.5, we have:
$\overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_1 \ \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \ (\mathtt{break}) :: \underline{\overrightarrow{s_{\mathsf{r}}}}, \overrightarrow{c_{\mathsf{b}}}, \underline{ts_{\mathsf{r}}}, \underline{mmts_\omega}\ .$

We prove the goals with $tr_x = tr_1 +\!\!+ tr_\mathsf{h}, \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = \underline{ts_\omega}$ as follows:

(1) $[\texttt{loop}\ r\ e\ ((r \coloneqq \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts$
$\xrightarrow{tr_1}{}^*_\delta\ \overrightarrow{s}, \overrightarrow{c_\mathsf{b}}, ts_{(1,1)}, mmts_1\ \xrightarrow{tr_\mathsf{h}}{}^*_\delta\ (\texttt{break}) :: \underline{\overrightarrow{s_\mathsf{r}}}, \overrightarrow{c_\mathsf{b}}, \underline{ts_\mathsf{r}}, \underline{mmts_\omega}\ \xrightarrow{[]}_\delta\ [], [], \underline{ts_\omega}, \underline{mmts_\omega}$

(2) From $tr = tr_1 +\!\!+ tr_2$ and $tr_\mathsf{h} \sim tr_2 +\!\!+ \underline{tr}$, we have: $tr_1 +\!\!+ tr_\mathsf{h} +\!\!+ \underline{tr} \sim tr +\!\!+ \underline{tr}$

(3) From $\overrightarrow{c_\omega} = \overrightarrow{c_\mathsf{pfx}} +\!\!+ \overrightarrow{c_\mathsf{b}}$, we have: $\neg\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega})$

(4) $[] = [] \ \wedge\ [] = [] \ \wedge\ \underline{ts_\omega} = \underline{ts_\omega}$

† (**LOOP-ONGOING**):

For this sub-case, we have:
$(r \coloneqq \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_\mathsf{b}}, ts_\mathsf{b}, mmts_\omega\ \xrightarrow{tr/\overrightarrow{c_\mathsf{b}}}{}^*_\delta\ \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

Again, we identify the second execution's first iteration by applying Theorem B.6.11 to
$(r \coloneqq \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, \overrightarrow{c_\mathsf{b}}, ts_\mathsf{b}, mmts_\omega\ \xrightarrow{tr/\overrightarrow{c_\mathsf{b}}}{}^*_\delta\ \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

‡ (**FIRST-DONE**):

For this sub-case, we have:
$\exists \underline{\overrightarrow{s_2}}, \underline{ts_2}, \underline{mmts_2}, \underline{tr_1}, \underline{tr_2}, e.\ \underline{tr} = \underline{tr_1} +\!\!+ \underline{tr_2} \ \wedge$
$(r \coloneqq \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts_\mathsf{b}, mmts_\omega$
$\xrightarrow{[]}_\delta\ \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega\ \xrightarrow{tr_1}{}^*_\delta\ (\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [], \underline{ts_2}, \underline{mmts_2} \ \wedge$
$(\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, \overrightarrow{c_\mathsf{b}}, \underline{ts_2}, \underline{mmts_2}\ \xrightarrow{tr_2/\overrightarrow{c_\mathsf{b}}}{}^*_\delta\ \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_\mathsf{h}, \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_1\ \xrightarrow{tr_\mathsf{h}}{}^*_\delta\ \overrightarrow{s_\mathsf{h}}, \overrightarrow{c_\mathsf{h}}, ts_\mathsf{h}, \underline{mmts_2} \ \wedge$
$tr_\mathsf{h} \sim tr_2 +\!\!+ \underline{tr_1} \ \wedge$
$(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\mathsf{pfx}}) \longrightarrow \overrightarrow{s_\omega} = \overrightarrow{s_\mathsf{h}} \wedge \overrightarrow{c_\mathsf{pfx}} = \overrightarrow{c_\mathsf{h}} \wedge ts_\omega = ts_\mathsf{h} \wedge mmts_\omega = \underline{mmts_2} \wedge [] \sim \underline{tr}) \ \wedge$
$(\mathsf{STOP}((\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, []) \longrightarrow (\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}} = \overrightarrow{s_\mathsf{h}} \wedge [] = \overrightarrow{c_\mathsf{h}} \wedge \underline{ts_2} = ts_\mathsf{h})$.

From $\mathsf{STOP}((\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, [])$, we have:
$(\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}} = \overrightarrow{s_\mathsf{h}} \ \wedge\ [] = \overrightarrow{c_\mathsf{h}} \ \wedge\ \underline{ts_2} = ts_\mathsf{h}$.

From Theorem B.6.5, we have:
$\overrightarrow{s}, \overrightarrow{c_\mathsf{b}}, ts_{(1,1)}, mmts_1\ \xrightarrow{tr_\mathsf{h}}{}^*_\delta\ (\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, \overrightarrow{c_\mathsf{b}}, \underline{ts_2}, \underline{mmts_2}$.

We prove the goals with $tr_x = tr_1 +\!\!+ tr_\mathsf{h} +\!\!+ \underline{tr_2}, \overrightarrow{s_x} = [], \overrightarrow{c_x} = [], ts_x = \underline{ts_\omega}$ as follows:

(1) $[\texttt{loop}\ r\ e\ ((r \coloneqq \texttt{chkpt}([\texttt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts$
$\xrightarrow{tr_1}{}^*_\delta\ \overrightarrow{s}, \overrightarrow{c_\mathsf{b}}, ts_{(1,1)}, mmts_1\ \xrightarrow{tr_\mathsf{h}}{}^*_\delta\ (\texttt{continue}\ e) :: \underline{\overrightarrow{s_2}}, \overrightarrow{c_\mathsf{b}}, \underline{ts_2}, \underline{mmts_2}$
$\xrightarrow{tr_2}{}^*_\delta\ \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}}, \underline{ts_\omega}, \underline{mmts_\omega}$

(2) From $tr = tr_1 +\!\!+ tr_2$ and $tr_\mathsf{h} \sim tr_2 +\!\!+ \underline{tr_1}$, we have: $tr_1 +\!\!+ tr_\mathsf{h} +\!\!+ \underline{tr_2} \sim tr +\!\!+ \underline{tr}$

(3) From $\overrightarrow{c_\omega} = \overrightarrow{c_\mathsf{pfx}} +\!\!+ \overrightarrow{c_\mathsf{b}}$, we have: $\neg\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega})$

(4) For some $\underline{\overrightarrow{c_\mathsf{pfx}}}$, we have: $\underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_\mathsf{pfx}}} +\!\!+ \overrightarrow{c_\mathsf{b}}$. Therefore, $\neg\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}})$

‡ (**FIRST-ONGOING**):

For this sub-case, we have:
$\exists \underline{\overrightarrow{c_{\mathsf{pfx}}}}.\ \underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_{\mathsf{pfx}}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}} \ \wedge$
$(r \coloneqq \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s}, [], ts, mmts_\omega$
$\xrightarrow{[]}_\delta \ \overrightarrow{s}, [], ts_{(1,1)}, mmts_\omega \ \xrightarrow{tr}{}^*_\delta \ \underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_{\mathsf{pfx}}}}, \underline{ts_\omega}, \underline{mmts_\omega}$.

From IH, we have:
$\exists tr_{\mathsf{h}}, \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}.$
$\overrightarrow{s}, [], ts_{(1,1)}, mmts_1 \ \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \ \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}}, ts_{\mathsf{h}}, \underline{mmts_\omega} \ \wedge$
$tr_{\mathsf{h}} \sim tr_2 +\!\!+ \underline{tr} \ \wedge$
$(\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_{\mathsf{pfx}}}) \ \longrightarrow \ \overrightarrow{s_\omega} = \overrightarrow{s_{\mathsf{h}}} \wedge \overrightarrow{c_{\mathsf{pfx}}} = \overrightarrow{c_{\mathsf{h}}} \wedge ts_\omega = ts_{\mathsf{h}} \wedge mmts_\omega = \underline{mmts_2} \wedge [] \sim \underline{tr}) \ \wedge$
$(\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_{\mathsf{pfx}}}}) \ \longrightarrow \ \underline{\overrightarrow{s_\omega}} = \overrightarrow{s_{\mathsf{h}}} \ \wedge \ \underline{\overrightarrow{c_{\mathsf{pfx}}}} = \overrightarrow{c_{\mathsf{h}}} \ \wedge \ \underline{ts_\omega} = ts_{\mathsf{h}})$.

From Theorem B.6.5, we have:
$\overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_1 \ \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \ \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{h}}, \underline{mmts_\omega}$.

We prove the goals with $tr_x = tr_1 +\!\!+ tr_{\mathsf{h}}, \overrightarrow{s_x} = \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_x} = \overrightarrow{c_{\mathsf{h}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}}, ts_x = ts_{\mathsf{h}}$ as follows:

(1) $[\mathtt{loop}\ r\ e\ ((r \coloneqq \mathtt{chkpt}([\mathtt{return}\ r], \mathsf{mid}.lab)) :: \overrightarrow{s})], [], ts, mmts$
$\xrightarrow{tr_1}{}^*_\delta \ \overrightarrow{s}, \overrightarrow{c_{\mathsf{b}}}, ts_{(1,1)}, mmts_1 \ \xrightarrow{tr_{\mathsf{h}}}{}^*_\delta \ \overrightarrow{s_{\mathsf{h}}}, \overrightarrow{c_{\mathsf{h}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}}, ts_{\mathsf{h}}, \underline{mmts_\omega}$

(2) From $tr = tr_1 +\!\!+ tr_2$ and $tr_{\mathsf{h}} \sim tr_2 +\!\!+ \underline{tr}$, we have: $tr_1 +\!\!+ tr_{\mathsf{h}} +\!\!+ \underline{tr} \sim tr +\!\!+ \underline{tr}$

(3) From $\overrightarrow{c_\omega} = \overrightarrow{c_{\mathsf{pfx}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}}$, we have: $\neg\mathsf{STOP}(\overrightarrow{s_\omega}, \overrightarrow{c_\omega})$

(4) From $\underline{\overrightarrow{c_\omega}} = \underline{\overrightarrow{c_{\mathsf{pfx}}}} +\!\!+ \overrightarrow{c_{\mathsf{b}}}$, we have: $\neg\mathsf{STOP}(\underline{\overrightarrow{s_\omega}}, \underline{\overrightarrow{c_\omega}})$

□

Bibliography

Jade Alglave. 2012. A formal hierarchy of weak memory models. *Form. Methods Syst. Des.* 41, 2 (oct 2012), 178–210. https://doi.org/10.1007/s10703-012-0161-5

Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (jul 2014), 74 pages. https://doi.org/10.1145/2627752

Arm. 2020. Arm architecture reference manual Armv8, for Armv8-A architecture profile (DDI 0487F.b). https://static.docs.arm.com/ddi0487/fb/DDI0487F_b_armv8_arm.pdf

Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL (2019). https://doi.org/10.1145/3290384

Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (jan 2018), 553–565.

Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. 2019. Tracking in Order to Recover: Detectable Recovery of Lock-Free Data Structures. https://doi.org/10.48550/ARXIV.1905.13600

Hagit Attiya, Ohad Ben-Baruch, Panagiota Fatourou, Danny Hendler, and Eleftherios Kosmas. 2022. Detectable Recovery of Lock-Free Data Structures. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 262–277. https://doi.org/10.1145/3503221.3508444

Hagit Attiya, Ohad Ben-Baruch, and Danny Hendler. 2018. Nesting-Safe Recoverable Linearizability: Modular Constructions for Non-Volatile Memory. In *Proceedings of the 2018 ACM Symposium on Principles of Distributed Computing* (Egham, United Kingdom) *(PODC '18)*. Association for Computing Machinery, New York, NY, USA, 7–16. https://doi.org/10.1145/3212734.3212753

Hillel Avni, Eliezer Levy, and Avi Mendelson. 2015. Hardware Transactions in Nonvolatile Memory. In *Proceedings of the 29th International Symposium on Distributed Computing - Volume 9363* (Tokyo, Japan) *(DISC 2015)*. Springer-Verlag, Berlin, Heidelberg, 617–630. https://doi.org/10.1007/978-3-662-48653-5_41

H. Alan Beadle, Wentao Cai, Haosen Wen, and Michael L. Scott. 2020. Nonblocking Persistent Software Transactional Memory. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Diego, California) *(PPoPP '20)*. Association for Computing Machinery, New York, NY, USA, 429–430. https://doi.org/10.1145/3332466.3374506

Naama Ben-David, Guy E. Blelloch, Michal Friedman, and Yuanhao Wei. 2019. Delay-Free Concurrency on Faulty Persistent Memory. In *The 31st ACM Symposium on Parallelism in Algorithms and Architectures* (Phoenix, AZ, USA) *(SPAA '19)*. Association for Computing Machinery, New York, NY, USA, 253–264. https://doi.org/10.1145/3323165.3323187

Eleni Vafeiadi Bila, Brijesh Dongol, Ori Lahav, Azalea Raad, and John Wickerson. 2022. View-Based Owicki–Gries Reasoning for Persistent x86-TSO. In *Programming Languages and Systems*, Ilya Sergey (Ed.). Springer International Publishing, Cham, 234–261.

Jim Blandy. 2022. Comparison of Rust async and Linux thread context switch time and memory use. https://github.com/jimblandy/context-switch

Robert Blankenship. 2020. CXL 1.1 Protocol Extensions: Review of the Cache and Memory Protocols in CXL. https://www.snia.org/educational-library/cxl-11-protocol-extensions-review-cache-and-memory-protocols-cxl-2020

Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Understanding and Optimizing Persistent Memory Allocation. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management* (London, UK) *(ISMM 2020)*. Association for Computing Machinery, New York, NY, USA, 60–73. https://doi.org/10.1145/3381898.3397212

Wentao Cai, Haosen Wen, Vladimir Maksimovski, Mingzhe Du, Rafaello Sanna, Shreif Abdallah, and Michael L. Scott. 2021. Fast Nonblocking Persistence for Concurrent Data Structures. In *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference) (LIPIcs, Vol. 209)*, Seth Gilbert (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 14:1–14:20. https://doi.org/10.4230/LIPIcs.DISC.2021.14

Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. *SIGPLAN Not.* 49, 10 (Oct. 2014), 433–452. https://doi.org/10.1145/2714064.2660224

Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2021. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *19th USENIX Conference on File and Storage Technologies, FAST 2021, February 23-25, 2021*, Marcos K. Aguilera and Gala Yadgar (Eds.). USENIX Association, 81–95. https://www.usenix.org/conference/fast21/presentation/chen-youmin

Zhangyu Chen, Yu Hua, Bo Ding, and Pengfei Zuo. 2020. Lock-free Concurrent Level Hashing for Persistent Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 799–812. https://www.usenix.org/conference/atc20/presentation/chen

Zhangyu Chen, Yu Hua, Yongle Zhang, and Luochangqi Ding. 2022. Efficiently Detecting Concurrency Bugs in Persistent Memory Programs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 873–887. https://doi.org/10.1145/3503222.3507755

Kyeongmin Cho, Seungmin Jeon, Azalea Raad, and Jeehoon Kang. 2023a. *Artifact for Article "Memento: A Framework for Detectable Recoverability in Persistent Memory"*. https://doi.org/10.5281/zenodo.7811928

Kyeongmin Cho, Seungmin Jeon, Azalea Raad, and Jeehoon Kang. 2023b. Memento: A Framework for Detectable Recoverability in Persistent Memory. *Proc. ACM Program. Lang.* 7, PLDI, Article 118 (jun 2023), 26 pages. https://doi.org/10.1145/3591232

Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. 2021a. *Mechanized Proof and Model Checker for Article: Revamping Hardware Persistency Models*. https://doi.org/10.1145/3410292

Kyeongmin Cho, Sung-Hwan Lee, Azalea Raad, and Jeehoon Kang. 2021b. Revamping Hardware Persistency Models: View-Based and Axiomatic Persistency Models for Intel-X86 and Armv8. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 16–31. https://doi.org/10.1145/3453483.3454027

Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

CCIX Consortium. 2024a. Cache Coherent Interconnect for Accelerators. https://www.ccixconsortium.com/

CXL Consortium. 2024b. Compute Express Link. https://www.computeexpresslink.org/

Coq. 2024. The Coq Proof Assistant. https://coq.inria.fr/

R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1989. An Efficient Method of Computing Static Single Assignment Form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 25–35. https://doi.org/10.1145/75277.75280

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 373–386. https://www.usenix.org/conference/atc18/presentation/david

John Derrick, Simon Doherty, Brijesh Dongol, Gerhard Schellhorn, and Heike Wehrheim. 2019. Verifying Correctness of Persistent Concurrent Data Structures. In *Formal Methods – The Next 30 Years*, Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira (Eds.). Springer International Publishing, Cham, 179–195.

Crossbeam Developers. 2019. Crossbeam. https://github.com/crossbeam-rs/crossbeam

Panagiota Fatourou and Nikolaos D. Kallimanis. 2011. A Highly-Efficient Wait-Free Universal Construction. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 325–334. https://doi.org/10.1145/1989493.1989549

Panagiota Fatourou and Nikolaos D. Kallimanis. 2012. Revisiting the Combining Synchronization Technique. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 257–266. https://doi.org/10.1145/2145816.2145849

Panagiota Fatourou, Nikolaos D. Kallimanis, and Eleftherios Kosmas. 2022. The Performance Power of Software Combining in Persistence. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 337–352. https://doi.org/10.1145/3503221.3508426

Keir Fraser. 2004. *Practical lock-freedom.* Technical Report UCAM-CL-TR-579. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-579

Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 377–392. https://doi.org/10.1145/3385412.3386031

Michal Friedman, Maurice Herlihy, Virendra Marathe, and Erez Petrank. 2018. A Persistent Lock-Free Queue for Non-Volatile Memory. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Vienna, Austria) *(PPoPP '18)*. Association for Computing Machinery, New York, NY, USA, 28–40. https://doi.org/10.1145/3178487.3178490

Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. *Mirror: Making Lock-Free Data Structures Persistent.* Association for Computing Machinery, New York, NY, USA, 1218–1232. https://doi.org/10.1145/3453483.3454105

Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. *SIGARCH Comput. Archit. News* 18, 2SI (May 1990), 15–26. https://doi.org/10.1145/325096.325102

Vaibhav Gogte, Stephan Diestelhorst, William Wang, Satish Narayanasamy, Peter M. Chen, and Thomas F. Wenisch. 2018. Persistency for Synchronization-free Regions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 46–61. https://doi.org/10.1145/3192366.3192367

James R. Goodman, Mary K. Vernon, and Philip J. Woest. 1989. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, USA) *(ASPLOS III)*. Association for Computing Machinery, New York, NY, USA, 64–75. https://doi.org/10.1145/70082.68188

Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu, and Brian Demsky. 2022a. Checking Robustness to Weak Persistency Models. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 490–505. https://doi.org/10.1145/3519939.3523723

Hamed Gorjiara, Guoqing Harry Xu, and Brian Demsky. 2022b. Yashme: Detecting Persistency Races. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 830–845. https://doi.org/10.1145/3503222.3507766

Timothy L. Harris. 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC '01)*. Springer-Verlag, Berlin, Heidelberg, 300–314.

Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. 2010. Flat Combining and the Synchronization-Parallelism Tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Thira, Santorini, Greece) *(SPAA '10)*. Association for Computing Machinery, New York, NY, USA, 355–364. https://doi.org/10.1145/1810479.1810540

Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (jul 1990), 463–492. https://doi.org/10.1145/78969.78972

Morteza Hoseinzadeh and Steven Swanson. 2021. Corundum: Statically-Enforced Persistent Memory Safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 429–442. https://doi.org/10.1145/3445814.3446710

Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. 2021. Persistent Memory Hash Indexes: An Experimental Evaluation. *Proc. VLDB Endow.* 14, 5 (jan 2021), 785–798. https://doi.org/10.14778/3446095.3446101

Taeho Hwang, Jaemin Jung, and Youjip Won. 2015. HEAPO: Heap-Based Persistent Object Store. *ACM Trans. Storage* 11, 1, Article 3 (Dec. 2015), 21 pages. https://doi.org/10.1145/2629619

Intel. 2021. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html

Intel. 2023. The libpmem2 library. https://pmem.io/pmdk/libpmem2/

Intel. 2024a. Intel® 64 and IA-32 Architectures Software Developer Manuals. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

Intel. 2024b. Intel® Optane™ Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

Intel. 2024c. Intel® Optane™ Persistent Memory 200 Series Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/series/203877/intel-optane-persistent-memory-200-series.html

Intel. 2024d. Persistent Memory Programming. https://pmem.io/pmdk/

Iris. 2024. Iris Project. https://iris-project.org/

Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016a. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*. Springer, 313–327.

Joseph Izraelevitz, Hammurabi Mendes, and Michael L Scott. 2016b. Linearizability of persistent memory objects under a full-system-crash failure model. In *International Symposium on Distributed Computing*. Springer, 313–327.

Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. arXiv:1903.05714 [cs.DC]

Eric H. Jensen, Gary W. Hagensen, and Jeffrey M. Broughton. 1987. A New Approach to Exclusive Data Access in Shared Memory Multiprocessors. Technical Report UCRL-97663 (Nov 1987).

Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2015. Efficient Persist Barriers for Multicores. In *Proceedings of the 48th International Symposium on Microarchitecture* (Waikiki, Hawaii) *(MICRO-48)*. ACM, New York, NY, USA, 660–671. https://doi.org/10.1145/2830772.2830805

Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: A Hugepage-Aware File System for Persistent Memory That Ages Gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 804–818. https://doi.org/10.1145/3477132.3483567

Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. 2017. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 175–189. https://doi.org/10.1145/3009837.3009850

Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. 2018. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 34, 15 pages. https://doi.org/10.1145/3190508.3190510

Artem Khyzha and Ori Lahav. 2021. Taming x86-TSO persistency. *Proc. ACM Program. Lang.* 5, POPL, Article 47 (jan 2021), 29 pages. https://doi.org/10.1145/3434328

Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. 2021a. LineFS: Efficient SmartNIC Offload of a Distributed File System with Pipeline Parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 756–771. https://doi.org/10.1145/3477132.3483565

Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021b. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 424–439. https://doi.org/10.1145/3477132.3483589

Kioxia. 2024. XL-FLASH | Storage Class Memory (SCM). https://americas.kioxia.com/en-us/business/memory/xlflash.html

Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad, and Viktor Vafeiadis. 2021. PerSeVerE: Persistency Semantics for Verification under Ext4. *Proc. ACM Program. Lang.* 5, POPL, Article 43 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434324

Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M. Chen, Satish Narayanasamy, and Thomas F. Wenisch. 2017. Language-level Persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. ACM, New York, NY, USA, 481–493. https://doi.org/10.1145/3079856.3080229

Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 399–411. https://doi.org/10.1145/2872362.2872381

R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 335–349. https://doi.org/10.1145/3373376.3378483

Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 460–477. https://doi.org/10.1145/3132747.3132770

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers* 28, 9 (Sept. 1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. 2020. Promising 2.0: global optimizations in relaxed memory concurrency. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 362–376. https://doi.org/10.1145/3385412.3386010

Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: Converting Concurrent DRAM Indexes to Persistent-Memory Indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 462–477. https://doi.org/10.1145/3341301.3359635

Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* 13, 4 (dec 2019), 574–587. https://doi.org/10.14778/3372716.3372728

Nan Li and Wojciech Golab. 2021. Brief Announcement: Detectable Sequential Specifications for Recoverable Shared Objects. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing* (Virtual Event, Italy) *(PODC'21)*. Association for Computing Machinery, New York, NY, USA, 557–560. https://doi.org/10.1145/3465084.3467943

Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 329–343. https://doi.org/10.1145/3037697.3037714

Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 411–425. https://doi.org/10.1145/3297858.3304015

Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (apr 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 773–785. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

Youyou Lu, Jiwu Shu, and Long Sun. 2016. Blurred Persistence: Efficient Transactions in Persistent Memory. *ACM Trans. Storage* 12, 1, Article 3 (Jan. 2016), 29 pages. https://doi.org/10.1145/2851504

Paul E McKenney. 2005. Memory ordering in modern microprocessors, part I. *Linux Journal* 2005, 136 (2005), 2.

John Meehan, Nesime Tatbul, Stan Zdonik, Cansu Aslantas, Ugur Cetintemel, Jiang Du, Tim Kraska, Samuel Madden, David Maier, Andrew Pavlo, Michael Stonebraker, Kristin Tufte, and Hao Wang. 2015. S-Store: Streaming Meets Transaction Processing. *Proc. VLDB Endow.* 8, 13 (sep 2015), 2134–2145. https://doi.org/10.14778/2831360.2831367

Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. 2017. Atomic In-Place Updates for Non-Volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 499–512. https://doi.org/10.1145/3064176.3064215

Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (jun 2004), 491–504. https://doi.org/10.1109/TPDS.2004.8

Maged M. Michael and Michael L. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing* (Philadelphia, Pennsylvania, USA) *(PODC '96)*. Association for Computing Machinery, New York, NY, USA, 267–275. https://doi.org/10.1145/248052.248106

Microsoft. 2023. High context switch rate. https://learn.microsoft.com/en-us/gaming/gdk/_content/gc/system/overviews/finding-threading-issues/high-context-switches

Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: Reusable Engineering of Real-World Semantics *(ICFP '14)*. https://doi.org/10.1145/2628136.2628143

Moohyeon Nam, Hokeun Cha, Young ri Choi, Sam H. Noh, and Beomseok Nam. 2019. Write-Optimized Dynamic Hashing for Persistent Memory. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 31–44. https://www.usenix.org/conference/fast19/presentation/nam

Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory Management Techniques for Large-Scale Persistent-Main-Memory Systems. *Proc. VLDB Endow.* 10, 11 (Aug. 2017), 1166–1177. https://doi.org/10.14778/3137628.3137629

Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 371–386. https://doi.org/10.1145/2882903.2915251

Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (Munich, Germany) *(TPHOLs '09)*. Springer-Verlag, Berlin, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory persistency. *SIGARCH Comput. Archit. News* 42, 3 (jun 2014), 265–276. https://doi.org/10.1145/2678373.2665712

Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *Proc. ACM Program. Lang.* 2, POPL, Article 19 (dec 2017), 29 pages. https://doi.org/10.1145/3158107

Christopher Pulte, Jean Pichon-Pharabod, Jeehoon Kang, Sung-Hwan Lee, and Chung-Kil Hur. 2019. Promising-ARM/RISC-V: a simpler and faster operational concurrency model. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3314221.3314624

Azalea Raad, Ori Lahav, and Viktor Vafeiadis. 2020. Persistent Owicki-Gries reasoning: a program logic for reasoning about persistent programs on Intel-x86. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 151 (nov 2020), 28 pages. https://doi.org/10.1145/3428219

Azalea Raad and Viktor Vafeiadis. 2018. Persistence Semantics for Weak Memory: Integrating Epoch Persistency with the TSO Memory Model. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 137 (Oct. 2018), 27 pages. https://doi.org/10.1145/3276507

Azalea Raad, John Wickerson, Gil Neiger, and Viktor Vafeiadis. 2019b. Persistency Semantics of the Intel-X86 Architecture. *Proc. ACM Program. Lang.* 4, POPL, Article 11 (dec 2019), 31 pages. https://doi.org/10.1145/3371079

Azalea Raad, John Wickerson, and Viktor Vafeiadis. 2019a. Weak persistency semantics from the ground up: formalising the persistency semantics of ARMv8 and transactional models. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 135 (oct 2019), 27 pages. https://doi.org/10.1145/3360561

Ganesan Ramalingam and Kapil Vaswani. 2013. Fault Tolerance via Idempotence. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 249–262. https://doi.org/10.1145/2429069.2429100

Matan Rusanovsky, Hagit Attiya, Ohad Ben-Baruch, Tom Gerby, Danny Hendler, and Pedro Ramalhete. 2021. Flat-Combining-Based Persistent Data Structures for Non-Volatile Memory. arXiv:2012.12868 [cs.DC]

Samsung. 2022. API list of Scalable Memory Development Kit (SMDK). https://github.com/OpenMPDK/SMDK/wiki/5.-Plugin#api-list

Samsung. 2024. CMM-H (CXL Memory Module, H: Hybrid). https://samsungmsl.com/cmmh/

Srinath Setty, Chunzhi Su, Jacob R. Lorch, Lidong Zhou, Hao Chen, Parveen Patel, and Jinglei Ren. 2016. Realizing the Fault-Tolerance Promise of Cloud Storage Using Locks with Intent. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 501–516. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/setty

Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. X86-TSO: A Rigorous and Usable Programmer's Model for X86 Multiprocessors. *Commun. ACM* 53, 7 (2010).

Ori Shalev and Nir Shavit. 2006. Split-Ordered Lists: Lock-Free Extensible Hash Tables. *J. ACM* 53, 3 (may 2006), 379–405. https://doi.org/10.1145/1147954.1147958

Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 323–337. https://doi.org/10.1145/3127479.3128610

Hongping Shu, Hongyu Chen, Hao Liu, Youyou Lu, Qingda Hu, and Jiwu Shu. 2018. Empirical Study of Transactional Management for Persistent Memory. 61–66. https://doi.org/10.1109/NVMSA.2018.00015

Arash Tavakkol, Aasheesh Kolli, Stanko Novakovic, Kaveh Razavi, Juan Gómez-Luna, Hasan Hassan, Claude Barthels, Yaohua Wang, Mohammad Sadrosadati, Saugata Ghose, Ankit Singla, Pratap Subrahmanyam, and Onur Mutlu. 2018. Enabling Efficient RDMA-based Synchronous Mirroring of Persistent Memory Transactions. *CoRR* abs/1810.09360 (2018). arXiv:1810.09360 http://arxiv.org/abs/1810.09360

The Rust Team. 2019. Rust Programming Language. https://www.rust-lang.org/

Shahar Timnat and Erez Petrank. 2014. A Practical Wait-Free Simulation for Lock-Free Data Structures. *SIGPLAN Not.* 49, 8 (feb 2014), 357–368. https://doi.org/10.1145/2692916.2555261

R.K. Treiber. 1986. *Systems Programming: Coping with Parallelism.* International Business Machines Incorporated, Thomas J. Watson Research Center. https://books.google.co.kr/books?id=YQg3HAAACAAJ

Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Building Blocks for Persistent Memory: How to Get the Most out of Your New Memory? *The VLDB Journal* 29, 6 (nov 2020), 1223–1241. https://doi.org/10.1007/s00778-020-00622-9

Simon Friis Vindum and Lars Birkedal. 2023. Spirea: A Mechanized Concurrent Separation Logic for Weak Persistent Memory. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 244 (oct 2023), 26 pages. https://doi.org/10.1145/3622820

Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

Guozhang Wang, Lei Chen, Ayusman Dikshit, Jason Gustafson, Boyang Chen, Matthias J. Sax, John Roesler, Sophie Blee-Goldman, Bruno Cadonna, Apurva Mehta, Varun Madan, and Jun Rao. 2021. Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2602–2613. https://doi.org/10.1145/3448016.3457556

Tianzheng Wang, Justin Levandoski, and Per-Ake Larson. 2018. Easy Lock-Free Indexing in Non-Volatile Memory. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 461–472. https://doi.org/10.1109/ICDE.2018.00049

Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch, and Erez Petrank. 2022. FliT: A Library for Simple and Efficient Persistent Algorithms. In *Proceedings of the The 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, South Korea) *(PPoPP 2022)*.

Haosen Wen, Wentao Cai, Mingzhe Du, Louis Jenkins, Benjamin Valpey, and Michael L. Scott. 2021. A Fast, General System for Buffered Persistent Data Structures. In *ICPP 2021: 50th International Conference on Parallel Processing, Lemont, IL, USA, August 9 - 12, 2021*, Xian-He Sun, Sameer Shende, Laxmikant V. Kalé, and Yong Chen (Eds.). ACM, 73:1–73:11. https://doi.org/10.1145/3472456.3472458

Wikipedia. 2022. Tagged pointer. https://en.wikipedia.org/wiki/Tagged_pointer

Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

Yi Xu, Joseph Izraelevitz, and Steven Swanson. 2021. Clobber-NVM: Log Less, Re-Execute More. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 346–359.

Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. 2020. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1187–1204. https://www.usenix.org/conference/osdi20/presentation/zhang-haoran

Pengfei Zuo, Yu Hua, and Jie Wu. 2018. Write-Optimized and High-Performance Hashing Index Scheme for Persistent Memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 461–476. https://www.usenix.org/conference/osdi18/presentation/zuo

Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 128 (Oct. 2019), 26 pages. https://doi.org/10.1145/3360554

Acknowledgments

The following tree rigorously demonstrates the structure of the acknowledgments for this Ph.D. dissertation. Any alteration or removal of the components within this tree would result in the non-existence of the dissertation itself. For simplicity, I omit all leaves from the tree, which assume the premises of *Divine Providence*.