Testing and Analyzing Correctness in Concurrent Systems: From Microprocessors to IoT and Distributed Systems

Themis Melissaris

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Advisers: Margaret Martonosi and Kelly Shaw

September 2021

Abstract

Applications such as the Internet of Things (IoT), multimedia streaming and self driving cars are drivers for explosive data growth. This data growth along with the slowdown of Moore's law presents new challenges for computing and spurs innovation for the next generation of hardware and software. To extract performance at manageable power, systems ranging from microprocessors to IoT devices and distributed systems increasingly leverage parallelism and concurrency. However, the performance and power benefits of parallelism come at the cost of increased complexity in designing, specifying and testing correctness of these systems.

This dissertation draws insights across the domains of computer architecture, and distributed systems, resulting in a set of practical tools and methodologies that allow for rigorous and efficient testing of correctness on multiprocessors, IoT systems and distributed systems.

More specifically, this thesis introduces PerpLE, a tool that enables consistency testing for multiprocessors without per-iteration synchronization and relies on property based testing using litmus tests. This dissertation demonstrates how PerpLE speeds up testing performance by orders of magnitude against the state of the art, enabling fast error discovery in commercial microprocessors. Additionally, this dissertation presents Musli Tool, a practical tool for empirical consistency testing of distributed systems that leverages techniques and methodologies based on the computer architecture concept of litmus tests. Musli Tool automates testing by generating property-based tests and increases testing performance and fast portability in open source distributed systems. Finally, this thesis presents a study of rigorous concurrency testing on commercial and open source IoT systems. The study demonstrates correctness deficiencies of IoT systems and identifies the implications of race conditions, lack of atomicity and weak consistency on applications. To mitigate erroneous behavior for IoT applications, the OKAPI tool is introduced.

This dissertation presents how to vastly improve the performance and efficiency of consistency testing by leveraging a synchronization free variant of litmus tests and by developing tools to enable analysis and generation of tests based on this new methodology. This thesis also demonstrates how to take lessons learned from testing correctness in microprocessors and adapting to other domains of concurrent systems, particularly distributed systems and IoT.

Acknowledgements

First and foremost, I would like to thank my advisors Prof. Margaret Martonosi and Prof. Kelly Shaw for their support and guidance over the years. I am grateful for their faith in me and their constant support and encouragement during challenging times. They taught me a great deal about research. I am really grateful for their patience and perseverance in helping me improve my writing, communication and presentation skills by showing me how to frame my research clearly and convincingly. I also appreciate that they gave me the freedom to pursue my research interests and ideas.

I thank the rest of my dissertation committee: Prof. David August, Prof. Kyle Jamieson and Prof. Wyatt Lloyd. I want to additionally thank Prof. Kelly Shaw and Prof. Kyle Jamieson for taking the time to serve as readers on my committee. Their feedback helped improve the quality of this dissertation. Prof. Wyatt Lloyd has also been a great source of feedback and support for the work presented in this dissertation, his seminar in distributed systems was a great source of inspiration to pursue related research directions.

I thank the members of Margaret Martonosi's Research Group (MRM Group) for all their help throughout the years. The culture of collaboration and solidarity in the MRM Group made my research much more enjoyable and rewarding. I want to thank Yavuz, Ozlem, Dan, Wenhao, Ali, Tae Jun, Caroline, Yatin, Prakash, Aninda, Naorin, Teague, Wei, Daeki, Luwa, Tyler and Yipeng for bringing energy, enthusiasm and stimulating conversations in the daily life in the group. I would like to specially thank Markos for being a great friend and collaborator and for his contributions to the PerpLE project, part of which is presented in Chapter 3. I would also like to thank Haonan Lu for always being available to discuss about distributed systems.

I would also like to thank my mentors and people I worked with throughout my internships; Andrei Warkentin, Chris Ramming, Aasheesh Kolli, Nitin Agrawal,

Masoud Saeida Ardekani, Rayman Preet Singh, Michael O'Gorman and Kristen Wright for their valuable help and for introducing me to interesting new topics which helped shape the research directions of this dissertation.

I would like to thank the Department of Computer Science at Princeton. I would also like to thank the Center for Career Development and Gaeun Seo for helping me in various ways in preparation for my job search. I would also like to thank Stephanie Landers and the Princeton Entrepreneurial Hub for offering numerous talks and opportunities that enabled me to learn about numerous topics outside the scope of my research. I would also like to thank Prof. Dimitri Gondicas for his support for the Educational Trip program and for all events of the Hellenic Students' Association.

I am deeply grateful to all my friends for everything we shared during the past seven years. I thank the many friends I was lucky to meet at Princeton that helped me adapt to a new way of life both academically and socially. I would like to thank all my Greek friends at Princeton throughout these years: Michalis, Constantinos, Alexandros, Giorgos, Ioannis, Adam, Akis, Androklis, Anastasia, Nikitas, Jason, Efthymios, Ioanna, Ariadni, Zoe, Orestis, Dimitris, Konstantinos, Alexandros, Vassilis, Yannis, Eva, Rafail, Theano, Narek, George, Fotis and Marilena. I would also like to thank all friends from the Computer Science department and the rest of the Princeton community, especially Nayana, Logan, Marcela, Wafa, Irene, Susan, Murat, Noemi and Kevin. I would also like to thank my long-time friends from back home in Greece, especially Panos K., Danae, Evgenia, Aggeliki, DTsip, Pankgeorg, Adam, Stratos, Sofia, Antonis, Ioanna, Evi, Vaggos, Giannis, Efi, Kelly, Maria A., Maria M., Maria K. and Maria P. Thank you for your support and for all the great memories during the holidays and the summer downtime in Greece.

I would like to specially thank Manos, Jason, Kyriakos, Panagiotis, Kostas, Sotiris and Nick. My time at Princeton would be significantly harder without having the daily interactions and fun moments with Sotiris and Nick, both inside and outside

of the Computer Science department. I am grateful for my friendship with Manos, Jason, Kyriakos, Panagiotis and Kostas and I am thankful for all the great memories and support. At the same time, I could have possibly graduated earlier without them occasionally being a bad influence. Finally, I would like to thank Eleni, for her patience, encouragement, and love. She made life during the pandemic exciting, supported me, and helped me grow a lot as a person over the last year.

Most importantly, I would also like to thank my family. Theia Christine, theios George, Tom, Karl, Joanne, Clay, Diane, Jay, Alex, Athan, Samantha, Luke and the late theia Georgia have been my second family, and it has been amazing spending time and celebrating holidays with them; it always felt like home.

I want to thank my uncle Giannis who has always been the academic in the family. I also want to thank my cousins Dora and Themis for their support. We all started our PhDs at around the same time and have all finally gotten to the end of that journey. Congratulations to both! In 2021 our family will go from zero to two Themis Melissaris-es with a PhD.

I am thankful to my parents for their unconditional love and support, for their personal sacrifices to provide me the education and opportunities that enabled my path, and for the values they instilled in me. They stimulated my curiosity for engineering and the sciences from an early age and have been the greatest supporters in everything my brother and I do. I also want to thank my brother for always believing in me, for keeping me accountable and for pushing me to succeed. He has always been my best friend, a role model and a source of creativity. I am looking forward to what lies ahead for both of us.

I gratefully acknowledge generous financial support by the Seeger Center for Hellenic Studies through the Stanley J. Seeger fellowship, AG Leventis Foundation, Gerondelis Foundation, the Applications Driving Architectures (ADA) Research Center, and the

National Science Foundation (NSF) through the grants CNS-1135953, CCF-1533837 and CNS 1739674.

To my parents, Letti and George.

And to my brother, Dimitris.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Modern applications, systems and hardware take advantage of concurrency, making it increasingly prevalent in all areas of computing. Both hardware and software have been leveraging parallelism and concurrency for years, with applications ranging from laptop processors to embedded processors and from mobile applications to a large number of cloud software products running concurrently in data centers.

The trend towards highly concurrent hardware, systems and applications poses an increasing challenge for testing and validating the correctness of such systems. Bugs are present in all systems and with the system complexity rising in all levels of the system architecture, bugs become increasingly hard to observe. Under-specification, lack of specification, or incorrect system or hardware implementation can lead to incorrect program behavior. Notable recent correctness bugs involving concurrent hardware and systems have been reported in a range of systems [9, 81, 121].

This dissertation focuses on testing methodologies for ensuring correctness in hardware and systems and in the performance and efficiency of these methodologies. This chapter highlights the diversity of the hardware and the systems used to enable a

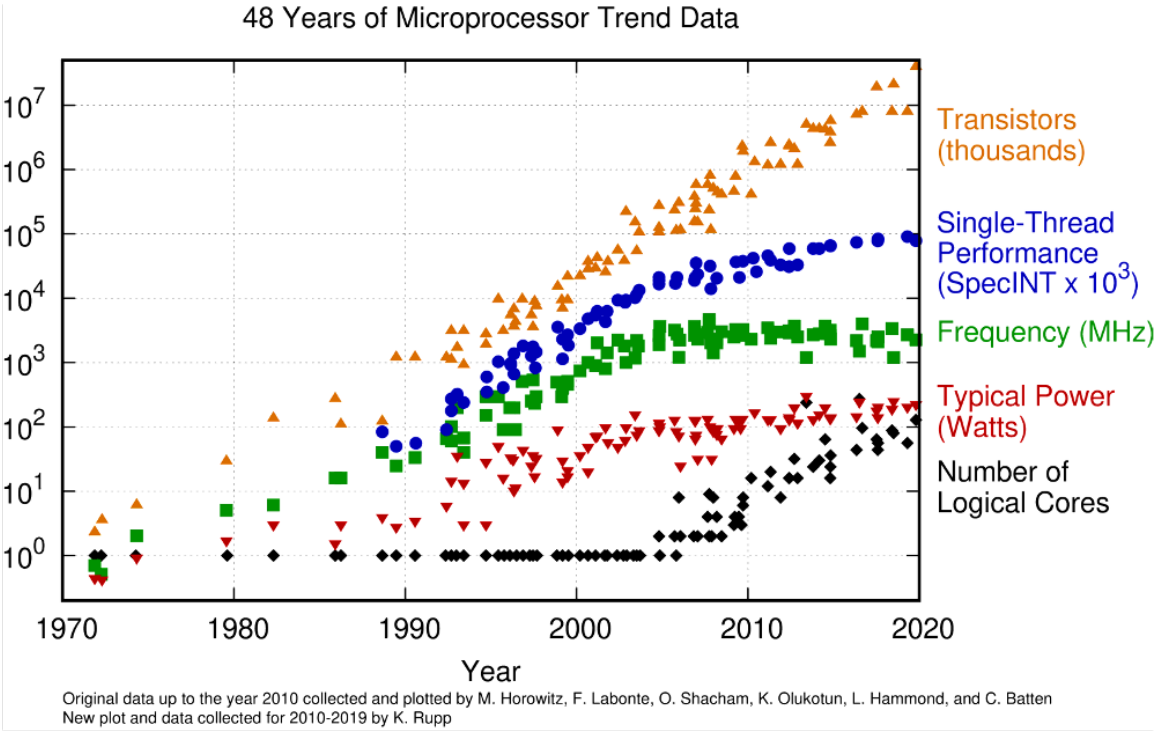


Figure 1.1: Trends in microprocessors from 1972 to 2020, including number of transistors, cores and power usage [110].

wide range of concurrent applications and the guarantees that ensure correct ordering of concurrent operations in both hardware and distributed systems, operations and updates respectively. This chapter also presents challenges that exist for testing methodologies that aim to successfully validate correctness in concurrent hardware and systems. It then concludes with the contributions of this thesis.

1.2 Evolution of Computation and Hardware Heterogeneity

It is important to look at the innovations and trends in the computer industry that lead to the wide range of computing devices used today, and in particular, the recent explosion in parallel and concurrent systems.

Computing has relied heavily on the evolution of transistors and the emergence of integrated circuits [74] allowed early computer chip designers to make complex designs with thousands of transistors. From the development of integrated electronics in this era, there were two trends that enabled computing capabilities to increase exponentially for a period of 50 years. Gordon Moore predicted in 1965 that transistor power densities on integrated circuits would double every two years, which is called Moore's law [108]. The second trend was Dennard scaling, which is the approach by which transistor power densities can remain constant as transistors scaled down in size [42]. These technology trends made transistors scale down in size and allowed for the creation of chips that doubled in numbers of transistors across generations.

Figure 1.1 presents trends in microprocessor data from 1972-2020. It plots the number of transistors (in thousands), single-thread performance (in SpecINT $\times 10^3$), frequency (in MHz), typical power (in Watts) and number of logical cores for different real processors over the presented time period. Due to physical limitations, Dennard scaling stopped in the mid 2000s, indicated in Figure 1.1 by a plateau in the single-thread performance plot. In order to continue to increase processing performance, the computer industry started shifting away from the single-core design paradigm. Instead, architects turned to multicore designs which could harness the increasing number of transistors on a chip while having a constrained power budget [19]. Figure 1.1 demonstrates that after the mid-2000s the number of transistors on a chip continued to increase in correlation with the number of logical cores. Essentially, placing multiple processing cores on a chip enabled continued increases in chips' instruction throughput.

This continued trend of increasing performance enabled by parallel processing and concurrency is one of the driving factors for modern applications and systems. Multicore architectures and devices are part of a heterogeneous ecosystem, including devices ranging from small embedded devices like sensors, to mobile devices, to systems for autonomous driving, to desktop and datacenter processors. The computer

architecture community has also embraced heterogeneity by developing specialized cores, each used for specific tasks in order to increase performance and power efficiency significantly. The use of such application specific accelerators has enabled applications such as machine learning [29] or graph algorithms [63] to execute on a wider range of devices.

1.2.1 Computing Ecosystem Today

Distributed systems have become increasingly important in the last decade due to growth in distributed applications. Distributed systems have increasingly been used to handle storage, synchronization, and computation both for specialized use by a single application (e.g., smart home system runtime) and in combination, building complex system architectures. Computer systems today have multiple layers which span across the datacenter, infrastructure, platform and applications, each of which interfaces with the layer above and below. These system architectures rely on hardware at the datacenter and the availability of high speed networks to successfully enable and scale communication, processing and storage.

Modern applications generate a large amount of data which need to be stored, transferred and accessed effectively [104, 106]. In order to provide the best possible experience to the user, each application has a set of correctness, performance and reliability requirements.

1.3 Correctness in Concurrent Systems

1.3.1 Defining Consistency

A computing ecosystem includes elements that support communication, storage and processing. To perform these functions, systems leverage concurrency. Distributed systems such as distributed shared memory or distributed data stores (e.g., filesystems,

caching systems, databases etc.) execute processes that run on different servers and share data via the network. Data consistency defines the specific rules for the communication between the nodes via the network in a distributed or parallel system. The consistency model essentially specifies which orderings of read, write and update operations from different nodes are allowed and which are forbidden in a parallel program. A system supports a consistency model if the operations on data respect the rules defined by the model. At the lower levels of the architecture, the shift to multicore processors led to the development of techniques to share data in memory and exploit processor parallelism. Software parallelism for multicore processors is achieved by providing programs the ability to generate concurrent reads and writes to the same global memory space, called shared memory. To preserve the correct execution of parallel programs in the face of instruction reorderings stemming from performance optimizations, there is a need for rules to control the legal ordering and visibility of concurrent shared memory accesses [27].

A variety of different consistency models that specify ordering guarantees have been proposed for multiprocessors and distributed systems to meet different program and system needs. We discuss a small subset of these consistency models for each system type in the next subsections.

1.3.2 Consistency Models for Multiprocessors

Sequential consistency, the first memory consistency model for multiprocessors was defined by Lamport [86]. A parallel program consists of multiple threads, which can execute instructions, that access shared data. A parallel program execution is sequentially consistent if it appears to execute as a strict in-order interleaving of the instructions from individual threads. The concurrent execution of threads operating on shared memory can be viewed as a single thread executing some interleaving of these operations without violating the program order of each thread. Sequential

consistency's requirement to preserve per-thread program order globally inhibits performance. For this reason, modern processors turn to other memory models that allow for performance optimizations [5, 34, 109, 111], covered in more detail in Section 2.1.

1.3.3 Consistency in Distributed Storage Systems

In distributed systems, there exists a wide range of application and system requirements for data consistency. More specifically, there are many different consistency models that can provide different combinations of consistency, availability, and performance [25].

Linearizability [67] provides the strongest form of non-transactional consistency in distributed systems and implies that every operation appears to take place atomically, in some order, consistent with the real-time ordering of those operations [79]. Apart from linearizability, there are a large family of consistency models that occupy the space of consistency, availability and performance trade-offs. Causal consistency captures the notion that causally-related operations should appear in the same order on all processes [85]. Eventual consistency provides no guarantees as to when updates become visible in the system but guarantees the updates will eventually become visible in the system [128]. Weaker consistency guarantees, such as eventual consistency, allow ordering of updates that can compromise the correctness of the program or the application executing the updates. Section 2.3 presents the intricacies of consistency models in distributed systems in more detail.

Typically, software systems are accompanied by documents describing their design and expected behavior, called specifications. A specification describes the intended guarantees for the system, but a specific implementation may contain bugs that create a mismatch between what is intended and what is actually provided.

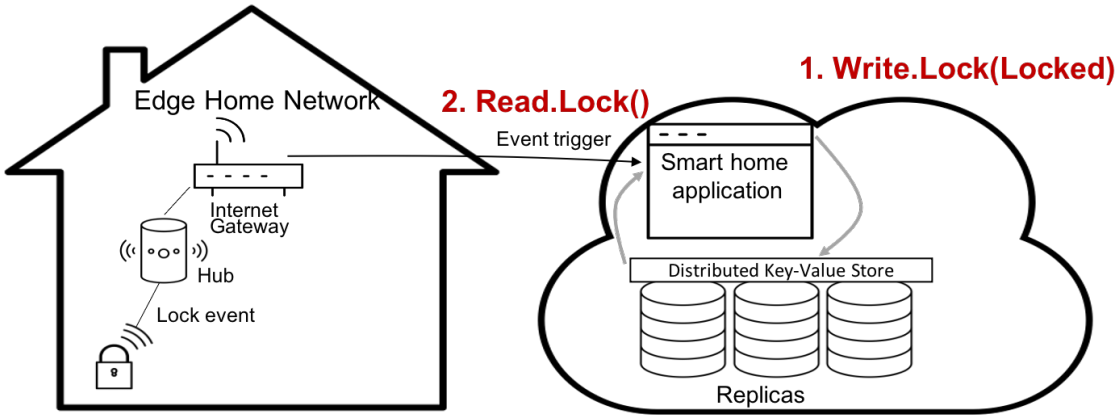


Figure 1.2: Smart Home ecosystem [105].

1.3.4 Example: Ordering in IoT Environments

One example software system is an IoT platform. IoT architectures enable deployment and manage the lifecycle of multi-layer applications whose code executes concurrently on many levels of the architecture. Figure 1.2 presents an example IoT ecosystem for a smart home application. At the edge Home network, there is a smart lock connected to the Internet via a smart home hub and an internet gateway. This smart lock example also presents the system infrastructure which is a cloud-based compute platform that executes the smart home app and serves its data in a key-value store.

For the IoT ecosystem to operate correctly and provide the end user with expected correctness guarantees, all the system components must adhere to the specifications, which is not easy to implement in practice. Table 1.1 presents a test intended to check correctness of a smart lock application in a smart home ecosystem. Suppose the smart lock of Figure 1.2 is initially unlocked and the state of the smart home application for the smart lock, called "Lock", is set to "Unlocked" and is stored in the key-value store. When a lock event occurs at the smart lock device, the event propagates via the home network to the smart home application, where the state of the lock changes to "Locked" with update 1 as seen in Table 1.1. Every time

the smart home application performs a read of the lock value with update 2, the read is expected to return "Locked". However, the smart home application performs read and write operations to the Lock variable, which relies on the distributed key value store to persist them (i.e., hold in non-volatile storage). Different behaviors can be expected of the underlying system depending on the theoretical correctness guarantees chosen for the system and the corresponding implementation. For this scenario, when a write is followed by a read, we only have two potential outcomes: the value being "Locked" or "Unlocked". As users, we expect the written value to become globally available before any other progress is made in the test and the read update completes. In this scenario, the only allowed outcome of update 2 is the "Locked" value from the application's perspective. However, systems exist where "Unlocked" can also be observed due to more or different orderings between updates allowed by the system's correctness guarantees. "Unlocked" is an outcome that may be correct for the storage system, but may not be the right choice for the application. Similar ordering scenarios are possible due to networking protocols with best effort delivery where events are delivered out of order due to the network and due to orderings of operations in multicore hardware where the hardware allows thread operations to be reordered, thus affecting the application behavior.

In order for the expectation of a user of a concurrent application to match the application's behavior, it is essential to study the correctness of individual elements of the concurrent system architecture and validate the system guarantees. This type of potential mismatch between the application expectation and the actual outcome will be revisited throughout this dissertation, with a focus on multicore processors, distributed key-value stores and smart home systems.

Lock initially Unlocked
1.) Lock.Write(Locked)
2.) X = Lock.Read()
Check if $X == Unlocked$ (*Stale Value)*

Table 1.1: A simple test to check if the state of the smart lock in the distributed system architecture returns a stale value after a lock operation.

1.4 The Need for Efficient and Scalable Consistency Testing Methodologies

Concurrent systems require ordering guarantees to mediate between updates of data by concurrent processing elements. This brings the challenge of testing and verifying these complex systems. Erroneous behaviors, violations of correctness properties, faults and concurrency bugs can occur due to synchronization errors across multiple threads as well as across nodes, with significant impact on the normal operation of software. In multiprocessor systems, problematic conditions can appear locally on a node due to thread interleavings erroneously occuring on the hardware. In a distributed setting, a concurrency bug can be caused by the non-deterministic ordering of events distributed across multiple nodes of the system. Such events can be read or write updates, message arrivals or sends, local computation, faults, crashes, and reboots. Diagnosing and fixing such bugs can take a long time to complete in production code. Reproducing bugs is the traditional way of debugging software, but with concurrency bugs, executions are non-deterministic.

Ensuring that systems are correct is essential, and a significant number of resources is spent on testing, validation and verification of systems; these activities can represent as much as 80 percent of total software design cost [82]. Testing is fundamental in the development of software and hardware and is universally used in the development life cycle. Simple testing methodologies have been shown to be capable discovering most critical failures in systems [131]. This dissertation refers to the term “empirical

testing" for any testing methodology that uses a large number of test cases that are run against a system implementation. Unlike formal verification approaches which test systems for provable correctness, empirical testing allows for observation and evidence collected while running tests on the system under test to find violations of correctness. Empirical testing has been very effective at uncovering both unspecified and underspecified behaviors and even finding bugs in faulty system implementations; it has been shown to be successful at finding bugs in different domains, for example in microarchitectural features of hardware [9, 121], GPUs [115], distributed systems [81] and in other domains [130].

In this dissertation, I will explore testing methodologies that target consistency properties in multiple domains of concurrent systems, including parallel hardware, distributed systems and IoT domains, and evaluate their performance, efficiency and ability to uncover outcomes and exercise a system. Listed below are challenges associated with successful testing methodologies:

- **Efficiency**: Testing methodologies need to be fast in order to minimize the time and cost of testing. Increased efficiency can be achieved both by accelerating execution of tests on a system under test, and by improving the testing methodology so that tests can uncover desired outcomes at a higher rate.

- **Scalability**: In order to be able to keep up with rising system complexity, testing methodologies need to be able to run large test suites on the system under test and/or more iterations per test if bugs become harder to observe.

- **Variety**: Typically, each test has a set of distinct outcomes that can appear using a given methodology when the test is run against the system, defined as outcome variety. Outcome variety is a metric that represents the importance for a testing methodology to be able to exercise all the distinct outcomes possible in the testing space, in order to increase testing coverage and increase the testing

methodology's ability to uncover undocumented or undesired behaviors of the system.

1.5 Contributions

This dissertation demonstrates that widely used correctness testing techniques in computer architecture [8] can be effectively adopted in other system domains. This dissertation also rethinks the fundamentals of existing testing methodologies and enhances them to provide significant performance and efficiency benefits.

The specific contributions of this thesis are:

- **An empirical memory consistency testing approach without the requirement for per-iteration synchronization, called perpetual litmus tests [103]:** This dissertation demonstrates the benefits of removing the synchronization bottleneck in litmus tests, a well-established empirical consistency testing approach for multiprocessors. Not only does removal of synchronization accelerate testing, but it also allows the observation of many more and interesting interleavings to appear more efficiently. I introduce a smart encoding scheme leveraging monotonic sequences which enables better analysis and fast heuristics.

- **A software suite capable of generating, running and analyzing perpetual litmus tests from regular litmus tests, called the Perpetual Litmus Engine (PerpLE):** This thesis presents a tool suite that supports synchronization free Memory Consistency Model (MCM) testing, resulting in a 9x speedup over the state of the art. PerpLE also demonstrates four orders of magnitude better ability to uncover test outcomes and can achieve testing goals with less than 1% of the iterations required by the state of the art. In addition, this thesis defines a metric, called outcome variety, that expresses how wide the coverage of distinct outcomes observed during an execution of a litmus test is.

Outcome variety is used to analyze the effectiveness of our consistency testing methodologies in both multiprocessors and distributed key value stores.

- **A software suite capable of applying the concept of litmus tests to distributed key value stores by generating, running and analyzing such tests from regular litmus tests.** Musli Tool is an adaptation of litmus tests in the domain of distributed systems.

- **A measurement study evaluating the use of litmus tests in distributed key value store testing:** Using Musli Tool, this work explores the parameters that affect the efficiency of testing with litmus tests in distributed systems using key-value stores and presents results from testing on real world systems that demonstrate the value of distributed litmus tests. Based on the measurement study, litmus test outcomes are distinguishable across consistency guarantees. The measurement study provides insights into distributed litmus test execution parameters, highlighting that insertion of random delays and the use of asynchronous requests to the distributed system under test are Musli Tool settings that provide benefits in exploring litmus test outcomes.

- **Analysis of smart home systems for correctness [105]:** This thesis presents an analysis of smart home environments, identifying major concurrency and data consistency issues, based on a systematic empirical testing infrastructure. These weaknesses expose platform vulnerabilities that affect system correctness and security.

- **OKAPI, an application-level API that provides strict atomicity and event ordering in Smart Home applications [105]:** In addition to identifying shortfalls of cloud-based smart home platforms, this dissertation proposes design guidelines for smart home systems that preserve application correctness and provides a framework to allow application developers to run applications

without deep understanding of smart home platforms' consistency and concurrency intricacies.

1.6 Dissertation Outline

The remainder of this dissertation is organized as follows. Chapter 2 presents background information on memory consistency models and methodologies to uncover instruction orderings, including an overview of litmus tests. Chapter 2 also presents an overview of fundamental concepts, consistency models and testing methodologies in distributed systems. Chapter 3 focuses on correctness issues of concurrent hardware. More specifically, this chapter discusses consistency testing in multiprocessors and introduces PerpLE, a tool suite to achieve faster and more efficient testing by leveraging *perpetual litmus tests*, a new variant of litmus tests. Chapter 4 outlines correctness issues in distributed key value stores and adapts techniques that have proven useful in memory model testing to consistency models of distributed systems, demonstrated by a measurement study. Chapter 5 presents scenarios that compromise correctness in smart home systems and a measurement study of the frequency of their occurrences in a commercial smart home system and a framework to mitigate on. Finally, Chapter 6 presents ongoing and future research directions and subsequently concludes this thesis.

Chapter 2

Background and Related Work

This chapter presents background for different system domains of IoT ecosystems, multiprocessors, distributed systems and smart home systems that are the focus of this dissertation work. Section 2.1 focuses on consistency models and testing methodologies for uncovering instruction, update, and event orderings, and it serves as the foundation for the remainder of this dissertation. Section 2.2 provides an overview of the fundamentals in memory consistency model testing and analysis and introduces litmus tests. Section 2.3 provides a summary of the features and characteristics of distributed systems that differentiate how consistency models and consistency testing operate in this domain and Section 2.4 concludes this chapter.

2.1 Memory Consistency Models

2.1.1 Sequential Consistency

In a multicore context, memory consistency models define the rules that determine the ordering of concurrent loads and stores made by different threads of execution in a single program, and therefore determine which values can be legally returned to program loads. In Lamport's *sequential consistency (SC)* model [86], all threads see

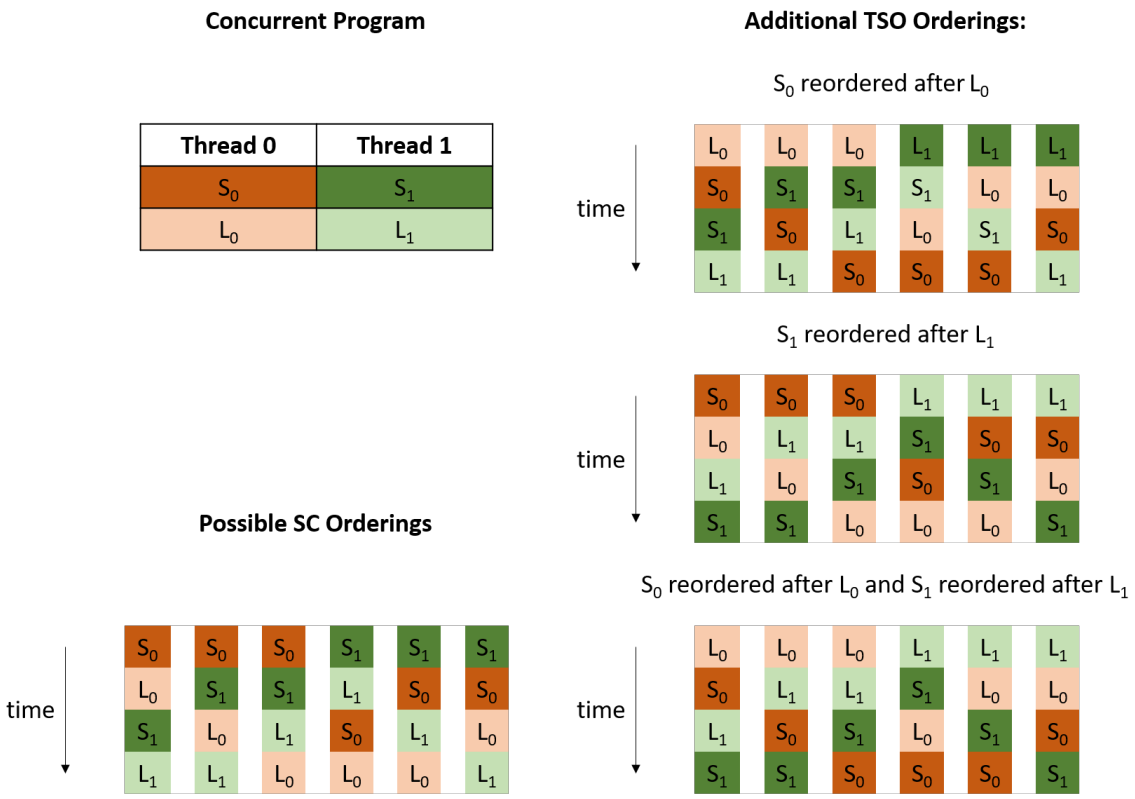


Figure 2.1: Possible memory operation orderings under SC (left) and TSO (right). S_0 and S_1 are stores, while L_0 and L_1 are loads. Under SC, the possible orderings (6 total) are interleavings of the memory operations of each thread in program order. TSO relaxes the program order requirement by allowing store buffering, so stores can appear to take effect later, yielding additional allowed orderings (18 additional ordering, 24 total).

loads and stores to memory in the same globally agreed-upon order. Consequently, the concurrent execution of shared memory operations by different threads can be viewed as a single thread executing some interleaving of these operations without violating the program order of each thread, as shown in the bottom left of Figure 2.1.

Even though sequential consistency is very intuitive, it is rarely found in real systems, because it limits memory system concurrency [22, 87]. Modern processors usually relax the guarantee that the per-thread program order is preserved globally and turn to other weaker memory models instead to improve performance.

2.1.2 TSO

One such weaker memory model is Total Store Ordering. Total Store Ordering (TSO) can be informally understood as the memory model that is obtained from SC by introducing store buffering to reduce the latency of store operations [117]. A store buffer is a microarchitectural structure that exists in CPUs for allowing the CPU to speculate on stores [66]. Each thread can load its own stored values early, directly out of its store buffer, but all threads share the same view of the global order in which stores appear to have occurred. This can provide additional allowed orderings in some cases, as shown in the right column of Figure 2.1.

Owens et al. have shown that a version of TSO is consistent with the behavior of x86 processors across a number of test cases [109, 111]. As such, TSO is one of the predominant memory consistency models one encounters when analyzing the memory behavior of shared memory architectures. Chapter 3 focuses on empirical testing applied in TSO scenarios, but the methodologies presented are applicable to weaker models as well.

2.2 Uncovering Instruction Orderings

2.2.1 Litmus Tests

In order to test that software and hardware systems adhere to their specified memory models, small stressmark tests known as *litmus tests* are often used in multiprocessors [116]. Litmus tests consist of simple combinations of shared memory loads and stores. Figure 2.2 shows three example litmus tests: store buffering (sb), load buffering (lb) and podwr001. For each litmus test, T is used to denote the total number of test threads and $T_L \leq T$ to denote the number of test threads that perform loads.

sb

Initially [x] = 0, [y] = 0	
Thread 0	**Thread 1**
(i_{00}): [x] ← 1	(i_{10}): [y] ← 1
(i_{01}): reg_0_0 ← [y]	(i_{11}): reg_1_0 ← [x]

lb

Initially [x] = 0, [y] = 0	
Thread 0	**Thread 1**
(i_{00}): reg_0_0 ← [y]	(i_{10}): reg_1_0 ← [x]
(i_{01}): [x] ← 1	(i_{11}): [y] ← 1

podwr001

Initially [x] = 0, [y] = 0, [z] = 0		
Thread 0	**Thread 1**	**Thread 2**
(i_{00}): [x] ← 1	(i_{10}): [y] ← 1	(i_{20}): [z] ← 1
(i_{01}): reg_0_0 ← [y]	(i_{11}): reg_1_0 ← [z]	(i_{21}): reg_1_0 ← [x]

Figure 2.2: Store buffering (sb), load buffering (lb) and podwr001 litmus tests. Note: podwr001 extends sb to 3 threads. (i_{tn}) is the nth test instruction of thread t, $[x]$ and $[y]$ are shared memory locations and reg_t_r is register r of thread t. For sb and lb, $T = T_L = 2$. For podwr001, $T = T_L = 3$.

In each iteration of a litmus test, an outcome is generated when run against a system in practice. Running a litmus test can produce one of a set of possible *outcomes*, depending on the values loaded from shared memory during test execution. Each outcome can be expressed as a number of *conditions* involving register values. For example, the sb test as presented in Figure 2.2 has 4 possible outcomes, each consisting of 2 conditions: $reg_0\_0 = 0$ && $reg_1\_0 = 0$; $reg_0\_0 = 0$ && $reg_1\_0 = 1$; $reg_0\_0 = 1$ && $reg_1\_0 = 0$; and $reg_0\_0 = 1$ && $reg_1\_0 = 1$.

Depending on the specifications of the consistency model under test, an outcome can be *Permitted* or *Forbidden* by the consistency model. Also, based on the system implementation, an outcome can be *Observable* or *Unobservable* by the system. If an outcome is *Forbidden* by the consistency model but is in practice *Observable* by the system, then that constitutes a bug and a mismatch between the specification and the implementation. The remaining combinations of Permitted/Forbidden and

Observable/Unobservable are correct, but can be stricter than necessary (*Permitted* and *Unobservable*).

The first outcome of the store buffering litmus test in Figure 2.2 ($reg_0\_0 = 0$ && $reg_1\_0 = 0$) requires the hardware to implement store buffering in order to occur, i.e. it cannot occur under SC by simply interleaving the instructions from each thread. For this reason, it is the most informative outcome in terms of hardware capabilities, letting us distinguish between different possible consistency models. Each litmus test has at least one such outcome, which is called the *target outcome*. After a test run of multiple iterations, a subset of these distinct outcomes is generated and the frequency of each outcome occuring can be measured; this subset is called the *outcomes of interest*.

In an empirical execution in which concurrent timings can vary, the outcomes of many iterative executions of a litmus test indicate which interleavings of memory accesses occurred. These observed interleavings can then be checked against the model that the system claims to implement to ensure they are allowed. Since there is a non-deterministic element in individual runs of a litmus test, such approaches typically cannot guarantee that all possible interleavings have been exercised.

In practice, widely available tools for litmus testing use a combination of two approaches to address this non-determinism [6, 8]. First, they run large numbers of iterations of the litmus test to give a statistically more accurate picture of how frequently each outcome is expected to appear. For models not yet formally specified, this can aid attempts at formulating a formal description. Second, testing suites might apply further stress on the system, like frequent memory operations to addresses not used by the test, to check whether the distribution of outcomes is affected because it changes the interleavings of memory accesses in the litmus test. Especially in GPUs, recent work has shown that such methods can be very effective [115, 119].

2.2.2 Happens-Before Graphs

Each litmus test outcome offers information on the memory operation ordering that gave rise to it, which can be revealed using a *happens-before graph*. Happens-before graphs are essential in modeling litmus test executions; by building a happens-before graph for each outcome, litmus test outcomes can be classified as allowed or forbidden under a particular consistency model. Using the happens-before graphs we can set the ground truth for a given litmus test for the system under test. Constructing a happens-before graph starts by considering the different shared memory operations in a litmus test as vertices in a graph. Edges are then added to represent temporal relationships between individual operations, based on the outcome of a specific execution of the test. Happens-before edges are meant to represent temporal relationships and as such are transitive. Alglave [5] provides formal descriptions of four types of such edges between two memory operations m_1 and m_2, summarized informally below:

- Program order (po) edges: a po edge from m_1 to m_2 means a sequential processor executes m_1 before m_2.
- Read-from (rf) edges: an rf edge from a store m_1 to a load m_2 means that m_2 loads the value stored by m_1.
- Write serialization (ws) edges: assuming m_1 and m_2 are both stores to the same memory location x, a ws edge from m_1 to m_2 means that m_1 updates x before m_2.
- From-read (fr) edges: an fr edge from a load m_1 to a store m_2 means that m_1 loads a value stored by an instruction earlier than m_2 in ws order.

2.3 Overview of Distributed Systems

This section introduces key-value storage systems and discusses basic notions of distributed systems, time synchronization, replication, availability and consistency that will be used to build testing methodologies for distributed systems, as discussed in Chapter 4.

2.3.1 Key-value Stores

A key-value store, or key-value database, is a database that uses a simple data model leveraging an associative array (e.g., map or dictionary) where each unique key is associated with a single value in a collection, referred to as a key-value pair [4]. Values are stored as blobs requiring no upfront data modeling or schema definitions, unlike in relational databases. The storage of a value as a blob removes the need to index the data to improve performance. Key-value stores have no query language but allow storage, retrieval and updates of data using simple get, put and delete operations. The benefits of key-value stores are that they provide high performance, ease of use and scalability.

2.3.2 Notions of Time in Distributed systems

In order to control the order of operations in shared-memory multicore processors, hardware synchronization is used [116]. In centralized systems (including CPUs), there is no need to perform clock synchronization as there is generally only a single clock. A process can retrieve the time by issuing a system call to the kernel and when another process tries to get the time at a later time, it will get a higher time value.

Dealing with time is a harder problem at the level of distributed systems. A collection of processes running concurrently need to synchronize and coordinate efficiently with each other in order to perform updates over shared data. These

processes may share data indirectly through a key-value store they all access, and coordinating accesses from multiple nodes can be a challenge. In particular, there is no global clock or common memory and each node in the system maintains its own view of time, typically using a hardware on-chip clock device. Physical time on each node is in general not going to be perfectly in sync with other nodes in the system. In addition, clocks on different networked machines synchronize with each other over the network, with additional network latencies involved. As a result, there is no "absolute" time in a distributed system to refer to and improving techniques for clock synchronization is a topic of systems research. Obtaining perfect clock synchronization can provide a means to order events absolutely, regardless of which node an event originated at, and allow forward progress with high performance and global ordering guarantees. In practice, there are different time synchronization methods used that do not quite provide one global clock, which will be introduced next.

Logical clocks

The concept of logical clocks was first introduced by Lamport in 1978 for timestamping and ordering of events in a distributed system [85]. Logical clocks are used to create a partial or total ordering of events, based on the concept of a "happens-before" relationship explained previously. Wall clock information is completely ignored and events are ordered by just tracking logical causality relations between events. Relations between events include processes sending and receiving messages as well as executing local events. If events a and b are local and a occurred before b, then a happened-before b. If a is a sending of a message m and b is the recipient of m, then a happened-before b. Events a and b can also be concurrent, if both a happened-before b and b happened-before a are false. By applying these rules transitively, logical clocks use a mathematical function to assign numbers to events. These numbers

act as timestamps and allow partial or total logical ordering of events based on "happens-before" relationships.

Vector Clocks

Vector clocks [112] are a more advanced version of logical clocks that derive more information about the ordering of events based on their timestamp. With logical clocks, if two events are not causally related (e.g., when there is no communication between the processes), the order of events cannot be determined. With vector clocks, a vector of N logical clocks is kept for each process to keep track of causality, initially initialized to zero [112]. Every time a process experiences an internal event, it increments its own logical clock in the vector by one. Also, every time a process sends a message, its own logical clock is incremented by one and then a copy of its vector is sent. Similarly, when a process receives a message, its own logical clock is incremented by one and each element of the vector is updated by taking the maximum of the value between its own vector and the vector received.

True Time

An alternative clock synchronization method was introduced by Google Spanner [33]. GPS and Atomic clocks are used by Spanner to correct the time and provide guarantees of very small clock skew and uncertainty. The API that leverages specialized hardware for fine grained clock synchronization is called True Time. The need for True Time arose because the number of servers in Spanner can be in the order of tens of thousands, and systems tracking causality are very costly both in terms of storage and the ability to query on the causal relationships. With True Time, Spanner manages to reduce the size of the uncertainty intervals to be negligible and order events using wall clock time, which reduces the need for any communication between two events in order to provide ordering [41]. Spanner assigns transactions commit timestamps that are globally

meaningful and reflect a global order. If a transaction T1 commits (in absolute time) before another transaction T2 starts, then T1's assigned commit timestamp is smaller than T2's.

2.3.3 Replication, Availability, Performance and the CAP Theorem

Before looking at consistency properties and models in distributed systems, it is important to understand the concept of replication, and the trade-offs between consistency, availability and performance with respect to replication.

Replication

Data replication is the process of making multiple copies of data and storing them at different locations across a distributed system. There are two primary reasons for replicating data: reliability and performance [120]. First, data are replicated to increase the reliability of a system, since a replicated system can possibly continue working after one replica crashes by switching to another replica. By maintaining multiple copies, it becomes possible to provide better protection against corrupted data and write operations that fail. Second replicating improves performance. Replication aids performance when a system is scaling in the number of nodes and geographically by replicating the server and subsequently dividing the work amongst replicas. When scaling geographically, copies of data are placed in the proximity of the process using them, and therefore access latency to the data decreases. However, the challenge of keeping copies of the data up to date arises with data replication, leading to consistency problems. Whenever a copy is modified, that copy becomes different from the rest. Consequently, modifications have to be carried out on all copies to ensure consistency. When and how those modifications need to be carried out is specified by the consistency model and determine the price of replication.

CAP Theorem

When systems scale and involve thousand of nodes and millions of lines of code, it is critical for the systems to be available [59]. Multiple sources of faults can make a system unavailable, including hardware, software faults and network outages. A wide range of techniques for fault tolerance exist to mitigate these problems [37, 61, 120].

The CAP theorem tries to summarize the relationships between consistency, availability and partitions. The theorem states that any networked shared-data system can have at most two of three desirable properties: 1) consistency (C) equivalent to having a single up-to-date copy of the data, 2) high availability (A) of that data (for updates), and 3) tolerance to network partitions (P) referring to a network split caused by device failure [24, 25, 26, 54]. For example, a system with only a single copy of a shared data object is easy to keep consistent, but the system's availability and tolerance to partitions is poor. The purpose of this theorem is to navigate the system design space, enabling applications to choose the best combination of consistency and availability that fits their needs. The effects of this theorem become clearer when thinking of the behavior of nodes under the presence of a partition. Under a partition if at least one node updates state, nodes in the system will become inconsistent because nodes in the other partition cannot be updated, which violates consistency. Similarly, if the system's goal is to preserve consistency under a partition, one side of the partition will become unavailable. When nodes communicate it is possible to maintain both consistency and availability but nodes cannot tolerate partitions. In practice, systems cannot assume partition tolerance; system design needs to navigate consistency and availability trade-offs. The next section presents design choices that focus on either availability, such as eventually consistent systems on key-value stores, or consistency, such as systems designed to provide strong consistency guarantees like linearizability.

2.3.4 Consistency Models in Distributed Systems

The emergence of the cloud has been a catalyst for a range of application and service domains. More specifically, developers now have the ability to utilize cloud infrastructure to develop complex software systems and applications quickly. However, leveraging the distributed cloud infrastructure signifies that all software or applications developed in the cloud rely on the cloud systems' correctness. Ensuring the correctness of these cloud systems is therefore necessary for ensuring correctness of user applications. This correctness is often critical to the users, e.g., for example when the software controls access to users' smart homes or is responsible for bank transactions.

In distributed systems, different degrees of data consistency can be provided [127]. More specifically, there are many different consistency guarantees that can provide different combinations of consistency, availability, and performance. These consistency guarantees range from strong consistency, ensuring that only consistent state can be seen at the cost of high latency, to a range of weak consistency guarantees that prioritize low latency. Here we discuss several consistency guarantees of distributed systems drawing from Section 2.3.3's discussion of tradeoffs in preparation for Chapter 4.

Linearizability

Linearizability [67] is one of the strongest single-object (non transactional) consistency models. Linearizability is a single-object model, which means that systems provide linearizability on individual keys in a key-value store. Linearizability implies that every operation appears to take place atomically, in an order that is consistent with the real-time ordering of those operations. This means in practice that if an operation a completes before an operation b begins, then a should logically take effect before b. Linearizability, however, can cause a system to become unavailable in the event of a network partition, meaning some or all nodes will be unable to make progress.

Causal Consistency

Causal consistency requires that operations related causally to each other appear in the same order on all processes. For example, a process P1 performs a write with value 1 to shared object x and process P2 performs a read of object x that returns value 1 followed by a write to the same object that writes value 2. Process P2 observes and reads the earlier write by process P1 and therefore, the two writes are causally dependent. For causally independent operations, processes may disagree about the order of said operations [79]. For example, consider the following causally-independent operations on a social medium: i) removing a boss from a list of friends and ii) posting to friends "Time for a new job". If these two actions become visible in the wrong order, then the boss will observe the post, which was unintended by the user. In a causally consistent system, both orderings are possible. Systems implementing causal consistency remain available even in the presence of network partitions.

Client-centric Consistency Guarantees

In distributed systems, there exist consistency guarantees that are client-centric; these guarantees concern the consistency of accesses to a data store by a single client. These guarantees include *Monotonic Reads, Monotonic Writes*, and *Writes Follow Reads*. *Monotonic Reads* and *Read Your Own Writes (RYOW)* only apply to operations performed by a single process. The *Monotonic Reads* guarantee defines that if a process performs a read r1, followed by a read r2, then r2 cannot observe a state prior to the writes which were reflected in r1; intuitively, reads cannot go backwards. Under Monotonic Reads all nodes can make progress in the presence of partitions. Read Your Own Writes requires that if a process performs a write w, followed by that same process performing a subsequent read r, then r must observe w's effects. In the case of a network partition, every node can make progress, as long as clients never change which server they talk to. Monotonic Reads is a consistency guarantee that is similar

to store buffering in CPUs; store buffers hold store operations that need to be sent to memory so loads can bypass stores to other addresses [116]. Monotonic Writes ensure that if a process performs a write w1, followed by a write w2, then all processes observe w1 before w2. Finally, Writes Follow Reads ensures that if a process reads a value v, which came from a write w1, and later performs a write w2, then w2 must be visible after w1, meaning that once read, a read's past value cannot be modified.

Eventual Consistency

Linearizability and eventual consistency are two guarantees on opposite ends of the consistency spectrum. Eventual consistency is an example of a weak consistency guarantee. It provides no guarantees as to when updates become visible in the system [128]. Weakly consistent systems allow ordering of updates that can compromise the correctness of the program or the application executing the updates, if it was written for a stronger model. Although many systems are designed to permit weak orderings, strongly consistent systems forbid these behaviors.

Tunable Consistency in Cassandra

Cassandra is a distributed key-value store designed to manage large amounts of data while providing high availability without single points of failure. Cassandra aims to provide horizontal scaling on clusters that can span hundreds of nodes. Cassandra performs data partitioning and keeps N replicas of each partition on different nodes. Cassandra implements tunable quorums, where a majority of the replicas must respond to complete an update, to enable consistency guarantees that are configurable and keep replicas up to date [51]. Cassandra will be used extensively in Chapters 4 and 5.

Cassandra supports a per-operation trade-off between consistency and availability through consistency levels. In Cassandra, consistency levels are consistency mechanisms where operators can configure the number of nodes that must participate in

reads (R) and writes (W) to be larger than the replication factor (RF), a version similar to Dynamo's $R + W > RF$ [40]. The consistency level controls how many responses from replicas the coordinator performing a data access waits for before responding to the client. For example, if RF = 3, a QUORUM request will require responses from at least 2 out of 3 replicas. Generally writes will be visible to subsequent reads when the read consistency level contains enough nodes to guarantee a quorum intersection with the write consistency level. If QUORUM is used for both writes and reads with RF=3, at least one of the replicas is guaranteed to participate in both the write and the read request, which in turn guarantees that the quorums will overlap and the write will be visible to the read [28]. Write operations are always sent to all replicas, regardless of consistency level. For read operations, the coordinator only issues enough read commands to replicas to satisfy the consistency level.

If this type of strong consistency is not required, lower consistency levels like ONE may be used to improve throughput, latency, and availability. ONE will succeed even if only a single replica is available in any datacenter.

2.3.5 Testing methodologies for Distributed Systems

All software systems have bugs that can affect the software's correctness, performance and energy efficiency. Bugs can also compromise the software's security. Debugging software is not an easy task; there are numerous debugging tools available to aid in the search of bugs, but debugging is a process that might not be fully automated or provide coverage for the system tested. The problem of debugging becomes increasingly complex in concurrent systems, as software can run on thousands of instances of machines that coordinate with each other over a network and are prone to faults. The network delays and faults introduce non-determinism which can cause bugs that are difficult to replicate as they can be a result of intricate timing and ordering of updates or a result of the presence of faults.

2.4 Conclusion

This chapter presented background information for consistency guarantees in concurrent systems and fundamentals used in consistency testing throughout this dissertation. First, this chapter presented the Sequential Consistency and Total Store Order (TSO) Memory Consistency Models (MCMs) and introduced litmus tests and happens-before graphs, fundamental concepts used in this dissertation. Chapter 3 builds on litmus tests and happens-before graphs to improve the speed and efficiency in memory consistency testing. This chapter also presents key concepts around distributed systems and distributed systems' consistency guarantees. Chapter 4 leverages litmus tests to adapt methodologies targeting consistency testing to distributed key-value stores and Chapter 5 identifies ordering and consistency related deficiencies in commercial smart home environments and introduces a tool to mitigate them.

Chapter 3

Improving the Speed and Effectiveness of Memory Consistency Testing

As indicated in Chapter 2, correctness is essential but hard to achieve in a complex computing ecosystem. Correctness deficiencies are hard to pinpoint in a generalized manner as concurrency exists on multiple levels. To test for correctness properties empirically and trace the source of incorrectness, this thesis focuses on specialized testing methodologies at different layers of the computing ecosystem, using a bottom-up approach.

This chapter concentrates on CPU correctness issues[1]. Even as most of today's computer systems have turned to parallelism to improve performance, the systems' memory consistency models can often remain underspecified, incomplete or even incorrectly implemented [121]. This leads to programmer confusion and to buggy concurrent systems. Section 3.1 discusses how existing tools for empirical memory consistency testing rely on large numbers of iterations of simple multi-threaded

[1]The work in this chapter was completed in collaboration with student co-author Markos Markakis and was published in [103].

litmus tests to perform correctness testing. This approach typically employs thread synchronization after every iteration of the litmus test, which imposes a significant overhead and can reduce testing performance.

3.1 Introduction

As traditional approaches to increasing system performance have been constrained by the end of Moore's Law/ Dennard scaling, exploiting parallelism has become the primary way to further improve performance across system types [35, 36, 68]. When working with these parallel systems, understanding their memory consistency models is critically important for correct design and programming. Otherwise, the resulting systems and applications can contain correctness bugs that manifest as subtle, non-deterministic errors [115].

As hardware complexity rises, it is becoming increasingly challenging to ensure that a specific implementation conforms to the memory model it claims to implement, generating the need for thorough testing. Time spent in validation and verification can be more than half of the hardware design effort [49, 50], which highlights the need for approaches that can accelerate these processes; faster methodologies can, in turn, reduce cost and improve time-to-market. While some comprehensive formal techniques exist for exploring litmus test execution [94, 95], industry testing of real hardware also relies heavily on empirical and probabilistic testing [72], as it is likely to provide results at a significantly shorter time scale. Most current approaches to this type of testing leverage litmus tests, designed to expose different orderings of memory operations. Orderings that manifest in the empirical execution of litmus tests are called *observable* for an implementation. Observing an ordering that the system's published memory model lists as forbidden indicates an implementation bug;

to maximize the probability of detecting bugs, empirical testing tools aim to expose as large a variety of outcomes as possible.

Currently, tools achieve outcome variety by executing litmus tests iteratively, with different orderings arising probabilistically during test execution due to factors like system load and thread timing [72]. However, each litmus test might need to be run thousands or even millions of times before sufficiently varied testing outcomes are observed [115]. The frequency with which a given test outcome, indicative of a particular ordering, can be observed depends on: (i) the extent to which the implementation under test favors it (including whether it is technically feasible) and (ii) the ability of the testing approach to create the conditions that would reveal it. For less frequent outcomes, some testing approaches may require executing large numbers of iterations, taking significant amounts of testing time, to observe a desired outcome. Our new testing framework, PerpLE, is designed to more efficiently expose and analyze a wider range of orderings than existing approaches, improving the effectiveness of empirical testing approaches.

A state of the art tool for empirical testing of memory models on multiprocessors is litmus7 [72]. Litmus7 enables running of tests against parallel hardware. While tools like litmus7, provided by the *diy* suite [8, 72], are effective empirical testing frameworks, many, including most configurations of litmus7, rely on synchronizing the participating threads before every test iteration. Such synchronization is critical to ensure that different threads execute their part of the test sufficiently close in time for the testing tool to observe their interaction via shared memory, but it can have negative implications. The synchronization overhead dominates runtime, significantly slowing down testing and reducing the total number of iterations executed. For example, based on experiments with litmus7 using the default (`user`) synchronization mode and for different iteration counts of the store buffering (sb) litmus test, synchronization overhead never falls below 85% of total execution time. Furthermore, the

tight synchronization might reduce the number and type of orderings that are ever experienced during the iterative test. Lastly, iterative synchronization-based testing can be ineffective for systems that incur long synchronization overheads compared to CPUs, e.g. for example GPUs, as well as for systems not optimized for performance, like many mobile processors [21]. In fact, consistency testing approaches in these systems exhibit orders of magnitude lower performance compared to microprocessors [115]. Empirical testing of memory models in these systems might also be hindered by multi-core memory interference that can slow programs down by over 100x [73].

Our work on PerpLE aims to increase the opportunities to observe different outcomes per unit time. This approach rethinks the design of litmus tests to eliminate per-iteration synchronization, thus reducing the overall testing time. At the same time, letting threads run longer without synchronizing creates more opportunities for interesting interactions, leading to a greater variety of outcomes and, by extension, of orderings. While other synchronization-free litmus testing tools exist, PerpLE offers advantages over existing approaches. Specifically, this work offers the following contributions:

- We propose an empirical memory consistency testing approach without the requirement for per-iteration synchronization. This approach is based on *perpetual litmus tests*, a concept defined and analyzed in this chapter.

- This chapter demonstrates a method for converting litmus tests to their perpetual counterparts and generating a suite of perpetual litmus tests.

- The work culminates in a software suite capable of generating, running and analyzing such tests from regular litmus tests, called the Perpetual Litmus Engine (PerpLE).

- We present an algorithm for detecting outcomes of interest in *perpetual litmus tests*. We also present a linear heuristic that allows PerpLE to scale to millions

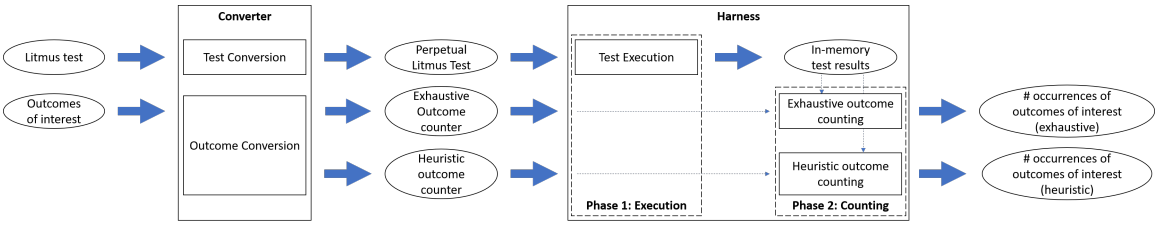


Figure 3.1: PerpLE control flow diagram for generation, execution and analysis of perpetual litmus tests.

of test iterations while maintaining a rate of outcome discovery that is orders of magnitude higher than existing approaches.

- PerpLE is evaluated on an x86 system, showing increased outcome variety and target outcome occurrence rate compared to a variety of litmus7 modes, including its synchronization-free mode.

- PerpLE is found to achieve a runtime speedup over all litmus7 synchronization modes (8.89x over `user` mode and 2.52x over the synchronization-free, `none` mode).

- Finally, PerpLE is also found to observe an increase of over four orders of magnitude over the default synchronization mode of litmus7 (`user` mode) in the chosen metric of outcomes of interest observed per unit time, which makes testing practical across parallel hardware systems.

3.2 PerpLE Tool Suite Overview

As Figure 3.1 illustrates, the proposed approach consists of two steps, each performed by a separate tool. First, the *Converter* converts an input litmus test to a format capable of exposing the same interleavings as the original test, but without requiring per-iteration thread synchronization, which is called a *perpetual litmus test*. This test is then executed by the *Harness*, which keeps the test run results in memory.

Meanwhile, the Converter also produces the *exhaustive outcome counter* for this particular litmus test and set of outcomes of interest. The exhaustive outcome counter is a function that the Harness can apply to the in-memory test results, once all iterations of the test have been executed, to determine how many times each outcome of interest occurred. Alongside the exhaustive outcome counter, the Converter produces the *heuristic outcome counter*, a function with the same inputs and outputs as the exhaustive outcome counter, but which only searches part of the results space and can be dramatically faster.

Section 3.3 presents the methodology for test conversion, while Section 3.4 deals with generating the exhaustive and heuristic outcome counter functions.

3.2.1 Synchronization in Litmus Test Executions

Since litmus tests tend to only be a few instructions long, the execution time of a single iteration is very short. Unless we synchronize before each iteration, it is most likely that the participating threads will execute their corresponding parts of the test at different points in time, so interactions among individual memory operations executed by different threads will be unlikely. Perpetual litmus tests are designed to adjust for this effect by eliminating per-iteration synchronization.

Moreover, determining the outcome of a litmus test requires comparing register values from different threads at the end of each iteration. Each thread t that performs loads has a designated array buf_t where it stores the values that were loaded into its registers in each iteration for later analysis. Testing tools do not launch the next iteration of the test on any thread until this information has been collected, to avoid overwriting the registers of interest.

For these reasons, a synchronization barrier is enforced across test threads, as seen on the left side of Figure 3.2 for the sb test from Figure 2.2. The barrier is a blocking

call which guarantees that the litmus test threads start executing each iteration of the test simultaneously.

3.3 Removing Synchronization Using Arithmetic Sequences

To address the overhead created by the synchronization required after every iteration, called a barrier, perpetual litmus tests are introduced. Perpetual litmus tests maintain the core approach of litmus tests, but remove per-iteration thread synchronization. Perpetual litmus tests synchronize test threads upon launch, but then do not synchronize them again until all iterations are complete. Since synchronization no longer keeps the test threads in lockstep across iterations, each test thread can run behind or ahead of the others. However, as long as a large number of test iterations are executed consecutively, each iteration of a given test thread has the chance to interact with *some* iteration of another test thread. In this setting, storing the same integer value to memory in each test iteration would be ambiguous, since multiple different store operations (from different iterations of the same thread) might have stored the same value. As such, it is not possible to reason about the ordering of specific instructions based on loading that single integer value. To address this challenge, guaranteeing the *uniqueness* of each stored value is needed across the entire run of a perpetual test, which is achieved using arithmetic sequences.

With perpetual litmus tests, each integer value that appears in a store operation of the original litmus test is mapped to a monotonically increasing arithmetic sequence of values. This lets us distinguish between stores from different iterations and eliminate ambiguity. In particular, storing a positive integer value a to a shared memory location $[mem]$ is replaced by storing an element of the arithmetic sequence $k_{mem} \cdot n_t + a$. Here, k_{mem} is the number of different integer values that appear in store operations to mem

Litmus Test Loop Body (sb)	
Initially [x] = 0, [y] = 0	
Thread 0	**Thread 1**
barrier	*barrier*
(i_{00}): [x] ← 1	(i_{10}): [y] ← 1
(i_{01}): reg_0_0 ← [y]	(i_{11}): reg_1_0 ← [x]
buf_0[n] ← reg_0_0	buf_1[m] ← reg_1_0
zero out [x], [y]	*zero out [x], [y]*

Perpetual Litmus Test Loop Body (sb)	
Initially [x] = 0, [y] = 0	
Thread 0	**Thread 1**
(i_{00}): [x] ← n + 1	(i_{10}): [y] ← m + 1
(i_{01}): reg_0_0 ← [y]	(i_{11}): reg_1_0 ← [x]
buf_0[n] ← reg_0_0	buf_1[m] ← reg_1_0

Figure 3.2: Store buffering (sb) litmus test and its perpetual version. n and m are iteration indices for threads 0 and 1.

across all threads of the original litmus test. The iteration index n_t, which starts at 0, specifies which element of the arithmetic sequence should be written at each iteration of thread t.

Additionally, perpetual litmus tests still need access to all values loaded into registers during testing, in order to determine test outcomes at the end of the run. To achieve that, the buf arrays of the original approach are kept, maintaining the same storage complexity of $N \cdot T_L$. The size of each buf_t varies depending on the number of loads per iteration executed by thread t, i.e. T_L. For example, for a run of N iterations indexed by n, if thread t performs r_t load operations per iteration into registers reg_t^0 through $reg_t^{r_t-1}$ respectively, buf_t will need to be of size $r_t N$ and the value of each reg_t^i will be saved into $buf_t[r_t n + i]$.

Figure 3.2 shows this conversion for the sb test, where n and m are the iteration indices of threads 0 and 1 respectively. Since only the value 1 is stored to $[x]$ by an instruction in the original test, in particular in (i_{00}), k_x is 1 and so the corresponding instruction (i_{00}) of the perpetual test now stores the value $n + 1$. Similarly, storing 1 to $[y]$ in (i_{10}) has been replaced by storing $m + 1$, since k_y is also 1. Thus, by leveraging arithmetic sequences, each stored value can be expressed as a function of the iteration that stored it. This means resetting shared memory locations to 0 at the end of each iteration is not needed, since it is possible to distinguish previously

Line of litmus test	**Line of perpetual litmus test**
$[mem] \leftarrow a, a \in \mathbb{Z}^+$	$[mem] \leftarrow k_{mem} \cdot n_t + a$
$reg_t \leftarrow [mem]$	$reg_t \leftarrow [mem]$
any fence	the same fence
(iteration end) $buf_t \leftarrow reg_t, \forall reg_t$	*(iteration end)* $buf_t \leftarrow reg_t, \forall reg_t$

Table 3.1: Perpetual litmus test conversion paradigm.

existing values from newly stored values. Each thread of the perpetual litmus test will run independently, incrementing indices (i_{0n}) and (i_{1m}) by one for each iteration. The values stored at each one of these indices are specified by the values of $n+1$ and $m+1$. In the litmus test loop body and the perpetual litmus test loop body each thread stores the values for registers reg_0 and reg_1 at the corresponding iteration numbers into the buf arrays. The perpetual store buffering litmus test omits the per-iteration initialization process.

Note that load operations or fences appearing in the original litmus tests can remain unchanged. In particular, loads can proceed as usual because the shared memory locations and thread registers are still used by the perpetual litmus test, while fences within the test itself will have the same effect they had in the original litmus test. The process of converting original litmus tests into perpetual litmus tests is summarized in Table 3.1.

3.4 Outcome Conversion and Analysis of Perpetual Litmus Tests

3.4.1 Exhaustive Outcome Counter

After removing per-iteration synchronization, test outcomes cannot be determined from the contents of the buf arrays in the same way as existing approaches do, since the relative timing of threads can now vary across iterations. Figure 3.3 presents an

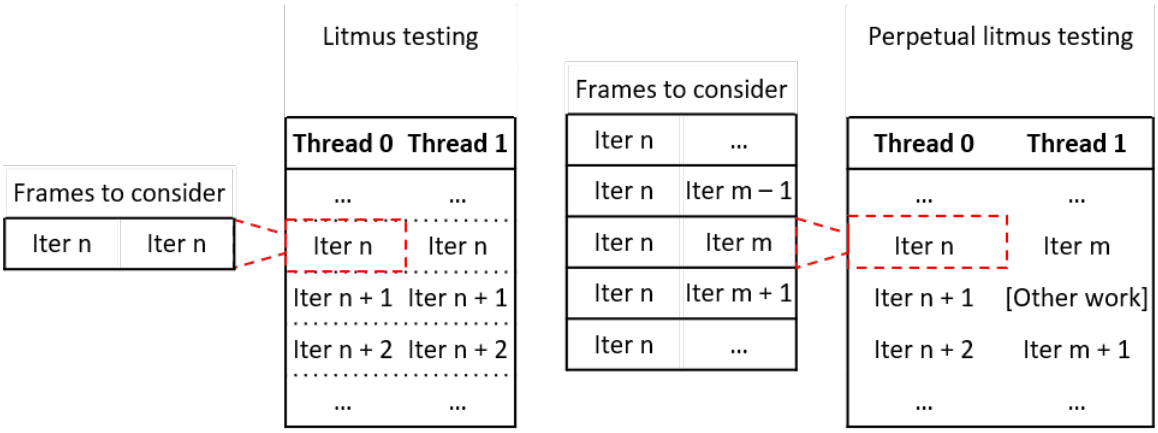


Figure 3.3: To determine the outcomes in a run of perpetual tests, all frames are examined, each made up of one iteration from each thread.

example using side by side comparison between original (left) and perpetual litmus tests (right) to determine which iterations of each thread will be considered for a test outcome. As Figure 3.3 shows, for iteration n of thread 0, it is no longer possible to simply consider its interaction with the same iteration n in other threads as was the case for original litmus tests (as seen on the left), since those iterations might have happened temporally far from each other. Instead, the interaction of each iteration of each thread with every iteration of every other thread must be examined, since they could happen concurrently (as seen on right). The term *frame* is defined to refer to a tuple of T_L iteration numbers, one per load-performing thread, where iteration indices need not be the same. Only load-performing threads are used, since their buf arrays hold all the information needed to determine the test outcome. Examining all frames therefore has time complexity N^{T_L} for a run of N iterations.

Detecting Outcomes Without Per-iteration Synchronization

To examine the way that threads interacted for a given frame, the outcomes of the original litmus test need to be expressed as *perpetual outcomes*, a format that takes the use of arithmetic sequences and the independent running of each thread into account.

Original Outcome	$reg_0\_0 = 0$ && $reg_1\_0 = 0$	$reg_0\_0 = 0$ && $reg_1\_0 = 1$
1) Happens-before edges	$i_{00} \rightarrow i_{01}$ (*po*), $i_{10} \rightarrow i_{11}$ (*po*), $i_{01} \rightarrow i_{10}$ (*fr*), $i_{11} \rightarrow i_{00}$ (*fr*)	$i_{00} \rightarrow i_{01}$ (*po*), $i_{10} \rightarrow i_{11}$ (*po*), $i_{01} \rightarrow i_{10}$ (*fr*), $i_{00} \rightarrow i_{11}$ (*rf*)
2) Replace registers	$buf_0[n] = 0$ && $buf_1[m] = 0$	$buf_0[n] = 0$ && $buf_1[m] = 1$
3) Replace integer values	$buf_0[n] = m$ && $buf_1[m] = n$	$buf_0[n] = m$ && $buf_1[m] = n + 1$
4) Turn to inequalities	***$p\_out_0(n, m, buf_0[], buf_1[])$ =*** $buf_0[n] <= m$ && $buf_1[m] <= n$	***$p\_out_1(n, m, buf_0[], buf_1[])$ =*** $buf_0[n] <= m$ && $buf_1[m] >= n + 1$
Original Outcome	$reg_0\_0 = 1$ && $reg_1\_0 = 0$	$reg_0\_0 = 1$ && $reg_1\_0 = 1$
1) Happens-before edges	$i_{00} \rightarrow i_{01}$ (*po*), $i_{10} \rightarrow i_{11}$ (*po*), $i_{10} \rightarrow i_{01}$ (*rf*), $i_{11} \rightarrow i_{00}$ (*fr*)	$i_{00} \rightarrow i_{01}$ (*po*), $i_{10} \rightarrow i_{11}$ (*po*), $i_{10} \rightarrow i_{01}$ (*rf*), $i_{00} \rightarrow i_{11}$ (*rf*)
2) Replace registers	$buf_0[n] = 1$ && $buf_1[m] = 0$	$buf_0[n] = 1$ && $buf_1[m] = 1$
3) Replace integer values	$buf_0[n] = m + 1$ && $buf_1[m] = n$	$buf_0[n] = m + 1$ && $buf_1[m] = n + 1$
4) Turn to inequalities	***$p\_out_2(n, m, buf_0[], buf_1[])$ =*** $buf_0[n] >= m + 1$ && $buf_1[m] <= n$	***$p\_out_3(n, m, buf_0[], buf_1[])$ =*** $buf_0[n] >= m + 1$ && $buf_1[m] >= n + 1$

Figure 3.4: Mappings of all the outcomes of the sb test to the corresponding perpetual outcomes, based on Figure 3.2. The conditions in red are perpetual litmus test outcomes that correspond to the original litmus test outcome and are generated by following the four stages presented in Section 3.4.

The Converter creates perpetual outcome templates using the following steps, which Figure 3.4 shows for each distinct outcome of the sb test:

1. Determine the happens-before edges [5] for the original litmus test outcome.

2. Replace all registers with accesses into the appropriate locations within the buf arrays, using a different iteration index for each thread in order to cover all frames.

3. Replace all integer values to account for the use of arithmetic sequences. Any integer value in the original outcome is loaded from shared memory, so it must have originated in some store operation (including the initializing store of 0). Thus, select a generic member of the arithmetic sequence now used by that store operation.

4. Different iterations of the same store instruction are connected with ws edges in iteration order. Since happens-before edges are transitive as temporal relations, the types of edges are the following:

    - fr edges: Some load L must have happened before some store S. But, L might also have happened before an even earlier store to the same location. So, L can load any term of the appropriate sequence *smaller than* that stored by S.

    - rf edges: Some load L must have happened after some store S. But, L might also have happened after an even later store to the same location. So, L can load any term of the appropriate sequence *larger than or equal to* that stored by S.

To create relational operations out of happens-before edges, litmus tests leverage the definitions happens-before edges for each happens-before edge type; for rf edges buf array holds the value of the store operation that participates in the happens-before

edge and for fr edges the buf array holds a value of a store operation that preceded the store operation participating in the happens-before edge. This becomes clear in the example of the sb test in Figure 3.4 as in step 2 the buf arrays in the equations hold 0 and 1 or n and $n+1$ or m and $m+1$ for the fr and rf edges respectively. For perpetual litmus tests, due to transitivity, equations become inequalities, as any rf or fr edge of a previous iteration can satisfy the inequalities due to transitivity, as explained in step 4 of the conversion process.

As Figure 3.4 shows, after all 4 steps have been performed on an outcome o of the original litmus test, the result is the corresponding perpetual outcome, which forms the body of a function p_out_o that the Converter defines. Provided with each load-performing test thread's iteration index and buf array, p_out_o returns true if and only if the conditions for the corresponding perpetual outcome are satisfied. The Converter repeats this process for each of the O outcomes of interest, creating the functions p_out_0 through p_out_{O-1}, each capable of detecting if the corresponding perpetual outcome occurred in a given frame.

Counting perpetual outcome occurrences

Now that the outcomes are in a format capable of being applied to any frame, the Converter generates the exhaustive outcome counter function following the general format of Algorithm 1. The Converter replaces each reference to a p_out function in this generic algorithm with a specific function generated through the process above for some outcome of interest. At a high level, $COUNT$ is given the number of iterations N and the buf arrays, which include the in-memory results of a test run. It goes through all the frames and, for each frame, evaluates each of the p_out functions. When one of the p_out functions evaluates to true, $COUNT$ increments the corresponding entry in the *counts* array, which has one entry for each perpetual outcome of interest. Note that up to one entry of *counts* is incremented for every frame.

The left part of Figure 3.5 shows the generation of the exhaustive outcome counter function for a target outcome of the sb test. After p_out_0 is defined for sb (top left of Figure 3.4), the Converter generates a version of the $COUNT$ function where the p_out_0 definition is substituted in the function of the exhaustive outcome counter.

Algorithm 1 Exhaustive Outcome Counter

```
 1: function COUNT(N, buf_0[], ..., buf_{T_L-1}[])
 2:
 3:     // Initialize array of occurrences of outcomes of interest
 4:     counts[O] = {0, ..., 0}
 5:
 6:     // Loop through all frames, n_i is the index of thread i
 7:     for (n_0 = 0; n_0 < N; n_0++) do
 8:         for ... do
 9:             for  (n_{T_L-1} = 0; n_{T_L-1} < N; n_{T_L-1}++) do
10:
11:                 // If an outcome of interest occurred, count it
12:                 if p_out_0(n_0, ..., n_{T_L-1}, buf_0, ..., buf_{T_L-1}) then
13:                     counts[0]++
14:                 else if ... then
15:                     ...
16:                 else if p_out_{O-1}(n_0, ..., n_{T_L-1}, buf_0, ..., buf_{T_L-1}) then
17:                     counts[O - 1]++
18:
19:     return counts
```

3.4.2 Heuristic Outcome Counter

The $COUNT$ function can detect all occurrences of each outcome of interest in a given test run. However, given that $COUNT$'s complexity is N^{T_L}, it can be slow when the number of iterations or test threads is large. To address this, a heuristic version of $COUNT$ called $COUNTH$ has been developed which has linear complexity. This function does not examine all frames unlike the COUNT function, so some opportunities to observe outcomes of interest are lost. Still, outcomes of interest are found experimentally to be detected more frequently than when using the traditional litmus testing approach, while the sharp decrease in runtime ultimately provides an attractive target outcome detection rate, as presented in Section 3.7.

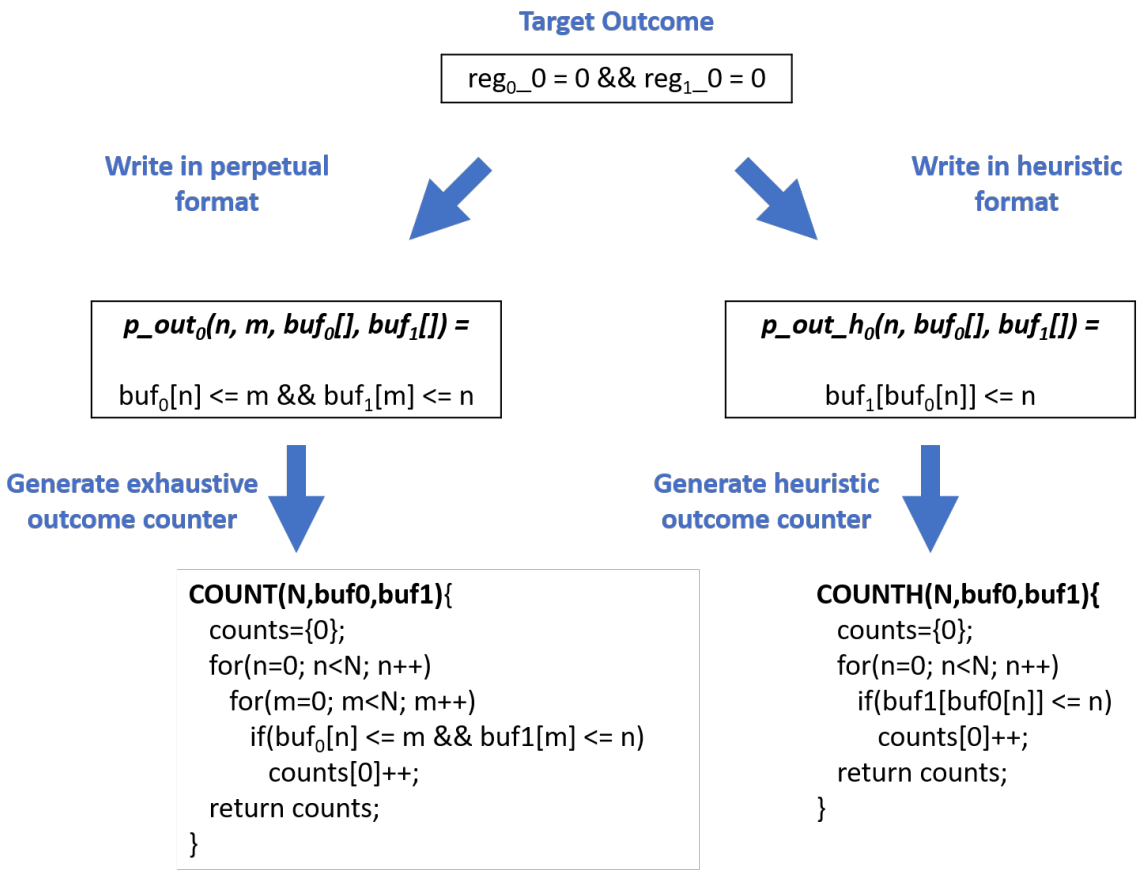


Figure 3.5: Example outcome conversion and generation of the exhaustive outcome counter and heuristic outcome counter for the target outcome of the sb test.

The intuition behind this heuristic is as follows. Because of the use of arithmetic sequences, any value loaded from shared memory indicates the iteration in which it was stored. For example, assume thread t loads the value val from the shared location $[x]$ in iteration n. This value must have been written to $[x]$ by a store in some iteration m of some thread s. The knowledge of the arithmetic sequences used by stores of thread s is used to determine m from val. Then, m is the most recent iteration of thread s, from the point of view of iteration n of thread t. Because of this proximity in time between iteration n of thread t and iteration m of thread s, examining the frame containing n and m can be more insightful than the frame containing e.g. $n + 100$ and $m - 100$, two iterations which happened farther away in time and are therefore less likely to have interleaved memory operations.

Original Outcome	$reg_0\_0 = 0$ && $reg_1\_0 = 0$	$reg_0\_0 = 0$ && $reg_1\_0 = 1$
1) to 3)	...	...
4) Turn to inequalities	$buf_0[n] = m$ && $buf_1[m] <= n$	$buf_0[n] = m$ && $buf_1[m] >= n + 1$
5) Substitute	***p_out_h$_0$(n, buf$_0$[], buf$_1$[]) =*** $buf_1[buf_0[n]] <= n$	***p_out_h$_1$(n, buf$_0$[], buf$_1$[]) =*** $buf_1[buf_0[n]] >= n + 1$
Original Outcome	$reg_0\_0 = 1$ && $reg_1\_0 = 0$	$reg_0\_0 = 1$ && $reg_1\_0 = 1$
1) to 3)	...	...
4) Turn to inequalities	$buf_0[n] = m + 1$ && $buf_1[m] <= n$	$buf_0[n] = m + 1$ && $buf_1[m] >= n + 1$
5) Substitute	***p_out_h$_2$(n, buf$_0$[], buf$_1$[]) =*** $buf_1[buf_0[n] - 1] <= n$	***p_out_h$_3$(n, buf$_0$[], buf$_1$[]) =*** $buf_1[buf_0[n] - 1] >= n + 1$

Figure 3.6: Heuristic condition generation for the perpetual outcome corresponding to each outcome of the sb test. By comparing to Figure 3.4, steps 1 to 3 are performed identically. step 4 is not performed for the conditions in red, which are then used for substitutions.

In order to generate the heuristic condition for each outcome, the Converter repeats steps 1-3 of the outcome mapping procedure from Section 3.4.1, but it then only performs step 4 for one of the outcome conditions in the compound conditionals. A new step is then added, step 5, in which the remaining conditions are used to substitute terms of the condition turned into an inequality. For example, in the top left of Figure 3.6, which considers the sb test outcomes, $buf_0[n] = m$ is used to replace m in the inequality $buf_1[m] <= n$ with $buf_0[n]$, yielding $buf_1[buf0[n]] <= n$. By repeating this process for each outcome of interest, the functions $p\_out\_h_0$ through $p\_out\_h_{O-1}$ are obtained. Each of these functions only depends on a single iteration index, since the rest is eliminated by the substitutions in step 5.

The Converter then generates the heuristic outcome counter function in a way similar to the exhaustive outcome counter function. Specifically, Algorithm 2 shows the general format of the heuristic outcome counter function. The Converter replaces each reference to a *p_out_h* function in this generic algorithm with a specific function generated through the modified 5-step process described above for some perpetual outcome of interest. *COUNTH* loops through the values of the remaining iteration index and, for each of them, evaluates each of the *p_out_h* functions. When one of the *p_out_h* functions evaluates to true, *COUNTH* increments the corresponding entry in *counts*, which has one entry for each perpetual outcome of interest.

Algorithm 2 Heuristic Outcome Counter

```
 1: function COUNTH(N, buf_0[], ..., buf_{T_L-1}[])
 2:
 3:     // Initialize empty array of perpetual outcome counts
 4:     counts[O] = {0, ..., 0}
 5:
 6:     for (n = 0; n < N; n++) do
 7:
 8:         // If an outcome of interest occured, count it
 9:         if p_out_h_0(n, buf_0[], ..., buf_{T_L-1}[]) then
10:             counts[0]++
11:         else if ... then
12:             ...
13:         else if p_out_h_{O-1}(n, buf_0[], ..., buf_{T_L-1}[]) then
14:             counts[O - 1]++
15:
16:     return counts
```

3.5 PerpLE Tool suite

The suite of tools developed is called the Perpetual Litmus Engine (PerpLE), and it is capable of converting a large family of litmus tests and their outcomes to their perpetual counterparts, as well as of executing them on x86 processors. Since test conversion only needs to happen once per litmus test and outcome conversion only

once for each set of outcomes of interest, PerpLE is organized into 2 distinct tools, as shown in Figure 3.1. The *Converter* deals with test and outcome conversion, while the *Harness* can use the outputs of the Converter to run tests and count the occurrences of the perpetual outcomes of interest.

3.5.1 Test Conversion

The Converter, implemented in Python, receives as inputs a litmus test and a set of outcomes of interest in the format used by the litmus7 suite [8]. It then generates the corresponding perpetual test by following the strategy outlined in Section 3.3, as well as the exhaustive and heuristic outcome counters for the outcomes of interest through the process described in Section 3.4. The Converter outputs the following:

- One assembly file per test thread, including that thread's instructions for the perpetual test enclosed in a loop construct, together with additional set-up and clean-up instructions as needed to handle arithmetic sequences.

- Two C files containing the exhaustive and heuristic outcome counters for this test, respectively. As described in Section 3.4, the exhaustive outcome counter file includes $COUNT$ after replacing the functions p_out_0 through p_out_{O-1} with their definitions for the outcomes of interest. The heuristic outcome counter file includes $COUNTH$ after replacing the functions $p\_out\_h_0$ through $p\_out\_h_{O-1}$ with their definitions for the outcomes of interest.

- An additional file with the parameters t_0_reads through t_{T-1}_reads, corresponding to the number of loads that each of the T test threads performs per iteration. This is required by the Harness to allocate appropriately sized buf arrays for each test thread, as discussed in Section 3.3.

For this work the Converter generates the perpetual tests in x86 assembly language, since that was the ISA of the system used for the evaluation. However, one could easily

adapt the process to different ISAs by providing the Converter with the instructions for loads, stores and fences in the corresponding assembly language.

3.5.2 Test Execution and Perpetual Outcome Counting

The Harness is a C program that runs N iterations of a perpetual litmus test and outputs the number of occurrences of each outcome of interest. In particular, the Harness allocates a buf array for each of the T_L load-performing test threads based on the values of t_0_reads through t_{T-1}_reads. It then launches the test threads and passes N and the appropriate buf array to each. After synchronizing once at the very beginning of the run, threads run synchronization-free for N iterations.

After the end of the run, the Harness calls the exhaustive and/or the heuristic outcome counter and provides them with the buf arrays. The *counts* arrays returned by the exhaustive and/or the heuristic outcome counter are then returned to the user, alongside runtime information both for test execution and for the outcome counting.

3.5.3 Perpetual Litmus Suite

To evaluate PerpLE, we used the PerpLE Converter to convert a comprehensive set of TSO litmus tests from the literature [109] into perpetual litmus tests, generating the *perpetual litmus suite*. Each of these tests involves between two and four threads. For each test, the PerpLE Converter generates the corresponding perpetual test, as well as exhaustive and heuristic outcome counters for the target outcome.

Not all litmus test outcomes can be converted to their perpetual equivalents. Perpetual litmus tests lack per-iteration synchronization and shared memory locations are therefore altered in an unpredictable pattern during test execution until the run of perpetual tests terminates. PerpLE can only stop to inspect the values in these shared locations *after* the end of the entire run; inspecting them earlier gives no information, since the current iteration of each thread is not known.

Perpetual Litmus suite

Target outcome allowed by x86-TSO		
amd3 [2,2]	iwp23b [2,2]	iwp24 [2,2]
n1 [3,2]	podwr000 [2,2]	podwr001 [3,3]
rfi009 [2,2]	rfi013 [2,2]	rfi015 [3,2]
rfi017 [2,2]	rwc-unfenced [3,2]	sb [2,2]
Target outcome forbidden by x86-TSO		
amd10 [2,2]	amd5 [2,2]	amd5+staleld [2,2]
co-iriw [4,2]	iriw [4,2]	lb [2,2]
mp [2,1]	mp+staleld [2,1]	mp+fences [2,1]
n4 [2,2]	n5 [2,2]	rwc-fenced [3,2]
safe006 [2,2]	safe007 [3,3]	safe012 [3,2]
safe018 [3,2]	safe022 [2,1]	safe024 [3,2]
safe027 [4,2]	safe028 [3,2]	safe036 [2,2]
wrc [3,2]		

Table 3.2: Perpetual litmus test suite for x86-TSO. The litmus tests are split into two groups based on whether their target outcome is allowed or forbidden by the x86-TSO specification. For each test, the values of $[T, T_L]$ are reported.

However, a class of litmus tests has target outcomes that require inspection of a value stored in shared memory at the end of every iteration. Due to the mechanics of PerpLE as described above, such outcomes cannot be converted into perpetual outcomes using the approach proposed here and therefore their occurrences cannot be counted using the exhaustive or the heuristic outcome counter. Tests with target outcomes of this nature are excluded from the perpetual litmus suite.

Table 3.2 presents the perpetual litmus test suite. The suite includes 34 litmus tests generated for the x86-TSO memory model out of the 88 tests found in the original test suite. The table splits the test suite into two groups of tests, based on whether their target outcomes are allowed or forbidden by the specification of the x86-TSO memory model.

3.6 Evaluation Methodology

3.6.1 Testing Environment & Tools

PerpLE is evaluated on a x86 computing cluster and the suite of tests presented in Table 3.2. Experiments are run on a CentOS 7.6 Linux cluster with 32 Intel Xeon E5-2667 CPUs with two threads per core. As explained in Section 2.1.2, the memory consistency model of CPUs in this cluster is a TSO variant [109], so only target outcomes from the "Allowed" group of tests in Table 3.2 are expected to be observed if the CPUs are correctly implemented.

PerpLE is evaluated against litmus7 on this system. For litmus7, all available thread synchronization modes are used for the evaluation [8]: the default `user`, with polling synchronization; `userfence`, which also uses memory fences to accelerate write propagation; `pthread`, which uses a pthread-based barrier; `timebase`, which relies on the architecture's timebase counter for synchronization; and `none`, where no thread synchronization is used [72]. Timebase counters are not available in some architectures (e.g., ARM).

The `none` mode is distinct from PerpLE's approach since the concept of *frames* is not utilized; iteration n of thread t_0 is only considered with respect to iteration n of thread t_1, even though they might be executed far in time from each other. This makes fine-grained thread interaction more elusive in `none` compared to PerpLE.

3.6.2 Metrics of Interest

Target Outcome Occurrences

Because perpetual outcomes are determined per frame and the number of frames is polynomial in the number of iterations (N^{T_L}), each particular perpetual outcome of interest in PerpLE is expected to be observed many more times than the corresponding outcome in litmus7, simply by virtue of exploring a much larger space, as shown

in Figure 3.3. Moreover, since PerpLE allows different types of thread interaction compared to litmus7, outcomes which appear only rarely in litmus7 may be observed more frequently. Since observing the target outcome of a test tends to be both rarer than observing other outcomes and more helpful in understanding the underlying hardware, the frequencies of how often the target outcome is observed in each system for a given number of test iterations is compared.

Testing Runtime

Since per-iteration thread synchronization dominates runtime on litmus7 in `user` mode, its removal should significantly reduce test runtime. However, the exhaustive outcome counter must examine all N^{T_L} frames in search of perpetual outcomes after a run of N iterations and T_L load-performing test threads, as opposed to the N frames examined by litmus7. This more extensive search will likely erode the speedup achieved by eliminating synchronization. In contrast, using the heuristic outcome counter should preserve a considerable speedup over litmus7, since it only examines N frames.

Target Outcome Detection Rate

This composite metric shows the number of times a target outcome is observed during a test run over the time taken by the run. Since the number of occurrences is expected to increase and runtime is projected to decrease when using the heuristic outcome counter compared to litmus7 synchronization modes, a much higher target outcome detection rate is expected for PerpLE.

Heuristic Outcome Counter Accuracy

To evaluate the heuristic outcome counter, the accuracy for the target outcome of each of the tests in the test suite is determined. For the target outcome of each test, the

exhaustive and the heuristic outcome counter are run on the in-memory test results from the same run of perpetual tests. A check is then performed whether, whenever the target outcome was found by the exhaustive outcome counter, it was also found by the heuristic outcome counter (not necessarily the same number of times).

Thread Skew

Due to the lack of per-iteration thread synchronization in PerpLE, threads can run ahead of each other by a varying number of iterations. The difference between the index of the iteration being executed by thread t and the index of the iteration being executed by thread s is defined as the *thread skew* between t and s. The width of the distribution of thread skew values indicates the degree to which the perpetual test run deviates from what is explored with per-iteration synchronization. This deviation can enable the perpetual test to observe system behavior that the original test misses.

To measure thread skew, the same insight that guided the development of heuristic conditions in Section 3.4.2 is used. Namely, the value loaded by thread t through a load operation in iteration n of the perpetual test run uniquely identifies a store in some iteration m of some thread s. The difference between n and m is exactly the skew between threads t and s around the time of iteration n in thread t.

Outcome Variety

One goal of perpetual litmus tests is to increase the effectiveness of memory consistency testing by enabling more thread interactions and generating a greater variety of test outcomes. To evaluate PerpLE with respect to this goal, a comparison is performed on how frequently each possible test outcome occurs in PerpLE and in litmus7 for the same number of iterations.

3.7 Evaluation

This section evaluates PerpLE and litmus7 using the metrics described in Section 3.6. In particular, PerpLE (i) detects more occurrences of target outcomes, (ii) is generally superior to litmus7 when using the heuristic outcome counter, in terms of both test runtime and target outcome detection rate and (iii) provides increased outcome variety.

3.7.1 Target Outcome Occurrences

Figure 3.7 compares the abilities of PerpLE and of litmus7 in different synchronization modes to detect the target outcome of tests in the perpetual litmus suite, across 10k iterations. PerpLE with exhaustive counter performs strictly better than litmus7 in all cases, observing many occurrences of each target outcome. PerpLE with heuristic counter generally performs better than litmus7 in most cases. For the iwp24 and rfi013 tests, litmus7 in `timebase` and synchronization modes marginally outperforms PerpLE heuristic. When the number of iterations increases beyond 10k, PerpLE is able to markedly outperform litmus7 in all synchronization modes, even for these cases.

As shown in Table 3.2, many of the tests in the perpetual test suite have target outcomes that are forbidden under x86-TSO, as determined using the herd memory model simulator [9]; this means that neither tool is expected to observe them on widely-used and heavily-verified commercial CPUs. These forbidden target outcomes are annotated in Figure 3.7 with an "X". Therefore, PerpLE's failure to observe these forbidden outcomes can be viewed as a reassurance that PerpLE does not generate false positives. In addition, PerpLE exposes target outcomes from *all* litmus tests that are allowed under x86-TSO, whereas litmus7 fails to do so for most tests (see

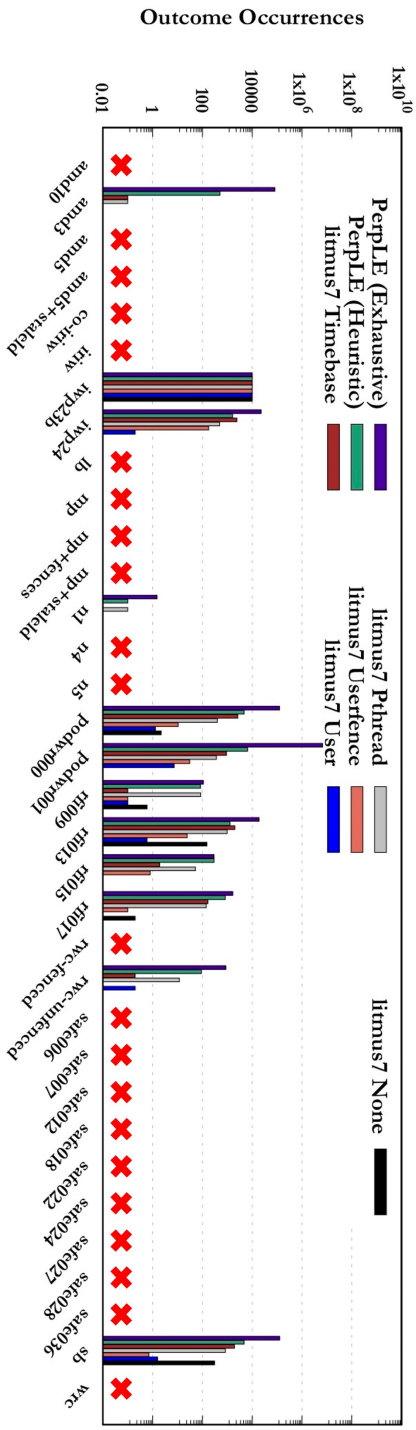


Figure 3.7: Target outcome occurrences for each test of the perpetual litmus suite for 10k iterations. Higher is better. PerpLE with either outcome counter generally outperforms litmus7 in most synchronization modes. Note that PerpLE does not generate "false positives" for any test, and Unobservable target outcomes appear as 0 here. Also note that PerpLE exposes the target outcomes for all tests that are allowed under x86-TSO. A red X symbol marks each test with a target outcome that is forbidden under x86-TSO.

amd3, iwp24, n1, podwr001, rfi015, rfi017, rwc-unfenced) for certain synchronization methods.

3.7.2 Testing Runtime

Figure 3.8 presents the runtime speedup over litmus7 in `user` mode of PerpLE when using the exhaustive and heuristic outcome counters, as well as over litmus7 in the other synchronization modes. All tools execute every test in the perpetual litmus suite for 10k iterations. All runtimes include both test execution and outcome counting.

Figure 3.8 focuses on the comparison between the PerpLE exhaustive and heuristic Counters. As discussed previously, the runtime of the PerpLE exhaustive outcome counter is polynomial to the number of test threads that perform loads, due to the examination of each frame. For each additional load-performing thread, there is an increase in computational complexity by a factor of N, e.g. checking N^2 frames for two threads increases to N^3 frames for 3 threads. Most tests in the test suite have two threads that perform loads, so examining a quadratic number of frames makes PerpLE with the exhaustive outcome counter significantly slower than the heuristic counter in these cases. The perpetual litmus tests that present the exhaustive outcome counter's performance comparable to the heuristic counter, e.g. mp, only have a single test thread performing loads, making the exhaustive outcome counter linear. Finally, the podwr001 and safe007 tests have three threads performing loads, so the exhaustive outcome counter needs to examine N^3 frames for a run of N iterations, yielding a dramatic slowdown. As a result the geometric average speedup of the heuristic outcome counter over the exhaustive outcome counter is 305x.

Therefore, if only focusing on runtime, using the PerpLE exhaustive outcome counter scales poorly as the number of iterations increases. The heuristic outcome counter scales much better, always taking time linear in the number of test iterations. As such, PerpLE exhaustive outcome counter performance constraints make it imprac-

tical and the remaining evaluation will therefore focus on the heuristic outcome counter. In subsequent text, the term PerpLE refers to PerpLE-heuristic. As Figure 3.8 shows, when using the heuristic outcome counter, PerpLE provides a (geometric) average speedup of 8.89x over litmus7 in the default `user` mode and 17.56x, 8.85x, 2.52x and 161.35x over the `timebase`, `userfence`, `none` and `pthread` modes respectively. Note that the runtime of PerpLE with the heuristic outcome counter is comparable to `none`, since both use no synchronization and examine a linear number of frames. However, PerpLE offers significantly better outcome variety, as presented in Figure 3.11.

3.7.3 Target Outcome Detection Rate

To determine PerpLE's overall efficiency, the target outcome detection rate between PerpLE using the heuristic outcome counter and litmus7 in different synchronization modes are compared. For the comparison between methods different options for averaging target outcome detection rates were evaluated. Averaging outcome detection rates across different tests would implicitly skew the average towards the tests that intrinsically observe higher numbers of target outcomes. Therefore, PerpLE's relative detection rate improvement is defined by dividing PerpLE's detection rate for a given litmus test by the detection rate of litmus7 in default `user` mode for the same test; by using these ratios when averaging across all litmus tests, the aforementioned skewing problem is avoided. When reasoning about ratios of detection rates, test cases where the baseline testing method is zero (i.e., no outcomes were detected) are conservatively omitted and additional details about the number of outcomes PerpLE detects in these cases are provided.

Figure 3.9 presents the average relative target outcome detection rate improvement. Experiments were run for 100 iterations (not shown) to 100M iterations. Each bar corresponds to the arithmetic mean of the relative improvement across all tests of the perpetual litmus suite that have target outcomes allowed in x86. The PerpLE

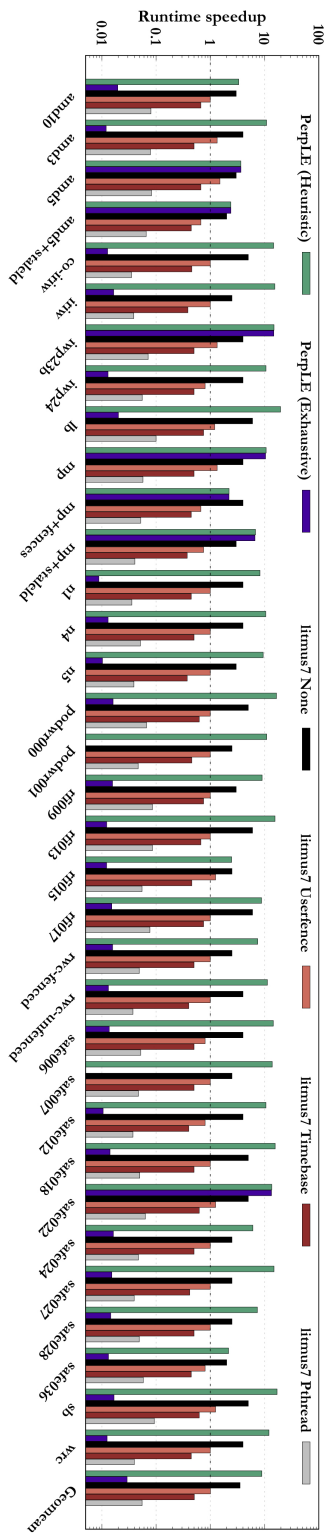


Figure 3.8: Relative speedups compared to litmus7 in `user` mode (=1). Comparison between PerpLE Exhaustive, PerpLE Heuristic, and several litmus7 modes, for each test in the perpetual litmus suite. All runtimes include both test execution and outcome counting. Higher is better. PerpLE heuristic is always fastest.

detection rate is nonzero for all allowed target outcomes for all test iteration numbers except the *n1* litmus test for 100 iterations. The litmus7 `user` mode is zero (i.e. does not detect any outcomes) for all litmus tests for 100 iterations and becomes nonzero for all tests only after 1M iterations. The remaining litmus7 synchronization modes either require a similarly high number of iterations to become nonzero for all tests or never achieve that. PerpLE demonstrates the ability to discover target outcomes at low iteration counts, which litmus7 generally fails to do. More specifically, for 100 iterations (not shown), PerpLE heuristic counter and litmus7 `pthread` mode are the only tools with a non-zero detection rate.

Additionally, PerpLE is able to provide a target outcome detection rate that is strictly higher than any of litmus7's synchronization modes. For 10k iterations, PerpLE's average relative outcome detection rate improvement is between 24x (over `timebase`) and 31000x (over `user`). PerpLE is able to scale gracefully, maintaining a high relative target outcome detection rate improvement for high iteration counts: between 1800x-140000x for 10M and 1200x-44000x for 100M iterations. Overall, the target outcome detection rate of PerpLE is at least four orders of magnitude higher than that of litmus7 in the `user` (default) synchronization mode for all iteration counts.

3.7.4 Heuristic Accuracy

Figure 3.7 also indirectly showcases the perfect accuracy of PerpLE's heuristic outcome counter, since it tracks the exhaustive outcome counter in terms of whether the target outcome was found or not.

3.7.5 Thread Skew

Figure 3.10 presents a probability density function of the skew between the two threads participating in the perpetual sb litmus test, as defined in Section 3.6. Thread skew

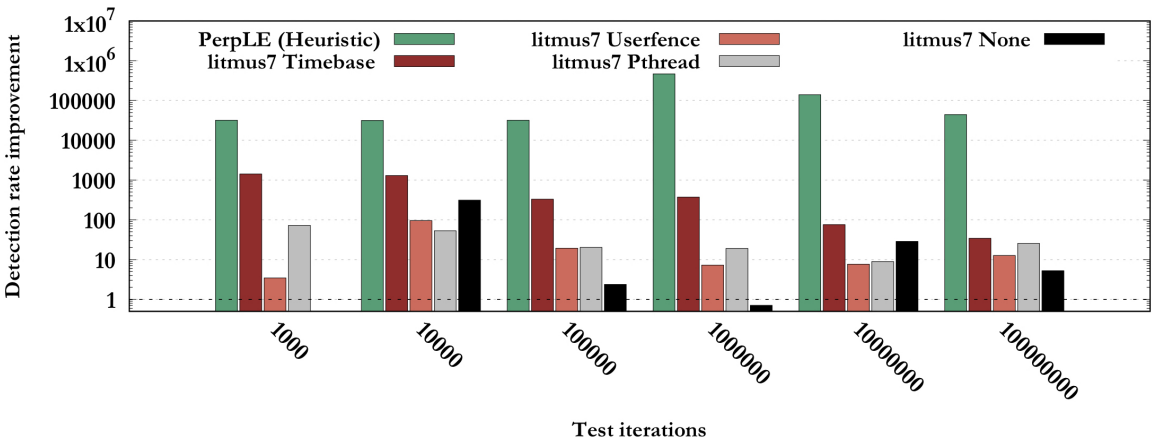


Figure 3.9: Relative target outcome detection rate improvement of PerpLE using the heuristic outcome counter and litmus7 in different synchronization modes, for different numbers of test iterations. litmus7 `user` mode is the baseline. Bars represent the arithmetic mean of the relative target outcome detection rate improvement across all perpetual litmus tests with target outcomes allowed by x86-TSO. Higher is better. PerpLE outperforms litmus7 in all modes by one to five orders of magnitude. Unlike PerpLE, litmus7 in most modes does not observe the target outcome for small iteration counts (baseline litmus7 `user` is zero for fewer than 1000 iterations).

is a result of numerous system factors, like operating system scheduling and small differences in the time when each thread starts executing. The distribution is very wide, indicating threads can run far behind or ahead of each other. Still, it is denser around 0, since system factors might delay either test thread during execution and these effects then cancel out.

A wide range of skew values contributes to the success of perpetual litmus tests, since it is indicative of the potential for interesting cross-iteration interleavings. In contrast, traditional litmus tests are limited by synchronization and the only interleavings possible across different threads are between operations of the same iteration (no skew).

3.7.6 Outcome Variety

Figure 3.11 plots the number of occurrences of each outcome for the sb, lb and powdr001 litmus tests over 1k iterations. The code for the three tests are shown

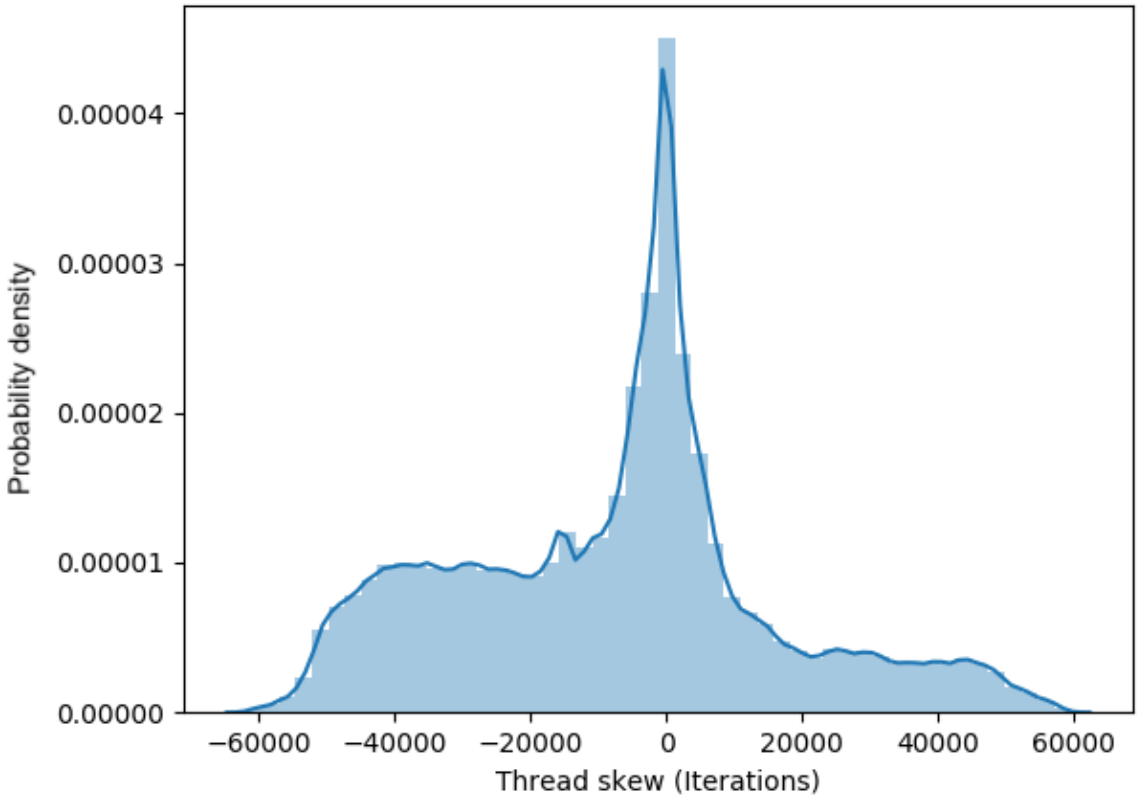


Figure 3.10: Probability density of the thread execution skew (in iterations) between the two threads for 100k iterations of the store buffering (sb) perpetual litmus test. Store buffering (sb) is used as an illustrative example; other tests exhibit similar thread skew results based on the experimental evaluation.

in Figure 2.2. Litmus7 across different synchronization modes and PerpLE using the heuristic outcome counter are evaluated in terms of (i) their ability to observe a large variety of possible outcomes and (ii) the high number of observations generated for each individual outcome. All outcomes presented are observable under x86-TSO except for $reg_0\_0 = 1$ && $reg_1\_0 = 1$ in the lb litmus test (lb outcome 11 on Figure 3.11), which is forbidden.

litmus7 only observes outcomes sb 11 in the `timebase` mode and powdr001 111 in the `timebase` and `userfence` modes for 1k iterations. When running 1M iterations instead, these two outcomes are observed in other litmus7 synchronization modes as well, which shows that PerpLE heuristic is capable of observing outcomes of interest using a much smaller number of test iterations. Moreover, compared to litmus7, the

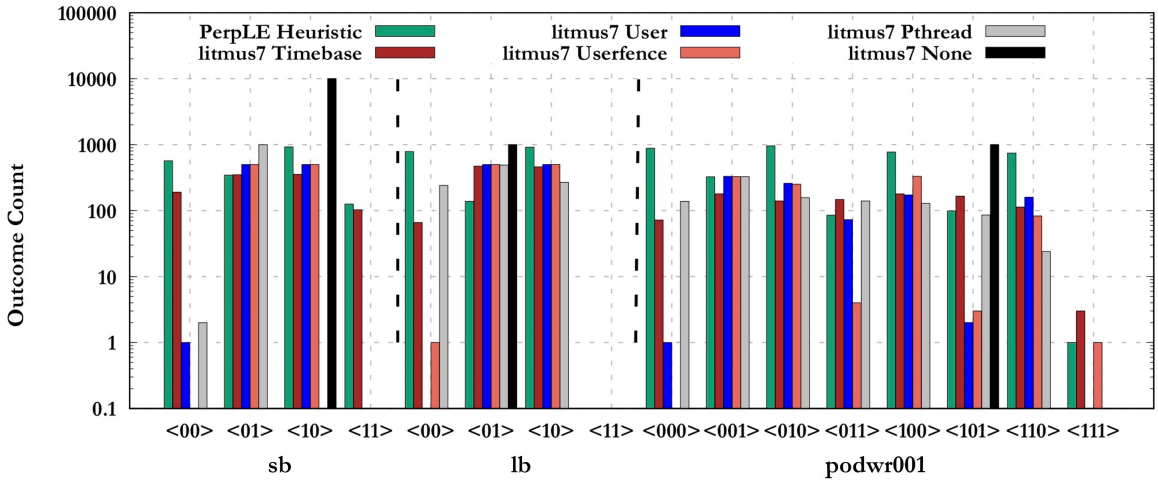


Figure 3.11: PerpLE using the heuristic outcome counter and litmus7 in different synchronization modes compared in terms of outcome variety for the sb, lb and podwr001 litmus tests, for 1k iterations. PerpLE heuristic samples 1k frames *per outcome*. Higher outcome variety is better. The 11 outcome for lb is forbidden in x86-TSO.

number of the occurrences of each outcome observed when using PerpLE heuristic is typically higher than litmus7 synchronization methods.

As specified in Table 3.2, $T_L = 2$ for sb and lb, while $T_L = 3$ for podwr001. Therefore, for podwr001 N^3 frames are examined, compared to N^2 for the other two tests presented, which explains why PerpLE is able to determine a higher increased total number of perpetual outcomes for podwr001 compared to the sb and lb test outcomes. The heuristic for each outcome evaluates N samples out of the N^2 or N^3 available frames. It is important to note that for litmus7 the total number of occurrences for each test equals the number of test iterations, spread across the observable outcomes.

With the exception of `timebase` mode, where the two tools' results are comparable, PerpLE provides a better outcome variety than litmus7 for the same number of iterations with higher numbers of outcome occurrences.

3.7.7 Overall Impact on Testing

PerpLE provides runtime acceleration and relative detection rate improvement benefits for the overall consistency testing process. PerpLE can accelerate some important tests and notify the user about non-convertible tests, which can still be run using litmus7. When running each of the 88 tests for 10k iterations, it is 1.47x faster to use PerpLE instead of litmus7 for the 34 convertible tests (and litmus7 in `user` mode for the rest). For tests with target outcomes allowed in x86-TSO, PerpLE improves average relative target outcome detection rate by over 20000x compared to litmus7 for the convertible tests for 10k iterations.

3.8 Related Work

As discussed in Chapter 2, memory models range widely, from sequential consistency (SC) [86] to weak memory consistency models in modern architectures [3, 31, 34, 53, 118]. Several previous works have focused on formalizing the memory consistency models of different architectures, e.g., x86 [109], POWER or more generic models [7, 9, 30], while others have focused on language level formalization of memory models [78, 83, 123]. Frequently, such efforts resort to running litmus tests on hardware and comparing their empirical observations to the published memory model specifications [2, 101]. Other efforts use random testing instead, which can also be very effective for detecting deviations from the published model, at the cost of having a consistent test suite [44, 64, 119]. Using perpetual litmus tests can accelerate such efforts on newly developed architectures while also exposing a greater variety of outcomes, giving a fuller picture of the capabilities of the underlying hardware.

Other work has focused on specifying and verifying microarchitectural implementations of consistency models using happens-before graphs [94, 98, 99, 100, 121]. These

tools focus on design time verification whereas PerpLE performs runtime evaluation of litmus tests against the hardware specification.

Based on formal memory consistency models, other efforts have addressed litmus test *generation*, for the purposes of comparing memory models or for the empirical conformance testing of new implementations of an architecture in hardware [44, 64, 96, 129]. The Converter tool in PerpLE extends such tools by converting newly generated litmus tests to their perpetual counterpart, providing automatic access to the benefits of running tests without per-iteration synchronization.

Another existing line of research has been concerned with developing tools for running non-deterministic tests, including litmus tests [1, 6, 43]. Central among these is the diy suite of tools, which includes the litmus7 tool used in this chapter's experimental evaluation [72]. PerpLE adds to such tool development efforts, by providing a critical twist (removal of per-iteration synchronization) on a familiar approach (litmus tests). A key difference between PerpLE and litmus7 pertains to frames. Namely, PerpLE not only allows longer-term cross-iteration interleaving of events, which enriches the event orderings considered, but it also implements the logging needed to properly see these interactions from the results. Litmus7's different synchronization modes may allow for some of the same orderings, but that tool does not have the logging to see cross-iteration interleavings. Furthermore, litmus7 cannot automatically generate perpetual litmus tests from original tests, and its synchronization modes cannot enable analysis methodologies that use frames similar to PerpLE. In comparison, PerpLE includes automatic test generation from original tests.

Finally, past work has been concerned with developing techniques to increase the effectiveness of litmus tests by creating different system environments for the test threads, in the hopes of exposing otherwise rare outcomes. Such approaches can have a dramatic impact: for example, Sorensen et. al [115] shows that the use of stressing and fuzzing can increase the occurrence rate of the target outcome in the

lb, sb and mp litmus tests in GPUs. PerpLE also creates unusual (compared to traditional approaches) conditions for the test threads by enabling longer stretches of synchronization-free execution by each thread. The thread skew generated this way can be valuable in exercising the system, as indicated by the results.

3.9 Conclusion

Given parallelism's centrality in computing today, memory consistency testing is critical to ensure that systems and applications adhere to their formal specifications. However, current empirical litmus testing approaches waste most of the testing time waiting for threads to synchronize, a requirement that also can hurt the variety and types of outcomes of the tests. In response, this thesis proposes perpetual litmus tests, a litmus test variant that allows for consistency testing without per-iteration synchronization, by tracing happens-before edges between load and store operations using unique arithmetic sequences. This chapter introduces PerpLE, a set of tools to generate, execute and analyze perpetual litmus tests and their outcomes.

PerpLE is evaluated on an x86 system, showing both greater outcome variety and more occurrences of the outcomes of interest. PerpLE can use a polynomial algorithm or an efficient, linear heuristic to identify outcomes of interest. The highly accurate heuristic provides a significant speedup, leading to an overall target outcome detection rate that is orders of magnitude higher than prior state of the art. This dissertation focused on x86-TSO, but the covered methodologies can also be applied to architectures implementing weaker memory models. These improvements can help expand the applicability and effectiveness of empirical memory consistency testing.

The work in this chapter reinforces the value of litmus tests as a testing tool for Memory Consistency Model testing and presents novel testing and analysis methodologies and tools building on litmus tests, indicating the promise for impactful future

research in this domain. We also identified that there is promise in outcome variety in measuring how a tool exercises possible observable interleavings. Chapters 4 and 5 take some of these lessons learned from empirical MCM testing based on litmus tests and apply them to distributed systems and smart home systems.

Chapter 4

Testing Consistency Using Distributed Litmus Tests

At its core, this dissertation builds fast, efficient empirical testing methodologies for consistency properties across different domains of systems. The focus of Chapter 3 was in developing empirical testing methodologies for CPUs using litmus tests that can keep up with rising hardware complexity while addressing challenges in efficiency, scalability and outcome variety.

This chapter takes lessons learned from the use of litmus tests in microprocessors and applies them to highly concurrent, distributed systems. Distributed systems present new challenges for testing. As systems coordinate over the network, processing and storage are decoupled due to replication, and systems can experience network partitions and failures. My work explores how to adapt litmus tests to distributed key-value stores and presents a tool to enable testing using the novel litmus test variant, called Distributed Litmus Tests, in practice.

4.1 Introduction

The emergence of the cloud has been a catalyst for a wide range of applications and services and has given rise to high-performance data center infrastructure and systems. Developers now utilize cloud infrastructure to create complex software systems and applications quickly and at scale. However, ensuring correctness becomes increasingly complex in distributed systems, as software can run on thousands of instances of machines that coordinate with each other over a network and that are prone to hardware faults. In many cases, correctness is a critical user requirement, for example, when the software controls access to users' smart homes or is responsible for bank transactions.

All software systems have bugs that can affect the software's correctness, performance and energy efficiency and that can compromise the software's security. Testing software for correctness is not an easy task; there are numerous testing tools available to aid in the search of bugs, but testing is a process that might not be fully automated or provide coverage for a desired correctness guarantee. Network and hardware faults introduce non-determinism which can cause bugs that are difficult to replicate as they can be a result of intricate timing and ordering of updates or a result of the presence of faults.

This chapter introduces Distributed Litmus Tests (DLTs), a new way to test correctness of systems software using small, property-based tests that capture the non-determinism present in distributed systems. Testing using litmus tests is a testing methodology that has been adopted by the computer architecture community for testing of consistency properties in multiprocessors and other types of heterogeneous and parallel hardware, as discussed previously in Chapters 2 and 3. To perform testing using DLTs, this chapter examines deployment and execution parameters that can affect testing on distributed systems, the intricacies that distributed systems and the network present in testing and the required adaptations of the litmus test methodology

required to efficiently test correctness using targeted tests in distributed systems. Testing using DLTs is possible in practice using Musli Tool, a tool this dissertation work developed that enables generation, automatic testing and analysis of outcomes of DLTs on distributed systems.

This dissertation presents a new set of tools that provide the ability to use property-based tests to stress test and reason about the correctness of specific properties in distributed systems. Here are this chapter's contributions:

- This chapter leverages techniques inspired from the computer architecture community to develop a rigorous methodology called Distributed Litmus Tests that provide focused, property-based testing.

- Distributed Litmus Tests are small, parallel tests that capture specific consistency scenarios and function as non-deterministic distributed unit tests. These idiomatic tests are effective at exercising specific correctness properties and provide high performance for testing. Unlike random testing techniques that are widely used in distributed systems, Distributed Litmus Tests are designed to target specific consistency properties to empirically test if a system implementation behaves according to properties under test. This chapter demonstrates how litmus tests are applicable in distributed key-value stores.

- This chapter presents the litmus suite converter, a tool that converts conventional CPU litmus suites into Distributed Litmus Tests. Enabling conversion of conventional litmus tests allows the leveraging of existing architectural litmus tests and litmus test suites to test distributed systems for correctness.

- This chapter presents Musli Tool, an empirical testing tool that deploys Distributed Litmus Tests on distributed key-value stores and manages the execution and analysis of test results. Musli Tool allows us to study a range of execution

parameters and the characteristics of Distributed Litmus Tests across a range of system configurations.

- This chapter demonstrates the ability to observe weak outcomes in widely used open source distributed key-value stores by using a combination of Musli Tool and the litmus test suite.

- Finally, we use Musli Tool to show how litmus tests can be used to identify key properties that define a consistency model.

4.2 Adaptation of Litmus Tests to Distributed Systems

In order to adapt litmus tests from CPUs to distributed systems, our methodology needs to be adapted to test effectively despite the concurrency and complexity in these systems architectures, including mechanisms for replication, ordering over the network, reaching consensus, as well as different cluster topologies and cluster sizes. In this section, we discuss 3 adaptations introduced when creating DLTs.

DLTs must recognize that computation and data storage may happen at different physical locations and that data may be replicated at multiple physical locations. This means a client can execute on one node but store results at a different one. For each client in a litmus test, the updates to data will execute on a single node. However, updates are not *local* operations; each update can be directed to any replica that holds a copy of the object an update operates on. Computer servers, called nodes, and replicas can be co-located. Nodes can keep replicas of data objects, but it is not necessary that every node maintains a copy of a data object. As a result, for every litmus test, there exist multiple possible Distributed Litmus Tests variants that stem from the same litmus test specification but vary depending on the system parameters.

The resulting variants are called equivalent litmus tests. Depending on the cluster topology and the replica each update is performed on, there are different possible interleavings that can expose a bug or consistency violation. Equivalent litmus tests can cause a variation in which outcomes can be observed and the frequency with which each outcome happens. As updates are directed to different replicas, the difference in timing (i.e. propagation delay over the network) can be used to create a suite of tests that provide improved stress testing properties as compared to a single version of a litmus test. For example, a Cassandra key-value store routes updates to the replica that is closest to the client by proximity; changes in placement of litmus test clients could then affect if and how frequently an outcome can be observed. Therefore, in order to stress test the system, this work enumerates a set of possible litmus test variants and executes them on the cluster to create a robust testing framework.

Distributed updates are operations that are initiated by a client running on a node and that are received over the network by a replica possibly running on a different node. Program order is maintained by the clients executing the litmus test and initiating updates. However, the client behavior can affect litmus test execution depending on if the client allows execution to continue when initiating a distributed update or blocks waiting for the update to complete. Updates that block client execution waiting for a response from the replica before resuming execution are called synchronous updates. Asynchronous updates initiate an update and continue execution; the client is updated with the response of the update when the update has completed using a callback function. Client behavior can affect the execution of a Distributed Litmus Test and which outcomes are observable, as asynchronous clients allow more interleavings between updates to appear whereas synchronous clients limit them.

To run a Distributed Litmus Test, distributed updates are initiated over the network which incur delays inherently due to the network latency present in reaching a replica from a client on a different node. The presence of delays enables updates

running concurrently to observe more outcomes as a result of update interleavings. To enhance the effect of network delays, Distributed Litmus Tests can insert additional delays between updates.

4.3 Testing Consistency using Distributed Litmus Tests

This section describes how property based testing techniques from the computer architecture literature (i.e. litmus tests) can similarly be used to enable consistency testing in distributed systems.

4.3.1 Comparing DLTs to Litmus Tests

Throughout this chapter, Distributed Litmus Tests (DLTs) will be used on distributed key-value stores. As demonstrated in Chapter 2, distributed key-value stores are storage systems that enable updates (reads, writes, updates and deletes) to replicated data objects. In Figure 4.1 DLTs are explained with an example. Consider a distributed key value store with four nodes and three replicas of object x. For the test, a client on each node runs a set of updates. In this test, four clients execute concurrently, two of these clients perform writes of different values to object x and another two clients perform two consecutive reads on the same object, with each value stored in the variables r1 through r4. Based on the interleavings of these reads and writes, variables r1 through r4 can store different values, depending on the order that updates actually execute. The values observed by the read operations and stored in the variables are collectively called a *litmus test outcome*. In this example, the outcome depends on the values of reads r1, r2, r3 and r4. Initially, all shared objects (i.e., x) are set to zero. Combinatorially, there are 81 possible combinations that variables r1 through r4 can take. However, the consistency model that the underlying system implements

Node 1	Node 2	Node 3	Node 4
Client 1	Client 2	Client 3	Client 4
(i1) x.write(1)	(i2) x.write(2)	(i3) r1=read(x) (i4) r2=read(x)	(i5) r3=read(x) (i6) r4=read(x)

Distributed Key-Value Store Cluster			
Replica 1	Replica 2		Replica 3

Figure 4.1: Example of a litmus test on a distributed key-value store. Read and write updates on a replicated object x are executed by clients running on a key-value store cluster. The observation of an outcome is used to determine correct or incorrect implementation of consistency model guarantees of the system under test.

dictates what interleavings are observable by the system. In litmus tests, a test is performed under a reference consistency model, so that observable outcomes of a test can be deemed allowed or forbidden. An example outcome under test for this example is *(1,2,2,1)*. For this outcome to appear using this test, Clients 1 and Client 2 must write values 1 and 2 concurrently. For outcome *(1,2,2,1)*, the Client 3 reads r1=1 and r2=2 and Client 4 reads r3=2 and r4=1. In order for this outcome to be observable, Client 3 and Client 4 need to observe the writes from Client 1 and Client 2 in a different order. Consider what happens when using linearizability as a reference consistency model; there is a requirement for a total order of updates for a single object. Consequently, the outcome *(1,2,2,1)* is impossible unless i2 occurs before i1 and i1 occurs before i2 (cycle). This is not possible because it would imply a timing cycle where operations happen before themselves. Thus, the outcome is forbidden and observation of the outcome indicates a bug in the implementation if it is observed on a system that implements linearizability. Since client execution is decoupled from the storage operations executed by the distributed key-value stores, there are many possible ways to place the clients on nodes in order for the clients to interface with the replicas differently. Execution of clients can also happen with both synchronous and asynchronous execution and delays can be added between consecutive updates.

Causality

Client 0	Client 1	Client 2	Client 3
(i1) x.write(1)	(i2) r1=read(x) (i3) x.write(2)	(i4) r2=read(x) (i5) r3=read(x)	(i5) r4=read(x) (i6) r5=read(x)

Causality 2

Client 0	Client 1	Client 2	Client 3
(i1) x.write(1)	(i2) x.write(2) (i3) x.write(3)	(i4) r1=read(x) (i5) r2=read(x)	(i5) r3=read(x) (i6) r3=read(x)

Figure 4.2: Causality and Causality 2 DLTs.

4.4 Litmus Test Suite

To perform property based testing across a range of systems and consistency models, we need to compile a suite of litmus tests, which are non-deterministic, distributed unit tests. The test suite is designed to test a range of distributed consistency guarantees, as distributed key-value stores can implement and be configured for a range of consistency guarantees, ranging from weak to strong consistency. The litmus suite is inspired by litmus tests for memory consistency models. There exists a large number of CPU litmus tests available for different memory consistency models (e.g., x86 TSO, arm, powerpc, etc.) each of which specify an outcome that is put under test. The test suite contains distributed tests, which are converted by our tools from their CPU counterparts.

4.4.1 Distributed Litmus Test Examples

Figures 4.2-4.5 present the set of DLTs used in the evaluation of Musli Tool and used for the consistency measurement study performed on the tool. Since the DLTs are used as part of a consistency testing study, we evaluate outcome variety on Musli Tool using all DLT outcomes and not a single DLT target outcome. Figure 4.2 presents two litmus tests focusing on consistency properties related to causality. More specifically, happens-before relationships between i1, i2 and i2, i3 are present due to program order

Concurrent Writes

Client 0	Client 1	Client 2	Client 3
(i1) x.write(1)	(i2) x.write(2)	(i3) r1=read(x) (i4) r2=read(x)	(i5) r3=read(x) (i6) r4=read(x)

Stale Reads

Client 0	Client 1	Client 2
(i1) x.write(1)	(i2) r1=read(x) (i3) y.write(1)	(i4) r2=read(y) (i5) r3=read(x)

Figure 4.3: Concurrent Writes and Stale Reads DLTs.

Store Buffering

Client 0	Client 1
(i1) x.write(1) (i2) r1=read(y)	(i3) y.write(1) (i4) r2=read(x)

Load Buffering

Client 0	Client 1
(i1) r1=read(x) (i2) y.write(1)	(i3) r2=read(y) (i4) x.write(1)

Message Passing

Client 0	Client 1
(i1) x.write(1) (i2) y.write(1)	(i3) r1=read(y) (i4) r2=read(x)

CO-MP

Client 0	Client 1
(i1) x.write(1) (i2) x.write(2)	(i3) r1=read(x) (i4) r2=read(x)

Figure 4.4: Store Buffering, Load Buffering, Message Passing and CO-MP DLTs.

and reads in the first causality test and similarly happens before relationships between writes can appear in the second test. The values read in Clients 2 and 3 indicate if the system producing a resulting outcome maintains causal relationships.

In Figure 4.3, the Concurrent Writes test examines if the write updates executing concurrently on Clients 0 and 1 become observable in the reverse order on Clients 3 and 4, indicating that writes are not replicated consistently across replicas, a weak outcome. The Stale Reads test potentially identifies behaviors where an older value of the shared object x is read in i5 after the newer value is read in i2.

Figure 4.4 presents the Store Buffering, Load Buffering, Message Passing and CO-MP litmus tests, commonly found in architectural consistency model test suites for identifying instruction reorderings. Store Buffering tests if write to read reorderings are observable on the system under test, and the Load Buffering test checks if read to write reorderings are observable on the system. In the Message Passing test, object y

Monotonic Reads

Client 0
x.write(1) x.write(2) r1=read(x) r2=read(x)

Monotonic Reads 2

Client 0	Client 1
(i1) x.write(1) (i2) r1=read(x)	(i3) x.write(1) (i4) r2=read(x)

Read your own writes

Client 0
x.write(1) x.write(2) r1=read(x)

Figure 4.5: DLTs exercising client-centric consistency properties, Monotonic Reads, Monotonic Reads 2 and Read Your Own Writes tests.

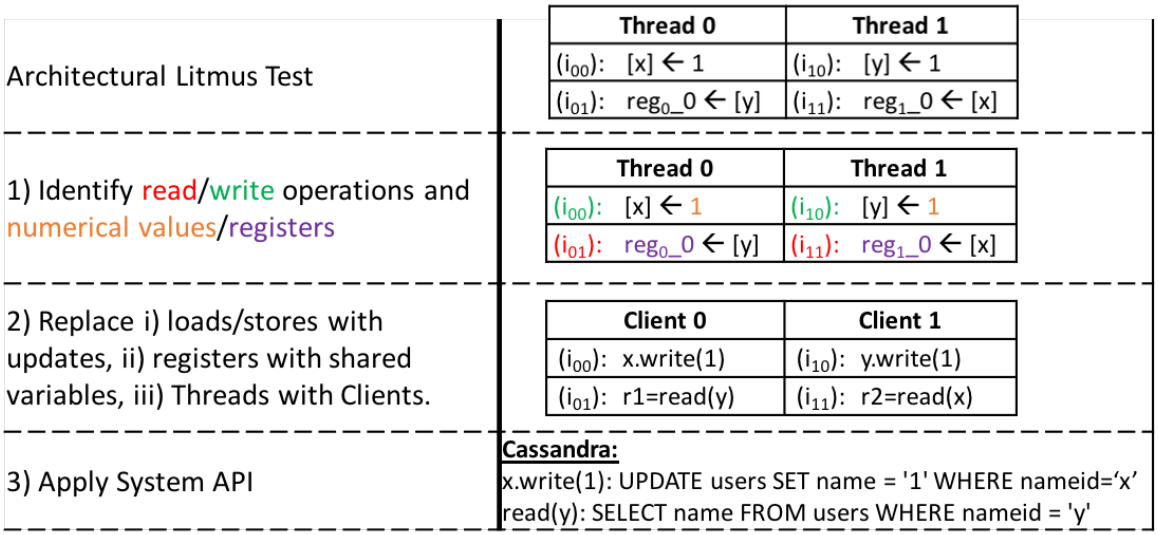


Figure 4.6: Presentation of the methodology for converting architectural litmus tests into DLTs.

acts as a flag and the test checks if the flag has been set and the value 0 is observable in i4, indicating a weak condition. CO-MP tests if writes can become observable in a different order than the one indicated by program order.

Figure 4.5 presents three different scenarios testing client-centric consistency properties as explained in Chapter 2, namely Monotonic Reads and Read Your Own Writes. For the former property, the two tests identify if the value read corresponds to the latest value written or an older written value, whereas Read Your Own Writes tests if writes are observed in the program order for a single client.

4.4.2 Musli Tool Test Converter

To generate DLTs we need to leverage CPU litmus tests. Musli Tool includes a converter tool, that generates tests automatically out of CPU litmus tests and enables automatic testing of outcomes by using a Musli Tool-compatible format for the resulting DLTs. The converter tool is presented in Figure 4.6. Based on the original architectural litmus test, the converter tool identifies load and store operations, the numerical values that these operations operate on, the memory locations affected by these operations and the registers where loaded values are stored. In step 2, loads and stores are replaced with updates for the distributed system and registers with variables, and threads are transformed to be represented as clients. The DLT format applies generally to systems that perform updates. To translate DLTs to a specific system, a System API specifies a mapping between updates and the system specific syntax. This involves placing read and write queries where load and store operations were originally.

Similarly to Musli Tool, the converter tool is extensible and can be used to convert tests to many different systems. To do so, an API needs to be provided for converting from multiple memory consistency models in order to reuse thousands of available tests found in the literature. The test converter also allows creation of different variants of each test by varying test parameters and test configurations. Due to the decoupling of test execution and storage, there are multiple ways to place clients on nodes, taking into account the nodes that have key-value store replicas. The converter can output a set of equivalent DLTs that can be generated for all combinations of clients on the number of available nodes in a cluster.

4.4.3 Distinguishing Between Consistency Guarantees

Based on a system's consistency guarantees, DLTs have the ability to validate which consistency guarantees are implemented in a system in practice via observable or

unobservable outcomes. This property is also useful in classifying systems with respect to which offers stronger or weaker guarantees; observing an outcome in one and not observing an outcome in another provides a partial ordering based on the guarantee under test. DLT outcomes that can be used to classify between systems are defined as *distinguishable*. This subsection presents an example of a DLT running on a distributed key-value store under different consistency settings and demonstrates how a test outcome can be distinguishable by presenting key-value store internals.

Figure 4.7 presents a timing diagram of the Monotonic Reads test for a Cassandra key-value store cluster with a replication factor of 3 for Cassandra's QUORUM (top) and ONE (bottom) consistency settings. On the vertical axis there are three different local copies of the data, replicas 1, 2 and 3 and the coordinator node. Each read/write request can be received by any node in the cluster. When a client connects to a node with a request, that node becomes the coordinator for that request. Time is shown along the horizontal axis. Operations are carried out by a single client and the arrows indicate the participating system components in each event. The Monotonic Reads DLT executes i1-i4 updates in sequence; presented between the dashed lines are the corresponding key-value store operations.

The outcome presented in the Monotonic Reads DLT example in Figure 4.7 is able to distinguish between systems that guarantee monotonic reads and ones that do not. In order to understand why an outcome is unobservable in the top case and observable in the bottom case, we need to look at how reads and writes execute the Monotonic Reads test, introduced earlier in Section 2.3.4, when QUORUM and ONE consistency levels are used.

In the top scenario in Figure 4.7, the Client performs two write updates in i1 and i2, followed by two read updates in i3 and i4. When the client makes the write request, the coordinator receives x.write(1) and issues a write to all replicas for shared object x in the cluster. In Cassandra all replicas are sent a write request in parallel

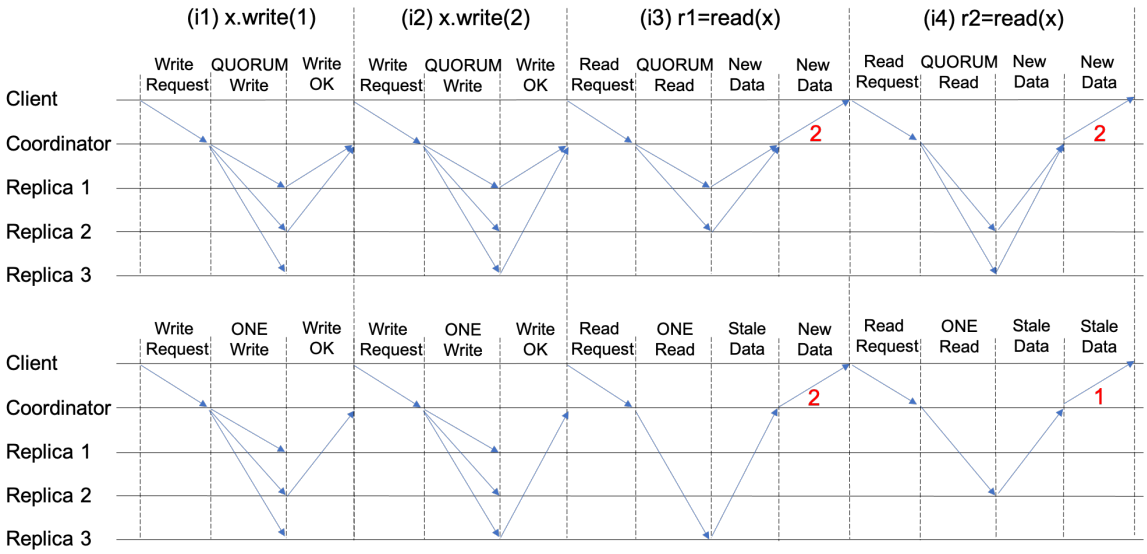


Figure 4.7: Timing diagram for the Monotonic Reads test on the Cassandra key-value store under the QUORUM (top) and ONE (bottom) consistency settings. A single client performs consecutive write updates i1 and i2 followed by consecutive read updates i3 and i4. The Cassandra cluster has a replication factor of 3 for shared object x. The values r1=2 and r2=1 in the bottom diagram correspond to an outcome that is distinguishable between the QUORUM (Forbidden) and ONE (Allowed) consistency settings.

by default, but since the write consistency level is QUORUM, an acknowledgement is only returned after a majority of the copies are updated with the new entry and hold the new data value for object x. The write for update i2 proceeds similarly.

Each replica holds a value for object x, either new or stale, and a timestamp; this timestamp allows the system to determine which value between replicas is the most recent and is taken as the true copy of the data for the shared object. For updates i3 and i4, the read consistency level is QUORUM; a majority of the replicas for object x are queried, and the copy with the latest timestamp is returned to the client. In this Monotonic Reads example, to establish QUORUM, 2 out of 3 replicas will be queried, and based on the previous write updates, the value 2 will be returned for all combinations of replicas queried for i3 and for all combinations of QUORUM writes in i1 and i2. Therefore, i4 cannot return a value for x older than 2, satisfying the monotonic reads property.

If the contacted replicas disagree on the value of the shared object, the replica with the most recent version will return the object. In the background, the third replica is checked for consistency with the first two, and if needed, a read repair is initiated for the out-of-date replicas. Read repair is the process of repairing data replicas during a read request if replicas involved in a read request are not consistent at the given consistency level and the replicas are updated accordingly. Cassandra performs a blocking read repair to ensure that under QUORUM, monotonic reads are preserved. This is achieved by the read repair running in the foreground in a blocking fashion, so that a response is not returned to the client until the read repair has completed and all replicas are updated [52].

In the bottom scenario in Figure 4.7, the Client performs two write updates in i1 and i2, followed by two read updates in i3 and i4 using both read and write consistency level ONE. The write updates in i1 and i2 only require a response from a single replica; two or more replicas can now be in disagreement about the value of object x. In the case of ONE consistency for read updates, a single replica for a given object is contacted to complete the read request. Consecutive reads can read from different replicas that hold different values; there is no guarantee that a replica will not return a stale value after another replica returns the most recent copy of object x. Running the Monotonic Reads test on a Cassandra cluster with ONE consistency setting does not ensure monotonic reads.

4.5 Generating, Executing and Analyzing Distributed Litmus Tests

4.5.1 Musli Tool

This section introduces Musli Tool, a tool that takes its name as an anagram of "litmus". The development of Musli Tool is one of the contributions of this dissertation and a stepping stone for executing DLTs on clusters of key-value stores. Since litmus tests are not deterministic, repeated testing is important for allowed outcomes to become observable. Musli provides a framework for issuing large numbers of iterations of a given test.

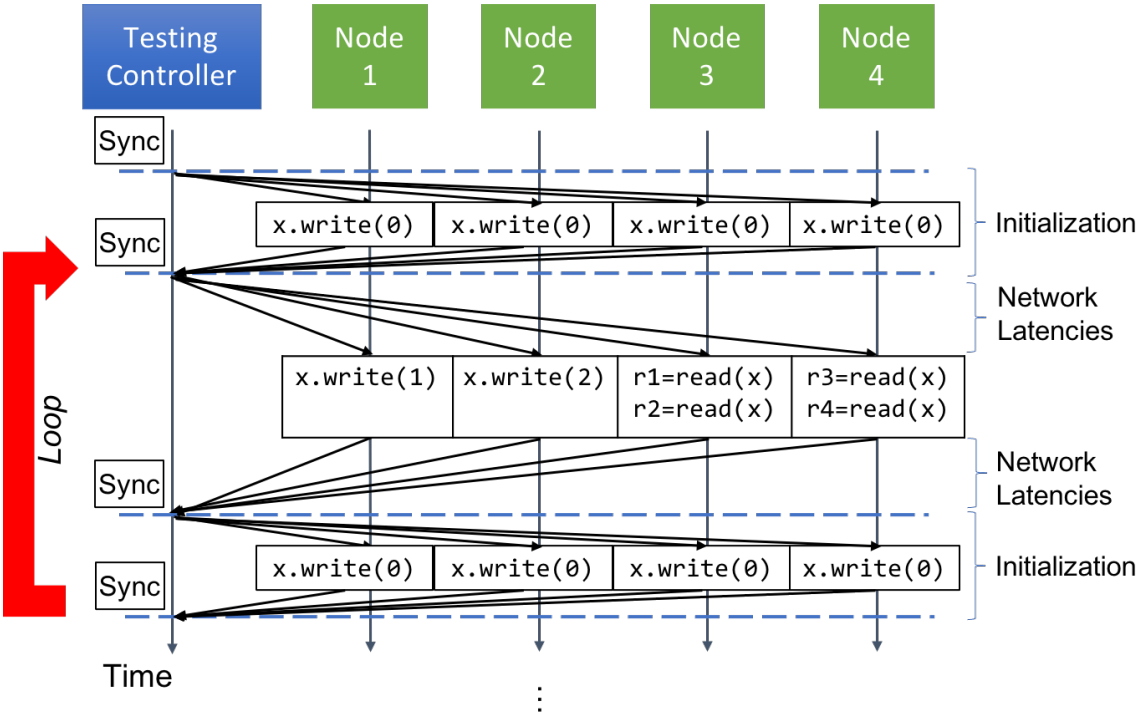


Figure 4.8: Presentation of the execution of a DLT on a distributed key-value store. Read and write updates on a replicated object x are executed by clients running on Worker nodes of a key-value store cluster. The Testing Controller is responsible for test initialization, distribution of updates which execute on every Worker node and test coordination and synchronization.

Figure 4.8 demonstrates the execution of the *Concurrent Writes* DLT presented in earlier sections on a Musli Tool cluster. The Musli Tool testing cluster for this DLT includes four Worker Nodes and one Testing Controller node. Time in this figure progresses from top to bottom and a loop denotes the iterative nature of empirically testing distibuted systems using DLTs. The Testing Controller performs an initialization step for all shared objects accessed in the DLTt, before the test executes repeatedly in a loop. In this example, the shared objects is simply object x. Within the loop, the Testing Controller is responsible for distributing updates to the Worker nodes, which are then executed. Since the Testing Controller coordinates the execution of the DLT over the network there is need for synchronization between Worker nodes between each iteration. Synchronization is required to i) execute updates across nodes simultaneously and ii) to guarantee the same initial values in all copies of data in the system.

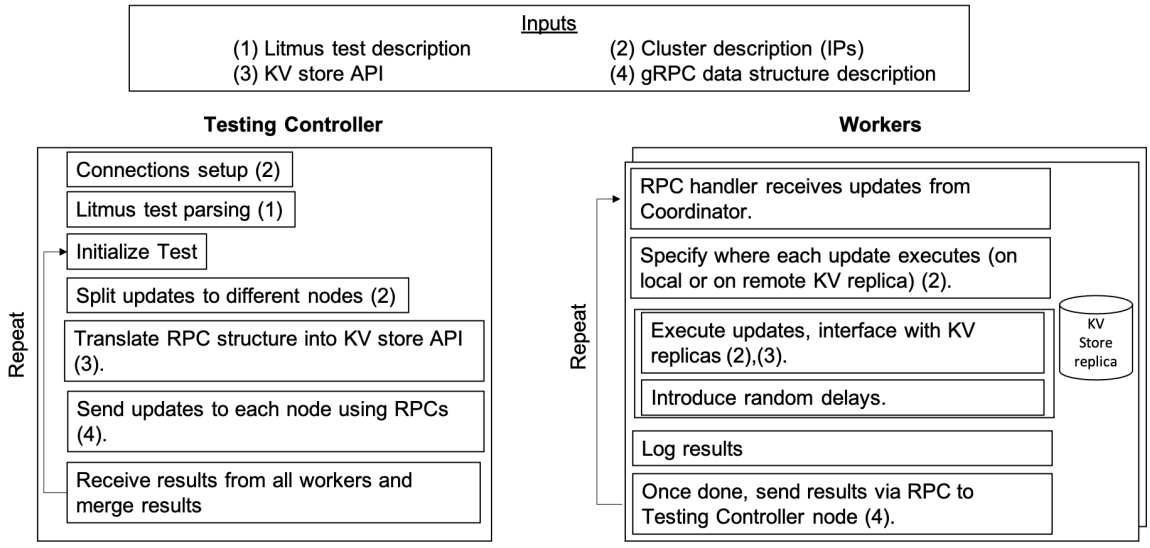


Figure 4.9: Example of a litmus test on a distributed key-value store. Read and write updates on a replicated object x are executed by clients running on a kv-store cluster. The observation of an outcome is used to determine correct or incorrect implementation of consistency model guarantees of the system under test.

Figure 4.9 presents the architecture of Musli Tool. Musli Tool is composed of a Testing Controller node, responsible for the orchestration of DLT execution, and

Worker nodes that execute DLT clients. The Testing Controller is aware of the Cluster description and configuration, the litmus test description, the key-value store API in order to translate the DLT updates to distributed updates on Cassandra and the protocol specifications for the internal communication between Musli Tool nodes.

Musli Tool follows a client-server architecture; the Testing Controller node receives execution parameters as input and also parses the litmus test description, extracting the type of updates that will be performed by each Worker node and the order in which such operations will be performed. The Testing Controller distributes the respective updates to the Worker nodes using RPC calls. Communication between the Testing Controller and the Worker nodes happens using gRPC [60], a Remote Procedure Call framework that encapsulates the litmus test description. The Workers then execute the updates, interfacing with the key-value store replicas on every update and by translating between the DLT description and the Cassandra key-value store updates using a mapping provided by the Testing Controller. During execution, each Worker node receives the execution parameters via the gRPC data structure and performs either synchronous or asynchronous updates and introduces random delays if specified. For each iteration of the DLT, each Worker node logs results that the coordinator will use for analysis. Each client logs the values of its read updates in a file at the local node and the Testing Controller retrieves the logs after the execution of all iterations in order to perform analysis based on the outcomes. The Testing Controller initiates multiple test iterations in series and retrieves all the recorded observable outcomes from the Worker nodes for the system under test. It then builds a histogram of outcome frequencies.

A conversion process from the litmus test format to the key-value store API of the system under test (e.g., Cassandra) happens offline. In the litmus test description, there is information about the clients and the placement of each client on cluster nodes along with the updates and the order these updates will execute.

Musli is highly configurable and enables expansion of testing to multiple systems and topologies. A user can introduce an API for a new system under test by specifying the system specific functions and queries for the corresponding read, write and delete updates. Also, Musli Tool is aware of the cluster configuration and can adapt a DLT to changes in the topology and changes in parameters.

To stress test the system, Musli Tool enforces variable delays to increase randomness and allow different interleavings of updates to appear. Delay ranges are specified for every single test update as part of the test description. By specifying a delay range, a randomly chosen delay within the range of 0 ms to *delay* ms is inserted before every DLT update, which generally allows for more interleavings to appear.

Musli Tool additionally enables two different ways for distributed updates to interface with the distributed key-value store, to perform *asynchronous* or *synchronous* distributed updates. In the case of *synchronous* updates, when an update is performed to the key-value store, the client blocks execution until a response is received by the key-value store. In the case of *asynchronous* updates, the client can start executing the next update to the key-value store in program order and the response for the first update will be updated using a callback function.

4.5.2 Analysis of Distributed Litmus Test Outcomes

For DLTs to be effective, observable test outcomes need to be classified as allowed or forbidden in order for the testing methodology to be able to identify consistency violations for the system under test or determine system behavior in line with the consistency model specification. Compared to testing consistency model properties in multiprocessors, distributed systems may allow for multiple and different system configurations that affect in practice which consistency properties are enforced by the system and which DLT outcomes become observable or unobservable. To establish a ground truth, modeling tools can be used to specify allowed and forbidden outcomes

for a given consistency model theoretically. Modeling tools exist and are widely used for litmus tests in multiprocessors [9, 23]. In distributed systems, modeling tools to aid with the formal specifications of consistency models and system configurations are limited. As a result, it is challenging to establish a ground truth for outcomes of DLTs as a consistency testing methodology.

In order to draw insights from test results using the test suite on a practical system, this chapter introduces theoretical bounds as ground truth; an upper and a lower bound for outcome variety based on different theoretical consistency guarantees. Since there exists a hierarchy of consistency models, providing both strong and weak consistency guarantees, relationships can be established between consistency models to bound the number of outcomes that are allowed. A stronger consistency model provides a lower bound in the number of allowed outcomes and a weaker guarantee provides an upper bound. Since testing using DLT is empirical, performing a test for a finite number of iterations provides a practical lower bound of outcomes that can be observed for a given configuration of a distributed system and its consistency levels.

To determine theoretical bounds of outcomes, this thesis leverages modeling tools used for multiprocessors [9]. In sequential consistency, operations appear to take place in some total order, and that order is consistent with the order of operations on each individual process [84]. Sequential consistency is a consistency model applicable both on multiprocessor operations and distributed updates. Therefore, sequential consistency can serve as a theoretical lower bound on number of outcomes for the consistency testing survey in this chapter. To determine a theoretical upper bound, the number of outcomes is calculated when considering all potential interleavings of updates combinatorially.

4.6 Evaluation Methodology

This section presents the evaluation methodology for Musli Tool and for the Distributed Litmus Test measurement study later in this chapter.

4.6.1 Testing Environment & Setup

We evaluate Musli Tool by deploying it to test a replicated distributed Cassandra key-value store cluster. Musli Tool is evaluated on a five-node x86 cluster and the suite of Distributed Litmus Tests presented in Section 4.5. The five-node cluster is hosted on the Linode [88] cloud provider. Experiments are run on Ubuntu 16.4 Linux nodes with four virtual cores and 8 GB of memory. Musli Tool is implemented in Python using a client-server model as described in earlier sections. For this evaluation, a Cassandra cluster, version 3.11.4, is set up to use a replication factor of three, and all replicas are distributed across the five available nodes.

4.6.2 DLT Measurement Study

For the evaluation of Distributed Litmus Tests as a testing methodology and the evaluation of the characteristics of Musli Tool, a measurement study is performed. The measurement study demonstrates the ability of Musli Tool to test distributed systems for specific consistency guarantees efficiently and effectively. In addition, this measurement study establishes the value of using DLTs in testing, as it allows correctness monitoring of a system by tracking outcome variety over time as well as profiling different systems for the consistency guarantees they provide.

DLTs are executed using Musli Tool to observe how outcome variety changes over time across the number of tests in the test suite across the range of possible cluster configurations and across the number of evaluated parameters. Monitoring the system behavior can allow the user to infer how the system behaves over time and if weak

behaviors appear in the system under test and with what frequency. In addition, a user can observe the behavior of Distributed Litmus Tests over time, how many distinct outcomes are observed over the course of an experiment, how fast each distributed litmus test converges to a number of distinct outcomes and how convergence and the number of distinct outcomes varies across the set of parameters evaluated.

As part of this study, we show how our suite of DLTs with different configurations and a range of execution parameters enables stress testing of the system. By varying consistency settings on a system, it becomes possible to allow fewer or higher numbers of outcomes to appear. When the number of outcomes is different between two distinct consistency properties, this test becomes distinguishable across these consistency settings, e.g. monotonic quorum reads is distinguishable across QUORUM and ONE consistency settings on Cassandra as seen in Section 4.4.3.

4.6.3 Execution Parameters

Musli Tool is evaluated for a set of execution parameters. Table 4.1 summarizes all the execution parameters that were evaluated while running Musli Tool as part of the measurement study in this chapter along with their respective parameter values. A total of 11 Distributed Litmus Tests are evaluated for the combinations of parameters introduced in Table 4.1. In order to vary Distributed Litmus Test execution and exercise consistency properties, delays are introduced. Since DLT updates are performed over the network to different replicas, delays are placed before each update to introduce randomness. Musli Tool allows specification of a delay range. For the measurement study, a choice of 0 ms (no delays), 10 ms, 20 ms and 50 ms are evaluated. There is an additional random option, which allows selection of a random delay within a range of 0 ms and 50 ms. For every choice of the delay setting, the delay between updates takes a value randomly between 0 ms and the randomly selected integer value.

Execution Parameter	Parameter Values
Client Placement	local — non-local
RPC Request Type	asynchronous — synchronous
Delay	0 ms — 10 ms — 20 ms — 50 ms — random
Cassandra Consistency Level	ONE — QUORUM

Table 4.1: Execution parameters and their associated values used for this measurement study.

Musli Tool allows configurable placement of clients. The majority of DLT clients can be placed on cluster nodes that host key-value store replicas, in which case clients have faster access to the key-value store and the test is called *local*. Similarly, when the majority of clients are placed on nodes not hosting a key-value store replica, the test is called *non-local*. Another parameter of Musli Tool is the type of request the Musli clients perform on the Worker nodes to contact the key-value store nodes for the updates to be executed. Synchronous requests block the execution of the client until a response is performed, whereas asynchronous requests continue executing and receive the response asynchronously.

As part of this measurement study, two different consistency levels on the distributed Cassandra key-value store are used, namely ONE and QUORUM. ONE is a weak consistency setting which specifies that only a single replica needs to return a response to the client after a read update. QUORUM requires a majority of nodes to respond to the client [28]. These two consistency settings are evaluated across all combinations of the local/ non-local and synchronous/asynchronous parameters for a fixed value of delay.

4.6.4 DLT Outcome Ground Truth

To perform a consistency measurement study, it is required for each DLT for each system configuration or consistency model implementation to specify which outcomes are allowed or forbidden in order to compare them against outcomes that are observable

DLT Name	Sequential Consistency (SC)	Combinatorial
Causality	83	243
Causality 2	138	256
Concurrent Writes	47	81
Stale Reads	7	8
Store Buffering	3	4
Load Buffering	3	4
Message Passing	3	4
CO-MP	6	9
Monotonic Reads	1	9
Monotonic Reads 2	1	4
Read Your Own Writes	1	3

Table 4.2: Number of possible outcomes on a correct implementation for the DLT suite for sequential consistency and combinatorial calculation. The allowed outcomes in this table serve as theoretical lower and upper bounds for the observable outcomes observed under different Cassandra configurations.

and unobservable after executing DLTs using Musli Tool. As explained in Section 4.5, the herd tool is used in order to generate upper and lower bounds for the consistency settings evaluated in this study [9]. Table 4.2 presents the upper and lower bounds of the number of outcomes for DLTs by using herd models for sequential consistency and a combinatorial upper bound without inserting happens before relationships between updates.

4.6.5 DLT Measurement Data Metrics

Outcome Variety

One goal of Distributed Litmus Tests is to exercise a set of update interleavings effectively by enabling interaction between clients and replicas and by generating a great variety of test outcomes effectively. To evaluate Musli Tool with respect to this goal, an evaluation is performed about how frequently each possible test outcome occurs in Musli Tool across the different test parameters for the same number of iterations. The theoretical lower and upper bounds discussed previously serve as the baselines for comparison of the expected numbers of outcomes. Outcome variety is

also evaluated over time, as it helps draw insights into the dynamic behavior of DLT execution.

Execution of the updates in each DLT is non-deterministic and as a result each iteration of a test is not guaranteed to yield the same outcome. For each execution parameter combination examined in this study, the average outcome variety is shown. When comparing the outcome variety across execution parameters, the subset of observable outcomes might differ slightly. Throughout this study, all outcomes observed are allowed, but not all appear within a specified test length as executions are non-deterministic.

Testing Throughput

To evaluate the efficiency of Musli Tool, performance needs to be considered in addition to outcome variety. The throughput of Musli Tool is evaluated for all execution parameters considered in this thesis. Musli Tool throughput is calculated by measuring the total measurement execution time for a given measurement length in terms of DLT iterations. Throughput data are presented in DLT iterations per minute across all parameters.

4.7 Distributed Litmus Test Measurement Study

This section demonstrates the value of Distributed Litmus Tests and evaluates Musli Tool by performing a measurement study using the execution parameters and metrics described in Section 4.6. In particular, this measurement study (i) validates the usefulness of Musli Tool as a testing tool for correctness and evaluates the effects of each execution parameter with respect to achieving fast and efficient consistency testing, (ii) demonstrates how Musli Tool and DLTs can be used in practice for consistency testing and system monitoring, and (iii) presents use cases about observing the runtime

behavior of the system under test during DLT execution and how DLTs outcomes become distinguishable across systems implementing different consistency guarantees in practice.

4.7.1 Distributed Litmus Test Characteristics

Figure 4.10 presents the runtime behavior of all 11 DLTs in the test suite during a 500 iteration execution. The DLTs execute using asynchronous RPCs, a non-local client placement, and a 20ms delay between updates, when the ONE consistency setting is used. The results demonstrate each DLT's outcome variety over time, how many distinct outcomes are observed during the execution of the test and at which point in time. From the conducted measurements, it is clear that the 11 different DLTs have distinct behavior and runtime characteristics within the 500 iteration execution window. Out of the 11 tests, the majority (Store Buffering, Load Buffering, Message Passing, CO-MP, Read Your Own Writes, Monotonic Reads, Monotonic Reads 2 and Stale Reads tests) converge within the first 50 iterations to a number of distinct outcomes that remains unmodified until the end of the execution. DLTs Concurrent Writes, Causality and Causality 2 observe a higher number of distinct outcomes throughout the course of the test execution. Concurrent Writes only observes a few additional distinct outcomes past the 150 iteration mark, Causality continues to observe distinct outcomes frequently until the 300 iteration mark whereas Causality 2 observes new distinct outcomes throughout the test execution. The difference in DLTs' convergence is inherent to each test's design; the higher the number of clients and read updates in DLTs, the higher the number of allowed outcomes and the higher the number of outcomes that are observable in practice as there exist more possible interleavings.

When evaluating a system using a litmus test, we need to account for the execution length of DLTs; if the number of test iterations is high, the number of allowed distinct

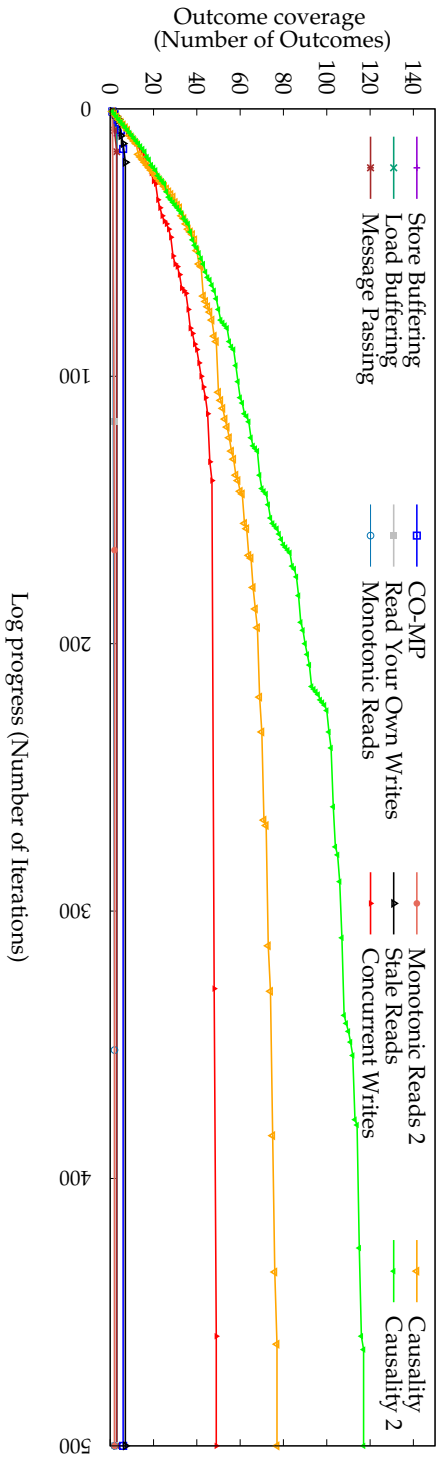


Figure 4.10: Outcome Variety over time results for all tests in the litmus suite for the combinations of parameters (async, non-local, 20ms delay, ONE consistency setting). Results are presented over the course of 500 iterations, and the y-axis presents the number of different outcomes observable on Cassandra.

outcomes that will become observable will plateau. However, in the case the number of test iterations is limited, the number of newly observable allowed outcomes will continue to increase throughout the course of the test execution.

Apart from the outcome variety expressed by the number of distinct outcomes, the design of a Distributed Litmus Test can affect how frequently a particular interleaving of updates can be expressed when run on a system and consequently how frequently an outcome will be observed. In addition to the DLT design, the frequency of an outcome also depends on the consistency guarantees of a system under test and how the testing methodology exercises update interleavings leading to a specific outcome. For example, when running Monotonic Reads and Monotonic Reads 2 on DLTs for 500 iterations under the ONE consistency setting, the outcome corresponding to preserving monotonic reads was observed 499 times whereas the outcome violating that property was only observed once. By monitoring the DLT outcome frequencies we can identify rarely occurring outcomes, and use the outcome frequency information while varying execution parameters to better understand system behavior.

The outcome variety, the frequency in outcome observations and the discrepancy in outcome frequencies for a DLT can provide indications about the guarantees that the system provides. By running DLTs on a system periodically, we can use Musli Tool as a system monitoring tool and draw insights on the cluster behavior and health.

4.7.2 Execution Parameter Exploration on Musli Tool

Client Placement and Request Type

Figures 4.11, 4.12 and 4.13 present the outcome variety observed for individual DLTs over time across all combinations of parameters of synchronous/asynchronous RPC requests and local/non-local client placement using 20ms delay when Cassandra is configured to use the ONE consistency setting. For each figure, there are horizontal lines denoting the theoretical lower and upper bounds corresponding to allowed

outcomes under Sequential Consistency (labeled as SC Bound) and the combinatorial calculation of outcomes (labeled as Combinatorial Bound). These figures provide more insight into the runtime behavior and characteristics of each individual DLT, allowing comparison of how Musli Tool fares against the theoretical bounds; we would expect the outcome variety of DLTs run on Musli Tool to be between theoretical bounds for the Cassandra ONE setting, but the outcome variety can be below the lower bound as well, as Musli Tool is not guaranteed to exercise and observe all allowed DLT outcomes within a test length of 500 iterations. This DLT measurement study evaluates the outcome variety metric across different execution parameters of Musli Tool.

For each subfigure in 4.11, 4.12 and 4.13, we can observe the outcome variety for all combinations of client placement parameters and request type execution parameters. For each of the options available (labeled as local/sync, local/async, non-local/sync, non-local/async), we observe the maximum outcome variety reached and how these outcome variety values compare to the theoretical bounds. Synchronous RPC calls to initiate DLT execution are blocking calls and allow significantly less update interleavings in comparison to asynchronous RPCs, as indicated by the experimental results in Figures 4.11, 4.12 and 4.13. Consequently, using synchronous RPC requests usually leads to fewer unique outcomes than asynchronous RPC requests. When evaluating client placement as an execution parameter, local and non-local perform similarly in terms of outcome variety. Both client placement options observe similar outcome variety and follow a similar rate of convergence to the respective outcome variety numbers.

The number of outcomes observed for each litmus test is typically higher or equal to the lower bound of distinct outcomes corresponding to Sequential Consistency. This observation is expected as measurements are performed on a Cassandra cluster operating under ONE consistency setting, which is providing significantly weaker consistency guarantees than Sequential Consistency and allows the observation of a

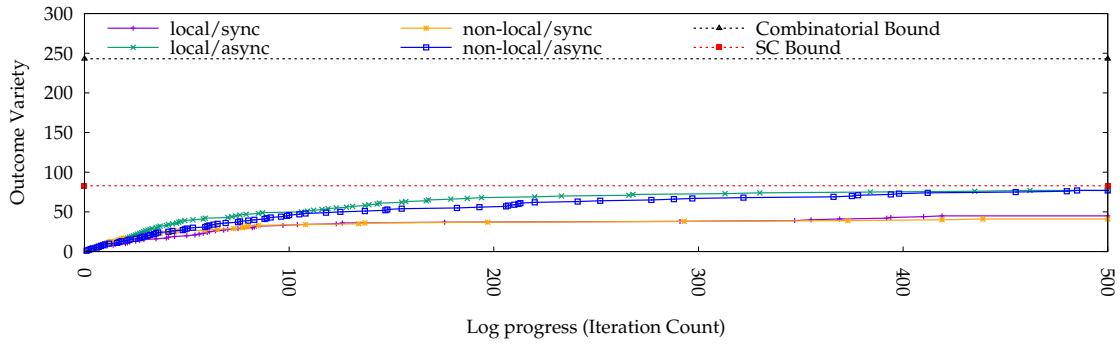
higher number of distinct outcomes. The outcome variety for each DLT is always lower than the theoretical combinatorial upper bound, as the combinatorial bound includes both allowed and forbidden outcomes.

Distributed Litmus Tests Causality, Causality 2 and Concurrent Writes in Figure 4.11 demonstrate higher outcome variety and exhibit higher variance in outcome variety across the different execution parameters compared to the other tests in the test suite. Across the remainder of the evaluation we will focus the evaluation on this subset of tests. Synchronous RPCs limit randomness and, consequently fewer interleavings can appear, whereas local and non-local client placement yield similar results.

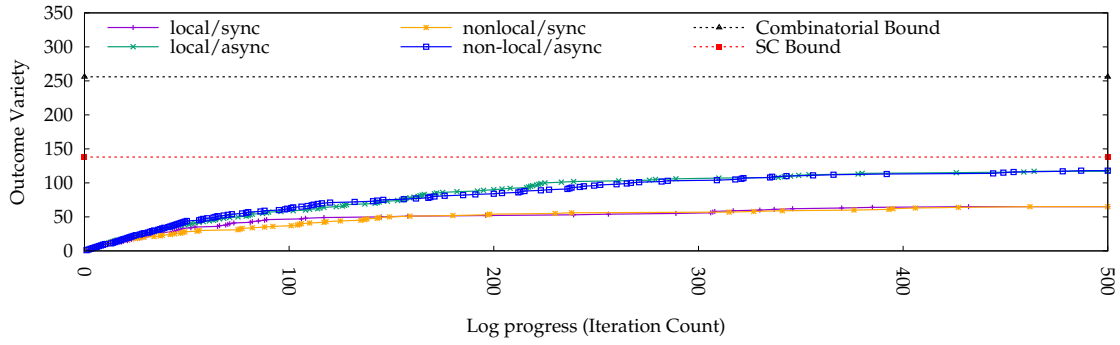
Stress Testing Using Delays

In addition to the impact of client placement and the type of RPC calls, we evaluate how delays affect DLT execution and what effect they have on outcome variety. Figure 4.14 presents the outcome variety of each DLT in the test suite across a range of delay values (0ms, 10ms, 20ms, 50ms and random) when using asynchronous RPC calls and non-local client placement, on a Cassandra system configured with the ONE consistency setting on Cassandra. For each DLT a set of bars corresponds to the five different delay values. The y-axis shows outcome variety where higher is better. In general, one expects the delay values for 10ms, 20ms and 50ms to indicate similar performance in terms of outcome variety, significantly higher than the 0ms and random choice of delay values. However, the measurements do not indicate that an increase in the delay between updates has a strict increase in outcome variety. Insertion of delays introduces additional randomness in the execution of updates and allows for more update interleavings to appear.

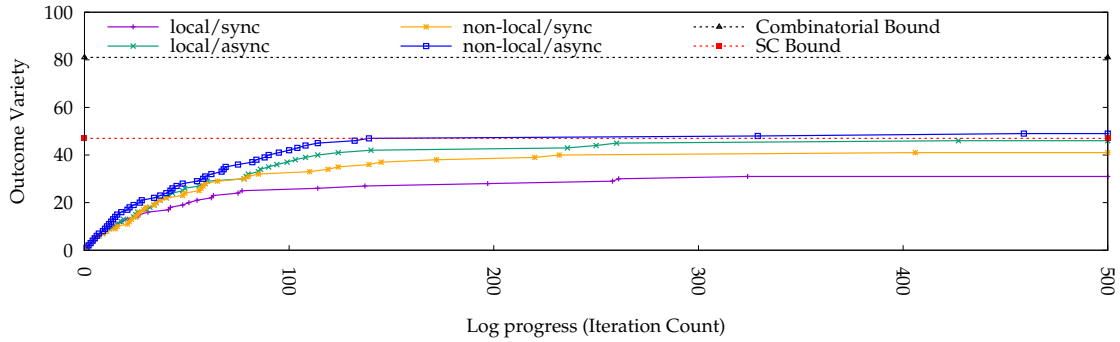
Similarly, Figure 4.15 presents the outcome variety results for each test in the test suite for synchronous RPC calls and the same execution parameters. Results for 10ms,



(a) Causality

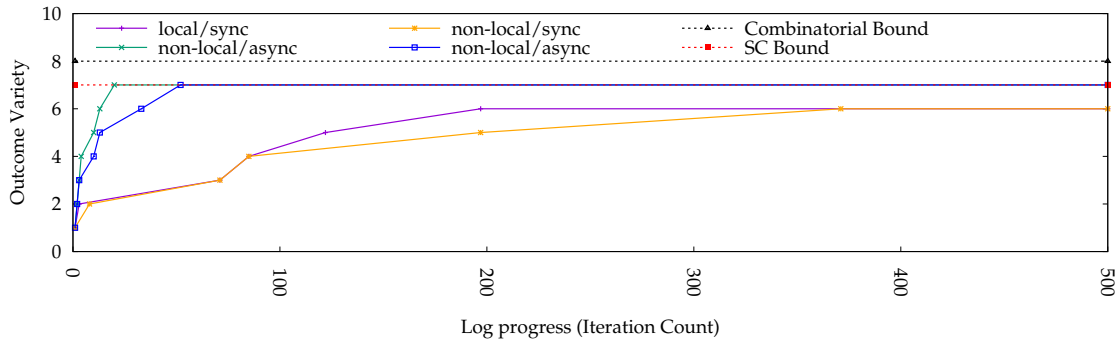


(b) Causality 2

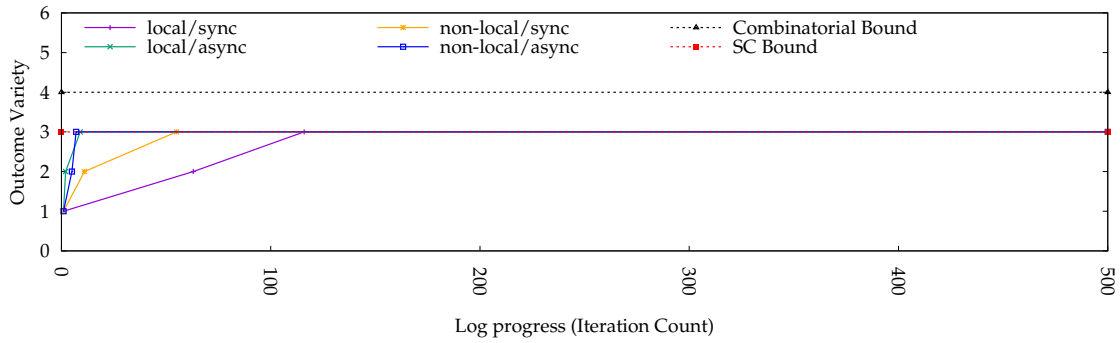


(c) Concurrent Writes

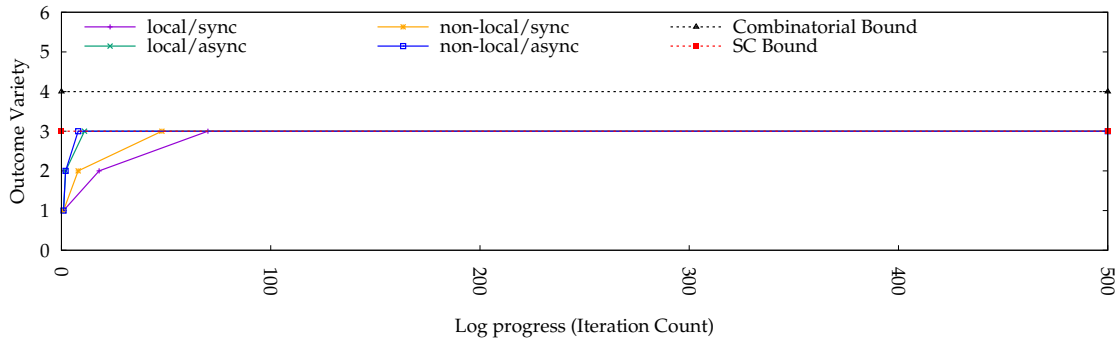
Figure 4.11: Causality, Causality 2 and Concurrent Writes tests' outcome variety over time results for all 4 combinations of parameters (sync/async, local/nonlocal) with 20ms delay before updates. Results are presented over the course of 500 iterations, and the y-axis presents the number of different outcomes observable on Cassandra. The x-axis presents progress over the course of the experiment by indicating the iteration count of DLT executions. The Combinatorial Bound line corresponds to the maximum number of outcomes calculated combinatorially for the particular Distributed Litmus Test. The SC Bound line corresponds to the number of outcomes theoretically allowed for the particular DLT with a system implementing Sequential Consistency.



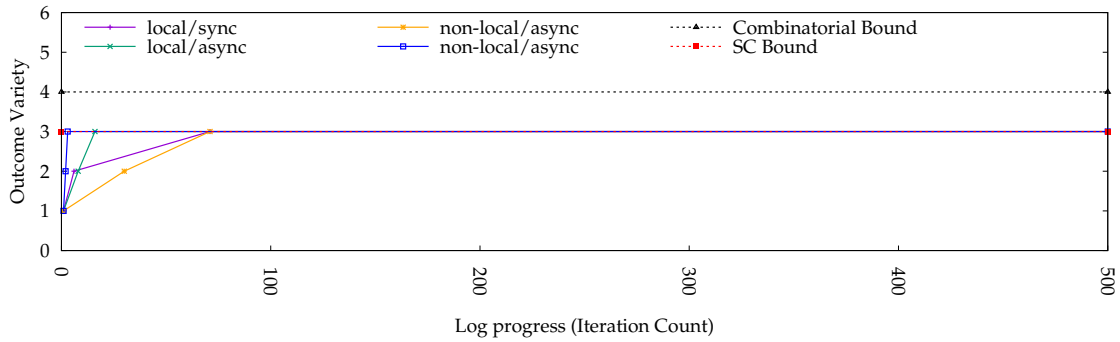
(a) Stale Reads



(b) Store Buffering

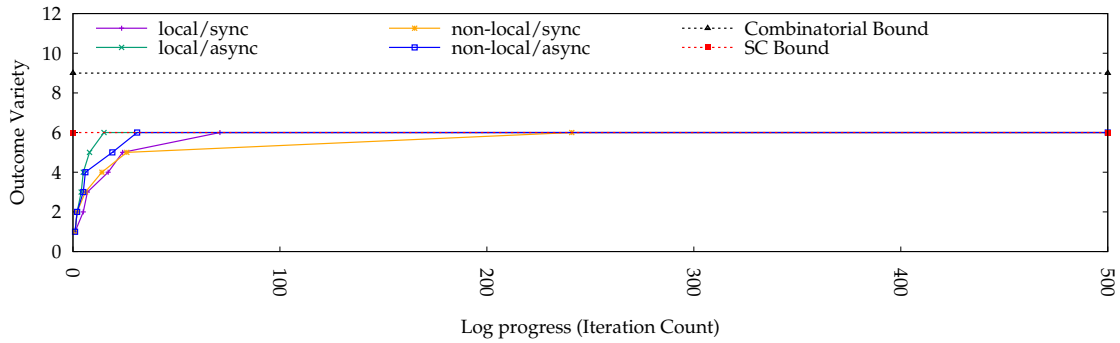


(c) Load Buffering

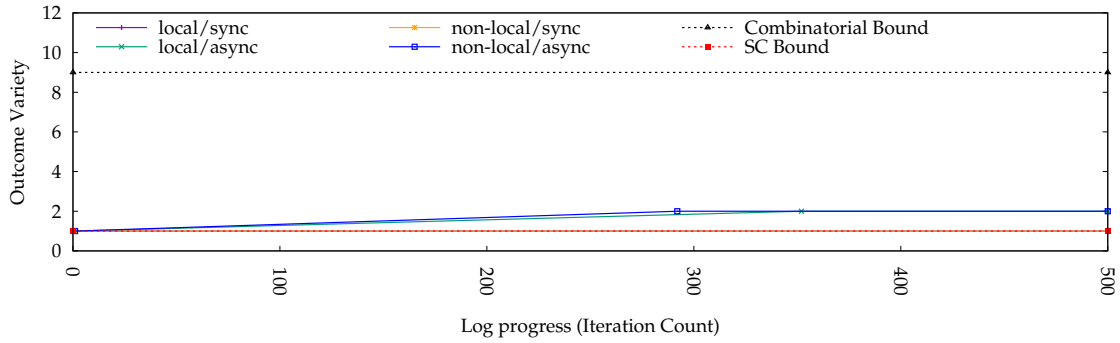


(d) Message Passing

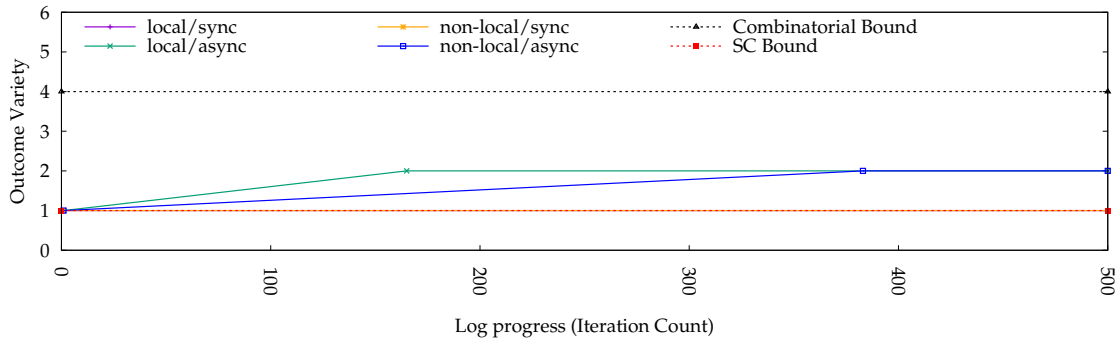
Figure 4.12: Stale Reads, Store Buffering, Load Buffering, Message Passing tests' outcome variety over time results for all 4 combinations of parameters (sync/async, local/nonlocal).



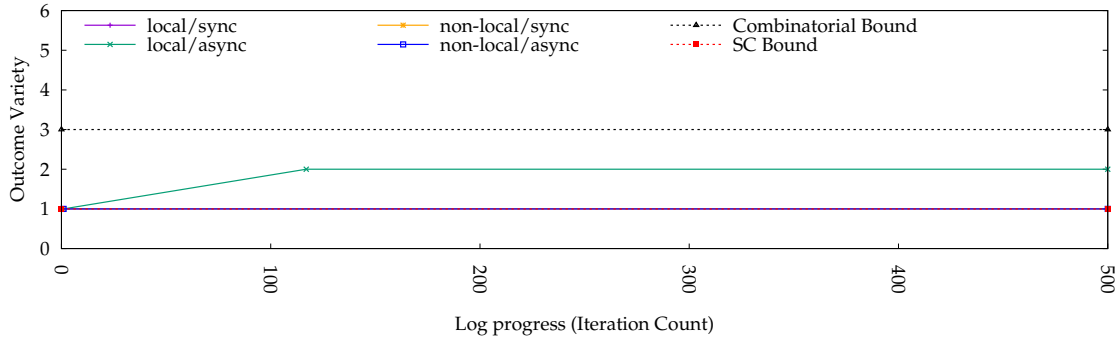
(a) CO-MP



(b) Monotonic Reads



(c) Monotonic Reads 2



(d) Read Your Own Writes

Figure 4.13: CO-MP, Monotonic Reads, Monotonic Reads 2, Read Your Own Writes tests' outcome variety over time results for all 4 combinations of parameters (sync/async, local/nonlocal).

20ms or 50ms delays demonstrate benefits in terms of outcome variety compared to zero or random delays as the corresponding bars trend higher compared to the other delay values. However, the difference in outcome variety is less apparent compared to the case of asynchronous RPCs as blocking calls limit how delays introduce randomness.

In both Figures 4.14 and 4.15 there is limited to no variability for DLTs with fewer overall outcomes of interest (all DLTs except for Causality, Causality 2 and Concurrentwrites) across the range of delay values.

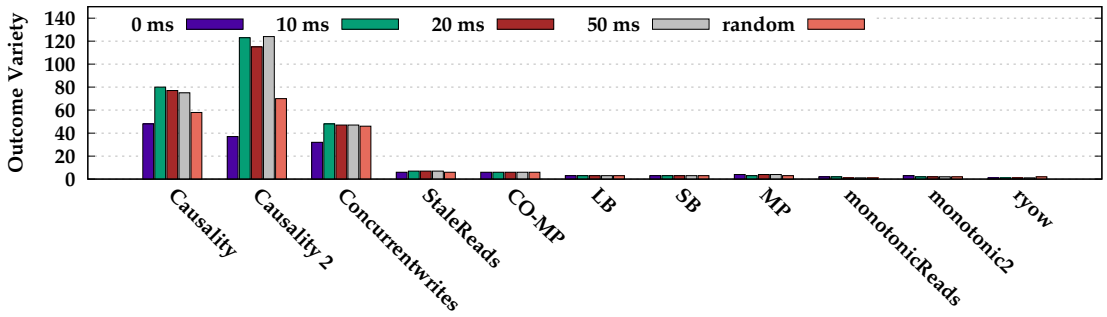


Figure 4.14: Musli Tool's outcome variety during testing across a range of delay values (0ms, 10ms, 20ms, 50ms, random) for asynchronous RPC calls for all DLTs in the test suite. Outcome variety is measured in distinct outcomes observed, and higher is better.

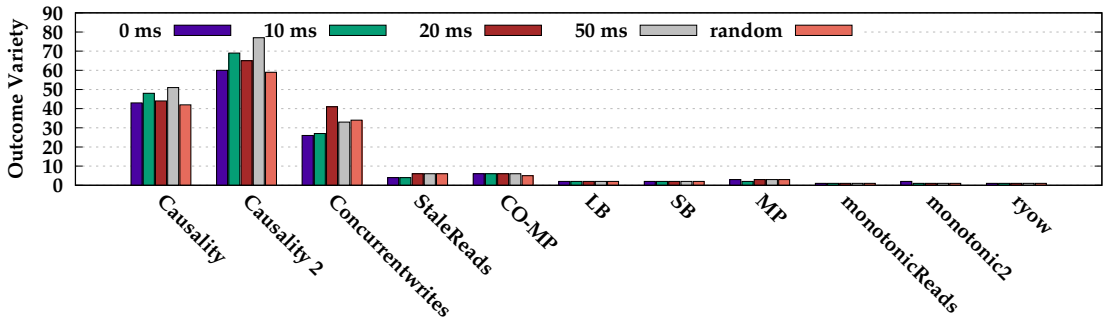


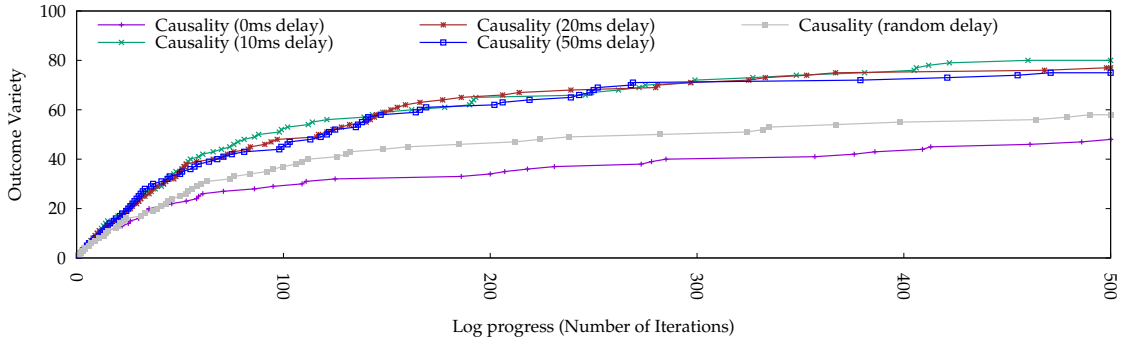
Figure 4.15: Musli Tool's outcome variety during testing across a range of delay values (0ms, 10ms, 20ms, 50ms, random) for synchronous RPC calls for all DLTs in the test suite. Outcome variety is measured in distinct outcomes observed, and higher is better.

Figure 4.16 presents the runtime behavior of DLTs across different choices of delay between updates for the Causality, Causality 2 and Concurrent Writes DLTs. Delays affect how a DLT converges to a certain outcome variety, with zero or random delays observing a lower outcome variety value in comparison to the other choices of delay values. If measurements were run for a higher number of iterations, testing using 10ms, 20ms or 50ms would lead to a higher outcome variety or would converge to an outcome variety number sooner compared to using zero or random delays.

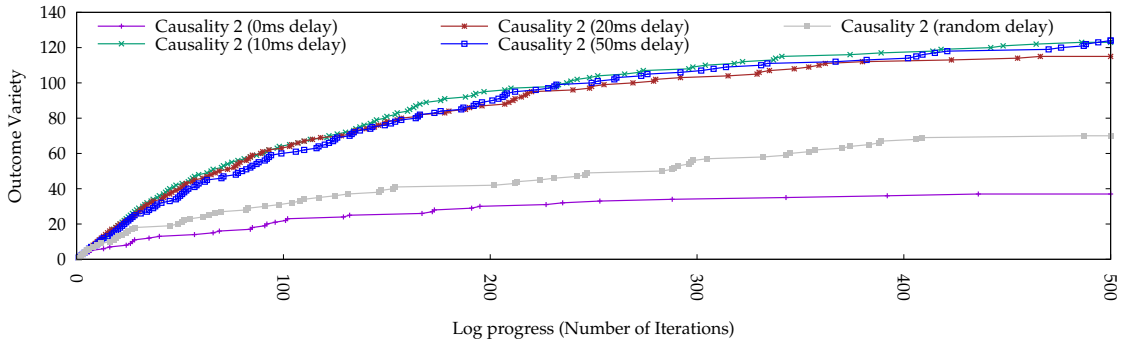
Measurements on Musli Tool across the test suite for varying values of delays between updates demonstrate a outcome variety benefit in the presence of non-zero delays. Delay values for 10ms, 20ms or 50ms demonstrate higher outcome varieties compared to 0ms or random delays. Increasing the delay value past 20ms provides diminishing returns as 50ms does not yield higher outcome variety than 20ms. As a result, a 20ms delay value was the default choice of delay value across the measurement study. Finally, it is important to note that when increasing the delay value, the testing performance and throughput are reduced and therefore a choice of 20ms can be a suitable delay value for the test suite as demonstrated by Figure 4.16, where longer delays yield comparable results to 20ms. We discuss in more detail the impact delay has on throughput in the next section. The effect in outcome variety is less apparent when using synchronous RPCs due to blocking calls.

Testing Across Consistency Guarantees

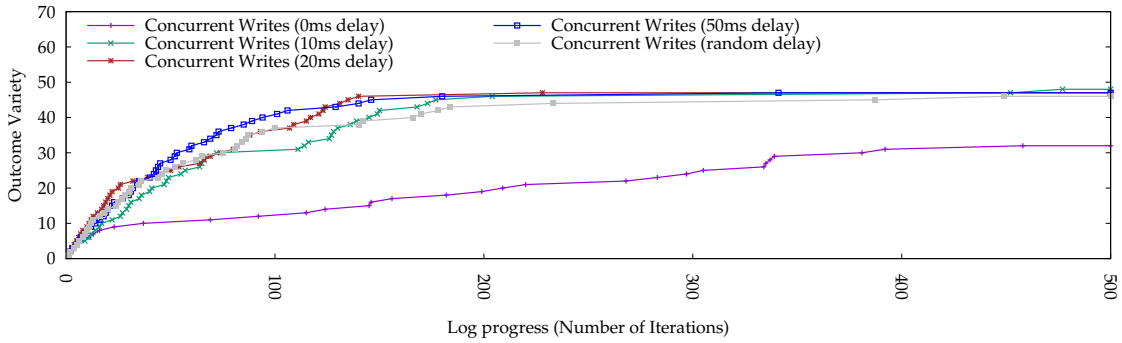
Musli Tool exercises key-value stores using DLTs to uncover observable outcomes under a set of execution parameters. Apart from client placement, type of RPC call and delay between updates, configuring and testing under different consistency settings provides insights into the guarantees of the distributed key-value store and allows validation of the systems' expected behavior and theoretical promises. Musli Tool has the ability to distinguish between consistency models or settings by exercising systems



(a) Causality



(b) Causality 2



(c) Concurrent Writes

Figure 4.16: Musli Tool's outcome variety across the execution of a 500 iteration long experiment for a range of delay values (0ms, 10ms, 20ms, 50ms, random) for asynchronous RPC calls for the (a) Causality, (b) Causality 2, (c) Concurrent Writes Distributed Litmus Test. Outcome variety is measured in distinct outcomes observed, and higher is better.

using DLTs that have outcomes that are observable under ONE consistency setting and unobservable in another. On Cassandra, running DLTs using the QUORUM consistency setting restricts the observation of certain outcomes that are observable under a weaker guarantee such as the ONE consistency setting. Table 4.3 presents the number of observable outcomes for each Distributed Litmus Test for both the ONE and QUORUM consistency settings for a 500 iteration run using asynchronous RPC calls, non-local client placement and 20ms delay between updates. The difference in number of distinct outcomes indicates that the QUORUM consistency setting is more restrictive for the particular DLT and allows fewer outcomes to be observed in comparison with the ONE consistency setting. For DLTs where the difference in outcomes is present, the specific DLT and the particular outcomes are distinguishable across the two consistency settings ONE and QUORUM. Subsection 4.4.3 presents the Monotonic Reads DLT example under both the ONE and QUORUM settings, demonstrating why non-monotonic reads are forbidden and unobservable under the QUORUM consistency setting and allowed and observable under the ONE setting.

When comparing the outcomes observed against the theoretical lower bound (Sequential Consistency), the number of outcomes that belong to the ONE consistency setting are higher than the lower bound for the majority of tests. Since Sequential Consistency provides a stronger consistency guarantee than the ONE setting provided by Cassandra, the outcomes beyond the set of outcomes allowed by the lower bound provide orderings indicative of a weaker consistency guarantee. For the Causality and Causality 2 DLTs, the number of observable outcomes does not exceed the number of allowed outcomes by the lower bound set by Sequential Consistency, as the number of allowed outcomes by SC near the total execution length (500 iterations), as presented in Table 4.2, and therefore not all outcomes are easily observable without a longer execution length. The non-deterministic nature of this testing methodology does not allow results to exceed the number of outcomes set by the theoretical SC bound. The

DLT Name	ONE Outcomes	Quorum Outcomes	SC	Combinatorial
Causality	82	80	83	243
Causality 2	117	113	138	256
Concurrent Writes	49	47	47	81
Stale Reads	7	7	7	8
Store Buffering	3	3	3	4
Load Buffering	3	3	3	4
Message Passing	3	3	3	4
CO-MP	6	6	6	9
Monotonic Reads	2	1	1	9
Monotonic Reads 2	2	1	1	4
Read Your Own Writes	2	1	1	3

Table 4.3: Observable outcomes for the DLT suite for ONE and QUORUM consistency settings in Cassandra key-value store. The DLTs that observe a different number of outcomes for different consistency settings are able to empirically validate which guarantee is implemented in practice and are called distinguishable. The SC and Combinatorial columns indicate the number of outcomes allowed by a system implementing Sequential Consistency and the number of outcomes possible based on all combinations of updates for a particular DLT, respectively. The SC and Combinatorial columns serve as a theoretical lower and upper bound for the outcome variety of the DLT observed in practice on Cassandra.

Combinatorial upper bound always remains higher than the observable outcomes as the upper bound also includes forbidden outcomes that remain unobservable.

Using Musli Tool to exercise distributed systems with Distributed Litmus Tests can allow identification of how many different outcomes are observed per test and with what frequency. Musli Tool can observe outcomes corresponding to weak consistency guarantees and validate these outcomes against the system specification and documentation. With DLTs it is also possible to identify if sets of litmus tests are distinguishable across different consistency models, which allows creation of test suites to infer consistency levels and guarantees.

Performance Evaluation

Profiling DLT performance characteristics allows evaluation of how Musli Tool can be used in practice and enables insights to be drawn into the selection of Musli Tool's

execution parameters. Figure 4.17 presents the testing throughput of Musli Tool for each test in the test suite for asynchronous RPC calls and non-local client placement when using the ONE consistency setting on Cassandra. Throughput is measured in Distributed Litmus Test operations per minute, and in this evaluation of the 5 node Cassandra cluster, we vary delays between updates ranging across the values of 0, 10, 20, 50 (ms) and random for all Distributed Litmus Tests. The figure indicates that throughput ranges between 53 and 511 DLT test iterations per minute.

Insertion of delays naturally affects the throughput of Musli Tool, as having zero delays or inserting low values of delays increases throughput compared to having high delay values. When considering the outcome variety for each of the choices of delay, insertion of 10ms or 20ms as seen in Figure 4.14 performs better compared to the other options as outcome variety remains high while throughput is less compromised compared to higher delay values as presented in Figure 4.17.

Similarly, Figure 4.18 presents throughput measurements for synchronous RPC calls. In comparison to the previous figure, synchronous RPC calls introduce additional latency and as a result, throughput is significantly reduced in comparison to asynchronous RPC calls (note different y-axes values).

Based on the measurements, it is clear that testing throughput heavily depends on the nature of a DLT itself. Tests with more threads, such as Causality, Causality 2 or Concurrent Writes, register lower throughput compared to tests like Store Buffering, Load Buffering, Message Passing or CO-MP that have two threads; this is due to the increased synchronization overhead due to high numbers of threads. Tests like Read Your Own Writes or Monotonic Reads have a higher number of updates per client (e.g., Mononic Reads DLT has one client with 4 consecutive updates) that have to be executed per iteration, which leads to reduced throughput.

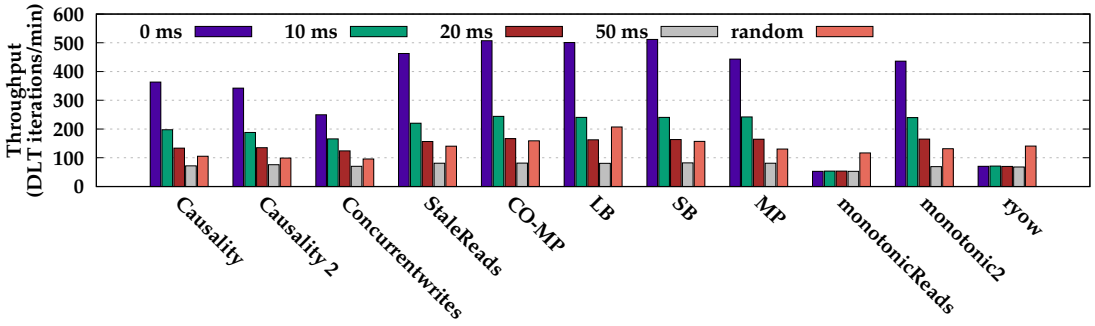


Figure 4.17: Musli Tool's testing throughput across a range of delay values (0ms, 10ms, 20ms, 50ms, random) for asynchronous RPC calls and non-local client placement for all DLTs in the test suite. Throughput is measured in test iterations per minute and higher is better.

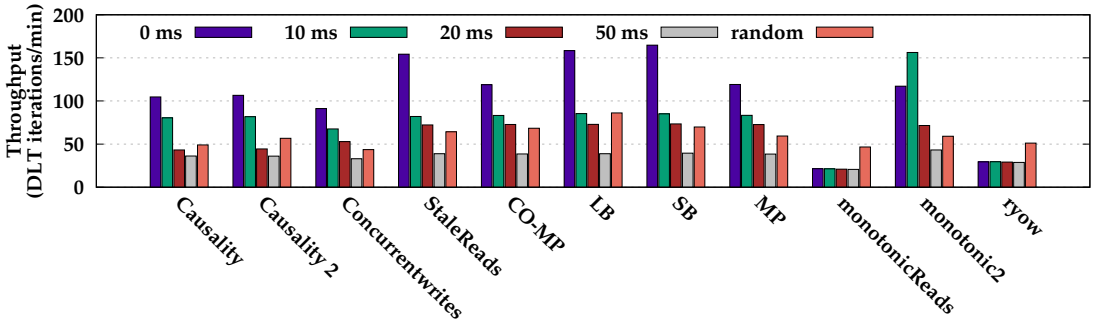


Figure 4.18: Musli Tool's testing throughput across a range of delay values (0ms, 10ms, 20ms, 50ms, random) for synchronous RPC calls for non-local client placement for all DLTs in the test suite. Throughput is measured in test iterations per minute, and higher is better.

4.8 Related Work

Prior work has shown that a number of bugs found in distributed systems have been attributed to errors in implementing consistency [62]. Several general purpose testing methodologies have been proposed for bug finding for distributed systems. Typically, these methodologies either rely on randomly generating inputs that lead to bugs or on tests that precisely exercise a specific property.

4.8.1 Correctness Testing Methodologies

Random testing has proven to be an effective way to catch bugs in distributed systems even if the space of executions leading to potentially faulty or forbidden behaviors is very large [97]. Correctness testing methodologies span a wide range of domains and systems, including finding concurrency bugs in programming languages like Go [122], and discovering distributed consistency bugs [93]. Jepsen is a framework for black-box testing of distributed systems that relies on random testing. Random testing is effective at detecting inconsistencies between the system specification and implementation at the cost of not having an established test suite.

One of the common techniques to exercise the large test space is fuzzing, where the execution of a concurrent program is randomized by perturbing thread or event scheduling in order for testing tools to explore a wide variety of possible outcomes [39, 76, 107]. Another work [32] leverages model checking to reduce the search space for faulty orderings and fuzzing is then performed to allow orderings to manifest themselves. Similarly, in order to build confidence that distributed systems are correctly implemented and to increase system resiliency, faults can be deliberately injected to actively perturb production systems, a practice called Chaos Engineering [13], adapted from prior research on fault injection [15]. Another technique that enables testing of distributed systems as an alternative to property-based testing techniques is the automatic generation of invariants that capture the operation of distributed systems' processes and allow to assert their correctness [58].

A notable approach to discover bugs in distributed systems using random testing is Jepsen [81], a tool that has reported analyses and bug reports on a variety of distributed systems [80].

Jepsen

Jepsen is a black box testing tool designed to test for correctness in distributed systems [81]. Black box testing refers to a testing methodology where the tests do not require any knowledge of the internal code structure and implementation details of the software and mainly focus on the inputs and outputs of the system under test. Jepsen is a Clojure library used to set up a distributed system, run a set of generated updates and operations on that system, and analyze the history of updates and operations to uncover behaviors counter to the specification of the system under test.

The tool architecture for Jepsen is similar to Musli Tool. A control node instantiates a set of single-threaded processes, each with its own client for the distributed system. For each client, a client generator specifies a random sequence of client operations that are executed by the client on the cluster. Jepsen can execute with a variety of execution parameters along with a range of fault injection options like network partitions, process kill signals, slowing disks, etc. Each operation is recorded in a history which is then analyzed with built-in or custom operation history checkers, for example, a linearizability checker. Techniques such as Jepsen and Chaos Engineering perform random testing and additionally introduce fault injection to distributed data stores. We considered evaluating Distributed Litmus Tests on Jepsen in addition to Jepsen's random testing approach. However, Jepsen's programming model is limited and does not allow for the creation of tests structured similar to Distributed Litmus Tests, mainly because generators cannot generate operations that access multiple keys.

The search space of distinct outcomes using this random approach for the system under test is intractable. Existing testing solutions for system correctness require experts who need to study the systems under test, observe system executions, and then hypothesize about which faults and orderings are most likely to expose real system flaws and then craft generators for the updates the tool will perform and for the faults that will be injected [14]. Unlike this randomized testing approach, which

is fundamentally unscalable, DLTs can be generated either by existing suites of tests present in the computer architecture literature or by using modeling tools, allowing the users to run Musli Tool on test suites and identifying potential system flaws based on test outputs.

4.8.2 Property-Based Testing

A lot of related work exists around property-based testing for correctness as well. Essentially, problematic behaviors of a system can be concisely distilled into small unit tests or into testing programs called checkers. Property based tests focus on stress testing a specific system property whereas checkers can uncover violations of system properties in system execution histories. In distributed systems, the use of checkers is more prevalent not only to identify if violations did occur but to measure and monitor consistency properties of systems in production. Related work focuses on taking traces of operations and running checkers offline against the traces to check for violations of correctness properties [18, 91]. Checkers can be designed for different consistency properties such as linearizability, read-after-write consistency, or per-object sequential consistency. Creating checkers for consistency properties, however, might take significantly more effort than creating property based tests, and checking can have high algorithmic complexity.

Other distributed systems testing tools rely on modeling system properties using happens-before graphs to develop tests that identify bugs in transactions [132]. Property based tests are also used to find bugs in persistent memory systems [89]. Finally, related work leverages modeling of distributed consistency models as a way to generate property-based tests for consistency [126].

My work leverages a family of property based tests called litmus tests. Litmus tests are small, parallel tests consisting of a few operations per thread that have a history of being especially effective over the years for memory consistency testing; a

tool called litmus is widely used for MCM testing [72]. My work introduces a variant of litmus tests, called Distributed Litmus Tests (DLTs), for distributed systems. Apart from being fast to execute and tailored to stress specific update interleavings that are indicative of consistency properties, the large suites of tests used in the memory consistency testing literature can be largely reused, therefore reducing the need for experts developing tests.

4.9 Conclusion

In this chapter we proposed Distributed Litmus Tests, a methodology for testing consistency in distributed systems using property-based tests. The methodologies presented are inspired by tests designed for testing correctness of memory consistency models and adapted to distributed systems. To enable testing on distributed systems using litmus tests, we designed Musli Tool which manages execution and analysis of the tests. We also present a converter tool that allows us to leverage large, existing suites of litmus tests for CPUs and to convert those tests to a distributed systems setting. The measurement study I conducted evaluates both the DLTs and Musli Tool on outcome variety and this chapter presents how the tool and the methodology can be used effectively for consistency testing in distributed systems.

Distributed systems provide the foundations for building a wide range of systems and supporting applications. Other systems rely on distributed systems' correctness guarantees to reason about and operate on shared objects concurrently, and applications leverage distributed systems to perform distributed updates and to function according to user expectations. In Chapter 5, we will look at how smart home environments leverage distributed systems and manage concurrency.

Chapter 5

In Support of Event Ordering and Data Consistency in Smart Home Environments

Typical Internet of Things (IoT) and smart home environments are composed of smart devices that are controlled and orchestrated by applications developed and run in the cloud. Correctness is important for these applications, since they leverage concurrency to control the home's physical security (i.e., door locks) and systems (i.e., HVAC). Unfortunately, many smart home applications and systems exhibit poor security characteristics and insufficient system support. Instead, they force application developers to reason about a combination of complicated scenarios—asynchronous events and distributed devices. Previous chapters proposed related testing approaches for the CPU and distributed systems domains. This chapter uses testing to demonstrate that existing cloud-based smart home platforms provide insufficient support for applications to correctly deal with concurrency and data consistency issues. These weaknesses expose platform vulnerabilities that affect system correctness and security (e.g., a smart lock erroneously left unlocked). To

address this, this dissertation presents OKAPI, an application-level API that provides strict atomicity and event ordering. The work in this chapter is evaluated using the Samsung SmartThings smart home devices, hub, and cloud infrastructure. In addition to identifying shortfalls of cloud-based smart home platforms, design guidelines are proposed to make application developers oblivious of smart home platforms' consistency and concurrency intricacies.

5.1 Introduction

Increasing focus on Internet of Things (IoT) approaches has led to the development of many platforms, systems and applications to enable and connect IoT devices. As IoT systems are used more broadly in an growing number of safety-critical environments, their security and reliability are increasingly important. Unfortunately, current IoT devices and systems do not yet meet these expectations, as can be seen in the frequent news reports about their vulnerabilities, e.g. [65, 69, 90].

This dissertation focuses on smart home environments, although most of the relevant system characteristics are common in other IoT systems that perform monitoring and actuation as well. Figure 5.1 depicts an example of a smart home system. In smart home environments, "smart" embedded devices are wirelessly connected to a hub. Typically, the hub coordinates interaction between devices and acts as a gateway to the Internet, but devices can also be directly connected to the Internet gateway. Via the Internet gateway, the Edge Home Network connects to the cloud, where all the processing for the devices will take place. In most current environments, smart home device state is stored (solely) in the cloud. This includes sensor/actuator transitions of device state and application logic controlling or interacting with these devices. In order to design smart home systems with efficiency and at scale, it is essential to leverage concurrency. However, as prior chapters have shown, relying on

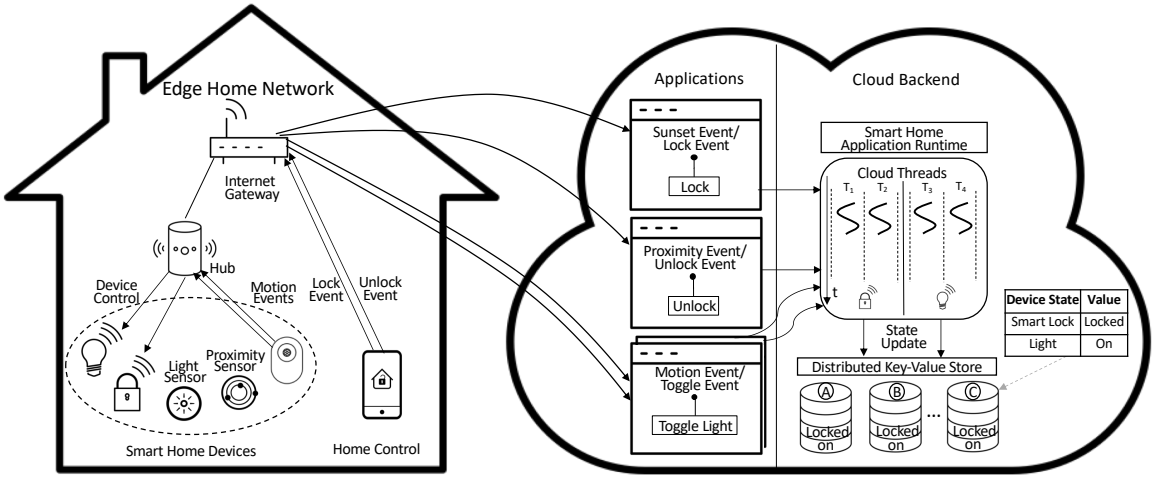


Figure 5.1: A typical smart home architecture. Events are generated in the smart home network by smart home devices and home controllers (e.g. smartphone app). Each event triggers the execution of an application in the cloud, potentially changing device state. Concurrent events are processed by the application runtime and data is persistently stored in a distributed key-value store.

Table 5.1: IoT platform comparison in terms of popularity (Installations), application domain (General/Smart Home), application programmability (Progr.), presence of atomic primitives and consistency guarantees.

IoT Platform	Installations	Generality	Progr.	Atomics	Consistency
Google Home Assistant	100M+	General	Yes	Yes	Configurable
Amazon Alexa	50M+	General	Yes	Yes	Configurable
Apple HomeKit	-	Smart Home	Yes	Yes	Strong(phone)
Samsung SmartThings	5M+	Smart Home	Yes	No	Eventual
Logitech Harmony	1M+	Smart Home	No	N/A	N/A
August Home	500k+	Smart Home	No	N/A	N/A
Vera Home	50k+	Smart Home	Yes	No	Strong

concurrency can in itself be a challenge and provide many opportunities for things to go wrong. For example, Figure 5.1 shows that events for the same devices may be transmitted concurrently, processed concurrently and update cloud state concurrently. This concurrency offers convenience in some ways but also exacerbates correctness challenges regarding atomicity, data consistency, and event ordering. For correct execution of applications that access or modify shared resources, state accesses need to execute atomically and with the correct consistency ordering in order to ensure that state reads and updates reflect the expected and correct values.

These correctness issues can be observed in existing IoT platforms. Table 5.1 presents popular IoT platforms used for home automation, control and orchestration. These platforms are compared based on their programmability, system support and the consistency guarantees they provide. The platforms are presented in decreasing order of popularity; the number of installations of the platforms' mobile applications, as reported by the Google Play Store [57], is used as an estimate of their user base. Alexa [16] and Google Home Assistant [55] are general purpose personal assistants that allow development of applications with any choice of cloud storage systems from Amazon Web Services [17] and Google Cloud Platform [56] respectively, which allow development of applications with both strong and weak consistency guarantees. Out of the smart home platforms presented in Table 5.1, only Apple Homekit [71], Samsung SmartThings [113] and Vera Home [124] allow development of applications. Homekit uses a mobile phone to control smart home devices and stores their state in a common database on the phone, whereas Vera performs home automation using a programmable hub. We found correctness violations in two of these widely used platforms, SmartThings and Vera. In particular, SmartThings lacks primitives to allow applications to perform atomic code execution, and it uses a weak consistency model for its event ordering. Similarly, Vera allows creation of plugins for its programmable hub using scripts that are not thread-safe and cannot guarantee mutual exclusion while accessing smart home devices [125].

As Table 5.1 notes, smart home platform implementations often do not provide application developers with the atomic operations and predictable consistency ordering required for correct concurrent executions. Even when such mechanisms are provided, it is complex for application developers to know when and how to use them. This dissertation proposes the OKAPI system which offers atomicity and strict event ordering as an external add-on functionality in support of correctness. This chapter first focuses on understanding unexpected behavior of smart home applications by studying

and measuring the impact of weak data consistency, lack of atomic execution and out of order event delivery in popular smart home architectures. All observations, experiments and measurements are carried out using a realistic smart home setup. Specifically, analysis and diagnosis of consistency and ordering problems were performed on the Samsung SmartThings platform [113], but the techniques and observations presented here apply to other cloud enabled smart home platforms as well. These empirical measurements and experiences are used to motivate the need for OKAPI. The specific contributions of this chapter are as follows:

- Foremost, this work demonstrates the serious correctness concerns where application logic breaks due to i) use of weak consistency in the cloud storage, ii) race conditions caused by the absence of guarantees for atomic execution on shared resources and iii) insufficient support for in-order event delivery. Smart home applications are measured and analyzed, and our findings are presented. Results from the commercial Samsung SmartThings platform demonstrate that up to 1.4% of accesses to the storage system can return stale values.

- Furthermore, this dissertation proposes OKAPI, a synchronization service that provides atomicity and ordering guarantees to smart home applications. OKAPI can be implemented on a server external to the smart home platform provider. OKAPI is evaluated on the Samsung SmartThings platform by investigating its latency and throughput penalties. OKAPI eliminates ordering and consistency violations while introducing 1.6% average throughput and less than 24.5% latency penalties.

- Finally, this chapter discusses the impact of this approach on smart home applications and proposes guidelines for designing smart home architectures with respect to correctness.

Section 5.2 presents atomicity, ordering and consistency problems in smart home platforms. Building on that earlier discussion, Section 5.3 explains how these problems manifest themselves in the SmartThings platform and presents examples. Section 5.4 presents the OKAPI solution for providing ordering and atomicity. Section 5.5 describes the evaluation methodology. Finally, Section 5.6 presents the evaluation of OKAPI, Section 5.7 discusses related work, and Section 5.8 concludes.

5.2 Ordering, Atomicity and Consistency in Smart Home Systems

5.2.1 Smart Home Architectures

Figure 5.1 presents a typical smart home architecture, split between the Edge Home Network and the Cloud. In the Edge Home Network, "smart" embedded devices that can range from motion and light sensors to smart door locks and smart bulbs are wirelessly connected to a hub. These devices communicate via wireless protocols such as Zigbee [11], ZWave [12] or BLE [10] to a hub that coordinates interaction between devices and acts as a gateway to the Internet. Devices such as smart phones can be connected to the Internet gateway as well, serving as endpoints for the users' home control and orchestration. Any event generated by the smart home devices or the Home Control devices reaches the Internet gateway and is managed by the cloud component of the smart home architecture.

In order to control the behavior of devices in the Edge Home Network and manage the influx of events, a smart home architecture features applications that run on a supporting cloud infrastructure. Smart home applications contain the logic behind home automation and are initiated by external triggers, events that are sent from smart home and home control devices. The types of events and the order in which

they arrive affect the action that smart home applications perform. As most of the smart home architectures are event driven, the runtime system maintains a thread for each event received by an application, called Cloud Threads. The Cloud Threads execute the application logic, access and modify device state in the distributed storage layer and send messages to actuate physical devices within the Edge Home Network.

The distributed storage layer is an important component of cloud-based smart home architectures, responsible for applications' persistent storage as discussed in Section 2.3.4. Persistent storage in stateful applications, as in the case of applications interfacing with smart home devices, refers to any data storage where the data is retained after power to that device is shut off. Typically, a distributed key-value store is in the core of the cloud storage layer. Key-value stores allow representation of data in key-value pairs $< key, value >$, allowing retrieval and storage of data from these data structures.

5.2.2 Event Ordering

Event ordering issues are introduced here, as events can be sent to Cloud Threads concurrently, creating the need for event ordering to ensure correctness. If we revisit Figure 5.1, we observe multiple events generated by the motion sensor and the smartphone responsible for home control. Although events are generated by devices connected to the same network, the edge home network and many current IoT services do not offer strong guarantees regarding their ordering. In this case, multiple events generated by a single device can be *reordered* at multiple locations before arriving at the cloud service. Messages can be reordered by the wireless protocol connecting smart home devices with the smart home hub and also as they travel from the home's gateway to the cloud using transport protocols. In the example in Figure 5.1, both lock and unlock events are generated by the same phone. A possible reordering however can change the order of lock and unlock events. In that case, the device's view of

state will become inconsistent with the state intended by the user's requests, and the door will be erroneously left in a state contrary to the user's expectation. After the events arrive at the cloud platform, there are additional potential locations for reordering, due to arbitrary delays in accesses and updates in cloud storage as well as in the processing of events. Reordering can result from the fact that events can arrive out of order at the hub or home gateway within the edge home network or at the cloud and there is no mechanism to guarantee order of arrival identical with the order at the origin.

5.2.3 Atomicity Violations

Apart from event ordering, concurrent execution of Cloud Threads operating on shared data can affect correctness due to the potential presence of race conditions. Figure 5.2 depicts a scenario of a motion sensing application in a smart home system that demonstrates the need for atomic operations. Atomic operations are operations in a concurrent program that execute completely isolated by any other processes. In Figure 5.2, two different events are created by a motion sensor due to physical triggers (labels 1 and 2 in Figure 5.2) and reach a smart home application in the cloud via the edge home network (labels 3 and 4). The motion events are received by an event handler in an application that toggles a smart bulb, changing its state from on to off and vice versa. The state of the physical smart light bulb is a state variable that is stored in the key-value store in the cloud. In order to execute the application logic and perform accesses and modifications to the light bulb state, the smart home system runtime spawns a cloud thread per event. For the toggle operation, each cloud thread will access the current light bulb state in the key-value store (labels 5 and 6), flip its value and perform an update to the cloud store (labels 7 and 8). If the executions of both cloud threads corresponding to the motion events overlap and occur non-atomically, however, both cloud threads will read the initial state of the smart bulb instead of the

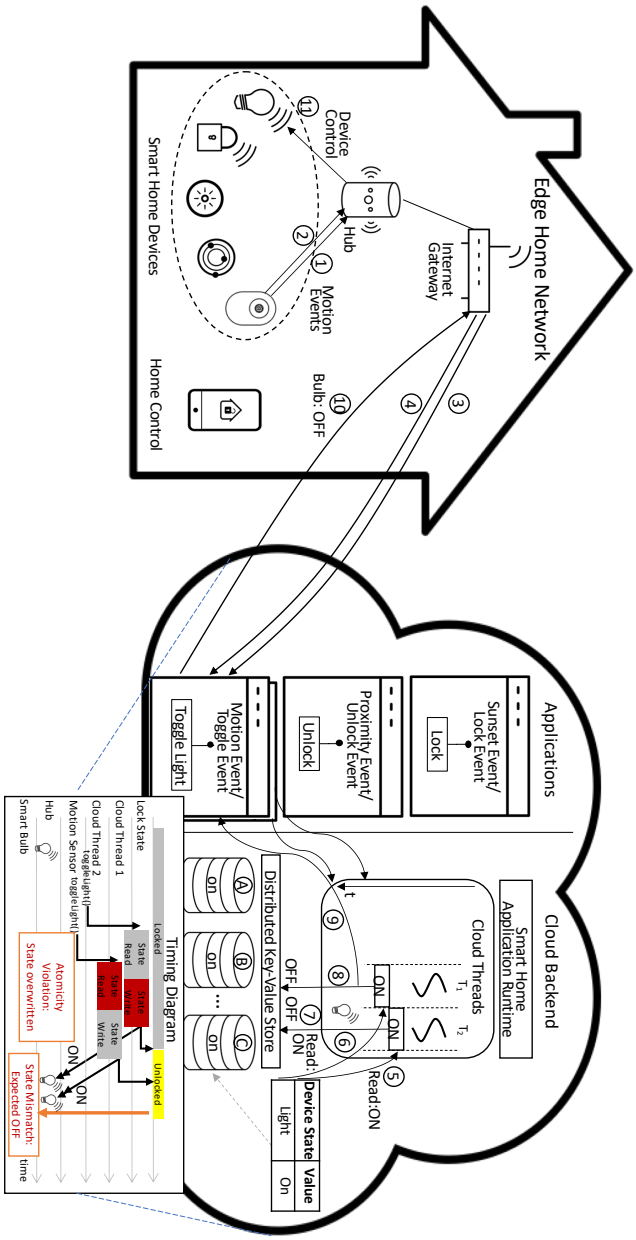


Figure 5.2: Atomicity violation scenario in a smart home system. In some smart home environments, multiple events can perform writes to shared state unrestrictedly. As such, the lack of read-modify-write atomicity leads to incorrect or non-intuitive results.

second cloud thread viewing the state as altered by the first. The timeline of these accesses is presented by the timing diagram in Figure 5.2. The simultaneous accesses to cloud state create a race condition and will cause both cloud threads to read the

state of the light bulb as ON and set it to OFF (labels 9,10 and 11) which is counter to expectations.

Although often challenging, application developers can reason about and enforce atomicity while developing their applications. However, some smart home platform providers have designed their runtime systems in ways that do not allow use of atomic primitives by smart home applications due to cost considerations; when cloud threads do not restrict concurrent accesses to shared state, cloud thread execution times will remain shorter, which translates to lower resource usage and consequently cost reduction. This example, however, indicates that having no mechanism to guarantee atomicity leads to correctness violations, which are critical in smart homes as the operation and physical security of users' homes might be compromised.

5.2.4 Weakly Consistent Cloud Storage

For cloud platforms and services supporting IoT and smart home applications, weaker consistency models like eventual consistency, presented in detail in Section 2.3, are commonly used to optimize for performance and trade off consistency for availability [114]. Figure 5.3 presents a scenario of a proximity event and a sunset event (light sensor) and (labels 1 and 2 in Figure 5.3) that are handled by two different applications (labels 3 and 4), the first one causing a smart door lock to unlock and the second one to lock respectively. For each of these events, there are corresponding cloud threads that perform an update to an eventually consistent cloud storage, in the order of their arrival. An eventually consistent cloud storage system means that the updates will be eventually applied in the same order to all replicas, but there is no guarantee as to when the updates will be visible throughout the system. In the example, the first update writes Unlocked into replica B (labels 5 and 7) and subsequently, the second update writes Locked into replica C (labels 6 and 8). In practice, that means that a read of the value of the Smart Lock state can return Unlocked, even if the more

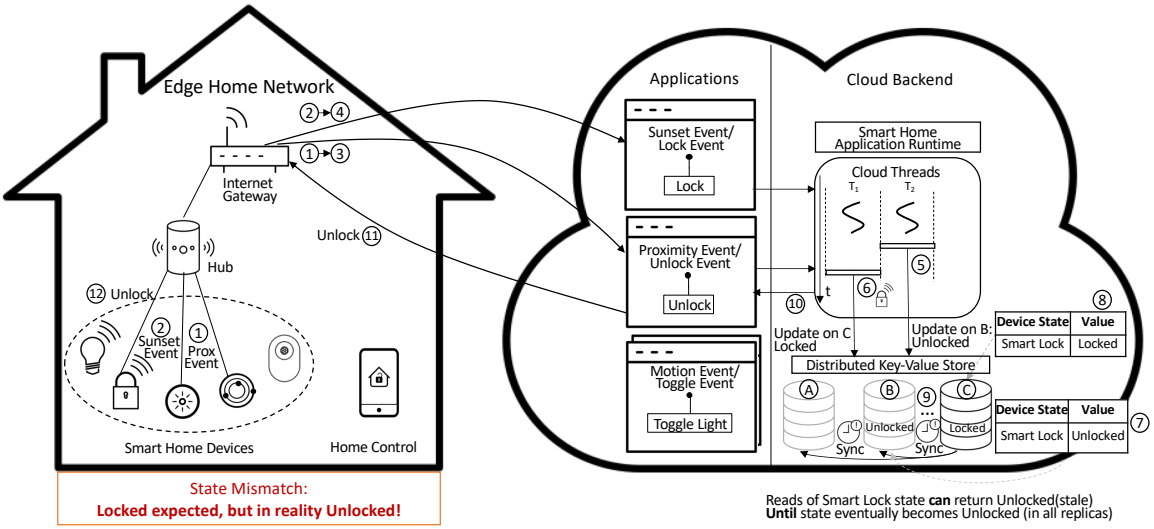


Figure 5.3: Consistency violation scenario in a smart home system. Multiple events are received and processed by cloud applications, designed to control the state of a smart door lock, performing updates to cloud storage in the process. Due to weak consistency, stale values can be returned by the key-value store and cause unexpected application behavior.

recent update set the value to Locked. This result can only happen during the window when the Locked update is not yet visible across all replicas (label 9). During that window, any application that uses the lock state to control smart home devices can access a stale lock state from one of the replicas for which the latest update is not yet visible. In this scenario, the application will Unlock the smart lock as this is the stale value returned by a read after the updates to the key-value store (labels 10,11 and 12). Due to the staleness of the data, the application's control of the physical lock can be counter to the user's expectation.

5.3 Testing Real-world Smart Home Systems

Figures 5.2 and 5.3 offer simple scenarios where network ordering, atomicity and weak consistency affect application correctness. Despite the seemingly naive aspects to these scenarios, this work indicates they actually occur in real-world systems. In this section, it is demonstrated how these scenarios can appear in the context of smart

home systems and other platforms that build on distributed storage. This chapter uses the SmartThings smart home platform as a case study and presents measurements that provide concrete evidence of these behaviors, along with the testing methodology.

5.3.1 SmartThings Smart Home Architecture

The SmartThings home automation platform allows development of applications in the cloud, called SmartApps, that interact with devices in the user's home. The SmartThings platform follows the smart home architecture described previously; the rest of this subsection presents how network reordering, atomicity and consistency violations appear on SmartThings.

There are two ways of persisting application state in the SmartThings cloud platform, *state* and *atomicState*. These storage mechanisms have different expectations about when their modifications are propagated to persistent storage in the cloud backend.

To understand updates to persistent data, one must understand the SmartThings execution model. SmartApps are not continuously running in the cloud. Their execution is triggered when an event arrives at the cloud (initiated by either a physical device or a mobile application), resulting in an event handler in the SmartApp executing on a cloud server. Multiple instances of the same SmartApp may execute simultaneously and independently in the cloud. This design means that when event handlers execute, their only context is the persistent state stored in the backend and that multiple event handlers may be accessing that state at the same time.

Given the potential overlap of reads and writes from different event handlers, it is difficult for programmers to develop correctly functioning applications without understanding the consistency guarantees of a smart home platform. The cloud storage component of SmartThings is built with Cassandra [51], which, as described in Chapter 3, provides weak consistency guarantees to maximize availability. For the two

types of SmartThings state, *state* and *atomicState*, there are different expectations for when updates are sent to persistent storage. When an event handler executes, modifications made to *state* objects are only updated in persistent storage *after* the event handler's execution completes. In contrast, modifications to *atomicState* are reflected in the cloud backend "more or less immediately" [114], meaning updates to persistent storage do not wait for the event handler's execution to complete.

While the name *atomicState* implies atomicity can be achieved, this is not how it is actually implemented. As the SmartThings platform does not provide any form of atomic read-write-modify capabilities, *it is not possible to achieve atomicity.* As a consequence, two simultaneous executions of event handlers accessing the same state (*state* or *atomicState*) can result in classic race conditions; both event handlers may read the state before either updates the state and their subsequent serialized updates result in the first update performed on that state being obliterated by the second event handler's update to that same state. The previous scenario demonstrates why the *atomicState* primitive constitutes a bad design choice. If race conditions are to be prevented, another mechanism must be created to provide atomic updates to the state.

5.3.2 Experimentation on Smart Home Deployments

In order to validate the existence of network reordering, atomicity and consistency violations in smart home systems, tests are designed for the respective properties that testing targets, as presented in Table 5.2. These tests follow the concept of litmus tests, but the only test in Table 5.2c is designed to test for consistency properties by checking the shared variable for a stale value. The test in Table 5.2a checks if the concurrent operations of two Cloud Threads on a shared object x result in a race condition and the test in Table 5.2b checks if events generated by the same client are reordered in the network. For all the tests, there is a single device that is generating

(a) Atomicity Test

Cloud Thread 1	Cloud Thread 2	Result	Outcome
x=x+1	x=x+1	$x_1 = 1, x_2 = 2$	OK
write(x)	write(x)	$x_1 = 2, x_2 = 1$	OK
read(x)	read(x)	$x_1 = 1, x_2 = 1$	Race condition

(b) Network Reordering Test

Cloud Thread 1	Cloud Thread 2	Result	Outcome
write(x=1)	write(x=2)	$x_1 = 1, x_2 = 2$	OK
read(x)	read(x)	$x_1 = 2, x_2 = 1$	Reordering

(c) Weak Consistency Test

Cloud Thread	Result	Outcome
write(x=1)	$x = 1$	OK
read(x)	$x = 0$	Stale value (initially $x = 0$)

Table 5.2: Tests for detecting atomicity violations, network reordering and weak consistency.

the events that trigger the executions of the cloud threads. (See Section 5.5 for details on the experimental setup.) All tests are non-deterministic due to concurrency and therefore multiple trials are performed.

For the atomicity test in Table 5.2a, a shared persistent variable x is incremented every time an event is handled. Since the cloud storage does not successfully implement mutual exclusion and executions of event handlers can overlap, concurrent executions can lead to values being erroneously overwritten due to race conditions. When both cloud threads read the value $x = 1$, this is an indication of an atomicity violation that leads to a race condition.

Requests generated by a device can be generated in order but can be handled out of order by the application in the cloud. The network ordering test in Table 5.2b identifies this condition, when the cloud threads read the values in a different order than the order the corresponding events were generated.

Finally, this chapter tests the impact of a weakly consistent key-value store on the application in Table 5.2c. However, it is impossible to test the impact of weak consistency in the absence of an atomicity mechanism, as it is not possible to distinguish

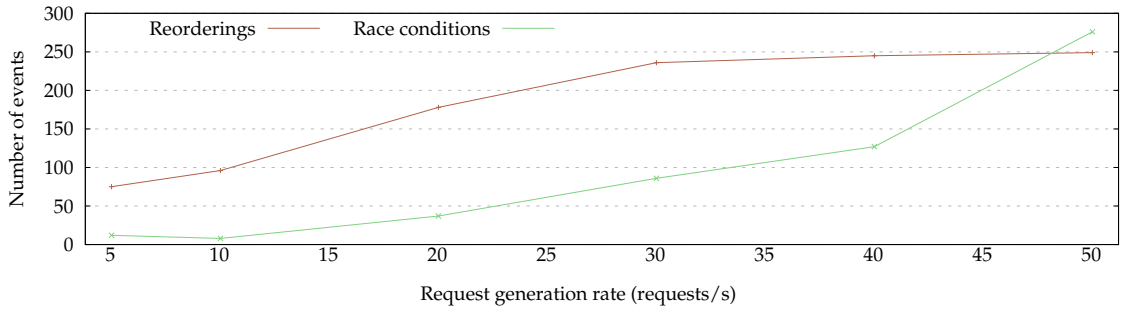


Figure 5.4: Number of occurrences of event reorderings and race conditions. A test application executes the Atomicity and Network Reordering tests respectively using shared persistent variables. Results are presented in increasing request generation rate and out of a total of 1000 events. Lower is better.

consistency violations from atomicity violations. For this reason, the weak consistency test is performed on a standalone Cassandra key-value store deployment, the same key-value store that the SmartThings platform uses. This test aims to identify if a value is stale, meaning if a read returns previous state (x initialized to 0) after a more recent update has set the value. That is possible when a write update is performed on a different replica than the one the read accesses.

Event reordering and atomicity violation statistics are presented for the tests on the SmartThings platform. As Figure 5.4 shows, the measurements indicate that event reorderings and race conditions are present and that they increase as the event generation rate on the device increases. For the consistency test, stale values are measured at a frequency of 1.4% over the number of the tests in a Cassandra deployment. Without improved support for the ordering and atomicity with which state is observed and updated and without storage systems with stronger guarantees to build upon, even simple smart home systems will suffer from severe correctness and concurrency problems.

5.4 OKAPI: API for Restoring Smart Home Consistency and Atomicity

To avoid the erroneous scenarios discussed earlier and to ensure correctness in smart home applications, support for atomic operations and a mechanism for preventing event reordering is needed. However, cloud platforms like SmartThings are often either proprietary or restrictive in terms of platform modifications; application developers are unable to add the required functionality to the cloud backend. Therefore, this thesis introduces OKAPI, a synchronization service that creates consistency and atomicity mechanisms for use in Smart Home applications. Developers can use OKAPI to create the guarantees not provided by the platform's cloud backend but needed by their applications.

OKAPI's design consists of two components: an OKAPI server responsible for guaranteeing atomicity and ordering and an OKAPI synchronization layer that initiates a two-phase communication protocol with the OKAPI server on the application side. Applications use the OKAPI synchronization layer to send requests to the OKAPI server. The OKAPI server then disallows concurrent writes to shared state stored in the cloud, effectively serializing access to critical sections of code from concurrent event handlers.

OKAPI's design is flexible and allows adoption across various scenarios with the purpose of restoring consistency and atomicity in the smart home. When the platform does provide atomicity and ordering guarantees, consumers can still choose to deploy OKAPI in case an application does not use platform-available concurrency features correctly. OKAPI may be deployed in different ways. If a user's platform allows programmable hubs, the OKAPI server could be implemented on the hub. Alternatively, OKAPI can be hosted as a synchronization service on a third party server. Due to privacy concerns about third party servers and given the rise of personal

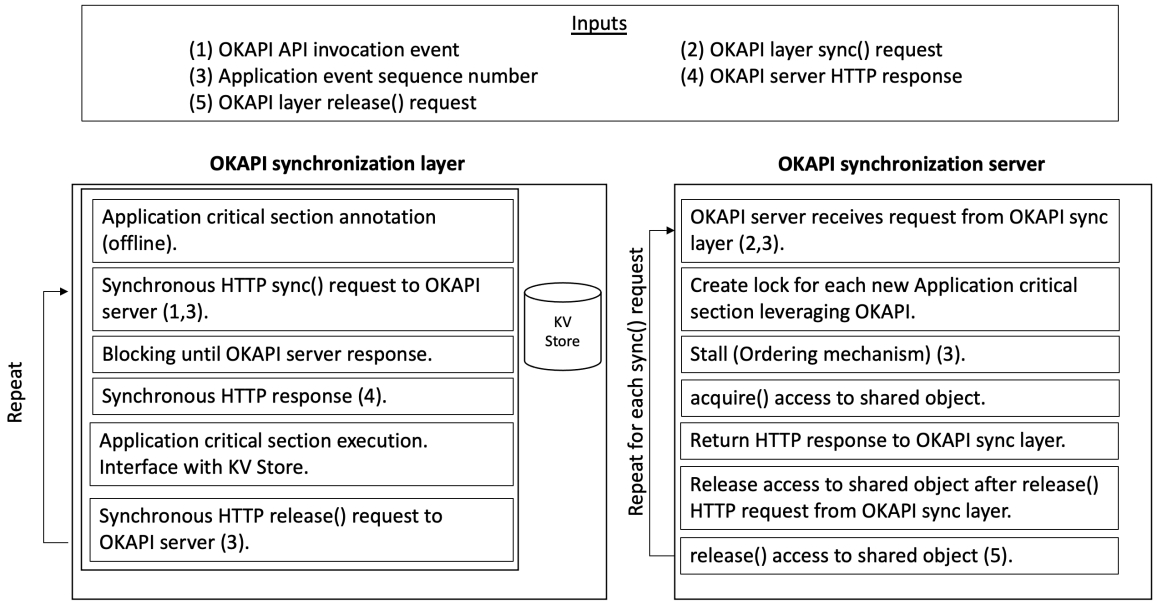


Figure 5.5: The OKAPI architecture and functionality of OKAPI Synchronization Layer and OKAPI Server. Numbers indicate the inputs necessary to perform each step.

cloud systems, individuals could also potentially deploy OKAPI in their personal cloud infrastructure.

5.4.1 OKAPI Usage

OKAPI relies on two mechanisms to provide atomicity and ordering functionality, the OKAPI Server and the OKAPI Synchronization Layer. To invoke these, applications annotate their critical sections accordingly and OKAPI does the enforcement. The architecture of the OKAPI Server and the OKAPI Synchronization Layer is presented in Figure 5.5.

OKAPI Server: The OKAPI synchronization server is used to implement atomic locks and their corresponding acquire and release operations. For each block of code within a SmartApp that requires mutual exclusion, the server maintains a lock. The OKAPI server receives and handles external lock acquire and release requests from SmartApps trying to access a critical section in the code; for example, a SmartApp performs updates to shared *atomicState* in the cloud backend. Also, in such a scenario

where the application needs to perform atomic updates, a mechanism is required in order to block an executing event handler while it waits to be granted sole access to the critical section by the synchronization server.

OKAPI Synchronization Layer: The OKAPI Synchronization layer provides Smart Home applications with an interface to utilize the atomicity and ordering features the OKAPI Server provides. Mutual exclusion is achieved by sending lock acquire and release requests to the remote synchronization service via synchronous HTTP requests. A synchronous HTTP request blocks the execution of the event handler issuing the request until a response to the HTTP request is received. In this way, concurrent execution of event handlers is prevented, enabling mutual exclusion of event handlers accessing shared state in the application's critical section. When the response is received from the OKAPI server, the event handler's execution can proceed, knowing it currently has sole access to the shared *atomicState*. When the event handler finishes accessing the shared state, it sends another synchronous HTTP request to the remote server releasing the lock to the shared data. The event handler then waits until a response is received from the remote server to guarantee release of the critical section. For the Synchronization layer protocol to be correct, updates to the shared state need to be propagated to persistent storage before the release of the lock at the remote server. In SmartThings, an update to *atomicState* needs to complete before the lock is released from the OKAPI server, as code outside the critical section could otherwise execute concurrently. Tests across thousands of events did not reveal any updates to *atomicState* that happened out of the order of execution specified by the application or before the release of the lock at the OKAPI server. Based on the experimentation performed, all *atomicState* updates are regarded as synchronous in the application logic.

Timeout Handling and Deadlock Prevention: If no response is returned within a specified time limit for each synchronous HTTP request, the HTTP request times

out. The HTTP request timeout for the SmartThings platform is 10s. If a timeout occurs, an additional synchronous HTTP request is generated by the application to gain access to the critical section. During the execution, the OKAPI server can potentially run into a deadlock, for example when a remote device gets a lock but does not explicitly unlock it due to the loss of its network connection. Since the synchronization protocol is based on HTTP, the OKAPI server uses the 10s HTTP timeout value as the expiration for exclusive access to the critical section. In addition to HTTP timeouts, the SmartThings applications' execution time is limited to 20 seconds. This limit is imposed by the SmartThings Application Runtime in order to optimize allocation of resources; OKAPI is not limited by the application execution time.

5.4.2 Enforcing Atomicity

For atomicity, Figure 5.6 sums up OKAPI's two-phase communication between Smart Home applications and the remote synchronization server along with sequence numbers for the order of requests and operations. In this example, one phone generates two requests corresponding to the unlock() (label 1) and lock() (label 2) methods for a smart lock. The unlock() event handler in Execution 1 sends a synchronous HTTP request to the remote server (sync(), label 3) requesting the lock for its shared data to be acquired. The remote server executes its acquire() method for the lock (label 5) and sends a response to Execution 1 indicating it has access to the shared data and can proceed. Execution 1 is able to read and write the shared data, with those updates being sent to the cloud backend (label 7), and send a lock message to the physical device (label 9). Execution 1 concurrently sends a synchronous HTTP request releasing the lock which causes the remote server to release the lock (label 8). While Execution 1 has the lock, a second synchronous HTTP request is sent to the remote synchronization server by the lock() event handler (label 4), but the lock() event

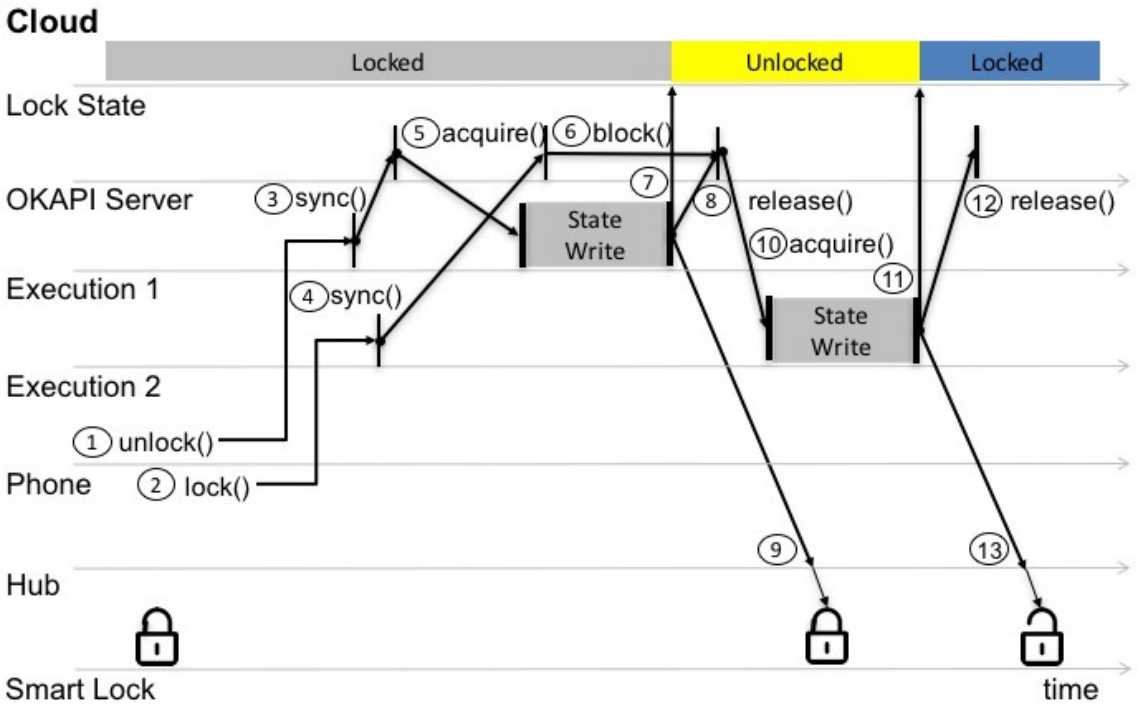


Figure 5.6: OKAPI two-phase protocol enforcing atomicity.

handler is blocked (label 6) until it receives a response to its request (label 10) after Execution 1's lock release is complete. Then Execution 2 has the lock, writes the shared data to the cloud backend (label 11), a release() request (label 12) is performed to the server while an unlock message is sent to the physical lock (label 13).

5.4.3 Ordering

Overall, OKAPI provides *Per-Object Sequential Consistency* [84, 91] at the application level. According to this guarantee, for each shared state object (e.g., shared Smart Lock state), there exists a legal, total order over all events which is reflected in each clients order.

Ordering is achieved by building upon the atomicity enforcement in the OKAPI synchronization server. The ordering mechanism is implemented in the remote server and provides a serial order of events per client and a total order of events per shared state object. In practice, OKAPI allows a single copy of an object to progress over

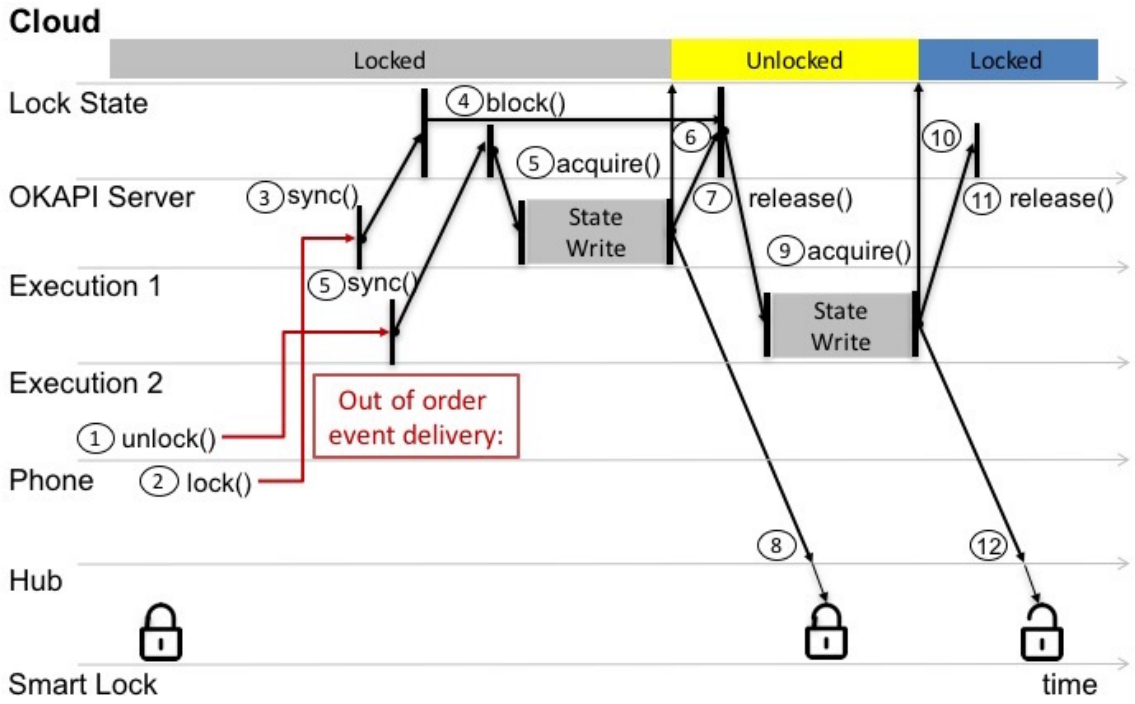


Figure 5.7: OKAPI two-phase protocol enforcing atomicity and in-order event delivery in Smart Home applications.

time. Clients will always see a newer value of shared state as they interact with it, but these values can be different across clients.

To provide ordering guarantees, each client provides metadata in the form of a sequence number embedded in each request which allows the remote server to order and to serialize these requests. A sequencing scheme is introduced in the events received by the application to achieve ordering of events per client. Sequencing of events can also be achieved by modifying the smart home platform or hub, if the implementations are open source. Figure 5.7 summarizes OKAPI's enforcement of both atomicity and ordering. For one of the two requests that arrives out of order at the Smart Home application, the synchronization server "stalls" its acquisition of the lock (i.e., execution of its acquire() method at the remote server seen at label 4) until the sequence number of said request matches the next sequence number allowed to proceed. Once that *acquire* operation is complete, the protocol proceeds as in Figure 5.6.

5.5 Evaluation Methodology

Experimental setup—Measurements: To concretely study event reorderings in Smart Home environments, components are introduced into the experimental infrastructure for the client and physical device that enable automation for event triggering and for the collection of information about messages sent by the cloud backend.

Typically, a mobile application client sends a message to the cloud to trigger an event. To remove user involvement and automate experiments, the experimental infrastructure exploits part of the SmartThings API that creates an endpoint that is accessible over the web. HTTP requests sent to these endpoints trigger event handler executions just like messages sent from a mobile application to the cloud. A computer generates endpoint requests in order to automate event triggering and therefore simulate a client. Event triggering and request generation is performed similarly for the consistency experiment, interfacing with a 3-node Cassandra cluster instead of the SmartThings platform.

Since the cloud backend and hub are proprietary in SmartThings, instrumentation cannot be added to these parts of the system, and they must be used as black boxes. One of the contribution of this dissertation is demonstrating that OKAPI can overlay needed atomicity and ordering functionality over an IoT cloud platform that does not offer these capabilities.

To track message order, messages sent by the client are tagged with sequence numbers that propagate across the system. The cloud generates sequence numbers for each message it receives and sends a reply message to the client. The OKAPI synchronization service is a Go HTTP server which responds to requests providing access to the critical section of the Smart Home application. The HTTP server handles HTTP requests concurrently and is able to provide atomicity locally using mutexes. Given atomicity, one of the concurrent HTTP requests that has not yet been processed

gets hold of the mutex and participates in the two-phase protocol with the Smart Home application. The remaining concurrent requests must block, waiting to get hold of the mutex.

OKAPI runs on a Ubuntu 14.04 LTS virtual machine, with 1 CPU Core, 30 GB SSD storage and 2GB memory and is hosted on Linode [88]. Utilizing this OKAPI synchronization server for the evaluation incurs an additional latency, corresponding to the round trip time between the SmartApps and the OKAPI server. This additional latency is reflected in the results. OKAPI is easily portable to any server or cloud infrastructure that enables OKAPI to interface with a storage system and function as a synchronization service to applications in order to enforce correctness.

Latency & Throughput Evaluation: The OKAPI request latency is evaluated by measuring from the time a request was generated by the user application until the Smart Home application responds. In addition, the throughput is measured as events processed per second. For testing purposes, a client sends requests to the Smart Home application at a controlled rate. The request generation rate is varied to test the overhead of OKAPI under both low and high contention for the critical section. The code of the critical section remains constant and short-lived across experiments and consists of test code as presented in Table 5.2. An unmodified version of a test application (denoted as Unmodified) is compared with a version that enforces atomicity (denoted as Atomic) and a version that enforces both atomicity and ordering (denoted as Atomic + Ordering).

5.6 Evaluation

This section evaluates the performance of OKAPI's (i) Atomic and (ii) Atomic + Ordering approaches versus the Unmodified approach. OKAPI is evaluated in terms of the latency and throughput overhead.

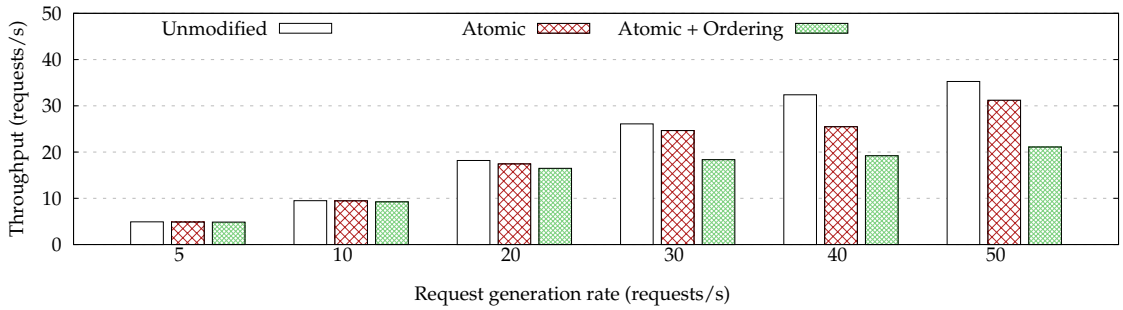


Figure 5.8: Average throughput of the test SmartThings application across three methods. Compared are the (i) Unmodified (ii) Atomic and (iii) Atomic + Ordering approaches. Results are presented in increasing request generation rates. Higher throughput is better.

Throughput overhead evaluation: Figure 5.8 presents the throughput in requests/s for a single client simulating a mobile application across different rates of incoming requests. The lost throughput for request generation rates between 5-10 requests/s is 0.4%-0.7% and 0.8-1.6% for the Atomic and the Atomic + Ordering respectively. When the number of requests generated per second increases, the number of executions concurrently performing *acquire* operations to obtain access to the critical section increases as well. Reorderings cause the OKAPI synchronization server to delay responses to out of order requests, thus reducing throughput. The lost throughput of OKAPI is 11.5% for the Atomic approach and 40.2% for the Atomic + Ordering approach compared to Unmodified.

Latency overhead evaluation: Figure 5.9 presents the completion latency (ms) for requests generated by a single client across different request generation rates. Both OKAPI approaches use an external synchronization service, which adds one more hop in the network and increases the request completion latency. The latency overhead for the Atomic and Atomic + Ordering approaches increases when the request generation rate increases, demonstrating a linear behavior for rates below 30 requests/s and remaining below 21.0% and 64.1% respectively. The communication latency between the SmartApp and the OKAPI server is included in these results.

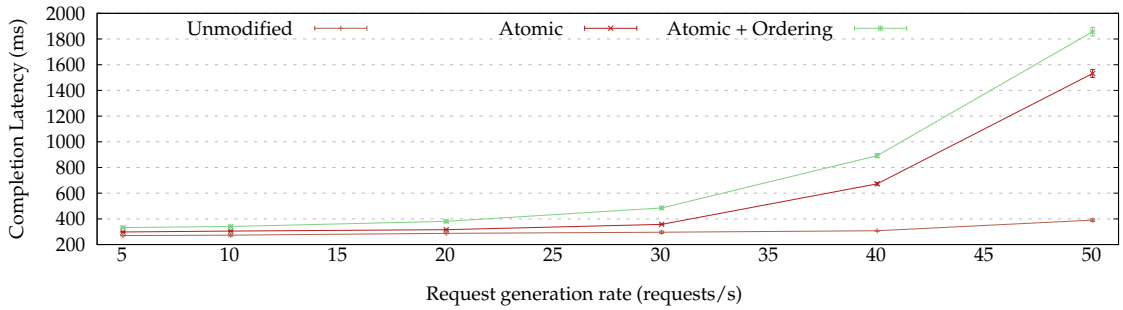


Figure 5.9: Average request completion latency of the test SmartThings application across three methods. Compare are the (i) Unmodified (ii) Atomic and (iii) Atomic + Ordering approaches. Results are presented in increasing request generation rates. Lower latency is better.

Beyond the 30 requests/s generation rate, the latencies increase significantly and application developers need to account for the responsiveness of their smart home applications. Throughout these experiments, there are no timeouts for the request generation rates evaluated. However, when the request generation rate was increased to 100 requests/s in order to stress test the Atomic + Ordering approach, 65.6% of requests time out causing the average latency to increase significantly.

Discussion: Typical smart home system workloads don't typically exceed 2 requests/s [77], even while considering scenarios with multiple users performing multiple requests. In addition, latency values between 300-600ms are typical from the moment the event is triggered until an event notification reaches the application. To stress the system, the evaluation is performed using request rates higher than 5 requests/s (300 requests/minute). The corresponding average latency for the request rates 5 requests/s and 10 requests/s is 298 ms and 305 ms for the Atomic and 332 ms and 340 ms for the Atomic + Ordering version of OKAPI respectively. Under these conditions, OKAPI enforces atomicity and ordering with minimal impact on throughput and with latency that is within typical values for smart home systems.

5.7 Related Work

Prior work has explored event-driven architectures, focusing on exposing race conditions [39], as well as security deficiencies [38]. While this chapter focuses on event-driven platforms, this approach concentrates on atomicity and ordering and considers the whole Smart Home environment, including hubs, devices and the cloud platforms. Related work [45, 46] has investigated misuse of smart home application privileges and has created solutions for data protection and access control. Ensuring the correct execution of the individual devices as well as the interaction of these devices requires checking that applications are written correctly. In this work, the focus is on the atomicity and ordering problems that arise from these platforms.

Detailed study of security mechanisms and tools that analyze applications' permissions can help in statically analyzing Smart Home application code and identify and pinpoint code blocks with atomicity and ordering issues. [70] investigates the security of smart locks, presents attacks against them and performs a security analysis of consumer products. The authors build a solution to mitigate the vulnerabilities found and suggest the use of eventual consistency for systems that use smart locks as it offers a nice intersection between availability and consistency. OKAPI focuses on providing ordering of events and atomicity in Smart Home application execution, concluding that stronger guarantees are required from a consistency standpoint.

In another line of work, [94, 102] investigate the use of happens-before graphs, adding new constraints that model which operations can occur before other actions. By creating these graphs, it is possible to detect what new ordering constraints are necessary to prevent these race conditions. The use of happens-before graphs has also been used by [20] to formalize the concurrency semantics of the Android programming model and to perform race detection on real Android applications. In [47], a systematic exploration of kernel concurrency bugs is performed using schedule exploration. In addition the authors of [92] perform a study of concurrency bug characteristics in

real systems. The authors of [48] provide a bug finding analysis on formally verified distributed systems, and [75] leverages model checking to discover errors in particular schedules of events. Such techniques could help detect whether reads or writes to a distributed platform could trigger possible race conditions and facilitate a more fine grained use of OKAPI.

This dissertation presents OKAPI, a framework that provides ordering and atomicity guarantees to Smart Home applications. OKAPI is platform independent and provides strong consistency and ordering guarantees, even if cloud platforms hosting Smart Home applications have more relaxed guarantees. The framework utilizes an external synchronization service and a two phase protocol for communication between the Smart Home applications and the remote service to provide atomic execution and in order event processing.

5.8 Conclusion

This chapter demonstrated that current commercial cloud-based smart home platforms do not provide applications with system support to guarantee atomic application execution and elimination of event reordering and data staleness. This chapter presents experiments that validate that events can be reordered at a high rate of 5-25% and reading of shared objects can return stale values in 1.4% of cases in smart home applications. To address these weaknesses, this chapter proposed and evaluated OKAPI, a platform-independent synchronization service that allows insertion of atomicity and ordering guarantees into smart home applications. Consistency deficiencies, event reorderings and race conditions are challenges for commercial smart home platforms and any system that leverages concurrency. OKAPI offers a methodology towards improving correctness guarantees in smart home applications. OKAPI can be used as a service to provide atomicity and ordering for Smart Home

applications without depending on cloud providers' design decisions with latency and throughput overheads that do not hinder applications' functionality.

Chapter 6

Conclusions

This chapter first gives an overview of the research presented in this dissertation. It then discusses possible future directions that can build on the advances in this dissertation and concludes.

6.1 Dissertation Summary

6.1.1 Memory Model Testing

This dissertation introduces the concept of perpetual litmus tests, a variant of litmus tests without per-iteration synchronization, and PerpLE, a tool suite capable of generating, executing and analyzing perpetual litmus tests. The impact and opportunity of not having per-iteration synchronization in consistency testing is tremendous. The lack of per-iteration synchronization allows each thread to run independently of the others that are concurrently tested, thus speeding up execution and enabling a wider variety of interleavings to appear more frequently. Perpetual litmus tests are a consistency testing methodology that is distributed and does not require any coordination between the threads running on different cores. This work can save thousands of hours in validation and verification, driven by 4 orders of magnitude higher efficiency and

9x speedup. Such improvement significantly reduces the cost of the validation and verification processes and consequently improves time-to-market. In Chapter 3, we describe how the perpetual litmus test methodology is applicable in consistency testing, and also demonstrate how PerpLE is a practical tool for empirical testing.

6.1.2 Distributed Litmus Tests

This dissertation also presents Distributed Litmus Tests, an adaptation of litmus tests to distributed systems. Distributed Litmus Tests adapt litmus tests to distributed key value stores' and distributed systems' intricacies and take lessons learned from the architecture community to enable an efficient and practical testing methodology for distributed systems' consistency properties. To enable Distributed Litmus Tests, this thesis introduces Musli Tool, a testing tool that allows test generation, execution and analysis of Distributed Litmus Tests and enables testing across a range of execution parameters. Chapter 4 presents a measurement study that demonstrates the value of litmus tests as a testing methodology in the domain of distributed systems and evaluates Musli Tool in practice.

6.1.3 Concurrency in Smart Home Systems

This thesis also provides direct insight into commercial smart home systems and their design considerations and implementation in regards to correctness. Chapter 5 identifies a series of correctness inefficiencies that lead to applications behaving counter to user expectation due to insufficient system support. An analysis and a measurement study are presented for on Samsung SmartThings, with findings indicating correctness becoming compromised at the level of the application runtime and its supporting storage system. To mitigate the insufficient correctness guarantees, this dissertation proposes OKAPI, a library that provides correctness support independently of the smart home system architecture.

6.2 Future Directions

There are many exciting opportunities to follow up on the work presented in this dissertation in the future. This section summarizes some of these ideas.

6.2.1 MCM Testing Beyond CPUs

With both software and hardware validation, verification and testing becoming prevalent in the systems community, my dissertation is in a position to inspire future research. Creating tools and methodologies to validate the correctness of hardware, as well as the need to optimize the performance and efficiency of these tools, is becoming increasingly important as system complexity rises. The concept of perpetual litmus tests is based on the idea of removing synchronization and of adding sequences to be able to track the origin of memory operations. There is an opportunity for the architecture community to improve on the benefits showcased by PerpLE by applying perpetual litmus tests to other hardware domains and in order to increase test coverage. The value of perpetual tests and PerpLE could be highlighted further by using perpetual litmus tests extensively in both academia and industry as a bug finding tool. This thesis evaluated PerpLE on x86-TSO which did not highlight the distributed character of the testing methodology. However, the unrealized potential of perpetual litmus tests is to apply the methodology to systems implementing weaker consistency models such RISC-V, GPUs and other heterogeneous parallel hardware. For example, perpetual litmus tests could be applied to hardware implementing newer architectures and weaker consistency guarantees with the potential of identifying erroneous behaviors.

PerpLE has the potential to be used in combination with existing tools such as litmus and to improve upon them significantly across a variety of metrics. I envision current testing tools used in consistency testing, especially ones that leverage litmus

tests like the diy suite [72], to adapt to testing using perpetual litmus tests or to be integrated with PerpLE.

6.2.2 Synchronization-free Testing Across the Stack

This thesis makes the observation that analyzing correctness in concurrent systems boils down to a search for problematic event orderings and interleavings of operations and updates that could appear in practice during program executions on a system under test. Based on this observation, I expect insights drawn from this dissertation to translate well to other aspects of concurrency testing. This dissertation is a prime example of adapting lessons learned from testing correctness in microprocessors to IoT and distributed systems. Testing the correctness of concurrent programs is a process that requires looking for erroneous orderings across the system stack [121]. It is possible that there exist more opportunities across the Memory Consistency Model stack to apply perpetual litmus tests and develop synchronization-free testing methodologies, such as for the hardware OS interface. I believe that researching efficient synchronization free testing methodologies and advancing perpetual litmus tests will greatly benefit the performance and efficiency of correctness testing.

6.2.3 Consistency Modeling Tools

Automatic verification techniques using formal methods typically suffer from scalability issues due to state explosion (e.g., model checkers) and incur a high cost in terms of time and resources. Empirical testing can provide solutions that can effectively reduce the need for formal verification or shift towards a more targeted use of these techniques. However, for property based techniques like litmus tests to be successful, it is necessary to test using suites of tests tailored to stress desired correctness properties. Existing work that addresses litmus test generation as a means of comparing memory models or for the empirical conformance testing of new implementations of architectures

[44, 64, 96, 129] will need to adapt to generate and support perpetual litmus tests. Regarding Musli Tool, it would be very useful to develop modeling tools for distributed systems to automatically infer if an outcome is allowed or forbidden for a particular consistency property and also to generate litmus tests tailored to provide coverage for a particular property.

6.2.4 Exploration of Stress Testing and Fault Injection

Musli Tool could become more practical as a testing tool by exploring a wider variety of execution parameters and by introducing additional techniques for stress testing and fault injection. Distributed Litmus Tests allow reasoning about the correctness of test outcomes based on system and consistency model guarantees. Stress testing and fault injection techniques can function complimentary to litmus tests and enable a wider range of outcomes to become observable in a system under test.

6.3 Dissertation Conclusions

By identifying performance and efficiency bottlenecks in testing methodologies and adapting techniques from microprocessors to system domains like IoT and distributed systems, this dissertation represents an important advance in testing and analyzing correctness in concurrent systems.

Creating tools and methodologies to guarantee the correctness of concurrent systems, as well as the need to optimize performance and efficiency of these tools is becoming increasingly important as system complexity rises. In particular, automatic verification techniques using formal methods typically suffer from scalability issues and have very high costs in terms of time and resources; time spent in validation and verification can be more than half of the system design effort. Empirical testing, however, provides solutions that can effectively reduce the need for formal verification or

could cause a shift towards a more targeted use of these techniques. The work presented in this dissertation can save thousands of hours in concurrent systems' testing, driven by orders of magnitude higher efficiency and speedup. Such improvement significantly reduces the cost of the validation and verification processes and consequently improves time-to-market.

Overall, perpetual litmus tests, Distributed Litmus Tests and the tools presented in this thesis advance the way we think about consistency testing in concurrent systems and inspire the systems community to further advance testing methodologies and hardware validation and verification techniques. Testing on multiprocessors, distributed systems and smart home systems using variants of litmus tests has proven to improve performance, efficiency and outcome variety dramatically, clearly demonstrating the value of the methodologies and tools presented in this dissertation and advocating for their adoption as a platform for efficient validation, conformance testing and bug finding in both experimental and mature systems and architectures.

Bibliography

[1] A. Adir and G. Shurek. Generating Concurrent Test-programs with Collisions for Multi-processor Verification. In *Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop*, HLDVT '02, pages 77–. IEEE Computer Society, 2002.

[2] Allon Adir, Hagit Attiya, and Gil Shurek. Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture. *IEEE Trans. Parallel Distrib. Syst.*, 14(5):502–515, May 2003.

[3] Sarita V. Adve and Mark D. Hill. Weak ordering -a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 2–14. ACM, 1990.

[4] Aerospike. What is a Key-Value store? https://www.aerospike.com/what-is-a-key-value-store/.

[5] Jade Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2):178–210, 6 2012.

[6] Jade Alglave, Mark Batty, Alastair F. Donaldson, Ganesh Gopalakrishnan, Jeroen Ketema, Daniel Poetzl, Tyler Sorensen, and John Wickerson. GPU Concurrency: Weak Behaviours and Programming Assumptions. *SIGPLAN Not.*, 50(4):577–591, March 2015.

[7] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in Weak Memory Models. In *Proceedings of the 22nd International Conference on Computer Aided Verification*, CAV'10, pages 258–272. Springer-Verlag, 2010.

[8] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Litmus: Running Tests against Hardware. *Tools and Algorithms for the Construction and Analysis of Systems Lecture Notes in Computer Science*, pages 41–44, 2011.

[9] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, July 2014.

[10] Bluetooth Alliance. Bluetooth, a global wireless standard for simple, secure connectivity. https://www.bluetooth.com/.

[11] Connectivity Standards Alliance. Zigbee wireless mesh network technology. http://www.zigbee.org/.

[12] Z-Wave Alliance. Z-Wave wireless control technology. http://www.z-wave.com/.

[13] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. Automating failure testing research at internet scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 1728. Association for Computing Machinery, 2016.

[14] Peter Alvaro, Kolton Andrus, Chris Sanden, Casey Rosenthal, Ali Basiri, and Lorin Hochstein. Automating failure testing research at internet scale. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 1728. Association for Computing Machinery, 2016.

[15] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 331–346. ACM, 2015.

[16] Amazon Alexa webpage. https://developer.amazon.com/alexa.

[17] Amazon web services. https://aws.amazon.com/.

[18] Eric Anderson, Xiaozhou Li, Mehul A. Shah, Joseph Tucek, and Jay J. Wylie. What consistency does your key-value store actually provide? In *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, HotDep'10, page 116, USA, 2010. USENIX Association.

[19] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[20] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable Race Detection for Android Applications. *SIGPLAN Not.*, 2015.

[21] Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam. A Detailed Analysis of Contemporary ARM and x86 Architectures. *UW-Madison Technical Report*, 2013. https://minds.wisconsin.edu/handle/1793/64923.

[22] James Bornholt. Memory Consistency Models: A Tutorial, Feb 2016. https://homes.cs.washington.edu/ bornholt/post/memory-models.html.

[23] James Bornholt and Emina Torlak. Synthesizing Memory Models from Framework Sketches and Litmus Tests. *SIGPLAN Not.*, 52(6):467481, June 2017.

[24] Eric Brewer. A certain freedom: Thoughts on the cap theorem. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, PODC '10, page 335. Association for Computing Machinery, 2010.

[25] Eric Brewer. CAP Twelve years later: How the "Rules" have Changed. *Computer*, 45:23–29, 02 2012.

[26] Eric Brewer. Pushing the CAP: Strategies for Consistency and Availability. *Computer*, 45(2):2329, February 2012.

[27] Caroline June Trippel. Concurrency and Security Verification in Heterogeneous Parallel Systems. PhD thesis, November 2019. `http://arks.princeton.edu/ark:/88435/dsp01gt54kq90n`.

[28] Apache Cassandra. Cassandra Architecture. https://cassandra.apache.org/doc/latest/architecture/index.html.

[29] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 269–284, 2014.

[30] William W Collier. *Reasoning about parallel architectures*. Prentice-Hall, Inc., 1992.

[31] Compaq Computer Corporation. Alpha architecture reference manual, 1992.

[32] Katherine E. Coons, Sebastian Burckhardt, and Madanlal Musuvathi. Gambit: Effective unit testing for concurrency libraries. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 15–24. ACM, 2010.

[33] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Googles globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), August 2013.

[34] F. Corella, J. M. Stone, and C. M. Barton. A formal specification of the PowerPC shared memory architecture. *CS Tech. Report RC 18638 (81566)*, 1993.

[35] Intel Corporation. 9th Generation Intel Core Desktop Processors, 2019. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/9th-gen-core-desktop-brief.pdf.

[36] Nvidia Corporation. The GeForce 16 Series Graphics Cards are Here, 2019. https://www.nvidia.com/en-us/geforce/graphics-cards/gtx-1660-ti/.

[37] Flavin Cristian. Understanding fault-tolerant distributed systems. *Commun. ACM*, 34(2):5678, February 1991.

[38] James Davis, Gregor Kildow, and Dongyoon Lee. The case of the poisoned event handler: Weaknesses in the node.js event-driven architecture. In *EuroSec'17*, 2017.

[39] James Davis, Arun Thekumparampil, and Dongyoon Lee. Node.fz: Fuzzing the server-side event-driven architecture. In *EuroSys '17*, 2017.

[40] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205220. Association for Computing Machinery, 2007.

[41] M. Demirbas and S. Kulkarni. Beyond TrueTime : Using AugmentedTime for Improving Spanner. In *LADIS '13: 7th Workshop on Large-Scale Distributed Systems and Middleware*, 2013.

[42] R. H. Dennard, F. H. Gaensslen, H. Yu, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[43] Multiprocessor Diagnostics. ARCHTEST program, 2009. http://www.mpdiag.com/archtest.html.

[44] M. Elver and V. Nagarajan. McVerSi: A test generation framework for fast memory consistency verification in simulation. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 618–630, March 2016.

[45] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security Analysis of Emerging Smart Home Applications. In *S&P '16*, 2016.

[46] Earlence Fernandes, Justin Paupore, Amir Rahmati, et al. FlowFence: Practical Data Protection for Emerging IoT Application Frameworks. In *Proceedings of the 25th USENIX Security Symposium*, August 2016.

[47] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. Ski: Exposing kernel concurrency bugs through systematic schedule exploration. In *OSDI '14*, 2014.

[48] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *EuroSys '17*, 2017.

[49] Harry Foster. Part 8: The 2018 Wilson Research Group Functional Verification Study, January 2019. https://blogs.mentor.com/verificationhorizons/blog/2019/01/29/part-8-the-2018-wilson-research-group-functional-verification-study/.

[50] Harry D. Foster. Trends in functional verification: A 2014 industry study. In *Proceedings of the 52nd Annual Design Automation Conference*, DAC 15. Association for Computing Machinery, 2015.

[51] Apache Foundation. Apache Cassandra Database. http://cassandra.apache.org.

[52] Apache Foundation. Apache Cassandra Read Repair. https://cassandra.apache.org/doc/latest/operating/read_repair.html.

[53] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 15–26, 1990.

[54] Seth Gilbert and Nancy Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.

[55] Google Personal Assistant webpage. https://assistant.google.com.

[56] Google Cloud Platform. https://cloud.google.com/.

[57] Google Play App Store. https://play.google.com/store/apps.

[58] Stewart Grant, Hendrik Cech, and Ivan Beschastnikh. Inferring and asserting distributed system invariants. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1149–1159. ACM, 2018.

[59] J. Gray and D. P. Siewiorek. High-availability computer systems. *Computer*, 24(9):39–48, 1991.

[60] gRPC Community. A high performance, open source universal rpc framework. https://grpc.io/.

[61] Rachid Guerraoui and André Schiper. Fault-tolerance by replication in distributed systems. In Alfred Strohmeier, editor, *Reliable Software Technologies — Ada-Europe '96*, pages 38–57, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[62] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 7:1–7:14. ACM, 2014.

[63] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49. IEEE Press, 2016.

[64] S. Hangal, D. Vahia, C. Manovit, J. . J. Lu, and S. Narayanan. TSOtool: a program for verifying memory systems using the memory consistency model. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 114–123, June 2004.

[65] Michael Heller. BlueBorne Bluetooth vulnerabilities affect billions of devices. http://searchsecurity.techtarget.com/news/450426377/BlueBorne-Bluetooth-vulnerabilities-affect-billions-of-devices.

[66] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[67] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[68] Mark D. Hill and Vijay Janapa Reddi. Accelerator-level parallelism. *CoRR*, abs/1907.02064, 2019.

[69] Scott Hilton. Dyn Analysis Summary of Friday October 21 Attack. https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/.

[70] Grant Ho, Derek Leung, Pratyush Mishra, et al. Smart locks: Lessons for securing commodity internet of things devices. In *ASIA CCS '16*, 2016.

[71] Developing for Homekit. https://developer.apple.com/homekit/.

[72] Inria. The diy software suite, July 2019. http://diy.inria.fr/.

[73] D. Iorga, T. Sorensen, J. Wickerson, and A. F. Donaldson. Slow and steady: Measuring and tuning multicore interference. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 200–212, 2020.

[74] Assignor to Texas Instruments. Jack S. Kilby, Dallas Texas. Miniaturized electronic circuits. In *US Patent 3,138,743*, Filed February 6, 1959. Issued June 23, 1964.

[75] Casper S. Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. Stateless model checking of event-driven applications. *SIGPLAN Not.*, 2015.

[76] Pallavi Joshi, Mayur Naik, Chang-Seo Park, and Koushik Sen. Calfuzzer: An extensible active testing framework for concurrent programs. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 675–681, Berlin, Heidelberg, 2009. Springer-Verlag.

[77] Andreas Kamilaris, Vlad Trifa, and Andreas Pitsillides. Homeweb: An application framework for web-based smart homes. 2011.

[78] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 175–189, 2017.

[79] Kyle Kinsbury. Consistency Models. https://jepsen.io/consistency.

[80] Kyle Kinsbury. Jepsen Analyses. http://jepsen.io/analyses.

[81] Kyle Kinsbury. Jepsen: Distributed Systems Safety Research. https://jepsen.io/.

[82] Philip Koopman. Reliability, safety, and security in everyday embedded systems (extended abstract). In Andrea Bondavalli, Francisco Brasileiro, and Sergio Rajsbaum, editors, *Dependable Computing*, pages 1–2, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[83] Ori Lahav, Nick Giannarakis, and Viktor Vafeiadis. Taming release-acquire consistency. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 649–662, 2016.

[84] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[85] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[86] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[87] Changhui Lin, Vijay Nagarajan, and Rajiv Gupta. Efficient Sequential Consistency Using Conditional Fences. *International Journal of Parallel Programming*, 40(1):84–117, 2012.

[88] Linode Cloud Hosting. https://www.linode.com/.

[89] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 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*, ASPLOS '19, pages 411–425. ACM, 2019.

[90] Natasha Lomas. Update bricks smart locks preferred by Airbnb. https://techcrunch.com/2017/08/14/wifi-disabled/.

[91] Haonan Lu, Kaushik Veeraraghavan, Philippe Ajoux, Jim Hunt, Yee Jiun Song, Wendy Tobagus, Sanjeev Kumar, and Wyatt Lloyd. Existential consistency: Measuring and understanding consistency at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 295–310. ACM, 2015.

[92] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS '08*, 2008.

[93] Jeffrey F. Lukman, Huan Ke, Cesar A. Stuardo, Riza O. Suminto, Daniar H. Kurniawan, Dikaimin Simon, Satria Priambada, Chen Tian, Feng Ye, Tanakorn Leesatapornwongsa, Aarti Gupta, Shan Lu, and Haryadi S. Gunawi. Flymc: Highly scalable testing of complex interleavings in distributed systems. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 20:1–20:16. ACM, 2019.

[94] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Pipecheck: Specifying and verifying microarchitectural enforcement of memory consistency models. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 635–646, 2014.

[95] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. ArMOR: Defending Against Memory Consistency Model Mismatches in Heterogeneous Architectures. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 388–400, 2015.

[96] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. Automated synthesis of comprehensive memory model litmus test suites. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 661–675, 2017.

[97] Rupak Majumdar and Filip Niksic. Why is random testing effective for partition tolerance bugs? *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[98] Yatin Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. CCICheck: Using μhb Graphs to Verify the Coherence-Consistency Interface. In *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-48, 2015.

[99] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Aarti Gupta. PipeProof: Automated memory consistency proofs for microarchitectural specifications. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 788–801. IEEE Press, 2018.

[100] Yatin A. Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. RTLcheck: Verifying the Memory Consistency of RTL Designs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 463–476, 2017.

[101] Luc Maranget, Susmit Sarkar, and Peter Sewell. A Tutorial Introduction to the ARM and POWER Relaxed Memory Models. 2012.

[102] Margaret Martonosi. Check Research Tools and Papers. https://check.cs.princeton.edu/.

[103] T. Melissaris, M. Markakis, K. Shaw, and M. Martonosi. PerpLE: Improving the Speed and Effectiveness of Memory Consistency Testing. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 329–341, 2020.

[104] T. Melissaris, K. Shaw, and M. Martonosi. Locomotive: Optimizing Mobile Web Traffic Using Selective Compression. In *2017 IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–4, 2017.

[105] Themis Melissaris, Kelly Shaw, and Margaret Martonosi. OKAPI: In Support of Application Correctness in Smart Home Environments. In *2019 Fourth International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 173–180, 2019.

[106] Themis Melissaris, Kelly A. Shaw, and Margaret Martonosi. Optimizing IoT and Web Traffic Using Selective Edge Compression. *CoRR*, abs/2012.14968, 2020.

[107] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. Finding crash-consistency bugs with bounded black-box crash testing. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 33–50, Berkeley, CA, USA, 2018. USENIX Association.

[108] G. E. Moore. Cramming more components onto integrated circuits, Reprinted from Electronics, Volume 38, Number 8, April 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[109] Scott Owens, Susmit Sarkar, and Peter Sewell. A Better x86 Memory Model: x86-TSO. *Lecture Notes in Computer Science Theorem Proving in Higher Order Logics*, pages 391–407, 2009.

[110] Karl Rupp. 48 years of microprocessor trend data, 2020. https://github.com/karlrupp/microprocessor-trend-data/blob/master/48yrs/48-years-processor-trend.pdf.

[111] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

[112] Mukesh Singhal and Ajay Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43(1):47–52, 1992.

[113] Samsung SmartThings. Samsung SmartThings Smart home Monitoring kit. https://www.smartthings.com/.

[114] Samsung SmartThings. SmartThings Architecture Documentation. http://docs.smartthings.com/en/latest/architecture/.

[115] Tyler Sorensen and Alastair F. Donaldson. Exposing Errors Related to Weak Memory in GPU Applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 100–113. ACM, 2016.

[116] Daniel J. Sorin, Mark D. Hill, and David A. Wood. A Primer on Memory Consistency and Cache Coherence. 2011.

[117] SPARC International. *The SPARC architecture manual, version 8*. Prentice-Hall, Inc., 1991.

[118] SPARC International. *The SPARC architecture manual, version 9*. Prentice-Hall, Inc., 1994.

[119] Tuan Ta, Xianwei Zhang, Anthony Gutierrez, and Bradford M. Beckmann. Autonomous Data-Race-Free GPU Testing. In *IEEE International Symposium on Workload Characterization*, 2019.

[120] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., USA, 2006.

[121] Caroline Trippel, Yatin Manerkar, et al. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. In *ASPLOS '17*, 2017.

[122] Tengfei Tu, Xiaoyu Liu, Linhai Song, and Yiying Zhang. Understanding real-world concurrency bugs in go. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 865–878. ACM, 2019.

[123] Viktor Vafeiadis, Thibaut Balabonski, Soham Chakraborty, Robin Morisset, and Francesco Zappa Nardelli. Common Compiler Optimisations Are Invalid in the C11 Memory Model and What We Can Do About It. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 209–220, 2015.

[124] Vera Control Smart Home Hub. http://getvera.com/.

[125] Vera forum. http://forum.micasaverde.com/.

[126] Paolo Viotti, Christopher Meiklejohn, and Marko Vukolić. Towards property-based consistency verification. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, PaPoC '16, pages 1:1–1:4. ACM, 2016.

[127] Paolo Viotti and Marko Vukolić. Consistency in non-transactional distributed storage systems. *ACM Comput. Surv.*, 49(1), June 2016.

[128] Werner Vogels. Eventually Consistent. *Commun. ACM*, 2009.

[129] John Wickerson, Mark Batty, Tyler Sorensen, and George A Constantinides. Automatically Comparing Memory Consistency Models. In *ACM SIGPLAN Notices*, volume 52, pages 190–204, 2017.

[130] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, page 283294. Association for Computing Machinery, 2011.

[131] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, Broomfield, CO, October 2014. USENIX Association.

[132] Kamal Zellag and Bettina Kemme. Consistency anomalies in multi-tier architectures: Automatic detection and prevention. *The VLDB Journal*, 23(1):147–172, February 2014.