Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

A Verified and Scalable Hash Table for the TLC Model Checker

Towards an Order of Magnitude Speedup

MSc Thesis

Research Group Theoretical Foundations of Computer Science (TGI)
Department of Informatics
Faculty of Mathematics, Informatics and Natural Sciences
University of Hamburg

by

Markus A. Kuppe

on

June 13, 2017

Reviewer: Dr. Daniel Moldt (UHH),
Leslie Lamport, PhD, DHC (mult), (Microsoft Research)

---

msc-thesis-2017@lemmster.de
Master of Science in Computer Science
Matrikelnummer: 6095945

The applicability of explicit state model checking is restricted by the state space explosion problem which makes the verification of large scale models infeasible. An approach to mitigate state space explosion, is to scale up and benefit from today's multi-core computers. According to Gunther's universal scalability law, scalability is constrained by contention and coherence. For breadth-first model checking with TLC, contention is already minimal. Consequently, we analyze the effects of coherence on TLC's scalability and present a lock-free hash-table algorithm for a model checker's set of seen states. Contrary to most hash-table algorithms present in model checkers, ours is not memory bound. It can be extended to disk. We verify the correctness of our algorithm with TLC.

We provide a production-ready implementation of the algorithm in JAVA for which we demonstrate and quantify superior performance and scalability over TLC's legacy set of seen states. We then propose two additional algorithms for the set of unseen states and the state forest, for which we also include prototypic implementations. When we combine all our proposed changes into TLC, its scalability then exceeds that of SPIN.

Our work on TLC's scalability also reveals that the popular approach of lock-striping does not scale. This result is of interest as an argument in favor of lock-free programming, despite its inherent complexity.

Die Einsetzbarkeit von explizitem Model Checking wird durch das explosionsartige Anwachsen des Zustandsraums eingeschränkt, wodurch die Größe der prüfbaren Modelle begrenzt ist. Dem Anwachsen des Zustandsraums lässt sich jedoch mit vertikaler Skalierung auf Multiprozessor-Systemen begegnen. Gunthers universelles Scalability Law, unterteilt Skalierbarkeit in Contention und Coherence. Die in TLC eingesetzte Breitensuche ist in Bezug auf Contention bereits optimal. Daher werden die Auswirkungen von Coherence auf die Skalierbarkeit von TLC untersucht. Zur Beseitigung der beobachteten Contention, wird ein nicht-blockierender Hashtable Algorithmus vorgeschlagen, der zur Speicherung der beim Model Checking bereits besuchten Zustände dient. Dieser Algorithmus kann im Gegensatz zu bereits bekannten Algorithmen die Menge der Zustände auf die Festplatte auslagern, so dass das Model Chechking nicht durch den verfügbaren Hauptspeicher begrenzt ist. Die Korrektheit des Algorithmus wurde selbst mit Hilfe von TLC überprüft.

Die Skalierbarkeit des gezeigten Algorithmus wird anhand einer produktiv nutzbaren Implementierung empirisch überprüft und die Überlegenheit gegenüber der vorherigen Lösung demonstriert. Ferner werden zwei weitere Algorithmen mit dazugehörigen Prototypen vorgestellt, mit denen die Menge der noch zu prüfenden Zustände und der State Forest in Bezug auf Coherence optimiert werden. Sind alle drei Verbesserungen kombiniert, übersteigt die Skalierbarkeit von TLC, die Skalierbarkeit von SPIN.

Im Rahmen dieser Arbeit wurde außerdem die Erkenntnis gewonnen, dass das Konzept des Lock-Stripings zur Verbesserung der Skalierbarkeit von blockierenden Algorithmen nicht skaliert. Dadurch steigt die Bedeutung von nicht-blockierenden Algorithmen, trotz der damit einhergehenden höheren Komplexität. Diese Komplexität kann aber unter anderem mit Hilfe von Model Checking beherrscht werden.

This document can be found by searching the Web for the 17-letter string `uidmscthesis17mak`. Please do not put this string in any other document. You can refer to it e.g. as "the string obtained by removing the - from `uid-mscthesis17mak`".

Contents

List of Algorithms

List of Figures

1. Introduction

In 1988, Edsger W. Dijkstra argued that a "*programmer's task is not just to write down a program, but that his main task is to give a formal proof that the program he proposes meets the equally formal functional specification.*" [Dijkstra, 1988]. Unfortunately, since then formal methods have neither received a wide recognition nor been adopted throughout the industry. However, today's computer systems are both ubiquitous and pervasive and thus touch every aspect of life. This fact, combined with a young generation of technology-literates that embrace computer systems, will force industry to provide better systems. To this end, programmers require the training to add formal methods to their tool-belt and be provided with proper tools to verify programs. Such tools ought to be capable of verifying programs — despite the state space explosion problem — beyond the size of toy examples. One well-studied class of such tools are Model Checkers.

1.1. Problem Statement

Moore's law — the doubling of transistors on a chip every 18 months — is predicted to end eventually. The laws of physics put a limit to ever-increasing transistor density. This fact has shifted the way computers are built from single to multi-core architectures. Today, even small portable computers such as smart phones are equipped with multiple cores. This change does not come for free however. The software running on the hardware has to be rewritten in order to benefit from multi-core architectures. This is a non-trivial task because parallel algorithms have to be devised that exhibit minimal coherence and contention among their concurrently executing parts. Otherwise, it is possible that a program will run more slowly on multiple cores than its sequential counterpart.

The algorithm used by most model checkers to check safety properties has been shown to be parallelizable. What remains an open problem is to scale explicit state model checking up to hundreds or more cores. This problem is among the list of challenges, which Clarke [2008] — the father of model checking — believes to "*require major breakthroughs*

in order to become sufficiently practical for widespread use in industry." Limiting coherence and contention is the key area to make parallelization efficient. While high core counts still do not solve the state space explosion problem, they make feasible the verification of larger, previously intractable models.

Additionally, scaling a model checker vertically on multi-core architectures paves the road for distributed explicit state model checking, thus scaling horizontally where contention and coherence are even more limiting.

1.2. Reading Guide

Chapter 2 introduces basic concepts required to understand the later chapters of this thesis. It starts with definitions relevant in the context of explicit state model checking. It then outlines TLA$^+$ and TLC and discusses parallel model checking.

The concepts found in chapter 3 represent the fundamentals outside the scope of model checking. In particular, 3.1 introduces concepts of concurrent systems, such as locks and lock-freedom. Afterwards, the mathematics to quantify the scalability of concurrent system is explained. It concludes with the theory of hash tables and thoughts relevant to sorting. A reader with existing knowledge of these topics may wish to skip to the following chapter.

Chapter 4 identifies the shortcomings in TLC that hinder scalability. This is followed by chapter 5, which list the requirements for the new algorithm and the implementation presented as part of this thesis. The chapter 6 outlines the contributions made by other authors to explicit state model checkers that are relevant for this thesis. The three chapters follow the same structure, such that the discussion first focuses on the level of a model checker to then move the focus to the seen set C (hash table) specifically.

In chapter 7, we present the new algorithm and discuss its specification. Its implementation is introduced in detail in chapter 8. The performance and scalability of the implementation is studied empirically in chapter 9.

Chapter 10 summarizes the work carried out as part of this thesis. Finally, chapter 11 ends with an outline of future work.

The appendix, besides the acknowledgement, an index, the glossary, acronyms and bibliography, contains a copy of the TLA$^+$ specifications discussed throughout this thesis. It also introduces the subset of the TLA$^+$ and PlusCal language constructs, which appear in this thesis. A reader unfamiliar with either TLA$^+$ and PlusCal, may want to bookmark the reference on page 117.

1.3. About this Thesis

The author wrote this thesis in his capacity as a Microsoft Research employee and member of the TLA$^+$ project. The thesis is not a project of the Theoretical Foundations of Computer Science (TGI) research group at the University of Hamburg.

The work carried out as part of this thesis was embedded in efforts to push the envelope of explicit state model checking with TLC. The author's other ongoing research work in this area — together with his colleagues of the TLA$^+$ project — includes:

- Distributed TLC
- Cloud Distributed TLC
- Checking liveness properties under symmetry
- Concurrent Search for Strongly Connected Components

Distributed TLC scales explicit state model checking horizontally across multiple nodes of a network of computers. Cloud distributed TLC — an extension to distributed TLC — allows users of TLC to tap into the compute power provided by cloud computing while hiding the idiosyncrasies of the distributed system.

Checking liveness properties under symmetry [compare Emerson and Sistla, 1996] is about exploiting symmetries — corresponding to the group of the state graph automorphisms — to reduce the number of states that have to be checked during liveness checking. Also relevant to liveness checking, is the scalability of the search for strongly connected components. The canonical algorithm invented by Tarjan [1972] is sequential, which severely limits the scalability of liveness checking with TLC.

2. Background

This chapter introduces the theoretical foundations for (explicit state) model checking. This is followed by section 2.2, which puts the focus on the Temporal Logic of Actions (TLA$^+$) specification language and its companion tools. Most importantly, the TLA Checker (TLC) — one of the companion tools — will be introduced from a user and an algorithmic perspective.

2.1. Model Checking

A model checker is a tool that aids in the verification of the system being modeled. A user specifies the system, such as a computer program or an algorithm, and asks the model checker if the given specification adheres to a set of desired properties. Where validation such as testing shows the presence of errors, model checking proves the absence of errors. It therefor provides a higher assurance then testing.

2.1.1. States and Behaviors

Lamport [2008] discusses various ways to represent the possible executions of concurrent or distributed systems. Among which, Lamport suggests to describe a single execution as a sequence of states that the system possibly assumes. A state is defined to be the assignment of values to a finite set of variables. The set of all states is the system's state space, which is usually denoted by S. To describe executions of continuously running systems, the sequence of states is infinite. A terminating execution is represented by an infinite sequence, of which the last state repeats indefinitely. Lamport [1994] calls an infinite sequence of states a behavior. The meaning of the system is the set of all behaviors. The set of behaviors is typically denoted by Σ.

Throughout section 2.1, we will explore the concepts with the help of the example system ω. The system ω writes a value from the set $\{a, b\}$ to the variable x and a value from the set $\{0, 1, 2\}$ to the variable y. However, the system is only allowed to flip the

value of x and set $y = 0$ if and only if (iff) $y = 1$. Listing 2.1.1 shows the system's state space S_ω and its initial states I_ω with $I_\omega \subseteq S_\omega$.

$$\begin{gathered} S_\omega = \{(x = a \wedge y = 0), (x = b \wedge y = 0), (x = a \wedge y = 1), \\ (x = b \wedge y = 1), (x = a \wedge y = 2), (x = b \wedge y = 2)\} \\ I_\omega = \{(x = a \wedge y = 0), (x = b \wedge y = 0)\} \end{gathered} \tag{2.1.1}$$

2.1.2. Transition System and State Graph

The set of behaviors Σ is often represented with a transition system T, which we loosely define to be the triple $\{S, I, \rightarrow\}$:

- S, the finite set of all states
- I, a set of initial states with $I \subseteq S$
- $\rightarrow$ denotes a left-total[1] next state relation

Note however, that it is possible to represent the transition system with other concepts such as a Petri Net [compare Peterson, 1981].

A possible next state relation for ω, consisting of a disjunct of the two possible transitions, is shown in 2.1.2. It uses the notation of primed variables to refer to the successor state of the current state.

$$\begin{gathered} \rightarrow_\omega \triangleq \\ (y = 1 \wedge y' = 0 \wedge x' \in \{a, b\}) \\ \vee (y' \in \{1, 2, 3\} \wedge x' = x) \end{gathered} \tag{2.1.2}$$

The transition system is often used to generate a directed graph (digraph) whose nodes represent (reachable) states $V = S$ and whose arcs represent transitions $E = S \times S$. In section 2.2.3 and in schematic 2.2.3 we will refer to the transition system as the procedure to generate the state graph. The digraph is called the state graph. A behavior of the system corresponds to an infinite path in the state graph.

[1] The transition relation $\rightarrow$ models deadlock states as states whose only (outgoing) transition is a self-loop.

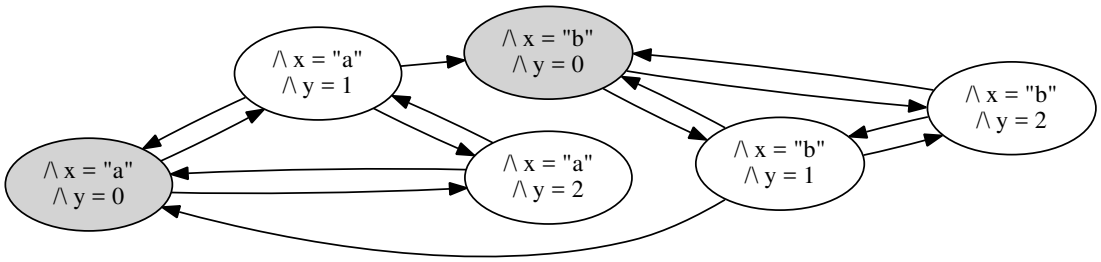


Figure 2.1.1.: The state graph of the ω system

An example state graph is illustrated in figure 2.1.1 for the concurrent system ω introduced above. The initial states are denoted by a gray background.

2.1.3. Safety and Liveness Properties

A system has various properties, such as performance properties, energy efficiency, or security properties. For example, chapter 9 studies performance and scalability properties of a lock-free algorithm. In the scope of this section however, we will focus on behavioral properties of a system. A (behavioral) property is a set Π of behaviors. Under this definition, showing that a system — described by its set Σ of behaviors — satisfies a property amounts to showing that all its behaviors are in Π, hence $\Sigma \subseteq \Pi$. This is the task of the class of explicit state model checkers, which we consider in this thesis (section 2.1.8 gives an overview of other classes).

One of the contributions of Alpern and Schneider [1984] is, that any (behavioral) property is the intersection of a safety and a liveness property. Lamport [1977] was the first to informally define safety and liveness, such that a safety property asserts that something *bad* never happens. A liveness property on the other hand asserts that something *good* eventually happens.

Goo [1985], Alpern and Schneider [1984] provide a formal definition of safety. An (infinite) behavior satisfies a safety property, if and only if its *finite* prefixes satisfy the property. In other words, a prefix of a behavior, that violates a safety property, can not be extended to an (infinite) behavior; the bad thing has already occurred and cannot be remedied. Alpern and Schneider [1984] define a liveness property to be one s.t. every finite prefix of a behavior can be extended to an infinite behavior which satisfies the property. In other words, a liveness property allows a prefix of a behavior, not yet satisfying the liveness property, to be extended to satisfy it; something good can still happen later. Thus, in order for a model checker to verify safety properties, it suffices to look at the finite prefixes of behaviors, whereas the verification of liveness properties

requires to check (infinite) behaviors.

With regards to the example system ω above, an example of a safety property is one that excludes states where $x = b \wedge y = 2$. All behaviors whose corresponding path visits the state with $x = b \wedge y = 2$ violate this safety property. For example, the prefix $\langle x = b \wedge y = 0 \rightarrow x = b \wedge y = 2 \rangle$ of a behavior violates the safety property. Conversely, an example of a liveness property is one, that mandates that $x = a$ eventually leads to $x = b$. All behaviors, whose corresponding path never visits a state with $y = b$ after it visited a state with $y = a$, violate this liveness property.

In a formalism to reason about systems, a property is a predicate on behaviors. Pnueli [1977] proposes to formally represent a property as a temporal logic formula. The observation however, that describing systems as lists of temporal formula is impractical, stirred Lamport [1994] to propose the Temporal Logic of Actions (TLA) language in turn. In TLA, the system is described by a specification Γ, which — in correspondence with Alpern and Schneider [1984] — is a conjunct of a safety property ψ_{safe} and liveness property ψ_{live}, hence $\Gamma = \psi_{safe} \wedge \psi_{live}$, where ψ_{safe} is described by a transition system. Showing that a system has (any) property ϕ, requires to validate $\Gamma \Rightarrow \phi$.

2.1.4. Model Checking Problem

The section 2.1.3 concludes, that in order to show that a specification Γ of a system has a property ϕ, an explicit state model checker has to verify $\Gamma \Rightarrow \phi$, which is equivalent to $\psi_{safe} \wedge \psi_{live} \Rightarrow \phi$. We define this to be the task of an explicit state model checker and call it the model checking problem.

A model checker can only verify specifications with a finite state space.[2] The user must define a model that constrains a specification with an infinite state space to a finite one. In order to verify infinite specifications, one uses a theorem prover (compare section 2.2.2).

If a violation of the property ϕ is detected, the model checker searches for a behavior (or a prefix of a behavior) that is a counterexample. A counterexample is defined to be a behavior that satisfies $\neg\phi$. It is obviously desirable to find a short — preferably the shortest possible — behavior that is a counterexample. A counterexample to a safety property is a finite prefix of a behavior, whereas it is an infinite (lasso shaped) behavior to a liveness property. With regards to the example ω, a counterexample to the example of a liveness property demanding that $x = a$ leads to $x = b$, is depicted in figure 2.1.2.

[2] We deliberately crafted the specification of the ω system to be a finite one.

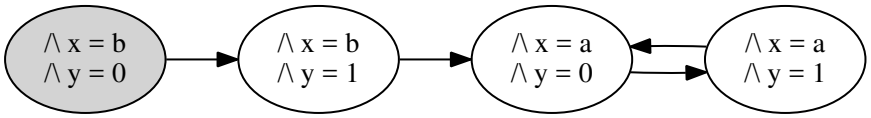


Figure 2.1.2.: A counterexample of the ω system to the liveness property $x = a$ leads to $x = b$

In the counterexample in figure 2.1.2, states with $x = a$ follow states with $x = b$, but not vice versa. Note, that this is not the shortest counterexample.

Abadi and Lamport [1991] prove, that if the specification Γ is machine closed and the property ϕ is a safety property, checking validity can be reduced to checking $\psi_{safe} \Rightarrow \phi$. Informally, a machine closed specification guarantees that the liveness property ψ_{live} is free of any additional safety properties not already stated by ψ_{safe} [compare Apt et al., 1987]. It is important for a specification to be machine closed for it to remain implementable. A machine closed specification is required to prevent a system from "*painting itself into a corner*" [Apt et al., 1987]. A system, executing an implementation of a non machine closed specification, would require a look-ahead to exclude those executions, from which no allowed behavior can be continued [see Henzinger, 1992]. As stated in Yu et al. [1999], the TLC model checker (see section 2.2.3) implicitly assumes the specification to be machine closed.

In order for a model checker to verify $\psi_{safe} \Rightarrow \phi$ when Γ is machine closed and ϕ is a safety property, it suffices to check finite prefixes of behaviors [Lamport, 1977]. This result is important, because the algorithm limited to checking finite prefixes — which can be described as a reachability analysis on the on-the-fly generated state graph — is computationally less expensive compared to the algorithm to check liveness properties. Thus, TLC checks safety properties with BFS, which reduces model checking time considerably (see section 2.2.3).

2.1.5. Complexity of Explicit State Model Checking

The asymptotic complexity of the model checking problem for several temporal logics is analyzed in Sistla and Clarke [1985]. The authors show that for an Linear-Time Temporal Logic (LTL) logic fragment restricted to future operators — thus comparable to TLA — satisfiability is $PSPACE-complete$. The complexity of the decision procedure for the model checking problem — given by $\mathcal{O}\left(|G(K_T)| \cdot 2^{sym(\phi)}\right)$ — consists of the size of the state graph $G(K_T) = |E|+|V|$ (compare section 2.1.1) and the number of symbols $sym(\phi)$ appearing in the formula ϕ. The LTL model checking algorithm in Lichtenstein

N	$\mathbb{N} = \{1, 2, 3\}$	$\mathbb{N} = \{1, 2, 3, 4\}$
1	23	29
2	6189	15362
3	4558640	>10649063[a]

[a] We manually terminated TLC after a few minutes with a queue of unexplored states of size 4169818.

Table 2.1.: Number of (distinct) states of the Bakery algorithm with N modeling the number of processes and two definition overrides for the set of all natural numbers $\mathbb{N}$

and Pnueli [1985] is proven to be exponential in $sym(\phi)$, but linear in the size of G. In other words, assuming that the property ϕ is relatively short, explicit state model checking remains feasible.

2.1.6. State Space Explosion Problem

The previous section 2.1.5 concluded, that the decision problem is linear in the size of the state graph G. However, one of the main challenges of explicit state model checking, is a combinatorial explosion called state space explosion [see Clarke et al., 2012]. The state space explosion is an exponential blowup of the state space corresponding to linear growth in the number of variables $\mathcal{V}$ and their domain returned by dom — the possible values of each $v \in \mathcal{V}$ — in the specification. All but the most trivial specifications face state space explosion. For a specification with $n = |\mathcal{V}|$, the upper bound for the number of possible states is $\prod_{i=1}^{n} dom(v_i)$. In the special case that $\forall v \in \mathcal{V} : dom(v) = k$, the upper bound is k^n. Table 2.1 illustrates state space explosion with Lamport's Bakery algorithm [compare Lamport, 1974] as specified in Lamport [2015a].

Opponents of explicit state model checking will argue that — due to the two previously identified exponential growth rates — this approach is an uphill battle. Several success stories on earth [Newcombe et al., 2015, Joshi et al., 2003] and even the planet Mars [Holzmann, 2014] show, that explicit state model checking has its place in both academia and industry nonetheless. Much of its success is owed to advances in combating state space explosion.

2.1.7. Parallelizing Model Checking

Computing problems range from inherently sequential to embarrassingly parallel problems [Herlihy and Shavit, 2012]. A problem that is said to be inherently sequential,

gains no speedup from parallelization. Conversely, an embarrassingly parallel problem gains ideal speedup from parallelization. An embarrassingly parallel problem incurs no contention or coherence (see 3.2), whereas an inherently sequential problem is defined to have a maximum contention factor preventing any form of parallelization.

Section 2.1.5 notes that explicit state model checking of LTL formulas of arbitrary length is in *PSPACE*. In practice however, formulas are usually sufficiently short to regard them as fixed sized formulas. Under this assumption, Filippidis and Holzmann [2014, section 8.] show that explicit state model checking — generalized to directed graph reachability — is in *NL*; the problems decidable in nondeterministic logarithmic time. The authors further give the relationship of $NL \subseteq NC \subseteq P \subseteq PSPACE$. Nick's class ($NC$) is defined to be the set of problems decidable in polylogarithmic time [see Papadimitriou, 1994]. In other words, problems which are not inherently sequential and instead can be efficiently solved with parallelization. Yet they are neither embarrassingly parallel. Note however that Reif [1985] proved depth-first search (DFS) to be $P-complete$. Thus, DFS is inherently sequential.

The previous paragraph confirms that parallelizing a model checker will result in satisfactory reductions of model checking time. This result justifies the attempt to improve the actual breath-first search (BFS) algorithm and the efficiency of the underlying implementation.

2.1.8. Methods, Specification Languages and Tools

The model checking community proposed a plethora of specification languages and tools to formally define and verify systems, that would warrant a thesis of its own. An introductory overview beyond this paragraph is provided in Frappier et al. [2010].

For the camp of explicit state model checkers, the specification language Promela and the accompanied model checker Spin [Holzmann, 2005] are prominent representatives for which the author even received the ACM System Software Award in 2001 [Holzmann, 2001]. Other explicit state model checkers — which receive active research — are, among others, DiVinE [Barnat et al., 2013], jpf [Havelund and Pressburger, 2000] and LTSMin [Kant et al., 2015]. Where Promela has only limited support to verify implementation code, DiVinE verifies code written in implementation languages such as C and C++. Similarly, jpf can verify Java (byte) code. Generally, verification at the implementation language is geared towards fine grained specifications, whereas dedicated languages are usually meant for high-level specifications. A language independent model checker is LTSMin, which accepts specification written in process algebras,

implementation languages and timed automata. Through the use of PNML2PINS [Meijer, 2015], it is possible to verify (unfolded) Petri nets with LTSMIN [compare McMillan and Probst, 1995]. This allowed LTSMIN to participate and win the LTL formulas category in the Model Checking Contest (MCC) in 2016 [see Kordon et al., 2016].

Additionally, LTSMIN supports symbolic model checking, which is an alternative method to explicit state model checking. Symbolic model checking combats the state space explosion problem by representing sets of states as (boolean) formulas [see McMillan, 1993]. Symbolic model checking exploits regularities in the state space to achieve a more succinct encoding of states. Given there are regularities, symbolic model checking can verify models an order of magnitude larger than those verifiable by an explicit state model checker. NUSMV is another noteworthy symbolic model checker. Its specifications are written in the SMV description language [Cimatti et al., 2002].

Verifying systems that exhibit stochastic behavior is the provenance of probabilistic model checking. Here, transitions are assigned probabilities. This makes probabilistic model checking amendable to model and verify not just computer, but also biological systems. In addition to the decision problem described above in 2.1.3, probabilistic model checking answers queries such as "the property φ holds 95% of the time". A prominent probabilistic model checker is PRISM [Kwiatkowska et al., 2011] whose specification language is based on the reactive modules formalism [see Alur and Henzinger, 1999]. Oldenkamp [2007] presents a comparative study of probabilistic model checkers.

The most exhaustive listing of model checking tools on the web appears to be on Wikipedia [2017a]. The annually held MCC and the comparisons done with the BEEM database of models [see Pelánek, 2007] are other sources to find model checkers.

2.2. TLA$^+$

This section builds up on TLA and introduces the TLA$^+$ specification language. Section 2.1.8 additionally provided an overview on the methods and tools prevalent in the model checking community. In the scope of TLA$^+$, the TLC model checker is the canonical tool used to verify TLA$^+$ specifications. The section 2.2.3 below assumes a user centric view on TLC and maps this view and an algorithmic perspective of TLC to the formal foundation in the previous section 2.1.

2.2.1. TLA$^+$ Specification Language

TLA$^+$ is a formal specification language based on TLA. TLA$^+$ is not limited to, but is primarily used to write high-level specifications of concurrent and distributed systems. TLA$^+$ differs from typical specification languages, which are usually inspired by programming languages. Instead, TLA$^+$ favors expressiveness by the use of mathematical abstraction to aim for compact specifications. As such, TLA$^+$ is built on set theory, first-order and temporal logic. TLA$^+$ is an action based language where an action is a first-order formula on primed and unprimed variables. The language is untyped. Concerns, like the program efficiency, are outside its scope. Lamport et al. [2002] go into more detail on this topic. TLA$^+$ has been applied to a broad spectrum of problems, ranging from hardware protocols (see Joshi et al., 2003) to distributed algorithms running today's data-centers [Newcombe et al., 2015]. It is actively used in industry and academia alike (e.g. Lu et al., 2011).

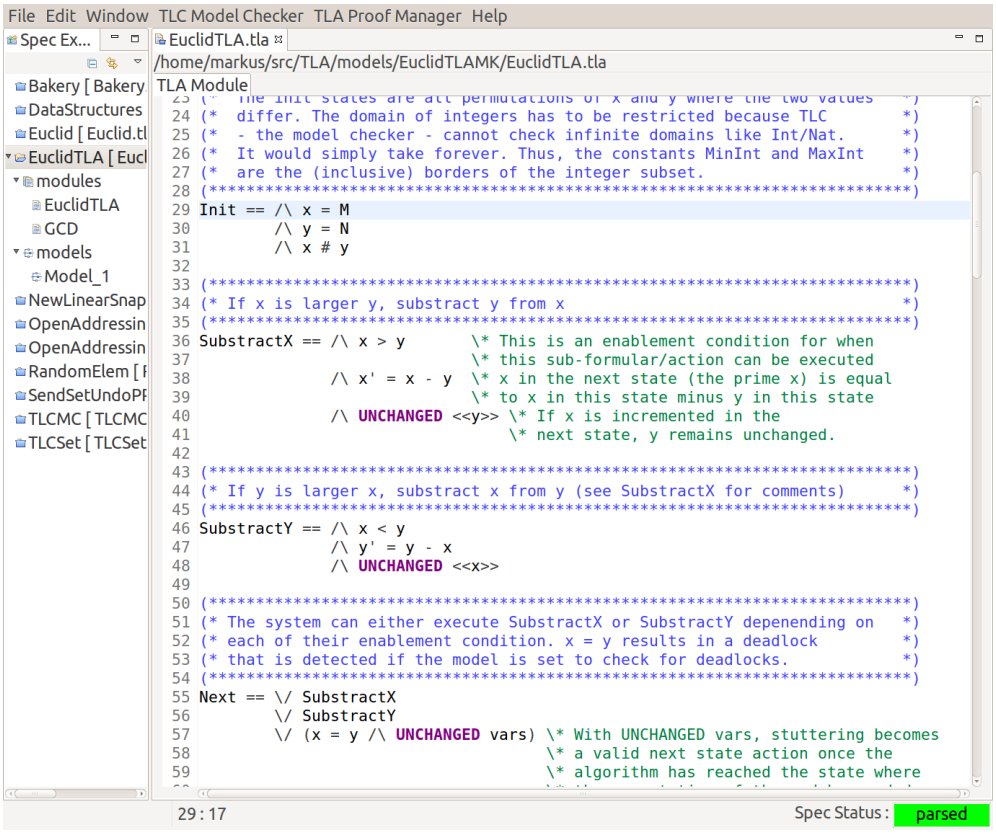
Lamport [2006, 2009] introduced PlusCal as an algorithm language to replace informal pseudo-code. A PlusCal expression can be any TLA$^+$ expression. It is thus equally expressive as TLA$^+$ but — due to its resemblances of traditional programming languages — the gateway to TLA$^+$ for engineers. The TLA Toolbox (see 2.2.2 below) automatically transpiles PlusCal to TLA$^+$, but not vice versa. The canonical introductions to TLA$^+$ and PlusCal are found in Lamport [1994, 2003, 2009, 2014]. As of late, the first episodes of a multi-volume video course on TLA$^+$, primarily targeting programmers and engineers, have become available too [Lamport, 2017]. More episodes are expected to appear later.

2.2.2. TLA Toolbox

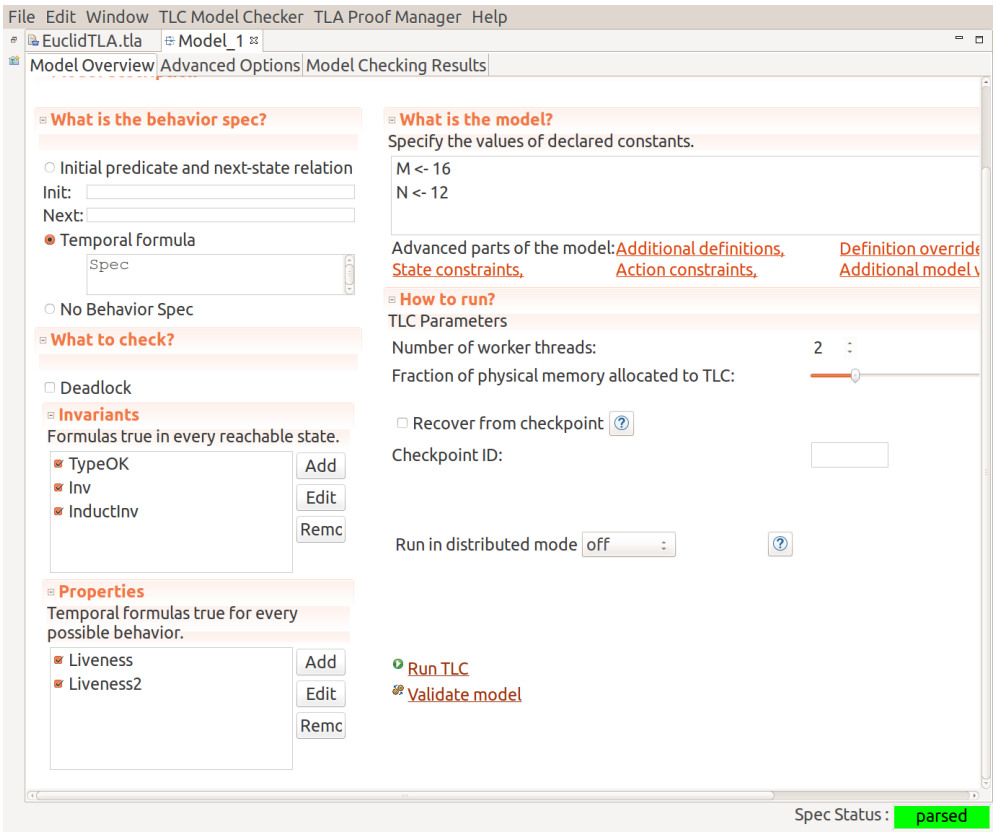
The TLA Toolbox is the official integrated development environment for TLA$^+$. It is based on Eclipse and provides support for writing TLA$^+$ and PlusCal specifications (see figure 2.2.1a). Additionally, it has a graphical user interface to configure and tune TLC's input parameters (figure 2.2.1b). TLC can be launched locally or remotely (compare Kuppe, 2014) from within the TLA Toolbox. The TLA Toolbox supports launching both the parallel and the distributed mode of TLC.

Model checking results are visualized and counterexamples can be inspected with the Error-Trace. The Error-Trace allows one to evaluate arbitrary state predicates and action predicates on a counterexample (figure 2.2.2a and figure 2.2.2b).

Besides the model checker, the TLA Toolbox also integrates the TLA$^+$ proof sys-

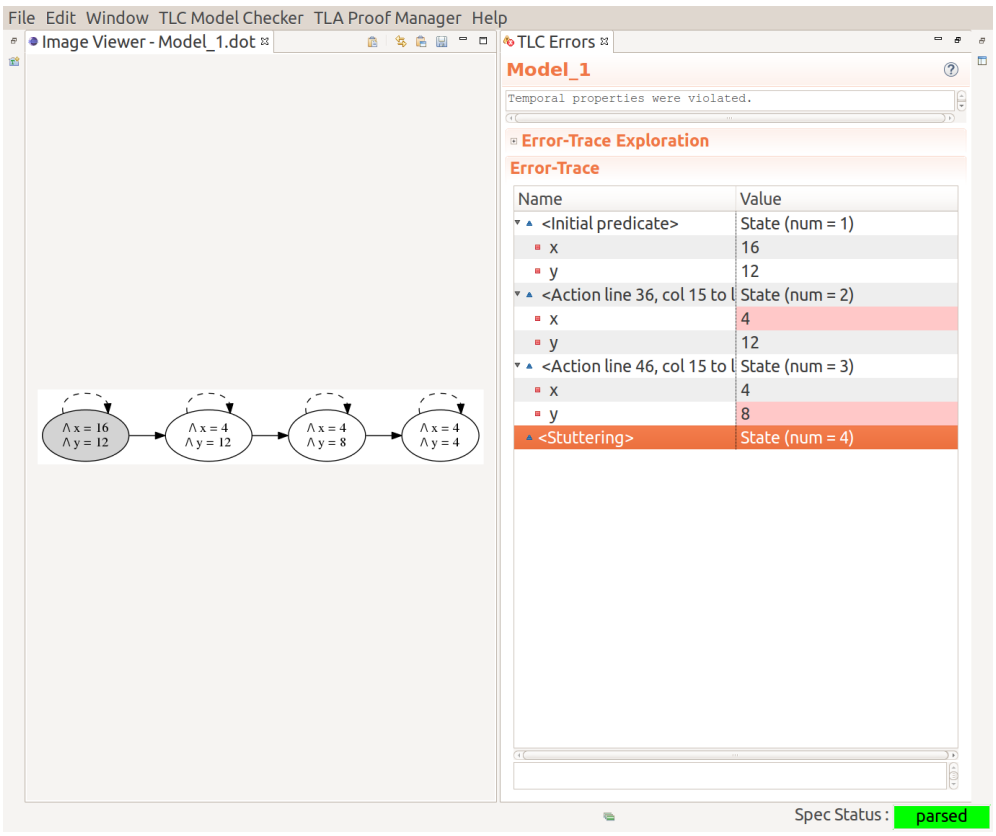


(a) Editing a specification inside the TLA Toolbox

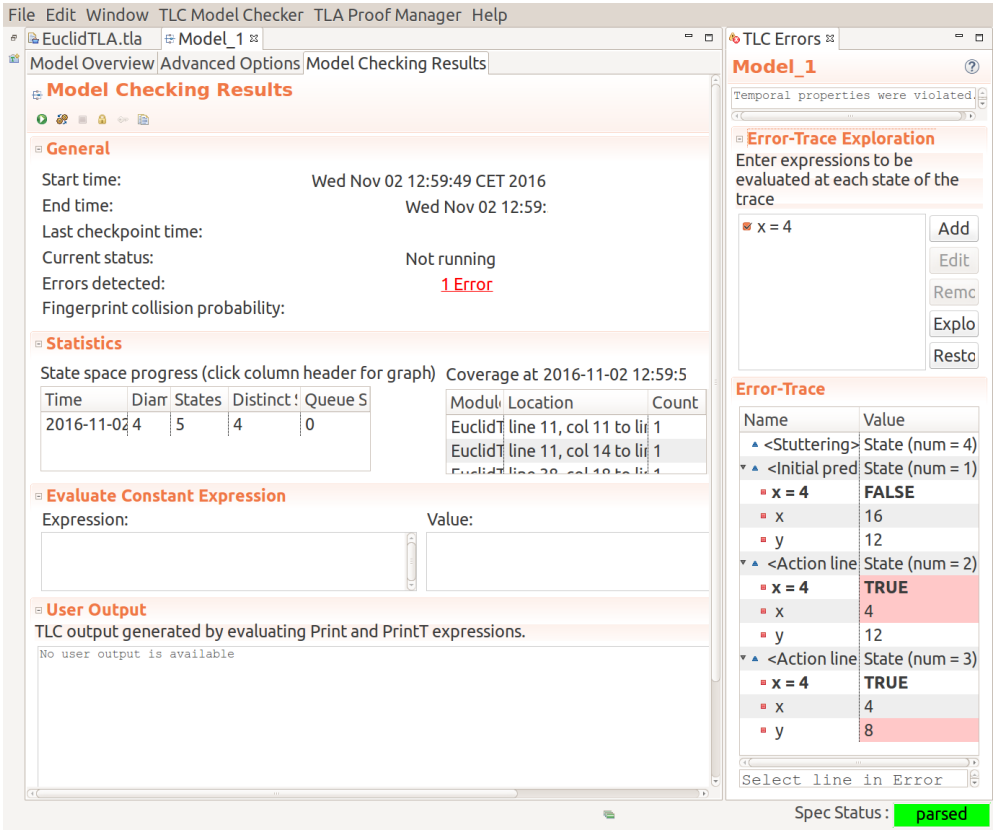


(b) Model configuration of input parameters for TLC

Figure 2.2.1.: TLA Toolbox screenshots of its specification and model editors



(b) Basic graphical counterexample visualization



(a) Exploring a counterexample with the Error-Trace after TLC found a violation

Figure 2.2.2.: TLA Toolbox screenshot of its Error-Trace and counterexample visualization

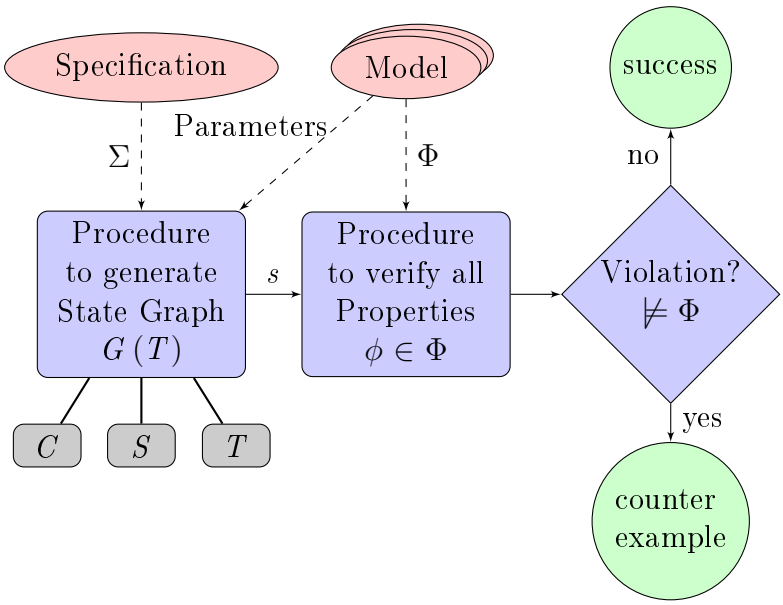


Figure 2.2.3.: Schematic of the TLC model checker.

tem (TLAPS) [Chaudhuri et al., 2010]. TLAPS can mechanically check proofs written in TLA$^+$ and thus supports the verification of infinite specifications. TLAPS proofs are structured hierarchically [Lamport, 1995]. TLAPS can be described to be a front-end theorem prover, which uses various existing theorem provers as back-ends. As of now, TLAPS is limited to reasoning about safety properties.

2.2.3. TLC Model Checker

TLC is an explicit state model checker originally written by Yu et al. [1999] to verify TLA$^+$ specifications. TLC can check a subset of TLA$^+$ that is commonly needed to describe real-world systems, most notably finite systems.

The schematic figure 2.2.3 is a simplified view on TLC. A user of TLC, provides TLC with a specification and one to n models. The specification — written as a temporal formula (see section 2.1.4) — specifies the system; such as the Bakery mutual exclusion algorithm (Lamport, 1974). Conceptualized, the specification can be seen as the definition of the system's set of behaviors Σ. A model on the other hand, declares the set of properties Φ, that the system is expected to satisfy. In case of the Bakery algorithm, a user may wish to verify the safety property, that no two processes are in the critical section simultaneously. Also, the model declares additional input parameters, such as (system) constants and TLC runtime options. The parameters shall not concern us here.

With regards to checking safety properties, TLC consists of a generator procedure to create the state graph $G(T)$, which we called the transition system in section 2.1. First, the procedure generates the specification's initial states I. Afterwards, given a state and the specification's actions, TLC enumerates all possible successor- or next-states (see $\rightarrow$ in section 2.1). Simplified, each successor state s is fed to the second procedure. The second procedure checks, if the state satisfies the model's safety properties Φ.[3] If a state is found to violate a property, a counterexample is constructed.

The first procedure generates the state graph on-the-fly by running BFS.[4] Consequently, it maintains two sets of states:

- The unseen set S to store newly generated, still unexplored states. A term synonymously used for S in the context of TLC is state queue.
- The seen set C to store explored states. TLC calls C the fingerprint set motivated by the fact that C stores state hashes rather than states (more on that later).

Contrary to regular BFS, TLC maintains a third data structure to construct a counterexample:

- A (state) forest T of rooted in-trees to construct the shortest path from any state $s_n \in S$ back to an initial state $s_o \in I$. The states in I correspond to the roots in T.

The PlusCal pseudo-code in algorithm 1 shows a simplified variant of TLC's algorithm to check safety properties (full listing in section B.1). Readers, unfamiliar with TLA$^+$ or PlusCal, are referred to the reference list in appendix A on page 117 for a brief introduction of the language constructs. First, TLC generates the initial states atomically and adds them to the set of unseen states S. The *init* loop checks each initial state for a violation. If the state does not violate the properties, it is added to C (algorithm 1 line 6 to line 13). The second *scsr* loop executes BFS (algorithm 1 line 14 to line 30). The *SuccessorsOf* operation corresponds to the next state relation (see $\rightarrow$ in section 2.1). The $succ \in ViolationStates$ check represents the procedure to verify the safety properties Φ_S.

TLC naïvely parallelizes BFS by concurrently executing the while loop *scsr* with multiple threads — called workers. The global data structures S, C, T are shared by all workers.[5] Consequently, each data structure has to be guarded by locks to guarantee

[3] Liveness properties are checked on behaviors instead of states.
[4] TLC also supports to run DFS.
[5] The actual implementation keeps the two global PlusCal variables *state* and *successors* thread-local.

Algorithm 1 : TLC's algorithm to check safety properties

```
1  --algorithm ModelChecker{
2        variables
3              S ∈ SetOfAllPermutationsOfInitials(StateGraph),
4              C = {}, state = null, successors = {},
5              i  = 1, counterexample = ⟨⟩, T = ⟨⟩ ; {
6        init: while (i ≤ Len(S)){
7              state := Head(S) ;
8              C := C ∪ {state} ;
9              i  := i + 1 ;
10             if (state ∈ ViolationStates){
11                   counterexample := ⟨state⟩ ; goto trc ;
12             } ;
13       } ;
14       scsr:  while (Len(S) ≠ 0){
15             state := Head(S) ; S := Tail(S) ;
16             successors := SuccessorsOf(state, StateGraph, C) ;
17             if (successors = {state}){
18                 counterexample := ⟨null⟩ ; goto trc ;
19             } ;
20             each: while (successors ≠ {}){
21                with (succ ∈ successors){
22                      successors := successors \ {succ} ;
23                      C := C ∪ {succ} ; S := S ∘ ⟨succ⟩ ;
24                      T  := T ∘ ⟨⟨state, succ⟩⟩ ;
25                      if (succ ∈ ViolationStates){
26                        counterexample := ⟨succ⟩ ; goto trc ;
27                      } ;
28                } ;
29             } ;
30       } ;
31       goto Done ;
32       trc :   while (TRUE){
33                 if (Head(counterexample) ∉ StateGraph.initials){
34                    counterexample :=
35                          ⟨Predecessor(T, Head(counterexample))⟩ ∘ counterexample ;
36                 }else {
37                   goto Done ;
38                 } ;
39       } ;
40   }
41 }
```

consistency. Sharing S among all workers achieves optimal load-balancing. We call this mode of execution parallel TLC. TLC can also execute on a network of computers in what is called distributed mode [Kuppe, 2012]. Distributed TLC speeds up model checking.

TLC is implemented in and available on all operating systems supported by JAVA. TLC runs on commodity hardware found in desktop computers, up to high-end servers equipped with hundreds of cores and terabytes of memory. It is capable of checking large-scale models, i.e. the maximum checkable size of a model is not bound to the available memory. Instead, TLC keeps the in-memory working set small:

1. TLC swaps unexplored states in the unseen set S from memory to disk. The unseen set S has a fixed space requirement.

2. TLC constantly flushes the forest to disk. Iff a violation is found, TLC reads the forest back to disk to construct the counterexample.

3. The seen set C — whenever it exceeds its allocated memory — extends to disk.

1 and 2 enable TLC to allocate the majority of memory to the seen set C. When the seen set C exceeds its allocated memory (3), TLC is said to run disk-based model checking. Once TLC switches to disk-based model checking, its performance, i.e. the number of states generated per unit of time, drops significantly. Still, disk-based model checking has its place. It makes the verification of state space tractable, for which the primary memory alone is insufficient, e.g. Newcombe [2014] gives an account where TLC ran for several weeks to check a state space of approximately 2^{34} states.

TLC supports the use of TLC module overwrites in TLA$^+$ and PlusCal specifications. A TLC module overwrite is a static JAVA method which matches the signature of a TLA$^+$ operator.[6] With a TLC module overwrite in place, TLC delegates the evaluation of the corresponding TLA$^+$ operator to the JAVA method. Generally, this is assumed to be faster. A TLC module overwrite is stateless and cannot access its context. Additionally, TLC module overwrites make the JAVA library available to TLA$^+$ users. Later, in section 7.3 and section 8.4 we will return to TLC module overwrites.

[6]TLC uses TLC module overwrites for most of the built-in operators in the standard modules.

3. Fundamentals

In alignment with the thesis' title, the previous chapter 2 covered explicit state model checking and set the stage for the work discussed in later chapters. This chapter introduces the generic material beyond explicit state model checking, which is needed by later chapters.

In the upcoming section 3.1, we detour from model checkers and discuss generic concurrent systems and relevant synchronization mechanisms to deal with concurrency. Later chapters build up on the synchronization mechanism. In section 3.2, this chapter also introduces the mathematics to quantify the scalability of concurrent systems. The section 3.3 will discuss the use of hash tables in the scope of explicit state model checking. Finally, section 3.4 lays out the fundamentals of sorting, which are relevant in later chapters.

3.1. Concurrent System

A concurrent system — such as a distributed, a multi-processor, or multi-core system[1] — is a set of processes. A task is a unit of work, consisting of n operations, which are sequentially carried out by an individual process. Collectively, the processes work on achieving a common goal, which is the combined result of the set of tasks. In short, a concurrent system executes multiple tasks at the same time. The processes communicate via shared or distributed memory.

Consequently, a concurrent system allows two (or more) operations a and b to happen concurrently. If these two operations are neither conflict-free, nor disjoint access parallel [compare Israeli and Rappoport, 1994], s.t. the operations manipulate the same regions of shared memory, the operations have to be ordered. Otherwise, it is easy to see that the concurrent system will eventually assume an inconsistent state, e.g. when multiple concurrent processes read and write the same set of variables non-atomically.

[1] We will not concern ourselves with differences of multi-processor and multi-core architectures.

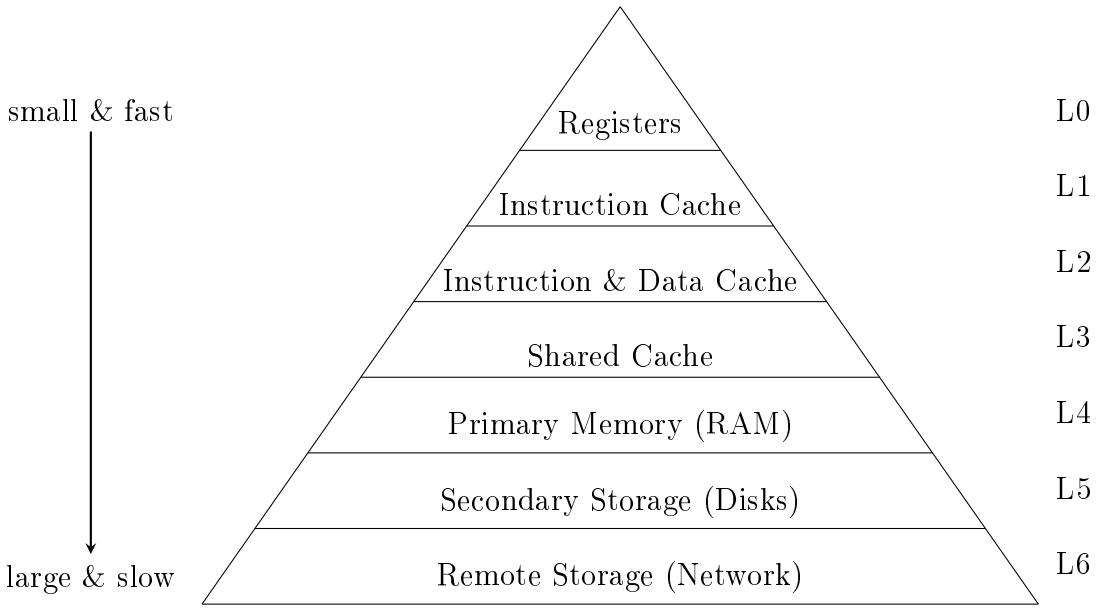


Figure 3.1.1.: Computer memory hierarchy [inspired by Wikipedia, 2017b]

Current processor generations provide weak or relaxed consistency models to achieve a higher level of concurrency and ultimately better performance [Gharachorloo et al., 1991]. Those consistency models accept optimization such as reordering of operations or speculative execution [compare Adve and Gharachorloo, 1996].

Besides optimization at the operation level, current hardware implements the Non Uniform Memory Access (NUMA) architecture, where several layers of caches are placed in-between the processor and primary memory (see 3.1.1). Each cache layer (L1 to L3) — with increasing processor proximity — provides several orders of magnitude faster access times. On the other hand, a cache layer in close processor proximity is of smaller size compared to far distanced caches.[2] Data or memory is fetched from primary memory into caches in fixed sizes, called cache lines. A cache miss happens, whenever a cache line has to be fetched from primary memory.

Implementing programs at such a low abstraction level is challenging where memory modifications have to be ordered with (basic) fence instructions.[3] A fence makes memory modifications globally visible by copying a cache line back to primary memory and invalidating copies in foreign caches.

Likewise, reasoning about relaxed consistency models as described above, is difficult due to the size of the accompanied state space. In other words, the set of required

[2]We use the term processor and core interchangeably. Shared Cache (L3) means, that it is shared by multiple processors.

[3]The level of concurrency is inversely proportional to the number of fence instructions executed.

behaviors to describe the concurrent system is significantly larger compared to models with stronger consistency guarantees.

In contrast, the sequential consistency model — introduced by Lamport [1979] — defines operations to be atomic, s.t. two (conflicting) operations a and b are represented by a set of two behaviors: One behavior in which a happens before b and one where a happens after b. This definition makes it easier to reason about a concurrent system.

The JAVA programming language uses the sequential consistency model. It exposes only a limited set of synchronization primitives, notably the two keywords `volatile` and `synchronized` [Manson et al., 2005]. Both are translated to fence instructions specific to the actual processor architecture. Newer JAVA releases also include the `java.util.concurrent` package. The package supplies the programmer with concurrent data structures, high-level locks and atomic primitives for single variables. Access to Compare and Swap (CAS) is unavailable at the vendor neutral API level but accessible via `sun.misc.Unsafe` [compare Frampton et al., 2009, Mastrangelo et al., 2015].

3.1.1. Compare and Swap

The availability of atomic instructions in addition to non-atomic memory reads and writes, makes programming concurrent systems possible. CAS is an atomic instruction provided by the hardware layer. CAS writes the (word-sized) value v_{new} to a memory location m iff the current value v_c of m equals an expected value v_e where v_e is an expected value. If $v_e \# v_c$ the CAS operations returns v_c or false depending on the actual implementation.[4] CAS can be further refined to:

DWCAS Double word CAS of two adjacent/contiguous word-sized memory locations.

DCAS Double word CAS of two arbitrary word-sized memory locations.

CASN CAS of any number of arbitrary memory locations. CASN is also called MCAS.

Only DWCAS is supported by current processor generations (see i.e. CMPXCHG16B instruction on the x86 architecture). However, within this thesis we are further restricted to JAVA's limited support for basic CAS. In the absence of hardware support, CASN — as shown by Harris et al. [2002] — can be implemented in software. Fraser and Harris [2007] too provide CASN. However, the performance provided by these solutions is unsatisfactory for practical purposes.

[4] In JAVA `sun.misc.Unsafe` returns true or false.

Herlihy [1991] shows, that the CAS instruction has a universal consensus number and can thus solve the consensus problem [see Fischer et al., 1985] for an unbound number of processes. It is therefor a universally applicable synchronization primitive and fits the sequential consistency model.

Generally, a CAS instruction takes significantly more CPU cycles to execute, compared to a non-atomic write [see Schweizer et al., 2015]. Thus, in cases where a group of writes has to occur, a lock based approach is likely to be the more applicable synchronization mechanism.

3.1.2. Locks

Locks are the de facto synchronization mechanisms to achieve mutual exclusion in concurrent systems. A lock blocks all other processes — waiting for the same lock — from making forward progress. For performance reasons, modern lock implementations follow a two-phased strategy: First, a process spins briefly while trying to acquire the lock. If the acquisition fails, the process is put to sleep by the scheduler until it can successfully acquire the lock later. Two-phase locking can reduce context switching and the number of cache misses. Context switching, or preemption, can have a degrading effect on concurrency if the system interrupts the process which holds a lock. Note that the spin-phase is usually implemented with CAS. Highly contended locks thus directly lead to excessive cache misses and consequently degrade concurrency.

It might be desirable to allow multiple readers to access a resource concurrently, for all to make forward progress. If a writer however requests exclusive write access, readers are blocked. This type of lock is called a shared lock.

If possible, lock-striping — at the application layer — can alleviate the problem of contended locks. Lock-striping is the concept of employing n locks to guard a data structure, separated into n disjunct partitions.

Choosing the right level of lock granularity is paramount. Coarse grained locking limits concurrency whereas fine grained locking produces significant overhead caused by lock acquisition and an increased memory footprint.

Unless proven otherwise, locking can result in deadlocks. A deadlock prevents the concurrent system from making any forward progress at all.

3.1.3. Lock-Free

The previous sections climbed the abstraction ladder from the lowest-level synchronization primitives called fence instructions over atomic instructions like CAS to high-level lock-based synchronization. This section introduces lock-freedom as an alternative to locking. Contrary to locking, it does not encapsulate the CAS synchronization primitive into a high-level synchronization mechanism, but instead is a programming pattern which directly uses CAS.

Formally, lock-freedom is the property, that no process in a concurrent system can block other processes from making forward progress. Herlihy and Shavit [2012] define a method as lock-free: "[...] *if it guarantees that infinitely often some thread calling this method finishes in a finite number of steps.*". In this sense, lock-freedom is a stricter property of a concurrent system compared to locking, because it gives a guarantee of overall forward progress. It is however possible for individual processes to be starved indefinitely (no fairness). From the perspective of concurrency, lock-freedom is robust against unfavorable process preemption. Lock-freedom can be regarded as nonblocking optimistic locking.

Algorithm 2 outlines a while-loop which reads a value *expected* from memory, executes — a potentially expensive — *someComputation* based on *expected* and attempts to atomically replace its result with *expected* using CAS. Iff $result = false$, another attempt is made which involves the re-execution of *someComputation* because the value of *expected* is known to have changed.

Algorithm 2 : Lock-free programming pattern

```
1  loop:  while (TRUE){
2             expected := memory[index];
3  cas:       CAS(result, memory[index], expected, someComputation(expected));
4             if (result){
5                goto done
6             };
7          };
8  done:   ...
```

A lock-free algorithm is generally regarded as more complex and thus more difficult to reason about than a lock-based counterpart. This stems from the fact, that it allows for a larger number of behaviors. This generally justifies a formal verification of the (high-level) algorithm and — due to the variety of memory models found in hardware

— a likewise verification of (core parts of) the implementation.

In contrast to locks, the advantage of lock-freedom - implemented with atomic instructions such as CAS - is that it can yield better performance and scalability. Tsigas and Zhang [2002] compare the effects of lock-freedom for seven applications. Four out of seven applications show significant speedups when their implementation uses lock-freedom. The remaining three applications exhibit no change in performance compared to the lock-based variants. Morrison, 2016 — on the other hand — shows for lock-free Michael and Scott queues [compare Michael and Scott, 1996] inferior performance. The underlying reason is a higher number of failed CAS operations. For JAVA's synchronization primitives, Thompson [2013] suggests that lock-freedom achieves higher performance in specific scenarios. In summary, lock-freedom is only warranted in the scope of well defined problems. Quantifying the expected performance gain achieved by applying lock-free programming appears difficult a-priori. The publications mentioned above all rely on empirical performance analysis. Thus, the next section introduces the math required to empirically examine the algorithm in chapter 9 proposed by this thesis in chapter 7.

3.2. Scalability

The lack of a rigorous definition of scalability stirred Hill [1990] to write that "[...] *calling a system "scalable" is about as useful as calling it "modern".*" While everybody has an intuitive understanding of what scalability means, the term remains ambiguous.

Informally, a scalable (concurrent) system is capable of finishing a larger amount of tasks within a fixed time interval when its computing resources are increased. Applied to explicit state model checking, a model checker can be called scalable if it can verify a larger number of states[5] on more powerful hardware within the same length of time.

Gunther [2007] provides a formal definition of scalability: the Universal Scalability Law (USL). Gunther describes scalability by the following function, which relates the relative capacity or speedup Z, to the number of processes N of a concurrent system:[6]

$$Z(N) = \frac{N}{1 + \varsigma(N-1) + \kappa N(N-1)} \tag{3.2.1}$$

[5] As shown in 2.1.5, the size of Φ - the properties to be checked - is an alternative dimension to the number of states.

[6] Gunther denotes the relative capacity with $C(N)$. We substitute $Z(N)$ for $C(N)$ to make it distinguishable from the seen set C. Likewise, we substitute σ for ς. We adopted the structure of this section from Möding [2016a].

The three terms in the denominator of equation (3.2.1) represent concurrency, contention, and coherence respectively:

Concurrency The first term represents optimal concurrency in a concurrent system. It is thus independent of N. An embarrassingly parallel problem exhibits neither coherence nor contention, and thus scales linearly. A model checker running BFS on the other hand, cannot exhibit optimal concurrency; it exhibits contention and coherence.

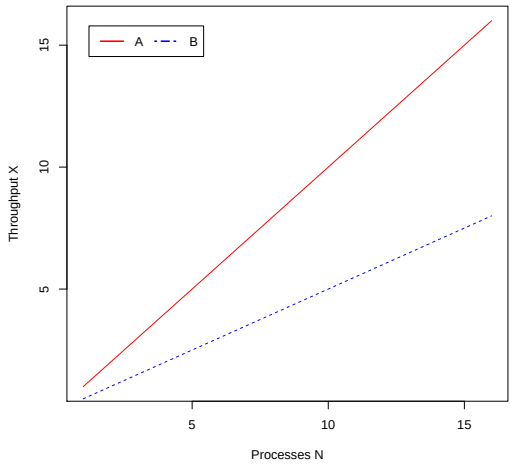
Contention (ς) The factor ς in the second term quantifies the inherent serial portion of the problem, thus an increase in ς leads to a decrease in speedup Z. The contention of the model checking problem is determined by the explicit state model checking algorithm (compare section on Model Checking) and the shape of the state graph. For example, if we consider a specification describing a single behavior, $\varsigma = 1$ and thus speedup $Z = 1$ for any N; the algorithm has to generate the sequence of states sequentially. Another source is lock contention, which also adds to the overall contention observed in a model checker.

Coherence (κ) The κ factor in the third term quantifies coherence; the overhead induced by maintaining a consistent system state. Consistency requires agreement among N processes, which is why coherence is quadratic in N. In a model checker, this is e.g. the overhead created by crosstalk of workers to maintain a consistent unseen set S and seen set C (see subsection TLC Model Checker).

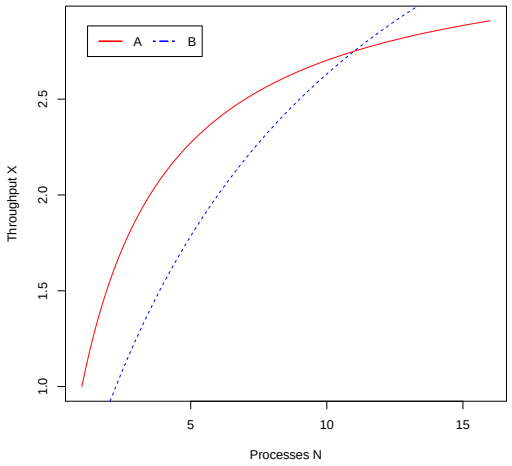
Amdahl [1967] shows, that a linear growth of contention prevents systems from scaling indefinitely. The same restriction is modeled by the contention term in the USL. Additionally, coherence's quadratic growth in the USL implies, that scalability eventually even retrogrades. In other words, additional resources cause the system to slow down. Contrasted to the linear growth of contention, reducing coherence in a concurrent system can therefor be the more effective optimization strategy. A strong indicator for coherence at runtime is a high rate of cache misses.

Ideally, a concurrent system displays linear scalability when $\varsigma = \kappa = 0$. In this case, two scalable systems A and B can only differ in their performance as shown in figure 3.2.1a. When systems exhibit contention ($\varsigma > 0$), a system with lower performance can exceed a higher performing system at a certain number of processes (Subfigure 3 3.2.1b). With coherence ($\kappa > 0$), throughput eventually retrogrades under increased process counts (Subfigure 3 3.2.1c).[7]

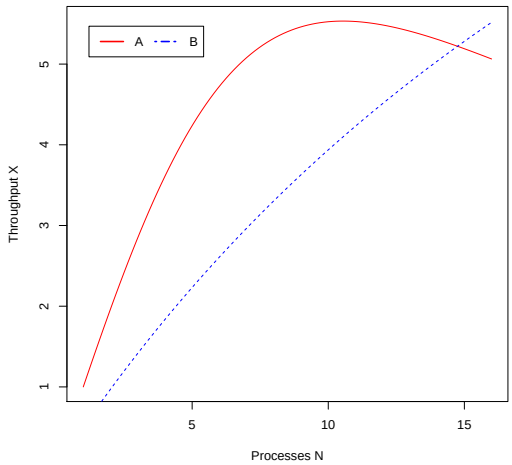
[7] All charts in this thesis are vsiualized with the USL CRAN package [Möding, 2016b] and R scripts



(a) A and B both show linear scalability ($\varsigma = \kappa = 0$). A has superior performance



(b) A and B with sub-linear scalability ($\varsigma > 0, \kappa = 0$). B has inferior performance but superior scalability characteristics. Throughput of B exceeds that of A at approximately 11 processes



(c) System A ($\varsigma = 0, \kappa > 0$) slows down at around 10 processes but is still superior to B up to 14 processes.

Figure 3.2.1.: Effects of different scalability and performance characteristics of two systems

For any real-world concurrent system — such as the TLC model checker — the values for (the physical effects) ς and κ are unknown. Thus, they have to be determined empirically. The standard method is to measure the system's throughputs $X_1, \ldots, X_N$ for varying numbers of processes $1, \ldots, N$ and to derive the corresponding speedups $Z(N)$. A speedup $Z(N)$ is defined to be the increase in throughput X_N normalized to the throughput of a single process X_1, thus $Z(N) = \frac{X_N}{X_1}$. Linear or ideal speedup means that doubling the number of processes, results in a proportional increase in speedup. The system has an ideal efficiency.

Depending on the context, we discuss throughput at the model checker level and at the level of the seen set C. For the latter, throughput X is given in $^{\text{puts}}/_{\text{ms}}$ where a *put* is defined to be an insertion operation into C. For TLC, throughput is defined as the number of states generated and checked per minute.

Combining scalability and the previous section 3.1.2 on locks, it is important to point out, that a lock exhibits not only contention. Maintaining consistency of a lock's internal state, also has coherence attached to it.

3.3. Hash Tables

We continue our detour from explicit state model checking, that we set out on with section 3.1 on concurrent systems, to introduce hash tables next. Later in this section though (see 3.3.3), we combine hash tables with explicit state model checking; specifically the role of hash tables in TLC.

Hash tables are the data structure of choice, if the use case demands constant (amortized) performance. Mathematically, a hash table t can be represented as a sequence of elements without duplicates. The elements are from a universe $\mathcal{U}$. Let K be the size or length of t. The relation $K \ll |\mathcal{U}|$ is assumed to be true.

In order to provide constant times for insertion, deletion and lookup operations, a deterministic index function determines the position in the sequence, at which an element is stored in t. An index function, denoted by idx is a surjective-only mapping of each element in $\mathcal{U}$ to a position in t. Moreover, we expect the index function to uniformly distribute the elements across t and that it can be computable quickly. A good candidate for an index function — hence a hash table's name — is a hash function. A hash function has the property, that it maps elements of arbitrary length to a fixed length k. We chose a hash function with a k, s.t. $2^k = K$.

[Gunther, 2016a].

Once the number of elements in t — denoted by n — exceeds K, the pigeonhole principle tells us, that two or more elements $e_1, e_2 \in \mathcal{U}$ have to be assigned to the same position. This is called a collision. A collision even occurs before $n \geq K$, because of $K \ll |\mathcal{U}|$: we map a larger universe to a smaller one. Counterintuitively, the probability for two or more elements to share the same index, s.t. $\exists e_1, e_2 \in \mathcal{U} : idx(e_1) = idx(e_2)$ is high, even for sparsely populated hash tables. This effect has been studied under the term birthday paradox [see Knight and Bloom, 1973].

The load factor α of a hash table t is defined to be $\alpha \triangleq \frac{n}{K}$. Additionally, we will look at the utilization. Let K_{gross} be the gross size of a hash table — the effective space required by the hash table data structure — we define the utilization to be $util \triangleq \frac{n \times k}{K_{gross}}$. In other words, *util* is a quality measurement of hash tables.

3.3.1. Collision Resolution

Literature lists various collision resolution strategies to deal with (index) collisions. The two most prominent collision resolution strategies are Separate Chaining (SC) and Open Addressing (OA). SC handles collisions by chaining colliding elements into a bucket. Mathematically, a hash table with SC can thus be represented by a sequence of sequences. With SC, the index function maps an element to the outer sequence. In other words, it maps each element to a collision bucket. Therefor, we will subsequently call the hash function used with SC *bucketIdx*.

If nested buckets are unordered sets of elements, the lookup complexity degrades to $\mathcal{O}(1 + \alpha)$: In addition to the constant bucket lookup based on *bucketIdx*, a linear search of the matching bucket — with load factor α — has to occur. To optimize insertion time, a bucket is implemented as a linked list. A linked list can grow in constant time because an insertion is an append to the list's end. However, a linked list is impractical in memory constrained scenarios due to the extra cost of list pointers. Alternatively, one can choose to represent buckets with basic arrays. However, an array does not grow dynamically. This results in degraded performance, when an insert hits a full array. Other variants of SC, keep buckets constantly sorted with e.g. trees. Such variants improve lookup performance, but trade insertion for lookup performance and thus are geared towards use cases where lookups dominate.

Contrary to SC, OA adheres to the single sequence definition of a hash table above. To resolve collisions, OA employs a probing sequence starting at a primary index, up to L alternative indices. An element is inserted at the first empty position that is found.

Thus, we redefine the function to $idx\,(e, p)$ with $idx : \mathcal{U} \times \{1, 2, \ldots, L\} \mapsto \{1, 2, \ldots, K\}$. In other words, repeated invocations with $p \in \{1, 2, \ldots, L\}$ and a fixed $e \in \mathcal{U}$, return e's primary, secondary, ... up to L-th index.

Several probing sequences are known, such as Linear Probing (LP), quadratic probing or double hashing, with LP being considered both simple and fast [Sedgewick and Wayne, 2011, page 469ff]. With LP, consecutive positions are probed in ascending (or descending) order, s.t. $\forall i, j \in 1..L : \; i+1 = j \implies idx_i + 1 = idx_j$. An advantage of this sequence is its locality preserving property. Locality preserving means, elements stay within close proximity to their primary index, which technically reduces cache misses (compare 3.1).[8] The absence of dynamic bucket resizing implies, that OA produces no (memory) garbage at runtime.

Although the utilization is constant s.t. $util = 1$, OA is no silver bullet. A drawback of OA is a performance deterioration starting at $\alpha \gtrapprox 0.7$. This is related to (primary) clustering, where elements pile up in clusters [compare Sedgewick and Wayne, 2011, p. 472ff]. Clustering leads to long probing sequences which degrade lookups to linear search (compare bounded disorder in section 3.4.1).

3.3.2. Hash Tables in Java

JAVA provides a vast array of hash tables, all of which are restricted to a maximum of 2^{31} elements. For larger sizes, programmers typically implement jagged or two-dimensional arrays with SC. While the outer array has a fixed size, the inner arrays grow dynamically on the heap. This flexibility has its price. Storing the meta-information, such as array dimensions as well as object headers, deteriorates the utilization *util*. This overhead increases proportional with the size of the outer array. In a hash table, it is advisable to maximize the size of the outer array. The size is inversely proportional to the likelihood of bucket collisions and ultimately time spent on linear search.[9] Let $\langle l \rangle$ be the average length of buckets, the graph in 3.3.1 plots the relationship between $\langle l \rangle$ and the hash table's utilization *util*. Larger $\langle l \rangle$ result in a higher utilization, but growths of *util* flattens out at $util = 0.8$. The underlying assumption of figure 3.3.1 are always fully occupied buckets. In reality, a trade off has to be made between frequent bucket resizing and inferior utilization.

With OA, `sun.misc.Unsafe` offers an alternative [compare Frampton et al., 2009, Mastrangelo et al., 2015]. `sun.misc.Unsafe` grants direct, unverified and unconstrained

[8] With SC, logically neighboring buckets do not necessarily occupy adjacent memory regions.
[9] Dynamically adapting the outer array's size causes costly rehashing.

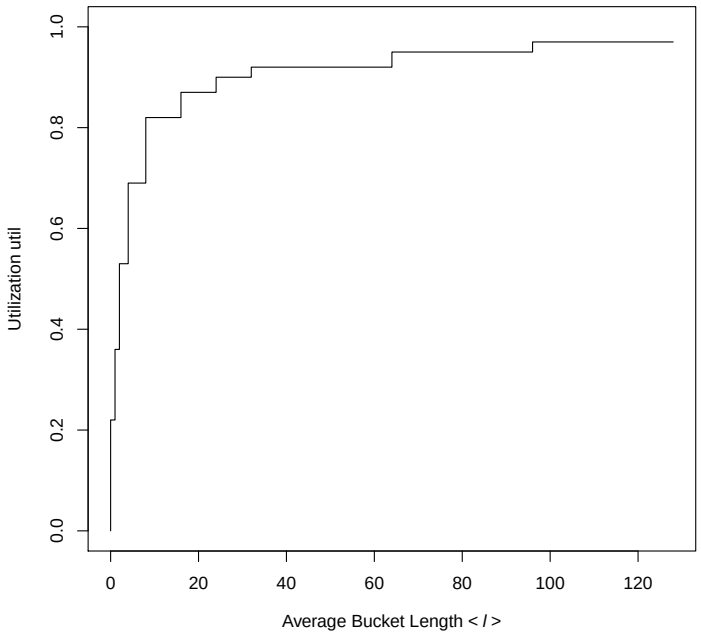


Figure 3.3.1.: Hash-table utilization versus bucket length for a jagged arrays

access to primary memory and can allocate contiguous blocks of up to 2^{63} bytes. The memory, allocated with `sun.misc.Unsafe`, is excluded from JAVA's built-in garbage collector. This is desirable with OA — a garbage collector cannot reclaim memory, due to the absence of dynamic array resizing.

3.3.3. Hash Tables in Model Checking

Holzmann [1988] shows, that maintaining the seen set C — the set of seen states — significantly adds to model checking time. This led Holzmann to propose the *supertrace* method. With supertrace, a model checker uses a hash table to represent the seen set C (see section 2.2.3). Hence, the model checker does not store seen states in C, but their hash values. Consequently, supertrace offers two advantages: It reduces the overall space consumed by C and the time required to determine a state's set membership in C.

A hash function $fps : st \mapsto fp$ — similar to the idx function in 3.3 — maps each state s to a hash value. We call the hash value a state's fingerprint, denoted with fp. A fingerprint is a fixed sized sequence of bits. Supertrace obviously voids the completeness guarantee of explicit state model checking. It is possible for two (or more) states s_1 and

s_2 to collide in a fingerprint collision, s.t. $\exists s_1, s_2 : fps\,(s_1) = fps\,(s_2)$.[10]

It suggests itself to reuse a fingerprint subsequence $fp\,[a:b]$ of length k bits s.t. $2^k = K$ as the fingerprint's index in t. The underlying assumption is, that $fp\,[a:b]$ uniformly distributes across the complete fingerprint space. However, the full fingerprint is still inserted into C, to not increase the probability of fingerprint collisions further.

In contrast to generic hash tables — as discussed in section 3.3 — a hash table in the scope of explicit state model checking, does not need to support element removal: A seen state cannot become unseen again. The seen set C can thus be optimized toward the find-or-put operation: Insert (put) a fingerprint or return false iff already present (find).[11]

With disk-based model checking (compare section 2.2.3), the difference in throughput between primary memory and external storage suggests (see 3.1.1), that C targets a high load factor α and a high utilization $util$. Despite the adversarial effect of a high α on the throughput of the find-or-put operation, the target remains valid for as long as find-or-put throughput exceeds that of external storage. Once the throughput of find-or-put surpasses that of an external storage, either all new or a subset of the current fingerprint generation should be evicted to external storage though.

Extension to External Storage in TLC

Disk-based model checker — which support the verification of state graphs beyond the size of the available memory — best represent the seen set C with a B-tree. A B-tree has $\mathcal{O}\,(log\ n)$ insertion and lookup complexity. While a B-tree yields good performance in the average case, a B-tree is inferior to a hash table in cases where the seen set C fits into memory.

This is why TLC chooses a different, hybrid approach where C is split into a hash table and (linearly sorted) external storage. For as long as the number of seen states fit into t, external storage is not used. Eventually however, once the table's load factor α exceeds a given threshold, the seen set C is extended to external storage. From that point on, find-or-put — for a given fp — requires an external storage lookup, iff the lookup in t returned a miss for fp.

An eviction policy selects fingerprints to be moved from the hash table to external storage. The ideal eviction policy evicts only those fingerprints to external storage, which have the lowest probability of being queried in the future. Hot-spots on the

[10] A fingerprint collision is orthogonal to a hash table collision, which we defined in section 3.3.1.
[11] Laarman et al. [2010] introduce the term "find-or-put".

other hand remain in memory. An on-the-fly model checker cannot predict hot-spot fingerprints, because the probability depends on unknown properties of the state graph. Thus, a naïve eviction policy — such as least recently used (LRU) — evicts fingerprints. Eviction is split in two phases:

The first phase takes place, when the load factor α reaches a pre-defined threshold. At this point, all current fingerprints in t — the old generation — are copied to external storage. This happens atomically from the point of find-or-put. In other words, insertions and lookups in the seen set C are suspended. The old generation however is not purged from t during the first phase. Instead, fingerprints are one by one replaced with new fingerprints during the second phase. This replacement strategy makes the eviction policy of TLC resemble LRU. No (write) access to external storage and pauses are necessary during the second (replacement) phase. The advantage of this two staged approach (copy & replace), is the ability to use the memory as a (read) cache for the external storage. The overall memory utilization is therefor constantly high. Pausing insertions and lookups mandates, that the first phase is short. The second phase does not degrade the performance of find-or-put.

It follows from the previously discussed eviction policy, that fingerprints — located in the hash table — require a mark to indicate their storage location $isMarked : m \mapsto \{table, external\}$. A fingerprint can be located in the hash table exclusively or additionally in the external storage. We will call replaceable fingerprints, those with $m = external$, as being marked.[12].

To maximize the hash table's utilization *util*, the m-bit is encoded in each fingerprint at position $fp\,[m]$. Consequently, this reduces the effective fingerprint length and thus increases the collision probability. The generic solution to keep the collision probability constant, is to split the fingerprint space in half and assign each half to a dedicated set C_{m0} and C_{m1}. Fingerprints are assigned to either one of the sets, depending on $fp\,[m]$.[13] Accordingly, the $fp\,[m]$ has to be excluded from the index calculation *idx* (compare section 3.3.3). Otherwise, C_{m0} and C_{m1} load factors will be limited to $\alpha = 0.5$, because $fp\,[m]$ is fixed in C_{m0} and C_{m1}.

The external storage layout — designed to reside on magnetic disks — is a numerically sorted list of all its fingerprints. Multiple disk seek and read operations may be required to lookup a fingerprint. An index is maintained to find a disk block candidate on which a fingerprint is located. Interpolated binary search is run on the candidate to lookup

[12] In this context, it is irrelevant to indicate if a fingerprint is located on external storage only.
[13] With SC, this solution worsens the overall memory utilization *util*.

the fingerprint. Perl et al. [1978] show that interpolated binary search has $\mathcal{O}(log\ log\ n)$ runtime complexity. Benchmarks revealed an average of 1.05 seek and 1.1 disk read operations per lookup request [Yu, 1999].

This section repeatedly noted, that the external storage is a linearly sorted file. Thus, in order to evict fingerprints from memory to external storage, the fingerprints have to be sorted. The next section will therefor discuss sorting algorithms which are well suited for sorting TLC's seen set C.

3.4. Sorting

The research on sorting algorithms has produced a plethora of useful results. A common differentiator is, if the algorithm works in-place or out-of-place. An in-place or in situ sort requires no auxiliary space — except for list pointers or temporary variables — beyond the number of elements n of the dataset to be sorted. Thus, its auxiliary space complexity is $\mathcal{O}(1)$. In contrast, standard merge sort — an out-of-place algorithm — has an auxiliary space complexity of $\mathcal{O}(n)$. It is easy to see, that only in-place sorting algorithms qualify for the usage in memory constrained environments such as explicit state model checking.

One of the fasted in-place algorithms is Quick Sort [Hoare, 1962]. Quick Sort is fast, because it swaps far distanced elements quickly. Its underlying assumption is completely random data.

3.4.1. Bounded Disorder

For partially sorted data, Quick Sort is not the best algorithm. Dijkstra [1982] proposed SmoothSort as an *adaptive* algorithm with excellent asymptotic behavior. However, it is known to have a high constant factor [Schwarz, 2011]. In Java and Python, the standard sorting algorithm today is TimSort [see Peters, 2011]. TimSort is adapted to patterns found in many real-world datasets. For small datasets, Cook and Kim [1980] show that (straight) Insertion Sort provides good performance.

Partially sorted data can be defined as a dataset where only a small number of elements with $m \ll n$ are misplaced. In another definition given by Estivill-Castro and Wood [1992] with m unbounded, the elements are maximally disordered by *dis* steps from their proper position. With the latter definition it is obvious, why Quick Sort is not the ideal sort algorithm for partially sorted data.

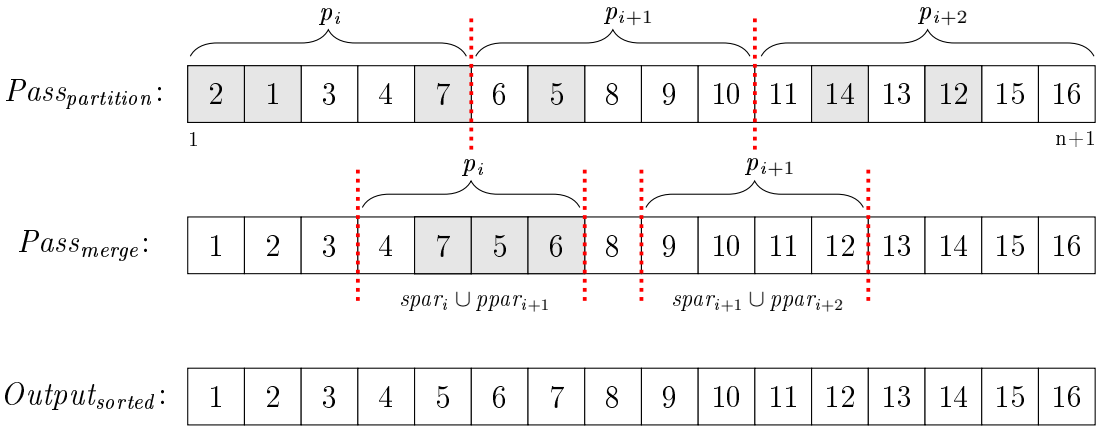


Figure 3.4.1.: Bounded disorder parallel sorting scheme

3.4.2. Naïve Parallel Sort with Bounded Disorder

If the maximum disorder dis of a dataset is known and $dis \ll n$, the dataset can be sorted with any sequential sorting algorithm in parallel. Let P be the set of processes, the dataset can be split into $\lfloor \frac{n}{|P|} \rfloor$ partitions. Each partition par_i is sorted by a process $p_i \in P$. We call this $Pass_{partition}$. In a second pass — a merge phase called $Pass_{merge}$ — each process sorts the elements in the $spar_i \cup ppar_{i+1}$ where $spar_i$ is the suffix of par_i of length dis. The identifier $ppar_{i+1}$ represents the prefix of length dis of partition par_{i+1}; the next higher (or lower) partition to par_i.

This naïve scheme is exemplified in figure 3.4.1 for a disorder $dis = 2$, $|P| = 3$, $n = 16$ and $i = 1$. The dotted, vertical lines mark the partition borders. Shadowed cells indicate disordered elements. The first sequence labeled $Pass_{partition}$, displays the initial, disordered sequence before the partitions par_i, par_{i+1} and par_{i+2} are sorted separately. The $Pass_{merge}$ sequence in the middle is partially sorted, such that only the union $spar_i \cup ppar_{i+1}$ has to be merged. The second union $spar_{i+1} \cup ppar_{i+2}$ is already fully ordered. The $Output_{sorted}$ at the bottom is fully sorted. The example demonstrates, that already sorted elements, i.e. element 6 can become disordered after $Pass_{partition}$.

For the scheme to work, the invariant $Inv_{dis} = \lfloor \frac{n}{|P|} \rfloor > (2 * dis)$ has to hold. Otherwise, the processes do not have exclusive access to partitions.

Under the assumption of uniform disorder, this scheme scales (compare section 3.2) infinitely with $|P|$ given that Inv_{dis} holds. Thus, n has to grow proportionally. The scheme is optimal with regards to cache locality (see 3.1). The scheme increases the number of elements to be sorted in total to $n + (2 * dis * (|P| - 1))$. Thus, this extra work created by parallelization is inversely proportional to $(dis * |P|)$. With a sufficiently

small coefficient *dis* or $|P|$, the second summand becomes negligible.

4. Problem Analysis

The Background and Fundamentals chapters above introduce all relevant concepts and terms to understand the chapters to follow. This chapter starts with an as-is view on TLC prior to this thesis. It identifies shortcomings of TLC and sets a performance and scalability baseline. The chapter 9 below compares the new algorithm — proposed in chapter 7 and implemented in chapter 8 — with this baseline.

Afterwards, this chapter drills down and focuses on the seen set C specifically to analyze its problems. This two staged approach is repeated in chapter 5 and chapter 6.

Throughout the remainder of this thesis, we discuss three variants of TLC. The first variant, denoted by TLC_{legacy}, corresponds to the first version of TLC, as present by Yu et al. [1999]. The second variant corresponds to an intermediate version of TLC, which uses lock-striping to guard the seen set C [Kuppe, 2012]. This variant will be denoted by TLC_{stripe}. The last variant — which includes the lock-free C_{proto} corresponding to the algorithm proposed in chapter 7 — is denoted by TLC_{proto}. Likewise, when we later discuss variants of the seen set C in isolation, the subscripts C_{legacy}, C_{stripe}, and C_{proto} denote the variants respectively. If a statement applies to all variants, no subscript is given.

4.1. Parallel TLC

The first versions of TLC date back to the '90s, the early stages of the multi-core revolution. As such, TLC_{legacy} has indeed been designed for and tested on multi-processor hardware. However, in the '90s the number of processes was still limited to low numbers. This reflects in TLC_{legacy}'s scalability, when TLC_{legacy} is run on today's hardware with high numbers of processes. Figure 4.1.1 shows that TLC_{legacy}'s scalability peaks at approximately 16 workers (indicated by the dashed vertical line), at which point coherence (see section 3.2) causes TLC to slow down. Overall, TLC_{legacy} never exceeds a speedup from parallelization of $Z_{peak} \leq 2.5$.

What are the reasons for TLC_{legacy}'s poor scalability? The bulk operation — the *scsr*

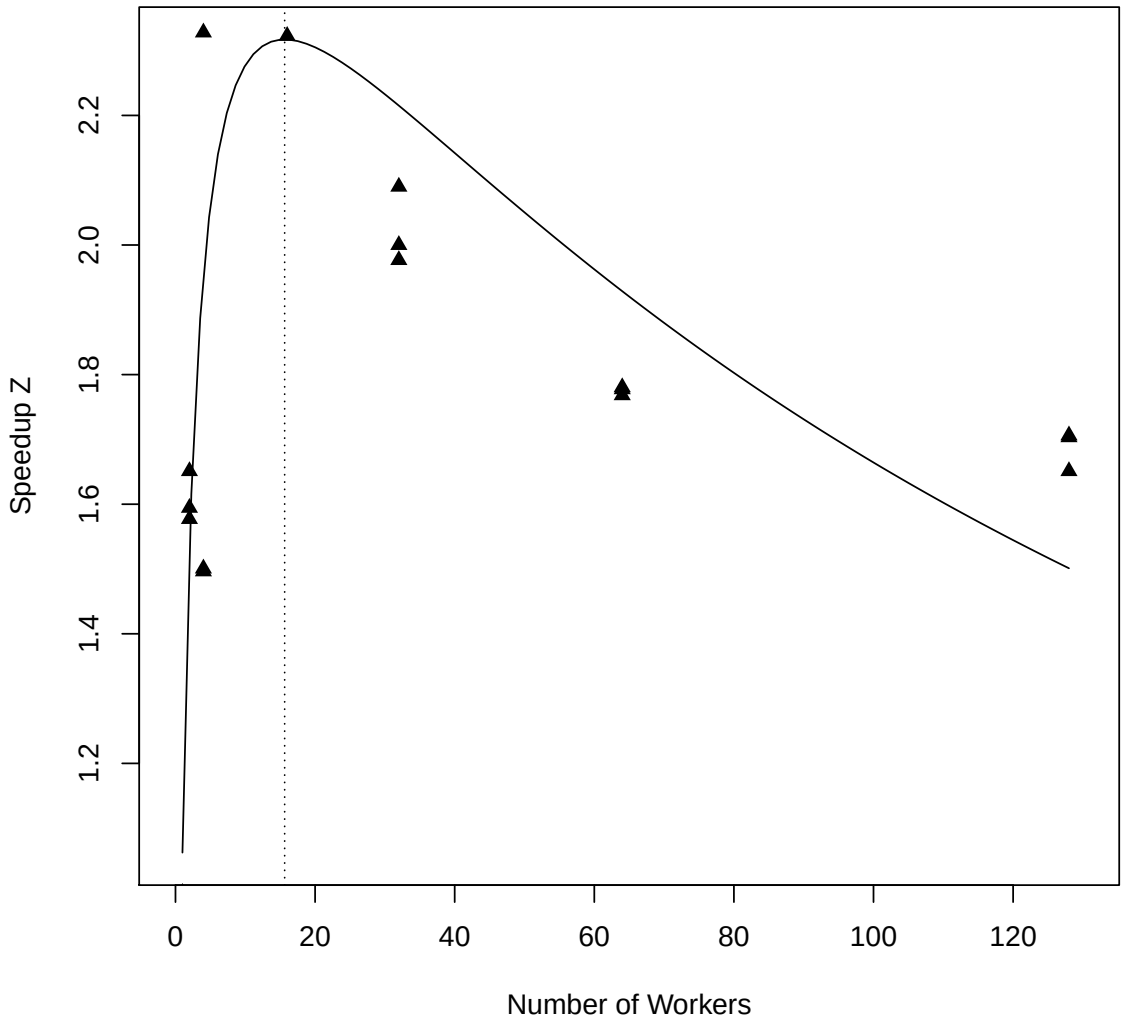


Figure 4.1.1.: Scalability of TLC_{legacy}

loop — of the model checking algorithm introduced in section 2.2.3 has only minimal contention, provided that the state graph is well-shaped. Before we loosely define well-shapedness, lets consider the opposite: a single sequence of states, s.t. each state has only one successor. It is obvious, that parallel successor generation cannot gain a speedup. Thus, for parallel successor generations, we assume state graphs to have a high branching factor and a tree-like shape [compare Pelánek, 2008]. With a well-shaped state graph, successor generations can execute in parallel (compare 2.1.7).

Under the well-shaped assumption, the cause for the scalability of TLC_{legacy} can be attributed to contention and coherence exhibited by the shared data structures S, C, T (compare section 2.2.3). The eventual negative speedup confirms the presence of coherence, which section 3.2 defined to be an effect of coherence only. Clearly, a scalability benchmark of each data structure in isolation can reveal the dominant contention and coherence bottlenecks.

The obvious choice to start benchmarking is the seen set C. For the unseen set S and forest T, a well-shaped state graph allows us to devise naïve partitioning: Instead of a globally shared unseen set, each worker w can operate its own subset S_w of unseen states with the following property: $\forall S_u, S_w \in S_{all} : S_w \subseteq S_{all} \wedge S_u \subseteq S_{all} \wedge S_w \cap S_u = \emptyset$ with S_{all} being the union of all unseen sets. This partitioning exhibits no contention or coherence and thus does not hinder TLC to scale.

The same argument applies to the forest. Let $(s,t)_w$ be a pair such that t is the successor of s and w identifies the worker which generated t. The pair $(s,t)_w$ is then stored in a forest T_w local to a worker. The overall forest is constructed by traversing the path in the union of all T_w. We return to these two approaches, denoted with S_{proto} and T_{proto} in section 9.2.

Ultimately, the seen set C is TLC's only data structure that cannot be partitioned among workers. By its definition (see section 2.2.3), the seen set C stops a model checker from carrying out redundant work. Therefor, it has to be shared.

4.2. Seen Set

When the scalability of the C_{legacy} is measured in isolation, we obtain figure 4.2.1. It confirms the earlier measurement in figure 4.1.1. The C_{legacy} exhibits both contention and coherence, but contention dominates. The reason is immediately obvious from the implementation: C_{legacy} is guarded by a single, global lock.

Earlier work by Kuppe [2012], replaced coarse with fine grained locking, by applying

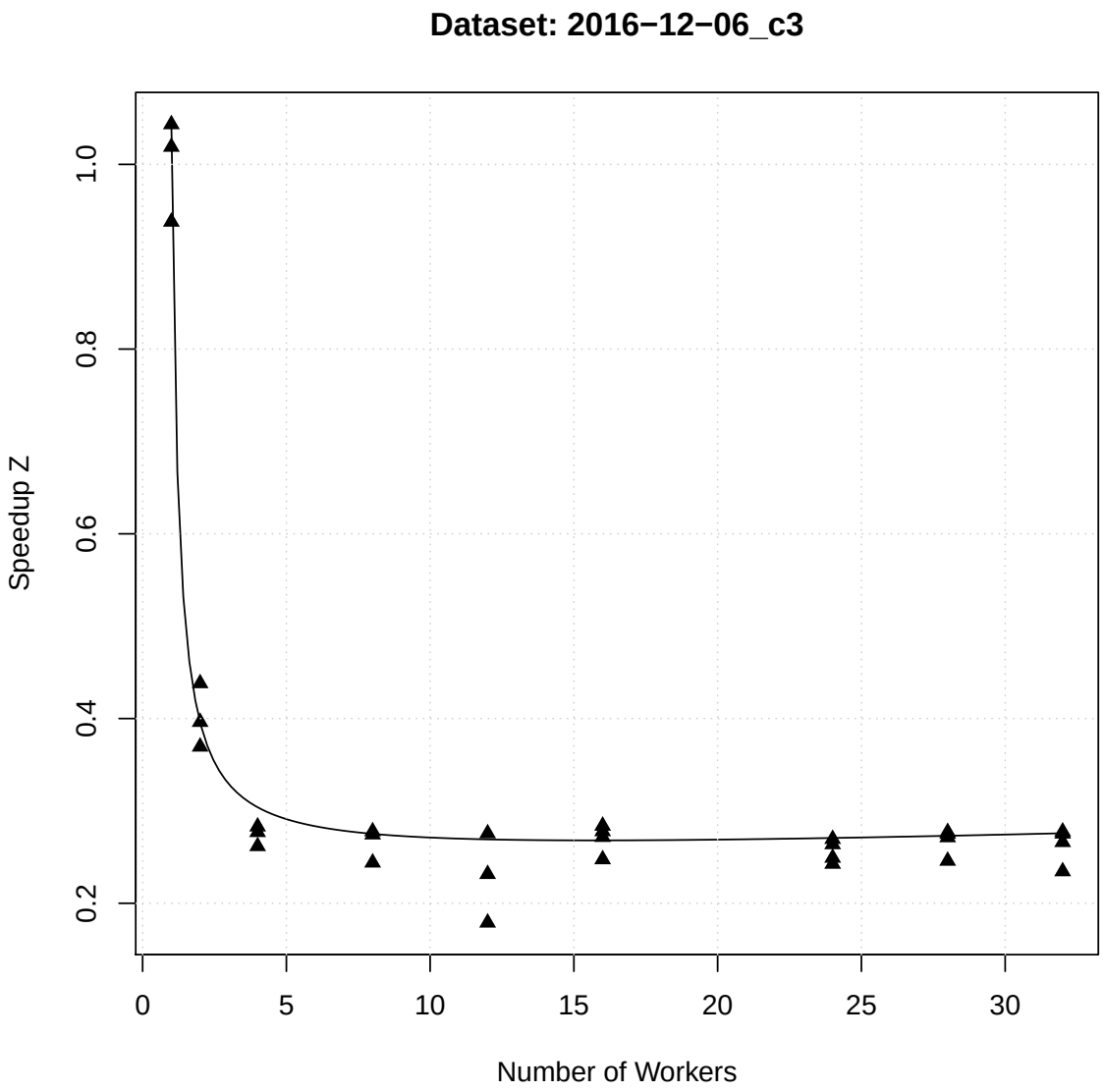


Figure 4.2.1.: Scalability of C_{legacy}

the lock-striping pattern described in section 3.1.2. Lock-striping can be directly applied to C_{legacy}. The C_{legacy} uses SC as its collision resolution strategy (compare section 3.3.1), allowing us to guard each collision bucket by a dedicated lock. At most $m \leq B$ locks are used, where B is the number of collision buckets. Lock-striping successfully eliminates contention.

However, lock-striping still does not scale infinitely. Figure 4.2.2 shows a growth in throughput with increasing lock numbers, but throughput peaks eventually. Afterwards, throughput even decreases.[1] This effect can be attributed to increased coherence, introduced by lock-striping. This assumption is plausible, since higher lock counts result in a large space requirement. In turn, this causes a higher number of cache misses per process. The effect of lock-striping on a hash table's utilization *util* however is negligible. E.g. the space requirements of 2^{23} locks is lower than one GB.

Another relevant property of C_{legacy} with regards to *util* surfaces, when C_{legacy} extends to external storage (compare section 3.3.3). Its hash function *bucketIdx* uses the k lowest fingerprint bits $fp\,[LSB, k]$, where LSB denotes the least significant bit (compare section 3.3.1 and 3.3.3). One consequence of this hash function is total disorder in the hash table t that is internal to C_{legacy}. Total disorder prohibits sorting fingerprints with the naïve scheme described in section 3.4.2. A second consequence is that sorting t in-place violates the invariant of *bucketIdx* (compare section 3.4). In-place sort moves fingerprints away from their index. Sorting t out-of-place on the other hand, doubles the space requirement of t. Consequently, utilization of t and ultimately C_{legacy} is halved.

At the implementation level, the technical design of a collision bucket can either be a linked list or an array (compare section 3.3.2). In the scope of TLC — where fingerprints are 64 bit JAVA (primitive) longs — a linked list mandates a wrapper object for each fingerprint to store the list pointers. This is prohibitive, if *util* is to be maximized. Table t is thus implemented as a jagged array (see section 3.3.2). A jagged array results in repeated memory allocations when fully occupied buckets grow. This leads to performance degradation due to excessive garbage collection.

In summary, the existing C_{legacy} and its replacement C_{stripe} suffer from the following shortcomings:

1. A 50% space overhead inherent in out-of-place sorting hash table t.
2. Non-parallel, sequential sort during eviction to external storage.
3. A performance penalty resulting from the jagged array implementation of t.

[1] The data was obtained with 128 processes, a table with size $K = 2^{35}$ and a load factor $\alpha \approx .38$.

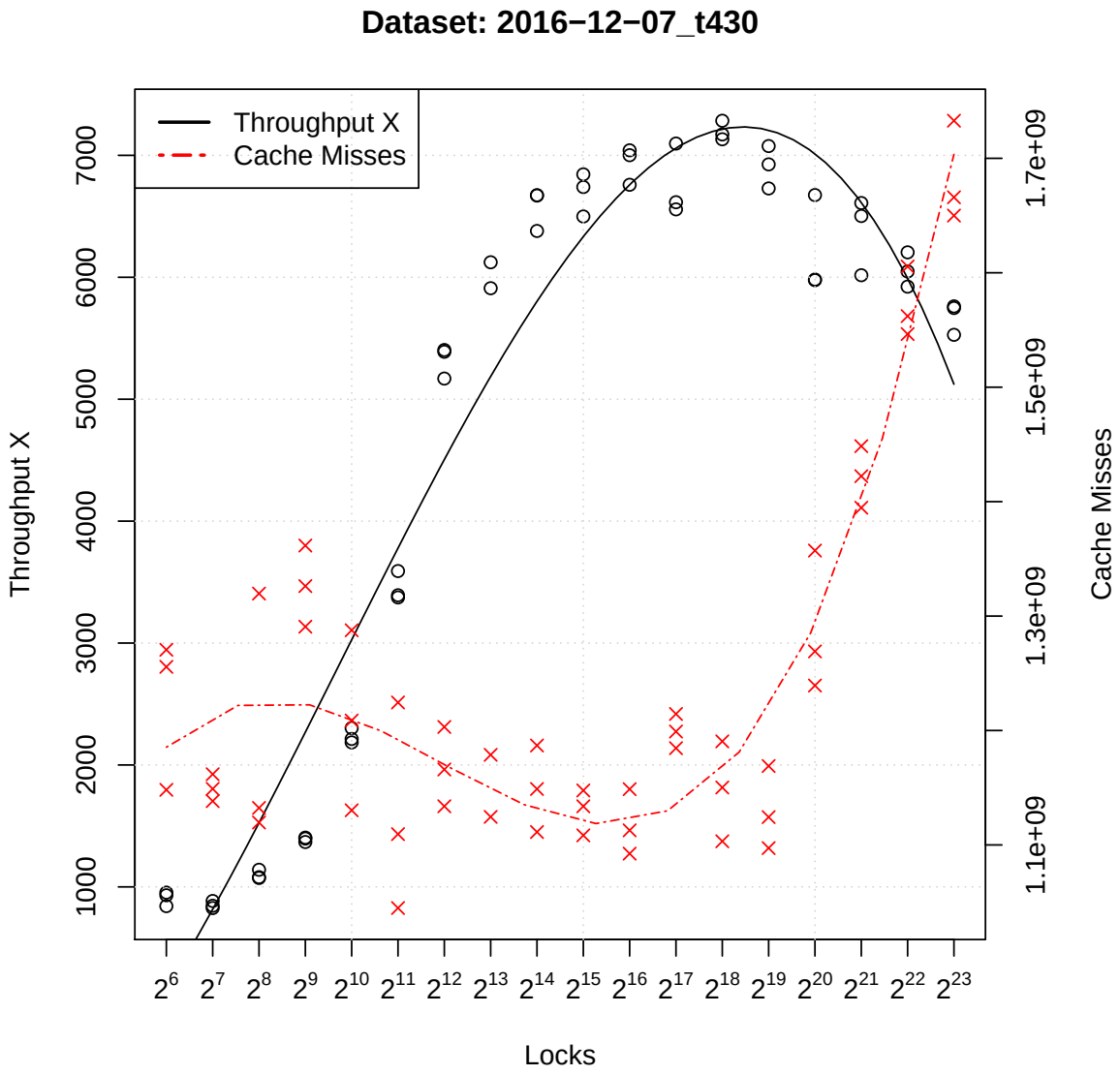


Figure 4.2.2.: Lock-striping increases coherence.

4. Limited scalability due to contention and coherence (latter with lock-striping).

5. Requirements

Based on the problems related to C_{legacy} and C_{stripe} we identified in the previous section, a list of requirements for C_{proto} are presented next. In this chapter, we present high-level requirements only, which are thus implementation independent. Implementation specific requirements can be found in section 8.1. This distinction is made in anticipation of alternative implementations, which will have unique implementation requirements but share the same high-level requirements. In this sense, it is valid to regard the requirements in this chapter as requirements for the algorithm proposed in chapter 7.

Similar to the previous chapter 4, we start with requirements that are relevant at the model checker level, before listing requirements specific to the seen set C.

5.1. Parallel TLC

Section 2.2.3 summarizes the hardware on which TLC is commonly executed. Any solution presented as part of this thesis, has to support the aforementioned environments. However, we further restrict our requirements to a symmetric resource set, meaning that the compute resources (processes) are matched by similar memory resources.

Ideally, TLC's throughput is constantly high (see section 3.2). Still, it is acceptable for short periods to slow down or suspend model checking, if thereby a higher overall throughput can be achieved. In other words, latency is not a concern. The only exception to this requirement affects models which can be checked within seconds. In this cases, TLC's startup time should be low.

Similarly, TLC's performance of non-disk-based model checking shall not negatively be influenced by disk-based model checking. In other words, support for disk-based model checking should have no upfront cost, s.t. it degrades performance of non-disk-based model checking. We expect the majority of users to not need disk-based model checking. In consequence, this leads to a high utilization (*util*) requirement for C_{proto}.

5.2. Seen Set

A replacement algorithm C_{proto} for C_{legacy} and C_{stripe} can be tailored to the specific requirements of TLC. As such, the algorithm does not have to rely on a generic hash table algorithm for C_{proto}'s internal t. It suffices, when t supports storing fingerprints only. Furthermore, the table t needs not support all operations normally supported by generic hash tables. Only a concurrently executable find-or-put operation with asymptotically constant time complexity is required. More specifically, no element removal is needed. Limiting the set of operations will simplify the new algorithm.

The new hash table has to have a memory utilization *util* that exceeds that of the existing C_{legacy} and C_{stripe}. This mandates to sort fingerprints in-place (compare section 4.2). The average throughput — even under a high load factor of $\alpha \geq 0.8$ — must be above a throughput achievable with disk-based model checking. Again, throughput has to at least match the one of C_{legacy} and C_{stripe}. The average latency should be low, which constraints (automatic) garbage collection.

The table t needs not support dynamic memory allocation. It does not have to be re-sizable. A static allocation of the assigned memory is acceptable. However, the hash table must be expandable to external storage (compare section 3.3.3) for TLC to continue to support disk-based model checking. Eviction can temporarily suspend the find-or-put operations. Forward compatibility, i.e. sorting fingerprints larger than 64 bits — likely multiples of 64 bits — should be considered as part of the algorithm design.

The new algorithm must scale with the number of processes and memory. More precisely, the static allocation, the find-or-put operation, as well as extension to external storage must all scale. As a consequence, initialization and extension to external storage should be done in parallel. The number of processes is fixed and known in advance. Scalability has to be quantified (compare section 3.2) based on empirically obtained scalability results.

The correctness of the new algorithm — formulated as safety properties and liveness properties — must be verified. We require that the core parts of the implementation must be verified and auxiliary parts must be validated.

The new algorithm must be compatible with the distributed fingerprint set described in section 2.2.3.

6. Related Work

This chapter discusses, to what extent existing work and known algorithms solve the previously identified problems (see chapter 4) and satisfy the requirements above (see chapter 5).

Before we discuss the seen set C specifically though, we broaden the scope to other explicit state model checkers in search for scalability improving variants of the original BFS algorithm (compare listing 1). Especially, section 6.1 outlines approaches similar in concept, to what we proposed in chapter 4 as S_{proto}. However, the authors also address load-balancing concerns.

6.1. Parallel Model Checking

For the SPIN model checker, Holzmann et al. [2011] suggest to speed up explicit state model checking on multi-core systems with large quantities of memory, or distributed systems, by transforming the model checking algorithm into an embarrassingly parallel algorithm (compare section 2.1.7). Following the definition in section 2.1.7, an embarrasingly parallel model checking algorithm executes its workers in isolation, s.t. it is acceptable to carry out redundant work. Technically, such an algorithm can be compared to the traditional algorithm outlined in listing 1, but without a shared seen set C. This change eliminates both contention and coherence. Essentially, n instances of a model checker execute concurrently with different start parameters, assuming that each instance verifies a disjoint subgraph of the state graph. Fittingly, Holzmann et al. call this a *swarm*. However, the correct start parameters to optimally partition the state graph are unknown a-priori, which turns the algorithm into a probabilistic approach. An exhaustive search of the complete state graph therefor still requires the same amount of time like with the original explicit state model checking algorithm. This approach is thus only applicable, if the size of the state graph renders an exhaustive search infeasible anyway.

As an optimization to SPIN's classic DFS algorithm, Holzmann [2008] introduces

stack slicing to improve concurrency of DFS up to linear speedup, but only for safety checking. Stack slicing assigns unexplored states to an idling worker, whenever the size of the stack of unvisited states reaches a threshold. With stack slicing, only a fragment of the counterexample can be constructed though, unless a similar data structure to TLC's forest T is maintained. SPIN does not maintain a full forest. It is therefor unknown, to what extent scalability is degraded when T is maintained.

In Holzmann [2012], the author describes an optimization to SPIN's BFS algorithm, to reduce coherence on SPIN's state queue (compare unseen set S in section 2.2.3). The global queue is (logically) partitioned among workers, to grant each worker exclusive access. For load balancing, each successor state is randomly added to one of the state queues. Synchronization — where necessary — is achieved by using CAS. *Strict* BFS is maintained by globally tracking the height of the BFS tree at which the workers operate.

Laarman et al. [2011] describe the concept of *pseudo*-BFS exploration, to reduce contention on the queue S, which achieves superior scalability. With pseudo-BFS, workers need not synchronize to constantly operate at the same height. Instead, workers explore at different height of the state graph. Load balancing only becomes necessary, when a worker runs out of unexplored states. Pseudo-BFS, introduces randomness into BFS, depending on how workers are scheduled. As a consequence, the LTSMIN model checker cannot guarantee to find the shortest counterexample. In practice however, under the assumption of the well-shapedness of state graphs (compare section 4.1) and a fair scheduler, an approximation of the shortest counterexample can be expected to be constructed.

6.2. Seen Set

Notably, SPIN differentiates itself from other model checkers by its adoption of BLOOM filters as a replacement of the seen set C [compare Holzmann, 2005]. A BLOOM filter is a probabilistic data structure, which supports insertion and lookup operations in constant time [see Bloom, 1970, Putze et al., 2009]. It does not support element removal. As a result of collisions, a BLOOM filter lookup can produce a false positive to a membership query. In order to achieve a sufficiently low false positive probability, Holzmann suggests to use multiple BLOOM filters. However, multiple BLOOM filters require multiple hash functions. Thus, compared to hash tables, BLOOM filters trade a reduction in space, for an increase in runtime. Holzmann argues, that two hash functions result in a sufficiently low emission probability.

Barnat et al. [2015] describe the design of DiVinE's hash table, which has been optimized to scale with the number of processes. Scalability tests went as far as 32 processes. The table's collision resolution strategy is OA. The design is different to the algorithm presented as part of this thesis. First and foremost, the authors argue that dynamic allocation provides a better overall resource usage. Dynamic allocation allows the table to grow with the number of elements inserted. On one hand, this leaves additional space to the unseen set S, while the hash table is still small. On the other hand, this is relevant for model checkers without support for disk-based model checking. Here, model checking terminates, when the hash table reaches its maximum size, even if the state graph has not been fully generated and verified. Thus, a user has to correctly estimate the size of the state graph and choose a sufficiently large size for the seen set C a-prior. If underestimated, model checking ends without verifying the complete state graph. If the environment supports a CAS operation, which matches the fingerprint length, the table is lock-free. Otherwise, each position is guarded by a (single-bit) spin-lock. To handle concurrent insertion and table resizing, each insert operation has to finally check whether the table has been resized. If the table has been resized concurrently, the insertion operation restart. The hash table does not support an extension to external storage, thus does not support disk-based model checking.

The work in Laarman et al. [2010] is most relevant to this thesis. The authors present a verified hash table algorithm for storing fixed-sized fingerprints and associated data: the corresponding state. Scalability has been shown for up to 48 processes. Collision resolution employs OA. The probing sequence is called *walking-the-line*, which is a combination of linear probing and double hashing to reduce clustering. The hash table's set of operations is limited to find-or-put. The table cannot be resized. The algorithm to find-or-put fingerprints is lock-free using CAS. However, in order to atomically write a fingerprint and its corresponding state, a spin-lock guards the table's position. This changes the overall algorithm to be wait-free. Similar to the algorithm presented by Barnat et al. and in contrast to this thesis, the table does not support being extended to external storage.

To conclude this chapter, none of the discussed solutions and algorithms fit our bill of requirements. Especially, no algorithm has support for disk-based model checking with its unique set of attached requirements. The resizable hash table proposed in Barnat et al. [2015] is primarily of interest for model checker without support for disk-based model checking. With TLC, underestimating the size of the state graph does not lead to termination, but premature disk-based model checking. With regards to Bloom

filters, we do not wish to further weaken the completeness guarantee of TLC (compare section 3.3.3). The results and design decisions of Laarman et al. [2010] validate our own findings. However, we have no need to store associated states in TLC's hash table. The proposed load-balancing techniques in section 6.1, influence our design of S_{proto}. More on this can be found in chapter 11.

7. Algorithm

Contrary to the previous chapters, which discuss TLC's scalability at both, the model checker level and the level of the seen set C, the following two chapters focus on the seen set C exclusively. We do not look at S_{proto} or T_{proto}, but propose an algorithm and its implementation for C_{proto}. The algorithm takes the sections related to the seen set C of the Problem Analysis, the Requirements and the Related Work into account. The section 9.2 further below however, combines C_{proto} with S_{proto} and T_{proto} and discusses the scalability of what we introduced as TLC_{proto} in chapter 4.

If a reader's interest is limited to the algorithm, this chapter can be read by itself, without reading the implementation chapter 8. Note though, that the development of the algorithm followed an iterative process, s.t. we specified a version of the algorithm in PlusCal and verified its correctness. Afterwards, we created a corresponding implementation and measured its performance and scalability. Depending on the outcome of the measurement, we altered the specification and changed the implementation correspondingly. We repeated the specification, implementation and measurement cycle several times and thus obtained different versions of the algorithm. In other words, the specification driven development process allowed us to explore different designs with an acceptable overhead. Thus, our own findings confirm the benefits of the TLA^+ based development process as stated in Newcombe et al. [2015].

We consider the version of the algorithm proposed to be the optimal solution within the window of constraints imposed by today's hardware. With future hardware generations — especially a more powerful CAS operation (compare section 3.1.1) — other versions are expected to provide better scalability. Other versions of the algorithms can be found alongside the implementation.

7.1. Design

To address the problem of insufficient utilization *util*, observed with the existing C_{legacy} and C_{stripe}, the algorithm for C_{proto} replaces the SC collision resolution strategy with

OA (compare section 3.3.1). OA removes the overhead inherent in a jagged array as discussed in section 3.3.2. With OA, the hash table — internal to C_{proto} — requires no meta-information. Replacing SC with OA thus eliminates one out of two causes for C_{legacy}'s unsatisfactory *util*.

OA requires to choose a probing sequence. The probing sequence selected for C_{proto} is LP. Why did we choose LP as the probing sequence, when LP leads to clustering (see section 3.3.1)? To answer this question, lets recall the second cause for the low *util* observed with C_{legacy}. Sorting C_{legacy} out-of-place, has an $\mathcal{O}(n)$ auxiliary space requirement attached to it. In order to sort fingerprints in-place however, a probing sequence is needed, which is invariant under sorting: Let t and t' denote the unsorted and the sorted hash table respectively and let fp be a fingerprint, $idx(fp, 1..L)$ — as defined in section 3.3.1 — is invariant, iff $\forall fp \in t : \exists p, p' \in 1..L : t[idx(fp, p)] = t'[idx(fp, p')]$. In other words, fingerprints remain within their probing sequence in the sorted hash table t'.

What does it constitute for $idx(fp, 1..L)$ with LP to be invariant under in-place sorting? The section 3.3.3 showed that $idx(fp, 1..L)$ uses a subsequence of $2^k = K$ fingerprint bits as the fingerprint's primary position, denoted by $idx(fp, 1)$. Clearly, if we choose $fp[MSB, k]$, s.t. $idx(fp, 1..L)$ uses the k numerically most significant bits (MSB), the following invariant for $idx(fp, 1..L)$ holds: $\forall pos \in 1..K : pos < idx(fp, 1) \Rightarrow t[pos] < t[idx(fp, 1)]$. In other words, all fingerprints located at lower positions of t than fp's primary position $idx(fp, 1)$, are numerically lower than the numerical value of fp. Consequently, under the assumption of a fully occupied t, s.t. $n = K$, an in-place sort of t does not have to move fp to a position that is lower than its primary position.

For the above to hold for t with $n \leq K$ (a t with empty positions), we simply restrict the in-place sort, s.t. it is not allowed to swap fingerprints with empty positions. In other words, a t with m empty positions is projected to $\prod_{empty}(t)$. The t and $\prod_{empty}(t)$ are identical, except that all empty positions are discarded and $K' = K - m$. We denote this restriction with R_1.

Next, we show that fp never moves past its last alternative position $idx(fp, L)$. For fp to move past $idx(fp, L)$ in t', more than L fingerprints, whose primary position is also $idx(fp, 1)$, have to be present in t. Obviously, this contradicts the definition of $idx(fp, 1..L)$ in section 3.3.1. Maximally L fingerprints, with an identical primary position, are present in t.

To summarize the above paragraphs, the choice of fingerprint bits $fp[MSB, k]$ combined with LP, provide an upper bound for the disorder of fingerprints (compare sec-

tion 3.4.1). This bounded disorder of t is a fundamental property of the proposed algorithm.

Algorithm 3 : Index calculation excerpt of Open Addressing specification

```
1 bitshift(fp, p) ≜ LET k ≜ CHOOSE k ∈ 1 .. K : 2^k = K
2                   IN   mod(shiftR(fp, k − 1) + 1 + p, K)
3 rescale(k, maxF, minF, fp, p) ≜ LET f ≜ (k − 1) ÷ (maxF − minF)
4                                 IN   mod((f ∗ (fp − minF + 1)) + p, k)
5 idx(fp, p) ≜ IF ∃ n ∈ 1 .. K : 2^n = K THEN bitshift(fp, p)
6              ELSE rescale(K, max(fps), min(fps), fp, p)
```

This property is found in the PlusCal specification of the algorithm in appendix B.2. The *idx* function (alg. excerpt 3) defines LP as its probing sequence. Additionally — knowing that the modulo operation *mod* is mathematically more expensive — it defines bitshifting to be an optimization, if the size of the table K is a power of two (alg. 3 line 1 and 2). The *shiftR* operator right-shifts the bits of the first operand by the number of positions of the second operand. For all other values of K, a fingerprint is scaled to the size of t (alg. 3 line 3 and 4). We discuss the rational behind the modulo operation in algorithm 3 on line 2 after the next paragraph.

The section 5.2 mandates that C_{proto} achieves a high utilization which is related to a high load factor α. Clearly, α is restricted by the upper bound L of the probing sequence. On average, the length of the probing sequence L is proportional to α. A lower bound L thus leads to a lower α. We study the relationship of α and L in section 7.2. In this section, it suffices to show, that L — in contrast to α — is a direct measure for the maximum disorder in t. An upper bound for the disorder is required, to sort t with the parallel scheme proposed in section 3.4.2.

Another reason, why the algorithm indirectly restricts α via L is scalability. Maintaining α would require to count the number of fingerprints in t. This counter — even if striped (compare section 9.1) — exhibits both high contention and coherence. All workers read it on each find-or-put and increment the counter on every successful find-or-put operation. With the upper bound L on the other hand, the algorithm does not need the counter. With increasing α, the probability of a find-or-put operation to hit L increases. Eventually, a worker hits L and starts eviction subsequently. The section 7.1.2 goes into more detail of how and when eviction is triggered.

Earlier, we promised to discuss the *mod* operator in algorithm 3 on line 2. So far, we

ignored the fact that our mental model of t, with a collision resolution strategy of OA, violates the hash table property stated in section 3.3. A hash table, which is assumed to be a sequence, does not provide a uniform collision probability with OA and LP. Towards the end of t, with end defined by the direction of the probing sequence, the probability of collisions increases linearly up to be L times higher for the last position $t\,[K]$. Therefor, it is common to define t, s.t. it is a circular list. A circular list has the property, that the collision probability is uniformly distributed, even for the positions we just defined to be at the end of t.

In the algorithm, the circular list is reflected by the *mod* operation of *bitshifting* and *rescale*. The probing sequence for a fingerprint fp, whose primary position is distanced less than L positions from position K, wraps around. A tail of fp's probing sequence points to positions that are lower than $idx(fp, 1)$. Thus, we can classify the set fps of all fingerprints into two disjunct sets $\overset{\circlearrowleft}{fps}$ and $\overrightarrow{fps}$: those whose probing sequence wraps and those whose probing sequence does not wrap.

A circular list design demands to revisit our earlier claim, i.e. LP is invariant under in-place sort. Earlier, we restricted in-place sort s.t. it is not allowed to swap fingerprints with empty positions. For LP to remain invariant, we state an additional restriction R_2: Fingerprints in $\overset{\circlearrowleft}{fps}$ in the range $t\,[1]$ to $t\,[L]$ must not be swapped with fingerprints in $\overrightarrow{fps}$. In other words, the (logically) circular list t is up to position L disentangled into a non-circular sequence once sorted. For the above restriction to hold, the minimal length K of the table is constrained to $2 * L \leq K$. Note, that the section 7.2 shows that the minimum length constraint is not relevant in practice. The section 7.1.2 provides more detail on the additional restriction. Especially, it discusses the effects of this restriction on Insertion Sort. First though, we shift the discussion from sorting to lock-freedom.

The new algorithm is designed to be lock-free (compare section 3.1.3) with regards to the two operations find-or-put and eviction to external storage individually. Both operations however need to be mutually exclusive; during eviction, find-or-put calls are suspended. Vice versa, no eviction is possible while workers execute find-or-put operations.

Designing the algorithm to allow eviction and find-or-put to happen concurrently, would require DCAS or CASN — to atomically swap two adjacent or two non-adjacent fingerprints with e.g. Insertion Sort or Quick Sort. As we discussed in section 3.1.1, neither DCAS nor CASN are provided by today's hardware. Using DCAS or CASN in the specification in appendix B.2, would thus render the algorithm unimplementable.

The following pages detail the find-or-put and eviction operations.

7.1.1. Find-or-Put without External Storage

Prior to the initial eviction, the external storage is empty. Thus, a fingerprint match can only be present in primary memory. Therefor, the find-or-put operation (see algorithm 4) directly puts the new fingerprint fp into t, iff the current position pos — given by $idx\,(fp, index)$ — is $empty$. In other words, find-or-put relies on the invariant, that an empty table position is a mark for all higher positions to be no match. This is obvious, because with LP, a possible match would have been put at the empty position. Generally, following an empty position, no higher position of the probing sequence can produce a match: $t\,[idx\,(fp, index)] = empty \Rightarrow \forall i \in index..L : fp \neq t\,[idx\,(fp, i)]$.

If another worker tries to concurrently put (compare algorithm 2) its fingerprint at $t\,[pos]$, CAS guarantees that only one worker succeeds (algorithm 4 line 6 and 7). The losing worker repeats find-or-put at $t\,[pos]$, to check if the winner put a match (alg. 4 line 9).

Iff $t\,[pos]$ is found to be occupied (non-empty), the worker compares the fingerprint at $t\,[pos]$ (alg. 4 line 12). Unless a match is found (alg. 4 line 15), the worker repeats find-or-put with the next higher index in the probing sequence ($index + 1$), until the upper bound of the probing sequence L is reached.

Algorithm 4 : Find-or-put excerpt of Open Addressing specification

```
1    insrt:  while (index < L){
2                expected := table[idx(fp, index)] ;
3                if (expected = empty ∨
4                    (expected < 0 ∧ expected ≠ (−1) ∗ fp)) {
5    cas:            CAS(result, idx(fp, index), expected, fp) ;
6                    if (result){
7                       goto pick  Match/Success
8                    }else {
9                      goto insrt  Retry current position
10                   }
11               } ;
12   isMth:      if (isMatch(fp, idx(fp, index), table)){
13                 goto pick  Found
14               }else {
15                 index := index + 1 ;   Try next position
16               } ;
17           } ;
```

Obviously, the find-or-put operation described above is lock-free. In this section, we

do not yet concern ourselves with the second disjunct of the first if-clause at line 4. Instead, we defer its discussion to section 7.1.3.

7.1.2. Initial Eviction

The first worker to hit the upper bound L, flips the global variable *evict* and waits for the remaining workers to suspend (algorithm 5, line 11 to 16). The other workers detect the pending eviction during the next invocation of find-or-put and suspend subsequently (alg. 5 line 4 to 8). This variant is preferable over a shared lock (compare section 3.1.2), where the first worker to hit L, converts its shared lock to an exclusive one. The overhead of shared lock acquisition outweighs that of the read of *evict*.

In the TLA$^+$ specification, the set of workers detect the pending evictions upon the next read of *evict*. An implementation however, is free to use a relaxed consistency model — such as eventual consistency — to reduce coherence of *evict*. This comes at the price of an extended period of time, where an increasing number of workers is prevented from making progress, until the last worker suspends. However, eviction occurs sufficiently infrequently for such an optimization to be viable.

An even simpler design variant is to omit *evict* entirely and wait for all workers to individually hit L. Under the assumption of a uniform distribution of fingerprints, the set of workers can be expected to hit L almost simultaneously. However, we did not choose this variant based on the assumption that infrequent writes allow NUMA architectures (see section 3.1) to efficiently cache *evict*.[1]

With all workers suspended, eviction sorts table t and afterwards flushes the fingerprints to the external storage. For large values of K — the size of the table — the runtime and scalability of eviction is unacceptable though. Thus, we require eviction to execute concurrently. In section 7.1 we show, that t has a bounded disorder L. This property can be exploited to partition t among workers and sort the partitions concurrently with the scheme described in section 3.4.2 with $dis = L$.

Assuming a sufficiently small L, Insertion Sort accomplishes best case complexity, while being simple to implement and equipped with negligible constant costs. Additionally, it provides a good cache locality. However, in accordance with the restriction R_1 introduced in section 7.1.1, we require a modified version of Insertion Sort to prevent find-or-put from producing false negatives. The comparison on line 3 of the *compare* operator in excerpt 6 accounts for the restriction R_1.

[1] We assume an invalidation based cache coherence protocol such as MESI.

Algorithm 5 : Synchronization to switch between find-or-put and eviction

```
1  variable evict = FALSE ;
2      process (p ∈ Writer){
3      put:      ...
4                if (evict){
5                    waitCnt := waitCnt + 1 ;
6      waitEv:       await evict = FALSE ;
7      endWEv:       waitCnt := waitCnt − 1 ;
8                    goto put
9                } ;
10     ... find-or-put operation here. tryEv only executed if find-or-put unsuccessful.
11     tryEv :   if (evict = FALSE){
12                   evict := TRUE ;
13     waitIns:      await waitCnt = Cardinality(Writer) − 1 + Cardinality(Reader) ;
14                   call Evict() ;
15     endEv:        evict := FALSE ;
16                   goto put ;
17               }
18 }
```

Algorithm 6 : Comparison operator used by INSERTION SORT of Open Addressing specification

```
1  wrapped(fp, pos) ≜ idx(fp, 0) > mod(pos, K)
2  compare(fp1, i1, fp2, i2) ≜
3                    IF fp1 ≠ empty ∧ fp2 ≠ empty
4                     THEN IF wrapped(fp1, i1) = wrapped(fp2, i2)
5                             THEN IF i1 > i2 ∧ fp1 < fp2 THEN − 1 ELSE 1
6                             ELSE IF i1 < i2 ∧ fp1 < fp2 THEN − 1 ELSE
7                                  IF i1 > i2 ∧ fp1 > fp2 THEN − 1 ELSE 0
8                     ELSE 0
```

A second modification is related to the circular table layout chosen to achieve a uniform collision probability (see section 7.1). We consider three cases:

1. The first if-clause considers two fingerprints $fp1, fp2 \in fps$, which — according to the operator *wrapped* in excerpt 6 on line 4 — both either wrapped or did not wrap. In this case, a basic comparison is done (see 6 on line 5), s.t. the numerically lower fingerprint is swapped to the lower position of the two fingerprints.

2. The comparison on line 6 handles two fingerprints $fp1, fp2 \in \overset{\circlearrowleft}{fps}$. Due to the fact, that this is the else-branch of the previous if-clause, one of $fp1, fp2$ is known to have wrapped. Thus, if the lower positioned fingerprint is numerically lower, it is swapped with the other.

3. Finally, line 7 of excerpt 6 handles two fingerprints, with one in $\overrightarrow{fps}$ and the other in $\overset{\circlearrowleft}{fps}$. The fingerprints swap, iff the fingerprint in $\overset{\circlearrowleft}{fps}$ is at a higher position than the fingerprint in $\overrightarrow{fps}$. Consequently, all wrapped fingerprints in $\overset{\circlearrowleft}{fps}$ form a cluster at the beginning of t and all fingerprints in $\overrightarrow{fps}$ are moved to higher positions of this cluster.

The comparison in case 3 contradicts the previously stated restriction R_2 in section 7.1, s.t. two fingerprints from $\overrightarrow{fps}$ and $\overset{\circlearrowleft}{fps}$ will not be swapped. First, we discuss, why the contradiction of R_2 is necessary. Then we show, why LP still remains invariant under in-place sorting with the modified Insertion Sort.

Let $fp_a, fp_b \in \overset{\circlearrowleft}{fps}$ and $fp_x \in \overrightarrow{fps}$. We assume fp_a to have wrapped and to be numerically lower than fp_b. We further assume that fp_x is at a position lower than fp_a. In other words, fp_x is in-between fp_a and fp_b. Obviously, in order for t to be sorted, fp_a and fp_b have to be swapped. In this scenario, the comparison in case 1 and case 2 do not apply. Insertion Sort only compares adjacent elements and thus neither compares nor swaps the fingerprints fp_a, fp_b. With the comparison in case 3 however, fp_a and fp_x swap places which eventually results in a comparison and a swap of fp_a and fp_b.

Clearly, swapping fp_x and fp_a does not invalidate our invariance claim above, because the fp_x can at most be moved to its $idx\,(fp, L)$ position. This follows from the earlier argument made in the scope of a non-circular table.

Contrary to B.2, the specification B.3 verifies this scenario in isolation from lock-freedom.

With regards to standard Insertion Sort, both modifications are limited to the comparison operator of the nested while loop (alg. 7 line 5 and 6). Thus, the specification

of INSERTION SORT resembles that of standard INSERTION SORT(see alg. 7).

Algorithm 7 : Modified INSERTION SORT of Open Addressing specification

```
1  procedure Evict()
2      variables ei = 1, ej = 1, lo = 0 ; {
3      strIns:      while (ei ≤ K + L){
4                       lo := table[mod(ei + 1, K)] ;
5      nestedIns:       while (compare(lo, mod(ei + 1, K),
6                                      table[mod(ej, K)], mod(ej, K)) ≤ −1){
7                         table[mod(ej + 1, K)] := table[mod(ej, K)] ;
8                         ej := ej − 1 ;
9                         if (ej = 0){goto set}
10                      } ;
11     set:             table[mod(ej + 1, K)] := lo ;
12                      ej := ei + 1 ; ei := ei + 1 ;
13                  } ;
14                  Write to external storage here.
15     rtrn:        return ; }
```

INSERTION SORT runs in parallel, with the scheme proposed in section 3.4.2. The partitions of t are of equal size to evenly distribute the workload across workers. Ideally though, partition borders are placed at empty positions. As shown in section 7.1.1, an empty position (or generally an *isMarked* position as introduced below) guarantees, that all fingerprints at higher positions are numerically larger. In other words, $\forall pos \in 1..K : pos = empty \Rightarrow \forall pos_{lower} \in 1..(pos-1) : \forall pos_{higher} \in (pos+1)..K : t[pos_{lower}] < t[pos_{higher}]$. This invariant implies, that iff partition borders are placed at empty positions, the second pass — the merge phase — of the naïve parallel sort scheme discussed in section 3.4.2 is unnecessary. We continue to create partitions of equal size though, because we do not yet know how to find empty positions in constant time.

Once t is fully sorted, the set of workers — reusing the previously chosen partitions — write the sorted fingerprints in t to the external storage in parallel and set the fingerprint's m-bits (compare section 3.3.3). Afterwards, each evicted fingerprint in t is marked as evicted. The TLA$^+$ operator is defined to be $isMarked(pos, table) \triangleq table[pos] < 0$.

A parallelized write scales with the number of disks available. We even expected it to outperform a sequential write to a single disk, because a set of processes never makes the disk wait for data. We did not model the parallelized external storage write, due to

its simplicity. Instead, algorithm 8 writes the fingerprints sequentially and atomically — as evident by the single label *flush* — into external storage.

Algorithm 8 : Eviction writes fingerprints to external storage

```
1     flush:   while (ei ≤ K + L){
2                  lo := table[mod(ei, K)] ;
3                  if (lo ≠ empty ∧
4                      lo > largestElem(newexternal) ∧
5                      ((ei ≤ K ∧ ¬wrapped(lo, ei)) ∨
6                       (ei > K ∧ wrapped(lo, ei)))){
7                      Copy all smaller fps than lo from
8                      secondary to newexternal.
9                     newexternal := Append(newexternal ∘
10                         subSeqSmaller(external, newexternal, lo), lo) ;
11                     Mark table[mod(cpy,table)] as being
12                     written to external.
13                    table[mod(ei, K)] := lo ∗ ( − 1) ;
14                 } ;
15                 ei := ei + 1 ;
16             } ;
17              Append remainder of external to newexternal and
18              assign newexternal to external.
19             external := newexternal ∘
20                         subSeqLarger(external, newexternal) ;
21             newexternal := ⟨⟩ ;
```

7.1.3. Find-or-Put with External Storage

In the presence of (non-empty) external storage (algorithm 9 line 1), the work carried out as part of find-or-put increases. Where a single loop used to be sufficient to find a match for the fingerprint *fp* or to put *fp* (see above), find-or-put — in the worst case — traverses the probing sequence twice now. The first loop (line 2 to 16 of algorithm 9) searches for a match of *fp* in t. If successful, find-or-put simply returns success (alg. 9 line 4). Otherwise, the loop continues to search the alternative positions defined by the probing sequence.

If the search remains unsuccessful, find-or-put directs search to external storage (alg. 9 line 17 to 22) where it runs binary search (compare section 3.3.3). If the binary search also fails to produce a match, the fingerprint is known not to be a member of C_{proto}.

The second while loop thus attempts to put fp into t (see excerpt 4 above). However, the while loop reuses work of the first loop. The first loop determines the first open position and stores its relative *index* in the probing sequence (alg. 9 line 6 to 12). With external storage, an open position is either an *empty* one or a position where its occupant fingerprint has the m-bit set (see section 7.1.2).

When the first loop finds no open position within $index..(L-1)$, the second while loop terminates immediately because of its $index = L$ check on line 24 of alg. 9: The worker hit the upper bound of the probing sequence and requests eviction correspondingly.

Algorithm 9 : Find-or-put with external storage

```
1     chkSnc: if (external ≠ ⟨⟩){
2     cntns:      while (index < L){
3                   if (isMatch(fp, idx(fp, index), table)){
4                     goto pick  Match/Success
5                   }else {
6                     if (isEmpty(idx(fp, index), table)){
7                       expected := minimum(expected, index) ;
8                       goto onSnc ;
9                     }else {
10                      if (isMarked(idx(fp, index), table)){
11                        expected := minimum(expected, index) ;
12                      }
13                      index := index + 1
14                    }
15                  }
16                } ;
17    onSnc:      if (containsElem(external, fp)){
18                  goto pick  Match in external storage.
19                }else {
20                  index := expected ;
21                  expected := −1 ;
22                } ;
23          } ;
24    insrt:  while (index < L){
25              See Insertion above.
26          }
```

Workers never hold locks despite the fact that find-or-put checks t and the external storage. Thus, find-or-put remains lock-free even with external storage. If two

find-or-put operations of the same fingerprint are interleaved, atomic CAS guarantees consistency: One worker will successfully put the fingerprint while the other finds the match. The external storage is read-only when the workers execute find-or-put operations. Hence, the first loop can be seen as a find in a read-only in-memory cache for the external storage, which — iff unsuccessfully — is followed by a actual external lookup.

7.1.4. Consecutive Evictions

Consecutive evictions are identical to the initial eviction, except that the existing external storage has to additionally be merged with the sorted t. Since both locations are sorted in the same (ascending) order, the merge is trivial. However, each eviction requires to re-merge with the complete external storage. Let N be the average number of fingerprints evicted, the first eviction unsurprisingly requires N and the second eviction $2N$ writes respectively. Starting with the third eviction, writes grow up to $3N$. Generally, the m-th eviction needs $m \times N$ writes. While asymptotically being linear, it is a noticeable constant factor.

7.2. Parameter Selection

All parameters of the algorithm discussed above are determined by the amount of space dedicated to the in-memory table t. The only exception is the length of the probing sequence L, which has to be selected with two goals in mind:

1. First, the average throughput — even under a high load factor α — has to exceed the throughput with external lookups. The performance gap between primary (table t) and external storage is known to be at least one order of magnitude. We even expect it to further widen in favor of t. Thus, we need not concern ourselves with this goal, for as long as the throughput does not degrade more than an order of magnitude.

2. Secondly, the table's load factor α should be maximized to achieve a high utilization *util*. The load factor α is proportional to L: A larger value for L on average leads to a higher α. Leaving L unrestricted, we can achieve a load factor of $\alpha = 1$. On the other hand, L is also proportional to the disorder *dis* of fingerprints in t. In section 3.4.1 we argue, that *dis* has to be sufficiently small for Insertion Sort to accomplish optimal $\mathcal{O}(n)$ asymptotic complexity. Thus, finding the sweet spot

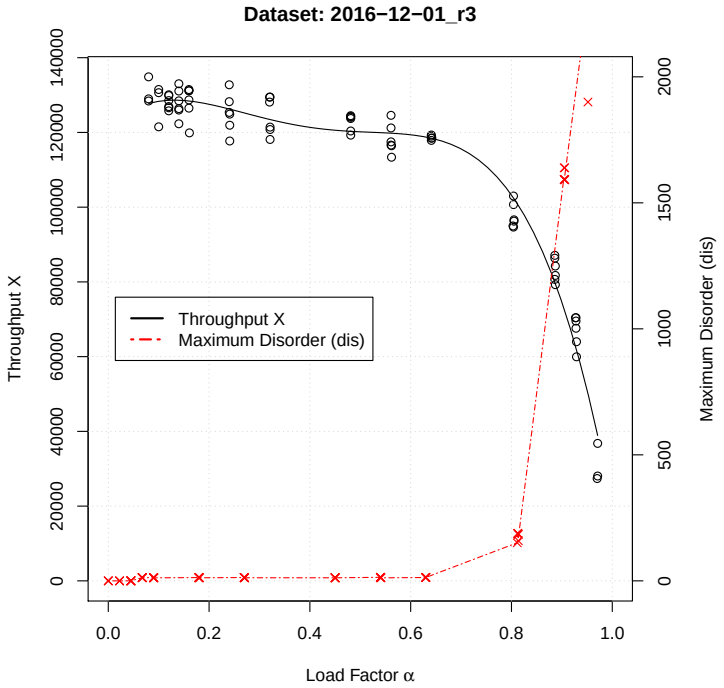


Figure 7.2.1.: Throughput with open addressing drops with increasing load factor, the disorder is inversely proportional

for L is an optimization problem with two objectives, namely maximizing α and minimizing dis.

Since we cannot determine the optimal L analytically, we approximate L from empirical data. The implementation, with which the data has been obtained, is discussed in chapter 8. However, there is no reason why the value of L is specific to our implementation. The relationship of L and α is in the data.

The graphed data 7.2.1 shows, that the maximum disorder dis of fingerprints (see section 3.4.2) starts to skyrocket at $\alpha \simeq 0.8$. On the other hand, for $\alpha \leq 0.8$, the value of dis growths slowly. Conversely, with a load factor of $\alpha \simeq 0.8$, the throughput's degradation is negligible.

In order to account for an error margin in the empirical data and to bias the implementation to higher load factors, the maximum length of the probing sequence has been set to be $L = 512$.[2]

[2]Users can elect to overwrite L with different values such as 128 by setting the system property *-Dtlc2.tool.fp.OffHeapDiskFPSet.probeLimit=128*.

7.3. Correctness

The algorithm described above and listed fully in B.2 has been model checked with TLC for various (non-trivial) models. All models have a four-dimensional parameter space, consisting of the constants K, L, *fps* and *Writer*.[3] The constant K corresponds to the table size of t and L to the length of the probing sequence (section 7.1). The constant *fps* defines the fingerprint universe $\mathcal{U}$ (see section 3.3). The constant *Writer* is defined to be a set of writers, synonymous with the workers executing the find-or-put and eviction operations.

The *Writer* set of model values has been declared to be a symmetry set. The specification and the checked properties are symmetric, s.t. a writer's identity is not relevant. The properties below and the specification do not refer to individual writers. Declaring the set to be symmetric, reduces the number of distinct states.

The specification declares an auxiliary variable called *history*, which is used by the properties below. This set records the history of fingerprints, which have been put into the seen set C by the *Writer*, thus $history \subseteq fps$. The history variable does not appear in the implementation. Abadi and Lamport [1991, page 20ff] introduce the concept of history variables.

We defined and checked the following safety properties and liveness properties, which are listed in algorithm 10. Additionally, TLC was instructed to check for deadlocks. We first discuss the safety properties, before we introduce the liveness properties:

Contains *Contains* asserts, that all fingerprints in *history* are always found in C within their probing sequence, whereas the fingerprints in subset $fps \setminus history$ are not allowed to be members of C (alg. 10 line 5). *Contains* thus guarantees, that the algorithm is indeed invariant under in-place sort (compare section 7.1). As discussed above, swapping two fingerprints does not occur atomically due to the lack of CASN. During eviction, a fingerprint $fp_a \in history$ disappears from table t temporarily in states where fp_a is swapped with another fingerprint $fp_b \in history$. Thus, for *Contains* to hold under eviction, the *contains* operator checks INSERTION SORT's variable to temporarily store fp_a in the last disjunction. This variable is called *lo* in algorithm B.2.

Duplicates *Duplicates* checks that t never contains two or more identical fingerprints (alg. 10 line 11). Note, that fingerprints are defined s.t. $empty \notin fps$. Multiple *empty* table positions are not considered duplicates. Contrary to *Contains*,

[3] *empty* is defined to be a model value which is distinct from all other constants.

Algorithm 10 : Safety properties and liveness properties of Open Addressing specification

```
1  contains(f, t, seq, Q) ≜  ∨ ∃ i  ∈ 0 .. Q : isMatch(f, idx(f, i), t)
2                            ∨ ∃ i  ∈ 1 .. Len(seq) : seq[i] = f
3                            ∨ IF f ∈ (Image(lo) \ {0}) THEN evict = TRUE
4                                                       ELSE  FALSE

5  Contains ≜  ∧ ∀ seen ∈ history :
6                    contains(seen, table, external, L)
7              ∧ ∀ unseen ∈ (fps \ history) :
8                    ¬contains(unseen, table, external, L)
9  ──────────────────────────────────────────────────────────────────────

10 FindOrPut ≜ evict = FALSE

11 Duplicates ≜ FindOrPut ⟹ LET sub ≜ SelectSeq(table, LAMBDA e : e ≠ empty)

12                              IN   IF Len(sub) < 2 THEN TRUE
13                                   ELSE  ∀ i ∈ 1 .. (Len(sub) − 1) :
14                                           ∀ j ∈ (i + 1) .. Len(sub) :
15                                             abs(sub[i]) ≠ abs(sub[j])

16 ──────────────────────────────────────────────────────────────────────

17 isSorted(seq) ≜ LET sub ≜ SelectSeq(seq, LAMBDA e : e ≠ empty)
18                  IN   IF Len(sub) < 2 THEN TRUE
19                       ELSE  ∀ i ∈ 1 .. (Len(sub) − 1) :
20                               sub[i] < sub[i + 1]

21 Sorted ≜ isSorted(external) ∧ isSorted(newexternal)

22 ──────────────────────────────────────────────────────────────────────

23 Termination ≜ ◇(∀ self ∈ ProcSet : pc[self] = "Done")

24 ──────────────────────────────────────────────────────────────────────

25 Complete ≜ ◇□(history = fps)
```

Duplicates does not hold under eviction. A fingerprint in *history* can occur in t twice due to non-atomic swaps.

Sorted The *Sorted* property verifies, that the external storage is always strictly sorted in (ascending) order (alg. 10 line 21). During eviction, the two variables *external* and *newexternal* represent the existing and the currently being written file.

The remaining properties are liveness properties:

Termination *Termination* — automatically generated by the PlusCal to TLA$^+$ transpilation (see section 2.2.1) — guarantees, that all *Writer* processes eventually terminate, i.e. reach the implicit "Done" label of the PlusCal algorithm (alg. 10 line 23).

Complete The *Complete* liveness property mandates that eventually all $\mathit{fingerprints} \in \mathit{fps}$ are members of the seen set C (alg. 10 line 25).

Due to resource constraints, we generally restricted model checking time to maximally 24 hours per model run. We define non-trivial model parameter combinations, s.t.:

- More than a single writer, thus $|\mathit{Writer}| > 1$
- At least one eviction to happen, thus $|\mathit{fps}| > K$
- Sufficient disorder, thus $L > 1$

Table 7.1 shows the parameters of the verified models and the corresponding model checking time. All models were checked on Amazon EC2 r3.8xlarge instances (see section 9.1 for details). Checking of the liveness graph has been deferred to a single, final check.[4] No violations have been found for the stated safety properties and liveness properties.

The specification does not seem to exhibit state space explosion for an increase of the K and L parameter values. The reason for this phenomenon is, that larger values for K and L, under a fixed *fps*, result in a lower load factor α. Consequently, fewer states are necessary to find-or-put the elements of *fps* into table t.

To speed up model checking, a TLC module overwrite (compare section 2.2.3) for the *shiftR* operator (see algorithm 11 line 1 to 3) was active. To do floating point arithmetic, we also revert to a TLC module overwrite for the *rescale* operator (line 5 to 21). TLC

[4] The JAVA property "tlc2.tool.liveness.Liveness.finalCheckOnly" forces TLC to check the liveness properties only once after the complete state graph is constructed.

K	L	*Writer*	*fps*	$Time_{Overall}$	$Time_{Liveness}$	Distinct States	Depth
5	2	$\{w1, w2\}$	1..8	9:25	02:24	6 807 304	203
5	3	$\{w1, w2\}$	1..8	7:05	00:59	5 016 426	175
5	4	$\{w1, w2\}$	1..8	7:27	01:37	5 040 917	140
5	2	$\{w1, w2\}$	1..9	01:19:00	15:16	46 225 095	219
5	3	$\{w1, w2\}$	1..9	56:38	10:15	29 542 762	189
5	4	$\{w1, w2\}$	1..9	45:54	07:38	23 075 716	195
5	2	$\{w1, w2\}$	1..10	11:26:00	01:58:00	316 862 498	269
5	3	$\{w1, w2\}$	1..10	08:11:00	01:13:00	198 961 652	248
5	4	$\{w1, w2\}$	1..10	05:53:00	00:44:44	132 757 645	215
7	2	$\{w1, w2\}$	1..9	18:27	03:56	10 870 283	192
7	3	$\{w1, w2\}$	1..9	12:46	02:46	7 504 379	203
7	4	$\{w1, w2\}$	1..9	14:12	02:44	8 199 637	207

Table 7.1.: TLC model checking times for various parameter combinations

does not support floating point arithmetic and silently falls back to integer arithmetic s.t. reals are truncated to naturals.

Algorithm 11 : TLC module overwrites for shiftr and rescale

```
1  public static IntValue shiftR(IntValue n, IntValue pos) {
2   return IntValue.gen(n.val >>> pos.val);
3  }
4
5  private static final int minT = 1;
6
7  public static IntValue rescale(IntValue k, IntValue maxF,
8                                 IntValue minF, IntValue fp,
9                                 IntValue p) {
10  return IntValue.gen(rescale(k.val, maxF.val,
11                              minF.val, fp.val, p.val));
12 }
13
14 static int rescale(int maxT, int maxF,
15                    int minF, int fp, int probe) {
16  float factor = (maxT - minT) / ((maxF - minF) * 1f);
17  int idx = Math.round(factor * (fp - minF) + minT) + probe;
18  while (idx > maxT) {
19    idx = idx % (maxT + 1) + 1;
20  }
21  return idx;
22 }
```

8. Realization

"*How to program if you cannot.*" Dijkstra [1988]

8.1. Software Specification

This chapter formulates requirements and constraints towards an implementation of the lock-free algorithm in chapter 7 for the seen set C. These requirements are to be seen in addition to the high-level and language independent requirements described in section 5.2. To reiterate, those (functional) requirements were:

- Hash table with limited operation set, namely find-or-put of 64 bit long values
- No support for element removal or table resizing

accompanied by the following nonfunctional requirements:

- Not limited to primary memory
- High memory utilization *util*
- High average throughput X
- Scales with number of processes (runtime) and available memory (space)
- Correctness

Also, an implementation of the lock-free algorithm has to be usable from Java, thus is preferably implemented in Java. This requirement stems from the fact that TLC itself is implemented in Java. More specifically, as of today TLC runs on Java 1.7 onwards. However, an increase to version 1.8 — as lower bound — is acceptable because Java 1.7 has ceased to receive (public) updates since April 2015 [see Oracle, 2015].

Java 1.8 is still restricted to the synchronization primitives as outlined in section 3.1. In order to implement the algorithm in Java, the implementation can rely on the non-public API `sun.misc.Unsafe` (see section 3.1). This API exposes the CAS operation

for long values and additionally grants direct, unverified and unconstrained access to and can allocate contiguous blocks of up to 2^{63} bytes of primary memory (compare section 3.3.2). Despite being located in the vendor specific `sun.misc` namespace, most JAVA VM implementations provide this API. In environments where `sun.misc.Unsafe` is unavailable for other reasons, TLC will revert to less efficient heap-based fingerprint set implementations.

Above (see 2.2.3), we also stated that TLC targets both, commodity and server hardware. Consequently, an implementation has to be compatible with a variety of hardware architectures for which a JAVA VM is available. The most prominent hardware on which TLC is executed, is the 64 bit version of the x86 instruction set though. Thus, the implementation can choose to adopt optimizations as long as they have only minor negative effects on other architectures. No additional assumptions or requirements are to be made towards the hardware architecture.

Given that the implementation is used as part of a model checker — a formal verification tool — the implementation faces has to conform to a high quality standard. Therefor, it is vital to verify and validate the implementation. Thus, the implementation has to be modular in order to be testable.

Modularity also achieves maintainability; judging from TLC's long history, we expect the implementation to be in productive use for several years. The implementation should be forward compatible with fingerprints larger than 64 bits, possibly increments of 64 bits.

To summarize the previous paragraphs, the implementation primarily targets scalability (runtime and space) followed by performance. Still, correctness is equally important. Other nonfunctional requirements are: portability, modularity and maintainability.

The functional specification will be given informally. We will implement what is mathematically a set which we loosely define to be a collection of elements without duplicates. On top, we restrict the insertable elements to be immutable fingerprints and prohibit set members from being removed. Contrary to a mathematical set, the maximum cardinality of the set is fixed and has to be chosen a priori. An excerpt of the `FPSet` JAVA class — reflecting the functional specification — is given in the program listing 12. Its simplicity indicates that the nonfunctional requirements are the non-trivial part of this implementation.

The implementation will be a subclass of the abstract `FPSet` base class shown in program listing 12. The `FPSet` also serves as the technical interface definition for the implementation.

Algorithm 12 : The FPSet class as the functional specification of the implementation

```
package tlc2.tool.fp;

abstract class FPSet {

      public FPSet(long fixedCardinality) {/* omitted */}

     /**
      * Returns true iff the fingerprint fp is in this
      * set.
      */
      abstract boolean contains(long fp);

     /**
      * Returns true iff the fingerprint fp is in this
      * set. If the fingerprint is not in the set, it is
      * added to the set as a side-effect.
      */
      abstract boolean put(long fp);
}
```

8.2. Software Design

Based on the Software Specification and the description of the algorithm in 7.1, we will next describe the software design of the implementation.

We capture the high-level software design in the (informal) UML component diagram 8.2.1. TLC consists out of three components, which map to the data structures C, S, T introduced in section 2.2.3. The diagram can be seen as a technical representation of the schematic given in figure 2.2.3 zoomed in on the procedure to generate the state graph. The implementation — denoted in 8.2.1 by the class `OffHeapFPSet` — will belong to the `FPSet` subcomponent of the TLC component. The `OffHeapFPSet` must not have any intercomponent dependencies to guarantee modularity and testability.

The `UML` class diagram shown in figure 8.2.2 details the `OffHeapFPSet` implementation.[1] Its coloring groups related classes to facilitate comprehension.

The `OffHeapFPSet` is a subclass of `DiskFPSet`, which in turn is derived from `FPSet`. Deriving from `FPSet` asserts that `OffHeapFPSet` can act as a distributed fingerprint set (see section 2.2.3).

`DiskFPSet` encapsulates logic related to the disk file (see section 3.3.3). These methods have been omitted in the UML diagram though. The `numThreads` attribute indicates, that `DiskFPSet` supports a static number of workers (compare section 2.2.3). The attribute of type `Flusher` will be discussed below.

The class `OffHeapFPSet` provides an implementation for the `put` and `contains` API defined by `FPSet`. This implementation in turn invokes the private `memLookup` and `memInsert` methods of `OffHeapFPSet`. The `memLookup` and `memInsert` methods map directly to the PlusCal excerpt 9 line 2 to 16 and algorithm 4 respectively. We make use of a `LongAdder` — available starting with JAVA 1.8 — in `memInsert` to reduce the worker contention by using a distributed counter for the overall table count. A distributed counter synchronizes on counter reads but avoids synchronization on writes.

`OffHeapFPSet` implements the wait-free worker synchronization to switch from the find-or-put phase to eviction (see section 7.1.2) with the aid of an `AtomicBoolean` and a `CyclicBarrier`; both provided by JAVA 1.5. onwards. It thus closely follows the PlusCal specification in algorithm 5. The `AtomicBoolean` wraps a volatile boolean for which it also provides a CAS operation. Worker poll the `AtomicBoolean` on each invocation of `put` and `contains` to check if an eviction is pending. Once a pending eviction is detected, a worker waits on the `CyclicBarrier` which corresponds to the

[1] Some class names have been abbreviated compared to the actual JAVA implementation.

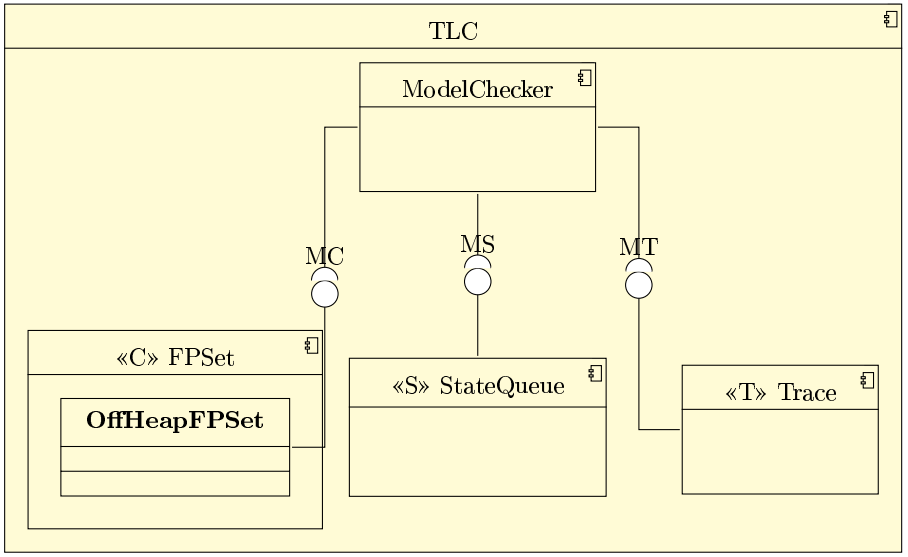


Figure 8.2.1.: UML component diagram of TLC

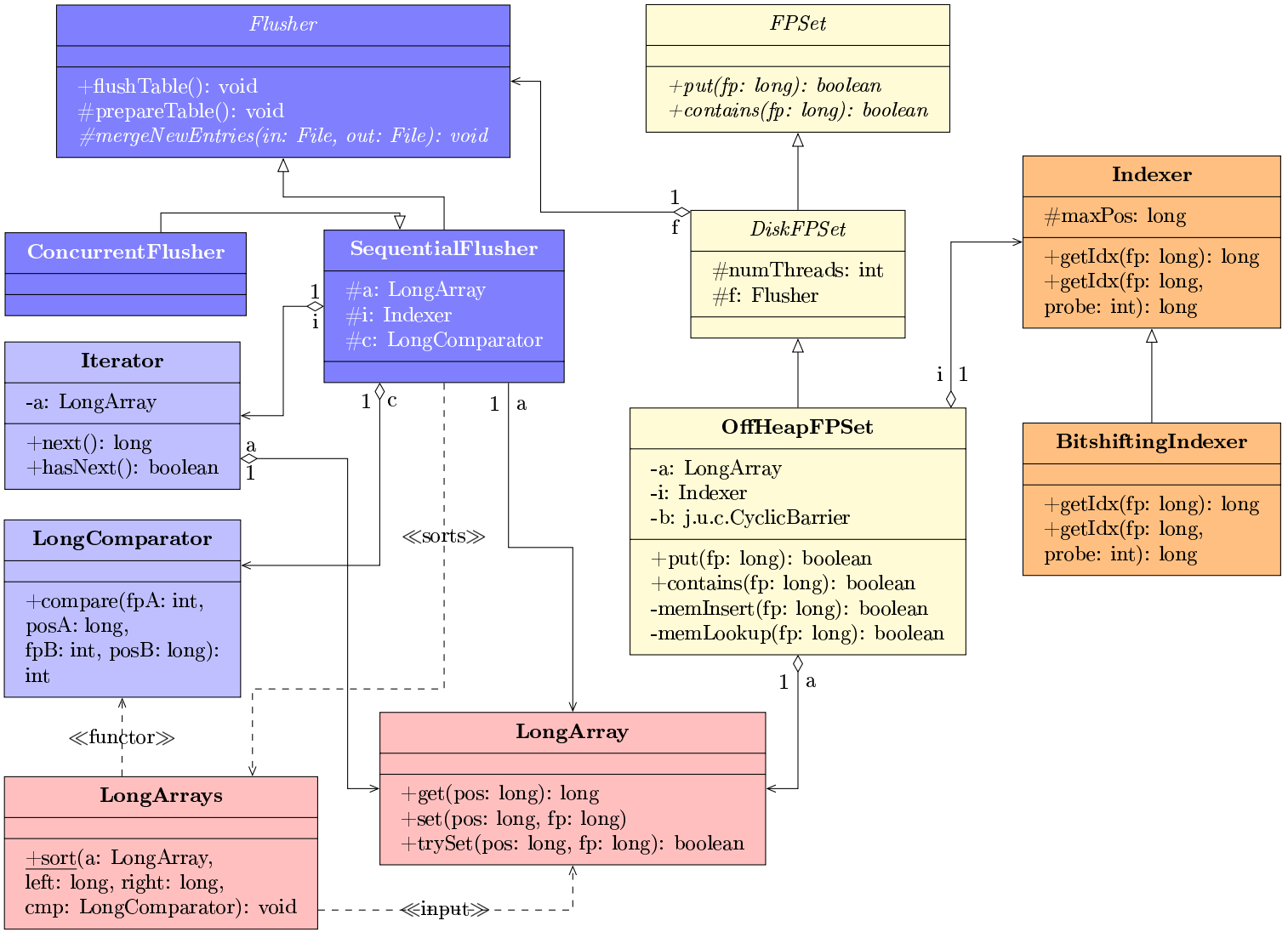


Figure 8.2.2.: UML class diagram for `OffHeapFPSet` and auxiliary classes

await statement at the *waitEv* label in algorithm 5. Brooker [2009] confirms that a volatile read incurs negligible overhead if the read operation dominates. This is true in this scenario.

In the implementation, the `TLA+ idx` function (compare algorithm 3) is encapsulated by the `Indexer` and `BitShiftingIndexer`, which are aggregates of `OffHeapFPSet`. This command pattern [compare Gamma, 1995] decouples the implementation from `OffHeapFPSet` for better modularity and testability. The `BitShiftingIndexer` command provides superior performance by avoiding the costly modulo operation iff the size of the hash table is a power of two. With the command pattern, a conditional on each index calculation is avoided.

Low-level memory access with `sun.misc.Unsafe` is wrapped by the class `LongArray`. `LongArray` translates the domain specific methods to `get` and `set` fingerprints to low-level `sun.misc.Unsafe` calls that manipulate raw memory. The use of raw, unmanaged memory reduces garbage collection in LongArray to a minimum. Conversely, the implementation has to manually allocate and free its memory. Additionally, `LongArray` acts as a facade to hide `sun.misc.Unsafe` in anticipation of its replacement with official API in a future JAVA release [see Reinhold, 2015]. The method `trySet` is the CAS equivalent to `set`. The method `trySet` returns true iff the CAS operation succeeds. `LongArray` is an aggregate of `OffHeapFPSet`.

Following JAVA's `Arrays` class approach — a class consisting exclusively of static methods that operate on or return array instances — `LongArrays` encapsulates various static methods that operate on `LongArray` instances. Most notably is the `sort` method, that sorts a `LongArray` or a range thereof. Sorting a range is used by the `Flusher` discussed below. As per the discussion about bounded disorder in section 3.4.1, `sort` implements the INSERTION SORT algorithm which is parallelizable with the scheme introduced in section 3.4.2.

In section 7.1.2 we established necessary modifications to the INSERTION SORT algorithm to skip open and marked positions. However, the modifications shall not appear in our implementation of standard INSERTION SORT. In other words, the implementation in `LongArrays` must remain a generic implementation of INSERTION SORT. Thus, we introduce the `LongComparator` functor which is passed as a parameter of the `sort` method. `LongComparator` implements the compare operator initially discussed in algorithm 6. This separation of concerns results in better modularity and maintainability. Especially, we expect `LongComparator` to be reusable if Insertion Sort is replaced with SMOOTHSORT or TIMSORT. Refactoring the sort algorithm is limited to swapping out

the INSERTION SORT algorithm for a new implementation. The method signature in `LongArrays` is not expected to change.

Eviction to external storage is encapsulated by the abstract class `Flusher`. An instance of `Flusher` provides a single-method interface to its aggregating class. This `flushTable` method triggers the eviction to external storage. Internally, `flushTable` is split into two stages which it delegates to the methods `prepareTable` and `mergeNewEntries`. If required, `prepareTable` allows subclasses of `Flusher` to prepare the table prior to eviction. In case of `OffHeapFPSet`, this entails sorting the subclasses' internal data structure — the `LongArray`. The method `mergeNewEntries` merges the table in primary memory with the linearly sorted `input` file located on external storage into the `output` file.[2] In order to hide the state inherent in traversing the `LongArray`, the standard iterator idiom is applied (see `Iterator`).

`OffHeapFPSet` supports a sequential and a concurrent eviction strategy. The `SequentialFlusher` strategy is chosen when `LongArray` is sufficiently small and does not warrant the overhead — i.e. instantiating threads and partitioning the `LongArray` — incurred by concurrent eviction. For all larger sizes of `LongArray`, the `ConcurrentFlusher` provides an implementation that is scalable in the number of workers by executing eviction collaboratively. It therefor partitions the LongArray instance into ranges for parallel sorting as discussed in section 3.4.2. In the subsequent `mergeNewEntries` stage, the `ConcurrentFlusher` writes the fully sorted partitions to external storage; again in parallel. Writing in parallel scales in the number of disks available.

We omit a sequence diagram as documentation of the interaction between the `ModelChecker` component and the `FPSet` component, i.e. the `OffHeapFPSet` implementation, because of its apparent simplicity. The behavioral aspects of the `OffHeapFPSet` are documented by the TLA$^+$ specification in B.2 instead.

8.3. Implementation

The implementation is included in the `master` branch of the official TLA$^+$ repository [see Lamport et al., 2017].

[2] `input` and `output` are the parameters of the `mergeNewEntries` method.

8.4. Verification and Validation

The conversion of the formal specification in B.2 into JAVA code is a manual task which requires due diligence. There are no guarantees, that the JAVA implementation correctly implements the PlusCal specification. In order to narrow the gap between the specification and implementation, we employed what we believe to be best practices beyond traditional approaches in software engineering.

The section 7.3 discusses the use of TLC module overwrites to speed up model checking. Besides reducing model checking time, a TLC module overwrite additionally allows one to model check code snippets of the actual implementation. The *rescale* TLC module overwrite in section 7.3 is the foundation for the *idx* calculation of the actual JAVA implementation of the algorithm. It was copied almost verbatim from the specification to the implementation.

Using TLC module overwrites is unfortunately impossible for PlusCal sections which span across multiple labels, e.g. the find-or-put operation as a whole.[3] Here, we cannot overwrite sections to derive a correct implementation. Instead we have to manually convert PlusCal into JAVA code. While both appear to be syntactically similar, their semantics differ. Thus, we manually encode the JAVA implementation but verify it with JPF [see Havelund and Pressburger, 2000], to guarantee correctness (compare 2.1.8). Verifying the implementation is beneficial to catch errors in the manual conversion. In our case, it detected a subtle error that intermittently lead to duplicate fingerprints. Traditional testing would have been unlikely to detect this error. The overhead to create the JPF safety properties is minimal, because TLA$^+$ properties can be manually converted into JPF ones. Obviously, only fundamental sections of the implementation are verified by JPF. Auxiliary parts of the implementation are not verified. First, auxiliary code is generally not specified at the abstraction level of TLA$^+$. Secondly, the resulting size of the state space makes this attempt infeasible. We are also confident, to find errors in auxiliary code with traditional validation.

We complemented verification for fundamental code with traditional validation, i.e. unit and functional testing of auxiliary code. In total, we added 141 unit tests for the `tlc2.tool.fp` package. Combined, the tests reach a code coverage in the range of 75% for `OffHeapDiskFPSet` and 61% at the package level [SonarQube, 2017]. In addition to unit testing, the code is equipped with several assertions to document and verify the pre- and post-conditions and (internal) invariants of `OffHeapDiskFPSet`.

[3] TLC module overwritesare limited to *stateless* TLA$^+$ operators.

The following chapter 9 presents the results of the validation of the non-functional requirements, namely scalability and performance. In the scope of this chapter, it is worth mentioning that the compiled (jitted) code of `OffHeapFPSet` has been manually inspected for performance bottlenecks as a result of poor optimizations by the HotSpot JIT compiler [compare The Hotspot Group, 1999]. No bottlenecks have been found in a runtime analysis with the aid of JITWatch [see Newland, 2013].

9. Evaluation

This chapter studies the performance and scalability of the lock-free algorithm proposed in chapter 7 and its implementation discussed in the previous chapter.

In order for experiments to be repeatable and reproducible, the chapter first highlights the environments on which the experiments were executed and the technical framework within measurements were collected.

The section 9.2 visualizes the measurements, evaluates the datasets and quantifies scalability with USL. In addition, it provides a representative comparison of TLC with the SPIN model checker.

9.1. Methodology

All experiments, except for the dataset corresponding to figure 4.2.2, were carried out on Amazon EC2 instances [Amazon Web Services, 2016]. The primary instance type used was x1.32xlarge, which as of today, is the most powerful instance type available. It is equipped with 128 cores, provided by four Intel E7-8880 v3 processors. The instance type has access to 1952 GB of memory. When experiments required less computer power, we used x1.16xlarge, r3.8xlarge and c3.8xlarge instances. They come with 64 cores and 976 GB main memory, 32 cores and 244 GB memory and 32 cores and 60 GB memory respectively. The experiment leading to figure 4.2.2 ran on an IBM Thinkpad T430s because, contrary to Amazon EC2 instances, it exposes counters for cache misses at the Linux kernel level.[1]

The operating system — on which all experiments ran — was Ubuntu Linux 16.04.1 LTS. Instances were provisioned with a script, which is part of Kuppe [2017]. At first, the JAVA virtual machine corresponded to version 1.8.0_101-b13 and later to 1.8.0_111-b14. The switch was made for security reasons. We are not aware of changes that would influence our experiments.

[1] Since then, performance measurement counters have been made available in EC2 [see Gregg, 2017].

Instead of micro-benchmarks, we opted for long running experiments to reduce the impact of startup times. Still, we modified TLC to collect fine grained execution times to exclude any startup time. We ran each experiment three times to be less prone to competing workloads at the OS level. Additionally, we had exclusive access to the underlying hardware of the x1.32xlarge and x1.16xlarge instances by requesting dedicated instances.

Kuppe [2017] contains all datasets, organized in folders which indicate the date and instance type of each experiment. Each folder also contains the script with which the experiments were started. The scripts do not require the caller to provide parameters. All figures indicate the corresponding dataset as part of their label. When applicable, a figure plots the 95% confidence interval. We removed outliers with a simple eyeball test, combined with an inspection of the Java Flight Recording for anomalies.

We collected profiling data at the JAVA VM level and specific TLC metrics, with the Java Flight Recording (JFR) tool [see Oracle, 2016]. The authors claim that JFR has negligible overhead. JFR does not require code modifications.

The TLA$^+$ specification used in experiments was Grid5k.tla [Lamport and Kuppe, 2011]. The shape of its state graph matches the well-shaped definition in section 4.1. The specification is part of the official TLA$^+$ repository [Lamport et al., 2017].

All experiments specific to the seen set C, measured its raw performance and scalability independent of other parts of TLC. Therefor, we created an artificial workload that resembles the workload of regular explicit state model checking. It is a random set of longs (fingerprints), generated by a pseudo random number generator (PNRG). We used the linear congruential random number generator [compare Knuth, 1997] provided by `java.util.Random`, but also experimented with Mersenne Twister and Marsaglia's xorshift [see Dyer, 2012]. We observed no measurable change in execution time with either of the three PNRGs. However, it is vitally important to assign each worker its dedicated PNRG. Concurrent access to a shared PNRG results in high contention, which distorts measurements.

Another systemic distortion in our test framework surfaced early on. A framework's loop condition was a comparison of the length K of the seen set C's internal table t. At the implementation level, K was maintained by a striped counter. A striped counter can be incremented without causing contention. Obtaining the current value on the other hand, requires synchronization of all writers and thus results in significant contention. We did not observe satisfactory performance and scalability, until we removed both bottlenecks (shared PNRG and shared counter).

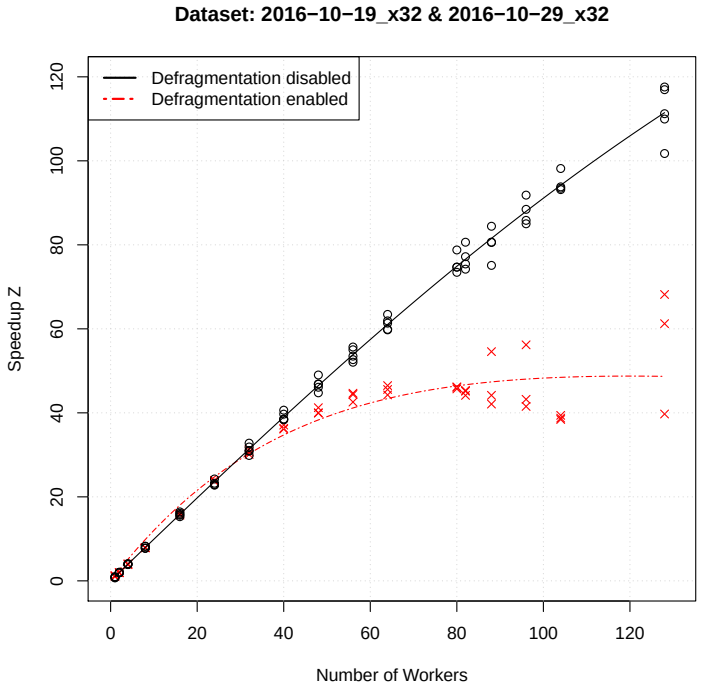


Figure 9.1.1.: Linux HugePage defragmentation has negative effects on scalability

At the Linux operating system, our experiments collided with the memory defragmentation efforts of the Linux kernel [see Torvalds, 2016a]. Unless we disabled huge page defragmentation, half of the system load could be attributed to the kernel's defragmentation process. Figure 9.1.1 shows the scalability of C_{proto} with and without defragmentation enabled. Additionally, we experimented with the settings for NUMA balancing [compare Torvalds, 2016c] and tested different CPU scaling governors [see Torvalds, 2016b]. Neither of those settings had a measurable effect though.

Finally, we identified hyper-threading as a source of distortion when worker counts saturated the instance's number of (virtual) cores. This effect is shown in figure 9.1.2 where the curve deteriorates from linear scalability (dashed line) with increasing worker counts. It shows the scalability of 1 to 128 independent — thus contention and coherence free — gzip processes compressing random chunks of data [concept by Fielding, 2014]. This workload can be considered an embarrassingly parallel problem. An x1.32xlarge instance has 72 physical cores. An r3.8xlarge instance has 20 physical cores.

For the scalability comparison of TLC_{proto} and SPIN, we used SPIN's current release [Holzmann, 2016]. We ran the comparison on Lamport's Bakery algorithm and obtained the TLA^+ specification from Lamport [2015a] and the PROMELA specification from

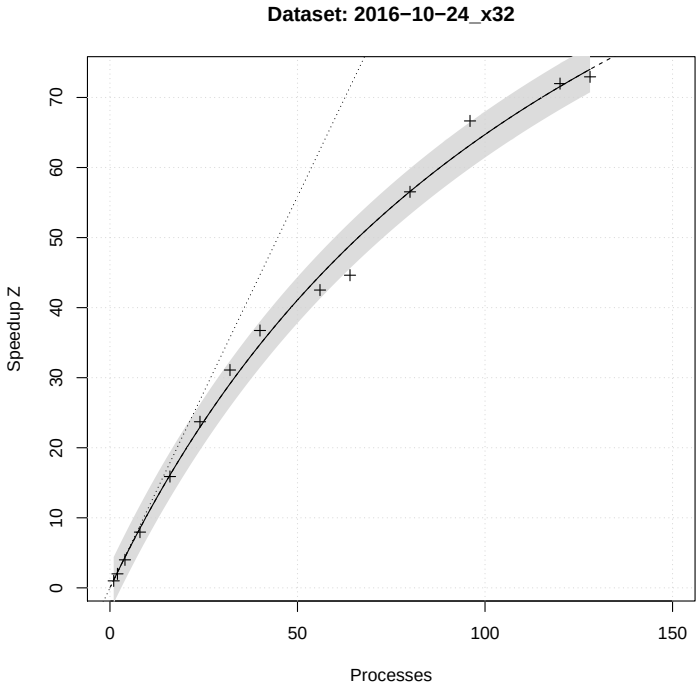


Figure 9.1.2.: Scalability degradation with hyper-threading exemplified on independent gzip processes

Kobayashi [2008]. Both models declare the number of processes to $N = 5$.

In order to make SPIN and TLC_{proto} comparable, we compiled SPIN with the *-DBFS_PAR* flag. This flag causes SPIN to execute its BFS algorithm. We manually determined the size for SPIN's set of fingerprints — what we call C — to be 2^{32}. By default, SPIN can maximally use 63 cores. All experiments were carried out on x1.32xlarge instances.

9.2. Results

As a preliminary result we expected — judging from the amount of test framework optimizations and OS tuning required (see previous section 9.1) — the new implementation to provide better performance and scalability than C_{legacy} and C_{stripe}. The legacy implementations show no speedup Z under increasing worker counts (compare section 4.1). This section will give a quantified view to substantiate this assumption.

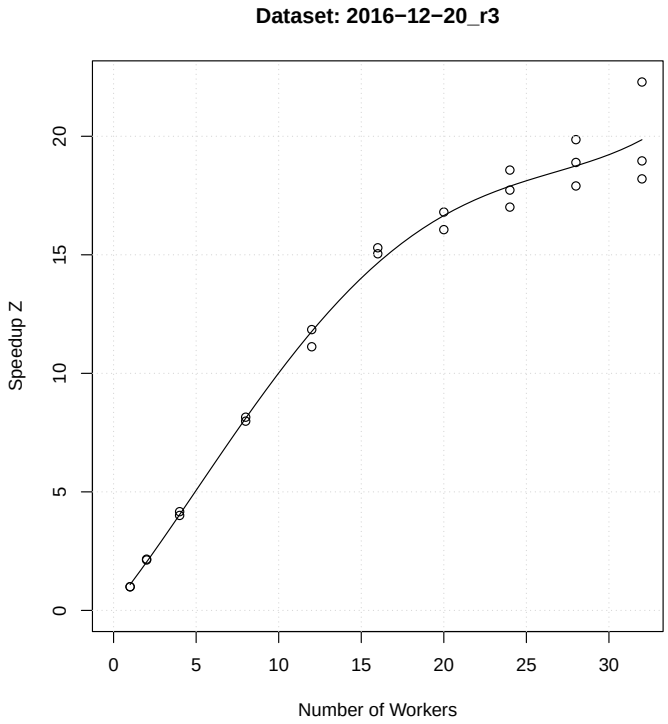


Figure 9.2.1.: Scalability of naïve parallel sort with bounded disorder

9.2.1. Seen Set

The C_{legacy} and C_{stripe} implementations do not support concurrent eviction. Eviction is a sequential implementation for both sorting and writing (compare section 7.1.2). Thus, it is pointless to compare C_{proto} to the legacy implementation. However, the graph 9.2.1 documents the scalability of C_{proto} with its implementation of naïve parallel sort under bounded disorder. The USL (see section 3.2) predicts scalability to peak at 37 workers. We partially attribute this number to the effects of hyper-threading as discussed in the previous section 9.1. Naïve parallel sort does neither exhibit contention nor coherence. Combined with Insertion Sort, it even provides optimal cache locality. The load factor of table t was chosen to be $\alpha \simeq 0.8$ in accordance with section 7.2. We did not measure the scalability of parallel writes.

Shifting the focus from eviction to the performance and scalability of the find-or-put operation, we compare C_{stripe} to C_{proto}. The performance of C_{stripe} and C_{proto} with a single worker is displayed in box plot 9.2.2. The median throughput of C_{proto} is on average 25% better than that of C_{stripe}. In addition, the plot reveals a larger variability

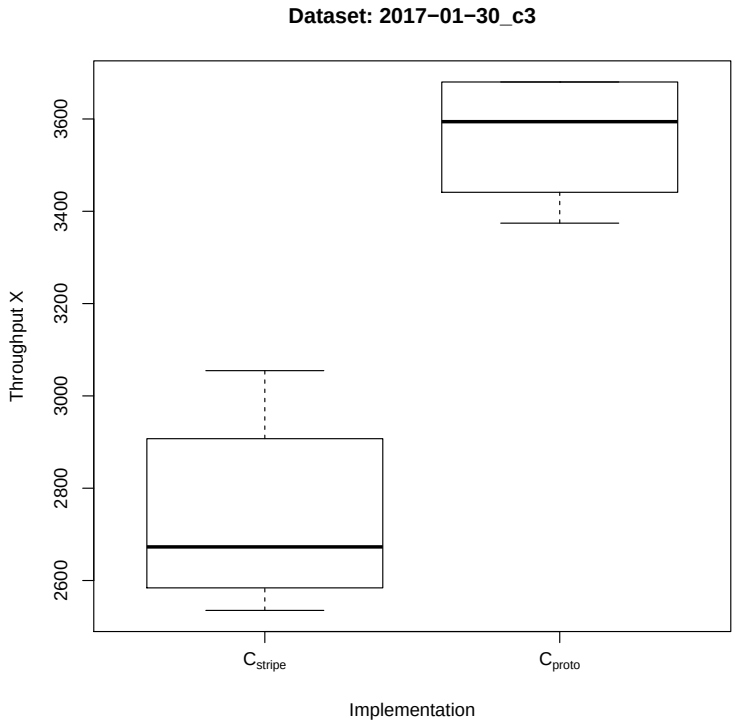


Figure 9.2.2.: The new C_{proto} provides superior throughput and lower variance over C_{stripe}

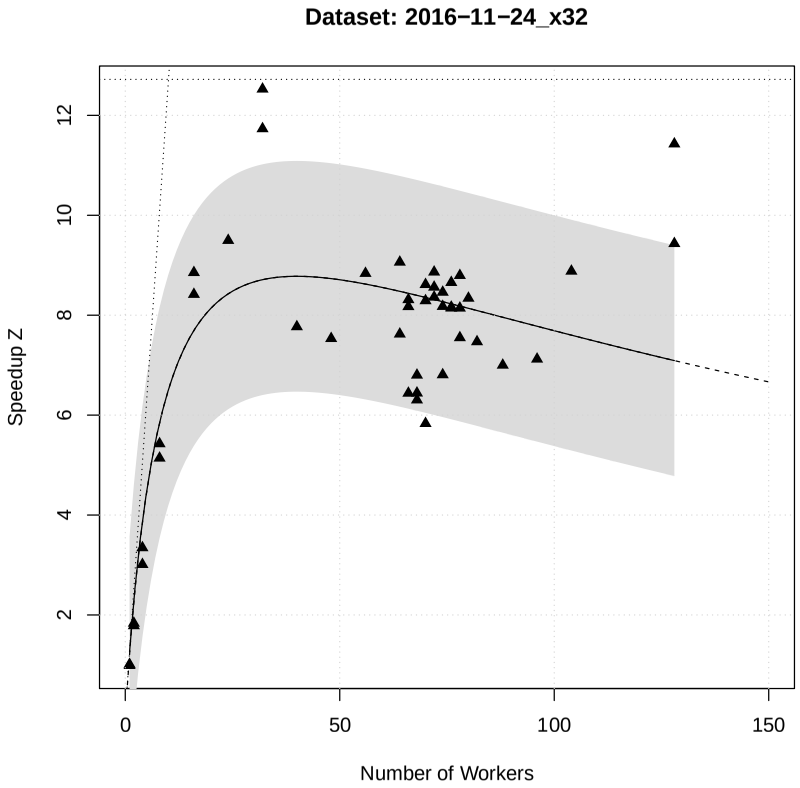
in C_{stripe} compared to C_{proto} as indicated by the quantiles.[2] The plot 9.2.2 omits outliers.

The graph 9.2.3a shows the scalability of the find-or-put operation for C_{stripe}. First and foremost, the wide 95% confidence interval, indicated by the gray area, grabs attention. The widening interval can be attributed to the dynamic array allocation and its related garbage collection during the execution of the find-or-put operation (see section 3.3.2).

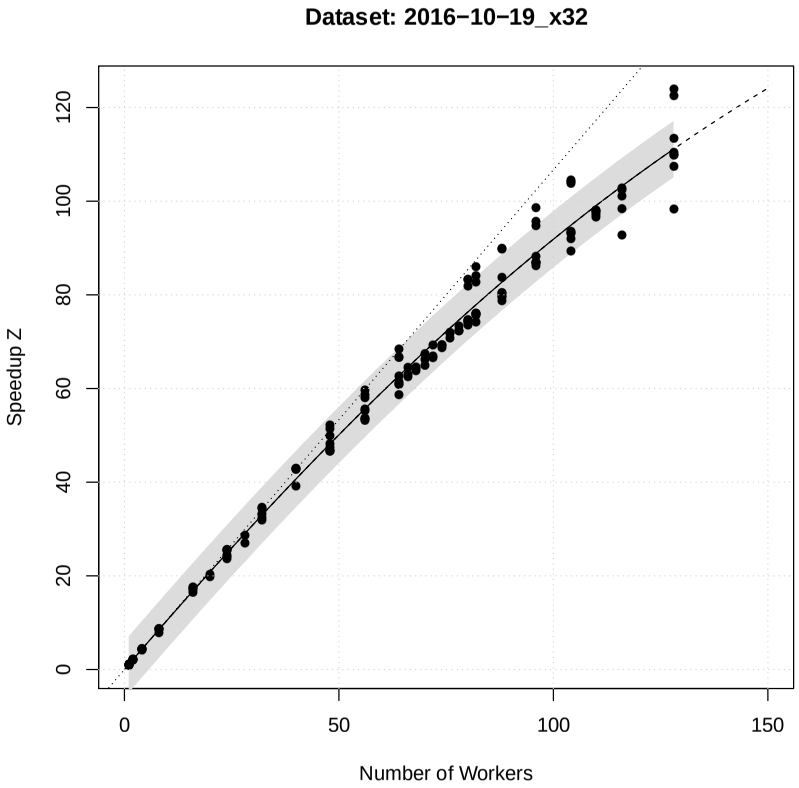
Likewise, dynamic allocation and garbage collection partially explains the overall unsatisfactory speedup. Combined with the slowdown attributed to coherence as a result of lock-striping (compare section 4.2), the speedup not only peaks at 40 workers, but generally never exceeds an order of magnitude. Linear speedup, indicated by the dotted line, ends at 4 cores.

Graph 9.2.3b displays the scalability plot for C_{proto}. The implementation exhibits a narrow 95% confidence interval, which we assume to be within the range of OS level noise and benchmark inaccuracy. The USL predicts scalability to peak at 400 workers; an order of magnitude higher compared to C_{stripe}. Similarly, the speed almost increases linearly and thus exceeds that of C_{legacy} by an order of magnitude.

[2]The box represents the middle 50% of the data. Thus, the box edges are the 75th and 25th percentiles.



(a) Scalability C_{stripe} with SC



(b) Scalability C_{proto} with OA

Figure 9.2.3.: Scalability of C_{proto} scales close to linearly and provides an order of magnitude speedup over C_{stripe}

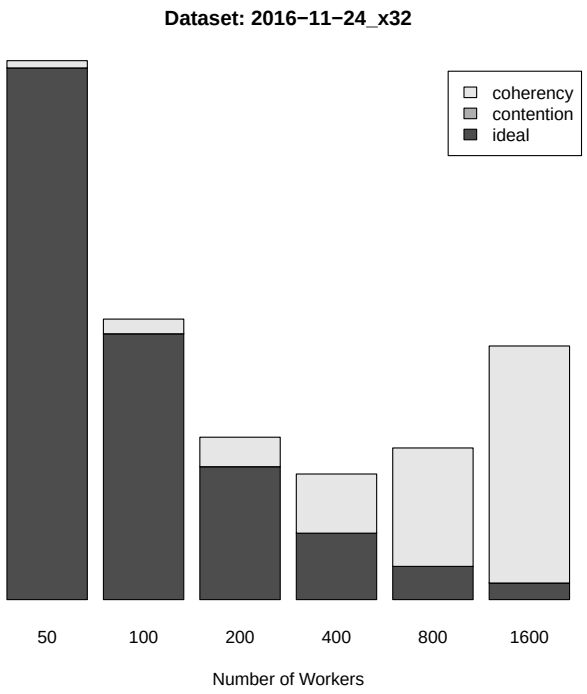


Figure 9.2.4.: Coherence and contention prediction for C_{proto} up to 1600 processes

Using the USL to predict the scalability of C_{proto} for higher numbers of workers, reveals increasing coherence. Starting at 400 processes, the plot shows diminishing returns for additional processes. At 1600 processes, coherence is the dominant contributor to runtime (see bar plot 9.2.4). To what extend this prediction is related to hyper-threading, should be clarified by running benchmarks on machines with larger core counts.

9.2.2. Parallel TLC

Next we expand the perspective to TLC_{proto}. In contrast to the previously shown scalability of TLC_{legacy} (see graph figure 4.1.1), the graph in figure 9.2.5 displays significantly higher scalability with a predicted peak at 119 workers. This improvement is close to an order of magnitude over TLC_{legacy}, which peaks at 16 workers. Similarly, comparing the absolute speedup, TLC_{proto} improves by more than an order of magnitude from $Z_{legacy} \gtrsim 2.2$ to $Z_{proto} \lesssim 60$.

Lastly, we measure how TLC_{proto} stacks up against SPIN. The figure 9.2.6 shows that TLC_{proto} achieves a higher speedup than SPIN and predicts TLC_{proto} to scale up to 139 workers. This number is in line with the predication for the Grid5k specification (see previous paragraph). However, TLC's absolute speedup is lower compared to the

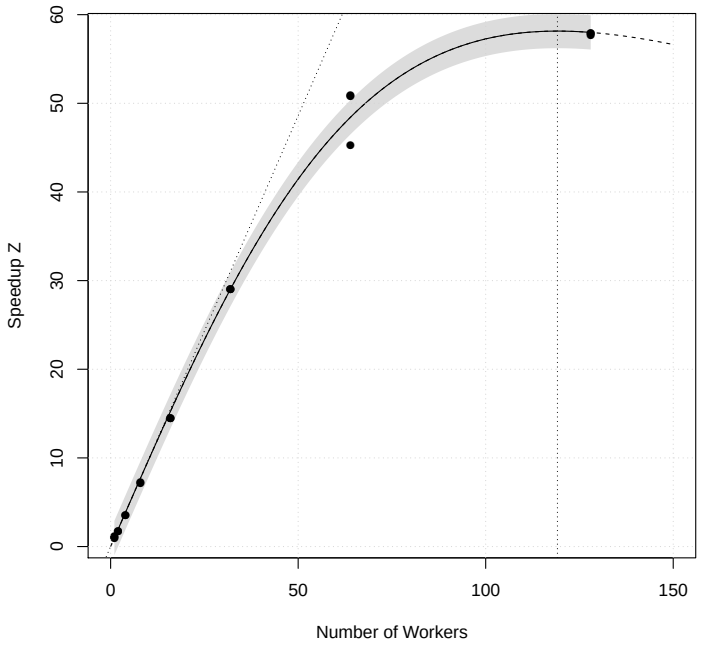


Figure 9.2.5.: Scalability of TLC_{proto} with Grid5k specification

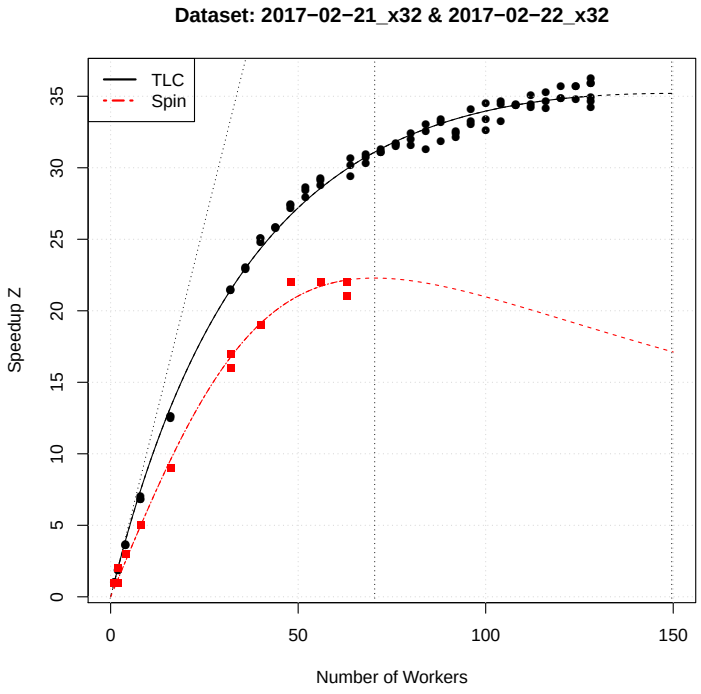


Figure 9.2.6.: Scalability of SPIN and TLC_{proto} with Bakery specification

Grid5k specification. SPIN's scalability is predicted to peak at 73 cores and its speedup below that of TLC_{proto}. For review, we privately sent our results to the author of SPIN, who approved of our measurement methodology [Holzmann and Kuppe, 2017].

All datasets are available on the attached CD-ROM. Additionally, the datasets can be found at Kuppe [2017].

10. Conclusion

This thesis contributes to the research on multi-core scalability of explicit state model checking. It presents a novel, lock-free hash table algorithm for the set of seen states commonly found in explicit state model checkers. The algorithm achieves a high space utilization through the use of open addressing. Contrary to other algorithms found in today's model checkers, the algorithm supports the extension of the hash table to external storage, i.e. disks. It is thus not memory bound, which allows the TLC model checker to verify otherwise intractable models.

All operations of the algorithm run in parallel, even the extension to disk. For that, we devised a parallel sorting scheme, that can be used universally, iff the data to be sorted has a bounded disorder. The scheme achieves linear scalability.

We provide a formal PlusCal specification of the algorithm. We thus partially deliver on the task set out by Dijkstra [1988], to formally specify a program and proof its correctness. We verify correctness of finite models with the TLC model checker, but lack a rigorous proof with e.g. TLAPS.

The thesis includes a JAVA implementation of the algorithm, which is freely available under a permissive license. We consider it to be production-ready. The core parts of the implementation have been checked with JPF. Auxiliary parts have been unit tested. Benchmarks demonstrate — and the universal scalability law even quantifies — the implementation's competitive performance and scalability.

Widening its focus with regards to scalability of explicit state model checking, the thesis shifts its attention to the two other primary data structures present in model checkers: the set of unseen states and the forest to construct counterexamples. The thesis argues, that a shared-nothing design for the two data structures eliminates contention and coherence and thus increases scalability. It includes prototypic implementations of both data structures.

Combining all three implementations into TLC_{proto}, the thesis conducts benchmarks at the model checker level, which reveal superior performance and scalability compared to its former version TLC_{legacy}, as well as current SPIN. In short, TLC_{proto} scales to

computers with hundreds of cores and terabytes of memory.

We show, that performance related optimizations at the OS level, have a negative impact on TLC's performance and scalability. We list tuning parameters, which remove the performance and scalability degradation. The tuning parameters have since been incorporated into Cloud Distributed TLC.

As a side effect of TLC's scalability analysis and related to software engineering in general, we identify two convincing arguments in favor of the lock-free programming pattern.

First, the popular approach of lock-striping — to eliminate lock contention — does not scale. Our experiments show, that an increase in the number of locks, leads to what can be described as lock coherence.

Secondly, the scalability results presented as part of this thesis confirm that (existing) programs can benefit from today's massive multi-core computers, if key parts of the program are redesigned to be lock-free. An increase in performance and scalability can be accomplished.

However, the inherent complexity of the lock-free programming pattern makes reasoning about programs more difficult. Generally, we believe that the ever increasing complexity of computer programs has to be matched by the wider adoption of formal methods. In this regard, the thesis confirms the practicality of a specification-driven, iterative development process. Additionally, it provides an outlook into how TLC module overwrites and a combination of TLA^+ and JPF contribute to a more robust (manual) translation from a high-level specification to a corresponding implementation.

11. Future Work

We consider the algorithm and implementation for the seen set C to be production-ready. Still, we see areas for future research. The naïve parallel sort scheme introduced in section 3.4.2 requires two passes. Under the assumption of a uniform distribution of fingerprints even under high load factors, the second pass — the merge phase — can be skipped, if unoccupied table positions could be found efficiently (see 7.1.2 on page 56).

Secondly, the external storage lookups have been identified as a performance bottleneck (see section 3.3.3). To alleviate this problem, a BLOOM filter (compare section 6.2) placed in front of the external storage, would reduce the number of lookups for non-existing fingerprints.

Similarly, section 7.1.4 discusses the significant constant factor of the eviction complexity, which implies that the algorithm does not scale with the number of evictions. A different external storage layout may be able to reduce the constant factor.

We expect the algorithm to generally be forward compatible with 128 bit fingerprints as requested in section 5.2. Yet, we believe compatibility to require DCAS support (compare section 3.1.1) at the JAVA language level.[1]

Lastly, a reduction in the space requirement of the hash table can be accomplished, if fingerprints are Cleary compacted [compare Cleary, 1984, van der Vegt and Laarman, 2012]. The LP collision resolution and the bounded disorder allows the complete fingerprint to be reconstructed from its suffix stored in table t and the suffix's position in t.

Expanding our perspective to TLC, section 9.2 shows that the prototypic implementation for the unseen set S and the forest — both following a shared nothing design as discussed in section 4.1 — increase TLC's overall scalability up to an order of magnitude. These two prototypes have to be formally specified, verified and implemented once a re-balancing algorithm for the unseen set S has been envisioned. Preliminary considerations see the re-assignment of the unseen set S's disk pages to idle workers as

[1] An efficient 128 bit fingerprint implementation needs user-defined primitive types as proposed in Rose [2015].

the most effective re-balancing approach. Nevertheless, the work of Holzmann [2012] is certainly of interest here. Also related to the unseen set S are the savings inherent in state compression. Yu et al. [1999] presented preliminary yet promising results. These results should be revisited, once load balancing of the unseen set S is researched.

Improved scalability of liveness checking has not been the scope of this thesis. Yet, as apparent from the results in table 7.1, checking liveness properties is a major contributor to the overall model checking time. The primary reason being the *sequential* search for strongly connected components [compare Tarjan, 1972]. A prototype [see Kuppe, 2015] of a *concurrent* algorithm proposed by Tarjan [2015], showed unsatisfactory results; primarily due to high contention as a result of locking. In the light of the performance and scalability gains demonstrated in this thesis, the prototype might exhibit reduced contention when implemented with the lock-free programming pattern.

A reduction of the size of the liveness graph is achievable, if TLC's algorithm for symmetry reduction would be enhanced to fully handle the verification of liveness properties.

Unrelated to TLC's scalability, it would be worthwhile to explore the connection between the shape and the size of a specification's state graph, to its implementation's performance and scalability characteristics; also requested by Newcombe [2014]. The existence of such connections, would not only shorten the time consuming task of running scalability benchmarks, but — more importantly — allow to short-circuit the specification-driven development process discussed above. Likewise, we believe the study of TLC module overwrites (see section 7.3) at the PlusCal language level to be valuable. This feature would make a translation of a PlusCal specification, to a corresponding implementation, less error prone.

Both features would bring us one tiny step closer to shifting programing from being an art to being a science, as requested by Dijkstra [1988].

Acknowledgement

I would like to thank my family for their support throughout the course of this thesis. Especially, I thank my wife for her continuous efforts to iron out orthographic errors. I look forward to returning the favor.

I would like to thank Dr. Daniel Moldt for his valuable feedback, for reviewing the thesis and for always providing a welcoming atmosphere within the research group.

Likewise, I want to thank Dr. Leslie Lamport not only for agreeing to review my thesis, but also for the opportunity to be part of the TLA$^+$ project. When I joined the project a few years ago, I came as a programmer. By now I have learned what it takes to be a computer scientist.

Thanks to Dr. Gunther for providing scripts to print the graphs.

Bibliography

Basic concepts. In G. Goos, J. Hartmanis, D. Barstow, W. Brauer, P. Brinch Hansen, D. Gries, D. Luckham, C. Moler, A. Pnueli, G. Seegmüller, J. Stoer, N. Wirth, M. Paul, H. J. Siegert, M. W. Alford, J. P. Ansart, G. Hommel, L. Lamport, B. Liskov, G. P. Mullery, and F. B. Schneider, editors, *Distributed Systems*, volume 190, pages 7–43. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985. ISBN 978-3-540-15216-3 978-3-540-39363-4.

Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991. ISSN 03043975. doi: 10.1016/0304-3975(91)90224-P. URL `http://linkinghub.elsevier.com/retrieve/pii/030439759190224P`.

S.V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, 1996. ISSN 00189162. doi: 10.1109/2.546611. URL `http://ieeexplore.ieee.org/document/546611/`.

Bowen Alpern and Fred B. Schneider. Defining Liveness. Technical report, Cornell University, Ithaca, NY, USA, 1984.

Rajeev Alur and Thomas A Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999. doi: 10.1023/A:1008739929481.

Amazon Web Services. Amazon EC2 Instance Types, 2016. URL `https://aws.amazon.com/ec2/instance-types/`. (Accessed 2016-12-16).

Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. page 483. ACM Press, 1967. doi: 10.1145/1465482.1465560. URL `http://portal.acm.org/citation.cfm?doid=1465482.1465560`.

K. R. Apt, N. Francez, and S. Katz. Appraising fairness in distributed languages. pages 189–198. ACM Press, 1987. ISBN 978-0-89791-215-0. doi: 10.1145/41625.41642. URL `http://portal.acm.org/citation.cfm?doid=41625.41642`.

J. Barnat, P. Ročkai, V. Štill, and J. Weiser. Fast, Dynamically-Sized Concurrent Hash Table. In Bernd Fischer and Jaco Geldenhuys, editors, *Model Checking Software*, volume 9232, pages 49–65. Springer International Publishing, Cham, 2015. ISBN 978-3-319-23403-8 978-3-319-23404-5. URL `http://link.springer.com/10.1007/978-3-319-23404-5_5`.

Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai, Vladimír Štill, and Jiří Weiser. DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Natasha Sharygina, and Helmut Veith, editors, *Computer Aided Verification*, volume 8044, pages 863–868. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-39798-1 978-3-642-39799-8. URL `http://link.springer.com/10.1007/978-3-642-39799-8_60`.

Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970. ISSN 00010782. doi: 10.1145/362686.362692. URL `http://portal.acm.org/citation.cfm?doid=362686.362692`.

Marc Brooker. Are volatile reads really free?, 2009. URL `https://brooker.co.za/blog/2012/09/10/volatile.html`. (Accessed 2017-01-24).

Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying Safety Properties with the TLA+ Proof System. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Jürgen Giesl, and Reiner Hähnle, editors, *Automated Reasoning*, volume 6173, pages 142–148. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-14202-4 978-3-642-14203-1. URL `http://link.springer.com/10.1007/978-3-642-14203-1_12`.

Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Ed Brinksma, and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404, pages 359–364. Springer Berlin Heidelberg, Berlin,

Heidelberg, 2002. ISBN 978-3-540-43997-4 978-3-540-45657-5. URL http://link.springer.com/10.1007/3-540-45657-0_29.

Edmund M. Clarke. The Birth of Model Checking. In Orna Grumberg and Helmut Veith, editors, *25 Years of Model Checking*, volume 5000, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-69849-4 978-3-540-69850-0. URL http://link.springer.com/10.1007/978-3-540-69850-0_1.

Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model Checking and the State Explosion Problem. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Bertrand Meyer, and Martin Nordio, editors, *Tools for Practical Software Verification*, volume 7682, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-35745-9 978-3-642-35746-6. URL http://link.springer.com/10.1007/978-3-642-35746-6_1.

John G. Cleary. Compact Hash Tables Using Bidirectional Linear Probing. *IEEE Transactions on Computers*, C-33(9):828–834, September 1984. ISSN 0018-9340. doi: 10.1109/TC.1984.1676499. URL http://ieeexplore.ieee.org/document/1676499/.

Curtis R. Cook and Do Jin Kim. Best sorting algorithm for nearly sorted lists. *Communications of the ACM*, 23(11):620–624, November 1980. ISSN 00010782. doi: 10.1145/359024.359026. URL http://portal.acm.org/citation.cfm?doid=359024.359026.

Edsger W. Dijkstra. Smoothsort, an alternative for sorting in situ. *Science of Computer Programming*, 1(3):223–233, May 1982. ISSN 01676423. doi: 10.1016/0167-6423(82)90016-8. URL http://linkinghub.elsevier.com/retrieve/pii/0167642382900168.

Edsger W. Dijkstra. On the Cruelty of Really Teaching Computing Science. circulated privately, December 1988. URL http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1036.PDF.

Daniel Dyer. Uncommons Maths, 2012. URL http://maths.uncommons.org/. (Accessed 2017-05-12).

E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Formal Methods in System Design*, 9(1-2):105–131, August 1996. ISSN 0925-9856, 1572-8102. doi: 10.1007/BF00625970. URL http://link.springer.com/10.1007/BF00625970.

Vladmir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *ACM Computing Surveys*, 24(4):441–476, December 1992. ISSN 03600300. doi: 10.1145/146370.146381. URL http://portal.acm.org/citation.cfm?doid=146370.146381.

Marc Fielding. Virtual CPUs with Amazon Web Services, 2014. URL https://www.pythian.com/blog/virtual-cpus-with-amazon-web-services/. (Accessed 2016-12-19).

Ioannis Filippidis and Gerard J. Holzmann. An improvement of the piggyback algorithm for parallel model checking. pages 48–57. ACM Press, 2014. ISBN 978-1-4503-2452-6. doi: 10.1145/2632362.2632375. URL http://dl.acm.org/citation.cfm?doid=2632362.2632375.

Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985. ISSN 00045411. doi: 10.1145/3149.214121. URL http://portal.acm.org/citation.cfm?doid=3149.214121.

Daniel Frampton, Stephen M. Blackburn, Perry Cheng, Robin J. Garner, David Grove, J. Eliot B. Moss, and Sergey I. Salishev. Demystifying magic: High-level low-level programming. page 81. ACM Press, 2009. ISBN 978-1-60558-375-4. doi: 10.1145/1508293.1508305. URL http://portal.acm.org/citation.cfm?doid=1508293.1508305.

Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar. Comparison of Model Checking Tools for Information Systems. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Jin Song Dong, and Huibiao Zhu, editors, *Formal Methods and Software Engineering*, volume 6447, pages 581–596. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-16900-7 978-3-642-16901-4. URL http://link.springer.com/10.1007/978-3-642-16901-4_38.

Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):5–es, May 2007. ISSN 07342071. doi: 10.1145/1233307.1233309. URL http://portal.acm.org/citation.cfm?doid=1233307.1233309.

Erich Gamma, editor. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Addison-Wesley, Reading, Mass, 1995. ISBN 978-0-201-63361-0.

Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. pages 245–257. ACM Press, 1991. ISBN 978-0-89791-380-5. doi: 10.1145/106972.106997. URL http://portal.acm.org/citation.cfm?doid=106972.106997.

Brendan Gregg. The PMCs of EC2: Measuring IPC, 2017. URL http://www.brendangregg.com/blog/2017-05-04/the-pmcs-of-ec2.html. (Accessed 2017-05-11).

Neil Gunther. *Guerrilla Capacity Planning*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-26138-4. URL http://link.springer.com/10.1007/978-3-540-31010-5.

Neil J. Gunther. Random R scripts, 2016a. URL https://github.com/DrQz/random-r/blob/master/lemmster-ggplot.r. (Accessed 2017-05-22).

Neil J. Gunther. Print USL with R, 2016b. URL https://twitter.com/lemmster/status/790558334936092672.

Timothy L. Harris, Keir Fraser, and Ian A. Pratt. A Practical Multi-word Compare-and-Swap Operation. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Dahlia Malkhi, editors, *Distributed Computing*, volume 2508, pages 265–279. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002. ISBN 978-3-540-00073-0 978-3-540-36108-4. URL http://link.springer.com/10.1007/3-540-36108-1_18.

Klaus Havelund and Thomas Pressburger. Model checking JAVA programs using JAVA PathFinder. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):366–381, March 2000. ISSN 1433-2779, 1433-2787. doi: 10.1007/s100090050043. URL http://link.springer.com/10.1007/s100090050043.

Thomas A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43 (3):135–141, September 1992. ISSN 00200190. doi: 10.1016/0020-0190(92)90005-G. URL http://linkinghub.elsevier.com/retrieve/pii/002001909290005G.

Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991. ISSN 01640925. doi: 10.1145/114005.102808. URL http://portal.acm.org/citation.cfm?doid=114005.102808.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Amsterdam, revised first edition edition, 2012. ISBN 978-0-12-397337-5.

Mark D. Hill. What is scalability? *ACM SIGARCH Computer Architecture News*, 18(4):18–21, December 1990. ISSN 01635964. doi: 10.1145/121973.121975. URL http://portal.acm.org/citation.cfm?doid=121973.121975.

C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, January 1962. ISSN 0010-4620, 1460-2067. doi: 10.1093/comjnl/5.1.10. URL http://comjnl.oupjournals.org/cgi/doi/10.1093/comjnl/5.1.10.

Gerard J. Holzmann. An improved protocol reachability analysis technique. *Software: Practice and Experience*, 18(2):137–161, February 1988. ISSN 00380644, 1097024X. doi: 10.1002/spe.4380180203. URL http://doi.wiley.com/10.1002/spe.4380180203.

Gerard J. Holzmann. ACM Software System Award 2001, 2001. URL http://awards.acm.org/award_winners/holzmann_1625680.cfm. (Accessed 2017-03-27).

Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, 2. print edition, 2005. ISBN 978-0-321-77371-5.

Gerard J. Holzmann. A Stack-Slicing Algorithm for Multi-Core Model Checking. *Electronic Notes in Theoretical Computer Science*, 198(1):3–16, February 2008. ISSN 15710661. doi: 10.1016/j.entcs.2007.10.017. URL http://spinroot.com/gerard/pdf/pdmc2007.pdf.

Gerard J. Holzmann. Parallelizing the Spin Model Checker. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri

Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Alastair Donaldson, and David Parker, editors, *Model Checking Software*, volume 7385, pages 155–171. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-31758-3 978-3-642-31759-0. URL http://spinroot.com/gerard/pdf/spin2012_a.pdf.

Gerard J. Holzmann. Mars code. *Communications of the ACM*, 57(2):64–73, February 2014. ISSN 00010782. doi: 10.1145/2560217.2560218. URL http://dl.acm.org/citation.cfm?doid=2556647.2560218.

Gerard J. Holzmann. Spin Sources, 2016. URL https://spinroot.com/spin/Src/. (Accessed 2017-02-21).

Gerard J. Holzmann and Markus Alexander Kuppe. Scalability comparison of SPIN and TLC, 2017.

Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm Verification Techniques. *IEEE Transactions on Software Engineering*, 37(6):845–857, November 2011. ISSN 0098-5589. doi: 10.1109/TSE.2010.110. URL http://ieeexplore.ieee.org/document/5661793/.

Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. pages 151–160. ACM Press, 1994. ISBN 978-0-89791-654-7. doi: 10.1145/197917.198079. URL http://portal.acm.org/citation.cfm?doid=197917.198079.

Rajeev Joshi, Leslie Lamport, John Matthews, Serdar Tasiran, Mark Tuttle, and Yuan Yu. Checking Cache-Coherence Protocols with TLA+. *Formal Methods in System Design*, 22(2):125–131, March 2003. ISSN 0925-9856, 1572-8102. doi: 10.1023/A:1022969405325. URL http://link.springer.com/10.1023/A:1022969405325.

Gijs Kant, Alfons Laarman, Jeroen Meijer, Jaco van de Pol, Stefan Blom, and Tom van Dijk. LTSmin: High-Performance Language-Independent Model Checking. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 9035, pages 692–707. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015. ISBN 978-3-662-46680-3 978-3-662-46681-0. URL http://link.springer.com/10.1007/978-3-662-46681-0_61.

William Knight and D. M. Bloom. E2386. *The American Mathematical Monthly*, 80 (10):1141, December 1973. ISSN 00029890. doi: 10.2307/2318556. URL http://www.jstor.org/stable/2318556?origin=crossref.

Donald Ervin Knuth. *The Art of Computer Programming.* Addison-Wesley, Reading, Mass, 3rd ed edition, 1997. ISBN 978-0-201-89683-1 978-0-201-89684-8 978-0-201-89685-5.

Fumiyoshi Kobayashi. Bakery Algorithm in Promela, 2008. URL `http://www.ueda.info.waseda.ac.jp/~kobayashi/Promela/benchmark/bakery.html`. (Accessed 2017-02-21).

F. Kordon, H. Garavel, L. M. Hillah, F. Hulin-Hubard, G. Chiardo, A. Hamez, L. Jezequel, A. Miner, J. Meijer, E. Paviot-Adet, D. Racordon, C. Rodriguez, C. Rohr, J. Srba, Y. Thierry-Mieg, G. Trdinh, and K. Wolf. Complete Results for the 2016 Edition of the Model Checking Contest. June 2016. URL `http://mcc.lip6.fr/2016/results.php`.

Markus Alexander Kuppe. Current State of Distributed TLC, 2012. URL `http://tla2012.loria.fr/contributed/kuppe-slides.pdf`. (Accessed 2016-09-06).

Markus Alexander Kuppe. Distributed TLC, 2014. URL `http://tla2014.loria.fr/slides/kuppe.pdf`. (Accessed 2016-09-06).

Markus Alexander Kuppe. TarjanConcurrentSCC, 2015. URL `https://bitbucket.org/lemmster/tarjanconcurrentscc`. (Accessed 2017-05-12).

Markus Alexander Kuppe. TLC Scalability datasets, 2017. URL `https://github.com/lemmy/tlc-scalability`. (Accessed 2017-05-12).

M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

Alfons Laarman, Jaco van de Pol, and Michael Weber. Boosting Multi-Core Reachability Performance with Shared Hash Tables. *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design*, April 2010. URL `http://doc.utwente.nl/75591/`. Technical Report of paper published at FMCAD 2010 with the same name.

Alfons Laarman, Jaco van de Pol, and Michael Weber. Multi-Core LTSmin: Marrying Modularity and Scalability. In David Hutchison, Takeo Kanade, Josef Kittler,

Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617, pages 506–511. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-20397-8 978-3-642-20398-5. URL `http://link.springer.com/10.1007/978-3-642-20398-5_40`.

Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979. ISSN 0018-9340. doi: 10.1109/TC.1979.1675439. URL `http://ieeexplore.ieee.org/document/1675439/`.

L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977. ISSN 0098-5589. doi: 10.1109/TSE.1977.229904. URL `http://ieeexplore.ieee.org/document/1702415/`.

Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, August 1974. ISSN 00010782. doi: 10.1145/361082.361093. URL `http://portal.acm.org/citation.cfm?doid=361082.361093`.

Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994. ISSN 01640925. doi: 10.1145/177492.177726. URL `http://portal.acm.org/citation.cfm?doid=177492.177726`.

Leslie Lamport. How to Write a Proof. *The American mathematical monthly*, 102(7):600–608, 1995. URL `http://www.jstor.org/stable/2974556`.

Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, 2003. ISBN 0-321-14306-X.

Leslie Lamport. Checking a Multithreaded Algorithm with + CAL. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, and Shlomi Dolev, editors, *Distributed Computing*, volume 4167, pages 151–163. Springer

Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-44624-8 978-3-540-44627-9. URL http://link.springer.com/10.1007/11864219_11.

Leslie Lamport. Computation and State Machines, 2008. URL http://lamport.azurewebsites.net/pubs/state-machine.pdf.

Leslie Lamport. The PlusCal Algorithm Language. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Martin Leucker, and Carroll Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009*, volume 5684, pages 36–60. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-03465-7 978-3-642-03466-4. URL http://link.springer.com/10.1007/978-3-642-03466-4_2.

Leslie Lamport. *Principles of Concurrent Computing and Specification of Concurrent Systems*. 2014. URL http://research.microsoft.com/en-us/um/people/lamport/tla/hyperbook.html.

Leslie Lamport. The Boulangerie Algorithm in PlusCal/TLA+, 2015a. URL http://research.microsoft.com/en-us/um/people/lamport/tla/boulangerie.html. (Accessed 2016-05-25).

Leslie Lamport. TLATeX–A Typesetter for TLA+ Specifications, 2015b. URL http://lamport.azurewebsites.net/tla/tlatex.html. (Accessed 2017-05-28).

Leslie Lamport. The TLA+ Video Course, 2017. URL http://lamport.azurewebsites.net/video/videos.html. (Accessed 2017-04-27).

Leslie Lamport and Markus Alexander Kuppe. Grid5k.tla, 2011. URL https://github.com/tlaplus/tlaplus/blob/master/general/performance/Grid5k.tla. (Accessed 2017-02-13).

Leslie Lamport, John Matthews, Mark Tuttle, and Yuan Yu. Specifying and verifying systems with TLA+. page 45. ACM Press, 2002. doi: 10.1145/1133373.1133382. URL http://portal.acm.org/citation.cfm?doid=1133373.1133382.

Leslie Lamport, Yuan Yu, Markus Alexander Kuppe, and others. The TLA+ Repository, 2017. URL https://github.com/tlaplus/tlaplus. (Accessed 2017-05-11).

Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 97–107. ACM Press, 1985. ISBN 978-0-89791-147-4. doi: 10.1145/318593.318622. URL http://portal.acm.org/citation.cfm?doid=318593.318622.

Tianxiang Lu, Stephan Merz, and Christoph Weidenbach. Towards Verification of the Pastry Protocol Using TLA +. In Roberto Bruni and Juergen Dingel, editors, *Formal Techniques for Distributed Systems*, volume 6722, pages 244–258. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-21460-8 978-3-642-21461-5. URL http://link.springer.com/10.1007/978-3-642-21461-5_16.

Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer, New York, 1995. ISBN 978-0-387-94459-3.

Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. *ACM SIGPLAN Notices*, 40(1):378–391, January 2005. ISSN 03621340. doi: 10.1145/1047659.1040336. URL http://portal.acm.org/citation.cfm?doid=1047659.1040336.

Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The Java unsafe API in the wild. pages 695–710. ACM Press, 2015. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814313. URL http://dl.acm.org/citation.cfm?doid=2814270.2814313.

K. L. McMillan and D. K. Probst. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1):45–65, January 1995. ISSN 0925-9856, 1572-8102. doi: 10.1007/BF01384314. URL http://link.springer.com/10.1007/BF01384314.

Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic, Boston, 1993. ISBN 978-0-7923-9380-1.

Jeroen Meijer. Pnml2pins, 2015. URL https://github.com/utwente-fmt/pnml2pins. (Accessed 2017-03-28).

Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. pages 267–275. ACM Press, 1996. ISBN 978-0-89791-800-8. doi: 10.1145/248052.248106. URL http://portal.acm.org/citation.cfm?doid=248052.248106.

Stefan Möding. Analyze System Scalability in R with the Universal Scalability Law, 2016a. URL https://cran.r-project.org/web/packages/usl/vignettes/usl.pdf. (Accessed 2017-05-20).

Stefan Möding. Usl: Analyze System Scalability with the Universal Scalability Law, 2016b. URL https://cran.r-project.org/package=usl. (Accessed 2017-05-22).

Adam Morrison. Scaling Synchronization in Multicore Programs. *acmqueue*, 14(4):20, 2016. URL http://dl.acm.org/citation.cfm?id=2984629.2991130.

Chris Newcombe. Why Amazon Chose TLA+. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Alfred Kobsa, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Demetri Terzopoulos, Doug Tygar, Gerhard Weikum, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, volume 8477, pages 25–39. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. ISBN 978-3-662-43651-6 978-3-662-43652-3. URL http://link.springer.com/10.1007/978-3-662-43652-3_3.

Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58(4):66–73, March 2015. ISSN 00010782. doi: 10.1145/2699417. URL http://dl.acm.org/citation.cfm?doid=2749359.2699417.

Chris Newland. Log analyser and visualiser for Java HotSpot JIT compiler, 2013. URL https://github.com/AdoptOpenJDK/jitwatch. (Accessed 2017-04-19).

H. A. Oldenkamp. Probabilistic model checking : A comparison of tools. May 2007. URL http://essay.utwente.nl/591/.

Oracle. Oracle Java SE Support Roadmap, 2015. URL http://www.oracle.com/technetwork/java/eol-135779.html. (Accessed 2017-04-05).

Oracle. Java Platform, Standard Edition Java Flight Recorder Runtime Guide, 2016. URL https://docs.oracle.com/javacomponents/jmc-5-5/jfr-runtime-guide/about.htm#JFRUH170. (Accessed 2016-12-16).

Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, Mass, 1994. ISBN 978-0-201-53082-7.

Radek Pelánek. BEEM: Benchmarks for Explicit Model Checkers. In Dragan Bošnački and Stefan Edelkamp, editors, *Model Checking Software*, volume 4595, pages 263–267. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007. ISBN 978-3-540-73369-0. URL http://link.springer.com/10.1007/978-3-540-73370-6_17.

Radek Pelánek. Properties of state spaces and their applications. *International Journal on Software Tools for Technology Transfer*, 10(5):443–454, October 2008. ISSN 1433-2779, 1433-2787. doi: 10.1007/s10009-008-0070-5. URL http://link.springer.com/10.1007/s10009-008-0070-5.

Yehoshua Perl, Alon Itai, and Haim Avni. Interpolation search—a log logN search. *Communications of the ACM*, 21(7):550–553, July 1978. ISSN 00010782. doi: 10.1145/359545.359557. URL http://portal.acm.org/citation.cfm?doid=359545.359557.

Tim Peters. Timsort description, 2011. URL http://svn.python.org/projects/python/trunk/Objects/listsort.txt. (Accessed 2016-11-17).

James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, N.J, 1981. ISBN 978-0-13-661983-3.

Amir Pnueli. The temporal logic of programs. pages 46–57. IEEE, September 1977. doi: 10.1109/SFCS.1977.32. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4567924.

Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-, and space-efficient bloom filters. *Journal of Experimental Algorithmics*, 14:4.4, December 2009. ISSN 10846654. doi: 10.1145/1498698.1594230. URL http://portal.acm.org/citation.cfm?doid=1498698.1594230.

John H. Reif. Depth-First search is inherently sequential. 1985. URL https://users.cs.duke.edu/~reif/paper/dfs.ptime.pdf.

Mark Reinhold. JEP 260: Encapsulate Most Internal APIs, 2015. URL http://openjdk.java.net/jeps/260. (Accessed 2017-04-11).

John Rose. JEP 169: Value Objects, 2015. URL http://openjdk.java.net/jeps/169. (Accessed 2017-01-30).

Keith Schwarz. Smoothsort Demystified, 2011. URL http://www.keithschwarz.com/smoothsort/. (Accessed 2016-12-15).

Hermann Schweizer, Maciej Besta, and Torsten Hoefler. Evaluating the Cost of Atomic Operations on Modern Architectures. pages 445–456. IEEE, October 2015. ISBN 978-1-4673-9524-3. doi: 10.1109/PACT.2015.24. URL http://ieeexplore.ieee.org/document/7429327/.

Robert Sedgewick and Kevin Daniel Wayne. *Algorithms*. Addison-Wesley, Upper Saddle River, NJ, 4th ed edition, 2011. ISBN 978-0-321-57351-3.

A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *Journal of the ACM*, 32(3):733–749, July 1985. ISSN 00045411. doi: 10.1145/3828.3837. URL http://portal.acm.org/citation.cfm?doid=3828.3837.

SonarQube. Quality Gate for tlaplusTLA+ Tools and TLA Toolbox mku-openaddressing, 2017. URL https://sonarqube.com/code?id=tlatoolbox:parent:mku-openaddressing&selected=tlatoolbox:org.lamport.tlatools:mku-openaddressing:src/tlc2/tool/fp. (Accessed 2017-04-19).

Robert Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2):146–160, June 1972. ISSN 0097-5397, 1095-7111. doi: 10.1137/0201010. URL http://epubs.siam.org/doi/abs/10.1137/0201010.

Robert E. Tarjan. The concurrent algorithm, 2015. (Accessed 2017-01-31).

The Hotspot Group. http://openjdk.java.net/groups/hotspot/, 1999. URL http://openjdk.java.net/groups/hotspot/. (Accessed 2017-04-19).

Martin Thompson. Lock-Based vs Lock-Free Concurrent Algorithms, 2013. URL http://mechanical-sympathy.blogspot.de/2013/08/lock-based-vs-lock-free-concurrent.html. (Accessed 2016-11-11).

Linus Torvalds. Transparent Hugepage Support, 2016a. URL https://www.kernel.org/doc/Documentation/vm/transhuge.txt. (Accessed 2016-12-19).

Linus Torvalds. CPU frequency and voltage scaling code in the Linux(TM) kernel, 2016b. URL https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt. (Accessed 2016-12-19).

Linus Torvalds. Documentation for /proc/sys/kernel/*, 2016c. URL https://www.kernel.org/doc/Documentation/sysctl/kernel.txt. (Accessed 2016-12-19).

Philippas Tsigas and Yi Zhang. Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. page 55. ACM Press, 2002. ISBN 978-1-58113-563-3. doi: 10.1145/584369.584378. URL http://portal.acm.org/citation.cfm?doid=584369.584378.

Steven van der Vegt and Alfons Laarman. A Parallel Compact Hash Table. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Zdeněk Kotásek, Jan Bouda, Ivana Černá, Lukáš Sekanina, Tomáš Vojnar, and David Antoš, editors, *Mathematical and Engineering Methods in Computer Science*, volume 7119, pages 191–204. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-25928-9 978-3-642-25929-6. URL http://link.springer.com/10.1007/978-3-642-25929-6_18.

Wikipedia. List of model checking tools — Wikipedia, The Free Encyclopedia. 2017a. URL https://en.wikipedia.org/w/index.php?title=List_of_model_checking_tools&oldid=770451387.

Wikipedia. Memory hierarchy — Wikipedia, The Free Encyclopedia. 2017b. URL https://en.wikipedia.org/w/index.php?title=Memory_hierarchy&oldid=775298110. [Online; accessed 15-May-2017].

Yuan Yu. Tlc2.tool.fp.DiskFPSet, 1999. URL https://github.com/tlaplus/tlaplus/blob/master/tlatools/src/tlc2/tool/fp/DiskFPSet.java#L43-L47. (Accessed 2016-12-15).

Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model Checking TLA+ Specifications. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Laurence Pierre, and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods*, volume 1703, pages 54–66. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. ISBN 978-3-540-66559-5 978-3-540-48153-9. URL http://link.springer.com/10.1007/3-540-48153-2_6.

Acronyms

BFS breath-first search, *Glossar:* BFS

CAS Compare and Swap, *Glossar:* CAS

DFS depth-first search, *Glossar:* DFS

LP Linear Probing, *Glossar:* LP

LTL Linear-Time Temporal Logic, *Glossar:* LTL

NUMA Non Uniform Memory Access, *Glossar:* NUMA

OA Open Addressing, *Glossar:* OA

SC Separate Chaining, *Glossar:* SC

TLA$^+$ Temporal Logic of Actions, *Glossar:* TLA$^+$

TLAPS The TLA$^+$ proof system.

TLA Temporal Logic of Actions, *Glossar:* TLA

TLC$_{legacy}$ TLA Checker, *Glossar:* TLC$_{legacy}$

TLC$_{proto}$ TLA Checker, *Glossar:* TLCTLC$_{proto}$

TLC$_{stripe}$ TLA Checker, *Glossar:* TLC$_{stripe}$

TLC TLA Checker, *Glossar:* TLC

USL Universal Scalability Law, *Glossar:* USL

PlusCal PlusCal, *Glossar:* PlusCal

Glossary

C_{legacy} The seen set C with SC.

C_{proto} The lock-free seen set C with OA.

C_{stripe} The seen set C with SC and lock-striping.

BFS Breadth-first search is an algorithm to traverse trees or graphs. A vertex's siblings are explored before its childs.

CAS Compare and Swap is a hardware synchronization primitive that reads a memory location and only writes a new value iff the read returned an expected value.

DFS Depth-first search is an algorithm to traverse trees or graphs. A vertex's childs are explored before its siblings.

DiVinE DiVinE is an explicit state model checker for real-world programming languages.

Insertion Sort Insertion Sort is an adaptive and stable sorting algorithm. It supports sorting in-place and has an O(n) best-case runtime complexity which it achieves on almost sorted data (bounded disorder). With bounded disorder, its locality allows it to be parallelized easily.

Java Java is a general purpose computing language.

LP Linear Probing is a possible sequence with Open Addressing.

LTL The Linear-Time Temporal Logic is a temporal logic proposed by Pnuelli for the formal verification of programs. It has a richer set of temporal operators compared to TLA.

LTSMin LTSMin is a language independent model checker.

NUMA A hardware architecture with memory access times depending on theprocessor proximity.

OA Open Addressing is a collision resolution strategy used in Hash Tables.

SC Separate Chaining is a collision resolution strategy used in Hash Tables.

Spin SPIN is an explicit state model checker.

TLA$^+$ The Temporal Logic of Actions is a formal specification language.

TLAPS The TLA$^+$ proof system.

TLA The Temporal Logic of Actions is the mathematical foundation to TLA$^+$. It is used to describe behaviors of concurrent systems.

TLC TLC is an explicit state model checker for the TLA$^+$ specification language. It is written in Java.

USL The universal scalability law is mathematical model to quantify scalability.

jpf Java Pathfinder is an explicit state model checker for Java byte code.

`sun.misc.Unsafe` sun.misc.Unsafe is a vendor specific API which exposes the CAS operation to JAVA applications.

action An action is a relation between a pair of states.

action predicate An action predicate is a boolean valued expression on two states s,t, where t is a successor to s.

behavior A behavior is an infinite sequence of states.

bucket SC collision resolution stategy resolves collisions by storing colliding elements in a collision bucket.

cache line A cache line is a fixed sized memory region copied from primary memory to a cache.

cache miss A cache miss occurs when a cache line has to be fetched.

coherence Coherence is a measurement to maintain a system's consistency.

collision A hash table collision occurs when two or more elements are hashed to the same index.

concurrent system A concurrent system can execute multiple tasks at the same time.

consistency model A consistency model allows one to reason about the order of operations in a concurrent system.

contention A mutual exclusive resource, such as a variable exhibit contention, when multiple processes access it concurrently.

counterexample A counterexample is a behavior that violates a safety property or liveness property φ, such that the bahvior satisfies $\neg\varphi$.

disjoint access parallel A set of processes can independently make forward progress if they access disjoint memory regions.

disk-based model checking Disk-based model checking is capable of checking model sizes exceeding the size of primary memory.

embarrassingly parallel problem An embarrassingly parallel problem is contention or coherence free and therefor gains ideal speedup from parallelization.

explicit state model checking *Glossar:*

find-or-put An composite operation of a hash table to insert an element unless it is already present.

fingerprint A fingerprint uniquely identifies a state.

fingerprint set *Glossar:* seen set C

forest The (state) forest T is used to construct the counterexample.

forward progress Forward progress means a process is not contended and makes progress towards its goal.

hash table An associative array which maps keys to values.

liveness property A liveness property is a property that is true eventually, perhaps once or multiple times.

load factor A measurement to indicate the number of elements stored in the hash table relative to its size.

lock A lock achieves mutual exclusion of two or more processes to a shared resource.

lock-freedom A concurrent system is said to be lock-free, iff no process can block other processes from making forward progress.

lock-striping Lock-striping is a programming pattern where n locks guard a shared resource partitioned in m partitions with $n<=m$.

model checker An (explicit state) Model Checker exhaustively and automatically checks, if a model meets a formal specification.

model checking problem The model checking problem answers the question if a system adhers to its specification.

next state relation A next state relation is a left-total mapping from states to states.

PlusCal PlusCal is an algorithm language and a dialect of TLA$^+$ geared to replace informal pseudo-code. It can be checked with TLC

primed variable A primed variable denotes the variable's assigned value in the successor state.

safety property A safety property is a property that is always true.

scalability A concurrent system is scalable, if its throughput increases when resources are added.

seen set C The seen set C maintains the set of explored states.

state A state is an assignment of values to variables.

state graph A state graph is a directed cyclic graph. A node represents a state and an arc represents a transition.

state predicate An state predicate is a specialized action predicate on a single state.

state queue *Glossar:* unseen set S

state space explosion State space explosion is the combinatorial blowup of the state space.

successor The state t is a successor state of state s, iff there is a transition from s to t.

throughput A measurement for the amount of work a concurrent system gets accomplished within a time interval.

TLA Toolbox The TLA Toolbox is the integrated development environment for TLA$^+$.

TLC module overwrite With a TLC module overwrite, a user can implement a TLA$^+$ operator in JAVA.

transition A transition is a mapping from a state s to a successor state t.

unseen set S The unseen set S maintains the set of newly generated, unexplored states.

utilization Utilization *util* is a quality measurement which quantifies the effictive usable space of a hash table .

worker In TLC, a worker is a thread running state exploration.

Index

A. Language Constructs

This section introduces those TLA$^+$ and PlusCal language constructs used in the specifications below, which we believe most readers to be unfamiliar with. This list neither attempts to be exhaustive nor tries to rigorously specify the syntax or semantics of the TLA$^+$ and PlusCal languages constructs. Instead, we define the subset required to understand the specifications below. We leave the complete and rigorous formal specification to the canonical literature on TLA$^+$ and PlusCal by Lamport [1994, 2003, 2009, 2014].

A TLA$^+$ specification appears in a module file, which defines the scope of the specification. PlusCal is only allowed to appear nested inside a comment of a TLA$^+$ module file. The keyword *algorithm* indicates the beginning of the PlusCal code. The PlusCal translator, transpiles PlusCal code into TLA$^+$, which is directly inserted after the comment containing the PlusCal. We start with TLA$^+$:

MODULE M Indicates the beginning of the declaration of module M.

EXTENDS P, Q Makes the TLA$^+$ definitions in module P and Q available to the current module. *Integers*, *Sequences*, and *FiniteSets* are standard modules of TLA$^+$ bundled with TLC.

*CONSTANT id*1, *id*2 A declaration of constants defined by the model.

LET $id \triangleq expr_{let}$ *IN* $expr_{in}$ Locally defines the identifier id to be the expression $expr_{let}$. The identifier id can then be used in $expr_{in}$.

CHOOSE $s \in S : P(s)$ Deterministically selects an element from the set S, for which P holds. If P holds for more than one element of S, it is undefined, which one exactly is selected. However, it is guaranteed to always be the same element. If P holds for none of the elements in S, the value of the complete *CHOOSE* expression is unspecified.

$\wedge \quad A$

$\wedge \quad \vee \quad B$ Parenthesis free, but indentation significant variant of $A \wedge (B \vee C)$.

$\quad \vee \quad C$

Temporal Operators:[1]

$\Box P$ A*lways*, requiring the property P to hold always.

$\Diamond P$ E*ventually*, defined to be $\neg\Box\neg P$.

$\Diamond\Box P$ *Eventually* the property P will *always* hold.

Sets:

$\{\}$ The empty set.

UNION S The set of all elements in the set of sets S, e.g. *UNION* $\{\{1,2\},\{2\}\} = \{1,2\}$.

SUBSET S A set of all subsets of set S, including S itself and the empty set. In other words, the powerset of S.

Cardinality (S) An operator defined by the *FiniteSets* standard module which returns the cardinality of the set S.

Functions:

$[a \mapsto 42]$ A function, whose domain is $\{a\}$ and whose image is $\{42\}$. Function application is written as $f\,[a]$. A function is different to an operator, in that its domain is a set (compare operator *Cardinality* (S) whose domain is a set of sets). A function can be assigned to a variable.

DOMAIN fcn The domain of a function *fcn*, e.g. *DOMAIN* $[a \mapsto 42] = \{a\}$.

$[s \in S \mapsto 42]$ A function whose domain is the set S and whose image is $\{42\}$. If S happens to be the set $\mathbb{N} \setminus \{0\}$, in programming languages this would be considered a one-indexed array with all elements set to 42. In TLA$^+$, such a function is called sequence.

$\langle 1,2,3 \rangle$ A sequence of the elements $1, 2, 3$.

[1] See Manna and Pnueli [1995, p. 42ff] for a formal introduction of temporal logic operators.

$\langle\rangle$ The empty sequence.

$1..3$ The sequence of integers from 1 to 3, thus $1..3 = \langle 1, 2, 3\rangle$.

$A \circ B$ The concatenation of the sequence A with the sequence B. For example, $\langle 1, 2, 3\rangle \circ \langle 3, 4, 5\rangle = \langle 1, 2, 3, 3, 4, 5\rangle$

$[S \to T]$ The set of functions, that map an element in set S to an element in set T. E.g.

$$[\{2, 3\} \to \{4, 5\}] = \{[2 \mapsto 4, 3 \mapsto 4], [2 \mapsto 4, 3 \mapsto 5], [2 \mapsto 5, 3 \mapsto 4] [2 \mapsto 5, 3 \mapsto 5]\}$$

Head (seq) The *Head* operator returns the first element of the sequence *seq*.

Tail (seq) The *Tail* operator returns the sequence *seq* without the *Head* (seq) element of *seq*.

Len (seq) The *Len* operator returns the length of *seq*, i.e. the number of elements in *seq*.

Lamport [2009] introduced PlusCal as executable pseudo-code. Thus, in contrast to TLA$^+$, control flow in PlusCal is made explicit; just like in other imperative programming languages.

algorithm Keyword to mark the beginning of an algorithm written in PlusCal code inside a TLA$^+$ comment.

variables var = val; The *variables* keyword begins the definition of a set of variables and their initial values. The *variables* declaration following *algorithm*, declares the global variables of the PlusCal specification. E.g. *variables* $a = 42$; assigns the variable a the initial value 42. The semicolon terminates the list of variable declarations.

var := *val* Assigns the value *val* to the variable *var* at the current state. E.g. $a := 23$ assigns 23 to *var* in a state (see labels below). The variable *var* has to be declared within a *variables* declaration before.

procedure proc ... {*body*} The declaration of a procedure identified by *proc*. A procedure can have its own set of variables defined by the *variables* statement immediately following the procedure declaration. *body* reflects the block of code executed by the procedure.

call proc Begins execution of the procedure *proc*, similar to other imperative programming languages. Once *proc* terminates, control flow returns to the caller.

process $(p \in S)$ {*body*} Keyword to indicate the beginning of a process declaration in a concurrent algorithm. The $p \in S$ causes the creation of a process for each element p of S. A *variables* statement appearing inside a process, declares the set of variables visible within the scope of the process. A *fair* process guarantees to make progress if possible.[2] Without a process declaration, a PlusCal specification describes a sequential algorithm.

await cond Suspends execution of the process in whose scope await is declared, until *cond* holds.

with $(s \in S)$ {*body*} Nondeterministically picks an element of S identified by s. The identifier s can be used in the body of *with*. However, *with* is different to *LET* described above. Where *LET* is a local definition, a *with* statement increases the set of behaviors. Likewise, *with* is different to *CHOOSE*, in that it picks all elements instead of just one. In other words, with regards to the possible behaviors of the specification, a *with* results in a behavior for each element in S.

macro MCR (*params*) {*body*} A text macro called *MCR*, whose declaration is syntactically expanded in places where it is called.

lbl : ... A label *lbl* which groups the block of PlusCal code, that follows *lbl* up to the next label, into an atomic operation with regards to parallelism. In other words, labels control the amount of concurrency allowed by the specification. The execution of a label, i.e. the block of code grouped by the label, corresponds to a state in the behaviors of the specification.

[2] See Apt et al. [1987] for a formal definition of fairness.

B. Specifications

The following specifications have been printed with the TLA$^+$ Pretty-Printer [Lamport, 2015b]. The Pretty-Printer is a tool to typeset TLA$^+$ and PlusCal specifications, which are written in ASCII, with LaTeX. The conversion from ASCII into LaTeX might not always be optimal. We apologize for any formatting problems, such as excessive use of white-spaces, created by this conversion.

The ASCII pendants of all specifications are part of the CD-ROM. The ASCII pedants also include the transpiled TLA$^+$ which corresponds to the PlusCal code. We omitted the transpiled TLA$^+$ from the pretty printed specifications below, for brevity reasons. For technical reasons, we also had to omit the surrounding comment into which PlusCal is normally embedded.

If possible, a reader should study the ASCII pendant of the specifications below inside the TLA Toolbox, which provides syntax highlighting and code navigation.

B.1. Model Checker

This is the complete TLC's algorithm to check safety properties discussed in section 2.2.3. We checked the algorithm with TLC on a small set of test inputs, that is manually defined state graphs. In this sequential algorithm we could have omitted labels completely. We choose to use labels, to improve readability.

MODULE *TLCMC*

EXTENDS *Integers*, *Sequences*, *FiniteSets*

Convertes the given Sequence seq into a Set of all the Sequence's elements. In other words, the image of the function that seq is.

$SeqToSet(seq) \triangleq \{seq[i] : i \in 1 \,..\, Len(seq)\}$

Returns a Set of those permutations created out of the elements of Set set which satisfy Filter.

$SetToSeqs(set, Filter(\_)) \triangleq$ UNION $\{\{perm \in [1 \,..\, Cardinality(set) \to set] :$

A filter applied on each permutation

generated by [S -> T]

$Filter(perm)\}\}$

Returns a Set of all possible permutations with distinct elemenents created out of the elements of Set set. All elements of set occur in in the sequence.

$SetToDistSeqs(set) \triangleq SetToSeqs(set,$
$\quad$ LAMBDA $p : Cardinality(SeqToSet(p)) = Cardinality(set))$

A (state) graph G is a directed cyclic graph.
A graph G is represented by a record with 'states' and 'actions' components, where G.states is the set of states and G.actions[s] is the set of transitions of s – that is, all states t such that there is an action (arc) from s to t.

$IsGraph(G) \triangleq \land \{\text{“states”}, \text{“initials”}, \text{“actions”}\} = \text{DOMAIN } G$
$\land G.actions \in [G.states \to \text{SUBSET } G.states]$
$\land G.initials \subseteq G.states$

A set of all permutations of the inital states of G.

$SetOfAllPermutationsOfInitials(G) \triangleq SetToDistSeqs(G.initials)$

A Set of successor states which are not in $seen \cup state$. Thus, they are the unseen successors of state.

$SuccessorsOf(state, SG, seen) \triangleq \{successor \in SG.actions[state] :$
$\quad successor \notin (seen \cup \{state\})\}$

The predecessor of v in a forest t is the first element of the pair «predecessor, successor» nested in a sequence of pairs. In an actual implementation such as TLC, pair[1] is rather an index into t than an id of an actual state.

$Predecessor(t, v) \triangleq SelectSeq(t, \text{LAMBDA } pair : pair[2] = v)[1][1]$

CONSTANT $StateGraph, ViolationStates, null$

ASSUME The given StateGraph is actually a graph
$\lor IsGraph(StateGraph)$
The violating states are vertices in the state graph.
$\lor ViolationStates \subseteq StateGraph.states$

--algorithm $ModelChecker\{$

variables

A FIFO containing all unexplored states. A simple set provides no order, but TLC should explore the StateGraph in either BFS (or DFS => LIFO).
Note that S is initialized with each

```
        possible permutation of the initial states
        here because there is no defined order
        of initial states.
      S ∈ SetOfAllPermutationsOfInitials(StateGraph),
        A set of already explored states.
      C = {},
        The state currently being explored in scsr
      state = null,
        The set of state's successor states
      successors = {},
        Counter
      i = 1,
        A path from some initial state ending in a
        state in violation.
      counterexample = ⟨⟩,
        A sequence of pairs such that a pair is a
        sequence «predecessor, successors».
      T = ⟨⟩ ;
  {
Check initial states for violations. We could be clever and check the inital states as part of the second
while loop. However, we then either check all states twice or add unchecked states to S.
  init: while (i ≤ Len(S)){
      state := Head(S) ;
        state is now fully explored,
        thus exclude it from any
        futher exploration if graph
        exploration visits it again
        due to a cycle.
      C := C ∪ {state} ;
      i  := i + 1 ;
      if (state ∈ ViolationStates){
            counterexample := ⟨state⟩ ;
             Terminate model checking
            goto trc ;
      } ;
  } ;
```

Explores all successor states until no new successors are found or a violation has been detected.

$scsr$: **while** $(Len(S) \neq 0)\{$

Assign the first element of
S to state. state is
what is currently being checked.

$state := Head(S)$;

Remove state from S.

$S := Tail(S)$;

For each unexplored successor 'succ' do:

$successors := SuccessorsOf(state,\ StateGraph,\ C)$;

if $(successors = \{state\})\{$

Iff there exists no successor besides
the self-loop, the system has reached
a deadlock state.

$counterexample := \langle null \rangle$;

goto trc ;

} ;

$each$: **while** $(successors \neq \{\})\{$

with $(succ \in successors)\{$

Exclude succ in this while loop.

$successors := successors \setminus \{succ\}$;

Mark successor globally visited.

$C := C \cup \{succ\}$;

Append succ to T and add it
to the list of unexplored states.

$T := T \circ \langle\langle state,\ succ \rangle\rangle$;

$S := S \circ \langle succ \rangle$;

Check state for violation of a
safety property (simplified
to a check of set membership.

if $(succ \in ViolationStates)\{$

$counterexample := \langle succ \rangle$;

Terminate model checking

```
                goto trc ;
              } ;
          } ;
      } ;
  } ;
```

Model Checking terminated without finding a violation.

```
  assert S = ⟨⟩ ;
  goto Done ;
```

Create a counterexample, that is a path from some initial state to a state in ViolationStates. In the Java implementation of TLC, the path is a path of fingerprints. Thus, a second, guided state exploration resolves fingerprints to actual states.

```
  trc : while (TRUE){
        if (Head(counterexample) ∉ StateGraph.initials){
          counterexample := ⟨Predecessor(T, Head(counterexample))⟩ ∘ counterexample ;
        }else {
          assert counterexample ≠ ⟨⟩ ;
          goto Done ;
        } ;
  } ;
   }
}
```

B.2. Open Addressing

Full listing of the PlusCal specification discussed in chapter 7. Ideally each line of the PlusCal code below would be labeled — with the exclusion of *CAS* — to correctly reflect the desired parallelism. However, this would render model checking of B.2 infeasible, due to the attached state space increase. Thus, we omitted those labels where the subsequent block of code is restricted to assignments to variables local to the process or *history*.

MODULE *OpenAddressing*

EXTENDS *Sequences*, *FiniteSets*, *Integers*

K: The overall number of fingerprints that fit into the table. fps: The set of fingerprints to be inserted into the hash table. empty: An empty (model) value. Used to mark an unoccupied table element. Writer: The set of processes/threads which insert fingerprints. L: The probing limit.

CONSTANT K, fps, $empty$, $Writer$, L

K is a positive natural. emtpy is different from all elements in fps. fingerprints are natural numbers and can be well-ordered.

ASSUME $\land K \in (Nat \setminus \{0\})$
$\land \forall fp \in fps : fp \in (Nat \setminus \{0\})$
$\land empty \notin fps$
$\land (2 * L) \leq K$

The image of the function F.

$Image(F) \triangleq \{F[x] : x \in \text{DOMAIN } F\}$

The element of position Len(seq) of a sequence seq.

$last(seq) \triangleq seq[Len(seq)]$

The largest element in the sequence, assuming sequence to be sorted in ascending order.

$largestElem(sortedSeq) \triangleq \text{IF } sortedSeq = \langle\rangle \text{ THEN } 0 \text{ ELSE } last(sortedSeq)$

All elements of seq1 smaller than elem and the largest element in seq2.

$subSeqSmaller(seq1, seq2, elem) \triangleq SelectSeq(seq1, \text{LAMBDA } p : p < elem \land p > largestElem(seq2))$

All elements of seq1 larger than the largest element in seq2.

$subSeqLarger(seq1, seq2) \triangleq \text{IF } seq2 = \langle\rangle \text{ THEN } seq1$

$$\text{ELSE}\ \ SelectSeq(seq1,\ \text{LAMBDA}\ p :\ p > largestElem(seq2))$$

TRUE iff the sequence seq contains the element elem.

$$containsElem(seq,\ elem) \triangleq elem \in Image(seq)$$

The minimum and maximum element in set S.

$$min(S) \triangleq \text{CHOOSE}\ s \in S : \forall\, a \in S : s \leq a$$
$$max(S) \triangleq \text{CHOOSE}\ s \in S : \forall\, a \in S : s \geq a$$

The smaller of the two values.

$$minimum(a,\ b) \triangleq \text{IF}\ a < b\ \text{THEN}\ a\ \text{ELSE}\ \ b$$

The given index i modulo the sequences' length.

$$mod(i,\ len) \triangleq \text{IF}\ i\%len = 0\ \text{THEN}\ len\ \text{ELSE}\ \ (i\%len)$$

Logical bit-shifting to the right (shifts in zeros from the left/MSB). TLC's standard division does not round towards zero, thus this is specified recursively, manually taking care of rounding.

$$\text{RECURSIVE}\ shiftR(\_,\ \_)$$
$$shiftR(n,\ pos) \triangleq \text{IF}\ pos = 0\ \text{THEN}\ n$$
$$\text{ELSE}\ \ \text{LET}\ odd(z) \triangleq z\%2 = 1$$
$$m \triangleq \text{IF}\ odd(n)\ \text{THEN}\ (n-1) \div 2\ \text{ELSE}\ \ n \div 2$$
$$\text{IN}\ \ \ shiftR(m,\ pos-1)$$

Bitshifting (faster for any real implementation).

$$bitshift(fp,\ p) \triangleq \text{LET}\ k \triangleq \text{CHOOSE}\ k \in 1\,..\,K : 2^k = K$$
$$\text{IN}\ \ \ mod(shiftR(fp,\ k-1) + 1 + p,\ K)$$

Re-scale.

$$rescale(k,\ maxF,\ minF,\ fp,\ p) \triangleq \text{LET}\ f \triangleq (k-1) \div (maxF - minF)$$
$$\text{IN}\ \ \ mod((f * (fp - minF + 1)) + p,\ k)$$

Calculates an fp's index where fp $\in$ fps. p is an alternative address, such that: p $\in$ 0..P. Uses bitshifting iff K is power of two.

$$idx(fp,\ p) \triangleq \text{IF}\ \exists\, n \in 1\,..\,K : 2^n = K$$
$$\text{THEN}\ bitshift(fp,\ p)$$
$$\text{ELSE}\ \ rescale(K,\ max(fps),\ min(fps),\ fp,\ p)$$

TRUE iff the fingerprint at table position index is equal to fp or its corresponding negative fp value (marked as to be copied to external).

$$isMatch(fp,\ index,\ table) \triangleq \lor table[index] = fp$$

$$\lor table[index] = (-1 * fp)$$

TRUE iff the table at position index is empty.

$$isEmpty(index,\ table) \triangleq table[index] = empty$$

TRUE iff the table at position index is marked evicted.

$$isMarked(index,\ table) \triangleq table[index] < 0$$

A fp wrapped around if its alternate indices are beyond K and its actual index is lower than its primary idx. Another mental picture for table is a circular list and wrapped means that a fingerprint crossed the logically first position of table.

$$wrapped(fp,\ pos) \triangleq idx(fp,\ 0) > mod(pos,\ K)$$

Compare the two fingerprints fp1 and fp2 for order with regards to their numerical values and their respective positions i1 and i2.
Returns -1, iff fp2 is less than fp1. Returns 0, iff fp1 and fp2 are equal. Returns 1, iff fp1 is less than fp2.
compare considers three cases: 1) Iff either one or both fingerprints are empty, they are defined to be equal. Under the assumption of a stable sorting algorithm, fp1 and fp2 are not swapped (ELSE 0). 2) Iff neither one or both fingerprints wrapped, a basic comparison is done. A basic comparison is one, where the lower positioned fp has to be numerically lower. 3) Iff the truth values for wrapped differed, two cases have to be distinguished:
Let: $\overset{\circlearrowleft}{fps} \triangleq fp \in fps : \exists i \in Image(PS[fp]) : wrapped(fp, i)$
$\overrightarrow{fps} \triangleq fps \setminus \overset{\circlearrowleft}{fps}$
3a) Comparison when fp1 and fp2 are both in $\overset{\circlearrowleft}{fps}$. If fp1 is at a lower position (thus wrapped) and numerically lower, swap it with fp2 which is at a higher position and thus did not wrap. For example, fp1 was inserted into the table after fp2 and thus wrapped, but is numerically lower than fp2.
3b) Special case comparison required by Insertion Sort. It compares a fingerprint in $\overset{\circlearrowleft}{fps}$ with one in $\overrightarrow{fps}$. Insertion Sort compares adjacent elements. Thus, without this case two fingerprints fp1 and fpX, which are eventually handled by 3a), would not be sorted, iff fp2 is inbetween of fp1 and fpX. Thus, fp1 is swapped with fp2 meaning it moves towards the beginning of table. Eventually, all wrapped fingerprints in $\overset{\circlearrowleft}{fps}$ will form a cluster at the beginning of t and can then be sorted with 3a). In other words, we allow the wrapped fingerprints to be compacted at the beginning ot the table and non-wrapping fingerprints to be moved to higher positions.
Assumeming that the beginning of table is: «1,23,22,...,24» (assuming fps is 1..24, L=3 and K=6. Sorted, table needs to change to «1,23,24,...,22». Without 3b), Insertion Sort compares 22 to 1 and 23 to 22. The outcome would be «1,22,23,...,24», which is cleary not fully sorted. Thus, in order to handle this case, we allow IS to swap 22 and 23 with 1. As a result, table - when sorted - is «23,24,1,...,22».
Can we be sure, that non-wrapping fingerprints do not get moved out beyond the end of their probing sequence? Obvously, at most, L-1 wrapping fingerprints can be located at the beginning of table. In this case, only one non-wrapping fingerprint will be in the table, which maximally will be moved L-1 positions to the right with regards to its primary position.

$$
\begin{array}{l}
compare(fp1,\ i1,\ fp2,\ i2) \triangleq \\
\qquad \text{IF } fp1 \neq empty \wedge fp2 \neq empty \qquad \text{1)} \\
\qquad \text{THEN IF } wrapped(fp1,\ i1) = wrapped(fp2,\ i2) \qquad \text{2)} \\
\qquad\qquad \text{THEN IF } i1 > i2 \wedge fp1 < fp2 \text{ THEN } -1 \text{ ELSE } 1 \\
\qquad\qquad \text{ELSE IF } i1 < i2 \wedge fp1 < fp2 \text{ THEN } -1 \text{ ELSE} \qquad \text{3a} \\
\qquad\qquad\qquad \text{IF } i1 > i2 \wedge fp1 > fp2 \text{ THEN } -1 \text{ ELSE } 0 \qquad \text{3b} \\
\qquad \text{ELSE } 0
\end{array}
$$

--algorithm *OpenAddressing*

table: The actual hash table specified as a TLA+ sequence. history: An auxiliary (history) variable unrelated to the actual hash table specification. It just records the inserted fingerprints to be verified by Inv. An implementation won't need history. external: The external storage where fingerprints are eventually evicted to. outer/inner: Index variables local to the sort action. P: The number of times an alterante index is to be tried.

```
{variable table = [i ∈ 1 .. K ↦ empty],
         external = ⟨⟩,
         newexternal = ⟨⟩,
         evict = FALSE,   AtomicBoolean in Java
         waitCnt = 0,     CyclicBarrier in Java
         history  = {} ;

  Atomically compare and swap an element of table.
  Atomicity is implicit due to the absence of labels.
 macro CAS(result, pos, expected, new){
   if (table[pos] = expected){
      table[pos] := new ;
      result := TRUE
   }else {
      result := FALSE
   }
 }

 procedure Evict()
   variables ei = 1, ej = 1, lo = 0 ; {
                Insertion sort.
   strIns:     while (ei ≤ K + L){
                 lo := table[mod(ei + 1, K)] ;
   nestedIns:    while (compare(lo, mod(ei + 1, K),
                                table[mod(ej, K)], mod(ej, K)) ≤ −1){
                   table[mod(ej + 1, K)] := table[mod(ej, K)] ;
                   if (ej = 0){
                      ej := ej − 1 ;
                      goto set ;
                   }else {
                      ej := ej − 1 ;
                   } ;
```

```
                    } ;
  set:              table[mod(ej + 1, K)] := lo ;
                    ej := ei + 1 ;
                    ei := ei + 1 ;
                  } ;
                  ei := 1 ;

                  Write to external storage.
  flush:          while (ei ≤ K + L){
                      lo := table[mod(ei, K)] ;
                      if (lo ≠ empty ∧
                          lo > largestElem(newexternal) ∧
                          ((ei ≤ K ∧ ¬wrapped(lo, ei)) ∨
                           (ei > K ∧ wrapped(lo, ei)))){
                          Copy all smaller fps than lo from
                          secondary to newexternal.
                        newexternal := Append(newexternal ∘
                              subSeqSmaller(external, newexternal, lo), lo) ;
                          Mark table[mod(cpy,table)] as being
                          written to external.
                        table[mod(ei, K)] := lo ∗ ( − 1) ;
                      } ;
                      ei := ei + 1 ;
                  } ;
                   Append remainder of external to newexternal and
                   assign newexternal to external.
                  external := newexternal ∘
                                subSeqLarger(external, newexternal) ;
                  newexternal := ⟨⟩ ;
  rtrn:           return ;
}

 A weak fair process.
fair process (p ∈ Writer)
  variables fp = 0, index = 0, result = FALSE, expected = − 1 ; {
  pick: while (TRUE){
```

```
                No deadlock once all fingerprints have been inserted.
              if ((fps \ history) = {}){
                goto Done;
              }else {
                 Select some fp to be inserted
                with (f ∈ (fps \ history)){fp := f ; } ;
              } ;
put:          index := 0 ;
              result := FALSE ;
                Set expected to infinity. expected is reused when
                the algorithm runs a primary lookup and finds a
                position which is either EMPTY or isMarked(...).
                expected stores the (open) position for later use
                where the fp is inserted. Maximally, a position
                can be K + L, thus expected is set to K + L + 1;
                as an approximation of infinity.
              expected := L ;
                Wait for eviction thread to do its work.
              if (evict){
                  waitCnt := waitCnt + 1 ;
waitEv:           await evict = FALSE ;
endWEv:           waitCnt := waitCnt − 1 ;
                  goto put
              } ;

                Check external unless empty. First though, we do
                a primary lookup in case the fp in question has not
                been evicted to external yet.
chkSnc:       if (external ≠ ⟨⟩){
                  Primary lookup.
cntns:            while (index < L){
                    if (isMatch(fp, idx(fp, index), table)){
                      goto pick
                    }else {
                      if (isEmpty(idx(fp, index), table)){
```

```
                    Found an EMPTY position which proves
                    that fp cannot be found at higher
                    positions. Thus, no need to continue.
                   expected := minimum(expected, index);
                   goto onSnc;
                 }else {
                   if (isMarked(idx(fp, index), table)){
                       None of the lower positions has
                       fp, thus keep the lowest position
                       for the second loop as the start
                       index. No point in checking known
                       lower positions in the loop again.
                     expected := minimum(expected, index);
                     index := index + 1;
                   }else {
                     index := index + 1
                   }
                 }
               }
             };

             External lookup.
onSnc:     if (containsElem(external, fp)){
             goto pick
           }else {
              Have next loop start at expected determined
              by previous loop.
             index := expected;
              Re-init expected to be used for its alternate purpose.
             expected := -1;
           };
         };

         Put inserts the given fp into the hash table by sequentially
         trying the primary to the P's alternate position.
insrt:   while (index < L){
```

```
                expected := table[idx(fp, index)];
                if (expected = empty ∨
                    (expected < 0 ∧ expected ≠ (−1) ∗ fp)) {
cas:                CAS(result, idx(fp, index), expected, fp);
                    if (result){
                       history := history ∪ {fp};
                       goto pick
                    }else {
                        Has been occupied in the meantime,
                        try to find another position.
                      goto insrt
                     }
                };

                  Has fp been inserted by another process? Check isMatch
                  AFTER empty and on-external because of two reasons:
                  a) Thread A finds table[pos] to be empty but fails
                  to CAS fpX. Thread B concurrently also finds
                  table[pos] to be empty and succeeds to CAS fpX.
                  b) Iff table[pos] is empty or -1, higher positions
                  cannot be a match.
isMth:          if (isMatch(fp, idx(fp, index), table)){
                   goto pick
                }else {
                   index := index + 1;
                };
            }; end of while/insrt

                We failed to insert fp into a full table, thus try
                to become the thread that evicts to external.
                The label tryEv makes sure, that the read and
                write occur atomically. An imlementation has to
                CAS or use some other mechanism to control
                concurrenry.
tryEv:      if (evict = FALSE){
                CAS evict!
```

```
                evict := TRUE ;
                 Wait for all other insertion threads and
                 the one reader to park.
    waitIns:    await waitCnt = Cardinality(Writer) − 1 + Cardinality(Reader) ;
                call Evict() ;
    endEv:      evict := FALSE ;
                goto put ;
              }else {
                goto put
              }
          }
        } end while/pick
    }
}
```

BEGIN TRANSLATION

TLAPlus translation partially omitted

VARIABLES $table$, $external$, $newexternal$, $evict$, $waitCnt$, $history$, pc, $stack$,
ei, ej, lo, fp, $index$, $result$, $expected$

$$Termination \triangleq \Diamond(\forall\, self \in ProcSet : pc[self] = \text{“Done”})$$

END TRANSLATION

$$\begin{aligned} contains(f,\, t,\, seq,\, Q) \triangleq\ & \lor \exists\, i \in 0\,..\,Q : isMatch(f,\, idx(f,\, i),\, t) \\ & \lor \exists\, i \in 1\,..\,Len(seq) : seq[i] = f \\ & \lor \text{IF } f \in (Image(lo) \setminus \{0\}) \text{ THEN } evict = \text{TRUE} \\ & \qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE FALSE} \end{aligned}$$

All fingerprint in history are (always) members of the seen set C, all (fps history) never are. During eviction, the sort algorithm might swap two fingerprints non-atomically s.t. the table does not contain one of the two fingerprints. The one not it table is then expected to be in the lo variable of the sort algorithms.

$$\begin{aligned} Contains \triangleq\ & \land \forall\, seen \in history : \\ & \qquad contains(seen,\, table,\, external,\, L) \\ & \land \forall\, unseen \in (fps \setminus history) : \\ & \qquad \neg contains(unseen,\, table,\, external,\, L) \end{aligned}$$

The absolute value of the given number.

$$abs(number) \triangleq \text{IF } number < 0 \text{ THEN } -1 * number \text{ ELSE } number$$

True when no eviction is running.

$$FindOrPut \triangleq evict = \text{FALSE}$$

FALSE iff table contains duplicate elements (excluding empty), unless Evict is running. During eviction, the sort algorithm might swap two fingerprints non-atomically s.t. the table contains duplicates of one of the two fingerprints temporarily.

$$\begin{aligned} Duplicates \triangleq FindOrPut \implies & \text{LET } sub \triangleq SelectSeq(table, \text{LAMBDA } e : e \neq empty) \\ & \text{IN } \text{IF } Len(sub) < 2 \text{ THEN TRUE} \\ & \text{ELSE } \forall\, i \in 1 \,..\, (Len(sub) - 1) : \\ & \quad \forall\, j \in (i+1) \,..\, Len(sub) : \\ & \quad abs(sub[i]) \neq abs(sub[j]) \end{aligned}$$

seq is sorted iff its empty-filtered sub-sequence is sorted. An empty sequence is defined to be sorted.

$$\begin{aligned} isSorted(seq) \triangleq & \text{LET } sub \triangleq SelectSeq(seq, \text{LAMBDA } e : e \neq empty) \\ & \text{IN } \text{IF } Len(sub) < 2 \text{ THEN TRUE} \\ & \text{ELSE } \forall\, i \in 1 \,..\, (Len(sub) - 1) : \\ & \quad sub[i] < sub[i+1] \end{aligned}$$

External storage is always sorted in ascending order.

$$Sorted \triangleq isSorted(external) \wedge isSorted(newexternal)$$

TRUE iff f is found in table within idx(f,0)..id(f,L).

$$containedInTable(f) \triangleq \exists\, l \in 0 \,..\, L : table[idx(abs(f), l)] = f$$

TRUE if all fingerprints $\in$ history correctly transition from table to the external storage. Models a three state FSM.

$$\begin{aligned} Consistent \triangleq & \; FindOrPut \implies \forall\, seen \in history : \\ & \wedge\, containedInTable(seen) \implies \neg containsElem(external, seen) \\ & \wedge\, containedInTable(seen * (-1)) \implies containsElem(external, seen) \\ & \wedge\, \neg containedInTable(seen) \implies containsElem(external, seen) \end{aligned}$$

Under all behaviors, the algorithm makes progress and eventually puts all fingerprints in fps into the table resulting in history = fps.

$$Complete \triangleq \Diamond\Box(history = fps)$$

Iff certain that Termination is guaranteed, the liveness property Complete can be rewritten to the safety property below. A safety property can be checked faster.

$$CompleteAsSafety \triangleq \forall self \in ProcSet : pc[self] = \text{“Done”} \implies (history = fps)$$

B.3. Sort

Full listing of the dedicated PlusCal specification discussed in section 7.1.2. Contrary to the previous specification, the Sort specification studies the properties of the Insertion Sort algorithm isolated from lock-freedom. It shares several operators and parts of the PlusCal code with B.2.

MODULE *Sort*

EXTENDS *Integers*, *Sequences*, *FiniteSets*

CONSTANT *empty*, A model value that marks unoccupied positions in table
L, The maximum length of the probing sequence.
fps, The set of fingeprints (Integers) in table.
K, The length of table.
PS A function whose domain are the fps and whose image are the correspdoing probing sequences.

The element of position Len(seq) of a sequence seq.

$last(seq) \triangleq seq[Len(seq)]$

The largest element in the sequence, assuming sequence to be sorted in ascending order.

$largestElem(sortedSeq) \triangleq \text{IF } sortedSeq = \langle\rangle \text{ THEN } 0 \text{ ELSE } last(sortedSeq)$

All elements of seq1 smaller than elem and the largest element in seq2.

$$subSeqSmaller(seq1, seq2, elem) \triangleq SelectSeq(seq1, \text{LAMBDA } p : p < elem \land p > largestElem(seq2))$$

All elements of seq1 larger than the largest element in seq2.

$$subSeqLarger(seq1, seq2) \triangleq \text{IF } seq2 = \langle\rangle \text{ THEN } seq1 \text{ ELSE } SelectSeq(seq1, \text{LAMBDA } p : p > largestElem(seq2))$$

The minimum and maximum element in set S.

$min(S) \triangleq \text{CHOOSE } s \in S : \forall\, a \in S : s \leq a$
$max(S) \triangleq \text{CHOOSE } s \in S : \forall\, a \in S : s \geq a$

The given index i modulo the sequences' length.

$$mod(i,\ len) \triangleq \text{IF } i\%len = 0 \text{ THEN } len \text{ ELSE } (i\%len)$$

Re-scale.

$$rescale(k,\ maxF,\ minF,\ fp,\ p) \triangleq \text{LET } f \triangleq (k-1) \div (maxF - minF)$$
$$\text{IN } \quad mod((f * (fp - minF + 1)) + p,\ k)$$

Calculates an fp's index where fp $\in$ fps. p is an alternative address, such that: p $\in$ 0..P.

$$idx(fp,\ p) \triangleq rescale(K,\ max(fps),\ min(fps),\ fp,\ p)$$

The image of the function F.

$$Image(F) \triangleq \{F[x] : x \in \text{DOMAIN } F\}$$

The position of elem in sequence seq.

$$At(seq,\ elem) \triangleq \text{CHOOSE } i \in \text{DOMAIN } seq : seq[i] = elem$$

The probing sequence with L=l of the given fingerprint fp.

$$ProbingSequence(fp,\ l) \triangleq [i \in 1\,..\,l \mapsto idx(fp,\ i-1)]$$

The probing sequences of all fingerprints $\in$ fpts with L=l.

$$ProbingSequences(fpts,\ l) \triangleq [fp \in fpts \mapsto ProbingSequence(fp,\ l)]$$

TRUE, iff the sequence's elements are all the empty element.

$$isEmpty(seq) \triangleq Image(seq) = \{empty\}$$

All non-empty elements (fingerprints) have to be within their corresponding probing sequence, s.t. ps = PS[t[i]]. Let p be the actual position of a fingerprint, we further request, that the positions lower than p in ps are non-empty. Finally, we exclude those t, which have duplicates (ignoring empty) or which are the empty seq.

$$Filter(t) \triangleq \ \land \forall\, i \in 1\,..\,Len(t) :$$

empty is ignored/skipped anyway.

$$\lor\ t[i] = empty$$

All fingerprints within their probing sequence.

$$\lor\ \land\ i \in Image(PS[t[i]])$$

All lower positions of the fingerprint's

probing sequence are non-empty.

$$\land \forall\, pos \in Image($$
$$SubSeq(PS[t[i]],\ 1,\ At(PS[t[i]],\ i))) :$$

$$t[pos] \neq empty$$

...and there are no duplicates (except empty).

$$\land \text{ LET } sub \triangleq SelectSeq(t, \text{ LAMBDA } p : p \neq empty)$$

$$\text{IN} \quad Len(sub) = Cardinality(Image(sub))$$

...and the sequence is the non-empty sequence.

$$\land \neg isEmpty(t)$$

Generate all possible permutations of length 1..K out of the elements $fps \cup empty$. Out of this (very large) set of permutations, exclude those, that do not adhere to Filter.

$$Inits \triangleq \text{ UNION } \{\{perm \in [1 \,..\, K \to (fps \cup \{empty\})] : Filter(perm)\}\}$$

A fp wrapped around if its alternate indices are beyond K and its actual index is lower than its primary idx.

$$wrapped(fp, pos) \triangleq idx(fp, 0) > mod(pos, K)$$

Compare the two fingerprints fp1 and fp2 for order with regards to their numerical values and their respective positions i1 and i2.
Returns -1, iff fp2 is less than fp1. Returns 0, iff fp1 and fp2 are equal. Returns 1, iff fp1 is less than fp2.
compare considers three cases: 1) Iff either one or both fingerprints are empty, they are defined to be equal. Under the assumption of a stable sorting algorithm, fp1 and fp2 are not swapped (ELSE 0). 2) Iff neither one or both fingerprints wrapped, a basic comparison is done. A basic comparison is one, where the lower positioned fp has to be numerically lower. 3) Iff the truth values for wrapped differed, two cases have to be distinguished:
Let: $\overset{\circlearrowleft}{fps} \triangleq fp \in fps : \exists i \in Image(PS[fp]) : wrapped(fp, i)$
$\overrightarrow{fps} \triangleq fps \setminus \overset{\circlearrowleft}{fps}$
3a) Comparison when fp1 and fp2 are both in $\overset{\circlearrowleft}{fps}$. If fp1 is at a lower position (thus wrapped) and numerically lower, swap it with fp2 which is at a higher position and thus did not wrap. For example, fp1 was inserted into the table after fp2 and thus wrapped, but is numerically lower than fp2.
3b) Special case comparison required by Insertion Sort. It compares a fingerprint in $\overset{\circlearrowleft}{fps}$ with one in $\overrightarrow{fps}$. Insertion Sort compares adjacent elements. Thus, without this case two fingerprints fp1 and fpX, which are eventually handled by 3a), would not be sorted, iff fp2 is inbetween of fp1 and fpX. Thus, fp1 is swapped with fp2 meaning it moves towards the beginning of table. Eventually, all wrapped fingerprints in $\overset{\circlearrowleft}{fps}$ will form a cluster at the beginning of t and can then be sorted with 3a). In other words, we allow the wrapped fingerprints to be compacted at the beginning ot the table and non-wrapping fingerprints to be moved to higher positions.
Assumeming that the beginning of table is: «1,23,22,...,24» (assuming fps is 1..24, L=3 and K=6. Sorted, table needs to change to «1,23,24,...,22». Without 3b), Insertion Sort compares 22 to 1 and 23 to 22. The outcome would be «1,22,23,...,24», which is cleary not fully sorted. Thus, in order to handle this case, we allow IS to swap 22 and 23 with 1. As a result, table - when sorted - is «23,24,1,...,22».
Can we be sure, that non-wrapping fingerprints do not get moved out beyond the end of their probing sequence? Obvously, at most, L-1 wrapping fingerprints can be located at the beginning of table. In this case, only one non-wrapping fingerprint will be in the table, which maximally will be moved L-1 positions to the right with regards to its primary position.

$$
\begin{array}{l}
compare(fp1,\ i1,\ fp2,\ i2) \triangleq \\
\qquad \text{IF } fp1 \neq empty \wedge fp2 \neq empty \qquad \text{1)} \\
\qquad \text{THEN IF } wrapped(fp1,\ i1) = wrapped(fp2,\ i2) \qquad \text{2)} \\
\qquad\qquad \text{THEN IF } i1 > i2 \wedge fp1 < fp2 \text{ THEN } -1 \text{ ELSE } 1 \\
\qquad\qquad \text{ELSE IF } i1 < i2 \wedge fp1 < fp2 \text{ THEN } -1 \text{ ELSE} \qquad \text{3a} \\
\qquad\qquad\qquad \text{IF } i1 > i2 \wedge fp1 > fp2 \text{ THEN } -1 \text{ ELSE } 0 \qquad \text{3b} \\
\qquad \text{ELSE } 0
\end{array}
$$

--algorithm *Sort*

table: The actual hash table specified as a TLA+ sequence. history: An auxiliary (history) variable unrelated to the actual hash table specification. It just records the inserted fingerprints to be verified by Inv. An implementation won't need history. external: The external storage where fingerprints are eventually evicted to. outer/inner: Index variables local to the sort action. P: The number of times an alterante index is to be tried.

```
{variable table ∈ Inits,
         history = {},
         external = ⟨⟩,
         newexternal = ⟨⟩,
         ei = 1,
         ej = 1,
         lo = 0; {

   Remember the fingerprints initially in table.
history:  history := Image(table) \ {empty};

          Insertion sort.
strIns:   while (ei ≤ K + L){
            lo := table[mod(ei + 1, K)];
nestedIns:  while (compare(lo, mod(ei + 1, K),
                           table[mod(ej, K)], mod(ej, K)) ≤ −1){
              table[mod(ej + 1, K)] := table[mod(ej, K)];
              if (ej = 0){
                ej := ej − 1;
                goto set;
              }else {
                ej := ej − 1;
              };
            };
set:        table[mod(ej + 1, K)] := lo;
            ej := ei + 1;
            ei := ei + 1;
          };
          ei := 1;

          Write to external storage.
flush:    while (ei ≤ K + L){
            lo := table[mod(ei, K)];
```

```
            if (lo ≠ empty ∧
                lo > largestElem(newexternal) ∧
                ((ei ≤ K ∧ ¬wrapped(lo, ei)) ∨
                 (ei > K ∧ wrapped(lo, ei)))){
                Copy all smaller fps than lo from
                secondary to newexternal.
                newexternal := Append(newexternal ∘
                      subSeqSmaller(external, newexternal, lo), lo);
                Mark table[mod(cpy,table)] as being
                written to external.
                table[mod(ei, K)] := lo;   * (-1);
            };
            ei := ei + 1;
        };
         Append remainder of external to newexternal and
         assign newexternal to external.
        external := newexternal ∘
                      subSeqLarger(external, newexternal);
        newexternal := ⟨⟩;
    }
}
```

BEGIN TRANSLATION

VARIABLES $table$, $history$, $external$, $newexternal$, ei, ej, lo, pc

omitted

END TRANSLATION

Invariant: external is always sorted.

$$IsSorted \triangleq \forall x \in 1 \,..\, Len(external) : \forall y \in x \,..\, Len(external) : external[x] \leq external[y]$$

Post-condition: Upon termination (pc="Done"), the fingerprints in external and the history set of fingerprints are identical.

$$IsComplete \triangleq pc = \text{“Done”} \implies history = Image(external)$$

Invariant: All fingerprints (ignoring empty) are always withing their corresponding probing sequence.

$$IsConsistent \triangleq \forall\, i \in 1 \,..\, Len(table) : \\ table[i] \neq empty \implies i \in Image(PS[table[i]])$$

Erklärungen

Versicherung lt. § 59 Abs. 3 HmbHG

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel — insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen — benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ahrensburg den 13. Juni 2017

Bibliotheksveröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Ahrensburg den 13. Juni 2017