Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

A Test Suite for Model Checking Persistent Memory Programs

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΣΠΥΡΙΔΩΝ ΠΑΥΛΑΤΟΣ

Επιβλέπων : Κωνσταντίνος Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2022

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

A Test Suite for Model Checking Persistent Memory Programs

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΣΠΥΡΙΔΩΝ ΠΑΥΛΑΤΟΣ

Επιβλέπων : Κωνσταντίνος Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 25η Ιουλίου 2022.

.....	.....	.....
Κωνσταντίνος Σαγώνας Αν. Καθηγητής Ε.Μ.Π.	Νικόλαος Παπασπύρου Καθηγητής Ε.Μ.Π.	Γεώργιος Γκούμας Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2022

Σπυρίδων Παυλάτος

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Οι τελευταίες εξελίξεις στις τεχνολογίες μνήμης έχουν φέρει στο επίκεντρο την Επίμονη Μνήμη (ΕΜ), η οποία προσφέρει επιδόσεις συγκρίσιμες με τις DRAM και πρόσβαση σε επίπεδο byte, ενώ παράλληλα είναι μη πτητική, δηλαδή τα δεδομένα της παραμένουν και μετά τη διακοπή παροχής ηλεκτρικού ρεύματος. Για να εξασφαλίσει κανείς την ορθότητα προγραμμάτων σε ΕΜ, οι σύγχρονες αρχιτεκτονικές, όπως η Intel x86, προσφέρουν ειδικές εντολές που γράφουν (flush) τα δεδομένα των πτητικών cache στην κύρια μνήμη. Αυτές οι εντολές δημιουργούν συγκεκριμένες σειρές εγγραφής στην ΕΜ. Ο φορμαλισμός αυτών των σειρών δίνεται από τα μοντέλα επίμονης μνήμης και καθορίζει τις επιτρεπτές καταστάσεις του συστήματος μετά από κάποια αποτυχία του.

Αυτή η εργασία αναπτύσσει μία σουίτα ελέγχου για προγράμματα ΕΜ. Η σουίτα αυτή περιλαμβάνει litmus tests και tests σε δομές δεδομένων. Τα litmus tests στοχεύουν στον έλεγχο της ορθότητας του εργαλείου που χρησιμοποιείται για την επαλήθευση ορθότητας του προγράμματος. Στην εργασία αυτή χρησιμοποιούμε το εργαλείο PerSeVerE, το οποίο υλοποιεί τεχνική ελέγχου μοντέλου για ΕΜ και βρίσκεται σε διαρκή ανάπτυξη. Μέσω των litmus tests καταφέραμε να βρούμε μία εσωτερική αστοχία του PerSeVerE, εξαιτίας της οποίας δεν μπορούσε να μοντελοποιηθεί σωστά το Px86 μοντέλο επίμονης μνήμης. Το γεγονός αυτό αναφέρθηκε στους δημιουργούς του PerSeVerE, οι οποίοι πρόσθεσαν επιπλέον υποστηρίξη στον πυρήνα του εργαλείου. Τα tests σε δομές δεδομένων στοχεύουν στον έλεγχο ορισμένων υλοποιήσεων δομών δεδομένων χωρίς κλειδώματα, καθώς και σε συγκεκριμενούς μετασχηματισμούς αυτών από την πρόσφατη βιβλιογραφία που μετατρέπουν αυτές τις δομές σε ανθεκτικώς γραμμικοποιήσιμες (durably linearizable). Μέσω της σουίτας μας μπορέσαμε να ελέγξουμε ότι οι βασικές εκδοχές των δομών αυτών δεν ήταν ανθεκτικές σε συγκρούσεις, ενώ οι μετασχηματισμοί τους περνάνε επιτυχώς τους ελέγχους μας. Επίσης, πειραματιστήκαμε με εξάλειψη ορισμένων flush εντολών των μετασχηματισμών, το οποίο οδήγησε σε παραβάσεις ορθότητας. Αυτό σημαίνει ότι οι εντολές αυτές είναι απαραίτητες για να εξασφαλίσουν την ορθότητα των δομών ως προς τις αποτυχίες συστήματος. Θεωρούμε ότι η σουίτα ελέγχου μας μπορεί να χρησιμοποιηθεί ως οδηγός για το πως να χρησιμοποιηθούν εργαλεία ελέγχου μοντέλου σε ΕΜ για την επαλήθευση ορθότητας ανθεκτικών δομών δεδομένων και βιβλιοθηκών, καθώς και ως σημεία αναφοράς για την επίδοση αυτών των εργαλείων.

Λέξεις κλειδιά

Επίμονη μνήμη, Μοντέλα μνήμης, Συνέπεια μνήμης, Τεχνικές ελέγχου μοντέλου, Επαλήθευση ορθότητας λογισμικού, Δομές δεδομένων χωρίς κλειδώματα

Abstract

The latest advances in memory technologies have brought Persistent Memory to the spotlight. Persistent Memory (PM) provides DRAM-like performance and byte-addressability, while preserving its content in case of a crash (non-volatility). To ensure correctness of programs targeting PM, memory architectures, like Intel's x86, have introduced new instructions that flush the contents of the volatile caches to the persistent domain. These instructions induce certain persist orderings, which are formalized by persistency memory models and define the allowed behaviours of the system after a crash.

This thesis develops a test suite for PM programs, consisting of both litmus and data structure tests. Litmus tests aim to check the sanity of the tool used for verification. In our case, we used PerSeVerE, which is a persistency model checking tool under current development. Through our tests, we were able to pinpoint an internal inability of PerSeVerE to support the Px86 memory model, which led its developers to provide additional support in the tool's core. The data structure tests try to test various implementations of lock-free data structures, as well as some adaptations and transformations found in recent literature that turn these implementations into durable linearizable ones. With our test suite, we were able to check that the original implementations of these data structures were not durably linearizable. On the other hand, the durable versions of the data structures pass our checks. Furthermore, we experimented with eliminating explicit flush instructions in these versions, which led to durably linearizability violations and therefore proving the necessity of these instructions to ensure durable linearizability. Our test suite can be used as a guideline for how to use model checking to verify durable data structures and persistent libraries, and can serve as a benchmark for persistency model checking tools.

Key words

Persistent Memory, Memory models, Memory consistency, Model checking, Software verification, Lock-free data structures

Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή αυτής της διπλωματικής, κ. Κωστή Σαγώνα, για τη συνεχή καθοδήγηση κατά τη διάρκεια της εκπόνησής της. Οι συμβουλές και οι κατευθύνσεις, που μου έδωσε, αποδείχθηκαν καίριες για την πορεία της έρευνας. Καθοριστική υπήρξε επίσης η συνεισφορά του Μιχάλη Κοκολογιαννάκη, υποψήφιου διδάκτορα στο Max Planck Institute for Software Systems, όπως και του επιβλέποντα καθηγητή του κ. Βίκτωρα Βαφειάδη. Φυσικά ευχαριστώ και τα υπόλοιπα μέλη της εξεταστικής επιτροπής, κ. Νίκο Παπασπύρου και κ. Γιώργο Γκούμα, όχι μόνο γιατί αποτέλεσαν μέλη αυτής, αλλά και γιατί μου κίνησαν το ενδιαφέρον να ασχοληθώ με τις Γλώσσες Προγραμματισμού και τα Υπολογιστικά Συστήματα μέσω της διδασκαλίας των μαθημάτων τους.

Ευχαριστώ, επίσης, τους κοντινούς μου φίλους που συνείσφεραν, ώστε να περάσουν τα χρόνια των σπουδών μου στο Ε.Μ.Π. ευχάριστα και επικοδομητικά. Ιδιαίτερη μνεία θα ήθελα να δώσω στους συμφοιτητές μου Αλέξανδρο Γ., Κωνσταντίνο Μ., Νικολέτα Η., Ιάσονα Ν., Χαρίτωνα Χ., Μαριλένα Ν.Π., Νίκο Υ., αλλά και τον παιδικό μου φίλο Νίκο Π., ο οποίος ήταν πάντα δίπλα μου και με στήριζε σε όσες δυσκολίες αντιμετώπισα.

Τέλος, ο,τίδηποτε έχω καταφέρει μέχρι στιγμής δε θα ήταν δυνατό χωρίς τη διαρκή υποστήριξη που μου παρείχε η μητέρα μου. Ήταν πάντα εκεί, όταν την χρειαζόμουν, και την ευχαριστώ μέσα από την καρδιά μου για αυτό.

Σπυρίδων Παυλάτος,

Αθήνα, 25η Ιουλίου 2022

Περιεχόμενα

Κατάλογος πινάκων

Πίνακες στο αγγλικό κείμενο

Κατάλογος σχημάτων

Σχήματα στο αγγλικό κείμενο

Εκτεταμένη Ελληνική Περίληψη

Η κύρια γλώσσα της παρούσας διπλωματικής εργασίας είναι η αγγλική, κυρίως για λόγους προσβασιμότητας και απόδοσης των τεχνικών όρων. Στην εκτεταμένη ελληνική περίληψη θα συνοψίσουμε το περιεχόμενό της, δίνοντας έμφαση στους βασικούς ορισμούς, τις μεθοδολογίες που ακολουθήθηκαν και τα αποτελέσματα που πήραμε. Η δομή της ενότητας αυτής είναι σε ένα προς ένα αντιστοίχηση με το αγγλικό κείμενο.

Εισαγωγή

Ένα από τα πιο ελπιδοφόρα τεχνολογικά επιτεύγματα των τελευταίων ετών στον τομέα της αρχιτεκτονικής υπολογιστών είναι η εισαγωγή της Επίμονης Μνήμης (ΕΜ). Η ΕΜ στοχεύει στην αντικατάσταση των παραδοσιακών DRAM, ενώ είναι μη πτητική, δηλαδή διατηρεί το περιεχόμενό της μετά από διακοπή ρεύματος, όπως κάνουν οι δίσκοι.

Ωστόσο, το να γράψει κανείς ορθά προγράμματα για ΕΜ εμφανίζει αρκετές δυσκολίες. Για παράδειγμα, μπορούν να συμβούν αποτυχίες συστήματος (system crashes) οποιαδήποτε στιγμή κατά την εκτέλεση του προγράμματος, γεγονός που μπορεί να οδηγήσει σε ασυνεπείς καταστάσεις της μνήμης κατά τη διάρκεια της διαδικασίας ανάκτησης (recovery) της κατάστασης του προγράμματος. Συνεπώς, οι προγραμματιστές πρέπει να κατανοήσουν πολύ προσεκτικά τις αναμενόμενες συμπεριφορές των προγράμματών τους ως προς τα πιθανά σφάλματα, αλλά και να χρησιμοποιήσουν ειδικές εντολές (π.χ. `flush`) οι οποίες γράφουν τα δεδομένα των caches στην ΕΜ. Οι συμπεριφορές της ΕΜ μπορούν να μοντελοποιηθούν με τη χρήση των *μοντέλων επιμονής (persistency models)*, τα οποία καθορίζουν τη σειρά με την οποία γράφονται τα δεδομένα στην επίμονη μνήμη.

Σε αυτή την εργασία, θα συζητήσουμε πώς μπορούν να χρησιμοποιηθούν τα μοντέλα επιμονής σε συνδυασμό με μεθόδους *ελέγχου μοντέλου (model checking)*, οι οποίες είναι πολύ ισχυρές τεχνικές επαλήθευσης λογισμικού, προκειμένου να ελέγξουμε την ορθότητα διαφόρων προγραμμάτων ΕΜ. Ο έλεγχος μοντέλου για ΕΜ εισάγει νέες προκλήσεις, όπως το γεγονός ότι ένα σφάλμα μπορεί να συμβεί οποιαδήποτε στιγμή κατά τη διάρκεια του προγράμματος. Αυτό οδηγεί σε πολύ μεγαλύτερο χώρο καταστάσεων από ό,τι τα ταυτόχρονα προγράμματα χωρίς crashes. Θα χρησιμοποιηθεί το εργαλείο PerSeVerE για να ελέγξουμε προγράμματα ΕΜ κάτω από το Px86 μοντέλο επιμονής.

Η κύρια συνεισφορά της εργασίας είναι η δημιουργία μίας σουίτας ελέγχου προγραμμάτων επίμονης μνήμης. Στη σουίτα αυτήν περιλαμβάνονται μικρά συνθετικά προγράμματα (litmus tests) με σκοπό τον έλεγχο της ορθότητας του εργαλείου, αλλά και προγράμματα για δομές δεδομένων χωρίς κλειδώματα, για τα οποία χρησιμοποιήσαμε τόσο τις απλές ταυτόχρονες υλοποιήσεις τους, όσο και καινούριες παραλλαγές τους τους, οι οποίες είναι *ανθεκτικώς γραμμικοποιήσιμες (durably linearizable)*. Η ανθετική γραμμικοποιησιμότητα είναι το βασικό κριτήριο ορθότητας για δομές δεδομένων στην ΕΜ και μέσω της σουίτας μας επιβεβαιώσαμε ότι οι απλές εκδοχές των δομών αυτών δεν είναι ανθεκτικώς γραμμικοποιήσιμες, αλλά και ότι η αφαίρεση `flush` εντολών από τις παραλλαγές των δομών οδηγεί σε παραβάσεις ορθότητας.

Εκτός από τα πειραματικά αποτελέσματα που παρουσιάζονται σε αυτήν τη διπλωματική εργασία με χρήση της τρέχουσας σουίτας, θεωρούμε ότι η σουίτα μας μπορεί να αποτελέσει οδηγό για την χρήση εργαλείων ελέγχου μοντέλου στην ΕΜ με σκοπό την επαλήθευση ανθεκτικών δομών και βιβλιοθηκών, αλλά και ως σημείο αναφοράς για τη σύγκριση τέτοιου είδους εργαλείων.

Συνέπεια Μνήμης

Τα σύγχρονα υπολογιστικά συστήματα χρησιμοποιούν πολλαπλούς πυρήνες με σκοπό την αύξηση της επίδοσής τους. Το πιο συνηθισμένο μοντέλο μνήμης σε αυτά τα συστήματα είναι η κοινή μνήμη, δηλαδή τα διάφορα νήματα δρουν πάνω στις ίδιες θέσεις μνήμης [Adve96]. Το ερώτημα που εύλογα γεννάται είναι: *ποια είναι η σειρά με την οποία τα διάφορα νήματα βλέπουν τα αποτελέσματα της εκτέλεσης των υπολοίπων νημάτων;* Η απάντηση δίνεται από τα *μοντέλα συνέπειας (consistency models)* με βάση τα οποία μπορεί κανείς να προσδιορίσει τα αποτελέσματα ενός πολυνηματικού προγράμματος.

Ένα από τα βασικότερα μοντέλα συνέπειας και ίσως αυτό που βρίσκεται πιο κοντά στην αντίληψη των προγραμματιστών για πολυνηματικά προγράμματα είναι η Ακολουθιακή Συνέπεια (Sequential Consistency) ο ορισμός της οποίας έχει δοθεί από τον Leslie Lamport [Lamp79] ως εξής.

Ορισμός. (Ακολουθιακή Συνέπεια) *Ένα πολυπύρηνο σύστημα είναι ακολουθιακά συνεπές εάν το αποτέλεσμα κάθε εκτέλεσης ενός προγράμματος είναι το ίδιο με το να έχουν εκτελεστεί όλες οι λειτουργίες ακολουθιακά (δηλαδή να περιμένει η μία εντολή να ολοκληρωθεί η προηγούμενη) και να διατηρείται η σειρά του προγράμματος για κάθε επεξεργαστή.*

Ουσιαστικά, η Ακολουθιακή Συνέπεια εξασφαλίζει ότι υπάρχει μία ολική διάταξη των load και των store εντολών, την οποία "βλέπουν" όλα τα νήματα ταυτόχρονα με τον ίδιο τρόπο. Το πρόβλημα με την Ακολουθιακή Συνέπεια είναι ότι πολύ περιοριστική, καθώς κάθε στιγμή μπορεί να εκτελείται μόνο μία έντολη. Χάνεται, δηλαδή, ο παραλληλισμός από τη χρήση πολλών πυρήνων.

Για να αντιμετωπιστεί αυτό το φαινόμενο, έχουν προταθεί πιο χαλαρά μοντέλα συνέπειας, όπως η Ολική Σειρά Αποθήκευσης (TSO - Total Store Ordering). Αυτό το μοντέλο συνέπειας εισάγει ειδικούς store buffers σε κάθε πυρήνα, μέσω των οποίων "κρύβεται" η καθυστέρηση των stores. Τα loads μπορούν να εκτελούνται πριν ολοκληρωθούν προήγουμενα κατά σειρά προγράμματος writes, χρησιμοποιώντας τον buffer του πυρήνα τους για να διαβάζουν τις τιμές των stores. Αυτό, όμως, έχει ως αποτέλεσμα να είναι επιτρεπτές περισσότερες συμπεριφορές σε σχέση με το SC μοντέλο. Για την αποφυγή τέτοιων συμπεριφορών είναι απαραίτητη η χρήση `fence` εντολών. Το TSO μοντέλο χρησιμοποιείται ευρέως από σύγχρονες αρχιτεκτονικές, όπως η x86 της Intel.

Οι παραπάνω περιγραφές των μοντέλων συνέπειας έγιναν με χρήση λειτουργικής σημασιολογίας, δηλαδή κάναμε μία αφαίρεση του συστήματος χρησιμοποιώντας ιδανικά δομικά στοιχεία, όπως τους store buffers, για να περιγράψαμε πως το σύστημα μεταβαίνει από μία κατάσταση στην επόμενη.

Υπάρχει, όμως, και μία διαφορετική προσέγγιση για να εκφραστούν τα μοντέλα συνέπειας με χρήση αξιωματικής σημασιολογίας [Algl12]. Συγκεκριμένα, ορίζονται σχέσεις μεταξύ των διαφόρων λειτουργιών της μνήμης και τίθενται συγκεκριμένοι περιορισμοί σε αυτές τις σχέσεις. Τα αξιωματικά μοντέλα χρησιμοποιούν πρακτικά έννοιες σχεσιακής άλγεβρας και οι περιορισμοί που τίθενται δημιουργούν ένα σύνολο αξιωμάτων, το οποίο ορίζει τις συμπεριφορές που είναι συνεπείς με το εκάστοτε μοντέλο.

Έλεγχος Μοντέλου

Η διαδικασία της επαλήθευσης ορθότητας ενός προγράμματος αναφέρεται στη μαθηματική απόδειξη ότι το πρόγραμμα είναι σύμφωνο με κάποιο κριτήριο ορθότητας. Μία ευρέως χρησιμοποιούμενη τακτική σε αυτό το πεδίο είναι ο έλεγχος μοντέλου (model checking), δηλαδή η συστηματική εξευρεύνηση του χώρου καταστάσεων του προγράμματος και η επαλήθευση του κριτηρίου ορθότητας σε κάθε κατάσταση.

Οι πρώτες προσεγγίσεις για τον έλεγχο μοντέλου αποθήκευαν τις καταστάσεις που έχουν επισκεφθεί, για να μην τις ξαναεπισκεφθούν στο μέλλον. Αυτό, όμως, είναι προβληματικό, καθώς

οι καταστάσεις αυτές μπορεί να χρειάζονται πολύ μεγάλο αριθμό στοιχείων του συστήματος για να προσδιοριστούν, όπως τα περιεχόμενα της μνήμης, των καταχωρητών κ.λπ. Τη λύση σε αυτό το πρόβλημα δίνουν οι τεχνικές του ελέγχου μοντέλου χωρίς κατάσταση (SMC - Stateless Model Checking), οι οποίες δεν χρειάζεται να αποθηκεύουν αυτές τις καταστάσεις.

Όμως, ακόμα και οι τεχνικές ελέγχου μοντέλου χωρίς κατάσταση παραμένουν ευάλωτες στο μεγαλύτερο πρόβλημα των πολυνηματικών προγραμμάτων, το οποίο είναι η εκθετική (ή και χειρότερη) αύξηση του αριθμού των καταστάσεων ως προς τον αριθμό των νημάτων και το μήκος του προγράμματος. Για να μειωθεί ο αριθμός των εκτελέσεων των εργαλείων ελέγχου μοντέλων χωρίς κατάσταση χρειάζεται κανείς να παρατηρήσει ότι πολλές από αυτές τις εκτελέσεις είναι ισοδύναμες, δηλαδή οδηγούν στις ίδιες καταστάσεις. Για παράδειγμα, αν αλλάξει η σειρά από δύο διαδοχικά loads, το καθένα σε διαφορετική μεταβλητή, το αποτέλεσμα ενός προγράμματος θα παραμείνει το ίδιο. Τέτοιου είδους αναμίξεις εντολών ονομάζονται *ισοδύναμες* και το SMC χρειάζεται να εξερευνήσει μόνο μία από αυτές.

Οι τεχνικές που εντοπίζουν τέτοιες κλάσεις ισοδυναμίας εκτελέσεων ενός προγράμματος με σκοπό τη μείωση των συνολικών καταστάσεων καλούνται *Partial Order Reduction (POR)* (Ελάττωση βασισμένη σε Μερικές Διατάξεις). Οι πρώιμες τεχνικές ελάττωσης βασισμένες σε μερικές διατάξεις υπολόγιζαν στατικά τις κλάσεις ισοδυναμίας, ενώ οι πιο σύγχρονες τέτοιες τεχνικές αναφέρονται ως τεχνικές *Δυναμικής Ελάττωσης βασισμένη σε Μερικές Διατάξεις* (DPOR - Dynamic Partial Order Reduction) [Flan05, Abdu14] διότι υπολογίζουν τις κλάσεις ισοδυναμίας δυναμικά κατά την εκτέλεση του προγράμματος με χρήση ενός δρομολογητή χρόνου εκτέλεσης.

Ένα σύγχρονο και πολύ αποδοτικό SMC εργαλείο, που χρησιμοποιεί μία μορφή DPOR, είναι το GenMC [Koko21b]. Το συγκεκριμένο εργαλείο κατασκευάζει σταδιακά ένα γράφο εκτέλεσης του προγράμματος και ελέγχει σε κάθε βήμα αν ο γράφος είναι συνεπής σύμφωνα με τον αξιωματικό ορισμό του εκάστοτε μοντέλου συνέπειας. Η εισαγωγή των διάφορων μοντέλων στο GenMC γίνεται με τη χρήση ενός υπό ανάπτυξη εργαλείου, το οποίο ονομάζεται kater. Το kater δέχεται σαν είσοδο ένα αρχείο που περιγράφει με όρους σχεσιακής άλγεβρας ένα μοντέλο συνέπειας και δίνει σαν έξοδο κώδικα, ο οποίος μπορεί πολύ εύκολα να ενσωματωθεί στον πυρήνα του GenMC.

Επίμονη Μνήμη

Οι εξελίξεις στις τεχνολογίες μνήμης τα τελευταία χρόνια και πιο συγκεκριμένα η ανάπτυξη της Επίμονης Μνήμης (EM) [Lee09, Kawa12] έχουν πρακτικά γεφυρώσει το χάσμα μεταξύ των ψηλότερων βαθμίδων της ιεραρχίας μνήμης (RAM, caches, καταχωρητές) και των χαμηλότερων βαθμίδων (αποθηκευτικός χώρος, π.χ. SSD, HDD). Η EM προσφέρει πρόσβαση σε επίπεδο byte, ενώ παράλληλα είναι μη πτητική, δηλαδή τα δεδομένα της παραμένουν και μετά τη διακοπή παροχής ηλεκτρικού ρεύματος. Οι επιδόσεις της, μάλιστα, είναι συγκρίσιμες με αυτές της DRAM. Αυτή τη στιγμή, η EM είναι εμπορικά διαθέσιμη από την Intel με την τεχνολογία Optane [Inte19b] και μπορεί να τοποθετηθεί μαζί με τη DRAM στο διάδρομο μνήμης ή να την αντικαταστήσει τελείως.

Η επίμονη μνήμη, όμως, δημιουργεί ορισμένες δυσκολίες για τους προγραμματιστές. Οι caches αναμένεται να παραμείνουν πτητικές, ενώ τα δεδομένα τους δεν εγγράφονται στη μνήμη αμέσως, αλλά σε κάποια μελλοντική (πιθανόν μη ντετερμινιστική) στιγμή. Ως αποτέλεσμα, τα προγράμματα EM μπορούν να οδηγήσουν τη μνήμη μετά από ένα crash σε καταστάσεις που δεν είναι επιτρεπτές από το μοντέλο συνέπειας του συστήματος, όπως δείχνουμε με το Παράδειγμα 0.1

Παράδειγμα 0.1: Εγγραφή δεδομένων στην EM εκτός σειράς

Ας θεωρήσουμε το ακόλουθο ακολουθιακό πρόγραμμα, όπου οι μεταβλητές x και y είναι αρχικοποιημένες με 0.

$$\begin{array}{l} \mathtt{x} := 1; \\ \mathtt{y} := 1; \end{array} \qquad \text{(WW)}$$

Σχήμα 0.1: Απλό ακολουθιακό πρόγραμμα.

Παρόλο που η εγγραφή στο x προηγείται της εγγραφής στο y, είναι πιθανό η γραμμή της cache του y να εγγραφεί στην EM πριν από αυτή του x. Εάν συμβεί ένα crash μετά την εγγραφή του y και πριν την εγγραφή του x, τότε είναι πιθανό κατά την ανάκτηση του προγράμματος, το y να περιέχει την τιμή 1, ενώ το x να εξακολουθεί να έχει την τιμή 0. Είναι σημαντικό να σημειωθεί ότι η κατάσταση της μνήμης $\mathtt{x} = 0 \wedge \mathtt{y} = 1$ δεν μπορεί να παρατηρηθεί κάτω από το μοντέλο Ακολουθιακής Συνέπειας ή το μοντέλο Ολικής Σειράς Αποθήκευσης (TSO).

Όπως και με τα μοντέλα συνέπειας, έτσι και στην περίπτωση της EM, μπορούμε να ορίσουμε τα λεγόμενα μοντέλα επιμονής (persistency models) [Pell14]. Μέσω αυτών τον μοντέλων, ο προγραμματιστής μπορεί να κατανοήσει τη σειρά των εγγραφών που μπορεί να εμφανιστεί στο πρόγραμμά του, και έτσι να βγάλει συμπεράσματα για τη συμπεριφορά του προγράμματός του, αυτήν τη φορά, όμως, ως προς τα crashes.

Τα μοντέλα επιμονής μπορούν να κατηγοριοποιηθούν ως προς τη σχέση τους με τα μοντέλα συνέπειας του συστήματος, δηλαδή αν η σειρά που γίνονται οι εγγραφές στην EM είναι η ίδια με τη σειρά που ορίζεται από το μοντέλο συνέπειας. Αν οι δύο σειρές ταυτίζονται, τότε μιλάμε για *αυστηρό* μοντέλο επιμονής, ενώ αν είναι διαφορετικές έχουμε *χαλαρό* μοντέλο επιμονής. Τα χαλαρά μοντέλα χρησιμοποιούνται για να προσφέρουν καλύτερες επιδοσείς από τα αυστηρά. Μία άλλη κατηγοροποίηση των μοντέλων επιμονής είναι αν οι εγγραφές στην EM γίνονται σύγχρονα, δηλαδή την στιγμή που εκτελείται η αντίστοιχη εντολή, ή ασύγχρονα, δηλαδή αποθηκεύονται σε κάποιον επίμονο buffer και εκτελούνται αργότερα.

Το πιο διαδεδομένο μοντέλο επιμονής είναι το Px86 [Raad20], το οποίο μοντελοποιεί τη σημασιολογία της αρχιτεκτονικής x86 της Intel, η οποία υλοποιεί το TSO μοντέλο συνέπειας. Το Px86 είναι χαλαρό και ασύγχρονο, καθώς χρησιμοποιεί έναν καθολικό επίμονο buffer για να αποθηκεύει τις εγγραφές, όταν αυτές βγαίνουν από τον αντίστοιχο (store) buffer του εκάστοτε πυρήνα.

Η x86 αρχιτεκτονική προσφέρει τις εντολές `clflush` και `clflushopt`, οι οποίες κάνουν flush μία cache line, καθώς και την `sfence`, η οποία χρησιμοποιείται για να συγχρονίσει τις εγγραφές στην EM από διαφορετικά threads. Η χρήση της `clflush` μπορεί να αποτρέψει την μη επιθυμητή συμπεριφορά του προγράμματος WW, αν τοποθετηθεί μεταξύ των δύο εγγραφών. Όμως, λόγω της ασύγχρονης φύσης του Px86, η εντολή αυτή περιορίζει μόνο τη σειρά που μπορεί να γίνουν οι εγγραφές στην EM και όχι ότι η εγγραφή του x στην EM θα έχει προηγηθεί της εκτέλεσης της εντολής του y.

Η ασύγχρονη φύση του μοντέλου Px86 είναι αντίθετη με την αντίληψη που έχουν οι προγραμματιστές για τη χρήση των εντολών `clflush` και `clflushopt`, καθώς και με την πρόσφατη βιβλιογραφία για την EM [Pell14, Izra16, Frie18, Frie21a, Scar20]. Το πρόβλημα αυτό έρχεται να λύσει μία πρόσφατη αναθεώρηση του Px86 μοντέλου, η οποία χρησιμοποιεί σύγχρονες εντολές και αποδεικνύεται ότι είναι ισοδύναμη με το Px86. Το μοντέλο αυτό, το οποίο ονομάζεται PTSO_{syn} [Khyz21], χρησιμοποιεί έναν επίμονο buffer για κάθε γραμμή της cache, το οποίο οδηγεί στην έκφραση της σειράς εγγραφής στην EM ως μερική διάταξη. Οι μερικές διατάξεις

μπορούν να χρησιμοποιηθούν από εργαλεία SMC, που χρησιμοποιούν κάποια DPOR μέθοδο για την εξερεύνηση του χώρου καταστάσεων του προγράμματος. Στην εργασία αυτή θα χρησιμοποιήσουμε το εργαλείο PerSeVerE [Koko21a], το οποίο έχει πρόσφατα επεκταθεί ώστε να μοντελοποιεί την αναθεωρημένη εκδοχή του μοντέλου επιμονής Px86.

Το PerSeVerE χρησιμοποιεί τον πυρήνα του GenMC και προσθέτει υποστήριξη για τον έλεγχο της επιμονής. Πιο συγκεκριμένα, προσθέτει στο πρόγραμμα ένα ακόμα νήμα, το οποίο καλείται παρατηρητής ανάκτησης (recovery observer). Το νήμα αυτό εκτελείται σε κάθε βήμα της κατασκευής του γράφου εκτέλεσης, μοντελοποιώντας τη διαδικασία ανάκτησης ενός προγράμματος μετά από κάποιο crash, και ελέγχει αν τηρούνται τα αξιώματα επιμονής που ορίζονται από το μοντέλο επιμονής.

Για τη χρήση του εργαλείου, ο προγραμματιστής πρέπει να ορίσει κάποιες επίμονες μεταβλητές, οι οποίες είναι οι μόνες που επιτρέπεται να διαβαστούν κατά τη διαδικασία ανάκτησης του προγράμματος. Ο ορισμός αυτών των μεταβλητών γίνεται με χρήση της μακροεντολής `__VERIFIER_persistent_storage()`. Το εργαλείο, επίσης, προσφέρει τις ακόλουθες εντολές και λειτουργίες EM:

1. `__VERIFIER_pbarrier()`, οποία χρησιμοποιείται ως "φράχτης", ο οποίος ορίζει ότι όλες οι προηγούμενες εντολές του προγράμματος θα θεωρούνται ότι έχουν εγγραφεί στην EM κατά τη διαδικασία της ανάκτησης του προγράμματος.

2. `__VERIFIER_clflush()`, η οποία δέχεται σαν παράμετρο τη διεύθυνση μίας μεταβλητής και υλοποιεί τη σημασιολογία της `clflush` εντολής με τη σημασιολογία του αναθεωρημένου μοντέλου Px86.

3. `__VERIFIER_recovery routine()`, η οποία είναι η συνάρτηση που περιέχει τον κώδικα που θα τρέξει κατά την ανάκτηση του προγράμματος

Για την περιγραφή του μοντέλου επιμονής χρησιμοποιείται το kater.

Δομές Δεδομένων

Σε ένα περιβάλλον ταυτόχρονης εκτέλεσης προγραμμάτων, διάφορα νήματα μπορούν να εκτελέσουν λειτουργίες πάνω σε μια δομή δεδομένων που μοιράζεται μεταξύ τους. Κάθε λειτουργία συνδέεται με την *αίτηση* (request) και την *απόκρισή* (response) της. Μία *ιστορία* μπορεί να οριστεί ως μια πεπερασμένη ακολουθία των αιτήσεων και των αποκρίσεων των λειτουργιών που εκτελούνται από διάφορα νήματα στο ταυτόχρονο περιβάλλον. Μία ιστορία ονομάζεται *σειριακή* εάν κάθε αίτηση μίας λειτουργίας ακολουθείται αμέσως στη σειρά της ιστορίας από την απόκρισή της.

Με βάση την έννοια της ιστορίας, μπορεί να οριστεί το βασικό κριτήριο ορθότητας μίας δομής δεδομένων σε ταυτόχρονο περιβάλλον, το οποίο είναι η *γραμμικοποιησιμότητα (linearizability)* [Herl90]. Πιο συγκεκριμένα, μία ιστορία είναι γραμμικοποιήσιμη ανν είναι ισοδύναμη με κάποια σειριακή ιστορία. Πρακτικά, η γραμμικοποιησιμότητα σημαίνει ότι οι διάφορες λειτουργίες φαίνεται να λαμβάνουν χώρα ατομικά μέσα στο διάστημα που ορίζεται από την αίτηση και την απόκρισή τους.

Οι γραμμικοποιήσιμες δομές δεδομένων ικανοποιούν επίσης την non-blocking ιδιότητα, δηλαδή μία αίτηση κάποιας λειτουργίας δε χρειάζεται να περιμένει την ολοκλήρωση κάποιας άλλης εκρεμμής λειτουργίας. Αυτό έρχεται σε αντίθεση με τη συνήθη προσέγγιση της χρήσης κλειδωμάτων για την εξασφάλιση συγχρονισμού στις δομές δεδομένων, καθώς τα κλειδώματα μπορούν να οδηγήσουν σε αδιέξοδα (deadlocks και livelocks).

Βάση της παραπάνω ιδιότητας μπορούν να οριστούν δύο συνθήκες προόδου για ταυτόχρονες δομές δεδομένων:

ελευθερία από κλειδώματα (lock-freedom) Μία μέθοδος λέγεται ότι είναι ελεύθερη από κλειδώματα εάν υπάρχει κάποια εγγύηση οτί μεταξύ των εκτελούντων νημάτων, τουλάχιστον ένα θα τερματίσει σε πεπερασμένο αριθμό βημάτων.

ελευθερία από αναμονή (wait-freedom) Μία μέθοδος λέγεται ότι είναι ελεύθερη από αναμονή εάν εγγυάται ότι όλα τα νημάτα θα τερματίσουν σε πεπερασμένο αριθμό βημάτων. Μία μέθοδος χωρίς αναμονή είναι πάντα και χωρίς κλειδώματα, αλλά δεν ισχύει το αντίθετο.

Στην εργασία αυτή θα ασχοληθούμε με κάποιες βασικές δομές ελεύθερες από κλειδώματα, τις οποίες θα χρησιμοποιήσουμε στην πειραματική μας διαδικασία. Οι δομές δεδομένων χωρίς κλειδώματα χρησιμοποιούν εντολές, όπως την CAS (compare-and-swap), η οποία συγκρίνει με ατομικό τρόπο το περιεχόμενο μίας θέσης μνήμης με μία αναμενόμενη τιμή και αν αυτή είναι όντως η αναμενόμενη, ανανεώνει κατάλληλα την τιμή της. Χρησιμοποιώντας, λοιπόν, την CAS εντολή, οι δομές χωρίς κλειδώματα ελέγχουν συγκεκριμένους δείκτες και κόμβους της εκάστοτες δομής και σε περίπτωση που η εντολή αποτύχει ξαναρχίζουν τη μέθοδό τους από την αρχή.

Ένα βασικό παράδειγμα δομής χωρίς κλειδώματα είναι η συνδεδεμένη λίστα του Harris (Harris' linked-list) [Harr01]. Η λίστα αυτή χρησιμοποιεί τη λεγόμενη μέθοδο διαγραφής σε δύο βήματα, όπου το πρώτο βήμα μαρκάρει τον προς διαγραφή κόμβο, αφαιρώντας λογικά από τη λίστα, και το δεύτερο τον αφαιρεί και φυσικά σε κάποια μεταγενέστερη χρονική στιγμή. Η συνδεδεμένη λίστα του Harris προσφέρει μεθόδους εισαγωγής και διαγραφής με ελευθερία από κλειδώματα και αναζήτηση στοιχείου με ελευθερία από αναμονή.

Επέκταση της συνδεδεμένης λίστας αποτελεί η δομή skiplist. Η δομή αυτή αποτελείται από απλά συνδεδεμένες λίστες πολλών επιπέδων, όπου στο χαμηλότερο επίπεδο βρίσκονται όλα τα στοιχεία της λίστας. Στα ψηλότερα επίπεδα βλέπουμε όλο και μικρότερο αριθμό κόμβων, τα οποία στοχεύουν στη γρηγορότερη αναζήτηση στοιχείων της λίστας, η οποία έχει χρονική πολυπλοκότητα $\mathcal{O}(\log N)$. Η εκδοχή χωρίς κλειδώματα της skiplist βασίζεται στο μηχανισμό μαρκαρίσματος της συνδεδεμένης λίστας του Harris και οι λειτουργίες της ικανοποιούν τις ίδιες συνθήκες προόδου με αυτή [Mich02b, Herl08].

Τέλος, μία ακόμη ευρέως διαδεδομένη δομή χωρίς κλειδώματα είναι η ουρά Michael-Scott (MS-Queue) [Mich96]. Η διαδικασία εισαγωγής (enqueue) και εξαγωγής (dequeue) στοιχείων από τη λίστα γίνεται από δύο σημεία πρόσβασης στη λίστα, το δείκτη *head* και το δείκτη *tail* αντίστοιχα. Η MS-Queue κάνει χρήση της CAS εντολής για να αλλάξει τους δείκτες αυτούς, όταν καλούνται οι αντίστοιχες μέθοδοι. Ιδιαίτερη προσοχή δίνεται στην περίπτωση που η λίστα είναι κενή. Οι δύο μέθοδοι enqueue και enqueue έχουν ελευθερία από κλειδώματα.

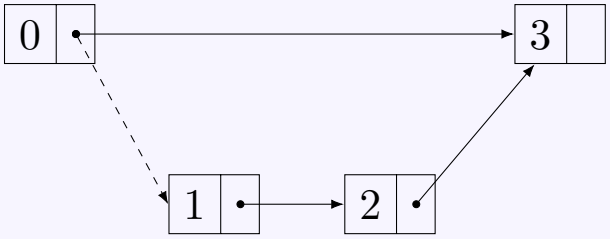
Η γραμμικοποιησιμότητα δεν μπορεί να εφαρμοστεί απευθείας στην επίμονη μνήμη λόγω των crashes. Για αυτό το λόγο έχει προταθεί μία επέκταση της, η οποία καλείται *ανθεκτική γραμμικοποιησιμότητα (durable linearizability)* [Izra16]. Η ανθεκτική γραμμικοποιησιμότητα επιβάλλει ότι έπειτα από ένα crash, όλες οι λειτουργίες που έχουν ολοκληρωθεί πριν από το crash θα πρέπει να έχουν εγγραφεί και στην EM και να είναι ανιχνεύσιμες κατά την ανάκτηση του προγράμματος. Μπορεί εύκολα να δειχθεί ότι οι δομές χωρίς κλειδώματα, που παρουσιάσαμε, δεν είναι ανθεκτικώς γραμμικοποιήσιμες, όπως φαίνεται στο Παράδειγμα 0.2 για την συνδεδεμένη λίστα του Harris.

Παράδειγμα 0.2: Παραβίαση της ανθεκτικής σειριοποιησιμότητας σε μία συνδεδεμένη λίστα

Ας θεωρήσουμε μία συνδεδεμένη λιστα αποτελούμενη από δύο κόμβους, όπου ο ένας περιέχει το κλειδί 0 (κόμβος n_0) και ο άλλος το κλειδί 3 (κόμβος n_3), όπως φαίνεται στο Σχήμα 0.2. Ας υποθέσουμε επίσης ότι δύο νήματα ($T1$ και $T2$) θέλουν να εισάγουν στη λίστα κόμβους με το κλειδί 1 (κόμβος n_1) και το κλειδί 2 (κόμβος n_2).

Σχήμα 0.2: Αρχική κατάσταση της λίστας με τους κόμβους n_0 και n_3.

Το νήμα $T1$ εκτελεί μία `CAS` εντολή και αλλάζει το δείκτη *next* του κόμβου n_0 προς τον κόμβο n_1, όποτε και σταματάει την εκτέλεσή του χωρις να εγγράψει τις αλλαγές στην EM. Έπειτα, το νήμα $T2$ ξεκινάει την εκτέλεσή του και εκτελεί μία `CAS`, η οποία αλλάζει το *next* δείκτη του κόμβου n_1, ώστε να δείχνει στον n_2. Τέλος, το $T2$ γράφει (με `flush`) τις αλλαγές στην EM και ολοκληρώνει επιτυχώς την εισαγωγή του n_2. Η κατάσταση της λίστας τη στιγμή αυτή φαίνεται στο Σχήμα 0.3. Μόλις τελειώσει, όμως, την εκτέλεσή του το $T2$, συμβάινει ένα crash.



Σχήμα 0.3: Κατάσταση της λίστας τη στιγμή πριν το crash. Οι συνεχείς γραμμές έχουν εγγραφεί στην EM, ενώ οι διακεκομένες όχι.

Μετά το crash, τόσο ο κόμβος n_1, όσο και ο n_2 έχουν χαθεί, αφού ο δείκτης από τον n_0 προς τον n_1 δεν έχει εγγραφεί στην EM πριν το crash. Έτσι, παρόλο που έχει ολοκληρωθεί η λειτουργία του $T2$ (εισαγωγή του n_2) πριν το crash, τα αποτελέσματά του δεν μπορούν να ανιχνευθούν κατά τη διαδικασία ανάκτησης. Αυτό είναι μία παραβίαση ανθεκτικής σειριοποιησιμότητας. Ας σημειώσουμε, ότι σε περίπτωση που το νήμα $T2$ είχε γράψει στην EM το δείκτη από τον n_0 στο n_1, τότε αυτό το παράδειγμα θα ήταν ανθεκτικώς γραμμικοποιήσιμο.

Στην πρόσφατη βιβλιογραφία έχουν προταθεί κάποιες τροποποιήσεις δομών χωρίς κλειδώματα, οι οποίες είναι ανθεκτικά γραμμικοποιήσιμες. Ένα παράδειγμα είναι η Persistent (ή Durable) Queue, η οποία είναι επέκταση της MS-Queue. Για την επίτευξη ανθεκτικότητας χρησιμοποιούνται προσεκτικά τοποθετημένες `flush` εντολές.

Η προσέγγιση για την επίτευξη ανθεκτικότητας της Persistent Queue προϋποθέτει πολύ καλή κατανόηση της δομής δεδομένων και πιθανότατα δεν μπορεί να γενικευτεί σε άλλες δομές. Για αυτόν το λόγο, έχουν προταθεί διάφοροι αυτόματοι μετασχηματισμοί. Ο πρώτος τέτοιος μετασχηματισμός καλείται μετασχηματισμός Izraelevitz [Izra16] και πρακτικά προσθέτει `flush` εντολές μετά από κάθε διάβασμα ή εγγραφή μίας μεταβλητής.

Ένας πιο αποδοτικός μετασχηματισμός είναι ο NVTraverse [Frie21a], ο οποίος μετατρέπει μία οικογένεια δομών δεδομένων ελεύθερες από κλειδώματα (στην οποία περιλαμβάνονται η συνδεδεμένη λίστα του Harris και η skiplist) σε ανθεκτικώς γραμμικοποιήσιμες με αυτόματο τρόπο.

Algorithm 1: Operation in a NVTraverse data structure

```
T Operation(Node root, T input):
    while true do
        Node entry = findEntry(root, input);
        List<Node> nodes = traverse(entry, input);
        ensureReachable(nodes.head());
        makePersistent(nodes);
        bool restart, T val = critical(nodes, input);
        if !restart then
            return val;
        end
    end
```

Ο Αλγόριθμος 1 παρουσιάζει τη λειτουργία μίας NVTraverse δομής δεδομένων, όπου με σκιασμένο χρώμα είναι οι μέθοδοι που προστίθενται για να εξασφαλίσουν την ανθεκτικότητα. Η μέθοδος **ensureReachable** εξασφαλίζει ότι υπάρχει ανθεκτικό μονοπάτι για τον κόμβο, στον οποίο γίνεται η εκάστοτε λειτουργία, ενώ η **makePersistent** κάνει flush τους αναγκαίους κόμβους του μονοπατιού, ώστε να διασφαλιστεί η ανθεκτική γραμμικοποιησιμότητα.

Σουίτα Ελέγχου για Επίμονη Μνήμη

Η κύρια συνεισφορά αυτής της εργασίας είναι η δημιουργία μίας σουίτας ελέγχου για προγράμματα επίμονης μνήμης, η οποία αποτελείται από:

1. **Litmus tests**. Τα litmus tests είναι μικρά συνθετικά benchmarks, τα οποία στοχεύουν στον έλεγχο του εργαλείου για συγκεκριμένες συμπεριφορές του μοντέλου που υλοποιεί.

2. **Tests δομών δεδομένων**. Προσεχτικά κατασκευασμένα tests ανθεκτικών και μη δομών δεδομένων χωρίς κλειδώματα.

Ένας στόχος αυτής της σουίτας ελέγχου είναι να ελέγξει την ορθότητα και την επεκτασιμότητα του εργαλείου PerSeVerE. Όπως θα δούμε σύντομα, μέσω της σουίτας μας μπορέσαμε να βρούμε μια εσωτερική αδυναμία του PerSeVerE, που οδηγούσε στη λάθος μοντελοποίηση μίας συμπεριφοράς του Px86. Ένας πιο γενικός στόχος της σουίτας είναι να χρησιμοποιηθεί ως βάση για τη δημιουργία σημείων αναφοράς (benchmarks) για εργαλεία επαλήθευσης ΕΜ προγραμμάτων.

Αρχικά, τονίζουμε ότι τα tests της σουίτας είναι δύο ειδών. Στην πρώτη κατηγορία είναι αυτά τα οποία έχουν κάποιο assertion, το οποίο δεν πρέπει να παραβιάζεται σε καμία εκτέλεση. Αυτά τα tests θεωρούνται ασφαλή. Στη δεύτερη κατηγορία βρίσκονται tests στα οποία κάποιο assertion αναμένεται να αποτυγχάνει και συμβολίζονται ως μη ασφαλή. Σε αυτά το PerSeVerE πρέπει να εντοπίζει κάποιο σφάλμα.

Θα ξεκινήσουμε την ανάλυση της σουίτας μας με την υλοποίηση του παραδείγματος WW, η οποία φαίνεται στο Πρόγραμμα 1. Για να ορίσουμε τις μεταβλητές `x` και `y` ως επίμονες, χρησιμοποιούμε τη μακροεντολή `__VERIFIER_persistent_storage()`. Αρχικοποιούμε τις δύο μεταβλητές με 0 και εκτελούμε το επίμονο φράγμα `__VERIFIER_pbarrier()` για να οριστεί η αρχική κατάσταση της ΕΜ. Στη ρουτίνα ανάκτησης `__VERIFIER_recovery_routine()` διαβάζουμε τις επίμονες μεταβλητές και εξετάζουμε με τη χρήση ενός assertion εάν μπορούν να πάρουν τον ανεπιθύμητο συνδυασμό $x = 0 \wedge y = 1$. Το PerSeVerE εντοπίζει παραβίαση του assertion, όπως φαίνεται και στο Σχήμα 0.4, γεγονός που σημαίνει ότι η κατάσταση αυτή γίνεται να επιτευχθεί.

```c
#include <stdio.h>
#include <stdlib.h>
#include <atomic>
#include <pthread.h>
#include <assert.h>
#include <genmc.h>

#define relaxed std::memory_order_relaxed

__VERIFIER_persistent_storage(std::atomic_int x);
__VERIFIER_persistent_storage(std::atomic_int y);

extern ''C''{
 void __VERIFIER_clflush(void*);
}

void __VERIFIER_recovery_routine(void)
{
  assert(!(x.load(relaxed) == 0 && y.load(relaxed) == 1));
  return;
}

int main()
{
  x.store(0, relaxed);
  y.store(0, relaxed);

  __VERIFIER_pbarrier();

  x.store(1, relaxed);
  y.store(1, relaxed);

  return 0;
}
```

Πρόγραμμα 1: `WW` litmus test

Μέσα από τη χρήση των litmus tests, καταφέραμε να βρούμε μία εσωτερική αδυναμία του PerSeVerE. Πιο συγκεκριμένα, το πρόγραμμα που φαίνεται στο Σχήμα 0.5, ανέδειξε ότι το PerSeVerE δεν μπορούσε να μοντελοποιήσει το `flush` μίας μεταβλητής από διαφορετικό νήμα από αυτό που έχει γίνει η εγγραφή. Οι δημιουργοί του PerSeVerE διόρθωσαν το πρόβλημα αυτό προσθέτωντας επιπλέον υποστήριξη στον πυρήνα του εργαλείου.

Όσον αφορά τα tests για τις δομές δεδομένων, πειραματιστήκαμε αρχικά με τη συνδεδεμένη λίστα και τη skiplist. Πιο συγκεκριμένα, ελέγξαμε τρεις εκδοχές αυτών των δομών:

1. Τις αρχικές εκδοχές των δομών (e.g., Harris' linked-list), οι οποίες δεν είναι ανθεκτικώς γραμμικοποιήσιμες.

2. Το μετασχηματισμό Izraelevitz.

3. Το μετασχηματισμό NVTraverse.

Τα tests μας εστίασαν σε δύο άξονες:

1. Να δείξουμε ότι οι αρχικές εκδοχές δεν είναι πράγματι ανθεκτικώς γραμμικοποιήσιμες, ενώ οι άλλες εκδοχές περνάνε επιτυχώς τους ελέγχους μας. Για αυτό, τα tests των αρχικών εκδοχών δηλώνονται ως ασφαλή, ενώ των δύο μετασχηματισμών ως μη ασφαλή.

2. Να φτιάξουμε κάποια μεγάλα και περίπλοκα tests για να δούμε πως ανταποκρίνεται σε αυτά το PerSeVerE.

```
Error detected: Recovery error!
Event (1, 2) in graph:
<−1, 0> main:
        (0, 1): Wrlx (x...__a_, 0) atomic:949
        (0, 2): Wrlx (y...__a_, 0) atomic:949
        (0, 3): PERSISTENCY_BARRIERrel L.31
        (0, 4): Wrlx (x...__a_, 1) atomic:949
        (0, 5): Wrlx (y...__a_, 1) atomic:949
        (0, 6): THREAD_END
<−1, 1> __VERIFIER_recovery_routine:
        (1, 1): Rrlx (x...__a_, 0) [(0, 1)] atomic:957
        (1, 2): Rrlx (y...__a_, 1) [(0, 5)] atomic:957
Coherence:
y...__a_: [ (0, 2) (0, 5) ]
x...__a_: [ (0, 1) (0, 4) ]

Assertion violation: !(x.load(relaxed) == 0 && y.load(relaxed) == 1)
Number of complete executions explored: 2
Total wall−clock time: 0.04s
```

Σχήμα 0.4: Έξοδος του PerSeVerE για το `WW` litmus test. Η έξοδος αποτελείται από το είδος του σφάλματος που εντοπίστηκε (assertion violation), την εκτέλεση που οδήγησε στο σφάλμα, τον αριθμό των εκτελέσεων που εξερεύνησε (και που μπλόκαρε) το εργαλείο, καθώς και το συνολικό χρόνο εκτέλεσης.

$$\begin{array}{l|l} \texttt{x} := 42; & \texttt{a} := \texttt{y}; \\ \texttt{y} := 7; & \texttt{if } (\texttt{a} \neq 0); \\ & \quad \texttt{flush x}; \\ & \quad \texttt{z} := 1; \end{array} \qquad \text{(2W+RFW)}$$

Σχήμα 0.5: Litmus test μεταφοράς μηνύματος (`2W+RFW`). Η μεταβλητή `x` γράφεται από το πρώτο νήμα και γίνεται `flush` από το δεύτερο.

$$((\texttt{sl})|\texttt{l})((\texttt{or})|(\texttt{iz})|(\texttt{tr})) - (((\texttt{pw})(+(\texttt{w}|\texttt{d})^+)^+)|((\texttt{w}|\texttt{d})^+(+(\texttt{w}|\texttt{d})^+)^*))$$

όπου

- το `l` αναφέρεται στη συνδεδεμένη λίστα, ενώ το `sl` αναφέρεται στη skiplist
- το `or` αναφέρεται στις απλές εκδοχές των δομών (π.χ. συνδεδεμένη λίστα Harris), το `iz` αναφέρεται στο μετασχηματισμό του Izraelevitz, ενώ το `tr` αναφέρεται στο μετασχηματισμό NVTraverse
- το `pw` σημαίνει ότι έχουν γίνει κάποιες εισαγωγές στη δομή πριν το φράγμα αρχικοποίησης EM
- ο όρος $+(\texttt{w}|\texttt{d})^+$ σημαίνει ότι ένα νέο νήμα δημιουργείται, το οποίο εκτελεί έναν αυθαίρετο αριθμό από εισαγωγές ή/και διαγραφές στοιχείων στη δομή.

Για παράδειγμα, το test `lor-pw+w+d` ελέγχει την αρχική εκδοχή της συνδεδεμένης λίστας, έχοντας εισάγει κάποια στοιχεία πριν το φράγμα αρχικοποίησης και στη συνέχεια δημιουργούνται δύο νήματα, όπου το ένα κάνει μια εισαγωγή και το άλλο μία διαγραφή.

Για τα tests της Persistent Queue ακολουθήσαμε παρόμοια ονοματοδοσία με το NVTraverse:

$$((\texttt{msq})|(\texttt{dq})) - (((\texttt{pe})(+(\texttt{e}|\texttt{d})^+)^+)|((\texttt{e}|\texttt{d})^+(+(\texttt{e}|\texttt{d})^+)^*))$$

όπου

- το `msq` αναφέρεται στην MS-Queue, ενώ το `dq` στην Durable Queue. Τα tests για την MS-Queue δηλώνονται ως μη ασφαλή, ενώ για την Durable Queue ως ασφαλή.
- `pe` σημαίνει ότι έχουν γίνει κάποιες εισαγωγές στη δομή πριν το φράγμα αρχικοποίησης EM
- ο όρος $+(\texttt{e}|\texttt{d})^+$ σημαίνει ότι ένα νέο νήμα δημιουργείται, το οποίο εκτελεί έναν αυθαίρετο αριθμό από εισαγωγές (enqueues) ή/και διαγραφές (dequeues) στοιχείων στη δομή.

Πέρα από τον απλό έλεγχο των δομών, πειραματιστήκαμε επίσης και με την αφαίρεση εντολών `flush` από τον κώδικα του NVTraverse. Σκοπός αυτής της διαδικασίας είναι να ελεγχθεί αν μπορούμε να αφαιρέσουμε κάποια από τα `flush` για να επιτύχουμε καλύτερη επίδοση ή αν είναι όλα απαραίτητα για να εξασφαλιστεί η ανθεκτική γραμμικοποιησιμότητα.

Η αφαίρεση των `flush` εντολών γίνεται με τη χρήση κάποιας σημαίας. Στο Πρόγραμμα 2 φαίνονται έχουμε δύο σημαίες που κάνουν αυτή τη δουλειά. Η σημαία `BIMF` αφαιρεί το `flush` μετά τη δημιουργία του προς εισαγωγή κόμβου, ενώ η `BICF` αφαιρεί το `flush` μετά την εκτέλεση της `CAS` για να αλλάξει το δείκτη του προηγούμενου κόμβου, ώστε να δείχνει στον κόμβο που εισάγεται.

```
1  bool insert(int k, int item) {
2    while (true) {
3      Window* window = find(head, k);
4      Node* pred = window->pred;
5      Node* curr = window->curr;
6      free(window);
7      if (curr && curr->key == k)
8        return false;
9
10     Node* node = getNewNode();
11     node->set(k, item, curr);
12 #ifndef BIMF
13     FLUSH(node);
14 #endif
15
16 #ifdef BICF
17     bool res = pred->CAS_next(curr, node);
18 #else
19     bool res = pred->CAS_nextF(curr, node);
20 #endif
21     if (res)
22       return true;
23   }
24 }
```

Πρόγραμμα 2: Κώδικας για τη χειροκίνητη αφαίρεση `flush` εντολών στη συνάρτηση εισαγωγής στοιχείων στην NVTraverse έκδοση της συνδεδεμένης λίστας. Φαίνονται δύο σημαίες που αφαιρούν `flush` (`BIMF` και `BICF`).

Ιδιαίτερο ενδιαφερόν για την αφαίρεση `flush` εντολών αποτελεί ο τρόπος που γίνονται `flush` αντικείμενα, όπως ο κόμβος της συνδεδεμένες λίστας. Η συνάρτηση `__VERIFIER_clflush()` του PerSeVerE παίρνει ως παράμετρο τη διεύθυνση μίας μεταβλητής και την κάνει `flush`. Όμως, όταν έχουμε μία ολόκληρη δομή, το PerSeVerE δεν κάνει `flush` όλα τα πεδία της, αλλά μόνο το πρώτο. Για να εξασφαλιστεί, λοιπόν, η σωστή εγγραφή των δεδομένων στην EM, πρέπει να χρησιμοποιήσουμε την `__VERIFIER_clflush()` για κάθε ένα από τα πεδία της δομής.

Στο Πρόγραμμα 3 έχουμε ορίσει τη σημαία BMFN, η οποία όταν ενεργοποιείται κάνει flush τη δομή και όχι το κάθε πεδίο ξεχωριστά.

```
1  void FLUSH(Node *n)
2  {
3  #ifdef BMFN
4    __VERIFIER_clflush(&n);
5  #else
6    __VERIFIER_clflush(&(n->key));
7    __VERIFIER_clflush(&(n->value));
8    __VERIFIER_clflush(&(n->next));
9  #endif
10 }
```

Πρόγραμμα 3: Η σημαία BMFN ενεργοποιεί τη λανθασμένη διαδικασία του flush του κόμβου μίας συνδεδεμένης λίστας.

Αποτελέσματα

Στον Πίνακα 0.1 παραθέτουμε τα αποτελέσματα για τα litmus tests. Όλα τα litmus tests επέστρεψαν τα αναμενόμενα αποτελέσματα και βλέπουμε ότι απαιτούν μικρό αριθμό από εκτελέσεις. Αυτό είναι λογικό, καθώς η πολυπλοκότητά τους είναι μικρή.

		Εκτελέσεις		
	Αναμ.Αποτέλεσμα	Πλήρεις	Φραγμένες	Χρόνος
2W+2W	μη ασφαλές	3		0.04
2W+RFW	ασφαλές	3		0.04
2WRW+WFW	μη ασφαλές	1		0.04
6W	ασφαλές	4		0.04
CAS+CAS	ασφαλές	5		0.04
WFW	ασφαλές	2		0.04
WFW+RW	ασφαλές	3		0.04
WMW+WFW	μη ασφαλές	2		0.04
WW	μη ασφαλές	2		0.04
WW+RMFW	ασφαλές	3		0.04

Πίνακας 0.1: Αποτελέσματα της επάληθευσης ορθότητας μικρών προγραμμάτων

Στον Πίνακα 0.2 παραθέτουμε τα αποτελέσματα για τα tests του NVTraverse. Οι αρχικές εκδοχές τόσο της συνδεδεμένης λίστας, καθώς και της skiplist, έχουν οριστεί ως μη ασφαλή, επειδή δεν είναι ανθεκτικώς γραμμικοποιήσιμες. Όλα τα σχετικά tests εντοπίζουν κάποια παραβίαση συνέπειας κατά το recovery.

Τα υπόλοιπα tests ορίζονται ως ασφαλή, καθώς δεν αναμένουν κάποια παραβίαση ανθεκτικής σειριοποιησιμότητας. Πράγματι, σχεδόν όλα τα tests επιστρέφουν τα αναμενόμενα αποτελέσματα, εκτός από δύο. Το liz-pw+d+d παράγει εσωτερικό σφάλμα στο PerSeVerE, ενώ το sltr-pw+w+w εντοπίζει κάποια παραβίαση ορθότητας, η οποία πιθανόν οφείλεται σε κάποιο λάθος στην υλοποίηση της skiplist, π.χ. λείπει κάποιο flush. Τέλος, τα μεγάλα tests με τα τρία νήματα παράγουν πάνω από 40 χιλιάδες εκτελέσεις για την απλή συνδεδεμένη λίστα, ενώ για την skiplist ξεπερνούν το όριο χρόνου που έχουμε θέσει, το οποίο είναι 24 ώρες.

Στον Πίνακα 0.3 φαίνονται τα αποτελέσματα για την Persistent Queue. Έχουμε υλοποιήσει λιγότερα tests για την Persistent Queue σε σύγκριση με το NVTraverse, καθώς είναι λιγότερο περίπλοκη και δεν επιδεικνύει πολλές προβληματικές συμπεριφορές. Όπως και με το NVTraverse, οι μη ανθεκτικώς γραμμικοποιήσιμες εκδοχές ορίζονται ως μη ασφαλή και τα tests για την Durable Queue ως ασφαλή. Τα αποτελέσματα είναι τα αναμενόμενα.

		Εκτελέσεις		
	Αναμ.Αποτέλεσμα	Πλήρεις	Φραγμένες	Χρόνος
`lor-ww`	μη ασφαλές	5		0.10
`lor-pw+d+d`	μη ασφαλές	9		0.28
`lor-pw+w+d`	μη ασφαλές	3		0.15
`lor-pw+w+w`	μη ασφαλές	2		0.15
`lor-pw+ww`	μη ασφαλές	6		0.17
`lor-pw+w+w+d`	μη ασφαλές	3		0.18
`lor-pw+w+ww`	μη ασφαλές	6		0.26
`liz-ww`	ασφαλές	3		0.13
`liz-pw+d+d`	ασφαλές	!!	!!	!!
`liz-pw+w+d`	ασφαλές	116		3.59
`liz-pw+w+w`	ασφαλές	35		1.37
`liz-pw+ww`	ασφαλές	3		0.16
`liz-pw+w+w+d`	ασφαλές	42293		1981.83
`liz-pw+w+ww`	ασφαλές	123		7.77
`ltr-ww`	ασφαλές	3		0.17
`ltr-pw+d+d`	ασφαλές	118	2	10.41
`ltr-pw+w+d`	ασφαλές	76		3.11
`ltr-pw+w+w`	ασφαλές	25		1.38
`ltr-pw+ww`	ασφαλές	3		0.20
`ltr-pw+w+w+d`	ασφαλές	13902		1019.78
`ltr-pw+w+ww`	ασφαλές	108		10.04
`slor-pw+d+d`	μη ασφαλές	4		0.14
`slor-pw+w+w`	μη ασφαλές	2		0.16
`slor-pw+w+w+d`	μη ασφαλές	5		0.32
`sliz-pw+d+d`	ασφαλές	7119	16	2869.06
`sliz-pw+w+w`	ασφαλές	15		7.39
`sliz-pw+w+w+d`	ασφαλές	🕒	🕒	🕒
`sltr-pw+d+d`	ασφαλές	26732	14	12247.60
`sltr-pw+w+w`	~~ασφαλές~~	~~1812~~	~~1~~	~~460.08~~
`sltr-pw+w+w+d`	ασφαλές	🕒	🕒	🕒

Πίνακας 0.2: Αποτελέσματα της επάληθευσης ορθότητας για το NVTraverse. 🕒: όριο χρόνου (>24ώρες), **!!**: εσωτερικό πρόβλημα στο PerSeVerE, ~~διαγράμμιση~~: λάθος αποτελέσμα (σε σύγκριση με το αναμενόμενο)

		Εκτελέσεις		
	Αναμ.Αποτέλεσμα	Πλήρεις	Φραγμένες	Χρόνος
`ms-e`	μη ασφαλές	3		0.08
`ms-pe+e+e`	μη ασφαλές	2		0.10
`dq-e`	ασφαλές	1		0.10
`dq-pe+e+d`	ασφαλές	6		0.18
`dq-pe+e+e`	ασφαλές	28	24	0.53
`dq-pe+d+d`	ασφαλές	28	2	0.83

Πίνακας 0.3: Αποτελέσματα της επάληθευσης ορθότητας για την Persistent Queue

Στον Πίνακα 0.4 παρουσιάζονται τα αποτελέσματα για τη διαδικασία αφαίρεσης `flush` εντολών. Όπως φαίνεται, οι περισσότερες σημαίες παράγουν σφάλμα σε τουλάχιστον ένα test. Αυτό υποδηλώνει ότι όλες οι `flush` εντολές είναι απαραίτητες για την εξασφάλιση ανθεκτικής γραμμικοποιησιμότητας. Η μόνη εξαίρεση είναι η σημαία `BRMF`, η οποία αφαιρεί το flush αμέσως μετά την πρόσβαση στον κόμβο που πρόκειται να αφαιρεθεί. Αυτό, ωστόσο, δεν σημαίνει ότι αυτό το `flush` μπορεί να αφαιρεθεί. Είναι πιθανό να ανακαλυφθεί κάποιο σφάλμα σε πιο περίπλοκα tests.

Το test `ltr-pw+w+w-tmf` αντιστοιχεί επακριβώς στο Παράδειγμα 0.2, καθώς έχει αφαιρεθεί το `flush` που εισάγεται από το NVTraverse για να εξασφαλιστεί ότι το μονοπάτι προς τον κόμβο που εισάγεται έχει εγγραφεί στην EM. Επίσης, μπορούμε να παρατηρήσουμε ότι η αφαίρεση `flush` εντολών οδηγεί σε περισσότερες εκτελέσεις, όπως για παράδειγμα στην περίπτωση του test `ltr-pw+w+d-tmf`, στο οποίο εξερευνείται μία επιπλέον εκτέλεση σε σύγκριση με την εκδοχή με όλα τα `flush`. Αυτό είναι λογικό, καθώς αφαιρώντας `flush` εντολές επιτρέπουμε περισσοτέρες πιθανές καταστάσεις μετά από κάποιο crash.

		Εκτελέσεις		
	Αποτέλεσμα	Πλήρεις	Φραγμένες	Χρόνος
`ltr-pw+w+w-mfn`	μη ασφαλές	2		0.55
`ltr-pw+w+d-mfn`	μη ασφαλές	3		0.29
`ltr-pw+w+w-imf`	μη ασφαλές	2		0.59
`ltr-pw+w+d-imf`	μη ασφαλές	3		0.19
`ltr-pw+w+w-icf`	μη ασφαλές	2		0.21
`ltr-pw+w+d-icf`	ασφαλές	76		3.37
`ltr-pw+d+d-tmf`	ασφαλές	119	2	5.77
`ltr-pw+w+w-tmf`	μη ασφαλές	2		0.23
`ltr-pw+w+d-tmf`	ασφαλές	77		1.63
`ltr-pw+d+d-rmf`	ασφαλές	118	2	11.30
`ltr-pw+w+d-rmf`	ασφαλές	76		3.43
`ltr-pw+d+d-rcf`	μη ασφαλές	71		7.29
`ltr-pw+w+d-rcf`	ασφαλές	76		3.41

Πίνακας 0.4: Αποτελέσματα της επάληθευσης ορθότητας για το NVTraverse με αφαίρεση ορισμένων `flush` εντολών..

Τέλος, κάναμε και μία σύγκριση μεταξύ του ελέγχου μοντέλου για μοντέλα συνέπειας και μοντέλα επιμονής. Τα αποτελέσματα φαίνονται στον Πίνακα 0.5. Για τον έλεγχο της συνέπειας χρησιμοποιήσαμε το GenMC με το TSO μοντέλο συνέπειας. Όπως αναμένεται, παρατηρούμε ότι στην περίπτωση του ελέγχου της επιμονής, το εργαλείο εξερευνεί πολύ μεγαλύτερο αριθμό εκτελέσεων, εώς και 4 τάξεις παραπάνω (`sl-pw+d+d`). Αυτό είναι απολύτως λογικό, καθώς στον έλεγχο της επιμονής πρέπει να ληφθούν υπόψη όχι μόνο η συνέπεια του προγράμματος, αλλά και οι πιθανές καταστάσεις μετά από ένα crash.

Επίλογος

Η εργασία αυτή πραγματεύτηκε την τομή διαφορετικών πεδίων και πιο συγκεκριμένα του ελέγχου ορθότητας μέσω τεχνικών ελέγχου μοντέλου, της EM και των μοντέλων επιμονής και των δομών δεδομένων χωρίς κλειδώματα. Παρουσιάσαμε μία σουίτα ελέγχου προγραμμάτων EM, η οποία μπορεί να χρησιμοποιηθεί από εργαλεία ελέγχου μοντέλου για την επαλήθευση ορθότητας των προγραμμάτων, αλλά και των ίδιων των εργαλείων.

		GenMC TSO			PerSeVerE		
	Ανάμ.Αποτέλεσμα	Πλήρεις	Φραγμένες	Χρόνος	Πλήρεις	Φραγμένες	Χρόνος
`CAS+CAS`	ασφαλές	4		0.04	5(44)		0.04
`l-pw+w+d`	ασφαλές	14		0.09	126	2	5.60
`l-pw+w+w+d`	ασφαλές	1543	2	5.49	13902		1019.78
`sl-pw+d+d`	ασφαλές	50	1	0.26	26732	14	12247.60
`sl-pw+w+w+d`	ασφαλές	1400		9.20	🕒	🕒	🕒

Πίνακας 0.5: Σύγκριση ελέγχου συνέπειας και επιμονής

Μέσα από τα tests της σουίτας μας μπορέσαμε να βρούμε ένα εσωτερικό πρόβλημα του PerSeVerE, το οποίο αναφέρθηκε στους δημιουργούς του εργαλείου, οι οποίο στη συνέχεια πρόσθεσαν επιπλέον υποστήριξη στον πυρήνα του εργαλείου για να αντιμετωπιστεί το πρόβλημα. Επιπλέον, επιβεβαιώσαμε ότι οι απλές εκδοχές κάποιων βασικών δεδομένων χωρίς κλειδώματα δεν είναι ανθεκτικά γραμμικοποιήσιμες, ενώ οι ανθεκτικές εκδοχές τους περάσαν όλους τους ελέγχους μας. Τέλος, με την αφαίρεση `flush` εντολών από την υλοποίηση του NVTraverse, δείξαμε την αναγκαιότητα τους για να εξασφαλίσουν την ορθότητα της δομής, καθώς προέκυπταν παραβιάσεις ορθότητας.

Αναφορικά με σχετική βιβλιογραφία, αλλά και μελλοντική επέκταση της σουίτας μας, υπάρχουν πολλές βιβλιοθήκες EM δημόσια διαθέσιμες, όπως η PMDK [Inte15]. Επιπλέον, υπάρχει μεγάλη ποικιλία υλοποιήσεων δομών δεδομένων για EM [Lee19, Izad21, Cai21, Kim21], όπως key-value stores, δέντρα κ.λπ. Με παρόμοιο τρόπο με το NVTraverse, έχουν προταθεί ανθεκτικοί μετασχηματισμοί γραμμικοποιήσιμων δομών δεδομένων [Frie21b], καθώς και ορισμένες βιβλιοθήκες που απαιτούν ελάχιστη παρέμβαση του χρήστη για την επίτευξη ανθεκτικότητας [Wei22]. Μία μελλοντική επέκταση της σουίτας μας, θα μπορούσε να περιλάβει πολλές από αυτές τις δομές και παραδείγματα από τις βιβλιοθήκες.

Επίσης, θα είχε ενδιαφέρον να προστεθούν και άλλα μοντέλα στο PerSeVerE (με τη χρήση του *kater*), όπως αυστηρά μοντέλα επιμονής, π.χ. TSOPER [Ekem21], αλλά και άλλων αρχιτεκτονικών, π.χ. PARMv8 [Raad18]. Θα μπορούσε, επιπλέον, να μελετηθεί η ευρωστία (robustness) μεταξύ χαλαρών και αυστήρων μοντέλων επιμονής, δηλαδή να εξεταστεί κατά πόσο το υπό εξέταση πρόγραμμα εμφανίζει τις ίδιες συμπεριφορές μεταξύν των δύο μοντέλων.

Τέλος, θα μπορούσε να προστεθεί υποστήριξη για τις εντολές `clflushopt` και `sfence` στο PerSeVerE, έτσι ώστε να μελετηθούν κάποιες βελτιστοποιημένες εκδοχές των δομών δεδομένων.

Κείμενο στα αγγλικά

Chapter 1

Introduction

1.1 Overview

In computer architecture, one of the most promising technological breakthroughs of recent years is the introduction of Persistent Memory. Persistent Memory aims to replace traditional DRAMs, while being non-volatile, i.e., preserving its contents after a crash, like disks do.

However, writing crash-safe programs for Persistent Memory is tricky. Crashes can happen at any time during the program's execution and can lead to inconsistent states during the recovery process. Therefore, programmers have to carefully reason about the behaviour of their programs with respect to crashes. This behaviour can be formalized with the use of persistency memory models, which define the order in which memory operations persist.

In this thesis, we are going to discuss how persistency memory models can be used in combination with model checking, which is a powerful verification technique, in order to check the correctness of various Persistent Memory programs. Persistency model checking introduces new challenges. For example, a crash can happen at any time during program execution leading to a far bigger state space than non-crashing concurrent programs. Recent model checking tools use state space reduction techniques like Dynamic Partial Order Reduction [Flan05, Abdu14] in order to avoid exploring redundant executions. One such tool is PerSeVerE [Koko21a], which we are going to use in this thesis.

An important class of concurrent programs are those that implement so called *lock-free data structures*. These data structures ensure that the memory is always in a consistent state. As a result, they may seem like very intuitive fit for Persistent Memory. However, in a crashing execution it is possible for these data structures to be left in an inconsistent state due to cache volatility. There have been various studies on how to ensure durability for lock-free data structures [Frie18, Frie21a] and we are going to investigate how to employ persistency model checking in order to test them.

In summary, the new challenges that arise with the introduction of Persistent Memory create new opportunities for research in the field of verification. In this thesis, we are going to discuss the basics of model checking, persistency memory models and lock-free data structures and we are going to employ model checking in order to verify Persistent Memory programs.

1.2 Contributions

The main contribution of this thesis is a test suite that can be used by testing and model checking tools to test Persistent Memory programs. This suite consists of some litmus tests and some tests on lock-free data structures. The former can be used to check some basic and more complex behaviours of the Px86 persistency model. The latter can test that the naive implementations of some lock-free data structures are not durably linearizable, while certain implementations from recent literature, which use carefully placed explicit persist instructions, are in fact durably linearizable. Furthermore, we added some tests on these lock-free data structures that can serve as benchmarks to check the efficiency of persistency

verification tools and the reduction that these tools can achieve. This test suite can serve as a guideline for using model checking to verify durable data structures and persistent libraries. As our testing tool we used PerSeVerE, which is a project under continuous development for persistency model checking. Through these tests we were able to point out some missing behaviour of PerSeVerE, which was later added to the tool.

1.3 Organization

This thesis is structured as follows:

- In Chapter 2, we summarize the basics of memory consistency models and the stateless model checking techniques that we are going to use for verifying the desired data structures
- In Chapter 3, we discuss the new opportunities that come with the recent development of Non-Volatile Memories and the challenges that arise with programming Persistent Memory programs. Furthermore, we delve into the details of memory persistency models with emphasis on the persistency model of Intel's x86 architecture and analyze how model checking can be utilized for verifying Persistent Memory programs designed for that architecture.
- In Chapter 4, we explore some basic designs principles of lock-free data structures and analyze how these data structures can be adapted in order to provide safety guarantees when they are used on a machine that utilizes Persistent Memory.
- In Chapter 5 we present the main contribution of this thesis, which is the design of a test suite for verification of Persistent Memory programs.
- In Chapter 6, we discuss the results from running our test suite.
- Finally, in Chapter 7, we conclude with a discussion on some related work and propose some possible future extensions of our work.

Chapter 2

Memory Consistency and Model Checking

2.1 Memory Consistency

Modern computer systems utilize multiple cores in order to achieve better performance. The most common memory architecture for these systems is the shared memory model [Adve96]. This model implies that the various cores share a common main memory, which can be accessed with *load*, *store* and *read-modify-write (e.g. Compare-And-Swap)* instructions. Memory operations can be issued concurrently by multiple cores. Finally, each core can have its own cache hierarchy.

A major question arises from the simultaneous issue of memory operations by multiple cores: *In what order do memory operations become visible to other threads and to main memory?* The answer to this question is given by memory consistency models. Memory consistency models specify the allowed behaviour of multithreaded programs on a computer system that supports the shared memory architecture and therefore allow programmers to reason about the visible order of reads and writes among the various threads.

We are going to focus on two prominent consistency models, namely Sequential Consistency and Total Store Order.

2.1.1 Sequential Consistency (SC)

The definition of Sequential Consistency was given by Leslie Lamport on his famous paper *"How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs"*.

Definition 2.1 (Sequential Consistency [Lamp79])**.** *A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program.*

Essentially, sequential consistency guarantees that a global total order of loads and stores exists and is observed in the same way by all threads. As stated in Definition 2.1, the program order of each thread is preserved, i.e., loads and stores are ordered with respect to program order. Finally, under SC, each read reads from the most recent write to the same location in that total order.

Lamport makes use of a FIFO queue to describe a sequentially consistent system. However, an easier way to illustrate SC is shown in Figure 2.1. A switch is placed between the various processors and the main memory and selects one core at a time to execute a memory operation. The switch waits until the core completes the operation, which makes the operation appear to execute *atomically*, and subsequently repeats this procedure. It should be noted that the assignment of which core executes an operation does not affect the consistency of the model (it can be random), but instead it just defines a memory order that is sequentially consistent.

Although SC seems to match the programmer's intuition that there exists a single main memory and all operations *must* happen atomically in (program) order, it showcases two

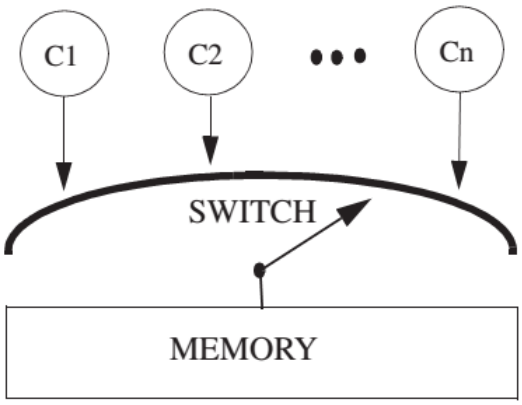


Figure 2.1: Modeling Sequential Consistency with a switch [Sori11].

main drawbacks. Firstly, it restricts many hardware and compiler optimizations in order to prevent reordering of operations or out-of-order executions. Secondly, SC is extremely slow. Since only one operation is allowed to execute at a time, while the processors should wait until this operation completes (in order to get a chance to be selected by the switch), the benefit of writing parallels programs is greatly diminished. As a result, most computer systems avoid implementing SC as their memory consistency model.

2.1.2 Total Store Order (TSO)

In order to achieve better performance, computer architects have proposed more *relaxed* memory models than SC. A very common memory model is **Total Store Order** (TSO). TSO introduces the notion of *store buffers*. Store buffers are First-In-First-Out (FIFO) buffers that lie between the thread's core and the memory hierarchy (caches and main memory).

The role of the store buffer is to hold pending write operations instead of immediately propagating them into the memory hierarchy. More precisely, when a thread executes a store, this store is delayed and placed in the store buffer. This delayed write will be propagated to the memory hierarchy and will therefore become visible to other threads non-deterministicly at some future time. It will maintain, however, the order in which it inserted the buffer, since the buffer is FIFO.

This practice may seem odd at first, but it produces huge performance improvement. The reason for that is the fact that it actually hides the latency of the write, which can be extremely costly if it leads to a cache miss or if the cache is write-through. The execution can proceed to the next instruction. Contrary to writes, loads can be executed immediately. More concretely, when a threads issues a load instruction, it first searches its own store buffer. If multiple entries of the memory location that is being read exist, the thread reads the last buffered write in that location. If no entry for that location exists in the thread's store buffer, it will then try to retrieve the value from memory. Essentially TSO allows loads to execute before earlier in program order stores, which helps to hide the write latency.

The implementation of TSO can also be modeled with a switch [Sori11] as shown in Figure 2.2. The only difference between SC and TSO is the addition of the store buffer for every core and the procedure that is followed after every store instruction, as described above.

TSO seems like a solid improvement from the poor performing SC model. There is, however, a catch: TSO can produce some counter-intuitive results like the one in Example 2.1.

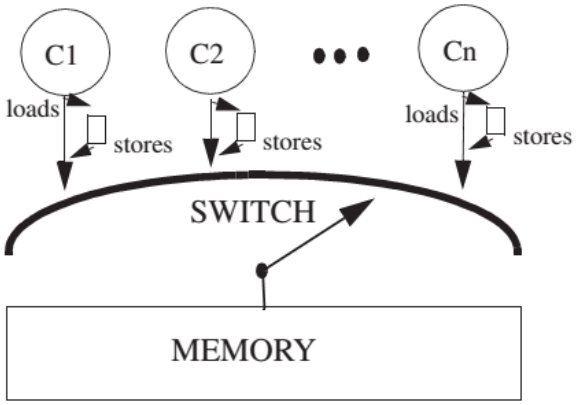


Figure 2.2: Modeling TSO with a switch and store buffers [Sori11].

Example 2.1: SC and TSO behaviours

Consider the following mutlithreaded program, where `x` and `y` are memory locations initialized with 0 and $\mathtt{r}_1$ and $\mathtt{r}_2$ are registers:

$$\begin{array}{l} \mathtt{x:=1;}\ (1) \\ \mathtt{r_1:=y;}\ (2) \end{array} \Bigg\| \begin{array}{l} \mathtt{y:=1;}\ (3) \\ \mathtt{r_2:=x;}\ (4) \end{array} \tag{SB}$$

Figure 2.3: Store buffering litmus test showing the difference between SC and TSO.

We want to find all possible combinations of values that the registers $\mathtt{r}_1$ and $\mathtt{r}_2$ can hold when the two threads have completed their execution.

Firstly, we are going to consider SC. It is easy to enumerate all the possible interleavings of this program as follows:

- $(1) \rightarrow (2) \rightarrow (3) \rightarrow (4) : \mathtt{r}_1 = 0 \wedge \mathtt{r}_2 = 1$
- $(1) \rightarrow (3) \rightarrow (2) \rightarrow (4) : \mathtt{r}_1 = 1 \wedge \mathtt{r}_2 = 1$
- $(1) \rightarrow (3) \rightarrow (4) \rightarrow (2) : \mathtt{r}_1 = 1 \wedge \mathtt{r}_2 = 1$
- $(3) \rightarrow (1) \rightarrow (2) \rightarrow (4) : \mathtt{r}_1 = 1 \wedge \mathtt{r}_2 = 1$
- $(3) \rightarrow (1) \rightarrow (4) \rightarrow (2) : \mathtt{r}_1 = 1 \wedge \mathtt{r}_2 = 1$
- $(3) \rightarrow (4) \rightarrow (1) \rightarrow (2) : \mathtt{r}_1 = 1 \wedge \mathtt{r}_2 = 0$

As we can see, there is no execution under SC in which the two registers have the value 0. However, this is not the case under TSO. Let's consider the interleaving $(1) \rightarrow (3) \rightarrow (2) \rightarrow (4)$. In this case, both the store to `x` and the store to `y` are stored in their thread's store buffer. Afterwards the two registers read `x` and `y`, but since there is no entry in their store buffer, they get the values from memory, which stills holds 0 for both `x` and `y`. Finally, at some future point the entries in the store buffers are propagated to main memory. Therefore, after the execution of the program we have the state $\mathtt{r}_1 = 0 \wedge \mathtt{r}_2 = 0$.

In order to control the behaviours introduced by TSO, programmers can use memory fence instructions, like `mfence`. Executing these instructions ensures that all store operations prior to the fence (in program order) will take effect in memory before the operations that are after the fence. In other words, a fence instruction flushes the store buffer of the executing thread.

Using fences it is possible to regain SC. This can happen in case a fence is added after every instruction, like in Example 2.2

Example 2.2: Use of `mfence` to restore SC

Placing an `mfence` after the store of each thread in SB, one can regain SC behaviour, since the fences would require to empty the store buffers and therefore the results of the two stores would be visible by the following reads.

$$\begin{array}{l|\!|l} \texttt{x} := 1; & \texttt{y} := 1; \\ \texttt{mfence}; & \texttt{mfence}; \\ \texttt{r}_1 := \texttt{y}; & \texttt{r}_2 := \texttt{x}; \end{array} \qquad \text{(SBF)}$$

Figure 2.4: Restoring SC with the use of `mfence` in SB.

Although TSO can produce these counter-intuitive behaviours, the performance improvements substantially outweigh the difficulties that may arise for programmers, and therefore store buffers have been adopted by every modern architecture. There are various architectures that support the TSO consistency model, such as SPARC, x86, RISC-V and AMD64, while there are architectures that support even more relaxed memory models.

2.1.3 Axiomatic versus Operational Memory Models

The formulation of SC and TSO was done in an operational manner. *Operational* memory models provide abstractions of the actual underlying hardware with the use of ideal components, such as queues, buffers and caches. They are very useful, since they present intuitive approaches to the memory model and can be easily simulated [Algl12].

However, using operational models is not the only way to formalize memory models. *Axiomatic* or *declarative* models define the allowed behaviours of the memory models by creating constraints on order relations of the various memory operations. There are three main relations:

- **po**, which is the *program order*, i.e. the order in which events like stores and reads appear in the program
- **rf**, which is called the *reads-from* relation and it relates each read with the write that it obtains the value to be read (note that this value can be the initial one, i.e. there has been no write on the variable that is being read).
- **mo**, which is called the *modification* or *coherence* order and is essentially the order in which writes reach the memory.

Based on these basic relations one can represent concurrent programs as graphs, in which memory events (like stores, reads, fences, etc.) form a set E and represent the nodes of the graph and these relations define their edges. Programs executions are therefore defined as tuples in the form $G = (E, \mathbf{po}, \mathbf{mo}, \mathbf{rf})$, where G is the graph. Axiomatic memory models define the allowed execution graphs by providing certain axioms that these graphs should

abide by. Furthermore, it is possible to use relational algebra to expand upon these relations in order to describe more complex memory models.

Example 2.3: Defining SC with the use of axiomatic models

We are going to define a new relation:

$$\mathbf{fr} \triangleq G.\mathbf{rf}^{-1}; \mathbf{mo}$$

This relation is called *from-reads* and relates each read to a write, which succeeds the write (ordered by **mo**) from which the read takes its value.

With the use of the **fr** relation, we can define the SC consistent executions as all the executions $G = (E, \mathbf{po}, \mathbf{mo}, \mathbf{rf})$ such that $\mathbf{po} \cup \mathbf{mo} \cup \mathbf{rf} \cup \mathbf{fr}$ is acyclic. In terms of relational algebra, the acyclicity implies that the transitive closure of the above union of relations should be irreflexive.

2.2 Verification

Verification is the process of ensuring that a certain program meets a given specification. Verifying a program essentially means mathematically proving that the program is correct under a specific notion of correctness, usually linked with the specification. There are various areas of computer science that deal with verification on both software and hardware level, e.g. formal methods.

2.2.1 Stateless Model Checking

It is important to segregate testing from verification. Testing usually proves that a program/system is *partially* correct, while verification means proving that it actually meets its correctness requirements [Gode96]. For instance, testing a program by giving input-output examples does not fully guarantee that the program is correct under all circumstances, since there might be some paths and branches that are not covered by the examples or there might me some non-determinism in the system under test, like thread interleavings.

However, there is a specific method of testing that can act as a verification procedure, namely *model checking*. Model checking refers to the systematic state space exploration of a program, while checking whether a certain property holds under each state. Model checking tests every possible behaviour of a program and therefore can actually verify it.

In order for model checking to actually work, the number of possible states should be finite. A naive approach to model checking would be to simulate the whole system and perform a search by calculating all possible states. However, this approach requires capturing and storing each global state, which is problematic for real-word machines. A novel implementation of a state space exploration algorithm is shown in Algorithm 2. Note that the exploration algorithm is actually a Breadth First Search (BFS) over global states.

Early model checkers were operating on abstractions of real systems and programming languages in order to reduce the cost of each state. Such approaches were not able to deal with realistic programs either on software or hardware level.

To deal with this problem, *stateless model checking* was introduced [Gode97]. *Stateless model checking* or *state space exploration* searches over the whole state space without storing global states of the system. In order to test real programs, a run-time scheduler is needed, which drives the execution so that it explores all states [Flan05].

Algorithm 2: Classical state space exploration

```
Stack: S ← ∅;
Set: H ← ∅;
Function Explore(s_0):
    /* s_0 is the initial state of the system                          */
    S.add(s_0);
    while S ≠ ∅ do
        s ← S.pop();
        if s ∉ H then
            H.add(s);
            /* enabled(s) returns all possible events that can be executed
               in state s                                               */
            E ← enabled(s);
            foreach e ∈ E do
                /* s.next(e) returns the state of the system after executing
                   event e in state s                                   */
                s' ← s.next(e);
                S.add(s');
            end
        end
    end
    return;
```

2.2.2 Partial Order Reduction

The main problem of stateless model checking is that the number of states usually grows exponentially with respect to the program length. This is called the *state explosion problem*. Let's take for example N threads, each consisting of a single event, as shown in Example 2.4.

Example 2.4: Combinatorial explosion

Consider the following program consisting of N threads:

$$\mathtt{x}_1 := 1; \parallel \mathtt{x}_2 := 1; \parallel \cdots \parallel \mathtt{x}_N = 1; \tag{NT}$$

Figure 2.5: N threads performing a write to variables $\mathtt{x}_1, \cdots, \mathtt{x}_N$.

There are $N!$ possible interleavings. Note, however, that since this program consists of a single write on a different variable for each thread, there is no difference on the outcome of each interleaving.

As illustrated in Example 2.4, there might be some interleavings that can be considered equivalent and their exploration is redundant. To determine whether interleavings are equivalent, the use of *partial order methods* [Gode96] or *partial order reduction* (*POR*) has led the research on the field.

Essentially, POR methods are based on the fact that concurrent program executions can be expressed as partial orders of their events, since concurrent events are considered un-

ordered. The main purpose of POR methods is to explore a reduced state space, which is, however, provably sufficient to verify the program under test. This reduced state space consists of the so called equivalence classes, i.e. a representative execution of a set of equivalent executions (e.g. two executions that have swapped adjacent independent events), and POR methods try to explore exactly one execution per equivalence class.

Early POR methods employed static analysis to extract potential conflicts and data races beforehand. More sophisticated approaches have recently been introduced, namely Dynamic Partial Order Reduction [Flan05, Abdu14, Abdu17, Aron18]. This technique records occurring conflicts and calculates a set of provably sufficient subset of enabled processes on the fly during the exploration. One such state-of-the-art tool is Nidhugg, which supports algorithms for SC [Abdu14, Abdu19] and relaxed memory models such as TSO and PSO [Abdu15].

2.2.3 GenMC

Another model checking tool for `C/C++` programs that was recently introduced and employs a DPOR technique is GenMC [Koko21b]. Contrary to Nidhugg, GenMC is parametric to the memory model under test, i.e., it employs a common internal algorithm for all memory models, while Nidhugg uses various algorithms depending on the memory model.

This internal algorithm is based on axiomatic semantics of the memory model. More specifically, GenMC incrementally constructs the execution graphs of a program, by checking for consistency violations (i.e. by checking whether the constructed relations satisfy the memory model's axioms) each time a new event is added to the graph. As a result, it avoids exploring executions that are inconsistent with respect to the memory model. Simultaneously, when GenMC encounters a read that may produce more than one consistent graph by getting its value from different writes, it selects one path and saves the alternative to a working set in order to be explored later. Once a complete execution of the program is produced, GenMC picks an option from the working set and revisits the graph.

A `C` or `C++` program with some assertions is given as input to GenMC, which compiles it to the intermediate representation of LLVM (LLVM-IR). Afterwards, certain transformations (e.g., bounding infinite loops) take place in order to make the program more suitable for stateless model checking. Finally, the tool employs its internal algorithm to check the executions of the program.

GenMC has some built-in implementations of the basic relations of axiomatic memory models, like **po**, **rf** and **mo**. A new memory model can be added to GenMC by implementing the necessary relations needed apart from the built-in ones, as well as the consistency checker based on these relations. There are certain requirements that the memory model should satisfy in order to be suitable for GenMC, like **po** **rf** acyclicity, extensibility, prefix-closedness and well-blocking. (Refer to the original paper [Koko19] for more details on these requirements.)

2.2.4 kater

In order to make the introduction of new memory models to GenMC easier, there is an under development project named kater, which takes as input a file written in the *kat* language (which is very similar to relational algebra) and produces source code for the consistency checker of GenMC. This tool is heavily inspired by herd, a tool for modelling and simulating weak memory models [Algl14].

The definition of SC in a declarative style is given in Example 2.3. It is really easy to implement this definition of SC in the *kat* language. In a similar fashion, one can define TSO in relational algebra[1]. The *.kat* files for SC and TSO are shown in Listing 2.1 and Listing 2.2

[1]We are going to omit the formal definition of TSO in a declarative way. For this definition we refer the reader to herd [Algl14] and GenMC [Koko19].

respectively.

```
let sc = (po | rf | mo | fr)

acyclic sc
```

Listing 2.1: .kat file for SC

```
save[ppo] ppo = ([R]; po | po ; [F] | [UW|F] ; po | po ; [W])+

let tso = (ppo | rfe | mo | fr)

acyclic tso
```

Listing 2.2: .kat file for TSO

The main advantage of KATER is that it immensely simplifies the introduction of new memory models to GenMC. The user only has to specify the declarative version of the memory model in the *kat* language. Afterwards, the produced source code can be embedded into GenMC's core with a minimal effort.

Chapter 3

Persistent Memory

3.1 Basics of Persistent Memory

Up until now the memory-storage hierarchy consisted of two parts: (1) volatile memory with low latency and high cost at the top layers (CPU registers, caches, main memory) and (2) non-volatile storage (also known as secondary storage) with low cost and very high latency at the lower layers (SSDs, HDDs, etc).

While main memory is directly connected to the processor, secondary storage can only be accessed through the I/O bus and its interface, which is the main reason behind the very high latency associated with it.

Emerging new memory technologies [Lee09, Kawa12] have recently been used in order to bridge the gap between memory and storage. Persistent Memory (PM) refers to memory technologies that are directly byte-addressable, while their contents are non-volatile, which means that they remain preserved after a power crash or failure. Therefore, the data are accessible and can be used after the crash. Furthermore, PM can be placed on the memory bus and offers performance comparable to regular main memory (within an order of magnitude), while being relatively cheaper. A brief summary of the memory-storage hierarchy and where PM is placed between them can be seen in Figure 3.1.

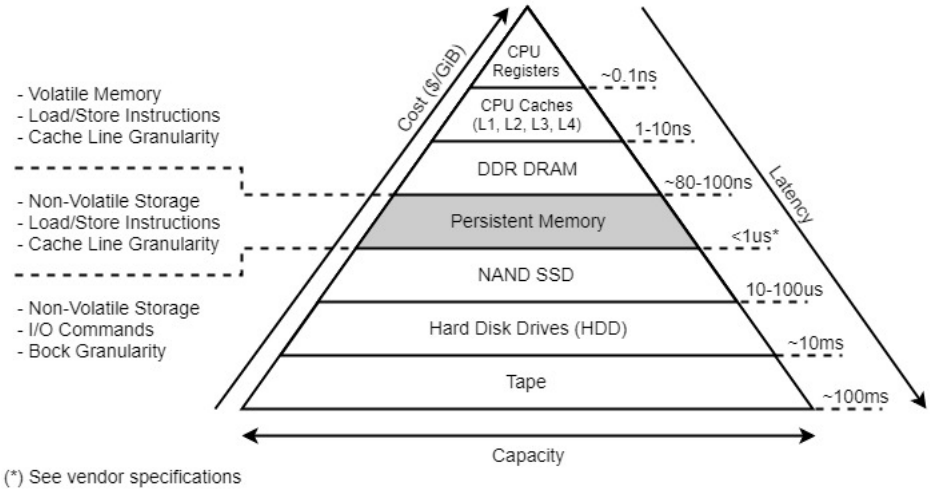


Figure 3.1: Memory-storage hierarchy with the newly introduced Persistent Memory layer

There are various NVM implementations available in the market, like Intel's Optane PM module [Inte19b], which can currently co-exist with (volatile) main memory or completely replace it. It is widely believed, however, that PM is going to replace DRAM in the future [Pell14], especially since DRAM technologies are reaching their limits as we approaching the end of Moore's Law [Kuma15].

It is evident that the non-volatile nature of PM opens up a lot of opportunities for programmers. In particular, it allows programmers to have direct access to data without the need

to rely on databases and file systems, techniques that introduce huge overheads. Additionally, the rise of big-data applications, data analytics, blockchain storage layers and extremely large neural networks require large in-memory databases, which can be accommodated by servers with larger memory capacities than what traditional DRAM can provide due to physical limitations. NVM technologies can potentially scale to these needed capacities, overcoming the limitations [Qure09].

Trying to write correct PM programs, however, is not easy, since PM imposes several complications on system architecture and software. The correctness of a PM program refers to ensuring consistency and durability in the same time. That means that programmers have to ensure a correct recovery after a crash, while maintaining the consistency of the data.

One of the main concerns about programming PM is that caches still remain volatile (and they most probably will remain volatile in the future). As a result, writes may not propagate to PM at the time they are issued. Instead, programmers should use explicit instructions that *flush* the contents of the caches to PM. However, this is sometimes not enough to ensure correctness! Explicit `flush` instructions are also subject to reordering by many CPUs. Furthermore, different cache lines may become persistent in a different order than the one they were written to the cache due to various cache replacement schemes (e.g., LRU). As a result, writes may become persistent in a completely different order than the program order. To deal with this, modern CPUs offer explicit *synchronization* fences that restrict the order in which writes become persistent.

3.2 Memory Persistency Models

The following example illustrates that even sequential programs can have unexpected results with respect to persistency.

Example 3.1: Minimal example of out-of-order persists

Consider the following simple sequential program. The variables `x` and `y` are both initialized with 0.

```
x := 1;
y := 1;
```
(WW)

Figure 3.2: Simple sequential program.

Although the write to `x` precedes the write to `y`, it is possible that the cache line of `y` persists before that of `x`. If a crash occurs after the persist of `y` and before the persist of `x`, then it is possible that during the recovery of the program, `y` is going contain the value 1, while `x` is still going to contain 0. It is important to note that the memory state $\texttt{x} = 0 \wedge \texttt{y} = 1$ cannot be observed under SC or TSO.

One important remark that comes from Example 3.1 is that a crash can potentially result in the memory reaching some states that are not reachable under any execution allowed by the underlining consistency model. Therefore, in order to write correct PM programs, it is necessary to first understand how the various memory states can be reached and what are their implications on data consistency after the crash. More precisely, the order in which writes are persisted to PM together with the program point that a crash happens define the memory state after the crash.

Similarly to memory consistency models, one can define memory persistency models [Pell14].

Memory persistency models describe the allowed behaviours with respect to failures. More concretely, persistency models define the order in which the effects of loads and stores are persisted in PM. This order is called *persistency memory order*. Note that this is analogous to the memory order defined by the memory consistency models, that is the order in which effects from memory instructions are made visible to other threads.

Strict versus Relaxed Persistency One common question that arises concerns the relation between consistency and persistency models. Does the persistency model enforce similar constraints to the underlying consistency model? Based on that question, persistency models fall under two categories:

Strict persistency couples the persistency order to the memory order imposed by the underlying consistency model. That is, stores are persisted to PM in the same order that they become visible to other threads. Examples of strict persistency models are Buffered Strict Persistency (BSP) [Josh15] and the recently proposed TSOPER [Ekem21] persistency model for TSO. Although strict models ease programmers from the burdens of persistency, they can seriously undermine performance.

As a result, there have been proposed various *relaxed* persistency models [Raad18, Raad19, Raad20, Khyz21]. Relaxed persistency decouples the persistency model from the underlying consistency model. The more the persistency order deviates from the order imposed by the consistency model, the more relaxed the persistency model is. Obviously, the goal of relaxing persistency models is to gain better performance.

Buffered versus Unbuffered Persistency Another way to improve performance of PM and utilize its abilities is persist buffering [Izra16]. Persist buffering offers the ability to avoid stalling when executing a persist instruction, i.e., persists occur *asynchronously*. On the other hand, an unbuffered persistency model stalls the execution in order to execute persist instruction *synchronously*.

3.2.1 Epoch Persistency

An example of a relaxed persistency model is *Epoch persistency* [Pell14]. There are two main persistency primitives supported by this model: `pfence` and `psync`.

The `pfence` instruction is used as a persist barrier, which divides the execution of each thread in different *epochs*. These persist barriers enforce an ordering between stores on the same thread and on different epochs: no stores after the barrier can be persisted before those prior to the barrier. Furthermore, stores to the same location are always ordered with respect to the program order. However, there is no persist ordering between stores in different locations within the same epoch.

Example 3.2: Using `pfence` under the epoch persistency model.

Let's revisit Example 3.1. This time we are going to add a `pfence` instruction between the two stores.

```
x := 1;
pfence;
y := 1;
```
(WPW)

Figure 3.3: Extending Example 3.1 with a `pfence` instruction. Epoch 1 is denoted by the blue box and epoch 2 is denoted by the red box

The use of the `pfence` divides the execution into two epochs. Epoch 1 contains the store to `x`, while epoch 2 contains the store to `y`. As such, the persist of `x` is ordered before the persist of `y` and therefore the memory state $\texttt{x} = 0 \wedge \texttt{y} = 1$ after a crash is not reachable.

There have been various extensions to the epoch persistency model; most importantly the addition of *persist buffering* [Izra16]. As described earlier, persist buffering leads to asynchronous persists. In order to tackle this issue, the second persistency primitive `psync` was proposed, which blocks the execution until all pending persists have been committed to PM.

The epoch persistency model offers better performance than strict persistency doest, since all stores to different locations within the same epoch can be executed concurrently. Since epoch persistency follows relaxed persistency semantics, it can be adapted to systems that support different consistency models. If we consider SC as the underlining consistency model, we can achieve strict persistency on an epoch persistency model by adding a persist barrier, which is a sequence of `pfence; psync`, after every load or store instruction. However, this method could seriously diminish performance. A downside of epoch persistency is that no known system architecture supports the `psync` instruction, which is also really costly in terms of performance, since it stalls the execution.

3.2.2 Persistent x86-TSO (Px86)

The first work that addressed the allowed memory behaviours with respect to persistency of the ubiquitous Intel-x86 architecture [Inte19a] was the **Px86** model [Raad20]. More precisely, the authors of this work extended the x86-TSO consistency model [Sewe10] with persistency semantics and introduced both an operational as well as a declarative persistency model for NVM in the x86-TSO architecture.

Px86 is a relaxed buffered persistency model. Following Example 3.1, x86-TSO does not allow the memory state $\texttt{x} = 0 \wedge \texttt{y} = 1$ under any normal execution. However, due to the relaxed persistency semantics of Px86, this memory state can be observed after a crash. Similarly to the epoch peristency model, stores to the same locations are expected to persist in the program order. What differentiates Px86 from the epoch persistency model, is that there is no notion of epochs: when two stores are on different locations, they can persist to PM in any order.

Intel x86 follows the TSO model with store buffers, as described in subsection 2.1.2. In order to model Px86, we need to add a *persistent buffer*, which sits between the thread buffers and PM. When stores are propagated from the store buffer of each thread, they enter the persistent buffer. At a future time (which is non-deterministic), the pending writes on the persistent buffer are propagated to PM. The whole storage subsystem of Px86 is shown in Figure 3.4.

Px86 offers explicit persist instructions that work on cache line width, which means that if a persist instruction is executed on cache line `x`, then all pending writes on locations $x \in X$ are going to be persisted as well. More specifically, these instructions are `flush` (namely `clflush`) and $\texttt{flush}_{\texttt{opt}}$ (namely `clflushopt`)[1]. It should be noted that these instruction take effect *asynchronously*, since Px86 follows a buffered persistency model.

The main difference between flush and $\text{flush}_{\text{opt}}$ has to do with the ordering with respects to writes. While flush is ordered with respect to both earlier and later writes to any location, $\texttt{flush}_{\texttt{opt}}$ is only ordered with respect to earlier writes on the same cache line. That means that $\texttt{flush}_{\texttt{opt}}$ can be reordered with later writes on different cache lines. Therefore, the pro-

[1]There is actually a third instruction supported by Intel's x86 ISA, which is `wb`. However, it has the same semantics as $\texttt{flush}_{\texttt{opt}}$ and therefore we choose to focus only on the latter.

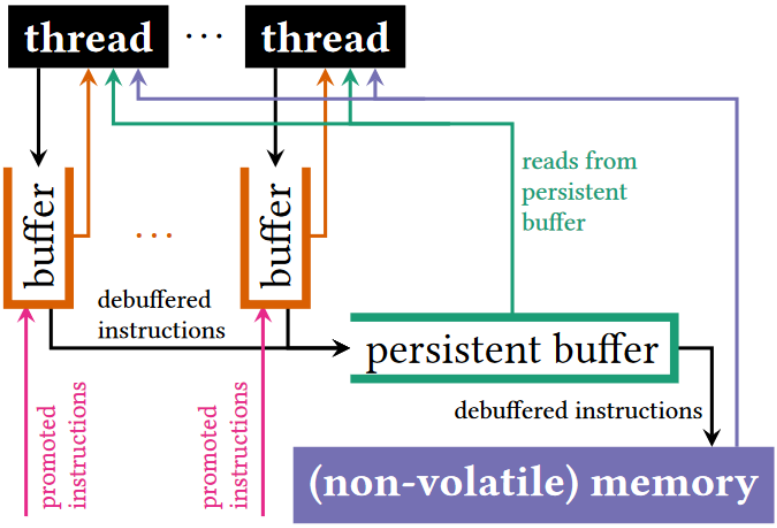


Figure 3.4: Px86 storage subsystem [Raad20].

grammer should be very careful when using this instruction. However, as the name suggests, $\texttt{flush}_{\texttt{opt}}$ offers greater performance than flush due to its more relaxed nature.

Intel's x86 offers not only the memory fence (`mfence`), but also a store fence (`sfence`) in order to enforce ordering constraints between writes and flushes. More precisely, an sfence instruction can be reordered with respect to reads, but cannot be reordered with writes and flushes. As such, any $\texttt{flush}_{\texttt{opt}}$ before an sfence is guaranteed to be ordered before any store after the sfence. It should be also noted that the mfence instruction is stronger than sfence, i.e. it enforces the same orderings constraints as sfence and also constraints with respect to reads.

Example 3.3: Difference between `flush` and $\texttt{flush}_{\texttt{opt}}$.

Let's consider Example 3.1, but we are going to add the explicit persist instructions supported by Intel's x86 architecture between the two stores. Assume that $\texttt{x} \in \texttt{X}$ and $\texttt{y} \notin \texttt{X}$.

```
x := 1;
flush X;          (WFW)
y := 1;
```

(a) Use of `flush`.

```
x := 1;
flushopt X;       (WFoW)
y := 1;
```

(b) Use of $\texttt{flush}_{\texttt{opt}}$.

Figure 3.5: Simple sequential programs showcasing the difference of the explicit persist instructions of Intel's x86.

Since `flush` instruction is ordered with respect to all writes on any location, the program in Figure 3.5a guarantees that the persist of the write on `x` is ordered with respect to the persist of the write to `y` and therefore the memory state $\texttt{x} = 0 \wedge \texttt{y} = 1$ is not allowed after a crash.

However, since $\texttt{flush}_{\texttt{opt}}$ is ordered only with respect to earlier writes on the same

location and because $y \notin X$, it can be reordered after the store on `y`. Therefore, the program in Figure 3.5b does not exclude the memory state $\texttt{x} = 0 \wedge \texttt{y} = 1$ after a crash. In order to avoid this behaviour, an sfence is needed after the $\texttt{flush}_{\texttt{opt}}$, as shown in the following figure. Since $\texttt{flush}_{\texttt{opt}}$ is ordered before sfence, then it is ordered before the store to `y`, too. Therefore, if `y` has persisted upon recovery, then $\texttt{flush}_{\texttt{opt}}$ has also executed before the crash and therefore `x` has persisted too.

```
x := 1;
flushopt X;
sfence;
y := 1;
```
(WFoSW)

Figure 3.6: Adding an `sfence;` instruction after $\texttt{flush}_{\texttt{opt}}$ enforces an ordering between the write on `x` and the write on `y`.

It should be noted, however, that these explicit persist instructions are really expensive and should be used with frugality. Writing correct PM programs can seriously undermine performance on various PM workloads [Wu20]. It is important, therefore, to minimize the necessary amount of flushes and fences that the application requires. In summary, the example in Figure 3.6 induces the constraint $\texttt{y} = 1 \Rightarrow \texttt{x} = 1$ upon recovery. This notion of constraints, i.e. if we observe that a certain variable has been persisted upon recovery then another variable must have also been persisted, is a very common practice when writing recovery procedures for PM programs.

Finally, the work of [Raad20] introduces two formal definitions of the Px86 model using (a) operational and (b) declarative semantics.

3.2.3 Refinements on Px86

Px86 seems to accurately model the persistency semantics of Intel-x86 architecture. However, it still fails to match the programmers' intuition that persist instructions happen *synchronously*. Most of previous work on PM systems [Pell14, Izra16, Zuri19, Frie18, Frie21a, Frie21b, Scar20] assumed the synchronous effect of flushes and sfences. For example, [Izra16] introduces a *psync* instruction, which blocks the program execution until all previous flushes have reached PM. Such an instruction cannot be implemented in Px86. Instead, Px86 *asynchronous* nature only restricts the order in which writes and flushes' effects have reached PM. Consequently, under Px86 an execution, in which no writes have reached PM and the state of PM is the same as the initial one, is always legal!

In order to bridge the gap between the formal semantics of Px86 and the programmers' understanding of PM, [Khyz21] presents a refined model of Px86. Their operational model $\textbf{PTSO}_{\textbf{syn}}$ offers *synchronous* persist instructions, which block the execution until certain writes reach PM. More precisely, the differences between Px86 and PTSO_{syn} are the following:

- **Per-cache-line persistence buffers**. Although such an idea cannot be implemented in real hardware for obvious reasons, it reflects better the idea that writes to the same location have to be persisted in program order, while writes to different locations can persist out of order.

- **Synchronous `flush` instructions**, which block the execution until the contents of the corresponding cache-line's persistence buffer are persisted.

- **Synchronous `sfence`** instruction, which blocks the execution until all $\texttt{flush}_{\texttt{opt}}$ of the same thread have taken their effect.

Note that all the effects of the *synchronous* instructions still take effect when they propagate from the store buffer and not when they are issued.

We note two important results from the work of Khyzha and Lahav [Khyz21]. The first one is the following theorem:

Theorem 3.1 ([Khyz21]). *$PTSO_{syn}$ and Px86 are observationally equivalent.*

Under *observational equivalence* two models are considered equivalent if the set of reachable states, while taking into account the possible crashes, is the same for both models. This equivalence notion actually implies that these two models can reach the same states after a crash, but they may do so by following different sequence of steps.

The second important result is that their formalization induces a partial order on persist events, rather than a total order, which is the case in Px86. Furthermore, they present an equivalent axiomatic model, namely $\text{DPTSO}_{\text{syn}}$. This representation opens up the opportunity of partial order reduction techniques tailored to persistency [Koko21a].

The refinements proposed by PTSO_{syn} were adopted by the authors of Px86 in the formalization of PEx86, which accounted also for non-temporal writes in Intel's x86 architecture.

3.3 Model Checking for Persistency

The declarative persistency models define a new derived relation called the *persist-before* (or *non-volatile-order*) relation, denoted as **pb**. Using this relation various axioms are added to model the persistency semantics (we refer to these as the persistency axioms of the model). This relation gives the order in which events are persisted. This relation is a strict total order in Px86, while in $\text{DPTSO}_{\text{syn}}$ is a partial order.

The first work that addressed model checking for persistency was done by extending GenMC in order to verify programs that interact with the `ext4` filesystem [Koko21a]. This model checking algorithm is called PerSeVerE. In order to model the checking of the persistent memory after a crash, PerSeVerE introduces the concept of a *recovery observer*, which is a thread that runs in parallel with the main program. This recovery observer consists only of reads (in the original implementation for `ext4` these reads are disk reads) and using the **rf** relation one can obtain the values that are going to be read during the recovery.

The verification process for PerSeVerE is extended as shown in Algorithm 3. A program P to test, as well a recovery observer P_R is given to the verification function. The VisitOne(P,G,Γ) function calculates a full execution (G) and updates the environment Γ[2], while checking that G is consistent with respect to the memory model. Afterwards, the RunRecovery(P_R,G,Γ) is executed and extends the calculated execution to a full execution of $\text{P}||\text{P}_\text{R}$, while checking whether this execution is consistent to the memory model, as well as to the persistency axioms of the model.

Since PerSeVerE is built upon GenMC, it is parametric to the memory model. Therefore, the ideas presented for the `ext4` filesystem can be adapted to support other declarative memory models expanded with their respective persistency axioms, like the refined Px86 model.

In order to reduce the search space, PerSeVerE requires that all variables that are read during the recovery routine are denoted *persistent*. These persistent variables should be declared globally and this is achieved by using `__VERIFIER_persistent_storage()`, which takes a variable as an argument. It also supports a `malloc`-like allocator for persistent variables, namely `__VERIFIER_palloc()`, which takes the size of the variable as an argument.

[2]The environment Γ contains information about alternative explorations that guide GenMC's algorithm.

Algorithm 3: PerSeVerE exploration algorithm

```
1: procedure VERIFY(P, P_R)
2:     ⟨G, Γ⟩ ← ⟨G_0, Γ_0⟩
3:     do
4:         VISITONE(P,G,Γ)
5:         RUNRECOVERY(P_R,G,Γ)
6:     while ⟨G, Γ⟩ ← Γ.pop()
7: end procedure
```

Note that the variable that is storing the result of `__VERIFIER_palloc()` should have been declared as persistent too.

At the time of the writing of this thesis, PerSeVerE supports three PM operations of the Px86 model:

1. a *persist initialization barrier*, with the use of the function `__VERIFIER_pbarrier()`. This function ensures that all previous (in program order) operations on persistent variables will be persisted in memory during recovery. However, it does not provide any guarantee for variables that are after it (in program order).

2. `clflush`, with the use of the function `__VERIFIER_clflush()`, which takes as input the address of a persistent variable `x` (`&x`) and simulates the flushing of this memory location.

3. a *recovery routine*, with the use of the function `__VERIFIER_recovery routine()`. This is the function that act as the recovery observer of the program in order to analyze the post crash state of the program. Inside the body of this function, one can insert assertions in order to verify the state of the persistent memory after the crash.

PerSeVerE does not currently support `clflushopt` or `sfence`, but as we mention in Chapter 7 this can be done as future work. However, as we already mentioned, the use of `clflush` is sufficient to enforce durability and is equivalent to the sequence of instructions `clflushopt; sfence`.

The corresponding `kat` file for the refined Px86 persistency model is shown in Listing 3.1. Note that the acyclicity condition remains the same as in TSO, while the **pb** relation is defined. The `recovery` keyword takes the **pb** relation as argument and checks its consistency during the exploration of the recovery routine.

```
1  let DW = [W]; [D]
2  let DR = [R]; [D]
3
4  save[ppo] ppo = ([R]; po | po ; [F] | [F] ; po  | po ; [W])+
5
6  let tso = (ppo | rfe | mo | fr)+
7
8  acyclic (tso)
9
10 let eco = (rf | mo | fr)+
11 let pb = ([DW]; mo) | ([DW]; po-loc; [CLF]; po? | [CLF]; tso; [DW] | [CLF|DW]; (
       po-loc | eco)+; [CLF|DW])
12
13 recovery (pb)
```

Listing 3.1: `.kat` file for Px86.

Chapter 4

Data Structures

4.1 Preliminaries

In a concurrent environment various threads can perform operations on a data structure that is shared between them. Each operation is associated with its *invocation* and its *response* events. In order to model executions in a concurrent environment, we need to define the *history* of an execution.

Definition 4.1 (History)**.** *A* history *is a finite sequence of the invocations and the responses of the operations performed by various threads in a concurrent environment.*

Execution histories can alternatively be seen as irreflexive partial orders $<_H$ of their events, i.e., $e_0 <_H e_1$ if the response of e_0 precedes the invocation of e_1 in H. This partial order corresponds to the actual ordering between operations. When two operations are not ordered by $<_H$, they are considered *concurrent*. When every invocation (except possibly the last one) of a history H is immediately followed by its corresponding response and the first event in H is an invocation, then H is called *sequential*.

Based on the above definitions, one can define the most common correctness criterion for concurrent data structures, which is *linearizability* [Herl90] :

Definition 4.2 (Linearizability)**.** *A history H is* linearizable *if it can be extended (by appending zero or more response events) to some history H' such that:*

1. *complete(H')*[1] *is equivalent to some legal sequential history S, and*
2. $<_H \subseteq <_S$ *(the partial order induced by H is a subset of the partial order induced by S).*

Property 2 actually means that if a method call m_0 precedes method call m_1 in H, then the same is true in the sequential history S. Essentially, linearizability means that operations appear to take effect atomically at some point between their invocation and response events.

The traditional approach for writing correct concurrent implementations of data structures was to use locks to protect shared variables. However, when a thread tries to acquire a lock held by another thread, then it would block until the lock is free. These implementations are generally called *blocking* implementations, since threads have to wait for the thread that is holding the lock. Obviously, this approach severely hurts performance, since only one thread makes progress, while the others are blocked. Aside from that, lock implementations are susceptible to deadlocks and livelocks and the whole implementation depends on the thread that is at any time the lock-holder, because a crash of this thread would lead to a state in which the threads waiting for the lock would not be able to ever acquire it.

This problems has led computer scientists to design *non-blocking* algorithms. Such algorithms, guarantee that the failure or the delay of a thread cannot lead to failure or delay of another thread. These designs abandon the use of locks and threads are not required to

[1] complete(H) is the maximal subsequence of H consisting only of invocations and matching responses

wait for another thread to complete. It should be noted that linearizable algorithms are also non-blocking [Herl90] and also more robust in case of unexpected events [Mich02b].

Based on the non-blocking property, we can define two main progress conditions for methods of concurrent data structures:

lock-freedom A method is *lock-free* if there is a guarantee that among some threads performing operations, at least one of them is going to complete in a finite number of steps. However, the rest of the threads might starve or loop indefinitely.

wait-freedom A method is *wait-free* if it guarantees that every operation is going to complete in a finite number of steps. It is easy to see that a lock implementation is not wait-free since a thread can take an unbounded number of attempts trying to acquire a lock. A wait-free algorithm is also lock-free, but not vice-versa.

4.2 Lock-Free Data Structures

In this section we are going to present some basic lock-free data structures. These data structures are going to be used as baselines in our experimental section.

4.2.1 Harris' Linked-List

Linked-lists are one the most fundamental data structures in computer science. Although their design is fairly simple, they play an important role in memory management and garbage collection and they are also used as an underlying infrastructure for more complicated data structures like hash tables and S-expressions. The most commonly used lock-free linked-list implementation the Harris's linked list [Harr01], which was later refined[Mich02a].

Harris' linked-list is an ordered list, containing nodes with a *key* field for the stored element and a *next* field, which points to the next node in the list. Additionally, one can add an extra *value* field in the node structure, which is useful for implementing hash tables based linked-lists. However, this additional field does not affect the operations of the linked-list.

As per common practice in lock-free data structures, Harris's linked-list uses the `CAS` primitive, in order to atomically change the *next* pointer of a node. For instance, when we want to insert a new node in the list, we set the *next* field of the new node to the desired node (the first one with a key greater than the inserting node) and we use a `CAS` to change the *next* pointer of the previous node to the new node. Similarly, when we want delete a node, we use `CAS` to swing the *next* field of the previous node to point to the next node.

However, this is not a linearizable implementation. For example, the linked-list pictured in Figure 4.1, containing the *head* node, a node with key 1 (n_1) and a node with key 3 (n_3), and two threads issue the following operations:

- Thread 1 inserts a node with key 2 (let's say n_2).
- Thread 2 deletes n_1.

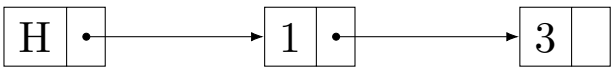


Figure 4.1: Initial state of linked-list.

Let's assume that thread 2 has found the *next* pointer of *head*, which is n_3. Right before it performs the `CAS` to swing the *next* pointer, thread 1 sets the *next* field of n_2 to n_3 and performs the `CAS` to change the *next* field of n_1 to itself. As a result, if thread 1 proceeds

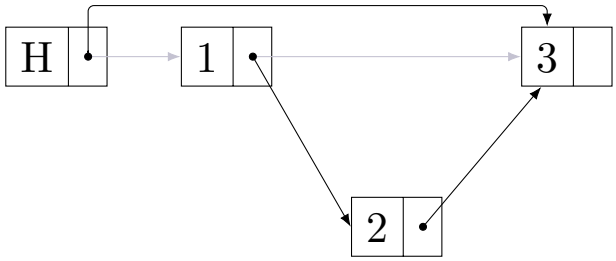


Figure 4.2: State of linked-list after concurrent insert and delete.

to perform the `CAS` instruction, then n_2 would not be accessible starting from *head* since it's predecessor (n_1) is no longer in the list. The state of the linked-list is shown in Figure 4.2.

In order to tackle the above problem, Harris proposed a *marking* mechanism, which uses two `CAS` instruction instead of one. More concretely, when we want to delete a node from the list, we first *mark* the *next* field of the node, which equals to a logical deletion. Logically deleted nodes can remain in the list, but they prohibit threads from inserting a node immediately after them. After a node has been logically deleted, the `CAS` to swing the *next* field of the previous node can safely be performed.

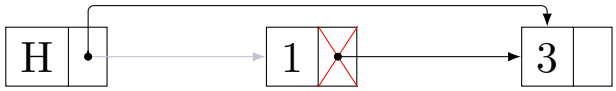


Figure 4.3: Marking the *next* field of the deleted node.

On the previous example, thread 1 would observe that n_1 is logically deleted and therefore n_2 cannot be inserted after it. So, the question is, what happens next? If thread 1 restarts it would fall into the same problem. The solution is to physically delete the logically deleted node n_1 before restarting. That way, thread 1 would actually make progress on the state of the linked-list before restarting.

It should be noted that the *marking* of a logically deleted node can be performed really easily by setting the last bit of the *next* field of the node. Assuming a modern machine that operates on words wider than 1 byte, the marking of the pointer would not affect its actual address. However, modern programming languages implementations can support objects that contain both an object of some type as well as a boolean mark and both fields can be updated atomically at the same time (e.g. Java's AtomicMarkableReference object [Herl08]).

Harris linked-list implementation is based on a *search* function, which given a search key returns two nodes: (a) a node with the largest key less than the search key (*pred*) (b) the node with least key greater than (or equal) the search key (*curr*). However, these nodes should be *unmarked*. This is achieved by removing all marked nodes between *pred* and *curr*. A `CAS` instruction tries to set *pred*'s *next* field to point to *curr* and if that check fails, it restarts the *search* function. Note that later refinements on Harris linked-list [Mich02a, Herl08] utilize a more optimized *search* function that does not traverse all the intermediate nodes between *pred* and *curr*, but uses a window that traverses through the list. If the right side of the window finds a logically deleted note, it tries to physically delete it using a `CAS` on the left's side *next*. If that `CAS` fails, which means that either the *next* reference has changed (the right side has been physically remove) or it has been marked (which means that the left side has been logically remove), it restarts the *search* function.

Based on that *search* function it is easy to implement the three main functionalities of Harris linked-list :

- **insert**, which adds a node to the list. Initially, *search* is performed with the key that is being inserted to find *pred* and *curr*. Then a `CAS` is performed to set *pred*'s *next* field

to the node being inserted. If the `CAS` fails (for the reasons we stated earlier), then the *insert* procedure starts over. This method is **lock-free**.

- **remove**, which deletes a node from the list. Initially, *search* is performed with the key that is being deleted to find *pred* and *curr*. Afterwards, *curr* is logically deleted by performing a `CAS` to mark it's *next* field. Finally, one can attempt to physically remove the node too, but even if that does not succeed, the node will be physically removed by some future operation. This method is **lock-free**.

- **contains**, which searches whether a certain key exists in the list. After the *search* for the given key is performed, it checks whether it is equal to *curr*'s key. In that case it returns true, otherwise it returns false. This method is **wait-free**.

4.2.2 MS-Queue

Queues are widely used data structures, which are mainly utilized as FIFO buffers or in BFS algorithms. We are going to focus on the most common lock-free queue implementation, which is the Michael-Scott(MS) queue [Mich96].

Most queue implementations consist of two pointers *head* and *tail*, which are the return and the entry point of the queue respectively. MS-Queue is implemented as a linked-list coupled with these *head* and *tail* pointers. The *head* node points to a sentinel/dummy node, which helps to deal with the edge case of an empty list. Initially, *tail* also points to the sentinel node. An example of an instance of a queue is given in Figure 4.4.

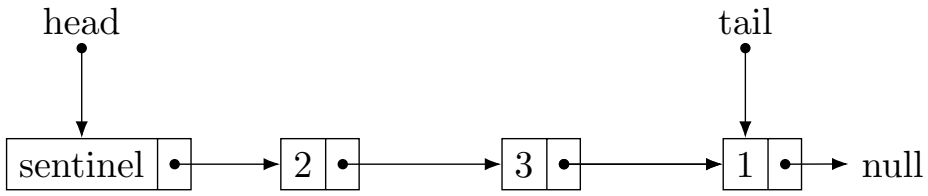


Figure 4.4: Example of an MS-Queue.

MS-Queue implements the two typical queue operations, namely **enqueue**, which inserts a node at the end of the queue, and **dequeue** which returns the head of the list. Following we are going to explain how these operations are implemented in order to be lock-free.

The enqueue method initially creates a new node and then it loops trying to insert the node. More precisely, it reads the *tail*'s *next* pointer. If this pointer is not NULL, it means that some previous operation has not yet completed and therefore the *tail* should be fixed to point to the actual last node. In that case, a `CAS` attempt is made to swing the *tail* and then (regardless the result of the `CAS`) the enqueue method is restarted. If the pointer is NULL, which means that it is currently the last node, then a `CAS` is issued in order to append the inserting node as it's successor. If the `CAS` fails, then the enqueue method is restarted. If it is successfull, then the inserting node has successfully been added and another `CAS` is issued in order to fix the *tail* to point to this node. However, the return status of this `CAS` does not force a restart of the enqueue operation.

The dequeue method tries to swing the head to point to sentinel's *next* node. If this `CAS` succeeds, then it returns the value of the sentinel node and makes it's *next* node the new sentinel. Otherwise, the method is restarted.

There is, however, an edge case for the dequeue operation, which is delineated in Example 4.1. In order to deal with this edge case, a check is needed to determine whether the queue is empty. If that is the case and the sentinel node has a non NULL *next* field, it means that *tail* should be moved. As before, a `CAS` is issued to swing the *tail* and in case it fails, the dequeue operation starts over again.

Example 4.1: Edge case of dequeuing a node concurrently with an enqueue

Consider an instance of an empty queue, i.e. both the *head* and *tail* are pointing to the sentinel node as shown in Figure 4.5.

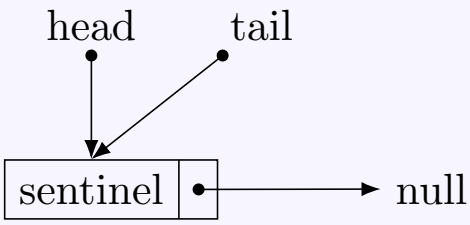


Figure 4.5: Empty queue. Both *head* and *tail* point to the sentinel node.

Let's assume that thread 1 attempts to enqueue a node, while threads 2 attempts to dequeue a node. Thread 1 has successfully swinged the sentinel's node *next* field (since *tail* is pointing to it) to the inserting node. However, *tail* has not yet been fixed. In that case, thread 2 is going to swing *head* to point to the newly inserted node, as it is going to be new sentinel. The problem is that *tail* is still refering to initial sentinel node, which leads to an inconsistent state as shown in Figure 4.6.

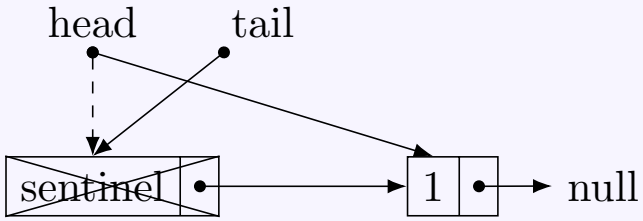


Figure 4.6: Dequeue and enqueue concurrently on an empty queue leads to an inconsistent state.

Both of these operations are **lock-free**. Another operation that can be added to the MS-Queue is one to return the size of the queue. It's implementation is straight forward and it is easy to prove that can be done in a wait-free way.

Finally, MS-Queue is prone to the ABA problem [Mich96], which is solved by using counting references. However, in the scope of this thesis we are not going to deal with this issue.

4.2.3 Skiplist

Linked-lists offer $\mathcal{O}(N)$ time complexity for the various operations, since in the worst case a full traversal of the list is required. Since we are dealing with ordered linked-list it would be really beneficial if we could use some form of binary search on the keys of the list. Such an approach would lower the average time complexity to $\mathcal{O}(\log N)$. However, single linked-lists offer only one point to access the list: the *head* of the list.

The data structure that tackles this issue is the skiplist. A skiplist is a collection of linked-lists, each characterized by its level. The bottom-level (level 0) list contains all the nodes, while the lists at a higher level are sublists of the lower level and serve as shortcuts to the lower level. The top level of each node of the bottom list is calculated randomly. Therefore, the skiplist is a probabilistic data structure. An example of a skiplist is shown in Figure 4.7. There are two nodes that server as sentinels, indicating the start (`head`) and the end (`tail`) of the skiplist. These nodes have always the maximum amount of levels allowed in the skiplist.

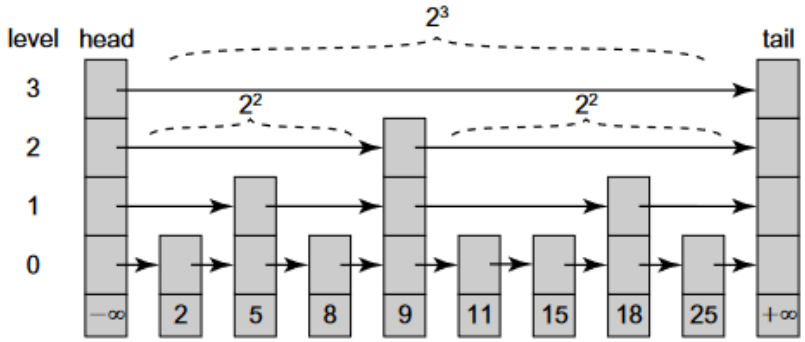


Figure 4.7: Example of a skiplist [Herl08].

The search method starts from the highest level of the `head` node and traverses the list at this level. When it finds a successor node that its key is greater (or equal) to the search key, it continues to the lower level of its current node. However, in order for the **insert** and **remove** methods to work properly, one must save the whole path, i.e. the links that were followed at each level, and update these links accordingly.

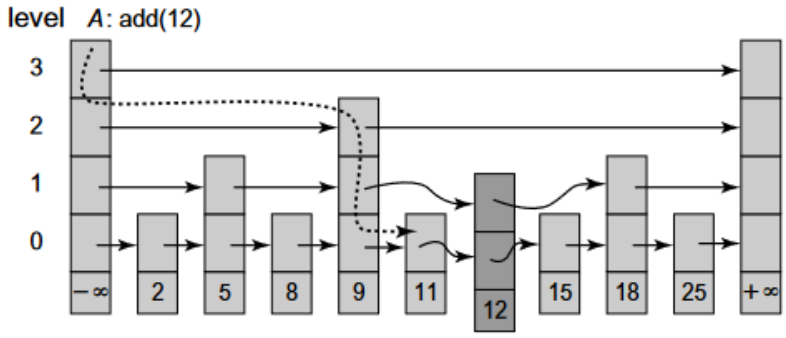


Figure 4.8: Inserting a node to a skiplist [Herl08].

The design of a lock-free skiplist is similar to the one of the Harris' linked-list. It utilizes the marking mechanism in order to remove nodes. It should be noted that each sublist above the bottom level might not be contained in the lower level due to marking. Similarly to Harris' linked list, the **insert** and **remove** functions are **lock-free**, while the **contains** function is **wait-free** [Mich02b, Herl08].

The main drawback of the skiplist is that it requires $\mathcal{O}(N \log N)$ space complexity instead of $\mathcal{O}(N)$.

4.3 Durable Data Structures

4.3.1 Durable Linearizability

Linearizability cannot be directly used as a correctness criterion for the persistent domain, since it does not take into account the effects of crashes and recovery procedures. In order to tackle this issue, a refined correctness criterion is established, namely *durable linearizability* [Izra16]. Since the formal definition of *durable linearizability* requires a lot of other definitions, abstractions and extensions from the *linearizability* definition, we are going to use a more informal definition of it [Frie18]:

Definition 4.3 (Durable Linearizability)**.** *An object is* durably linearizable *if a crash and recovery that follows linearizable history H, leaves the object in a state reflecting a consistent cut H'*[2] *of H such that*

1. *the subhistory of H' without the crashes is linearizable*
2. *every complete operation of H appears in H'.*

Intuitively, *durable linearizability* requires that after a crash all previously completed operations should have been persisted and their effects should be detectable in persistent memory. Alternatively, one can characterize a data structure as *durably linearizable* if by removing all crash events, all executions that remain are *linearizable*.

Based on the definition of *durable linearizability*, it is easy to show that the lock-free data structures mentioned in Section 4.2 (without making any adjustment related to persistency) are not *durably linearizable*. A minimal example that showcases a *durable linearizability* violation on a linked-list is shown in Example 4.2 [Wang19].

Example 4.2: Violation of durable linearizability on a linked-list

Consider a lock-free linked-list which contains two nodes, one with key 0 (node n_0) and one with key 3 (node n_3), as shown in Figure 4.9. Let us suppose that two threads ($T1$ and $T2$) want to insert nodes with keys 1 (node n_1) and 2 (node n_2) respectively.

Figure 4.9: Initial list containing n_0 and n_3.

Let's say that $T1$ performs a `CAS` and changes the next pointer of n_0 to point to n_1 and pauses without persisting this change. Afterwards, $T2$ starts executing and performs a `CAS`, which changes n_1 next pointer to point to n_2. Finally, $T2$ flushes its changes to persistent memory and completes the insertion of n_2. The state of the linked list is shown in Figure 4.10. Right after it finishes, a crash occurs.

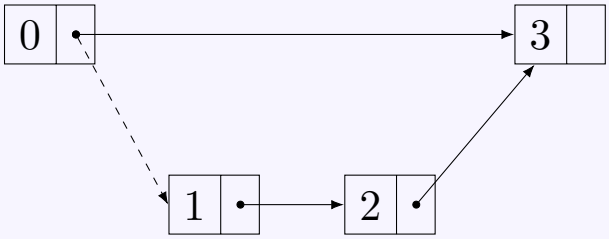


Figure 4.10: State of the linked-list right before the crash. Solid lines have persisted, dashed line has not persisted.

After the crash both n_1 and n_2 are lost, since the pointer from n_0 to n_1 has not been persisted before the crash. So, even though the operation of $T2$ (insertion of n_2) has completed prior to the crash, its effect cannot be detected during the recovery. This is a violation of *durable linearizability*. Note, however, that if $T2$ had been able to flush the next pointer of n_0 after it reads its value (and obviously before it performs the actual insertion of n_2), then this example would actually be *durably linearizable*.
Similarly, one can easily construct an example in which two concurrent deletes lead to a *durable linearizability* violation.

[2] A consistent cut H' a history H is a subhistory of H such that, if m_1 is contained in it, and $m_0 <_H m_1$, then m_0 is also in H'.

It is, therefore, evident that we need to use PM primitives, like `flush`, `flushopt` and `sfence`, in a precise way in order to turn these lock-free data structures into *durable linearizable* ones. We are going to focus on 2 implementations of *durably linearizable* data structures that have been recently proposed.

4.3.2 Persistent Queue

It can be easily shown that the MS-Queue is not durably linearizable, by constructing an example similar to Example 4.2. Some durable alternatives to MS-Queue have recently been proposed, which all fall under the name of *Persistent Queue* [Frie18]. We are going to analyze the *Durable Queue* implementation, since it is the least complicated among them and is sufficient to ensure durable linearizability.

The enqueue method of the Durable Queue creates a node with the given value and flushes it to PM. This is essential, because it ensures that the content of the new node has been persisted before being added to the list. Afterwards, the enqueue methods flushes the *next* pointer of the previous last node to the newly added node. If, however, another enqueue happens simultaneously and this flush has not been completed, the second enqueuer must take extra care by flushing this pending pointer.

For the dequeue operation, the node structure is modified. More precisely, a *deqThreadID* field is added to each node, which points to the last thread that performed a dequeue operation in this node. Furthermore, the queue is equipped with an array, that each dequeuing thread stores its returned value. Using a `CAS` the dequeuing thread tries to write its thread ID to the *deqThreadID* field of the next node of `head`. Right after this operation, the *deqThreadId* field of the node is flushed to PM. If the `CAS` was successful, then the value of the node is stored to the return array, flushes the array position and updates the `head` pointer (without flushing it). If it was not successful, the dequeuer starts over.

It should be noted that neither the `head` nor the `tail` pointer is ever flushed during normal execution. This happens because they can be fixed during recovery. The `head` pointer is set to the first node with a non-NULL *deqThreadID* value, while the `tail` is set to the last reachable (non-NULL) node starting from the `head`.

4.3.3 NVTraverse

The Durable Queue described in the previous section was designed with manual insertion of flushes and fences. Such an approach is not always feasible, since it requires extremely deep understanding of the data structure. Programmers need a more automatic way to write correct durable data structures implementations.

The first such transformation for non-blocking data structures was given in the seminal work for durable linearizability [Izra16]. This transformation assumes the epoch persistency model and requires that:

- after every store a write back (`pwb`) to persistent memory is issued
- a `pfence` is issued before a release store[3] and a `pwb` right after a load acquire
- a `pfence` before every `CAS` and a sequence of `pwb;pfence` after the `CAS`
- a `psync` before the operation returns

This transformation is generally referred to as the *Izraelevitz* transformation and as we can see requires a `pwb` instruction after almost every operation. Therefore, it is a very costly transformation that can seriously undermine performance of persistent memory.

[3]We refer to https://en.cppreference.com/w/cpp/atomic/memory_order for release-acquire semantics.

A more sophisticated approach for an automatic transformation is *NVTraverse*. NVTraverse [Frie21a] is an automatic transformation that turns a specific family of lock-free linearizable data structures into durably linearizable ones. More specifically, this transformation works on data structures that satisfy the following conditions:

- They are linearizable and lock-free.
- Their core should be a tree-like structure. There might be additional entry points to the tree core (e.g. the higher level of a skiplist).
- Every operation in the data structure follows the layout of Algorithm 4. The *findEntry()* function has two arguments: (a) the *root*, which is the main entry point to the data structure and (b) *input*, which is usually the key of the inserting node. The output of this function is a selected entry point to the core of the data structure. The *traverse()* function traverses the data structure from this selected entry point and returns a list of nodes (e.g. possible predecessor and successor nodes in a linked-list) that will be required for the *critical* method, which modifies the data structure.
- The *traverse* method should not modify shared memory and returns a suffix of the path traverse. The nodes of this suffix are called traverse nodes.
- Logical removal of nodes (marking) before the physical deletion.

Algorithm 4: Operation in a traversal data structure

```
T Operation(Node root, T input):
    while true do
        Node entry = findEntry(root, input);
        List<Node> nodes = traverse(entry, input);
        bool restart, T val = critical(nodes, input);
        if !restart then
            return val;
        end
    end
```

This family of data structures are referred to as *traversal data structures*. To turn an operation of a traversal data structure into a durable one, one should follow the layout of Algorithm 5. There are 2 new functions (which are shaded in the algorithm) that are executed after the traverse method:

- *ensureReachable*, which flushes the current parent (predecessor) pointer of the first node of the list that was returned with *traverse*. Recall from Example 4.2 that if we were able to flush the next field n_0 before we inserted n_2, then the implementation would be durable linearizable. The *traverse* function would return 3 nodes, n_1, n_3 (predeccessor and successor of n_2) and n_0 as the head of the list of nodes and the *ensureReachable* function would flush n_0 (and subsequently its *next* pointer) essentially ensuring that the inserted node would be reachable upon insertion.
- *makePersistent*, which flushes all the nodes returned by the *traverse* method and then executes a fence. That way all nodes read during the critical method (plus the parent pointer) have been persisted.

Finally, during the critical method, the NVTraverse transformation requires that a flush should be injected after every a read of a shared variable and after a write or `CAS`. Also, a fence is required before every write or `CAS` on a shared variable and before the return statement of the operation.

Algorithm 5: Operation in a NVTraverse data structure

```
T Operation(Node root, T input):
    while true do
        Node entry = findEntry(root, input);
        List<Node> nodes = traverse(entry, input);
        ensureReachable(nodes.head());
        makePersistent(nodes);
        bool restart, T val = critical(nodes, input);
        if !restart then
            return val;
        end
    end
```

In conclusion, NVTraverse utilizes the structure of the traversal data structures in order to insert far less flushes and fences than the Izraelevitz transformation. As a result, it seriously outperforms the latter.

Chapter 5

Test Suite

The main contribution of this thesis is a test suite for PM programs under the Px86 model. This suite can be divided into two categories:

1. **Litmus tests**. Litmus tests are synthetic, small and lightweight concurrent programs, that aim to test the tool against certain behaviours of the implemented memory model.

2. **Data structure tests**. Carefully designed tests on original and durable versions of some lock-free data structures found in the literature.

One goal of this test suite is to use it try to check the correctness and scalability of the PerSeVerE model checker. As we will soon see, through this test suite we were able to point out an internal inability of PerSeVerE to support the Px86 memory model. A more general goal is to use it as a basis for creating benchmarks for tools that test and verify PM programs.

5.1 Litmus Tests

In Chapter 3, we discussed some of the problematic behaviours of the Px86 model. For example, we showed that in the WW example it is possible to observe a post-crash state that it is not reachable under the x86-TSO model. In order to eschew this behaviour in Px86, we need to *flush* the variable `x` by issuing a `clflush(&x)` right before the write to variable `y`, as shown in the WFW example.

In order to model the WW example in PerSeVerE, we declare two persistent variables `x` and `y` with `__VERIFIER_persistent_storage()`, initialize them to 0, and issue a `__VERIFIER_pbarrier()` to create the initial state of the PM. Afterwards, we proceed to the two writes to variables `x` and `y`. In order to deal with the post crash states, we read the values of interest after a possible crash in the `__VERIFIER_recovery_routine()` function and issue an assertion on the conjunction of the two variables, i.e., `assert(!(x == 0) && (y == 1))`. In case this assertion fails, we have reached the undesired state $\texttt{x} = 0 \wedge \texttt{y} = 1$ and PerSeVerE will produce an output with this error detected. The corresponding `C++` code is shown in Listing 5.1, while the output of this example is shown in Figure 5.1.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <atomic>
4  #include <pthread.h>
5  #include <assert.h>
6  #include <genmc.h>
7
8  #define relaxed std::memory_order_relaxed
9
10 __VERIFIER_persistent_storage(std::atomic_int x);
11 __VERIFIER_persistent_storage(std::atomic_int y);
12
13 extern ''C''{
```

```
 void __VERIFIER_clflush(void*);
}

void __VERIFIER_recovery_routine(void)
{
  assert(!(x.load(relaxed) == 0 && y.load(relaxed) == 1));
  return;
}

int main()
{
  x.store(0, relaxed);
  y.store(0, relaxed);

  __VERIFIER_pbarrier();

  x.store(1, relaxed);
  y.store(1, relaxed);

  return 0;
}
```

Listing 5.1: `WW` litmus test

```
Error detected: Recovery error!
Event (1, 2) in graph:
<-1, 0> main:
        (0, 1): Wrlx (x...__a_, 0) atomic:949
        (0, 2): Wrlx (y...__a_, 0) atomic:949
        (0, 3): PERSISTENCY_BARRIERrel L.31
        (0, 4): Wrlx (x...__a_, 1) atomic:949
        (0, 5): Wrlx (y...__a_, 1) atomic:949
        (0, 6): THREAD_END
<-1, 1> __VERIFIER_recovery_routine:
        (1, 1): Rrlx (x...__a_, 0) [(0, 1)] atomic:957
        (1, 2): Rrlx (y...__a_, 1) [(0, 5)] atomic:957
Coherence:
y...__a_: [ (0, 2) (0, 5) ]
x...__a_: [ (0, 1) (0, 4) ]

Assertion violation: !(x.load(relaxed) == 0 && y.load(relaxed) == 1)
Number of complete executions explored: 2
Total wall-clock time: 0.04s
```

Figure 5.1: Output of PerSeVerE (assertion violation) for the erroneous `WW` litmus test. The output consists of the type of error, the execution trace, the error message, the number of explored (and blocked) executions and the time that was required.

In a similar way, we can model the WFW example by adding a `__VERIFIER_clflush(&x)` instruction between the two writes. This time the tool does not detect any errors as shown in Figure 5.2. As we can see, the purpose of litmus tests is to test the underlying memory model, with carefully designed tests that expose tricky behaviours.

Except from testing the Px86 persistency model, we were able to find out through these litmus tests that PerSeVerE required some additional internal support in order to implement the Px86 model. More precisely, we implemented the simple message passing example, shown

```
No errors were detected.
Number of complete executions explored: 2
Total wall-clock time: 0.04s
```

Figure 5.2: Output of PerSeVerE for the `WFW` litmus test. No errors were detected.

in Figure 5.3, which revealed that PerSeVerE was not able to correctly model the flushing of a variable that was previously written by another thread.

$$\begin{array}{l|l} \texttt{x} := 42; & \texttt{a} := \texttt{y}; \\ \texttt{y} := 7; & \texttt{if}(\texttt{a} \neq 0); \\ & \quad \texttt{flush x}; \\ & \quad \texttt{z} := 1; \end{array} \qquad \text{(2W+RFW)}$$

Figure 5.3: Message passing litmus test (`2W+RFW`). Variable `x` is written by the first thread and it is flushed by the second thread. This behaviour was initially problematic for PerSeVerE. The `C++` implementation is straightforward (see Appendix B).

The litmus test suite contains some more examples, that are more complex and utilize other consistency primitives like CAS. One such example ([Vafe22]) is shown in Figure 5.4.

$$\begin{array}{l|l} \texttt{a}_1 := \texttt{CAS}(\texttt{l}_\texttt{x}, 0, 1); & \texttt{a}_2 := \texttt{CAS}(\texttt{l}_\texttt{x}, 0, 2); \\ \texttt{if } (\texttt{a}_1 = 1) & \texttt{if } (\texttt{a}_2 = 1) \\ \quad \texttt{x} := 1; & \quad \texttt{b} := \texttt{y}; \\ \quad \texttt{y} := 1; & \quad \texttt{if } (\texttt{b} = 1) \\ \quad \texttt{flush x}; & \quad\quad \texttt{z} := 1; \\ \quad \texttt{flush y}; & \quad\quad \texttt{flush z}; \\ \quad \texttt{l}_\texttt{x} := 0; & \quad \texttt{l}_\texttt{x} := 0; \end{array} \qquad \text{(CAS+CAS)}$$

Figure 5.4: CAS-based locking litmus test (`CAS+CAS`). `CAS` is used as a lock: the two threads are battling to obtain the lock ($\texttt{l}_\texttt{x}$) and when they do they proceed to the body of their `if` statements. The invariant that we are testing is $\texttt{z} = 1 \Rightarrow \texttt{x} = 1 \wedge \texttt{y} = 1$. In order for `z` to obtain the value 1, `y` should also be 1. This also indicates that thread 1 should have been executed before thread 2 and should have finished, i.e., $\texttt{l}_\texttt{x}$ should have been set to 0 again. Therefore, the two writes on `x` and `y` should have taken place, as well as the `flush`es on both of them. As a result, `x` and `y` must have been persisted in memory before the write and the `flush` on `z` takes place.

As an additional observation, we model two kinds of checks for the recovery routine:

1. Assertions that should hold under every execution. We denote these test cases as safe, meaning that PerSeVerE should detect no errors when it executes these test cases. For example, the `WFW` and `CAS+CAS` test cases fall under this category.

2. Assertions that catch specific behaviours of the Px86 model and should fail. We denote these test cases as unsafe, meaning that PerSeVerE should detect a bug when it executes them. `WW` is an example of this category.

5.2 Data Structure Tests

As we already mentioned in Chapter 3, all variables that are read during the recovery routine of PerSeVerE should be declared globally as persistent. This is problematic for the data structures' implementations, since they are using nodes that are created during runtime. In order to deal with this problem, we declare a large persistent global array of nodes, and we populate this array with the use of `__VERIFIER_palloc()` during the initialization of the data structure. When a new node would be required with the use of `new` or `malloc` in the original implementation, we return one of the nodes stored in this global array. Listing 5.2 shows the corresponding code for the node allocation procedure, which is common for all data structures (except the implementation of the node itself).

```
__VERIFIER_persistent_storage(Node* nodes[MAXNODES]);

static int node_idx;

void allocateNodes()
{
  node_idx = 0;
  for (int i = 0; i < MAXNODES; i++) {
    nodes[i] = (Node *)__VERIFIER_palloc(sizeof(Node));
    new (nodes[i]) Node();
  }
}

Node* getNewNode()
{
  return nodes[node_idx++];
}
```

Listing 5.2: Code snippet for node allocation.

5.2.1 NVTraverse Tests

As far as NVTraverse is concerned, we tested the linked-list and skiplist implementations. More specifically, we used three different versions of these data structures:

1. The original implementation of the data structure (e.g., Harris' linked-list), which is non-durably linearizable.
2. Izraelevitz's transformation.
3. NVTraverse's transformation.

Our tests were focused around two axes:

1. Prove that the original implementations are in fact not durably linearizable, while the other two implementations pass the corresponding tests. Therefore, test cases for the original implementations are denoted as unsafe, while those for the two transformations are denoted as safe. Note that this, however, does not prove that the Izraelevitz's and NVTraverse's transformations are durably linearizable. The quality of the test cases and the coverage of the behaviours that these test cases achieve provide better assurances that these implementations might actually be correct.
2. Create some complex test cases in order to stress-test PerSeVerE.

An example of the first category is given in Listing 5.3, where we spawn two threads that perform an insertion and a deletion in the list. The recovery routine checks whether the list still contains the last node that was inserted before the persistency barrier. Running this test

with the original implementation, produces an assertion violation, which means that the last node was removed from the list, despite the fact that none of the two threads operated on it. Therefore, the list is in an inconsistent state after the crash. However, both the Izraelevitz and the NVTraverse implementations pass this test successfully.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <assert.h>

#include "../ListOriginal.h"

static pthread_t threads[2];
static int param[2] = {0, 1};

__VERIFIER_persistent_storage(static ListOriginal* list);

void *thread1(void *param)
{
  list->remove(3);
  return NULL;
}

void *thread2(void *param)
{
  list->insert(2, 10);
  return NULL;
}

void __VERIFIER_recovery_routine(void)
{
  assert(list->contains(4));
  return;
}

int main()
{
  list = (ListOriginal*)__VERIFIER_palloc(sizeof(ListOriginal));
  new (list) ListOriginal();

  list->insert(0,10);
  list->insert(3,10);
  list->insert(4,10);

  __VERIFIER_pbarrier();

  pthread_create(&threads[0], NULL, thread1, &param[0]);
  pthread_create(&threads[1], NULL, thread2, &param[1]);

  pthread_join(threads[0], NULL);
  pthread_join(threads[1], NULL);

  return 0;
}
```

Listing 5.3: `lor-pw+w+d` test.

For the second category, we constructed a test case with three threads. One thread performs a deletion, while the others perform one insertion each. Again, we check whether the list stays in an consistent state after the crash. The original implementation fails this test, while the two transformations pass it. Although it might seem that three threads with one operation each is not a very complex program, the combination of concurrency and the number of post-crash observable states produce a huge amount of executions ($> 40k$) and

can be used to stress-test the tool. This test case is shown in Appendix B.

As far as the notation of the test cases is concerned, we use the following regular expression:

$$((\texttt{sl})|\texttt{l})((\texttt{or})|(\texttt{iz})|(\texttt{tr})) - (((\texttt{pw})(+(\texttt{w}|\texttt{d})^+)^+)|((\texttt{w}|\texttt{d})^+(+(\texttt{w}|\texttt{d})^+)^*))$$

where

- `l` refers to linked-list, while `sl` refers to a skiplist test case
- `or` refers to the original implementation of the data structures, `iz` refers to the Izraelevitz transformation and `tr` refers to the NVTraverse implementation
- `pw` means that there are some writes (inserts) to the data structure before the persistency barrier.
- $+(\texttt{w}|\texttt{d})^+$ means that an additional thread is spawned, that performs an arbitrary number of inserts or deletions to the data structure.

For example, the program shown in Listing 5.3 is named `lor-pw+w+d`, because we are testing the original implementation of the linked-list with some initial inserts to the list before the persistency barrier and two threads are spawned, where one performs an insert, while the other performs a deletion.

The tests for the skiplist implementations are similar to those for the linked-list. We used three levels for the skiplist, in order to constrain the complexity of the tests.

5.2.2 Persistent Queue Tests

We followed a similar approach for the tests targeting the Durable Queue implementation. Our notation follows the same pattern, described by the regular expression:

$$((\texttt{msq})|(\texttt{dq})) - (((\texttt{pe})(+(\texttt{e}|\texttt{d})^+)^+)|((\texttt{e}|\texttt{d})^+(+(\texttt{e}|\texttt{d})^+)^*))$$

where

- `msq` refers to the original MS-Queue implementation, while `dq` refers to the Durable Queue one.
- `pe` means that there are some enqueues to the data structure before the persistency barrier.
- $+(\texttt{e}|\texttt{d})^+$ means that an additional thread is spawned, that performs an arbitrary number of enqueues or dequeues to the data structure.

The MS-Queue tests are not durably linearizable and are designed to fail (i.e., we denote them as unsafe). On the other side, the Durable Queue tests are in fact durably linearizable. The corresponding tests are denoted as safe and aim to test specific combinations of enqueues and dequeues.

The main difference with the NVTraverse tests is that the original implementations of the Persistent Queue (MS and Durable) did not provide any support for checking the results of an enqueue. To deal with that problem, we introduced a `getSize` function (shown in Listing 5.4) that calculates the size of the queue. This function was used by the recovery routine to check whether the results of the enqueue operations actually affected the queue.

```
int getSize(bool recovery = false)
{
  int size = 0;
  NodeWithID *aux = head;

  do{
    if (recovery && aux->threadID != -1) {
      size = 0;
      recovery = false;
    }
    aux = aux->next;
    size++;
  } while(aux->next);

  return size;
}
```

Listing 5.4: `getSize` function.

In Listing 5.5 we present the `de-pe+e+e` test, where we spawn two threads that perform one enqueue each and during recovery we check whether the size of the queue has actually been increased by two if both operations have completed. Note, that we are using a `__VERIFIER_assume()` statement, which is a built-in function that stops the execution if the argument given to it is zero. The purpose of this function is to reduce the state space. The corresponding test case for the MS-Queue fails this test.

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <assert.h>

#include ”DurableQueue.h”

static pthread_t threads[2];
static int param[2] = {0, 1};

__VERIFIER_persistent_storage(DurableQueue* queue);
__VERIFIER_persistent_storage(bool done1);
__VERIFIER_persistent_storage(bool done2);

void *thread1(void *param)
{
  done1 = queue->enq(1);
  return NULL;
}

void *thread2(void *param)
{
  done2 = queue->enq(2);
  return NULL;
}

void __VERIFIER_recovery_routine(void)
{
  __VERIFIER_assume(done1 && done2);
  assert(queue->getSize() == 2);
  return;
}

int main()
{
  queue = (DurableQueue*)__VERIFIER_palloc(sizeof(DurableQueue));
  new (queue) DurableQueue();

```

```
39    queue->enq(3);
40
41    __VERIFIER_pbarrier();
42
43    pthread_create(&threads[0], NULL, thread1, &param[0]);
44    pthread_create(&threads[1], NULL, thread2, &param[1]);
45
46    pthread_join(threads[0], NULL);
47    pthread_join(threads[1], NULL);
48
49    return 0;
50  }
```

Listing 5.5: `dq-pe+e+e` test.

The Durable Queue does not flush the `head` and `tail` pointers at any time during normal execution. As a result, one must be very careful when trying to reason about the state of the queue upon recovery. Although a recovery procedure is described in the original work of the Durable Queue [Frie18], there is no official implementation of it in the code repository. Furthermore, PerSeVerE only supports reads from persistent variables during recovery, which would make such a procedure useless in the context of the tool.

To alleviate this issue, we added a boolean variable `recovery` to the `getSize` function, which indicates that the recovery routine is running and therefore the `head` of the queue might have changed. This is particularly helpful when some dequeue operations have taken place and therefore the `head` pointer points to a node that has been dequeued during recovery. We present such an example in Appendix B.

5.3 Flush Elimination Tests

As mentioned earlier, it is important to minimize the amount of `flush` instructions used by the PM workload. Although NVTraverse reduces the number of required `flush`es and `sfence`s, it does not provide any guarantee about the optimality of its transformation. It may be possible that specific `flush`es and `sfence`s can be eliminated. Since PerSeVerE does not currently support the $\texttt{clflush}_{\texttt{opt}}$ and the `sfence` instruction, we are only going to work with the `clflush` instruction (which as we have already discussed is sufficient to enforce the desired persist ordering).

To that extent, we manually added some flags that enable or disable specific `flush`es. For example, in the `insert` function of the NVTraverse linked-list implementation, we can disable the `flush` of the inserting node before it is added to the list with the flag `BIMF`. Similarly, with the flag `BICF` we can remove the `flush` after the `CAS` instruction that swings the `next` field of the predecessor node from the successor to the new node.

```
1  bool insert(int k, int item) {
2    while (true) {
3      Window* window = find(head, k);
4      Node* pred = window->pred;
5      Node* curr = window->curr;
6      free(window);
7      if (curr && curr->key == k)
8        return false;
9
10     Node* node = getNewNode();
11     node->set(k, item, curr);
12 #ifndef BIMF
13     FLUSH(node);
14 #endif
15
16 #ifdef BICF
```

```
17      bool res = pred->CAS_next(curr, node);
18  #else
19      bool res = pred->CAS_nextF(curr, node);
20  #endif
21      if (res)
22        return true;
23    }
24  }
```

Listing 5.6: Code snippet for manually added bugs in NVTraverse linked-list `insert` function. Notice the two flags (`BIMF` and `BICF`) that remove flushes.

Similarly, we have modified the `remove` function of the NVTraverse linked-list. The changes and the flags that enable the bugs can be found in Appendix B.

Another point of interest for eliminating `flush`es is the flushing of a node object. In Listing 5.7, we can see the `Node` class, which has three fields: the `key`, the `value` and a `next` pointer. PerSeVerE provides functionality for flushing a single variable by giving its address as argument to the `__VERIFIER_clflush()` function. However, it is not able to `flush` a whole struct or class, but rather only its first field. Instead, one should take care to `flush` all the fields separately. Although this might seem restrictive, it is on par with how real hardware works: `flush` instructions work on cache line width. It is common for objects to not fit in a single cache line.

```
1  class Node{
2  public:
3    int key;
4    int value;
5    Node* next;
6    Node(int k, int val, Node* n) : key(k), value(val), next(n) {}
7  }
```

Listing 5.7: Linked-list node class.

Based on the above observation, we can use a flag (namely `BMFN`), which enables the flushing of the object, instead of each of its fields. This is showcased in Listing 5.8. This might create some cases, where the `next` field has not been persisted, although the `key` has, which can lead to inconsistent states upon recovery.

```
1  void FLUSH(Node *n)
2  {
3  #ifdef BMFN
4    __VERIFIER_clflush(&n);
5  #else
6    __VERIFIER_clflush(&(n->key));
7    __VERIFIER_clflush(&(n->value));
8    __VERIFIER_clflush(&(n->next));
9  #endif
10 }
```

Listing 5.8: Code snippet for manually added bug to the function that flushes a node. The bug is enabled with the flag `BMFN`.

Finally, we define the flag `BITF`, which disables the flushing of the traverse nodes during the traversal of the list. This should lead to the behaviour illustrated in Example 4.2.

Chapter 6

Results

We ran all tests on a system equipped with an Intel Xeon E5-4650 CPU and 128GB of RAM, running Debian 11.4. GenMC is compiled with and uses LLVM-11. All reported times are in seconds. We provide the expected result for every test case, i.e., what we would expect PerSeVerE to give as output based on the refined Px86 semantics. A timeout of 24 hours is set for running each test case.

6.1 Results of Litmus Tests

The results for the litmus tests are shown in Table 6.1. As we can see, litmus tests require only a small number of executions and very little time. Obviously, one can create more complex litmus tests with more than two threads and more events. However, as we already pointed out, our litmus test suite aims to test the sanity of the Px86 model implementation on PerSeVerE (and possibly other tools), not to stress-test their implementations.

		Executions		
	Expected	Explored	Blocked	Time
2W+2W	unsafe	3		0.04
2W+RFW	safe	3		0.04
2WRW+WFW	unsafe	1		0.04
6W	safe	4		0.04
CAS+CAS	safe	5		0.04
WFW	safe	2		0.04
WFW+RW	safe	3		0.04
WMW+WFW	unsafe	2		0.04
WW	unsafe	2		0.04
WW+RMFW	safe	3		0.04

Table 6.1: Results for litmus tests.

All litmus tests returned the expected results. This indicates that PerSeVerE models successfully some basics behaviours of the refined Px86 model that our litmus tests check. It should be noted that this does not indicate that PerSeVerE models the whole refined Px86 model correctly.

6.2 Results of Data Structure Tests

6.2.1 Results of NVTraverse Tests

The results for the NVTraverse tests are shown in Table 6.2.

		Executions		
	Expected	Explored	Blocked	Time
`lor-ww`	unsafe	5		0.10
`lor-pw+d+d`	unsafe	9		0.28
`lor-pw+w+d`	unsafe	3		0.15
`lor-pw+w+w`	unsafe	2		0.15
`lor-pw+ww`	unsafe	6		0.17
`lor-pw+w+w+d`	unsafe	3		0.18
`lor-pw+w+ww`	unsafe	6		0.26
`liz-ww`	safe	3		0.13
`liz-pw+d+d`	safe	`!!`	`!!`	`!!`
`liz-pw+w+d`	safe	116		3.59
`liz-pw+w+w`	safe	35		1.37
`liz-pw+ww`	safe	3		0.16
`liz-pw+w+w+d`	safe	42293		1981.83
`liz-pw+w+ww`	safe	123		7.77
`ltr-ww`	safe	3		0.17
`ltr-pw+d+d`	safe	118	2	10.41
`ltr-pw+w+d`	safe	76		3.11
`ltr-pw+w+w`	safe	25		1.38
`ltr-pw+ww`	safe	3		0.20
`ltr-pw+w+w+d`	safe	13902		1019.78
`ltr-pw+w+ww`	safe	108		10.04
`slor-pw+d+d`	unsafe	4		0.14
`slor-pw+w+w`	unsafe	2		0.16
`slor-pw+w+w+d`	unsafe	5		0.32
`sliz-pw+d+d`	safe	7119	16	2869.06
`sliz-pw+w+w`	safe	15		7.39
`sliz-pw+w+w+d`	safe	🕒	🕒	🕒
`sltr-pw+d+d`	safe	26732	14	12247.60
`sltr-pw+w+w`	~~safe~~	~~1812~~	~~1~~	~~460.08~~
`sltr-pw+w+w+d`	safe	🕒	🕒	🕒

Table 6.2: Results for the NVTraverse tests. 🕒: timed out (>24h), `!!`: internal crash of PerSeVerE, ~~struck-out~~: wrong result (compared to the expected result).

All test cases from the original implementations of the Harris' linked-list and the lock-free skiplist were designed to be unsafe, since these implementations are not durably linearizable. As can be seen, our test cases exposed the problematic behaviours of these implementations and detected the corresponding bugs.

Almost all test cases for both the linked-list and the skiplist produced the expected results. All the test cases, for the two transformations are denoted as safe, since we do not expect a durable linearizability violation from these transformations.

Looking more carefully into these results, one might observe that the NVTraverse transformation requires on average less explored executions than the Izraelevitz one. This might be counter-intuitive, since the Izraelevitz transformation uses more `flush`es and therefore it is expected to constrain the possible post-crash states. However, the two implementations are not directly comparable, since the NVTraverse one employs a slightly different approach

during the search function. Therefore, it is not safe to compare the number of explored executions between these two models.

We also encountered some test cases that did not behave as expected:

- `liz-pw+d+d` produced an internal error in PerSeVerE. This error was produced because PerSeVerE was not able to create a consistent **rf** relation for the various events. At the time of the writing of this thesis, it is still unknown whether this is a bug in the implementation of the linked-list or if it is an inherent problem of PerSeVerE.
- `sltr-pw+w+w` resulted in an assertion violation, although it was considered safe. We postulate that this comes from a mistake in the test (possibly a missing `flush`) and does not indicate that the data structure is not durably linearizable or that PerSeVerE fails to model the Px86 semantics correctly.

Finally, as stated in Chapter 5, we designed a test case that produces a lot of explored executions. For the linked-lists, we can observe that the Izraelevitz transformation explores more than 40k executions. The corresponding test cases for the skiplist implementations produced a timeout (of 24 hours). This demonstrates that even small concurrent programs for data structures (in our case three threads each performing a single operation) require significant time to be model checked.

6.2.2 Results of Persistent Queue Tests

The results for the Persistent Queue tests are shown in Table 6.3. The test cases are limited in comparison with the NVTraverse tests, since the Persistent Queue data structure is simpler in nature than NVTraverse and does not showcase a lot of tricky behaviours with respect to persistency.

		Executions		
	Expected	Explored	Blocked	Time
`ms-e`	unsafe	3		0.08
`ms-pe+e+e`	unsafe	2		0.10
`dq-e`	safe	1		0.10
`dq-pe+e+d`	safe	6		0.18
`dq-pe+e+e`	safe	28	24	0.53
`dq-pe+d+d`	safe	28	2	0.83

Table 6.3: Results for Persistent Queue tests.

It was easy to show that MS-Queue is not durably linearizable, since in both test cases an assertion violation was detected during recovery. These test cases checked whether the size of the queue was increased accordingly to the number of inserts that were completed before a crash.

On the other side, all of our tests for the Durable Queue, which were denoted as safe, returned the expected result. This indicates that the implementation of the queue successfully follows its (theoretically proven) durably linearizable design.

6.3 Results of Flush Elimination Tests

The results for the various tests of eliminating `flush`es are shown in Table 6.4.

As can be seen, most of the flags produce an error in at least one test case. This indicates that all `flush`es are necessary to ensure durable linearizability. The only exemption

		Executions		
	Result	Explored	Blocked	Time
`ltr-pw+w+w-mfn`	unsafe	2		0.55
`ltr-pw+w+d-mfn`	unsafe	3		0.29
`ltr-pw+w+w-imf`	unsafe	2		0.59
`ltr-pw+w+d-imf`	unsafe	3		0.19
`ltr-pw+w+w-icf`	unsafe	2		0.21
`ltr-pw+w+d-icf`	safe	76		3.37
`ltr-pw+d+d-tmf`	safe	119	2	5.77
`ltr-pw+w+w-tmf`	unsafe	2		0.23
`ltr-pw+w+d-tmf`	safe	77		1.63
`ltr-pw+d+d-rmf`	safe	118	2	11.30
`ltr-pw+w+d-rmf`	safe	76		3.43
`ltr-pw+d+d-rcf`	unsafe	71		7.29
`ltr-pw+w+d-rcf`	safe	76		3.41

Table 6.4: Results for NVTraverse tests with selectively removed `flush`es.

is removing the `flush` right after the node that is going to be removed is accessed. This, however, does not imply that this `flush` can be removed. It is possible that it requires more detailed and complex test cases to discover a potential bug.

As stated in Chapter 5, the `BTMF` flag removes the `flush` instructions performed on the traverse nodes. By enabling this flag with the `ltr-pw+w+w` test case, we should observe the durable linearizability violation illustrated in Example 4.2. Indeed, PerSeVerE detects the bug. However, this bug cannot be detected under the other two combinations that were tested (`ltr-pw+d+d` and `ltr-pw+w+d`).

Another interesting observation from these tests is that by removing some `flush`es, the number of executions explored might increase (in the absence of bugs). For instance, for the original `ltr-pw+w+d` test case PerSeVerE explores 76 executions, while for `ltr-pw+w+d-tmf` (removal of the `flush` instructions of the traverse nodes) it explores one extra execution. This is in fact the expected behaviour. `flush` instructions impose specific restrictions for the ordering of persists and therefore might decrease the total number of post-crash observable executions. Therefore, by removing those flushes, one may observe more post-crash states.

6.4 Comparison with Consistency Checking

Model checking for persistency should theoretically be more expensive in terms of executions explored and time than checking for consistency. In order to demonstrate this, and to put these numbers into perspective, we also ran some tests with GenMC and compared its results with those of PerSeVerE. For the test cases that are dealing with data structures, GenMC used the original implementations of the data structures (since it does not support persistency primitives), while PerSeVerE with the Traverse implementations (since the original implementations produce errors). The assertion inside the recovery routine in PerSeVerE was moved to end of the program when ran in the GenMC normal setting.

We ran both the SC and the TSO consistency model for GenMC. However, we are going to show only the results for TSO, since in most cases GenMC explored the same number of executions under both models. (With one exception that we are going to discuss later on.) The results are shown in Table 6.5.

		GenMC TSO			PerSeVerE		
	Expected	Explored	Blocked	Time	Explored	Blocked	Time
`CAS+CAS`	safe	4		0.04	5(44)		0.04
`l-pw+w+d`	safe	14		0.09	126	2	5.60
`l-pw+w+w+d`	safe	1543	2	5.49	13902		1019.78
`sl-pw+d+d`	safe	50	1	0.26	26732	14	12247.60
`sl-pw+w+w+d`	safe	1400		9.20	🕒	🕒	🕒

Table 6.5: Comparison between consistency and persistency verification. The number in parenthesis indicates the total number of executions that PerSeVerE might explore if all persistent variables are read during recovery.

The `CAS+CAS` litmus test produces 4 executions with GenMC for the consistency checking and 5 executions for the persistency checking at the recovery routine. This is expected, since persistency checking requires not only exploring the various interleaving caused by concurrency, but also takes into account the various crash points that would affect the observed values during recovery.

Another interesting test that we conducted was to measure the total number of executions that PerSeVerE might explore if we let it observe all the persistent variables during recovery. Recall that PerSeVerE only explores executions that are affected by the variables that are being read during recovery. If a variable is not read during recovery, PerSeVerE will consider executions with different values of this variable as equivalent. So, if we read all the persistent variables during recovery, the number of executions explored by PerSeVerE rises to 44, which is an order of magnitude more than the executions required for consistency checking.

In a similar fashion, we can observe that the same pattern is repeated for the various data structures tests. For instance, `l-pw+w+w+d` requires 1543 explored executions for the normal GenMC setting, while PerSeVerE explores 13902 executions.

In general, persistency checking is a lot more expensive in terms of executions explored and, as a result, time. We observed differences up to four orders of magnitude (`sl-pw+d+d`). It should be noted, however, that this is just an empirical observation: there might be even bigger differences for other PM workloads. The workaround that was required to fix PerSeVerE's support for modelling flushes by a thread different than the one that made the write to the variable being flushed, induced a time overhead per execution, which further increased the total time needed for persistency checking.

Chapter 7

Epilogue

7.1 Related Work

As far as testing tools are concerned, there is a wide range of work for testing persistent memory programs (e.g., [Liu19, Neal20, Fu21]) and fuzzing them [Liu21]. However, model checking has not been actively studied for this domain. There are only two other known model checkers for real persistent memory programs, namely Yat [Lant14] and Jaaru [Gorj21]. The former enumerates all possible crash states, which leads to even greater exponential complexity than concurrent programs. This exhaustive search, however, cannot scale and, therefore, it is not suitable for real world programs. On the other hand, Jaaru performs a form of DPOR, by inferring constraints on the last time that a cache line was flushed to persistent memory. The main difference between Jaaru and PerSeVerE is that it uses operational instead of axiomatic semantics for modelling Px86.

There are numerous PM libraries publicly available, like PMDK [Inte15]. Furthermore, there is a wide variety of data structures implementations aimed for PM [Lee19, Izad21, Cai21, Kim21], like key-value stores, trees, etc. In a similar fashion to NVTraverse, more transformations of linearizable data structures to durable ones have been proposed [Frie21b], as well as some libraries that offer great abstractions and require minimal user intervention [Wei22]. However, most of them are not formally verified. Instead, they mainly employ unit testing, which is not enough to prove (or disprove) their correctness.

7.2 Conclusion and Future Work

Through this thesis, we presented an overview of the basics of persistent memory. We discussed the notion of persistency models and how they are related with consistency models. Afterwards, we experimented with a state-of-the-art stateless model checker (GenMC) and, more precisely, the under development extension of PerSeVerE, which supports the Px86 model. We introduced a test suite for PM programs, which consists of both litmus and lock-free data structure tests.

Through litmus testing we managed to find an internal problem of PerSeVerE, which was leading to incorrect results. This lead the developers to add extra support in GenMC's core to fix this issue. Finally, we tried to verify some implementations of lock-free data structures. We proved through model checking that the original implementations of these data structures were not durably linearizable and we also tested the Izraelevitz and the NVTraverse transformation. Our test suite can be used as a guideline for creating tests for PM programs and potentially be included in a more complete benchmark suite in the future.

An obvious direction for future work is to add $\texttt{flush}_{\texttt{opt}}$ and `sfence` instructions to PerSeVerE. By using $\texttt{flush}_{\texttt{opt}}$, one can achieve better performance than `flush`. However, it should be used together with `sfence` in order to ensure the desired persist ordering. It would be interesting to test the various data structures with these two instructions and try to find the minimal numbers of fences that are required for correct recovery.

Another conspicuous extension of our work would be to try to adapt other implementations of data structures for PM, in order to be used with PerSeVerE. Our test suite can be used as a guideline for the design of the test cases. Afterwards, it would be interesting to compare the complexity and the reduction achieved among these data structures. It would also be intriguing to couple these data structure implementations with allocators and garbage collectors designed for PM [Cai20] and try to verify them together.

As far as the litmus test suite is concerned, it would be interesting to use the *Alloy** constraint solver [Mili15] to automatically produce litmus tests for Px86.

Various persistency models for both experimental and real architectures have been proposed in recent literature; these range from strict persistency models for TSO such as TSOPER [Ekem21] to relaxed ones such as PARMv8 [Raad19] and PSC [Khyz21]. In addition, language level persistency models have been recently discussed [Koll19]. These models can easily be embedded in PerSeVerE by using kater in the same way as Px86. It would be really interesting to also check for robustness violations between strict and relaxed persistency models, for the data structures included in our suite. (An implementation is considered *robust* iff it showcases the same behaviour between the two memory models under test.) A similar work has been done for Jaaru [Gorj22].

Finally, it would be really useful to modify PerSeVerE in order to support proper recovery procedures, like the one described in the Durable Queue. These procedures would be the first during the recovery routine of PerSeVerE. Most recent designs of durably linearizable data structures rely on such recovery procedures in order to ensure consistency after a crash. We postulate that such an addition would not be particularly hard.

Appendix A

Test Suite Interface

The test suite is available at https://github.com/spyrospav/pm-benchmarks[1]. In order to run it, a driver shell script (`pmdriver.sh`) is provided, which accepts the following (optional) arguments:

`-h | --help`, which is a help flag and prints the instructions of how to run the script, as well as explanation for the various command-line arguments.

`-m | --mode`, which takes another (required) argument, in order to select which mode to run. One mode is `default`, which runs the normal test suite, and the other is `missing`, which runs the buggy test cases. There is also an option to run both modes (`all`).

`-d | --debug`, which is a debug flag, that produces debug information like execution traces and error graphs. For the production of the error graphs, we utilize the built-in functionality of GenMC to produce *.dot* files, which we turn into *.png* files. An example of an error graph is given in Figure A.1.

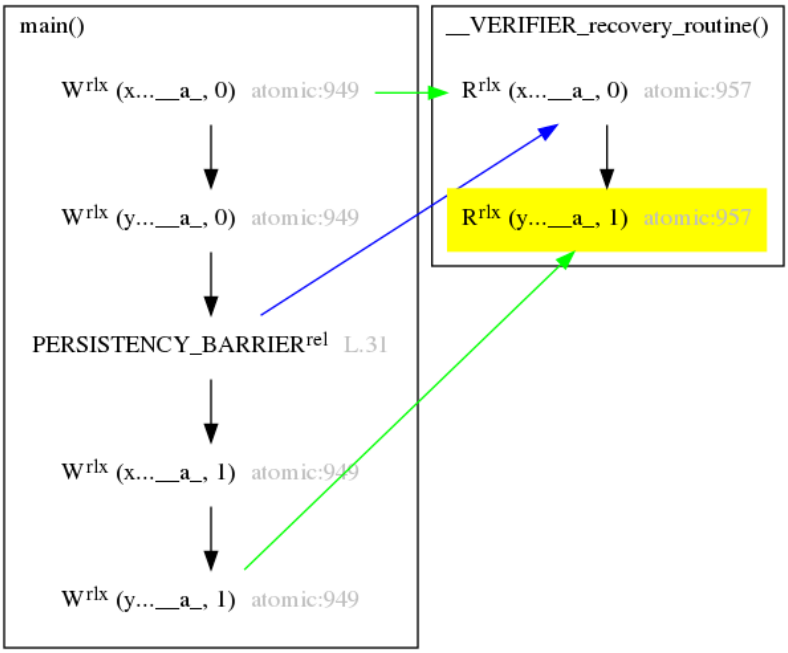


Figure A.1: Error graph for `WW` litmus test. → denotes coherence order between events, → denotes the reads-from relation.

The test suite contains a dictionary (in `catalog.sh`), which relates the various tests with their expected results. One can easily add more tests to the suite by inserting new entries to this file and follow the design of the driver script to run their tests.

[1] At the time of the writing of this thesis, the version of GenMC that we used had not been publicly released.

Appendix B

Source Code

In this appendix, we include the source code for some of the tests that were mentioned in the main text of this thesis.

B.1 Litmus Tests

```
#include <stdio.h>
#include <stdlib.h>
#include <atomic>
#include <pthread.h>
#include <assert.h>
#include <genmc.h>

#define relaxed std::memory_order_relaxed

__VERIFIER_persistent_storage(std::atomic_int x);
__VERIFIER_persistent_storage(std::atomic_int y);

extern ''C''{
 void __VERIFIER_clflush(void*);
}

void __VERIFIER_recovery_routine(void)
{
  assert(!(x.load(relaxed) == 0) && (y.load(relaxed) == 1));
  return;
}

int main()
{
  x.store(0, relaxed);
  y.store(0, relaxed);

  __VERIFIER_pbarrier();

  x.store(1, relaxed);
  __VERIFIER_clflush(&x);
  y.store(1, relaxed);

  return 0;
}
```

Listing B.1: `WFW` litmus test

```
#include <stdio.h>
#include <stdlib.h>
#include <atomic>
#include <pthread.h>
#include <assert.h>
#include <genmc.h>
```

```

#define relaxed std::memory_order_relaxed

static pthread_t threads[2];
static int param[2] = {0, 1};

__VERIFIER_persistent_storage(std::atomic_int x);
__VERIFIER_persistent_storage(std::atomic_int y);
__VERIFIER_persistent_storage(std::atomic_int z);
int a = 0;

extern ''C''{
 void __VERIFIER_clflush(void*);
}

void *thread1(void *param)
{
  x.store(42, relaxed);
  y.store(7, relaxed);
  return NULL;
}

void *thread2(void *param)
{
  a = y.load();
  if (a != 0) {
    __VERIFIER_clflush(&x);
    z.store(1, relaxed);
  }
  return NULL;
}

void __VERIFIER_recovery_routine(void)
{
  if (z.load(relaxed) == 1)
    assert(x.load(relaxed) == 42);
  return;
}

int main()
{
  x.store(0, relaxed);
  y.store(0, relaxed);
  z.store(0, relaxed);

  __VERIFIER_pbarrier();

  pthread_create(&threads[0], NULL, thread1, &param[0]);
  pthread_create(&threads[1], NULL, thread2, &param[1]);

  pthread_join(threads[0], NULL);
  pthread_join(threads[1], NULL);

  return 0;
}
```

Listing B.2: `2W+RFW` litmus test

```
#include <stdio.h>
#include <stdlib.h>
#include <atomic>
#include <pthread.h>
#include <assert.h>
#include <genmc.h>
```

```
7
8  #define relaxed std::memory_order_relaxed
9
10 static pthread_t threads[2];
11 static int param[2] = {0, 1};
12
13 __VERIFIER_persistent_storage(std::atomic_int x);
14 __VERIFIER_persistent_storage(std::atomic_int y);
15 __VERIFIER_persistent_storage(std::atomic_int z);
16 __VERIFIER_persistent_storage(std::atomic_int lx);
17
18 int zero = 0;
19
20 extern ''C''{
21  void __VERIFIER_clflush(void*);
22 }
23
24 void *thread1(void *param)
25 {
26   bool a1 = lx.compare_exchange_strong(zero, 1, relaxed);
27   if (a1) {
28     x.store(1, relaxed);
29     y.store(1, relaxed);
30     __VERIFIER_clflush(&x);
31     __VERIFIER_clflush(&y);
32     lx.store(0, relaxed);
33   }
34   return NULL;
35 }
36
37 void *thread2(void *param)
38 {
39   bool a2 = lx.compare_exchange_strong(zero, 2, relaxed);
40   if (a2) {
41     int a3 = y.load(relaxed);
42     if (a3 == 1) {
43       z.store(1, relaxed);
44       __VERIFIER_clflush(&z);
45     }
46     lx.store(0, relaxed);
47   }
48   return NULL;
49 }
50
51 void __VERIFIER_recovery_routine(void)
52 {
53   if (z.load(relaxed) == 1)
54     assert(x.load(relaxed) == 1 && y.load(relaxed) == 1);
55   return;
56 }
57
58 int main()
59 {
60   x.store(0, relaxed);
61   y.store(0, relaxed);
62   z.store(0, relaxed);
63   lx.store(0, relaxed);
64
65   __VERIFIER_pbarrier();
66
67   pthread_create(&threads[0], NULL, thread1, &param[0]);
68   pthread_create(&threads[1], NULL, thread2, &param[1]);
69
```

```
  pthread_join(threads[0], NULL);
  pthread_join(threads[1], NULL);

  return 0;
}
```

Listing B.3: `CAS+CAS` litmus test

B.2 Data Structure Tests

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <assert.h>

#include "../ListOriginal.h"

static pthread_t threads[3];
static int param[3] = {0, 1, 2};

__VERIFIER_persistent_storage(static ListOriginal* list);

void *thread1(void *param)
{
  list->remove(3);
  return NULL;
}

void *thread2(void *param)
{
  list->insert(2, 10);
  return NULL;
}

void *thread3(void *param)
{
  list->insert(1, 10);
  return NULL;
}

void __VERIFIER_recovery_routine(void)
{
  assert(list->contains(4));
  return;
}

int main()
{
  list = (ListOriginal*)__VERIFIER_palloc(sizeof(ListOriginal));
  new (list) ListOriginal();

  list->insert(0,10);
  list->insert(3,10);
  list->insert(4,10);

  __VERIFIER_pbarrier();

  pthread_create(&threads[0], NULL, thread1, &param[0]);
  pthread_create(&threads[1], NULL, thread2, &param[1]);
  pthread_create(&threads[2], NULL, thread2, &param[2]);

  pthread_join(threads[0], NULL);
  pthread_join(threads[1], NULL);
```

```
  pthread_join(threads[2], NULL);

  return 0;
}
```

Listing B.4: `lor-pw+w+w+d` test

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <assert.h>

#include ”DurableQueue.h”

static pthread_t threads[3];
static int param[3] = {0, 1, 2};

__VERIFIER_persistent_storage(DurableQueue* queue);
__VERIFIER_persistent_storage(bool done1 = false);
__VERIFIER_persistent_storage(bool done2 = false);

void *thread1(void *param)
{
  queue->enq(2);
  done1 = true;
  return NULL;
}

void *thread2(void *param)
{
  queue->deq(2);
  done2 = true;
  return NULL;
}

void __VERIFIER_recovery_routine(void)
{
  __VERIFIER_assume(done1 && done2);
  assert(queue->removedValues[2] == 1);
  int x = queue->getSize(true);
  assert(x == 1);
  return;
}

int main()
{
  queue = (DurableQueue*)__VERIFIER_palloc(sizeof(DurableQueue));
  new (queue) DurableQueue();

  queue->enq(1);

  __VERIFIER_pbarrier();

  pthread_create(&threads[0], NULL, thread1, &param[0]);
  pthread_create(&threads[1], NULL, thread2, &param[1]);

  pthread_join(threads[0], NULL);
  pthread_join(threads[1], NULL);

  assert(queue->getSize() == 1);

  return 0;
```

```
}
```

Listing B.5: `dq-pe+e+d` test

B.3 Flush Elimination Tests

```
bool remove(int key) {
  bool snip = false;
  while (true) {
    Window* window = find(head, key);
    Node* pred = window->pred;
    Node* curr = window->curr;
    free(window);
    if (!curr || curr->key != key) {
      return false;
    }
    else {
#ifdef BRMF
      Node* succ = curr->getNext();
#else
      Node* succ = curr->getNextF();
#endif
      Node* succAndMark = mark(succ);
      if (succ == succAndMark) {
        continue;
      }
#ifdef BRCF
      snip = curr->CAS_next(succ, succAndMark);
#else
      snip = curr->CAS_nextF(succ, succAndMark);
#endif
      if (!snip)
        continue;
      if (pred->CAS_nextF(curr, succ)){
        free(curr);
      }
      return true;
    }
  }
}
```

Listing B.6: Code snippet for manually added bugs in NVTraverse linked-list remove function. There are 2 flags, namely BRMF and BRCF, that remove flushes.

Bibliography

[Abdu14] Parosh Abdulla, Stavros Aronis, Bengt Jonsson and Konstantinos Sagonas, "Optimal Dynamic Partial Order Reduction", in *Symposium on Principles of Programming Languages*, POPL 2014, pp. 373–384, January 2014. ↪ p. 19, 35, and 43

[Abdu15] Parosh Abdulla, Stavros Aronis, Mohammed Faouzi Atig, Bengt Jonsson, Carl Leonardsson and Konstantinos Sagonas, "Stateless Model Checking for TSO and PSO", in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 9035 of *LNCS*, pp. 353–367, Berlin, Heidelberg, April 2015, Springer. ↪ p. 43

[Abdu17] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson and Konstantinos Sagonas, "Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction", *Journal of the ACM*, vol. 64, no. 4, pp. 25:1–25:49, September 2017. ↪ p. 43

[Abdu19] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo and Konstantinos Sagonas, "Optimal Stateless Model Checking for Reads-From Equivalence Under Sequential Consistency", *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 150:1–150:29, October 2019. ↪ p. 43

[Adve96] Sarita V. Adve and Kourosh Gharachorloo, "Shared memory consistency models: a tutorial", *Computer*, vol. 29, no. 12, pp. 66–76, 1996. ↪ p. 18 and 37

[Algl12] Jade Alglave, "A formal hierarchy of weak memory models", *Form Methods Syst Des*, vol. 41, no. 2, pp. 178–210, October 2012. ↪ p. 18 and 40

[Algl14] Jade Alglave, Luc Maranget and Michael Tautschnig, "Herding cats: modelling, simulation, testing, and data-mining for weak memory", in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, p. 40, ACM, June 2014. ↪ p. 43

[Aron18] Stavros Aronis, Bengt Jonsson, Magnus Lång and Konstantinos Sagonas, "Optimal Dynamic Partial Order Reduction with Observers", in *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference*, vol. 10806 of *LNCS*, pp. 229–248, Cham, April 2018, Springer. ↪ p. 43

[Cai20] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati and Michael L. Scott, "Understanding and optimizing persistent memory allocation", in *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, pp. 60–73, London UK, June 2020, ACM. ↪ p. 80

[Cai21] Wentao Cai, Haosen Wen, Vladimir Maksimovski, Mingzhe Du, Rafaello Sanna, Shreif Abdallah and Michael L. Scott, "Fast Nonblocking Persistence for Concurrent Data Structures", *arXiv:2105.09508 [cs]*, August 2021. arXiv: 2105.09508. ↪ p. 31 and 79

[Ekem21] Per Ekemark, Yuan Yao, Alberto Ros, Konstantinos Sagonas and Stefanos Kaxiras, "TSOPER: Efficient Coherence-Based Strict Persistency", in *IEEE International Symposium on High-Performance Computer Architecture*, HPCA 2021, pp. 137–150, IEEE Computer Society, March 2021. ↪ p. 31, 47, and 80

[Flan05] Cormac Flanagan and Patrice Godefroid, "Dynamic partial-order reduction for model checking software", in *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pp. 110–121, New York, NY, USA, January 2005, ACM. ↪ p. 19, 35, 41, and 43

[Frie18] Michal Friedman, Maurice Herlihy, Virendra Marathe and Erez Petrank, "A persistent lock-free queue for non-volatile memory", in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 28–40, Vienna Austria, February 2018, ACM. ↪ p. 20, 35, 50, 58, 60, and 70

[Frie21a] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch and Erez Petrank, "NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey", *arXiv:2004.02841 [cs]*, November 2021. arXiv: 2004.02841. ↪ p. 20, 23, 35, 50, and 61

[Frie21b] Michal Friedman, Erez Petrank and Pedro Ramalhete, "Mirror: making lock-free data structures persistent", in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 1218–1232, Virtual Canada, June 2021, ACM. ↪ p. 31, 50, and 79

[Fu21] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee and Changwoo Min, "Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores", in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pp. 100–115, ACM, October 2021. ↪ p. 79

[Gode96] Patrice Godefroid, J. van Leeuwen, J. Hartmanis, G. Goos and Pierre Wolper, *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, Springer-Verlag, Berlin, Heidelberg, 1996. ↪ p. 41 and 42

[Gode97] Patrice Godefroid, "Model checking for programming languages using VeriSoft", in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '97*, pp. 174–186, Paris, France, 1997, ACM Press. ↪ p. 41

[Gorj21] Hamed Gorjiara, Guoqing Harry Xu and Brian Demsky, "Jaaru: efficiently model checking persistent memory programs", in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 415–428, Virtual USA, April 2021, ACM. ↪ p. 79

[Gorj22] Hamed Gorjiara, Weiyu Luo, Alex Lee, Guoqing Harry Xu and Brian Demsky, "Checking Robustness to Weak Persistency Models", 2022. ↪ p. 80

[Harr01] Tim Harris, "A Pragmatic Implementation of Non-Blocking Linked Lists", in *Proceedings of the 15th International Symposium on Distributed Computing*, vol. 2180 of *LNCS*, pp. 300–314, Springer-Verlag, October 2001. ↪ p. 22 and 54

[Herl90] Maurice P. Herlihy and Jeannette M. Wing, "Linearizability: a correctness condition for concurrent objects", *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, July 1990. ↪ p. 21, 53, and 54

[Herl08] Maurice Herlihy and Nir Shavit, *The Art of Multiprocessor Programming*, Elsevier/Morgan Kaufmann, Amsterdam/Boston, 2008. ↪ p. 22, 55, and 58

[Inte15] Intel, "PMem.io", `https://pmem.io/`, 2015. ↪ p. 31 and 79

[Inte19a] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals", https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2019. ↪ p. 48

[Inte19b] Intel, "Intel® Optane™ Persistent Memory", https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html, 2019. ↪ p. 19 and 45

[Izad21] Ramin Izadpanah, Christina Peterson, Yan Solihin and Damian Dechev, "PETRA: Persistent Transactional Non-blocking Linked Data Structures", *ACM Trans. Archit. Code Optim.*, vol. 18, no. 2, pp. 1–26, June 2021. ↪ p. 31 and 79

[Izra16] Joseph Izraelevitz, Hammurabi Mendes and Michael L. Scott, "Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model", in *Distributed Computing*, vol. 9888 of *LNCS*, pp. 313–327, Springer, Berlin, Heidelberg, 2016. ↪ p. 20, 22, 23, 47, 48, 50, 58, and 60

[Josh15] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra and Stratis Viglas, "Efficient Persist Barriers for Multicores", in *Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 2015, pp. 660–671, ACM, December 2015. ↪ p. 47

[Kawa12] Takayuki Kawahara, Kenchi Ito, Riichiro Takemura and Hideo Ohno, "Spin-transfer torque RAM technology: Review and prospect", *Microelectronics Reliability*, vol. 52, no. 4, pp. 613–627, 2012. ↪ p. 19 and 45

[Khyz21] Artem Khyzha and Ori Lahav, "Taming x86-TSO persistency", *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–29, January 2021. ↪ p. 20, 47, 50, 51, and 80

[Kim21] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap and Changwoo Min, "PACTree: A High Performance Persistent Range Index Using PAC Guidelines", in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pp. 424–439, ACM, October 2021. ↪ p. 31 and 79

[Koko19] Michalis Kokologiannakis, Azalea Raad and Viktor Vafeiadis, "Model Checking for Weakly Consistent Libraries", in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 96–110, ACM, June 2019. ↪ p. 43

[Koko21a] Michalis Kokologiannakis, Ilya Kaysin, Azalea Raad and Viktor Vafeiadis, "PerSeVerE: persistency semantics for verification under ext4", *Proc. ACM Program. Lang.*, vol. 5, no. POPL, pp. 1–29, January 2021. ↪ p. 21, 35, and 51

[Koko21b] Michalis Kokologiannakis and Viktor Vafeiadis, "GenMC: A Model Checker for Weak Memory Models", in *Computer Aided Verification*, vol. 12759 of *LNCS*, pp. 427–440, Springer International Publishing, Cham, 2021. ↪ p. 19 and 43

[Koll19] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, William Wang, Peter M. Chen, Satish Narayanasamy and Thomas F. Wenisch, "Language Support for Memory Persistency", *IEEE Micro*, vol. 39, no. 3, pp. 94–102, May 2019. ↪ p. 80

[Kuma15] Suhas Kumar, "Fundamental Limits to Moore's Law", 2015. ↪ p. 45

[Lamp79] Leslie Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", *IEEE Transactions on Computers C-28*, vol. 9, pp. 690–691, September 1979. ↪ p. 18 and 37

[Lant14] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran and Jeff Jackson, "Yat: A Validation Framework for Persistent Memory Software", in *Proceedings of the 2014 USENIX Annual Technical Conference*, USENIX ATC'14, pp. 433–438, USA, 2014, USENIX Association. ↪ p. 79

[Lee09] Benjamin C. Lee, Engin Ipek, Onur Mutlu and Doug Burger, "Architecting Phase Change Memory as a Scalable Dram Alternative", in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 2–13, New York, NY, USA, 2009, ACM. ↪ p. 19 and 45

[Lee19] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim and Vijay Chidambaram, "Recipe: converting concurrent DRAM indexes to persistent-memory indexes", in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 462–477, ACM, October 2019. ↪ p. 31 and 79

[Liu19] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli and Samira Khan, "PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs", in *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 411–425, ACM, April 2019. ↪ p. 79

[Liu21] Sihang Liu, Suyash Mahar, Baishakhi Ray and Samira Khan, "PMFuzz: test case generation for persistent memory programs", in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 487–502, Virtual USA, April 2021, ACM. ↪ p. 79

[Mich96] Maged M. Michael and Michael L. Scott, "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms", in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pp. 267–275, New York, NY, USA, 1996, ACM. ↪ p. 22, 56, and 57

[Mich02a] Maged M. Michael, "High Performance Dynamic Lock-Free Hash Tables and List-Based Sets", in *Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '02, pp. 73–82, New York, NY, USA, 2002, ACM. ↪ p. 54 and 55

[Mich02b] Maged M. Michael, "Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes", in *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*, PODC '02, pp. 21–30, New York, NY, USA, 2002, ACM. ↪ p. 22, 54, and 58

[Mili15] Aleksandar Milicevic, Joseph P. Near, Eunsuk Kang and Daniel Jackson, "Alloy: A General-Purpose Higher-Order Relational Constraint Solver", in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 609–619, 2015. ↪ p. 80

[Neal20] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter and Baris Kasikci, "AGAMOTTO: How Persistent is Your Persistent Memory Application?", in *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, USENIX Association, USA, 2020. ↪ p. 79

[Pell14] Steven Pelley, Peter M. Chen and Thomas F. Wenisch, "Memory persistency", *SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 265–276, October 2014. ↪ p. 20, 45, 46, 47, and 50

[Qure09] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan and Jude A. Rivers, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology", in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 24–33, New York, NY, USA, 2009, ACM. ↪ p. 46

[Raad18] Azalea Raad and Viktor Vafeiadis, "Persistence semantics for weak memory: integrating epoch persistency with the TSO memory model", *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 1–27, October 2018. ↪ p. 31 and 47

[Raad19] Azalea Raad, John Wickerson and Viktor Vafeiadis, "Weak persistency semantics from the ground up: formalising the persistency semantics of ARMv8 and transactional models", *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 1–27, October 2019. ↪ p. 47 and 80

[Raad20] Azalea Raad, John Wickerson, Gil Neiger and Viktor Vafeiadis, "Persistency semantics of the Intel-x86 architecture", *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 1–31, January 2020. ↪ p. 20, 47, 48, 49, and 50

[Scar20] Steve Scargall, *Programming Persistent Memory: A Comprehensive Guide for Developers*, Apress, Berkeley, CA, 2020. ↪ p. 20 and 50

[Sewe10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli and Magnus O. Myreen, "x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors", *Commun. ACM*, vol. 53, no. 7, pp. 89–97, July 2010. ↪ p. 48

[Sori11] Daniel J. Sorin, Mark D. Hill and David A. Wood, "A Primer on Memory Consistency and Cache Coherence", *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, pp. 1–212, November 2011. ↪ p. 38 and 39

[Vafe22] Eleni Vafeiadi Bila, Brijesh Dongol, Ori Lahav, Azalea Raad and John Wickerson, "View-Based Owicki-Gries Reasoning for Persistent x86-TSO", in *Programming Languages and Systems*, vol. 13240 of *LNCS*, pp. 234–261, Springer International Publishing, Cham, March 2022. ↪ p. 65

[Wang19] William Wang and Stephan Diestelhorst, "Persistent Atomics for Implementing Durable Lock-Free Data Structures for Non-Volatile Memory (Brief Announcement)", in *The 31st ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '19, pp. 309–311, New York, NY, USA, June 2019, ACM. ↪ p. 59

[Wei22] Yuanhao Wei, Naama Ben-David, Michal Friedman, Guy E. Blelloch and Erez Petrank, "FliT: a library for simple and efficient persistent algorithms", in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 309–321, ACM, April 2022. ↪ p. 31 and 79

[Wu20] Kai Wu, Ivy Peng, Jie Ren and Dong Li, "Ribbon: High Performance Cache Line Flushing for Persistent Memory", in *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pp. 427–439, ACM, September 2020. ↪ p. 50

[Zuri19] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen and Erez Petrank, "Efficient Lock-Free Durable Sets", *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, October 2019. ↪ p. 50