Konrad Siek

Rozproszona Pesymistyczna Pamięć Transakcyjna: Algorytmy i Własności

Streszczenie rozprawy doktorskiej

Wydział Informatyki
Politechniki Poznańskiej

Promotor: dr hab. inż. Paweł T. Wojciechowski

Poznań · 2016

Spis treści

1 Wstęp

Programowanie współbieżne jest powszechnie uznawane za trudne (zob. np. [11, 18, 19, 30]). Źródłem trudności jest współbieżne wykonanie programu, które może potencjalnie prowadzić do przeplotu operacji wykonywanych przez wątki lub procesy na zmiennych współdzielonych, dając w efekcie nieprawidłowe wyniki. Przykładem jest błędne odczytanie przez proces innej wartości zmiennej współdzielonej, niż ta ostatnio zapisana do tej zmiennej przez ten proces. Dlatego więc programista musi przewidzieć i wyeliminować tego typu nieprawidłowe zachowanie, synchronizując wykonanie niektórych operacji. W tym celu programista ma do dyspozycji odpowiednie niskopoziomowe konstrukcje synchronizacyjne, np. zamki (ang. *locks*), monitory (ang. *monitors*), bariery (ang. *barriers*), czy semafory (ang. *semaphores*). Jednakże korzystanie z tych mechanizmów w sposób poprawny i efektywny nie jest łatwe, gdyż wymaga wnikliwej analizy całego systemu, a błędy wynikające z niewłaściwego zastosowania synchronizacji są często trudne do wykrycia z uwagi na niedeterminizm. Błędna synchronizacja ma poważne skutki dla działania systemu, np. odczyt niespójnego stanu (ang. *inconsistent views*), zakleszczenie (ang.*deadlock*, *livelock*), hazard (ang. *race condition*), lub inwersja priorytetów (ang. *priority inversion*).

Programowanie współbieżne jest jednak nieuniknione. Wynika to ze wzrastającej popularności procesorów wielordzeniowych, gdzie współbieżne wykonanie programu jest niezbędne, aby wykorzystać potencjał wielu rdzeni procesora. Ponadto, wraz z popularyzacją architektur zorientowanych na usługi (ang. *service oriented architectures*) oraz przetwarzania w chmurze (ang. *cloud computing*), także systemy rozproszone, które są z natury współbieżne, stały się wszechobecne. Jest to widoczne do tego stopnia, że rozmaite aplikacje, począwszy od edycji dokumentów tekstowych, a kończąc na bazach danych i aplikacjach typu *Big Data*, coraz częściej delegują przetwarzanie do usług zdalnych, które wykonują określony program równolegle z programem klienta.

Skoro programiści aplikacji coraz częściej stykają się z problemami programowania współbieżnego, niezbędnym staje się dostarczenie im odpowiednich abstrakcji, które pozwalałyby wyeliminować część z tych problemów oraz ukryć szczegóły implementacji mechanizmów synchronizacji. Abstrakcje takie są wykorzystywane w innych dziedzinach programowania. Przykładowo, programiści nie piszą w praktyce własnych rozwiązań do komunikacji przez sieć, lecz korzystają z hermetycznych bibliotek (np. Netty, JGroups, Java Message Service lub Java RMI), które dostarczają tego typu funkcjonalność w formie wysokopoziomowego API, które zwalnia programistę od implementowania szczegółów zarządzania gniazdami czy serializacji danych. W podobny sposób programiści aplikacji powinni móc tworzyć systemy współbieżne.

1.1 Pamięć Transakcyjna

Pamięć transakcyjna (ang. *transactional memory*, *TM*) [22, 32] jest zapożyczoną z systemów bazodanowych uniwersalną propozycją rozwiązania problemu synchronizacji w systemach współbieżnych poprzez zastosowanie abstrakcji transakcji (zob. np. [6, 9, 44]). W tym podejściu programista jedynie oznacza fragmenty kodu wymagające synchronizacji jako transakcje, a a system pamięci transakcyjnej jest odpowiedzialny za wykonanie poszczególnych operacji na danych współdzielonych w ramach transakcji w taki sposób, żeby zapewniona była wydajność i poprawność. Wykonanie kodu transakcji zostaje powierzone odpowiedniemu algorytmowi sterowania współbieżnością, który zapewnia, że przeplot operacji spełnia konkretne gwarancje poprawności. Gwarancje te określone są przez własności bezpieczeństwa które opisują dany algorytm, np. uszeregowalność (ang. *seria-*

lizability) [27] lub nieprzezroczystość (ang. *opacity*) [17]. Programista, z kolei, nie musi znać szczegółów działania zastosowanego algorytmu sterowania współbieżnością, a jedynie własności bezpieczeństwa, które ten algorytm spełnia. W konsekwencji, abstrakcja transakcji ukrywająca szczegóły implementacji algorytmów sterowania współbieżnością oraz spełniająca określone własności bezpieczeństwa, znacząco upraszcza programowanie, wspierając jednocześnie poprawność i wydajność działania systemów współbieżnych.

Rozproszona pamięć transakcyjna (ang. *distributed transactional memory*, *DTM*) [8, 25, 10, 30, 43, 5] przenosi ideę pamięci transakcyjnej do systemów rozproszonych. Generalizacja ta powoduje potrzebę rozwiązania dodatkowych problemów jak asynchroniczność i awarie częściowe, ale także stwarza nowe perspektywy. Cechą najbardziej odróżniającą transakcje w pamięci transakcyjnej od ich bazodanowych poprzedników jest możliwość wykonywania innych operacji niż tylko odczyt i zapis na zmiennych współdzielonych. W systemach pamięci transakcyjnej współdzielone między transakcjami mogą być obiekty, których operacje odczytu i zapisu mają bardziej złożoną semantykę, np. liczniki, czy kolejki. Mogą to być także obiekty których interfejsy są dowolnie zdefiniowane przez programistę i których implementacja ma arbitralną semantykę. W rozproszonej pamięci transakcyjnej dowolność definicji obiektów może dodatkowo służyć umiejscowieniu wykonania pewnego kodu na konkretnych (zdalnych) węzłach sieci. W szczególności wyróżnia się dwa modele. Model przepływu danych (ang. *data flow*) zakłada, że obiekt na którym wykonywana jest operacja migruje do węzła, na którym wykonywana jest transakcja, i właśnie tam wykonywany jest także kod operacji. W tym modelu efekty operacji są zawsze lokalne względem transakcji. Model przepływu sterowania (ang. *control flow*) zakłada, że obiekty są nieruchome i kod operacji wykonuje się zawsze na węźle „domowym". Oznacza to, że efekty operacji są lokalne względem obiektu, a nie transakcji. Każdy z modeli ma swoje wady i zalety, ale unikatową cechą modelu przepływu sterowania jest to, że pozwala on na „pożyczanie" przez transakcje mocy obliczeniowej od zdalnych węzłów. Pozwala to na większą elastyczność przy projektowaniu i implementacji aplikacji rozproszonych.

1.2 Optymistyczne Sterowanie Współbieżnością

Powszechnie stosowanym podejściem do synchronizacji w (rozproszonej) pamięci transakcyjnej jest podejście optymistyczne. W podejściu tym, w ujęciu ogólnym, transakcje są wykonywane jednocześnie bez względu na charakterystykę dostępów wewnątrz transakcji, a próba walidacji ich poprawności następuje później, np. w momencie zatwierdzenia (ang. *commit*) gdy wszystkie operacje transakcji zostały wykonane. Zatwierdzenie kończy się powodzeniem, gdy wykonanie transakcji przebiegło w sposób poprawny. Niepowodzenie zatwierdzenia następuje w wypadku gdy bieżąca transakcja wykonała nieprawidłowe operacje, np. usiłując operować na tym samym obszarze pamięci (zmiennej współdzielonej, obiekcie) jednocześnie z inną transakcją w sytuacji, gdy przynajmniej jedna z nich próbuje ten obszar modyfikować. Scenariusz taki nazywany jest konfliktem (ang. *conflict*) i wymaga on wycofania (ang. *abort*) jednej z transakcji. Wycofanie transakcji oznacza, że transakcja usuwa wszelkie oznaki swojego wykonania. Następnie transakcja wycofana zostanie wykonana po raz kolejny, w nadziei, że tym razem konflikt nie wystąpi i transakcja zostanie zatwierdzona poprawnie. Podejście optymistyczne można zaimplementować na wiele sposobów, ale typowym jest buforowanie operacji lub ich wyników podczas działania transakcji, i wprowadzanie zmian do oryginalnego obiektu dopiero podczas zatwierdzenia (ang. *commit-time*), raczej niż modyfikowanie obiektów już podczas wykonywania operacji (ang. *encounter-time*). Typowe jest wykrywanie konfliktów jak najwcześniej, w celu minimalizacji marnotrawienia pracy wykonanej przez ostatecznie wycofane transakcje.

Podejście optymistyczne jest dość uniwersalnym rozwiązaniem, ale ma ono dwa man-

kamenty. Po pierwsze, podejście optymistyczne napotyka na problemy wynikające ze spekulacyjnego wykonywania transakcji w środowiskach z wysokim współczynnikiem współzawodnictwa (ang. *contention*)—tj. takich, gdzie wiele transakcji jednocześnie ubiega się o wykonanie operacji na tym samym obiekcie. Wysokie współzawodnictwo sprawia, że prawdopodobieństwo wystąpienia konfliktów wzrasta, co z kolei powoduje, że wzrasta częstość z jaką transakcje są wycofywane. W efekcie zwiększone jest prawdopodobieństwo, że dana transakcja będzie wielokrotnie wycofana i wielokrotnie (co najmniej częściowo) wykonana, zanim w końcu zostanie zatwierdzona. Co więcej, wykonywanych jest wiele obliczeń, których wyniki są następnie ignorowane, a po odjęciu transakcji wycofanych, konfliktujące się transakcje w praktyce wykonują się sekwencyjnie. Istnieją mechanizmy, które eliminują wyżej opisany problem problem przez zarządzanie ponownym uruchamianiem konfliktujących transakcji, tak, aby nie doprowadzać do ponownych konfliktów. W tym celu można stosować proste rozwiązania jak wykładnicze opóźnienie (ang. *exponential back-off*) [21], lub rozwiązania złożone, jak szeregowanie oparte o prawdopodobieństwo wystąpienia konfliktów [13, 49], a także inne mechanizmy sterowania współzawodnictwem (ang. *contention management*) [14, 31]. Rozwiązania te mają opóźniać wykonanie niektórych transakcji (zazwyczaj po pierwszym konflikcie), powodując obniżenie liczby współbieżnych transakcji. Rozwiązania te natomiast wymagają parametryzacji, co wymaga strojenia parametrów ręcznie lub wyprowadzania ich podczas działania systemu, a także prowadzi do konieczności reakcji na zmiany profilu obciążenia. Często także rozwiązania te wymagają centralnej koordynacji, co uniemożliwia wykorzystanie ich w systemach rozproszonych.

Kolejnym problemem podejścia optymistycznego jest obsługa operacji niewycofywalnych (ang. *irrevocable operations*). Operacje niewycofywalne są to operacje, których efekty są obserwowalne, ale nie można ich usunąć ani nie powinno się powielać. Przykładami takich operacji są operacje na zamkach, operacje wejścia/wyjścia, czy komunikacja sieciowa. Operacje te często występują w złożonych aplikacjach i są zazwyczaj trudne do zlokalizowania w kodzie transakcji, ponieważ są wykonywane w ramach procedur będących częściami używanych przez programistę bibliotek. Natomiast jeśli operacja niewycofywalna znajdzie się w kodzie transakcji wykonywanej optymistycznie, to ta transakcja może spowodować wielokrotne wykonanie operacji niewycofywalnej na skutek konfliktu. Przykładowo, spowodować to może wielokrotne wysłanie tej samej wiadomości sieciowej, łamiąc protokół komunikacji, lub wielokrotne pobranie tego samego zamka prowadząc do zakleszczenia transakcji. Rozwiązanie tego problemu w systemach optymistycznych nie jest proste. Popularnym rozwiązaniem jest spowodowanie, że transakcja wykonująca operacje niewycofywalne nie zostaje nigdy wycofana. Przykładowo, w [45] zaproponowano system, w którym transakcja zawierająca operacje niewycofywalne staje się transakcją niewycofywalną, która zawsze wygrywa konflikty z innymi transakcjami. Jednakże tylko jedna taka transakcja może działać jednocześnie w całym systemie ze względu na paradoks w wypadku konfliktu między dwoma niewycofywalnymi transakcjami. Powoduje to jednak zmniejszenie wydajności systemu. W [4, 28] zastosowane inne rozwiązanie: system pamięci transakcyjnej utrzymuje wiele wersji tego samego obiektu, więc transakcje mogą operować na starszych wersjach jeśli operowanie na nowej wersji powodowałoby konflikt. Rozwiązanie to prowadzi do skomplikowania i spowolnienia systemu pamięci transakcyjnej. W konsekwencji, wiele istniejących systemów pamięci transakcyjnej zabrania wykonywania operacji niewycofywalnych (np. Haskell [20]) lub wymaga zdefiniowania operacji, które wykonają kompensacji ich efektów (np. [7]). Są to jednak rozwiązania niepraktyczne, zwłaszcza jeśli system rozproszonej pamięci transakcyjnej ma być zaaplikowany w złożonych systemach zorientowanych na usługi. Przykładowo, jeśli wykonanie operacji na obiekcie-usłudze powoduje efekt materialny (np. wydruk książki), to operacja niewycofywalna jest częścią semantyki usługi i nie istnieje kompensacja, która (bezstrat-

nie) odwróci jej efekt.

1.3 Pesymistyczne Sterowanie Współbieżnością

Prostszą metodą rozwiązywania problemów wymienionych powyżej jest użycie pesymistycznego podejścia do sterowania współbieżnością. Podejście pesymistyczne ma swoje początki w transakcjach bazodanowych (np. blokowanie dwufazowe – ang. *two-phase locking* [6, 44]) i zostało przeniesione do pamięci transakcyjnej w [26, 1, 5] oraz w pracach [46, 47]. Ogólna idea pesymistycznej pamięci transakcyjnej jest taka, że transakcje nie wykonują operacji spekulacyjnie, lecz najpierw sprawdzany jest warunek poprawności wykonania danej operacji, a operacje dla których warunek nie może być natychmiast spełniony ze względu na potencjalny konflikt są opóźniane do momentu, aż konflikt staje się niemożliwy. Oznacza to, że transakcje są wycofywane bardzo rzadko lub wcale, dzięki czemu wyeliminowane zostają problemy związane wyżej wymienionymi scenariuszami z wysokim współzawodnictwem i z operacjami niewycofywalnymi.

W [26] pokazano jednakże, że podejście pesymistyczne w formie stosowanej do tej pory ma negatywny wpływ na wydajność systemów pamięci transakcyjnej, ponieważ jest uzależnione od szeregowego wykonywania transakcji, które wykonują zapisy do zmiennych. Ograniczenie to ma na celu wykluczenia konfliktów, ale powoduje ono ograniczenie równoległości wykonania.

Głównym celem przedstawionej pracy było pokazanie, że obniżenie wydajności nie jest nieodzowną częścią podejścia pesymistycznego i może być całkowicie wyeliminowane. W tym celu, rozważono zastosowanie techniki wczesnego zwalniania zasobów (ang. *early release*). Wczesne zwalnianie zasobów to technika optymalizacyjna stosowana w pamięci transakcyjnej, gdzie pary transakcji między którymi zachodzi konflikt są jednak zatwierdzane [29], jeśli tylko przeplot operacji, który prowadzi do konfliktu jest w istocie poprawny. Technika ta jest szczególnie efektywna w podejściu pesymistycznym, gdzie transakcje nie są wycofywane. Przy takim założeniu, można zezwolić transakcjom na odczytywanie ostatecznego stanu zmiennych modyfikowanych przez inną transakcję bez troski o odczytanie stanu niepoprawnego, pomimo tego, że modyfikująca transakcja nie została jeszcze zatwierdzona. Systemy korzystające z techniki wczesnego zwalniania (np. [21, 29, 16, 7, 36]) pokazują, że powoduje ona znaczącą poprawę efektywności działania pamięci transakcyjnej. Dlatego też w niniejszej pracy wykorzystano tę technikę jako rdzeń zaproponowanych optymalizacji, dążąc do stworzenia bezpiecznego i wydajnego systemu pesymistycznej pamięci transakcyjnej.

1.4 Bezpieczeństwo

W odróżnieniu od transakcji bazodanowych, pamięć transakcyjna pozwala na włączanie dowolnych operacji do kodu transakcji obok odczytów i zapisów na danych współdzielonych. Powoduje to, że pamięć transakcyjna musi wykonywać transakcje bardziej ostrożnie, niż jest to wymagane w bazach danych. Przykładowo, uszeregowalność ustanawia, że jeśli transakcje, które są zatwierdzone, wykonały się poprawnie, to całość wykonania może być uznana za poprawną. Jeśli więc transakcja bazodanowa przeczyta ze zmiennej niespójną wartość, wystarczy, że nie dopuści się tej transakcji do zatwierdzenia, i cały przeplot pozostanie poprawny w świetle uszeregowalności. Natomiast, jeśli pamięć transakcyjna pozwoli transakcji na odczytanie wartości niespójnej, może dojść do złamania jakiegoś niezmiennika i wykonania nieprzewidzianej niebezpiecznej operacji, np. podziału przez zero lub wejścia w pętlę nieskończoną. W takim wypadku, klasyczne własności bazodanowe, takie jak uszeregowalność są niewystarczające dla zapewniania poprawności pamięci transakcyjnej. Pamięć transakcyjna wymaga własności bezpieczeństwa, które

będą ograniczać lub wykluczać możliwość odczytu niespójnego stanu przez transakcje niezależnie od tego, czy będą one ostatecznie zatwierdzone. W tym celu zaproponowano własność nieprzezroczystości, która, ponad wymagania uszeregowalności, wymaga także utrzymania między transakcjami porządku czasu rzeczywistego i uniemożliwia transakcjom czytanie modyfikacji wprowadzonych przez żywe (jeszcze niezatwierdzone) transakcje. Nieprzezroczystość stała się standardową własnością systemów pamięci transakcyjnej i jest *de facto* spełniana przez większość systemów prezentowanych w literaturze.

Jednak, jeśli wykluczyć czytanie modyfikacji wprowadzonych przez niezatwierdzone transakcje, niemożliwym staje się użycie techniki wczesnego zwalniania zasobów, nawet jeśli nie powoduje to niepoprawnych zachowań. Natomiast przed zaproponowaniem wydajnej pamięci transakcyjnej, niezbędnym staje się zdefiniowanie gwarancji poprawności które przez taki system muszą być spełnione i odkrycie lub zdefiniowanie własności bezpieczeństwa które będą te gwarancje określać, jednocześnie dopuszczając użycie wczesnego zwalniania zasobów. Ze względu na fakt, że nieprzezroczystość jest bardzo restrykcyjną własnością, w celach praktycznych w literaturze zaproponowano wiele innych, rozluźnionych własności bezpieczeństwa, takich jak spójność świata wirtualnego (ang. *virtual world consistency – VWC*) [24], specyfikacja pamięci transakcyjnej (ang. *transactional memory specification — TMS1* i *TMS2*) [12], nieprzezroczystość elastyczna (ang. *elastic opacity*) [16], nieprzezroczystość żywa (ang. *live opacity*) [15] i inne. W ramach tej rozprawy dokonano analizy tych własności, a także istniejących własności bazodanowych celem określenia czy pozwalają na optymalizację przez wczesne zwalnianie zasobów, jakie ograniczenia nakładają na tę optymalizację i jakich wymagają dodatkowych założeń. Na podstawie tej analizy wprowadzono nowe własności przeznaczone dla systemów pamięci transakcyjnej z wczesnym zwalnianiem zasobów, które jednocześnie dostarczają silnych gwarancji bezpieczeństwa.

1.5 Model Systemu

Pamięć transakcyjna może być używana w ramach wielu modelów systemu, które wpływają na założenia, jakie algorytm sterowania współbieżnością będzie przyjmować. Po pierwsze, pamięć transakcyjna może działać na zmiennych (obiektach-zmiennych), czyli na obiektach, których stan jest zdefiniowany przez pojedynczą wartość, która z kolei może być odczytana lub nadpisana. Model taki jest typowy dla nierozproszonej pamięci transakcyjnej (np. [11, 18, 29]), ale rozproszona pamięć transakcyjna częściej używa modelu gdzie obiekty współdzielone są bardziej złożone (np. [30, 43]). Konkretnie, wyróżniamy tutaj dwa modele obiektowe: jednorodny (ang. *homogeneus*) i niejednorodny (ang. *heterogeneous*). W modelu jednorodnym obiekty są jednakowe i relatywnie proste: odpowiadają strukturom takim jak liczniki czy stosy. Obiekty współdzielą jeden interfejs, który zawiera jedną operację odczytu i jedną operacje zapisu o znanej semantyce. W modelu niejednorodnym zakłada się, że każdy z obiektów definiuje własny interfejs zawierający dowolne operacje o dowolnej (i zazwyczaj nieznanej *a priori*) semantyce, działające na złożonym, hermetycznie odizolowanym stanie. Różne modele mają różne aplikacje: zmienne znajdują zastosowanie w systemach lokalnych i równoległych o dużej wydajności oraz bazach danych, podczas gdy modele obiektowe używane są w złożonych systemach chmurowych czy architekturach zorientowanych na usługi, gdzie każdy z obiektów wyrażać może nawet całe usługi.

Po drugie, systemy pamięci transakcyjnej mogą dostarczać różnych interfejsów dla samych transakcji. Wiele optymistycznych i pesymistycznych systemów jest systemami wyłącznie zatwierdzającymi (ang. *commit-only*), gdzie transakcja dąży zawsze do tego, żeby po wykonaniu swoich operacji wykonać zatwierdzenie (np. [46, 3, 26]). Alternatywnie, pamięć transakcyjna może być systemem z dowolnością wycofania (ang. *arbitrary abort*),

gdzie transakcja może w dowolnym momencie działania samoistnie wycofać się, zamiast podejmować próby zatwierdzenia. Dodanie operacji wycofania do interfejsu transakcyjnego powoduje, że system staje się bardziej ekspresywny, a także dostarcza istotnej dla wydajnej implementacji odporności na awarie częściowe funkcjonalności.

Warto odnotować, że systemy pamięci transakcyjnej działające na zmiennych mogą przyjąć dużo szersze założenia odnośnie do stanu systemu, niż pamięć transakcyjna działająca na obiektach w pozostałych modelach. Powoduje to, że jeśli porównamy wydajność takich dwóch systemów stosując tylko zmienne, pamięć transakcyjna działająca na zmiennych będzie bardziej wydajna od pamięci obiektowej. Natomiast, jeśli porównamy te dwa systemy używając modelu obiektowego, pamięć transakcyjna działająca na zmiennych ma szanse działać nieprawidłowo ze względu na jej zbyt silne założenia. Podobnie, pamięć transakcyjna przystosowana do dowolnych wycofań może być użyta w modelu wyłącznie zatwierdzającym, chociaż skutkować to będzie obniżeniem wydajności w porównaniu do odpowiednika przystosowanego do pracy w modelu wyłącznie zatwierdzającym. Z drugiej strony, pamięć transakcyjna wyłącznie zatwierdzająca nie może być użyta poprawnie i wydajnie w modelu, w którym transakcje mogą być dowolnie wycofywane. Stypulujemy, że praktyczność systemu pamięci transakcyjnej jest uwarunkowana możliwością jego aplikacji w szerokiej gamie modeli systemów przy zachowaniu wydajności i poprawności. W konsekwencji rozprawa rozważy aplikację wprowadzonych algorytmów w różnych modelach, starając się osiągnąć uniwersalność. Jednocześnie przedstawiamy warianty naszych algorytmów mające na celu poprawę wydajności w konkretnych modelach.

Przyjęto też założenie, że praktyczny system rozproszonej pamięci transakcyjnej nie może opierać się na centralnych strukturach, które wprowadzałyby pojedynczy punkt awarii (ang. *single point of failure*), gdyż ograniczałoby to skalowalność (ang. *scalability*) systemu, oraz uniemożliwiałoby jego funkcjonowanie pomimo częściowych awarii.

1.6 Żywotność i Postęp

Dodatkowo, poza poprawnością systemu, praktyczny system pamięci transakcyjnej powinien także gwarantować, że poszczególne operacje wewnątrz transakcji zostaną wykonane, tj. żywotność (ang. *liveness*), oraz, że każda z transakcji ostatecznie będzie zatwierdzona, tj. postęp (ang. *progress*). Podstawową własności żywotności dla pamięci transakcyjnej jest wolność od zakleszczenia (ang. *deadlock freedom*), która oznacza, że transakcje nigdy nie doprowadzają do zakleszczeń. Zakleszczenie to sytuacja, w której między transakcjami występuje cykl oczekiwania. Silna progresywność (ang. *strong progressiveness*) [17] to popularna własność postępu mówiąca, że konflikt nie może doprowadzić do sytuacji, w której wszystkie skonfliktowane transakcje zostają zmuszone do wycofania. System pamięci transakcyjnej który nie spełnia tych własności żywotności i postępu może doprowadzać do "klinowania się" całego systemu, a więc nie jest systemem praktycznym.

1.7 Teza

W świetle powyższych zamierzeń i wymogów, sformułowano w pracy następującą tezę:

Możliwym jest zaproponowanie pesymistycznego algorytmu sterowania współbieżnością dla rozproszonej pamięci transakcyjnej, który jednocześnie:

a) osiąga wysoką wydajność,

b) spełnia silne własności bezpieczeństwa, żywotności i postępu,

c) gwarantuje poprawne wykonanie operacji niewycofywalnych,

d) jest stosowalny w ogólnych modelach systemów.

1.8 Kontrybucja

Przedstawiona teza jest udowodniona poprzez kontrybucje wprowadzone poniżej i opisane szczegółowo w poszczególnych rozdziałach rozprawy.

I **Analiza istniejących własności i algorytmów pamięci transakcyjnych.**
Formalnie przeanalizowano istniejące własności bezpieczeństwa dla pamięci transakcyjnej oraz bazodanowe warunki spójności celem określenia, czy znajdują one zastosowanie w pamięci transakcyjnej z wczesnym zwalnianiem zasobów. W szczególności, określono, czy pozwalają one na wczesne zwalnianie zasobów, jakie klasy niespójnych odczytów są przez nie dopuszczane i jakie ograniczenia są przez nie nałożone na transakcje. Następnie, zbadano wybrane istniejące pesymistyczne i rozproszone pamięci transakcyjne, oraz pamięci transakcyjne stosujące wczesne zwalnianie zasobów, ustalając ich parametry i gwarancje bezpieczeństwa. Pozwala nam to wyciągnąć wnioski o stosowalności istniejących własności do systemów z wczesnym zwalnianiem zasobów. Ponadto, analiza pozwoliła na ustalenie, które algorytmy i techniki mogą być użyte do implementacji pesymistycznej rozproszonej pamięci transakcyjnej. Analizy przedstawione są w Rozdziałach 3 i 4, i stanowią rozszerzenie wyników zaprezentowanych w [38] i [40].

II **Nowe silne własności bezpieczeństwa dla pamięci transakcyjnej z wczesnym zwalnianiem zasobów.**
W pracy zaproponowano zostały dwie nowe własności bezpieczeństwa, *nieprzezroczystość do ostatniego użycia* i *silna nieprzezroczystość do ostatniego użycia*, które dają silne gwarancje spójności i wykluczają większość klas niespójnych odczytów stanu, jednocześnie pozwalając na wczesne zwalnianie zasobów. Własności wraz z ich charakterystyką przedstawiono oraz omówiono w Rozdziale 5. Własności te zostały wcześniej zaprezentowane w [37, 40].

III **Nowe pesymistyczne algorytmy sterowania współbieżnością dla pesymistycznej (rozproszonej) pamięci transakcyjnej.**
Ponadto w pracy opisano szereg nowych algorytmów sterowania współbieżnością dla pesymistycznej (rozproszonej) pamięci transakcyjnej. Rozpoczęto od rozszerzenia istniejących algorytmów wersjonowania [46, 47] celem wykluczenia pojedynczego punktu awarii i uogólnienia ich do modelu pozwalającego na dowolne wycofywanie transakcji, w efekcie uzyskując algorytmy *BVA+R*, *SVA+R* i *RSVA+R*. Następnie zastosowano szereg daleko idących optymalizacji ze względu na typy wykonywanych operacji, aby uzyskać algorytmy sterowania współbieżnością OptSVA+R i OptSVA-CF+R (wraz z wariantami), które dążą do wysokiego stopnia zrównoleglenia transakcji. Pokazujemy, że owe algorytmy pozwalają na większe zrównoleglenie wykonań transakcji niż ich poprzednicy i wykazujemy ich własności. Algorytmy wprowadzone są w Rozdziale 6, a ich dowody poprawności znajdują się w Rozdziale 7. Nowe algorytmy stanowią rozszerzenie badań zaprezentowanych w [35, 36, 39, 42, 48].

IV **Dowody bezpieczeństwa i techniki dowodzenia.**
Przedstawiono zostały także techniki dowodzenia pozwalające nam na wnioskowanie o bezpieczeństwie (nieprzezroczystość i nieprzezroczystość do ostatniego użycia) algorytmów z wczesnym zwalnianiem zasobów. Następnie zastosowano wprowadzone techniki, udowadniając własności wybranych algorytmów. Zaprezentowane jest to w Rozdziale 7 i stanowi rozszerzenie wyników pokazanych w [40, 41, 48].

V **Implementacja nowych algorytmów.** Efektem pracy były też implementacje systemów rozproszonej pamięci transakcyjnej w modelu przepływu sterowania dla

dwóch z zaprezentowanych algorytmów sterowania współbieżnością. Jedna z implementacji posłużyła do pokazania, że OptSVA-CF+R przewyższa efektywnością wysokiej klasy optymistyczny system rozproszonej pamięci transakcyjnej. Implementacje i ewaluacja są zaprezentowane w Rozdziale 8 i odzwierciedlają one wyniki badań w [36, 39, 42].

VI **Analiza statyczna i prekompilator.** Wprowadzono też prekompilator, który jest w stanie wygenerować informacje wymagane *a priori* przez niektóre z zaprezentowanych algorytmów pamięci transakcyjnej na podstawie analizy statycznej kodu źródłowego transakcji. Prekompilator zaprezentowany jest w Rozdziale 9 i odzwierciedla badania zaprezentowane w [33, 34].

2 Analiza Istniejących Własności

W celu analizy istniejących własności bezpieczeństwa, zarówno dla pamięci transakcyjnej, jak i warunków spójności dla baz danych, zdefiniowano formalnie wczesne zwalnianie zasobów jako scenariusz, gdzie wykonywane są przynajmniej dwie transakcje T_i i T_j w taki sposób, że transakcja T_i zapisuje do jakiejś zmiennej x wartość v, a następnie T_j odczytuje v z x , gdy T_i jest żywa. Mówimy wtedy, że T_i zwalnia x do T_j. Następnie zdefiniowano trzy warunki, które określają do jakiego stopnia jakaś własność $\mathfrak{P}$ pozwala na wczesne zwalnianie zasobów:

Def. 4 *(Dopuszczanie Wczesnego Zwalniania Zasobów)* Własność $\mathfrak{P}$ dopuszcza taką historię wykonania, gdzie występuje wczesne zwalnianie zasobów.

Def. 5 *(Dopuszczanie Nadpisywania)* Własność $\mathfrak{P}$ dopuszcza taką historię wykonania, gdzie występuje wczesne zwalnianie zasobów i gdzie transakcja T_i, która zapisuje wartość v do zmiennej x i zwalnia zmienną x do transakcji T_j, następnie zapisuje ponownie jakąś wartość v' do x po tym, jak T_j odczyta v z x.

Def. 6 *(Dopuszczanie Wycofywania Po Zwolnieniu Zasobów)* Własność $\mathfrak{P}$ dopuszcza taką historię wykonania, gdzie występuje wczesne zwalnianie zasobów i gdzie transakcja T_i, zwalnia jakaś zmienną do transakcji T_j, a następnie T_i ostatecznie zostaje wycofana.

Spośród tych definicji, Def. 4 jest warunkiem koniecznym stosowalności własności dla pamięci transakcyjnej z wczesnym zwalnianiem. Def. 5 służy wykluczeniu własności zbyt permisywnych, które pozwalają na czytanie stanu niespójnego w trakcie działania transakcji. Odczyt takiego stanu może prowadzić do niebezpiecznych sytuacji opisanych w [17]: błędów arytmetyki obiektowej, wejścia w nieskończoną pętlę, dzielenia przez zero. Def. 6 służy zapewnieniu, że nie ogranicza się możliwości wycofania transakcji, które wykonują wczesne zwalnianie zasobów. Takie ograniczenie czyni te transakcje niewycofywalnymi, co niesie ze sobą wiele problemów, np. ograniczenie możliwości wykonywania takich transakcji sekwencyjnie [45]. Co więcej wymuszenie ostatecznego zatwierdzenia w tych transakcjach doprowadza do tego, że w systemach z dowolnym wycofaniem (w szczególności w systemach potencjalnie awaryjnych) wczesne zwalnianie zasobów jest wykluczone całkowicie.

Analizujemy istniejące własności bezpieczeństwa dla pamięci transakcyjnej oraz warunki spójności używane w systemach baz danych pod względem wyznaczonych kryteriów i zaprezentowano wyniki analizy w Tablicy 3. Tablica ta informuje, które z wyżej wymienionych definicji są spełniane przez konkretną własność. Dodatkowo informuje, czy własność wywodzi się z systemów baz danych i czy jest szeroko używana w systemach

Własność	Zastosowanie	Def. 4	Def. 5	Def. 6	⊆Uszeregowalne
Serializability	bazy danych, TM	✓	✓	✓	✓
CO	bazy danych	✓	✓	✓	×
Recoverability	bazy danych	✓	✓	✓	×
Cascadelessness	bazy danych	×	×	×	×
Strictness	bazy danych	×	×	×	✓
Rigorousness	bazy danych	×	×	×	✓
Opacity	TM	×	×	×	✓
Markability	TM	×	×	×	✓
TMS1	TM	×	×	×	✓
TMS2	TM	×	×	×	✓
VWC	TM	✓	×	×	✓
Live opacity	TM	✓	×	×	✓
Elastic opacity	TM	✓	×	×	×

Tablica 1: Podsumowanie dopuszczalności wczesnego zwalniania zasobów w istniejących własnościach.

pamięci transakcyjnej, a także, w ostatniej kolumnie, ustalono czy dana własność jest silniejsza od uszeregowalności – tzn. czy każda historia spełniona przez daną własność spełnia także własność uszeregowalności. W tabeli podano nazwy własności w języku angielskim.

Przedstawiona w pracy analiza wykazuje, że tylko niewielka liczba istniejących własności pozwala na jakiekolwiek wykorzystanie wczesnego zwalniania zasobów. Spośród pozostałych własności, te które pozwalają na wczesne zwalnianie zmiennych są albo zbyt dozwalające, pozwalając na nadpisywanie uprzednio zwolnionej zmiennej, albo zbyt restrykcyjne, wymagając od transakcji zwalniających zmienne, aby były efektywnie niewycofywalne. W konsekwencji można zauważyć brak własności praktycznie stosowalnej w systemach pamięci rozproszonej z wczesnym zwalnianiem zasobów.

3 Analiza Istniejących Algorytmów

Następnym krokiem była analiza istniejących algorytmów sterowania współbieżnością dla pamięci transakcyjnej i systemów pamięci transakcyjnej. Ponieważ badania nad pamięcią transakcyjną zaowocowały dużą liczbą rozwiązań, skupiono się na systemach odzwierciedlających następujące aspekty: rozproszone systemy pamięci transakcyjnej, pesymistyczne pamięci transakcyjne, oraz pamięci transakcyjne wykorzystujące wczesne zwalnianie zasobów. Dla każdej kategorii algorytmów dokonano ogólnego przeglądu istniejących algorytmów, oraz dokładnniejszej analizy wybranych przedstawicieli z każdej kategorii.

Wyniki analizy podsumowano w Tablicy 2. Kolumna *podejście* wskazuje czy algorytm jest pesymistyczny czy optymistyczny. Kolumna *postęp* wskazuje, czy algorytm jest blokujący (oparty na zamkach) czy też jest pozbawiony czekania. Należy odnotować, że SemanticTM nie wymaga czekania, przy założeniu istnienia odpowiedniego modułu szeregującego transakcje przed wykonaniem. Kolumna *modyfikacje* wskazuje, czy algo-

Algorytm	**Podejście**	**Postęp**	**Modyfikacje**	**Wycofania**	**A priori**	**Obiekty**	**Zakleszczenie**	**Bezpieczeństwo**	**Zwalnianie zasobów**	**Operacje niewycofywalne**
B2PL	pesymistyczne	blokujący	p. wykonania	przy zakleszczeniu	∅	dowolne	tak	*serializable*	tak	wycofywane
C2PL	pesymistyczne	blokujący	p. wykonania	bez wycofywania	*RSet*, *WSet*	dowolne	nie	*serializable*	tak	poprawne
S2PL	pesymistyczne	blokujący	p. wykonania	przy zakleszczeniu	∅	dowolne	tak	*strict*	odczyty	wycofywane
R2PL	pesymistyczne	blokujący	p. wykonania	przy zakleszczeniu	∅	dowolne	tak	*rigorous*	nie	wycofywane
CS2PL	pesymistyczne	blokujący	p. wykonania	bez wycofywania	*RSet*, *WSet*	dowolne	nie	*opaque*	odczyty	poprawne
CR2PL	pesymistyczne	blokujący	p. wykonania	bez wycofywania	*RSet*, *WSet*	dowolne	nie	*opaque*	nie	poprawne
CAS2PL	pesymistyczne	blokujący	p. wykonania	dowolne	*RSet*, *WSet*	dowolne	nie	*opaque*	odczyty	wycofywalne przez użytkownika
CAR2PL	pesymistyczne	blokujący	p. wykonania	dowolne	*RSet*, *WSet*	dowolne	nie	*opaque*	nie	wycofywalne przez użytkownika
BVA	pesymistyczne	blokujący	p. wykonania	bez wycofywania	*ASet*	niejednorodne	nie	*opaque*	nie	poprawne
SVA	pesymistyczne	blokujący	p. wykonania	bez wycofywania	*ASet*, suprema	niejednorodne	nie	*serializable*	tak	poprawne
TL2/DTL2	optymistyczne	blokujący	p. zatwierdzenia	przy konflikcie	∅	zmienne	nie	*opaque*	nie	wycofywane
TFA	optymistyczne	blokujący	p. zatwierdzenia	przy konflikcie	∅	jednorodne	nie	*opaque*	nie	wycofywane
MS-PTM	pesymistyczne	blokujący	p. zatwierdzenia	bez wycofywania	∅	zmienne	nie	*opaque*	nie	poprawne
PLE	pesymistyczne	blokujący	p. wykonania	bez wycofywania	∅	zmienne	nie	*opaque*	nie	poprawne
SemanticTM	pesymistyczne	pozbawiony czekania*	p. wykonania	bez wycofywania	*ASet*, zależności	zmienne	nie	*opaque*	nie	powtarzane
DATM	optymistyczne	blokujący	p. zatwierdzenia	przy nadpisaniu, zakleszczeniu i kaskadzie	∅	zmienne	tak	*conflict serializable*	tak	wycofywane

Tablica 2: Podsumowanie algorytmów sterowania współbieżnością dla pamięci transakcyjnej.

rytm wprowadza modyfikacje do obiektów podczas wykonania operacji na obiekcie czy modyfikacje są opóźnione do momentu zatwierdzenia. Kolumna *wycofania* mówi, kiedy transakcje w danym algorytmie wykonują wymuszone wycofanie. W kolumnie *a priori* wskazane są dane wymagane do poprawnego wykonania wykonania transakcji jeszcze przed uruchomieniem. Kolumna *obiekty* mówi w jakim modelu ze względu na definicje obiektów działa algorytm. Warto zauważyć, że algorytmy działające w modelu obiektowym niejednorodnym mogą być użyte również w modelu jednorodnym, a algorytmy działające w modelu jednorodnym mogą być użyte także w modelu ze zmiennymi. Z drugiej strony algorytmy oznaczone jako działające z dowolnymi obiektami są używane w modelu zmiennych, ale mogą być trywialnie zgeneralizowane do dowolnego modelu obiektowego. Kolumna *zakleszczenie* informuje czy algorytm dopuszcza wystąpienie zakleszczenia. Kolumna *bezpieczeństwo* wskazuje, jakie własności bezpieczeństwa są spełnione przez algorytm (podane w języku angielskim). Kolumna *zwalnianie zasobów* mówi czy algorytm wykorzystuje technikę wczesnego zwalniania zasobów. Niektóre algorytmy dopuszczają wczesne zwalnianie przez transakcje zmiennych tylko-do-odczytu. W końcu, kolumna *operacje niewycofywalne* wskazuje jak algorytm radzi sobie z operacjami niewycofywalnymi: czy są one wykonywane zawsze poprawnie, czy mogą wystąpić w wycofanej transakcji lub czy mogą być wykonane wielokrotnie. Rozróżnić należy tutaj wycofanie w sensie ogólnym i wycofanie na życzenie programisty—jeśli programista nakaże transakcji wycofać się (poprzez wywołanie operacji wycofania), wycofanie operacji niewycofywalnej jest zamierzone przez programistę, więc jest zachowaniem poprawnym.

3.1 Własności Pamięci Transakcyjnej z Wczesnym Zwalnianiem

Warto zauważyć, że pomimo istnienia własności bezpieczeństwa, które pozwalają na wczesne zwalnianie zmiennych, takich jak spójność świata wirtualnego (ang. *virtual world consistency*, *VWC*), nieprzezroczystość elastyczna (ang. *elastic opacity*) [16], czy nieprzezroczystość żywa (ang. *live opacity*), to w praktyce algorytmy pamięci transakcyjnej które używają wczesnego zwalniania nie zaspokajają żadnej z nich, a jedynie relatywnie słabą własność uszeregowalności (ang. *serializability*) lub uszeregowalności konfliktowej (ang. *conflict serializability*). Zauważamy, że silniejsze z tych własności nie mogą być użyte ponieważ wymagają one, żeby transakcje, które zwalniają wcześnie zmienne nie mogły się wycofać. Z drugiej strony, zachowania przedstawionych algorytmów bardzo różnią się od siebie. Np. algorytmy z rodziny blokowania dwufazowego (ang. *two-phase locking*, *2PL*) i algorytm wersjonowania w oparciu o suprema (ang. *Supremum Versioning Algorithm*, *SVA*) nie pozwalają na nadpisywanie zmiennej po jej zwolnieniu, podczas gdy DATM na to pozwala. Ponieważ te różnice nie są określone przez własności, wnioskujemy, że brakuje odpowiednich własności bezpieczeństwa dla pamięci transakcyjnej wykorzystującej wczesne zwalnianie zasobów.

3.2 Rozproszona Pamięć Transakcyjna

Spośród przeanalizowanych algorytmów wyszczególniamy algorytmy, które mogą być użyte do implementacji rozproszonej pamięci transakcyjnej: algorytmy blokowania dwufazowego, algorytmy wersjonowania (ang. *versioning algorithms*): BVA i SVA, oraz DTL2 i TFA. Algorytmy te były projektowane ze szczególnym uwzględnieniem systemów rozproszonych lub są w nich wykorzystywane. Spośród tych systemów BVA i SVA używają globalnego zamka celem przyznania każdej uruchamianej transakcji numeru wersji. Jest to problematyczne ze względu na skalowalność systemu, ponieważ wymaga, żeby każdy klient kontaktował się z jednym konkretnym węzłem sieci. Natomiast, jak pokazano poni-

żej, globalny zamek można z tych algorytmów wyeliminować na rzecz bardziej złożonego schematu zamków opartych na wiedzy *a priori* wykorzystywanej przez te algorytmy. Ponadto, spośród tych algorytmów, TFA jest zaprojektowany do pracy w modelu przepływu danych, podczas gdy blokowanie dwufazowe, algorytmy wersjonowania i DTL2 działają w modelu przepływu sterowania.

Algorytmy MS-PTM, PLE i DATM są mniej zdatne do implementacji w środowisku rozproszonym, ponieważ używają globalnych zamków, które są problematyczne ze względu na skalowalność. Ponadto, MS-PTM, PLE i DATM implementują mechanizm bezruchu (ang. *quiescence*), który opóźnia zatwierdzanie transakcji do momentu kiedy wszystkie poprzednie transakcje zakończyły proces modyfikacji zmiennych. Dzięki temu, wszystkie transakcje mogą zakończyć się bezkonfliktowo, ale mechanizm ten wymaga komunikacji pomiędzy procesami obsługującymi transakcje. Komunikacja taka jest niepraktyczna w systemie rozproszonym, gdzie klienci mogą być geograficznie rozproszeni, oddzieleni zaporami ogniowymi (ang. *firewalls*) lub, w przypadku urządzeń mobilnych, mogą nie posiadać odpowiedniej mocy obliczeniowej do obsługi wykrywania bezruchu. W konsekwencji przed wprowadzeniem tych algorytmów do środowiska rozproszonego należy najpierw wprowadzić metody pozwalające transakcjom na wypychanie informacji niezbędnych do działania mechanizmu bezruchu oraz rozproszyć globalne zamki. W przeciwieństwie do algorytmów wersjonowania, wymagane tutaj rozwiązania są nietrywialne.

Trudno jest wyobrazić sobie zastosowanie SemanticTM w środowisku rozproszonym ze względu na wymaganie tego systemu co do porządku wykonywania operacji. SemanticTM wymaga, żeby operacje na zmiennych były umieszczone w kolejkach odpowiednich dla tych zmiennych w taki sposób, że dla dowolnych dwóch transakcji wszystkie operacje pierwszej z nich są umieszczone we wszystkich kolejkach przed operacjami drugiej transakcji lub *vice versa*. Wymaganie to jest trudne do zaspokojenia w systemie rozproszonym, gdzie jego egzekwowanie wymagałoby zastosowania mechanizmu szeregującego równie złożonego jak sama pamięć transakcyjna. Z tego powodu należy uznać SemanticTM za algorytm mało praktyczny w kontekście systemów rozproszonych.

Warto zauważyć, że spośród wymienionych systemów nadających się do implementacji w systemach rozproszonych, tylko blokowanie dwufazowe i algorytmy wersjonowania wspierają operacje niewycofywalne.

4 Nowe Własności

Zarówno wykonany przegląd własności, jak i przegląd algorytmów, sugerują, że brak jest własności dobrze opisujących praktyczne algorytmy sterowania współbieżnością w pamięci transakcyjnej. W konsekwencji, wprowadzono dwie nowe własności, które wypełniają ową niszę.

4.1 Nieprzezroczystość do Ostatniego Użycia

Pierwszą zaprezentowaną własnością jest własność nieprzezroczystości do ostatniego użycia (ang. *last-use opacity*). Własność ta jest oparta o definicję nieprzezroczystości, więc zachowuje ona silne gwarancje bezpieczeństwa. Wyjątkiem jest wymaganie, żeby transakcje zawsze czytały wartości zmiennych które były zapisane przez transakcje uprzednio zatwierdzone. Nieprzezroczystość do ostatniego użycia zezwala na czytanie danych zapisanych przez transakcje niezatwierdzone pod warunkiem, że dane te były zapisane przez ostateczne operacje zapisu (ang. *closing writes*). Ostateczna operacja zapisu to taka ope-

racja, po której nie wystąpi żadna inna operacja zapisu na tej samej zmiennej i będzie to prawdą dla wszystkich potencjalnych wykonań tej transakcji.

Własność nieprzezroczystości do ostatniego użycia mówi, że transakcje, które są zatwierdzone mogą czytać tylko wartości zapisane przez inne transakcje, które są również zatwierdzone. Ponadto, transakcje niezatwierdzone (żywe lub wycofane) mogą czytać wartości zapisane przez inne transakcje, jeśli transakcje te są zatwierdzone lub jeśli wartości te były zapisane przez ostateczną operację zapisu jakiejś niezatwierdzonej transakcji.

Określona w ten sposób własność bezpieczeństwa może być używana do opisu systemów z wczesnym zwalnianiem zasobów, jednocześnie wykluczając nadpisywanie i pozwalając na wycofywanie transakcji, które zwolniły zasoby wcześnie.

Gwarancje

Nieprzezroczystość do ostatniego użycia zapewnia następujące silne gwarancje względem poprawności wykonania kodu transakcyjnego:

Uszeregowalność (ang. *Serializability*) Jeśli transakcja zostanie zatwierdzona, to dowolna wartość przez nią odczytana może być wyjaśniona przez operacje poprzedzających lub współbieżnych zatwierdzonych transakcji. Transakcje czytające stan niespójny nie zostaną zatwierdzone.

Porządek Czasu Rzeczywistego (ang. *Real-time Order*) Kolejne transakcje nie będą zamieniane kolejnością celem zaspokojenia uszeregowalności, wiec poprawny przeplot będzie odpowiadać zewnętrznemu względem systemu zegarowi.

Odzyskiwalność (ang. *Recoverability*) Jeśli transakcja odczyta wartość zapisaną przez drugą transakcję, to pierwsza z nich zostanie zatwierdzona dopiero po tym, jak druga z nich zostanie zatwierdzona.

Wykluczenie Nadpisywania (ang. *Precluding Overwriting*) Jeśli transakcja odczytuje wartość zapisaną do jakiejś zmiennej przez drugą transakcję, to ta druga transakcją nie wykona ponownie zapisu do tejże zmiennej.

Wycofywanie po Wczesnym Zwolnieniu (ang. *Aborting Early Release*) Transakcja, która wykonała wczesne zwolnienie zasobu, może następnie zostać wycofana.

Wyłączny Dostęp (ang. *Exclusive Access*) Transakcja ma wyłączny dostęp do danej zmiennej, od chwili wykonania pierwszej operacji na tej zmiennej i co najmniej do momentu ostatecznego zapisu na niej.

między pierwszą operacją, którą wykona na tej zmiennej, i minimum do ostatecznej modyfikacji którą wykonuje na tej zmiennej.

Niespójny Odczyt Systemu

Nieprzezroczystość do ostatniego użycia nie wyklucza sytuacji, gdzie transakcja odczytuje wartość zapisaną przez inną transakcje, która to następnie zostanie wycofana. Rezultatem takiej sytuacji jest odczyt niespójnego stanu systemu przez pierwszą z transakcji, co może mieć różne implikacje dla poprawności działania programu w zależności od modelu systemu.

W modelu systemu z transakcjami wyłącznie zatwierdzającymi transakcje nie mogą samoczynnie zostać wycofane – mogą jednak zostać wycofane siłowo, np. w konsekwencji konfliktu. Ponieważ nieprzezroczystość do ostatniego użycia wyklucza odczyt danych zapisanych przez transakcje przed ostatecznym zapisem do danej zmiennej, transakcja wycofana nie zapisywałaby do tej zmiennej jakiejkolwiek innej wartości. W konsekwencji, wartość zapisana przez tę transakcje byłaby identyczna niezależnie, czy transakcja ostatecznie wykonałaby zatwierdzenie czy wycofanie. Dlatego też można uważać stan

zaobserwowany przez inne transakcje za bezpieczny. Innymi słowy, jeśli wartość odczytana z wycofanej transakcji spowodowałaby błąd, wystąpiłby on niezależnie od tego, czy ta transakcja została wycofana czy zatwierdzona. W takim wypadku, programista ma gwarancję, że odczyt technicznie niespójnego stanu nie wprowadzi niepoprawnego zachowania do systemu. Warto zauważyć, że taki model systemu jest powszechnie używany (np. [16, 2, 3]).

Alternatywnym jest model z transakcjami dowolnie wycofywanymi. W takim modelu transakcja może wykonać operację wycofania dowolnie, także w ramach swojej „logiki biznesowej". W takim wypadku możliwym jest, że transakcja będzie używała operacji wycofania właśnie do naprawieniu stanu systemu po zapisaniu do zmiennych niespójnych, niebezpiecznych wartości. Przykładowo, transakcja T_i może zapisać do jakiejś zmiennej x wartość v, po czym, jeśli v łamie predefiniowany warunek niezmienny, wykonać wycofanie. Natomiast, jeśli zapisanie do x wartości v będzie ostatecznym zapisem, x może zostać odczytane przez inną transakcję T_j, jeszcze zanim T_i wykona wycofanie. W takim wypadku T_j może podjąć niebezpieczne działanie na podstawie wartości v. W konsekwencji, używanie modelu dowolnego wycofania może powodować niebezpieczne zachowania. Zachowań tych można uniknąć, np. nigdy nie wprowadzając do zmiennych wartości łamiących niezmienniki, lub przesuwając potencjalne wykonanie wycofania przed operację ostateczną. Jeśli rozwiązania te są niewystarczające, poniżej przedstawiono silniejszą wersję własności nieprzezroczystości do ostatniego użycia, która wyklucza niespójny odczyt stanu systemu.

Wprowadzano także w pracy alternatywny wariant powyższego modelu, który nazwano modelem z ograniczonym wycofaniem (ang. *restricted abort model*). W modelu tym zakładamy, że transakcje mogą wykonać operację wycofania dowolnie, ale operacja ta jest umieszczana w kodzie transakcji automatycznie przez niezależne od transakcji zdarzenia, np. awarie lub przerwania, a nie są one częścią logiki biznesowej transakcji. Jeśli programista nie może związać logiki transakcji z operacją wycofania, własność daje takie same gwarancje względem niespójnego stanu jak w modelu z transakcjami dążącymi do zatwierdzenia.

4.2 Silna Nieprzezroczystość do Ostatniego Użycia

Własność silnej nieprzezroczystości do ostatniego użycia (ang. *strong last-use opacity*) jest odmianą własności nieprzezroczystości do ostatniego użycia, która zachowuje się podobnie, ale wykorzystuje inną definicję operacji ostatecznych. Silnie ostateczna operacja zapisu to taka operacja, po której nie wystąpi żadna inna operacja zapisu na tej samej zmiennej ani operacja wycofania i będzie to prawda dla wszystkich potencjalnych wykonań tej transakcji. Silna nieprzezroczystość do ostatniego użycia mówi, że transakcje, które są zatwierdzone, mogą czytać wartości zapisane przez inne transakcje, które są również zatwierdzone. Transakcje niezatwierdzone natomiast mogą czytać wartości zapisane przez inne transakcje, o ile transakcje te są zatwierdzone, lub jeśli odczytywane wartości były zapisane przez silnie ostateczną operację zapisu transakcji niezatwierdzonej.

Zdefiniowana w ten sposób własność daje te same własności co nieprzezroczystość do ostatniego użycia, a ponadto wyklucza negatywne konsekwencje niespójnego stanu we wszystkich modelach. Wiążę się to jednak z wykluczeniem potencjalnie poprawnych wykonań. Oznacza to także, że w systemach, gdzie operacja wycofania może wystąpić w dowolnym momencie (np. ze względu na awarie), wczesne zwalnianie zasobów jest całkowicie wykluczone.

Własność	Zast.	Def. 4	Def. 5	Def. 6	⊆Uszeregowalne
Last-use opacity	TM	✓	×	✓	✓
Strong last-use opacity	TM	✓	×	✓	✓

Tablica 3: Podsumowanie dopuszczalności wczesnego zwalniania zasobów we wprowadzonych własnościach.

5 Nowe Algorytmy

W pracy wprowadzono nowe algorytmy pesymistycznego sterowania współbieżnością dla pamięci transakcyjnej, które kierujemy w szczególności do systemów rozproszonej pamięci transakcyjnej, w których występować mogą operacje niewycofywalne. Nowe algorytmy opierają się o rodzinę algorytmów wersjonowania, a w szczególności algorytm SVA. Algorytmy wersjonowania są pesymistyczne i pozbawione wycofań, więc nie powodują błędnych wykonań operacji niewycofywalnych. Dodatkowo SVA wykorzystuje mechanizm wczesnego zwalniania zmiennych, który może być użyty do wykonania wchodzących w konflikt transakcji częściowo równolegle, co pozwala na krótsze przeploty, niż np. algorytmy blokowania dwufazowego.

W poniższych opisach algorytmów użyto notacji x, y, z kiedy odnosimy się do zmiennych, natomiast obiekty (zarówno w modelu obiektów jednorodnych, jak i niejednorodnych) oznaczono dla odróżnienia jako $\lceil x \rfloor, \lceil y \rfloor, \lceil z \rfloor$.

5.1 SVA

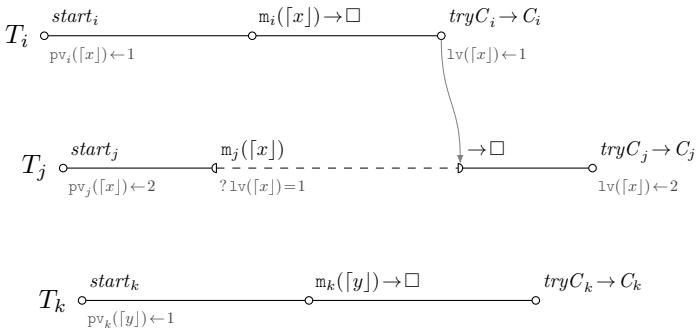
Algorytm SVA [46, 47] jest podstawowym algorytmem wersjonowania, na którym oparto algorytmy zaproponowane w rozprawie. Algorytmy wersjonowania używają liczników wersji w celu ustalenia, czy dana transakcja może w bieżącym momencie wykonać operację na konkretnym obiekcie współdzielonym czy owa operacja musi być opóźniona celem uniknięcia konfliktu.

Mechanizm Wersjonowania

Intuicyjnie, liczniki te działają przez analogię do zarządzania kolejką w banku: klienci którzy przychodzą do banku pobierają numerek z automatu i czekają z podejściem do okienka aż ich numerek zostanie wywołany. W analogii tej klient jest transakcją, okienko obiektem współdzielonym, a numerek pobrany przez klienta to wersja transakcji dla tego okienka, która jest porównywana z wywoływanym numerkiem—wersją obiektu.

Konkretnie, za każdym razem, kiedy transakcja T_i jest uruchamiana, pobiera ona *prywatną wersję* $\mathtt{pv}_i(\lceil x \rfloor)$ dla każdego obiektu współdzielonego $\lceil x \rfloor$, na którym transakcja będzie wykonywać operacje (zbiór tych obiektów jest znany z góry). Wartości wersji prywatnych dla obiektu $\lceil x \rfloor$ nadawane kolejnym transakcjom są kolejnymi liczbami naturalnymi. Dodatkowo, wartości te nadawane są atomowo w taki sposób, że mając dwie transakcje T_i, T_j, jeśli dla jakiegoś obiektu $\lceil x \rfloor$, $\mathtt{pv}_i(\lceil x \rfloor) < \mathtt{pv}_j(\lceil x \rfloor)$, to dla każdego innego obiektu $\lceil y \rfloor$, na którym obie transakcje będą wykonywać operacje, $\mathtt{pv}_i(\lceil y \rfloor) < \mathtt{pv}_j(\lceil y \rfloor)$. W celu zapewnienia tego warunku SVA używa globalnego zamka, który szereguje operacje rozpoczęcia wykonywane przez wszystkie transakcje.

Po pobraniu wersji, SVA używa wersji prywatnych do podjęcia decyzji, czy transakcja T_i może wykonać operacje na obiekcie $\lceil x \rfloor$, porównując $\mathtt{pv}_i(\lceil x \rfloor)$ z *wersją lokalną* obiektu $\mathtt{lv}(\lceil x \rfloor)$. Wersja lokalna obiektu jest równa wartości wersji prywatnej transakcji,



Rys. 1: Mechanizm wersjonowania w SVA.

która jako ostatnia tego używała obiektu i już zakończyła jego używanie (np. już została zatwierdzona). W takim wypadku, biorąc pod uwagę, że transakcje mają kolejne wartości wersji prywatnej, transakcja T_i może wykonywać operacje na $\lceil x \rfloor$ wtedy, gdy jej wartość wersji prywatnej dla $\lceil x \rfloor$ jest kolejna względem wartości wersji lokalnej $\lceil x \rfloor$, tj. $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor)$. Warunek ten nazwano *warunkiem dostępu*.

Kiedy transakcja zakończyła wykonywanie wszystkich operacji, wykonuje ona operację zatwierdzenia, kiedy to dla wszystkich obiektów, dla których pobrała wersje prywatne, zapisuje swoją wartość wersji prywatnej dla tego obiektu do licznika wersji lokalnej tego obiektu. W kontekście algorytmów wersjonowania nazywa się to *zwolnieniem obiektu* $\lceil x \rfloor$.

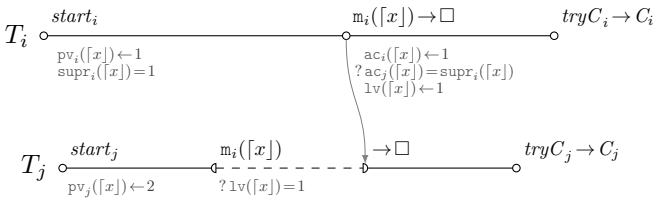
Na Rys. 1 pokazano przykład działania mechanizmu wersji. Tutaj T_i i T_j próbują wykonać operacje na obiekcie $\lceil x \rfloor$ w tym samym czasie. Transakcja T_i startuje wcześniej, więc $\mathtt{pv}_i(\lceil x \rfloor) = 1$, natomiast T_j stratuje jako druga, więc $\mathtt{pv}_j(\lceil x \rfloor) = 2$. Ponieważ początkowo $\mathtt{lv}(\lceil x \rfloor) = 0$, to T_j nie jest w stanie wykonać operacji na $\lceil x \rfloor$, więc czeka. Z kolei dla T_i warunek $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor)$ jest prawdziwy, więc T_i wykonuje operację (metodę) $\mathtt{m}$ na $\lceil x \rfloor$ bez czekania (zwracana wartość nie jest istotna, więc jest oznaczona $\square$). Kiedy T_i zostanie zatwierdzona, ustawia $\mathtt{lv}(\lceil x \rfloor)$ na $\mathtt{pv}_i(\lceil x \rfloor)$ czyli 1, co powoduje, że T_j spełni teraz warunek $\mathtt{pv}_j(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor)$ i przejdzie do wykonania operacji na $\lceil x \rfloor$. W międzyczasie T_k może wykonać operację na $\lceil y \rfloor$ zupełnie równolegle.

Mechanizm wersjonowania zapewnia transakcjom wyłączny dostęp do obiektów, jednocześnie powodując, że transakcje o rozłącznym zbiorach obiektów, na których wykonują operacje, nie blokują się nawzajem.

Wczesne Zwalnianie Zasobów

Dodatkowo, SVA używa mechanizmu wczesnego zwalniania zasobów opartego o *suprema*. Supremum dla obiektu $\lceil x \rfloor$ (oznaczone $\mathtt{supr}_i(\lceil x \rfloor)$) w transakcji T_i to liczba informująca jaka jest maksymalna liczba wykonań operacji na $\lceil x \rfloor$ przez transakcję T_i. Jeśli supremum jest zdefiniowane *a priori* w transakcji T_i, to transakcja liczy wywołania operacji na obiekcie $\lceil x \rfloor$ za pomocą licznika $\mathtt{ac}_i(\lceil x \rfloor)$, i jeśli po wywołaniu operacji na $\lceil x \rfloor$ liczba wywołań jest równa supremum, to obiekt $\lceil x \rfloor$ zostaje zwolniony przez zapisanie wartości wersji prywatnej transakcji do wersji lokalnej obiektu. Dzięki temu inna transakcja może zacząć wykonywać operacje na tym obiekcie jeszcze zanim T_i zostanie zatwierdzona. Z drugiej strony, ponieważ T_i osiągnęła supremum dla $\lceil x \rfloor$, to nie wykona kolejnych operacji na $\lceil x \rfloor$.

Przykład przeplotu wygenerowanego przez SVA z użyciem wczesnego zwalniania zmiennych jest pokazany na Rys. 2. Tutaj transakcje T_i i T_j wykonują operacje na $\lceil x \rfloor$. Ponieważ wersja prywatna T_i dla $\lceil x \rfloor$ jest niższa niż w przypadku T_j, ta pierwsza wykonuje swoje operacje na $\lceil x \rfloor$ jako pierwsza, a T_j czeka aż $\lceil x \rfloor$ będzie zwolniony. Tutaj natomiast T_i zna



Rys. 2: Przykład wczesnego zwalniania zmiennych w SVA.

swoje supremum dla $\lceil x \rfloor$, t.j. $\mathtt{supr}_i(\lceil x \rfloor) = 1$. Więc wykonawszy swoją operację na $\lceil x \rfloor$, T_i zwiększa $\mathtt{ac}_i(\lceil x \rfloor)$, co oznacza, że supremum zostało osiągnięte, tj. $\mathtt{ac}_i(\lceil x \rfloor) = \mathtt{supr}_i(\lceil x \rfloor)$. W takim wypadku T_i zwalnia $\lceil x \rfloor$ od razu, zamiast czekać do momentu zatwierdzenia. W rezultacie, T_j może wykonywać operacje na $\lceil x \rfloor$ wcześniej. Transakcja T_j może nawet być zatwierdzona przed T_i.

Mechanizm wczesnego zwalniania zasobów przez suprema pozwala transakcjom korzystającym z tych samych obiektów na wykonywanie się z większym współczynnikiem zrównoleglenia, niż w wypadku samego mechanizmu wersjonowania, jednocześnie zapewniając, że odczytany z niezatwierdzonych transakcji stan zawsze będzie prawidłowy. Pozwala to na efektywne i poprawne przeplatanie transakcji.

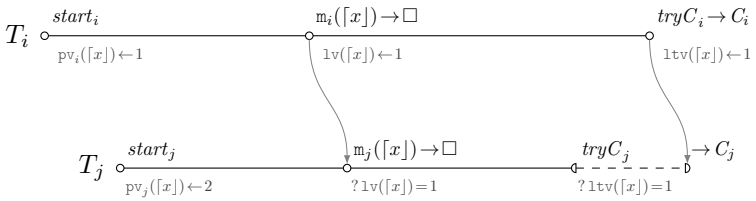
Własności

W ramach pracy zademonstrowano gwarancje bezpieczeństwa SVA. Konkretnie, wprowadzono dekompozycję – technikę dowodzenia opartą na rafinacji obserwacyjnej (ang. *observational refinement*) która pozwala pokazać, że dany przeplot jest nierozróżnialny od poprawnego przeplotu nieprzezroczystego pod względem efektów wykonania operacji. W ten sposób demonstrujemy, że mimo tego, że SVA nie jest nieprzezroczystym algorytmem, to każdy przeplot wygenerowany przez SVA jest nierozróżnialny od poprawnego przeplotu nieprzezroczystego (Twierdzenie 7).Ponadto, SVA nie powoduje zakleszczeń oraz zapewnia silną progresywność.

5.2 Rozproszone Pobieranie Wersji

Celem zapewnienia atomowości pobierania wersji przy starcie transakcji algorytm SVA używa globalnego zamka, który każda transakcja pobiera na początku swojej procedury inicjalizacji i zwalnia na jej końcu. Zamek ten rodzi problem, jeśli algorytm ma być zaimplementowany w środowisku rozproszonym, ponieważ globalny zamek stanowi przeszkodę dla skalowalności—niezależnie od tego, ile nowych węzłów dodamy do systemu, zawsze wszyscy klienci muszą kontaktować się z jednym węzłem, który jest odpowiedzialny za globalny zamek. Dodatkowo, jeśli węzeł, na którym znajduje się zamek, ulegnie awarii, to taka awaria (częściowa) paraliżuje cały system.

Aby wyeliminować problemy związane z globalnym zamkiem, zaprezentowano w pracy wariant SVA, który używa rozproszonych zamków w procedurze rozpoczęcia transakcji. Wariant ten wymaga, żeby z każdym obiektem współdzielonym $\lceil x \rfloor$ związany był zamek $\mathtt{lk}(\lceil x \rfloor)$ (zlokalizowany na tym samym węźle co $\lceil x \rfloor$). Wtedy każda transakcja, zamiast pobierać globalny zamek przed pobraniem wersji, będzie pobierać zbiór zamków związanych z obiektami, na których transakcja planuje wykonywać operacje. Oznacza to, że transakcje, które nie współdzielą obiektów, mogą wykonywać procedurę rozpoczęcia równolegle, a także, że nie ma pojedynczego zamka, który musi obsługiwać wszystkie transakcje.



Rys. 3: Wymuszenie porządku zatwierdzania transakcji w SVA+R.

Dodatkowo, celem uniknięcia zakleszczeń wymagamy, żeby zamki były zawsze pobierane w według globalnie ustalonego porządku. Jeśli ten porządek jest zachowany, nie mogą wystąpić cykle oczekiwań, a więc zakleszczenia są wykluczone. Jest to prostsze rozwiązanie niż np. istniejące rozwiązanie w konserwatywnych algorytmach blokowania dwufazowego, gdzie unikanie zakleszczeń zaimplementowane jest przez cykliczne odpytywanie o stan zamków.

5.3 SVA+R

SVA jest algorytmem działającym w modelu transakcji wyłącznie zatwierdzających (a nawet wykonania pozbawione są całkowicie wycofań). Natomiast, jeśli algorytm pamięci transakcyjnej ma być praktyczny w dowolnym systemie, powinien on wspierać operację wycofania, a więc działać w modelu dowolnych wycofań. Wprowadzono więc nowy algorytm wersjonowania SVA+R (ang. *SVA with rollback*), który rozszerza SVA o operację wycofania. Wymaga to wprowadzenia dodatkowych mechanizmów opisanych poniżej.

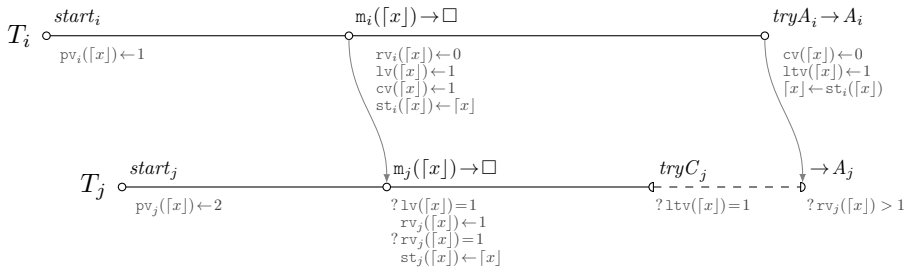
Odtwarzanie Obiektów

Po pierwsze, wprowadzono bufor $\mathtt{st}_i(\lceil x \rfloor)$, który transakcja T_i utrzymuje dla każdego obiektu $\lceil x \rfloor$, na którym wykonuje operacje. Obiekt $\lceil x \rfloor$ jest kopiowany do bufora $\mathtt{st}_i(\lceil x \rfloor)$ w momencie kiedy transakcja po raz pierwszy spełni warunek dostępu do obiektu $\lceil x \rfloor$, czyli tuż przed wykonaniem pierwszej operacji na $\lceil x \rfloor$. Bufor jest następnie używany wewnątrz samej procedury wycofania transakcji: transakcja przed zwolnieniem obiektu $\lceil x \rfloor$ przywróci go do wcześniejszej postaci, kopiując zawartość $\mathtt{st}_i(\lceil x \rfloor)$ z powrotem do $\lceil x \rfloor$.

Porządek Zatwierdzania

Druga modyfikacja ma na celu zapobieganie sytuacjom, gdzie transakcja T_i zwolni obiekt $\lceil x \rfloor$, pozwalając innej transakcji T_j na odczytanie jego stanu, a następnie T_j wykona zatwierdzenie, zanim T_i zostanie zatwierdzona. Jest to problematyczne, ponieważ istnieje możliwość, ze T_i ostatecznie zostanie wycofana, co powoduje, że T_j została zatwierdzona wykonawszy operacje na już-niespójnym stanie.

W tym celu SVA+R porządkuje wykonania operacji zakończenia transakcji: wycofań i zatwierdzeń, w taki sam sposób jak dostępy do obiektów. W związku z tym wprowadzono *lokalną końcową wersję* obiektu (ang. *local terminal version*) oznaczoną $\mathtt{ltv}(\lceil x \rfloor)$ dla obiektu $\lceil x \rfloor$. Wersja ta działa podobnie jak $\mathtt{lv}(\lceil x \rfloor)$, ale transakcje zapisują tam swoją wersję prywatną tylko wtedy, gdy zakończyły wykonywać zatwierdzenie lub wycofanie. Dodatkowo, każda transakcja T_i przed wykonaniem zatwierdzenia lub wycofania czeka, aż warunek $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{ltv}(\lceil x \rfloor)$ stanie się prawdziwy. W konsekwencji, jeśli transakcja T_j będzie operować na obiekcie zmodyfikowanym przez T_i, to zatwierdzenie T_j będzie opóźnione tak, żeby wystąpiło po zatwierdzeniu lub wycofaniu T_i. Przykład takiej sytuacji jest pokazany na Rys. 3.



Rys. 4: Wykrycie niespójnego stanu systemu w SVA+R.

Kaskadowe Wycofania

Trzecia modyfikacja to rozwiązanie sytuacji gdzie transakcja T_i zwolni obiekt $\lceil x \rfloor$, pozwalając innej transakcji T_j na odczytanie jego stanu, a następnie T_i zostanie wycofana. Powoduje to, że T_j operuje na technicznie niespójnym stanie, więc musi także zostać wycofana.

SVA+R rozpoznaje tą sytuację i zmusza T_j do wykonania wycofania. Wykrywanie niespójnego stanu jest osiągnięte za pomocą dwóch liczników: wersji obecnej (ang. *current version*) oznaczonej $\mathtt{cv}(\lceil x \rfloor)$ dla obiektu $\lceil x \rfloor$ i wersji naprawczej (ang. *recovery version* oznaczonej $\mathtt{rv}_i(\lceil x \rfloor)$ dla obiektu $\lceil x \rfloor$ i transakcji T_i. Wersja obecna wyznacza najnowszą spójną wersję obiektu, podczas gdy wersja naprawcza oznacza ostatnią spójną wersję tego obiektu, która została zaobserwowana przez daną transakcje. Transakcje przypisują coraz większe wartości wersji obecnej obiektu w momencie gdy transakcje te są zatwierdzane lub zwalniają obiekt wcześnie (na podstawie swojej wersji prywatnej). Natomiast, jeśli transakcja jest wycofywana, to wartość wersji obecnej obiektu jest cofana do wartości mniejszej (na podstawie wersji naprawczej transakcji). Z kolei wartość wersji naprawczej transakcji jest pobierana z wersji obecnej, kiedy transakcja po raz pierwszy spełni warunek dostępu do obiektu $\lceil x \rfloor$. Można więc zauważyć, że jeśli transakcja zaobserwowała jakąś wartość wersji obecnej obiektu, która została zapisana w liczniku wersji naprawczej tej transakcji, a następnie jakaś wcześniejsza transakcja została wycofana i zmniejszyła wartość wersji obecnej, to $\mathtt{rv}_i(\lceil x \rfloor) > \mathtt{cv}(\lceil x \rfloor)$. W konsekwencji transakcje SVA+R sprawdzają warunek $\mathtt{rv}_i(\lceil x \rfloor) > \mathtt{cv}(\lceil x \rfloor)$ przed wykonaniem dowolnej operacji oraz zatwierdzenia dla każdej zmiennej, dla której pobrana została wersja naprawcza. Jeśli warunek jest prawdziwy, to wiadomo, że transakcja działa na niespójnym stanie, więc będzie zmuszona do wykonania wycofania zamiast wykonania zamierzonej operacji.

Pokazano przykład takiego scenariusza na Rys. 4. Transakcje T_i i T_j wykonują operacje na obiekcie $\lceil x \rfloor$ i mają przyznane wersje prywatne dla $\lceil x \rfloor$ równe odpowiednio 1 i 2. W konsekwencji, T_i wykonuje operację na $\lceil x \rfloor$ jako pierwsza. W tym momencie T_i ustawia $\mathtt{rv}_i(\lceil x \rfloor)$ na obecną wartość $\mathtt{cv}(\lceil x \rfloor) = 0$ (wartość początkowa z założenia) i kopiuje obiekt do bufora. Następnie T_i wykonuje operację na $\lceil x \rfloor$ i zwalnia $\lceil x \rfloor$ (na podstawie supremum), przez co T_i także podnosi $\mathtt{cv}(\lceil x \rfloor)$ do wartości swojej wersji prywatnej dla $\lceil x \rfloor$, czyli 1. Następnie T_j spełnia warunek dostępu do $\lceil x \rfloor$ po raz pierwszy, więc kopiuje $\lceil x \rfloor$ do bufora i ustawia $\mathtt{rv}_j(\lceil x \rfloor)$ na obecną wartość $\mathtt{cv}(\lceil x \rfloor) = 1$. Skoro $\mathtt{rv}_i(\lceil x \rfloor) = \mathtt{cv}(\lceil x \rfloor)$, to zezwala się na wykonanie operacji. Następnie transakcja T_j próbuje wykonać zatwierdzenie, ale nie może tego zrobić, ponieważ T_i nie została zatwierdzona, dlatego T_j czeka. W międzyczasie transakcja T_i zostanie arbitralnie wycofana. Powoduje to, że transakcja zapisuje swoją wersję naprawczą z powrotem do wersji obecnej obiektu $\mathtt{cv}(\lceil x \rfloor) = 0$ i obiekt zostaje przywrócony z bufora. Wówczas, jeśli transakcja T_j wykona jakąkolwiek operacje lub spróbuje się zatwierdzić, to prawdziwy będzie warunek $\mathtt{rv}_j(\lceil x \rfloor) > \mathtt{cv}(\lceil x \rfloor)$, co spowoduje, że transakcja T_j zostanie forsownie wycofana.

Własności

SVA+R nie powoduje zakleszczeń, oraz zapewnia silną progresywność, tak jak SVA. Jednak w przeciwieństwie do SVA, nie można pokazać, że SVA+R będzie generował nieodróżnialne przeploty od przeplotów nieprzezroczystych. Jest tak, ponieważ może się zdarzyć, że transakcja odczyta stan innej transakcji, która to zostanie ostatecznie wycofana. Natomiast, jak pokazuje Twierdzenie 8, SVA+R spełnia własność nieprzezroczystości do ostatniego użycia.

5.4 RSVA+R

Ponieważ wprowadzenie mechanizmu dowolnego wycofywania transakcji do algorytmu powoduje, że SVA+R pozwala na kaskadowe wycofania i niespójne widoki, to występuje problem w kontekście operacji niewycofywalnych. To znaczy, może dojść do sytuacji, gdzie transakcja odczyta wartość obiektu zwolnionego wcześniej przez wcześniejszą transakcję i wycofanie wcześniejszej z transakcji spowoduje wycofanie obu. Jeśli którakolwiek transakcja zawierała operacje niewycofywalne, to są one w takim przypadku obsłużone niepoprawnie. W szczególności, transakcja zmuszona do wycofania na skutek wycofania innej, zwalniającej obiekty wcześnie transakcji nie jest w stanie takiego wycofania przewidzieć.

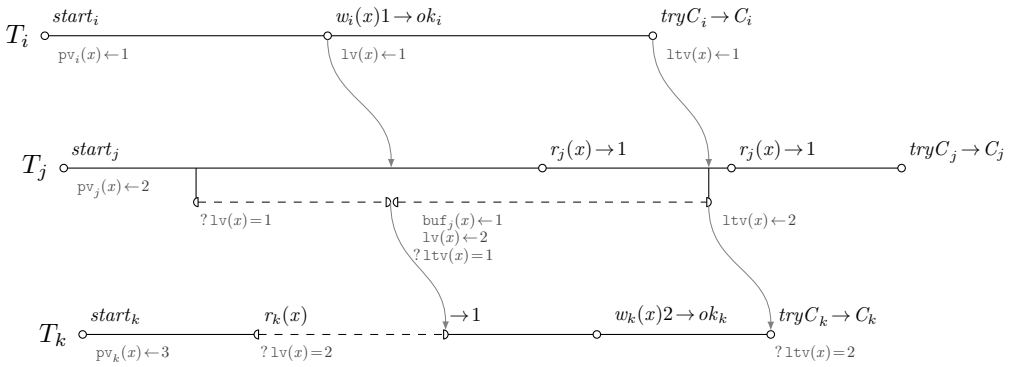
Celem zażegnania tego problemu wprowadzono w pracy wariant algorytmu SVA+R nazwany RSVA+R, który pozwala na zdefiniowane klasy transakcji *niechętnych* (ang. *reluctant*), które nigdy nie są wycofywane. Jest tak dlatego, że forsowne wycofanie transakcji w SVA+R wymaga, żeby transakcja operowała na zmiennej zwolnionej wcześnie przez inną transakcję która zostanie ostatecznie wycofana. Można tę sytuację wykluczyć, jeśli transakcja nie przyjmie zmiennej zwolnionej wcześniej, lecz poczeka, aż wcześniejsza transakcja wykona zatwierdzenie lub się wycofa. Transakcję, która nie przyjmuje zmiennych zwolnionych wcześniej nazywamy właśnie transakcją niechętną, a implementacja tej transakcji zmienia warunek dostępu do zmiennych z $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{lv}(\lceil x \rfloor)$ na warunek $\mathtt{pv}_i(\lceil x \rfloor) - 1 = \mathtt{ltv}(\lceil x \rfloor)$. Jeśli transakcja niechętna nigdy nie jest zmuszona do wycofania, to jest ona podobna do transakcji niewycofywalnej, ale może ona jednocześnie wykonać operację wycofania samodzielnie.

RSVA+R umożliwia doprowadzenie do poprawnego wykonania transakcji z operacjami niewycofywalnymi. Rozwiązanie to wprowadza *trade-off* między poprawnością wykonania operacji niewycofywalnych i wydajnością systemu. Jeśli zbiór transakcji niechętnych jest duży, to system pamięci transakcyjnej ma mało szans na zrównoleglenie wykonań skonfliktowanych transakcji. W konsekwencji transakcje skonfliktowane niechętne będą często wykonywane sekwencyjnie. Warto zauważyć, że w przeciwieństwie do rozwiązań z [45], które wymagają, żeby transakcje niewycofywalne były wykonywane pojedynczo, RSVA+R mimo wszystko pozwala na wykonanie niechętnych nieskonfliktowanych transakcji równolegle. Z drugiej strony, jeśli zbiór transakcji niechętnych jest niewielki, to algorytm RSVA+R zapewnia jednocześnie w pełni poprawne wykonanie operacji niewycofywalnych oraz wysoki stopień zrównoleglenia transakcji.

RSVA+R spełnia te same własności co SVA+R.

5.5 OptSVA+R

Algorytmy wersjonowania zaprezentowane do tej pory działają w oparciu o model obiektów niejednorodnych oraz przy założeniu, że semantyka operacji wykonywanych na obiektach jest zastrzeżona lub ulega zmianom dynamicznie w trakcie działania systemu. W takich systemach traktowanie wszystkich operacji konserwatywnie jako potencjalny odczyt i modyfikacja jest praktycznym uniwersalnym rozwiązaniem.



Rys. 5: Optymalizacja tylko-do-odczytu w OptSVA+R.

Z drugiej strony, w systemach takich jak rozproszone magazyny danych czy nierozproszone systemy transakcyjne częściej wykorzystuje się prostsze obiekty jednorodne lub zmienne. W takich systemach algorytmy wersjonowania są dużo mniej wydajne niż alternatywne pamięci transakcyjne, ponieważ nie wprowadzają one optymalizacji generowanych przeplotów na podstawie semantyki operacji. Przykładowo, dwie operacje odczytu zmiennej we współbieżnych transakcjach mogą być zawsze wykonane równolegle względem siebie, podczas gdy warianty SVA+R zawsze wykonają je sekwencyjnie.

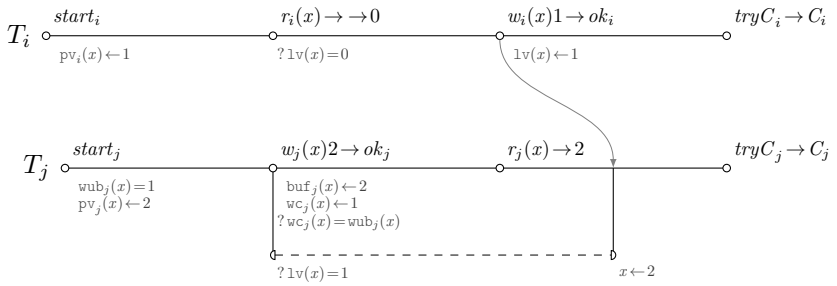
W konsekwencji zaprezentowano algorytm OptSVA+R, który rozszerza algorytm SVA+R i sprowadza go do modelu zmiennych, wykorzystując wiedzę odnośnie semantyki operacji i uproszczoną definicję stanu obiektów w celu wprowadzenia optymalizacji mających na celu maksymalne zrównoleglenie skonfliktowanych transakcji. OptSVA+R wprowadza do SVA+R buforowanie operacji których efekty nie są widoczne na zewnątrz transakcji celem opóźnienia sprawdzania warunku dostępu do zmiennych. Dodatkowo OptSVA+R wprowadza nowatorski mechanizm przekazywania wykonania niektórych operacji do osobnych wątków, jeśli operacje te opóźniłyby wykonanie transakcji, a ich wyniki nie są niezbędne do kontynuowania obliczeń.

Zmienne Tylko-do-odczytu

Pierwszą optymalizacją możliwą dzięki rozróżnieniu operacji odczytu od zapisu jest wykorzystywane w większości istniejących pamięci transakcyjnych zrównoleglenie wykonań transakcji tylko-do-odczytu. OptSVA+R idzie o krok dalej, pozwalając także na częściowe zrównoleglenie wykonań operacji odczytu na zmiennych tylko-do-odczytu nawet w transakcjach, które wykonują zapisy na innych zmiennych.

W OptSVA+R suprema rozbite są na dwie wartości: maksymalna liczba odczytów zmiennej, które transakcja T_i wykona na zmiennej x, czyli $\mathtt{rub}_i(x)$, oraz analogiczna maksymalna liczba zapisów $\mathtt{wub}_i(x)$. Jeśli transakcja T_i zastanie sytuację, gdzie dla zmiennej x $\mathtt{rub}_i(x) > 0$ i $\mathtt{wub}_i(x) = 0$, to taka zmienna jest zmienną tylko-do-odczytu. W przypadku takich zmiennych transakcja OptSVA+R zapisze spójną wartość zmiennej x do bufora oznaczonego $\mathtt{buf}_i(x)$ i wykona wszystkie odczyty nie bezpośrednio ze zmiennej, a właśnie z bufora $\mathtt{buf}_i(x)$. Podczas gdy zapisanie zmiennej do bufora wymaga, żeby transakcja zsynchronizowała się z innymi transakcjami (gdyż inna transakcja może w tym samym czasie wykonywać modyfikacje na zmiennej x), transakcja T_i musi przed wykonaniem buforowania spełnić warunek dostępu. Natomiast, od razu po zbuforowaniu zmiennej transakcja może już ową zmienną zwolnić, a także, biorąc pod uwagę, że zmienna nie ulegnie modyfikacji, wykonać procedurę zatwierdzenia transakcji dla tej zmiennej.

W celu zwolnienia zmiennej jak najwcześniej buforowanie można wykonać nawet przed



Rys. 6: Opóźniona synchronizacja przy pierwszym zapisie w OptSVA+R.

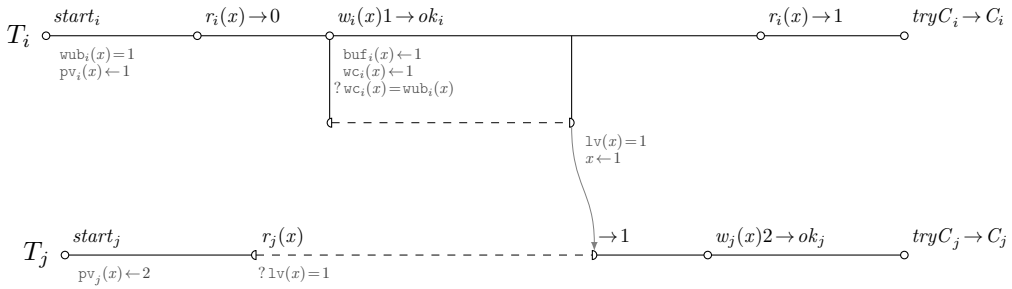
pierwszym odczytem zmiennej przez transakcje, więc w OptSVA+R buforowanie zmiennych tylko-do-odczytu uruchamiane jest już przy starcie transakcji. Natomiast ze względu na fakt, że buforowanie wymaga, żeby transakcja czekała na warunek dostępu, buforowanie oddelegowane jest do osobnego wątku. Pozwala to transakcji nie spowalniać wykonywania innych operacji ze względu na buforowanie zmiennych. Z drugiej strony, operacja odczytu na zmiennej tylko-do-odczytu może się odbyć tylko po tym jak zmienna zostanie zbuforowana, więc operacje na takich zmiennych czekają, aż wątek zakończy procedurę buforowania. Wykonanie buforowania w osobnym wątku pozwala transakcji na znalezienie najlepszego możliwego momentu w czasie kiedy ta procedura może być przeprowadzona, zwalniając zmienną najwcześniej jak to tylko możliwe, a jednocześnie blokując transakcje na warunku dostępu tylko, jeśli jest to absolutnie niezbędne.

Przykład optymalizacji zmiennej tylko-do-odczytu jest zilustrowany na Rys. 5. W momencie startu transakcja T_j uruchamia osobny wątek celem zbuforowania zmiennej x. Operacje wykonywane w osobnym wątku są zaprezentowane poniżej linii głównego wątku transakcji. Wątek wykonuje dwie kolejne operacje: buforowanie i zatwierdzanie pojedynczej zmiennej. Buforowanie zmiennej może odbyć się dopiero, gdy spełniony jest warunek dostępu do zmiennej x, więc wątek czeka do momentu, aż T_i zwolni zmienną x. Od razu po zbuforowaniu zmiennej, wątek zwalnia zmienną x i przechodzi do wykonania procedury zatwierdzenia. Pozwala to transakcji T_k na wykonanie operacji na x jak tylko T_j ją zbuforuje, lecz nie wymaga czekania aż T_j wykona na niej wszystkie swoje operacje. Dodatkowo, T_k może także zakończyć się wcześniej, ponieważ transakcja T_j wykona procedurę zatwierdzenia dla x jeszcze przed zakończeniem samej transakcji. Warto zauważyć, że w wypadku wycofania samej T_j wartość zmiennej x nie musi być zmieniona, więc nie ma potrzeby zmuszać T_k do wycofania się, natomiast w przypadku gdyby T_i została wycofana, zarówno T_i jak i T_j będą wycofane.

Synchronizacja przy Pierwszym Odczycie

Jeśli pierwszą (lub jedyną) operacją na jakiejś zmiennej jest zapis, to zapis ten można wykonać na buforze, bez odniesienia do aktualnego stanu zmiennej. Wykonanie operacji na buforze nie wymaga synchronizacji z innymi transakcjami, więc pierwsza operacja zapisu na zmiennej może się odbyć bez sprawdzania warunku dostępu. Dodatkowo, ponieważ wszystkie kolejne operacje zapisu lub odczytu na tej zmiennej mogą być wykonane na buforze, transakcja może odsunąć sprawdzanie warunku dostępu aż do ostatniego zapisu. Ostatecznie warunek dostępu musi być spełniony celem zapisania wartości zmiennej z bufora do zmiennej właściwej, co może zostać wykonane w dowolnym momencie między ostatnim zapisem a zatwierdzeniem transakcji. Ponieważ moment jest dowolny, to zadanie czekania na warunek dostępu i uaktualnienia zmiennej z bufora jest oddelegowane do osobnego wątku.

Przykład takiego przeplotu jest pokazany na Rys. 6, gdzie transakcja T_i ma dostęp



Rys. 7: Wczesne zwalnianie przy ostatnim zapisie w OptSVA+R.

do zmiennej x, ale T_j jednocześnie wykonuje operację zapisu na x, stosując bufor, a po wykonaniu tej operacji uruchamia wątek, który czeka na warunek dostępu (aż T_i zwolni x) i, gdy ten jest spełniony, zapisuje wartość z bufora do zmiennej.

Optymalizacja ta pozwala transakcji na wykonanie dodatkowych operacji na danym obiekcie (używając bufora), zanim spełniony zostanie warunek dostępu dla tego obiektu. W konsekwencji oznacza to, że konfliktujące ze sobą transakcje wykonują się bardziej równolegle niż było to możliwe w przypadku SVA+R.

Wczesne Zwalnianie przy Ostatnim Zapisie

Ponieważ OptSVA+R rozróżnia odczyty od zapisów, można zastosować kolejną optymalizację. Jeśli wykonywać wszystkie zapisy na lokalnym buforze i wprowadzać modyfikacje do zmiennych dopiero przy ostatnim zapisie, to odczyty, które następują po zapisach, mogą także korzystać z bufora. W konsekwencji, jeśli zmienna została zbuforowana i wszystkie zapisy zostały wykonane, zmienna może być zwolniona, a wszystkie kolejne odczyty mogą korzystać z wartości zmiennej zapisanej w buforze. Powoduje to, że wczesne zwalnianie zmiennych jest wykonywane wcześniej, niż w innych algorytmach wersjonowania.

Scenariusz taki jest zilustrowany na Rys. 7. T_i po wykonaniu wszystkich swoich zapisów na zmiennej x (tzn. jednego) zwalnia zmienną i wykonuje kolejne odczyty na buforze. Pozwala to transakcji T_j na wykonywanie operacji na zmiennej x mimo tego, że T_i wykonuje dalsze odczyty na tej zmiennej (korzystając z bufora). Powoduje to, że przeplot transakcji jest bardziej zrównoleglony, a więc wykonanie jest bardziej efektywne.

Zaprezentowana optymalizacja pozwala transakcjom na zwalnianie obiektów wcześniej, niż w przypadku SVA+R, ponieważ transakcje OptSVA+R nie muszą czekać na wykonanie wszystkich operacji odczytu na zmiennej, zanim zmienna zostanie zwolniona. Powoduje to zwiększenie stopnia współbieżności między transakcjami operującymi na tych samych zmiennych, co przekłada się na efektywność algorytmu. Co więcej, wykorzystanie wszystkich trzech optymalizacji powoduje, że transakcje wymagają wyłącznego dostępu do zmiennych w bardzo krótkich interwałach:

a) zmienne, które są tylko czytane, są trzymane na wyłączność tylko w momencie buforowania,

b) zmienne, do których się tylko zapisuje, są trzymane na wyłączność tylko w momencie przepisywania stanu z bufora do pamięci po ostatniej operacji zapisu,

c) zmienne, z których zarówno się czyta jak i do których się pisze, ale gdzie pierwszą operacją jest zapis, są również trzymane na wyłączność tylko w momencie przepisywana stanu z bufora do pamięci po ostatniej operacji zapisu,

d) pozostałe zmienne są trzymane na wyłączność tylko między pierwszym odczytem

a ostatnim zapisem.

Własności

W przypadku OptSVA+R formalnie porównano przeploty wygenerowane przez ten algorytm z przeplotami generowanymi przez SVA+R i pokazujemy, że przeploty wygenerowane przez OptSVA+R są zawsze nie dłuższe, a zazwyczaj krótsze, niż te wytworzone przez SVA+R. W konsekwencji OptSVA+R zapewnia większą wydajność niż jego poprzednik.

Jednocześnie OptSVA+R daje te same gwarancje bezpieczeństwa co SVA+R: spełnia nieprzezroczystość do ostatniego użycia. Natomiast ze względu na sposób, w jaki OptSVA+R oddziela wykonanie operacji od efektu, jaki dana operacja ma na zmienną, staje się trudnym udowodnienie tej własności wprost. W konsekwencji wprowadzono nową technikę dowodzenia nazwaną harmonią śladów (ang. *trace harmony*). Ślad definiujemy jako historię wykonania danego programu, w której ujęte są wykonania operacji transakcyjnych oraz wykonania instrukcji niskopoziomowych, takich jak dostępy do pamięci. Jeśli operacje i instrukcje w śladach spełniają zestaw cząstkowych wymagań określonych w Definicjach 29–52, to przeplot pokazany w danym śladzie jest nieprzezroczysty do ostatniego użycia, co pokazano w Twierdzeniu 9. Następnie w Lemacie 70 i Wniosku 22 pokazujemy, że każdy ślad wygenerowany przez OptSVA+R jest harmoniczny, więc OptSVA+R jest nieprzezroczysty do ostatniego użycia.

Dodatkowo OptSVA+R jest także wolny od zakleszczeń i silnie progresywny.

Warianty

Przez analogię do SVA+R i RSVA+R wprowadzono w pracy ROptSVA+R, wariant OptSVA+R, który pozwala na zdefiniowanie klasy transakcji niechętnych, które nigdy nie są zmuszane do wycofania kosztem wykonywania operacji na zmiennych zwolnionych wcześniej. Ponadto wprowadzono OptSVA, wariant OptSVA+R, który usuwa z algorytmu możliwość wykonania ręcznej operacji wycofania, co prowadzi do całkowitego wyeliminowania wycofań w algorytmie. Algorytm ten jest prostszy od OptSVA+R, ale nadaje się do użycia jedynie w modelu systemu z transakcjami dążącymi do zatwierdzenia.

5.6 OptSVA-CF+R

Ograniczenie OptSVA+R do modelu obiektów-zmiennych pozwala na wprowadzenie dużej liczby optymalizacji względem SVA+R. Natomiast przyjęcie tego modelu powoduje, że OptSVA+R nie jest stosowalny w wielu typach aplikacji. W szczególności, jeśli OptSVA+R miałby zostać wykorzystany do tworzenia rozproszonych systemów pamięci transakcyjnych działających w modelu przepływu sterowania, które z kolei byłyby aplikowane w złożonych systemach rozproszonych, założenia które OptSVA+R czyni względem stanu zmiennych i operacji które te zmienne wspierają są zbyt silne. W konsekwencji wprowadzono kolejny algorytm, OptSVA-CF+R, który ma zaaplikować optymalizacje wprowadzone w OptSVA+R do obiektów jednorodnych i niejednorodnych zarazem stanowiąc kompromis pomiędzy wydajnością a ogólnością stosowanego modelu.

Obiekty Niejednorodne

Jeśli przyjąć, że OptSVA-CF+R nie posiada żadnej wiedzy na temat obiektów na których operuje, żadna z wprowadzonych w OptSVA+R optymalizacji nie jest stosowalna. Nie można jednak oczekiwać od uniwersalnego rozwiązania, żeby znana i brana pod uwagę

była semantyka każdej operacji każdego obiektu w modelu niejednorodnym. W konsekwencji wprowadzono kompromis, który wymaga kategoryzacji każdej operacji każdego obiektu w jednej z trzech klas:

a) operacja *odczytu* to operacja, która wykonuje dowolny kod (także z efektami ubocznymi), który może zwracać dowolną wartość, ale który nie modyfikuje stanu obiektu,

b) operacja *zapisu* to operacja, która wykonuje dowolny kod, który może modyfikować stan obiektu, ale który nie czyta stanu obiektu ani nie zwraca wartości,

c) operacja *aktualizacji* to dowolna operacja która wykonuje dowolny kod i może zarówno odczytywać, jak i modyfikować stan obiektu, oraz zwracać wartość.

Klasyfikacja ta pozwala określić semantykę operacji w stopniu pozwalającym na zaaplikowanie optymalizacji. Rozróżnienie operacji „czystego" zapisu od operacji aktualizacji pozwala nam w szczególności na aplikowanie optymalizacji związanych z buforowaniem operacji zapisu. W wypadku gdy semantyka jakiejś operacji jest nieznana, może ona zawsze być sklasyfikowana jako operacja aktualizacji bez groźby niepoprawnego wykonania.

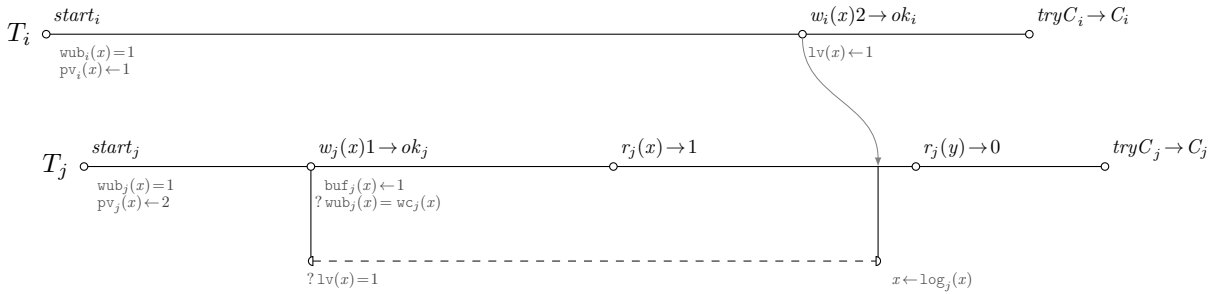
Buforowanie Obiektów

Ponieważ operacje zapisu na zmiennych mają prostą semantykę, która wiąże się z nadpisaniem całego stanu zmiennej, tj. jej wartości, możliwe jest wykonywanie operacji zapisu na „pustych" buforach. Nie jest to możliwe jednak w przypadku obiektów, których stan jest złożony: po wykonaniu operacji zapisu nie ma pewności, że następna operacja odczytu będzie korzystać z tego samego pola, które zostało zapisane przez operację zapisu.

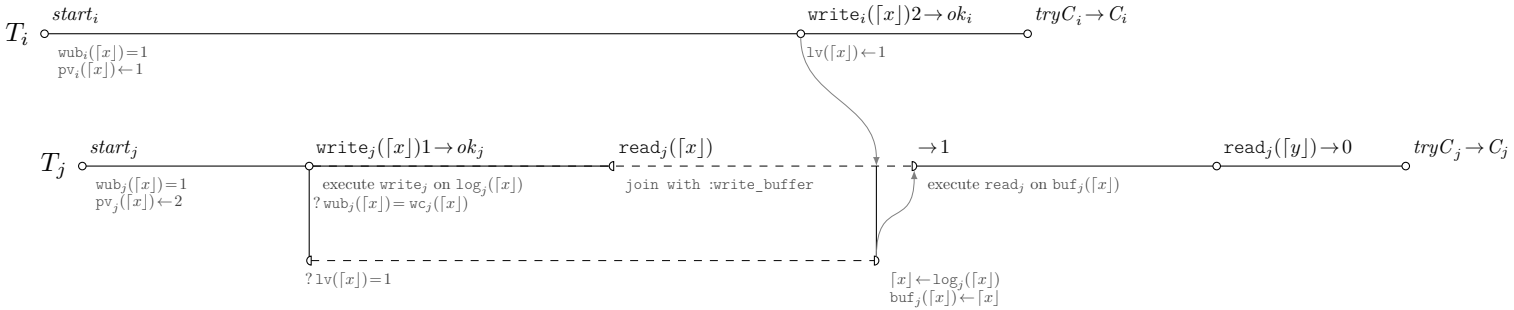
Celem rozwiązania tego problemu wprowadzono kolejny typ bufora: dziennik (ang. *log buffer*). Dziennik ma taki sam interfejs jak obiekt, z którym jest związany, ale operacja zlecona do wykonania nie zostaje wykonana *de facto*, a jedynie dodana do listy operacji do wykonania. Następnie taki dziennik może być zaaplikowany do obiektu, z którym jest związany, co spowoduje wykonanie wszystkich zleconych operacji. Dziennik może być użyty do odsuwania wykonań operacji zapisu i nie wymaga on uprzedniej inicjalizacji, tak jak jest to konieczne w przypadku standardowych buforów.

Asynchroniczne Buforowanie

OptSVA-CF+R obsługuje obiekty tylko-do-odczytu przez analogię do OptSVA+R. Podobnie jest w przypadku buforowania obiektów, na których wykonywany jest zapis, chociaż w tym wypadku procedura jest bardziej złożona i konserwatywna ze względu na użycie dwóch typów buforów. W wypadku, gdy pierwsza operacja na danym typie obiektu jest zapisem, zapis ten jest wykonywany bez synchronizacji na dzienniku dla tego obiektu. Jeśli po zapisie występują kolejne zapisy, to one także są kierowane do dziennika obiektu. Natomiast, jeśli po zapisach występuje aktualizacja lub odczyt, to należy dziennik zaaplikować do obiektu, żeby uzyskać zaktualizowany stan. Oznacza to, że OptSVA-CF+R w takim wypadku musi dokonać synchronizacji i czekać na warunku dostępu do danej zmiennej, co nie jest konieczne w przypadku OptSVA+R. Dodatkowo, w przeciwieństwie do OptSVA+R, gdzie zapisy zawsze wykonywane są przy użyciu bufora, transakcje OptSVA-CF+R, które wykonują zapisy, wykonują je bezpośrednio na obiekcie właściwym, jeśli warunek dostępu został uprzednio spełniony podczas odczytu lub aktualizacji. W końcu, OptSVA-CF+R, podobnie jak OptSVA+R, wykonuje kopię obiektu do bufora przy ostatnim zapisie lub aktualizacji (ostatniej potencjalnej modyfikacji dowolnego typu). W wypadku, gdy na obiekcie wykonywane były tylko zapisy, buforowanie wymaga także, aby dziennik był zaaplikowany do obiektu przed skopiowaniem go do bufora.



(a) OptSVA+R.



(b) OptSVA-CF+R.

Rys. 8: Obsługa buforowania w OptSVA+R vs OptSVA-CF+R.

Różnice między optymalizacjami zaaplikowanymi w OptSVA+R i OptSVA-CF+R są ilustruje Rys. 8. Pokazano tam wykonanie tego samego programu przy użyciu OptSVA+R (Rys. 8a) i OptSVA-CF+R (Fig. 8b). W obu przykładach $\lceil x \rfloor$ jest komórką z referencją; prostym obiektem który, zawiera pojedyncze pole stanowiące jego stan i interfejs analogiczny do zmiennej. W obu zaprezentowanych przeplotach transakcja T_i uruchamia się jako pierwsza, ale wykonuje operację zapisującą 2 do zmiennej x (obiektu $\lceil x \rfloor$) dopiero po długim opóźnieniu. W międzyczasie transakcja T_j wykonuje własny zapis do x ($\lceil x \rfloor$), zapisując 1. Ponieważ jest to początkowy zapis, to OptSVA+R wykonuje go na buforze $\texttt{buf}_j(x)$, a OptSVA-CF+R na dzienniku $\texttt{log}_j(\lceil x \rfloor)$, więc żaden z algorytmów nie powoduje, że T_j czeka na T_i. Następnie T_j wykonuje operację odczytu na x ($\lceil x \rfloor$). W OptSVA+R ta operacja jest wykonana na buforze $\texttt{buf}_j(x)$, co nie wymaga synchronizacji, więc operacja wykonuje się bez czekania. Natomiast w OptSVA-CF+R operacja odczytu nie może być wykonana na dzienniku, ponieważ dziennik nie zna stanu obiektu. Niezbędnym więc jest w OptSVA-CF+R, żeby T_j w tym momencie czekała, aż T_i nie zwolni obiektu $\lceil x \rfloor$. Dopiero wtedy T_j może zaaplikować $\texttt{log}_j(\lceil x \rfloor)$ do $\lceil x \rfloor$ i wykonać odczyt. W efekcie T_j wykonuje się dłużej w OptSVA-CF+R niż w OptSVA+R.

Przykład pokazuje więc, że generalizacja modelu niesie ze sobą potencjalny spadek efektywności wykonania. Jest to nieuniknione, biorąc pod uwagę złożoność obsługiwanych obiektów. Z drugiej strony, OptSVA-CF+R wciąż cechuje się bardzo wysokim stopniem zrównoleglenia transakcji skonfliktowanych i, jak pokazano poniżej, osiąga wysoką wydajność w praktyce.

Warianty

Tak samo, jak w przypadku OptSVA+R i SVA+R, OptSVA-CF+R ma wariant z transakcjami niechętnymi (ROptSVA-CF+R) oraz wariant działający w modelu z transakcjami dążącymi do zatwierdzenia (OptSVA-CF).

Własności

Pokazujemy, że OptSVA-CF+R jest nieprzezroczysty do ostatniego użycia przez analogię do OptSVA+R, oraz że jest on wolny od zakleszczeń i silnie progresywny.

5.7 Podsumowanie

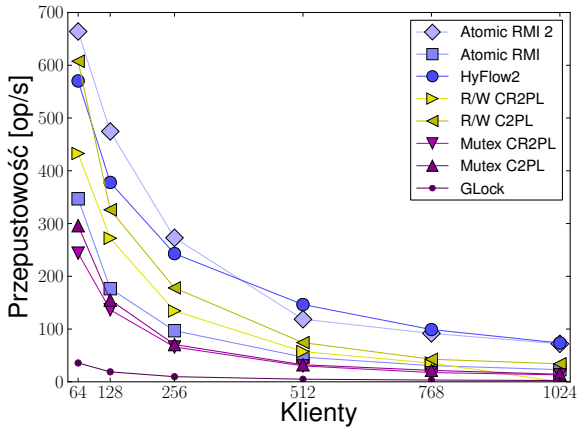
Wprowadzone algorytmy są podsumowane w Tablicy 4. Wszystkie algorytmy są pesymistyczne, blokujące i pozbawione zakleszczeń. Algorytmy z transakcjami niechętnymi zapewniają poprawne wykonanie transakcji niechętnych ze względu na operacje niewycofywalne, a algorytmy bez wycofań zapewniają poprawne wykonanie wszystkich transakcji ze względu na operacje niewycofywalne. Dodatkowo, algorytmy bez wycofania, poza spełnianiem nieprzezroczystości do ostatniego użycia, generują przeploty nierozróżnialne od przeplotów nieprzezroczystych pod względem efektów.

6 Implementacje

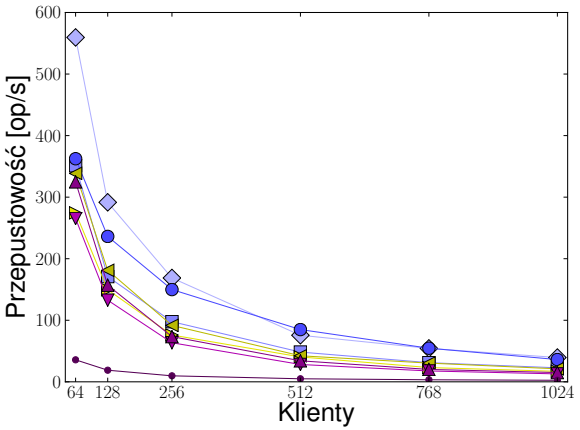
Zaimplementowano dwa z zaproponowanych algorytmów: SVA+R i OptSVA-CF+R (wraz z wariantami pozwalającymi na definicje transakcji niechętnych) jako systemy rozproszonej pamięci transakcyjnej oparte na Java RMI, nazwane odpowiednio Atomic RMI i Atomic RMI 2. Implementacje dostarczają mechanizmów niezbędnych do praktycznego

Algorytm	**Modyfikacje**	**Wycofania**	**A priori**	**Obiekty**	**Bezpieczeństwo**	**Zwalnianie zasobów**	**Operacje niewycofywalne**
SVA	p. wykonania	bez wycofania	*ASet*, `supr`	niejednorodne	opaque-equivalent	tak	$T_i \in \mathbb{T}$
SVA+R	p. wykonania	dowolne, przy kaskadzie	*ASet*, `supr`	niejednorodne	*last-use opaque*	tak	$\varnothing$
RSVA+R	p. wykonania	dowolne, przy kaskadzie	*ASet*, `supr`, $\mathbb{R}$	niejednorodne	*last-use opaque*	tak	$T_i \in \mathbb{R}$
OptSVA	p. zatwierdzeniu	bez wycofania	*ASet*, `wub`, `rub`,	zmienne	*last-use opaque**	tak	$T_i \in \mathbb{T}$
OptSVA+R	p. zatwierdzeniu	dowolne, przy kaskadzie	*ASet*, `wub`, `rub`,	zmienne	*last-use opaque*	tak	$\varnothing$
ROptSVA+R	p. zatwierdzeniu	dowolne, przy kaskadzie	*ASet*, `wub`, `rub`, $\mathbb{R}$	zmienne	*last-use opaque*	tak	$T_i \in \mathbb{R}$
OptSVA-CF	p. zatwierdzeniu	bez wycofania	*ASet*, `wub`, `rub`, klasy operacji	dowolne	*last-use opaque**	tak	$T_i \in \mathbb{T}$
OptSVA-CF+R	p. zatwierdzeniu	dowolne, przy kaskadzie	*ASet*, `wub`, `rub`, klasy operacji	dowolne	*last-use opaque*	tak	$\varnothing$
ROptSVA-CF+R	p. zatwierdzeniu	dowolne, przy kaskadzie	*ASet*, `wub`, `rub`, $\mathbb{R}$, klasy operacji	dowolne	*last-use opaque*	tak	$T_i \in \mathbb{R}$

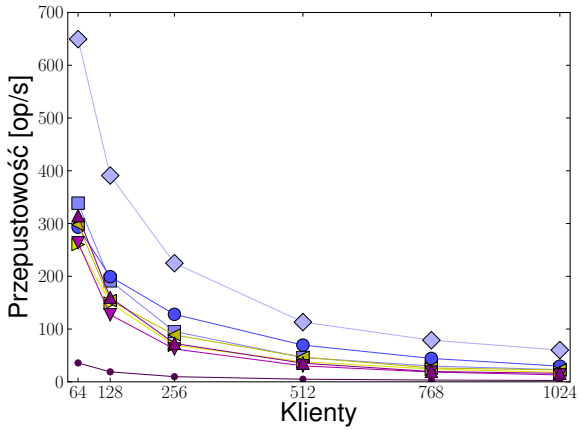
Tablica 4: Podsumowanie wprowadzonych algorytmów wersjonowania.



(a) 90% odczytów, 10% zapisów.

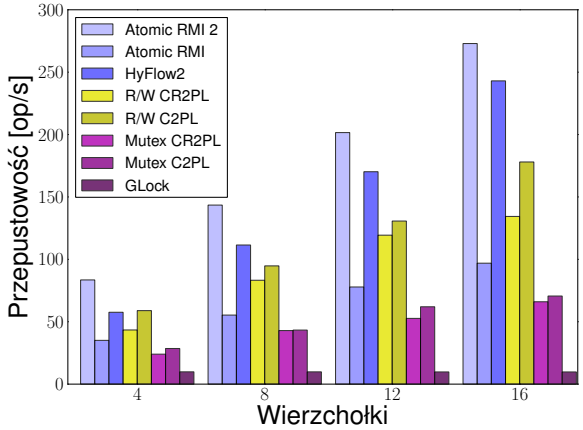


(b) 50% odczytów, 50% zapisów.

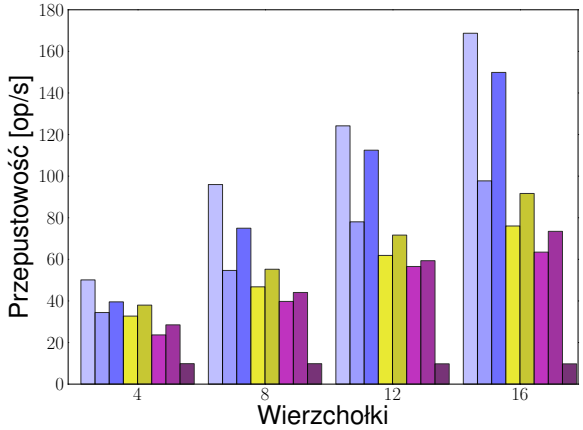


(a) 10% odczytów, 90% zapisów.

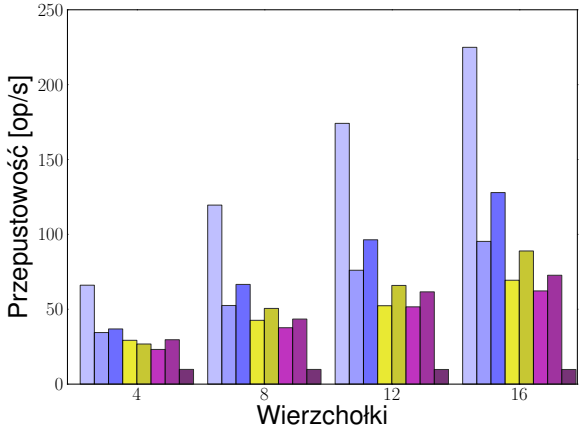
Rys. 9: Przepustowość (ang. *throughput*) vs liczba klientów.



(b) 90% odczytów, 10% zapisów, 10 tablic.



(c) 50% odczytów, 50% zapisów, 10 tablic.



(d) 10% odczytów, 90% zapisów, 10 tablic.

Rys. 10: Przepustowość (ang. *throughput*) vs liczba węzłów.

funkcjonowania systemu w środowisku rozproszonym, takich jak obsługa awarii częściowych, serializacja obiektów, implementacja buforów, oraz instrumentacja kodu w sposób ukrywający algorytmy sterowania współbieżnością przed programistą. Architektura zaimplementowanych systemów rozproszonej pamięci transakcyjnej rozszerza architekturę Java RMI o obiekty pełnomocników (ang. *proxy*), które implementują algorytmy sterowania współbieżnością przechwytując komunikację między klientami-transakcjami a obiektami współdzielonymi.

Implementacje zostały przetestowane pod względem wydajności przy użyciu programu wzorcowego (ang. *benchmarks*) EigenBench [23] przystosowanego do ewaluacji rozproszonej pamięci transakcyjnej. Ewaluacja porównuje Atomic RMI i Atomic RMI 2 z wysokiej klasy optymistycznym systemem rozproszonej pamięci transakcyjnej, HyFlow2 [43]. Dodatkowo porównano zaimplementowane w ramach pracy systemy z implementacjami algorytmów blokowania dwufazowego opartymi na zamkach z rozróżnieniem operacji odczytu i zapisu (R/W) lub traktującymi operacje jednakowo (Mutex), a także z implementacją zamka globalnego. Implementacje przetestowano na 16-węzłowym klastrze obliczeniowym połączonym siecią o prędkości 1Gb. Każdy węzeł posiada dwa czterordzeniowe procesory quad-core Intel Xeon L3260 taktowane 2.83 GHz z 4 GB pamięci RAM. Na każdym węźle działa system operacyjny OpenSUSE 13.1 (jądro 3.11.10, architektura x86_64). Użyto języka Groovy w wersji 2.3.8 oraz 64-bitowej Java HotSpot(TM) JVM w wersji 1.8 (build 1.8.0_25-b17).

Wyniki ewaluacji pokazane są na Rys. 9–10. Miarą wydajności jest przepustowość mierzona w liczbie operacji wykonanych na sekundę. Rys. 9 pokazuje, że wraz ze wzrostem liczby klientów (a więc i ze wzrostem współzawodnictwa o zasoby) spada przepustowość wszystkich systemów. Spadek wydajności jest szczególnie stromy do liczby 256 klientów i stabilizuje się dla 1024 klientów. Wszystkie algorytmy przewyższają wydajnością wykonanie sekwencyjne przy użyciu zamka globalnego. W scenariuszu, gdzie stosunek odczytów do zapisów wynosi 90%, HyFlow2 i Atomic RMI 2 wykonują się z wydajnością znacznie przewyższającą wydajność pozostałych systemów o 9–267%, lecz porównywalną względem siebie. W pozostałych dwóch scenariuszach wszystkie implementacje tracą na efektywności, z wyjątkiem Atomic RMI 2, który działa 9–359% lepiej od pozostałych implementacji (w tym HyFlow2). Różnica wydajności jest wynikiem optymalizacji operacji zapisu w Atomic RMI 2, która pozwala na skracanie przeplotów transakcji, gdy występują długie sekwencje operacji zapisu. Pozostałe implementacje nie optymalizują zapisów w takim stopniu. W szczególności HyFlow2 i 2PL opierają się głównie o zrównoleglenie odczytów. Degradacja wydajności Atomic RMI 2 jest wyjaśniona potrzebą zarządzania wątkami w celu obsługi asynchronii lokalnej. Powoduje to, że każdy węzeł musi obsłużyć więcej działających jednocześnie wątków niż w innych implementacjach. Spośród pozostałych implementacji, warianty C2PL działają zawsze lepiej, niż odpowiadające im warianty CS2PL, natomiast warianty R/W działają lepiej niż Mutex. Atomic RMI działa porównywalnie pod względem wydajności do C2PL i zdecydowanie gorzej niż Atomic RMI 2.

Rys. 10 pokazuje zmianę w przepustowości wraz z dodawaniem nowych węzłów do systemu (ale przy stałej przepustowości). Wraz z poszerzaniem systemu o dodatkową moc obliczeniową, przepustowość systemu wzrasta, dzięki powiększającej się liczbie potencjalnie wykonywanych równolegle operacji. Porównanie pokazuje, że Atomic RMI 2 zdecydowanie przewyższa wydajnością pozostałe implementacje, w tym także Atomic RMI i HyFlow2. Różnica ta uwydatnia się znów w wypadku gdy scenariusz jest zdominowany przez operacje zapisu, które są lepiej zoptymalizowane w Atomic RMI 2 niż w innych implementacjach. Ze względu na stosunkowo dużą liczbę obiektów współdzielonych w systemie (10 tablic na każdym z węzłów), sposób generowania transakcji powoduje, że Atomic RMI 2 jest także w stanie często zwalniać obiekty wcześnie, prowadząc do więk-

szego współczynnika zrównoleglenia transakcji. Spodziewać się można, że gdyby liczba obiektów była ograniczona, różnica między HyFlow2 i Atomic RMI 2 wyrównałaby się w przypadku scenariuszy analogicznych do Rys. 9b i 9c.

Istotnym jest także, że w ewaluacji liczba transakcji wycofanych przez Atomic RMI i Atomic RMI 2 wynosiła 0, podczas gdy HyFlow2 musiał wycofać i ponowić 60–89% transakcji (w zależności od scenariusza). Oznacza to, że w praktyce Atomic RMI 2 zachowuje się bezpiecznie względem operacji niewycofywalnych, podczas gdy w HyFlow2 mogą one być wykonane wielokrotnie.

7 Statyczna Analiza i Prekompilator

Dodatkowym elementem prezentowanym w pracy, poza algorytmami i ich własnościami, są dwa narzędzia mające na celu poprawienie praktyczności i wydajności zaprezentowanych systemów: prekompilator i moduł szeregowania transakcji.

Ze względu na fakt, że algorytmy wersjonowanie wymagają znajomości *a priori* obiektów używanych przez poszczególne transakcje, zadaniem prekompilatora jest analiza statyczna kodu każdej z transakcji i wydobycie tej informacji w sposób automatyczny. Dodatkowo, prekompilator bada kod, poszukując poszczególnych wywołań operacji na obiektach współdzielonych i oblicza przybliżone suprema dla każdego z obiektów wewnątrz transakcji. Prekompilator uwalnia programistę od potrzeby przygotowywania tej informacji ręcznie.

8 Podsumowanie

Aby udowodnić główną tezę niniejszej pracy zostały przeanalizowane istniejące własności bezpieczeństwa oraz ich przydatność w kontekście pamięci transakcyjnej z wczesnym zwalnianiem zmiennych (Rozdział 3). Następnie wprowadzone zostały własności bezpieczeństwa, które mają praktyczne zastosowanie dla tego typu pamięci transakcyjnych: nieprzezroczystość do ostatniego użycia i silna nieprzezroczystość do ostatniego użycia (Rozdział 5).

W dalszej kolejności opisano istniejące pesymistyczne algorytmy sterowania współbieżnością dla pamięci transakcyjnej, zarówno te rozproszone jak i nierozproszone, oraz optymistyczne algorytmy sterowania współbieżnością w rozproszonej pamięci transakcyjnej, a także algorytmy używające wczesnego zwalniania zasobów (Rozdział 4). Na podstawie tej analizy wybrano rodzinę algorytmów wersjonowania jako podstawę do dalszych badań. Następnie zmodyfikowano algorytm SVA, eliminując zależność od globalnego zamka, oraz pozwalając na swobodne wycofywanie transakcji (Rozdział 6, Sekcje 6.1 i 6.2).

Dalej zaprezentowano algorytmy OptSVA+R i OptSVA-CF+R oraz ich warianty. Są to nowatorskie algorytmy pesymistycznego sterowania współbieżnością rozszerzające istniejące algorytmy wersjonowania i stosujące optymalizacje, które pozwalają na wykonanie skonfliktowanych transakcji z większym stopniem zrównoleglenia niż było to możliwe do tej pory (Rozdział 6, Sekcje 6.3–6.4). Opracowane pesymistyczne algorytmy sterowania współbieżnością zostały użyte do implementacji dwóch systemów rozproszonej pamięci transakcyjnej. W pracy pokazano, że system oparty na OptSVA-CF+R i ROptSVA-CF+R jest w stanie uzyskać lepszą wydajność, niż wysokiej klasy implementacja opty-

mistycznej pamięci transakcyjnej, jednocześnie nie powodując konieczności wycofania transakcji (Rozdział 8).

Pokazano także, że choć zastosowane algorytmy reprezentują bardziej ogólne i wydajne podejście, zachowują one silne gwarancje bezpieczeństwa. Zostało to w pracy zademonstrowane przeprowadzając formalne dowody poprawności tych algorytmów, co wymagało wprowadzenia nowych technik dowodzenia poprawności algorytmów sterowania współbieżnością (Rozdział 7).

Ponadto, zaproponowano dodatkowy praktyczny moduł dla opracowanych systemów pamięci transakcyjnej, tj. prekompilator, który automatycznie generuje wiedzę *a priori* używaną do wczesnego zwalniania obiektów w algorytmach wersjonowania przez statyczną analizę kodu programu (Rozdział 9).

Dowód Tezy

Jako dowód postawionej w pracy tezy:

Przedstawiono Atomic RMI 2, system rozproszonej pamięci transakcyjnej w modelu przepływu sterowania, implementujący pesymistyczne algorytmy OptSVA-CF+R i ROptSVA-CF+R.

a) W Sekcji 8.2.2 pokazano, że Atomic RMI 2 przewyższa wydajnością wysokiej klasy optymistyczny system rozproszonej pamięci transakcyjnej, a więc Atomic RMI 2 jest systemem wydajnym.

b) Twierdzeniem 10 pokazano, że OptSVA-CF+R jest nieprzezroczysty do ostatniego użycia, Twierdzeniem 4 pokazano, że jest on silnie progresywny, a Twierdzeniem 3 pokazano, że jest on pozbawiony zakleszczeń. W konsekwencji OptSVA-CF+R spełnia silne gwarancje bezpieczeństwa, postępu i żywotności.

c) W Sekcji 8.2.2 pokazano, że OptSVA-CF+R w praktyce nie doprowadza do wycofań transakcji, a więc operacje niewycofywalne wykonywane są (w praktyce) poprawnie. Ponadto, ROptSVA-CF+R całkowicie wyklucza możliwość wycofywania transakcji niechętnych, a więc operacje niewycofywalne zawsze będą wykonywane poprawnie w przypadku ogólnym, zakładając, że będą wykonywane w ramach transakcji niechętnych.

d) OptSVA-CF+R wspiera dowolne wycofywanie transakcji, i operuje na niejednorodnym modelu obiektowym. Dodatkowo, nie ma pojedynczego punktu awarii. Ponadto, informacje niezbędne do działania tego algorytmu mogą być wygenerowane *a priori* przez prekompilator. W konsekwencji OptSVA-CF+R można uznać za algorytm mający praktyczne zastosowanie.

Reasumując, teza jest spełniona. □

Bibliografia

[1] Y. Afek, A. Matveev, and N. Shavit. Pessimistic software lock-elision. In *Proceedings of DISC'12: the 26th International Symposium on Distributed Computing*, Oct. 2012.

[2] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. A programming language perspective on transactional memory consistency. In *Proceedings of PODC'13: the 32nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 2013.

[3] H. Attiya, A. Gotsman, S. Hans, and N. Rinetzky. Safety of live transactions in transactional memory: TMS is necessary and sufficient. In *Proceedings of DISC'14: the 28th International Symposium on Distributed Computing*, 2014.

[4] H. Attiya and E. Hillel. Single-version STMs can be multi-version permissive. In *Proceedings of ICDCN'11: the 12th International Conference on Distributed Computing and Networking*, Jan. 2011.

[5] H. Avni, S. Dolev, P. Fatourou, and E. Kosmas. Abort free semantic TM by depedency aware scheduling of transactional instructions. In *Proceedings of NETYS'14: the International Conference on Networked Systems*, May 2014.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems.* Addison-Wesley, 1987.

[7] A. Bieniusa, A. Middelkoop, and P. Thiemann. Brief announcement: Actions in the twilight—concurrent irrevocable transactions and inconsistency repair. In *Proceedings of PODC'10: the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2010.

[8] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of PPoPP'08: the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[9] W. Cellary, E. Gelenbe, and T. Morzy. *Concurrency control in distributed database systems.* North-Holland, 1988.

[10] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *Proceedings of PRDC'13: the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, Nov. 2009.

[11] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Proceedings of DISC'06: the 20th International Symposium on Distributed Computing*, Sept. 2006.

[12] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25(5), Sept. 2013.

[13] S. Dolev, D. Hendler, and A. Suissa. CAR-STM: Scheduling-based collision avoidance and resolution for software transactional memory. In *Proceedings of PODC'08: the 28th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Aug. 2008.

[14] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh. Preventing vs curing: Avoiding conflicts in transactional memories. In *Proceedings of PODC'09: the 28th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Aug. 2009.

[15] D. Dziuma, P. Fatourou, and E. Kanellou. Consistency for transactional memory computing. *Bulletin of the EATCS*, 113, 2014.

[16] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *Proceedings of DISC'09: the 23rd International Symposium on Distributed Computing*, Sept. 2009.

[17] R. Guerraoui and M. Kapałka. *Principles of Transactional Memory.* Morgan & Claypool, 2010.

[18] T. Harris and K. Fraser. Language Support for Lightweight Transactions. In *Proceedings of OOPSLA'03: the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2003.

[19] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2nd edition, 2010.

[20] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of PPoPP'05: the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 2005.

[21] M. Herlihy, V. Luchangco, M. Moir, and I. W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *Proceedings of PODC'03: the 22nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2003.

[22] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of ISCA'93: the 20th International Symposium on Computer Architecture*, pages 289–300, May 1993.

[23] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *Proceedings of IISWC'10: the IEEE International Symposium on Workload Characterization*, 2010.

[24] D. Imbs, J. R. de Mendivil, and M. Raynal. On the consistency conditions or transactional memories. Technical Report 1917, IRISA, Dec. 2008.

[25] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *Proceedings of ICPP'08: the 37th IEEE International Conference on Parallel Processing*, Sept. 2008.

[26] A. Matveev and N. Shavit. Towards a fully pessimistic STM model. In *Proceedings of TRANSACT'12: the 7th ACM SIGPLAN Workshop on Transactional Computing*, Aug. 2012.

[27] C. H. Papadimitrou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4), 1979.

[28] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *Proceedings of PODC'10: the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2010.

[29] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. In *Proceedings of PPoPP'09: the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Feb. 2009.

[30] M. M. Saad and B. Ravindran. HyFlow: A high performance distributed transactional memory framework. In *Proceedings of HPDC '11: the 20th International Symposium on High Performance Distributed Computing*, June 2011.

[31] I. W. N. Scherer and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of PODC'05: the 24th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2005.

[32] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of PODC'95: the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Aug. 1995.

[33] K. Siek and P. T. Wojciechowski. Brief announcement: Statically computing upper bounds on object calls for pessimistic concurrency control. In *Proceedings of EC^2'10: the Workshop on Exploiting Concurrency Efficiently and Correctly*, July 2010.

[34] K. Siek and P. T. Wojciechowski. A Formal Design of a Tool for Static Analysis of Upper Bounds on Object Calls in Java. In *Proceedings of FMICS'12: the 17th International Workshop on Formal Methods for Industrial Critical Systems*, number 7437 in Lecture Notes in Computer Science, Aug. 2012.

[35] K. Siek and P. T. Wojciechowski. Brief announcement: Towards a fully-articulated pessimistic distributed transactional memory. In *Proceedings of SPAA'13: the 25th ACM Symposium on Parallelism in Algorithms and Architectures*, July 2013.

[36] K. Siek and P. T. Wojciechowski. Atomic RMI: a distributed transactional memory framework. In *Proceedings of HLPP'14: the 7th International Symposium on High-level Parallel Programming and Applications*, July 2014.

[37] K. Siek and P. T. Wojciechowski. Brief announcement: Relaxing opacity in pessimistic transactional memory. In *Proceedings of DISC'14: the 28th International Symposium on Distributed Computing*, Oct. 2014.

[38] K. Siek and P. T. Wojciechowski. Zen and the art of concurrency control: An exploration of tm safety property space with early release in mind. In *Proceedings of WTTM'14: the 6th Workshop on the Theory of Transactional Memory*, July 2014.

[39] K. Siek and P. T. Wojciechowski. Atomic RMI: A distributed transactional memory framework. *International Journal of Parallel Programming*, 44(3), June 2015.

[40] K. Siek and P. T. Wojciechowski. Last-use opacity: A strong safety property for transactional memory with early release support. June 2015. `arXiv:1506.06275 [cs.DC]` (w przygotowaniu).

[41] K. Siek and P. T. Wojciechowski. Proving opacity of transactional memory with early release. foundations of computing and decision sciences. *Foundations of Computing and Decision Sciences*, 40(4), Dec. 2015.

[42] K. Siek and P. T. Wojciechowski. Atomic RMI 2: Highly parallel pessimistic distributed transactional memory. Apr. 2016. `arXiv:1606.03928 [cs.DC]` (w przygotowaniu).

[43] A. Turcu, B. Ravindran, and R. Palmieri. HyFlow2: A high performance distributed transactional memory framework in scala. In *Proceedings of PPPJ'13: the 10th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, Sept. 2013.

[44] G. Weikum and G. Vossen. *Transactional information systems: Theory, algorithms, and the practice of concurrency control and recovery.* Morgan Kaufmann Publishers, 2002.

[45] A. Welc, B. Saha, and A.-R. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of SPAA'08: the 20th ACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.

[46] P. T. Wojciechowski. Isolation-only transactions by typing and versioning. In *Proceedings of PPDP'05: the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, July 2005.

[47] P. T. Wojciechowski. *Language design for atomicity, declarative synchronization, and dynamic update in communicating systems.* Publishing House of Poznań University of Technology, 2007.

[48] P. T. Wojciechowski and K. Siek. The optimal pessimistic transactional memory algorithm, May 2016. `arXiv:1605.010361 [cs.DC]` (w przygotowaniu).

[49] R. M. Yoo and H.-H. S. Lee. Adaptive transaction scheduling for transactional memory systems. In *Proceedings of SPACC'08: the 20thACM Symposium on Parallelism in Algorithms and Architectures*, June 2008.