Instytut Informatyki Politechniki Poznańskiej

Analiza Programów Wzorcowych dla Rozproszonej Pamięci Transakcyjnej

Jan Baranowski, Konrad Siek, Paweł T. Wojciechowski

Raport RB-3/15

Poznań, 2015

Streszczenie

Kluczowym elementem tworzenia i usprawniania implementacji pamięci transakcyjnej jest ewaluacja eksperymentalna w oparciu o zestaw standardowych programów wzorcowych (ang. benchmarks). Niestety, kompleksowe zestawy programów wzorcowych przeznaczone są tylko dla wieloprocesorowych systemów pamięci transakcyjnej. W przypadku systemów rozproszonej pamięci transakcyjnej zbiór istniejących programów wzorcowych jest bardzo wąski. W rezultacie, badacze zmuszeni są przygotowywać programy wzorcowe *ad hoc*, np. poprzez implementację wybranych problemów rozproszonych (np. rozproszona tablica mieszająca). Takie implementacje badają cechy systemów rozproszonej niedostatecznie dokładnie i często nie reprezentują problemów w których rozproszona pamięć transakcyjna ma mieć zastosowanie. Dodatkowo, implementacja własnych programów wzorcowych przez różne zespoły badawcze powoduje że porównanie pracy między zespołami jest trudne lub niemożliwe. Praca ta ma na celu analizę istniejących programów wzorcowych do ewaluacji pamięci transakcyjnej oraz opracowanie założeń i projektów pod implementację programów wzorcowych dla systemów rozproszonej pamięci transakcyjnej. W efekcie możliwe stanie się przygotowanie spójnych implementacji rozproszonej pamięci transakcyjnej.

Rozdział 1

Wstęp

W wielu publikacjach dotyczących przetwarzania równoległego, mechanizmów takie przetwarzanie wspomagających, a zwłaszcza pamięci transakcyjnych można znaleźć pewną charakterystyczną cechę. Będzie to stwierdzenie, zawarte zazwyczaj we wstępie, głoszące iż „programowanie współbieżne (ang. *concurrent programming*), zwłaszcza dostatecznie wydajne, jest wyjątkowo trudne" [21, 30, 31, 53, 62]. Bez wątpienia tworzenie programów składających się ze zbioru jednostek, które docelowo będą współbieżnie operowały na tych samych zasobach jest dla programisty wyzwaniem i naraża go na ryzyko popełnienia szeregu błędów. Nie może zatem dziwić istnienie ruchu w obszarze badań naukowych, którego uczestnicy stawiają sobie za cel ułatwienie tego zadania w wielu aspektach (wzmocnienie gwarancji poprawności przetwarzania, nawet biorąc pod uwagę niskie kompetencje użytkowników nowo proponowanych mechanizmów – programistów, ang. *mainstream programmers*; poprawa wydajności przetwarzania; uproszczenie interfejsów programistycznych, ang. *application programming interface*, API). Mimo pewnego stopnia oczywistości, stwierdzenie to jest godne uwagi ponieważ dobitnie pokazuje charakterystykę znaczącej gałęzi współczesnej informatyki.

Programowanie równoległe jest trudne. Powszechne przyjęcie modelu przetwarzania równoległego, na który składają się wątki[1] działające równolegle na kilku procesorach jednego systemu komputerowego lub na przemian wywłaszczane z jednego procesora w sposób niedeterministyczny[2] oraz zestaw współdzielonych zasobów (pamięć operacyjna, w mniejszym stopniu urządzenia wejścia-wyjścia) doprowadziło w praktyce do stworzenia środowiska podatnego na występowanie wielu błędów (ang. *fault* [3], na etapie projektowania i implemen-

[1] różnica pomiędzy wątkami a procesami wymaga podkreślenia. Wątki (*procesy lekkie*) jako jednostki przetwarzania wyodrębniane w ramach procesu, dziedziczące posiadane przezeń zasoby, z definicji będą posiadały współdzielony fragment pamięci operacyjnej przypisanej procesowi-rodzicowi. W pojęciu procesu współdzielenie zasobów nie jest wprost zawarte. Jednak z racji tego, że niniejszy raport dotyczy przede wszystkim synchronizacji jednostek przetwarzania przy dostępie do współdzielonych zasobów, a te mogą być współdzielone zarówno przez kilka procesów, jak i przez kilka wątków jednego procesu, terminy *proces* i *wątek* będą oznaczać to samo – wspomnianą właśnie jednostkę przetwarzania – do momentu, w którym ich rozróżnienie stanie się istotne. Ten moment to przeniesienie pojęć na grunt systemów rozproszonych, w przypadku których trudno jest mówić o centralnej puli zasobów, a więc i o wątkach.

[2] a więc taki, który nie pozwala programiście przewidzieć czasów potrzebnych na wykonanie kolejnych kroków przetwarzania (lub ewentualnie na nie wpływać). Terminem, którym opisywana jest ta właściwość będzie *asynchronizm procesów* (por. [26, str. 18]).

tacji programu) i anomalii (rozumianych dwojako: jako zachowanie systemu, którego programista nie przewidział, a więc szczególny przypadek *fault* oraz jako skutek w postaci błędnego działania programu, ang. *failure* [3]), z których najbardziej znane to zakleszczenia i inwersja priorytetów (przy założeniu, że metodą synchronizacji procesów przy dostępie do współdzielonych zasobów jest niewywłaszczalne zajmowanie tych zasobów na wyłączność), anomalie niesynchronizowanego współbieżnego dostępu (powstanie niespójności w obszarze współdzielonej pamięci operacyjnej), czy też brak zachowania warunku postępu przetwarzania z różnych przyczyn (poza typowymi zakleszczeniami mogą to być m.in. *livelocki* lub zagłodzenia procesów powodowane przez stosowanie błędnych, tj. niesprawiedliwych algorytmów wzajemnego wykluczania). Działanie w takim środowisku wymaga od programisty szczególnej uwagi i niemałych umiejętności.

Programowanie równoległe jest warte zachodu. Wiele zjawisk współczesnej informatyki, m.in. napotkanie granicy wydajnościowej pojedynczych rdzeni procesorów, dyktowanej przez własności fizyczne materiałów oraz próba obejścia tej granicy przy użyciu procesorów wielordzeniowych [31, punkt 1.1] (można zapewne założyć że przeskoczenie granicy pomiędzy procesorem jednordzeniowym a dwurdzeniowym otwiera drogę do dalszego, stosunkowo łatwego rozwoju typu *scale-out* systemów przetwarzania współbieżnego) wskazuje, że obserwowany dziś wzrost znaczenia technik i narzędzi programowania współbieżnego będzie postępował, przede wszystkim ze względu na brak analogicznej granicy dla zapotrzebowania na moc obliczeniową. Można też wskazać pewne trendy w projektowaniu systemów komputerowych, zwłaszcza mobilnych, gdzie przetwarzanie współbieżne już odgrywa i odgrywać będzie coraz bardziej istotną rolę. Dla urządzeń mobilnych, w których kluczowym zasobem jest energia, dąży się do zmniejszenia jej zużycia kosztem zmniejszenia szybkości działania procesorów, a tym samym – mocy obliczeniowej. Straty te próbuje się rekompensować wielordzeniowością, tworząc potencjał za którego zagospodarowanie miałoby odpowiadać właśnie programowanie współbieżne.

Wobec tak oczywistego wzrostu znaczenia systemów wieloprocesorowych, a razem z nimi zjawisk współbieżnego dostępu do współdzielonych zasobów i problemów występujących podczas prób synchronizacji dostępu do tych zasobów standardowymi metodami, nie dziwi wzmożony wysiłek badaczy, starających się dostarczyć narzędzia, które z jednej strony upraszczają zadanie programistów aplikacji wielowątkowych (tym samym minimalizując ryzyko popełnienia przez nich błędów), z drugiej zaś oferują systemowe podejście do zapewnienia odpowiedniego poziomu wydajności przetwarzania.

Jednym z proponowanych rozwiązań obu tych problemów jest *pamięć transakcyjna*. Ten mechanizm synchronizacji przeznaczony dla systemów przetwarzania wielowątkowego w założeniu pozwala na objęcie szeregu dostępów wykonywanych przez jeden wątek do współdzielonej pamięci operacyjnej transakcją, podobną do tych znanych z systemów baz danych. Prosty model transakcji zakłada, że wątek po zrealizowaniu serii odczytów i zapisów wybiera, czy należy je zatwierdzić (ang. *commit*), tak by stały się widoczne dla innych wątków współdzielących pamięć, czy też wycofać (ang. *abort*), nie pozostawiając pozostałym wątkom żadnej możliwości zarejestrowania choćby części wprowadzonych zmian. Spośród własności transakcji bazodanowych, dla przetwarzania wielowątkowego najbardziej istotną będzie izolacja (włączana później w skład zbioru własności nazywanych *atomowym wykonaniem* sekcji kodu,

ang. *atomic execution* [31, punkt 1.2.1]). Efekty transakcji zatwierdzonej powinny zostać naniesione na pamięć operacyjną tak, by żaden wątek nie kontynuował działania w oparciu o odczytany stan przejściowy, potencjalnie niespójny. Podobnie w przypadku transakcji anulowanych – stany przejściowe jako wynik wycofanych operacji nie powinny dawać się odczytać innym wątkom w taki sposób by mogły one kontynuować działanie w oparciu o wynik odczytu. To, być może nieco zawiłe, podejście do izolacji wynika z różnic w charakterze systemów transakcyjnych (imperatywnego dla pamięci transakcyjnych i deklaratywnego dla baz danych). Można bowiem wyobrazić sobie sytuację, w której wątek generalnie może postrzegać niespójne stany pamięci współdzielonej, jednak gdy ten fakt zostanie wykryty, potencjalnie przy zatwierdzaniu zmian, odczyty takie spowodują unieważnienie i anulowanie transakcji, której są elementem. Ponownie ze względu na imperatywny charakter środowisk wykorzystujących pamięci transakcyjne, takie podejście może okazać się niedopuszczalne.

1.1 Rys historyczny

Historia rozwoju różnych wariantów pamięci transakcyjnych nie jest jednoetapowa. Analizując literaturę przedmiotu można pokusić się o wyróżnienie trzech, a nawet czterech okresów cechujących się różnym podejściem do prac badawczych w tej dziedzinie. W pierwszym okresie, którego znaczenie dla niniejszego raportu jest nikłe, bezpośrednio po zaproponowaniu pierwszych, przede wszystkim sprzętowych, rozwiązań (np. w [33], publikacji wprowadzającej sprzętowy mechanizm dziś uważany za pierwowzór pamięci transakcyjnej [31, punkt 1.3]) prace mają charakter przede wszystkim innowacyjny (propozycje zupełnie nowych rozwiązań w miejsce ulepszania już istniejących). Punktem zwrotnym staje się artykuł [30], przedstawiający przykładową implementację programowej pamięci transakcyjnej i wyjątkowo obiecujące wyniki jej zastosowania w przypadku dostępów do specyficznych typów współdzielonych struktur danych. Artykuł ten rozpoczyna drugi okres – czas intensywnych prac nad programowymi pamięciami transakcyjnymi dla systemów wieloprocesorowych[3]. W kolejnym, trzecim okresie implementacje pamięci transakcyjnej poddawane są silnej krytyce zarówno w odniesieniu do ich wydajności (np. [12]) jak i poprawności przetwarzania transakcyjnego (np. [65]). Etap czwarty, jak dotąd ostatni, to odpowiedzi na zarzuty sformułowane w poprzednim. Jego charakterystycznymi elementami będą prace teoretyczne mające na celu m.in. sformalizowanie kryteriów poprawności implementacji pamięci transakcyjnej ([25, 26]) i wprowadzenie oraz ocena wydajnościowa systemów określanych jako *state-of-the-art*, a więc takich, które przy użyciu najnowszych rozwiązań i podejść odpowiadałyby na zarzuty stawiane pamięciom transakcyjnym w kwestii ich poprawności i wydajności (np. *SwissTM* [22]).

Sama możliwość wyróżnienia czwartego etapu, a więc spodziewana opłacalność wysiłków mających na celu poprawę istniejących rozwiązań pokazuje znaczenie przypisywane systemom pamięci transakcyjnej. Miały one z jednej strony uprościć programowanie wielowątkowe, dostarczając programistom znane im skądinąd mechanizmy i abstrakcje, z drugiej zaś,

[3] tj. systemów dopuszczających przetwarzanie współbieżne, z tym zastrzeżeniem, że uruchamianie transakcji w systemach jednoprocesorowych, choć być może ułatwiłoby tworzenie programów, wydajnościowo byłoby pozbawione sensu, ponieważ systemy takie nie pozwalają na uzyskanie faktycznej równoległości, a transakcje wykonywane sekwencyjnie, jedna po drugiej lub przeplatane razem z wątkami zapewne działają wolniej niż odpowiadające im rozwiązania bazujące na globalnym zamku współdzielonym.

poprzez współbieżne wykonywanie transakcji i nietrywialne podejście do rozwiązywania konfliktów przy dostępach miały po pierwsze przyspieszyć działanie aplikacji wielowątkowych, względem tych, które wykorzystywały zawłaszczanie przez wątek całej puli zasobów współdzielonych, po drugie uprościć implementację względem tych, które blokowały jedynie wybrane, małe porcje danych (choćby poprzez wyeliminowanie ryzyka wystąpienia zakleszczeń), a wreszcie dostarczyć akceptowalny (tj. nie gorszy niż inne, powszechnie stosowane rozwiązania w ogólnym przypadku, a znacząco lepszy dla pewnych wybranych scenariuszy) poziom wydajności. Wszystko to w systemach wieloprocesorowych.

Wysiłek włożony w badania na czwartym etapie rozwoju, a także powszechna dostępność dojrzałych systemów pamięci transakcyjnych w zestawach narzędzi dla programistów aplikacji wielowątkowych (np. [32]) może wskazywać, że choć pewnie nie wszystkie i nie w pełni, wymagania stawiane systemom pamięci transakcyjnych są generalnie w najnowszych implementacjach spełnione.

1.2 Kierunki rozwoju pamięci transakcyjnych

Kolejnym krokiem rozwoju pamięci transakcyjnych wydaje się być przeniesienie rozwiązań wypracowanych w systemach wieloprocesorowych na grunt systemów rozproszonych. Jest to krok, który mógł być oczekiwany z kilku względów. Po pierwsze, przy dużym uproszczeniu, modele obu klas systemów są podobne – w obu przypadkach mamy do czynienia z pulą współdzielonych zasobów, wykorzystywanych przez działające współbieżnie jednostki przetwarzania. To dawałoby pole do wykorzystania istniejących już mechanizmów, interesujących ze względu na wydajność. Po drugie, przy bliższym spojrzeniu, aspekty takie jak zapewnienie różnych wariantów przezroczystości, w tym uodpornienie systemu na awarie węzłów, utrudniają programowanie w środowisku rozproszonym. Każda pomoc, w tym uproszczenia oferowane przez pamięć transakcyjną, jest tu mile widziana. Dodatkowo należy zaznaczyć, że pojęcie transakcji jest już obecne w kontekście systemów rozproszonych, jednak w nieco innym wariancie. Sekcja 3.1 wpisuje pojęcie transakcji w kontekst systemów zarządzania bazami danych, także rozproszonych. Pewną formą transakcji jest też zapewne mechanizm *atomic actions* i wykorzystywany przezeń protokół trójfazowego zatwierdzania (ang. *three phase commit*, 3PC) [63], pozwalający osiągnąć niepodzielność wykonywania grupy operacji w środowisku podatnym na permanentne awarie węzłów (*fail-stop*, por. tabela 1.1).

Niniejszy raport wpisuje się w nurt przeniesienia koncepcji znanych z implementacji pamięci transakcyjnych w systemach wieloprocesorowych na grunt systemów rozproszonych.

1.3 Zarys problemu

O ile o problemach poprawnościowych pamięci transakcyjnych można powiedzieć, że zostały one rozwiązane, a nawet jeżeli nie, to – że przyszłe rozwiązania wypracowane dla systemów równoległych będą mogły być stosowane do systemów rozproszonych, o tyle nie sposób twierdzić czegoś podobnego o problemach wydajnościowych. Wydajność aplikacji rozproszonej uzależniona jest od znacznie większej liczby czynników niż szybkość działania aplikacji

Tablica 1.1: Porównanie wybranych cech systemów równoległych i rozproszonych.

	Systemy równoległe	**Systemy rozproszone**
Szybkość jednostek przetwarzających	Porównywalna, ewentualnie moderowana priorytetami procesów (aczkolwiek przyjmujemy, że procesy są asynchroniczne).	Bez ograniczeń. Dopuszczalna jest sytuacja, w której węzły działają ze skrajnie różnymi prędkościami.
Szybkość kanałów komunikacyjnych	Aspekt pomijalny. W praktyce wpływ na szybkość komunikacji między procesorami z użyciem pamięci współdzielonej może mieć architektura pamięci podręcznej.	Zależna od implementacji i otoczenia (prawdopodobieństwa uszkodzenia komunikatu w trakcie transmisji i zdolność tolerowania takiej awarii).
Zawodność kanałów komunikacyjnych	Aspekt pomijalny.	Cecha typowa dla systemów rozproszonych. Przy założeniu awarii przejściowych (ang. *transient failures*) [43, str. 2] może skutkować opóźnieniami w komunikacji i tym samym w postępie przetwarzania.
Model awarii	Zazwyczaj *fail-stop* [43, str. 9] dla całego systemu[4].	Model awarii całego systemu przewiduje zdolność tolerowania awarii pojedynczych węzłów i kanałów komunikacyjnych. Model awarii węzłów zakładający np. konieczność odtwarzania stanu może mieć wpływ na opóźnienia w przetwarzaniu.

wielowątkowej. Tabela 1.1 przedstawia wybrane różnice w charakterystykach obu klas systemów docelowych, które mogą mieć na nią wpływ.

Wymienione tam różnice sprawiają, że przeniesienie wyników badań wydajnościowych standardowych systemów pamięci transakcyjnej na grunt systemów rozproszonych i konstruowanie w ten sposób nowych rozwiązań to droga, która nie musi prowadzić do spodziewanego polepszenia. Analiza wydajności rozproszonych pamięci transakcyjnych urasta do rangi osobnej gałęzi badań.

[4] w [43] znaleźć można określenie typu awarii *stopping failure* jako awarii procesora/węzła, w której jednostka taka całkowicie przestaje reagować. Problem odporności na awarie nie jest elementem specyfiki programowania równoległego w takim stopniu, w jakim jest on istotny dla systemów rozproszonych. Jeżeli jeden wątek ulega awarii, to z faktu, że jest on częścią programu działającego równolegle nie wynika wprost, że przetwarzanie powinno móc być kontynuowane. Jeżeli przestaje działać równoległy odpowiednik kanału komunikacyjnego, to oznacza to, że awarii mógł ulec system pamięci podręcznej, a więc najprawdopodobniej cały procesor. Stąd zdecydowano się stwierdzić, że programy dla środowiska równoległego zazwyczaj ulegają awariom typu *fail-stop*, nawet pomimo prób sformalizowania problemu odporności na awarie pojedynczych wątków, w postaci choćby własności *obstruction-freedom* algorytmów równoległych, przytaczanej przez [34, str. 60].

Dodatkowym czynnikiem sugerującym skupienie uwagi na badaniach wydajnościowych pamięci transakcyjnych jest pojawienie się nowej klasy implementacji, której cechy charakterystyczne dobrze rokują dla systemów rozproszonych. Kluczowym elementem niniejszy raport jest przeciwstawienie sobie aspektów wydajnościowych rozproszonych pamięci transakcyjnych z optymistycznym (zakładającym, że konfliktowe dostępy będą występować rzadko i preferującymi rozwiązywanie konfliktów w miejsce ich unikania) i pesymistycznym (koncentrującym się w pierwszej kolejności na unikaniu – w założeniu częstych – konfliktów) podejściem do zarządzania wersjami przy dostępach do współdzielonych zasobów.

Należy zadać sobie pytanie, w jaki sposób testować wydajność proponowanych rozwiązań. Podejścia analityczne (szacowanie złożoności czasowej i komunikacyjnej najgorszego przypadku wykonania pojedynczego dostępu lub całej transakcji) czy probabilistyczne (wyznaczanie prawdopodobieństwa zajścia poszczególnych zdarzeń: wykonań transakcji, dostępów, itd. pod pewnymi warunkami) wydają się mieć ograniczone znaczenie praktyczne już dla systemów równoległych. Przykładowo próba oszacowania średniego czasu wykonania danego typu transakcji w oparciu o statyczną analizę programu natrafia na problem już przy próbie oceny czy kod objęty transakcją w ogóle skończy się wykonywać. Nawet gdyby pominąć ten aspekt, różnorodność algorytmów planowania przydziału procesora, czy niedeterminizm operacji wejścia-wyjścia całkowicie skreślają tę metodę.

Porównywanie systemów pamięci transakcyjnych w oparciu o optymalizacje przez nie wprowadzane, na przykładzie pojedynczych instancji transakcji wydaje się niepraktyczne. Nie wiadomo jak często, ani nawet czy w ogóle założone warunki początkowe wystąpią w trakcie rzeczywistego przetwarzania.

Jeżeli wyżej wspomniane metody nie są odpowiednie w środowiskach równoległych, to tym bardziej nie nadają się one do stosowania w przetwarzaniu rozproszonym. O ile o przetwarzaniu równoległym można powiedzieć, że jest pseudoniedeterministyczne (przez analogię do liczb pseudolosowych) z racji nieznajomości (przez wątek) algorytmu planowania przydziału procesora (który sam w sobie może być deterministyczny), o tyle w systemach rozproszonych mamy do czynienia z pełnym niedeterminizmem, wynikającym m.in. z różnic szybkości działania węzłów będących wynikiem rozbieżności w fizycznych parametrach generatorów częstotliwości, czy stanu środowiska zewnętrznego wpływającego na prawdopodobieństwo uszkodzenia transmitowanego komunikatu.

Dobrą metodą testowania wydajności staje się zatem po prostu jej pomiar podczas wykonywania pewnych wzorcowych programów, *benchmarków*, które, będąc prostymi rozwiązaniami, poddają pamięci transakcyjne obciążeniom podobnym do tych, jakich można się spodziewać po „prawdziwych” aplikacjach, będących w przyszłości wykorzystywać te mechanizmy. Z jednej strony metoda taka omija wspomniane wcześniej trudności. Z drugiej, zaraz po stosowaniu pamięci transakcyjnych w realnych aplikacjach – rozwiązaniu, które jest pracochłonne, a nie gwarantuje żadnego zysku – będzie ona najlepiej symulować rzeczywistość.

Pamięci transakcyjne są skomplikowanymi mechanizmami. Sposób konstrukcji wielu spośród ich elementów (a zwłaszcza każdego z osobna) może wpływać na całokształt wydajności. Stąd podejście wykorzystujące programy wzorcowe rozszerza się także na potrzeby ewaluacji pojedynczych komponentów takich pamięci lub też określonych scenariuszy ich wykorzystania. Programy takie, *mikrobenchmarki*, są tworzone w ten sposób, by były jeszcze prostsze od

ich pełnowymiarowych odpowiedników, nawet za cenę braku jakiegokolwiek odniesienia do rzeczywistych aplikacji.

1.4 Cel i zakres realizowanych zadań

Podstawowym problemem badanym w dziedzinie rozproszonej pamięci transakcyjnej jest porównanie w kontekście wydajności dwóch zasadniczo różnych podejść do problemu konfliktowych dostępów do danych, realizowanych w ramach transakcji, a tym samym dwóch różnych implementacji pamięci transakcyjnych dla systemów rozproszonych.

By tego dokonać, należy opracować *framework* pomiarowy, na który składają się: system pozwalający wykonać pomiary określonych parametrów oraz zestaw miar wydajności, których wartości można obliczyć w oparciu o uzyskane dane. Potrzebny będzie także zestaw programów rozproszonych generujących obciążenia podobne do tych spodziewanych po rzeczywistych aplikacjach.

Programy generujące obciążenia muszą spełniać szereg kryteriów. Dwa najbardziej istotne to możliwość przystosowania ich do współpracy z rozproszoną pamięcią transakcyjną oraz wspomniane właśnie generowanie realistycznych obciążeń. Na tym jednak zbiór kryteriów się nie kończy. Należy je zatem zebrać, sformalizować i przeanalizować pod kątem spełnialności.

Kwestia typowych zastosowań systemów rozproszonych pozostaje generalnie cały czas otwarta. O ile można mówić o wzorcowych zastosowaniach systemów równoległych, wynikających choćby z zaaplikowania opracowanych już metod zrównoleglania algorytmów do pewnych programów sekwencyjnych, o tyle ze względu choćby na różnorodność właściwości systemów rozproszonych (w tym np. 32 kombinacje asynchronizmu różnych ich elementów, wg. [11]), trudno jest mówić o praktycznych zastosowaniach, które dla ogółu systemów rozproszonych byłyby typowe. Z drugiej strony systemy rozproszone są jednak wykorzystywane w praktyce (np. [20, 40]), ale złożoność stosowanych tam rozwiązań z reguły wyklucza ich użycie w roli benchmarków. Ponadto skala rozwiązań (a więc fakt, że by ktokolwiek zdecydował się zastosować do rozwiązania problemu system rozproszony, sam problemu musi być odpowiednich rozmiarów) wymusza położenie nacisku na ich optymalizację, a tym samym rezygnację z silnej synchronizacji węzłów wszędzie tam, gdzie to możliwe. W efekcie może się okazać, że program dla systemu rozproszonego nie daje się zintegrować z systemem pamięci transakcyjnej lub produkuje obciążenia zbyt nikłe, by mogły ukazać badane cechy takiego systemu. Nie musi to jednak być prawdą w każdym przypadku. Stworzenie benchmarka bazującego na architekturze istniejącego systemu rozproszonego i intensywnie wykorzystującego pamięć transakcyjną pozwoli spełnić podstawowy wymóg stawiany dobremu programowi wzorcowemu – jego realizm. W połączeniu z wpisaną w pojęcie benchmarka prostotą, powstać może uniwersalne narzędzie do ewaluacji rozproszonych pamięci transakcyjnych, co jest przecież celem tej pracy.

Należy więc wyszukać problemy wymagające zastosowania systemów rozproszonych i poddać je ocenie względem wcześniej sformułowanych kryteriów (stopnia skomplikowania, zasadności stosowania pamięci transakcyjnych, itd.). Alternatywną drogą jest tutaj analiza możliwości „rozproszenia” programów generujących obciążenia, stosowanych już w testowaniu

pamięci transakcyjnych dla systemów wieloprocesorowych.

1.5 Układ pracy

Rozdział drugi ma za zadanie przedstawić szereg aspektów, na które należałoby zwrócić uwagę przy doborze i dostosowaniu programów testowych. W tym celu w pierwszej kolejności, zaraz po uściśleniu nomenklatury, przedstawione zostaną przykładowe implementacje pamięci transakcyjnych. Dalej pojawi się przegląd problemów poprawnościowych i wydajnościowych związanych z tymi systemami, wraz z określeniem sposobu ich rozwiązania. Wreszcie przejrzane zostaną benchmarki dostępne dla standardowych pamięci transakcyjnych i mikrobenchmarki dla pamięci rozproszonych.

Rozdział trzeci stanowi podbudowę teoretyczną pod prace implementacyjne. W pierwszej kolejności pojawią się tam elementy ściśle związane z systemem pomiarowym: wprowadzony zostanie model zdarzeniowy pamięci transakcyjnej, który następnie zostanie wpisany w kontekst systemów rozproszonych. Model taki posiada już swoją implementację, *Deuce*, która zostanie w tym rozdziale przedstawiona. W dalszej kolejności sformułowane zostaną wymagania względem programów generujących obciążenia, wskazana zostanie też ich sprzeczność, która wydaje się być niemożliwa do uniknięcia. Wreszcie analizie pod kątem możliwości „rozproszenia" poddane będą benchmarki opisywane w poprzednim rozdziale. Rozdział trzeci zakończy się podaniem propozycji benchmarków dla rozproszonych pamięci transakcyjnych, które stanowią główny rezultat niniejszego raportu.

W końcu, w kolejnym rozdziale krótko podsumowano kontrybucje raportu.

1.6 Uwagi edytorskie

Niniejszy raport, choć pisana w języku polskim, bazuje w przeważającej części na anglojęzycznej bibliografii. Rodzi to pewien problem, polegający na tym, że nie wszystkie angielskie terminy doczekały się już polskich odpowiedników. W dalszej części pracy rozwiązano go na trzy sposoby. Jeżeli pewien termin ma swój polski odpowiednik, który jest powszechnie znany (np. *deadlock* jako *zakleszczenie*), to ten odpowiednik został użyty w tekście. Gdy termin nie posiada jeszcze odpowiednika, zazwyczaj jest z pewną dozą dowolności tłumaczony na potrzeby tej pracy. Tłumaczenie takie w pierwszej kolejności ma oddawać sens pojęcia, a dopiero później odpowiadać słowo w słowo terminowi angielskiemu. Stąd np. *contention management policy* przetłumaczono jako *reguła arbitrażu*, zamiast *polityka zarządzania współzawodnictwem*. Angielski oryginał można zawsze znaleźć w miejscu pierwszego wystąpienia tłumaczenia. Wreszcie niektóre terminy wyjątkowo trudno będzie przetłumaczyć. W takiej sytuacji zwyczajnie je spolszczono, jak np. w przypadku *bucketu* („kubełka" tablicy haszowej), *overlapu* (nakładki, początkowego lub końcowego fragmentu segmentu, który nakłada się na inny), *frameworka* (szkieletu do budowy aplikacji lub systemu), czy *benchmarka* (programu wzorcowego, używanego do obciążenia pewnego systemu).

Algorytmy zostały w raporcie tak przedstawione, by można było łatwo odwoływać się do ich poszczególnych linii. Jeżeli pewna linia nie posiada numeru, oznacza to, że poprzednia nie

zmieściła się na stronie i została złamana, a bieżąca jest jej kontynuacją. Prezentowane algorytmy pochodzą z różnych źródeł, które używały różnych notacji. By uspójnić zapis, postanowiono je przetłumaczyć na pseudokod, łączący cechy języków imperatywnych C++ i Java. Jeżeli pewnych konstrukcji nie dało się wyrazić za pomocą tego języka, w blokach ograniczanych przez nawiasy klamrowe ujęto ich opis.

Rysunki i tabele również pochodzą z różnych źródeł. W każdym opisie takiego elementu podano informację o jego pochodzeniu. Gdy źródło podane jest w nim wprost, należy przyjąć, że oryginalny rysunek został w tym raporcie odwzorowany dokładnie lub z naniesionymi minimalnymi zmianami. Gdy mowa o opracowaniu własnym na podstawie określonego źródła, skala wprowadzonych zmian będzie większa. Zazwyczaj rysunki takie będą zawierały większą liczbę szczegółów, jednak przekazywana treść nadal będzie taka sama. Wreszcie gdy w opisie brak jakiejkolwiek informacji, należy przyjąć, że rysunek lub tabela zostały opracowane specjalnie na potrzeby raportu.

Część rysunków to diagramy przestrzenno-czasowe pokazujące wykonania programów współbieżnych i rozproszonych. Przyjęte w nich oznaczenia operacji odczytu i zapisu obiektów współdzielonych zostały zapożyczone z [35]. O ile jednak w przypadku [35, rysunek 2], `read(3)A` oznacza odczytanie z analizowanego właśnie obiektu wartości 3 przez proces `A`, w dalszej części pracy `A` będzie oznaczać odczytywany obiekt, a to jaki proces/wątek/węzeł go odczytuje ujęte zostanie w inny sposób. Operacje `read` pozbawione wartości, np. `read()X` oznaczać będą zlecenia wykonania odczytu (w tym przypadku: zmiennej `X`).

Wspomnienia wymaga też sposób nazwania benchmarków. Gdy tylko to możliwe, nazwy benchmarków podawane są w dokładnie takiej formie, jaką zawierają wprowadzające je źródła. Oznacza to m.in., że nazwy programów wzorcowych z zestawu STAMP zawsze pisane będą z małej litery. Podobne podejście zastosowano do nazw benchmarków wprowadzanych w tej pracy. Wynika to nie tylko z chęci zachowania spójności rozwiązań, ale także z konieczności rozróżnienia *cassandry* – benchmarka i *Cassandry* – aplikacji rozproszonej będącej jego pierwowzorem. W przypadku *genome*, wszędzie tam gdzie nie wynika to z kontekstu, o pierwowzorze będzie się mówić: *genome*, a o propozycji benchmarka dla rozproszonych pamięci transakcyjnych: rozproszona wersja *genome*.

Rozdział 2

Podstawy teoretyczne

Opisana w sekcji 1.1 czteroetapowa historia rozwoju pamięci transakcyjnych jest koncepcją stworzoną na potrzeby raportu. Choć trudno doszukać się podobnego podziału choćby w kompleksowej bibliografii przedmiotu, [9], znajduje on potwierdzenie zarówno w generalnie spodziewanych etapach prac nad jakimkolwiek nowym rozwiązaniem (tj. 1) propozycja, 2) rozwój-implementacja, 3) wytknięcie wad, 4) eliminacja wad – przy wszystkich zastrzeżeniach, jakie można żywić do tego typu metod „zdroworozsądkowych" jako narzędzi pracy naukowej), jak i w chronologii publikacji. Dodatkowo [58], publikacja prezentująca dorobek naukowy jednej z kluczowych dla pamięci transakcyjnych postaci – Mauricea Helihyego, wspomina o trzech pierwszych etapach, czwarty utożsamiając z przyszłym rozwojem tego mechanizmu. Stąd uzasadnione wydaje się uczynienie z tego podziału punktu wyjścia do następującego stwierdzenia, które dobrze oddaje myśl przewodnią tego raportu:

Systemy pamięci transakcyjnej posiadają niedopracowania. Zastrzeżenia do ich implementacji można mieć w dwóch głównych obszarach: poprawności i wydajności przetwarzania. O ile rozwiązania problemów w pierwszym obszarze są ze swojej natury przenośne ze świata systemów wieloprocesorowych do środowisk rozproszonych, o tyle problemy wydajnościowe, z racji zależności od kilku innych, poza sposobem implementacji systemów pamięci transakcyjnej, czynników, wymagają indywidualnego podejścia zarówno do samych implementacji, jak i systemów komputerowych, w których te implementacje funkcjonują.

Niniejszy rozdział przedstawia przykłady problemów poprawnościowych spotykanych w świecie pamięci transakcyjnych wraz z ich rozwiązaniami. W dalszej części koncentruje się na problematyce wydajności przetwarzania transakcyjnego, wymieniając sposoby jej badania i zatrzymując się przy jednym z nich – benchmarkach – stanowiącym temat tej pracy. Tutaj omawia przykłady wykorzystania tego sposobu badania wydajności – szereg programów wzorcowych dla systemów wieloprocesorowych, co stanowić ma podstawę dla opracowania podobnych rozwiązań działających w środowisku rozproszonym.

By jednak można to zrobić w sposób przejrzysty, należy wpierw zdefiniować samo pojęcie pamięci transakcyjnej, szereg innych terminów (co uprości późniejszy opis wielu problemów) oraz przedstawić, w ujęciu teoretycznym i praktycznym, implementacje systemu pamięci transakcyjnej. W ujęciu teoretycznym określony zostanie szereg *wyborów projektowych*

(ang. *design choices*) w sposób podobny do opisywanego w [53, str. 9]. Ujęcie praktyczne skupi się na algorytmach stanowiących podstawę dla analizowanych rzeczywistych systemów pamięci transakcyjnej, które są godne uwagi albo ze względów historycznych, albo z racji tego że kumulują wyjątkowo dużo zalet przetwarzania transakcyjnego.

2.1 Pamięć transakcyjna

W literaturze przedmiotu, w publikacjach, które starają się podsumować stan wiedzy w dziedzinie pamięci transakcyjnych (np. [26, 31] dla systemów wieloprocesorowych, [53] dla systemów rozproszonych) brakuje formalnej definicji pamięci transakcyjnej. Generalnie pisze się o *systemie* lub *mechanizmie sterowania współbieżnością* (ang. *concurrency control mechanism*), który z perspektywy programisty umożliwia definiowanie *sekcji atomowych* – grup operacji wykonywanych sekwencyjnie przez pojedynczy wątek na współdzielonym zasobie (przede wszystkim pamięci operacyjnej, ale też np. na grupie obiektów – oczywistość stosowania transakcji na współdzielonej pamięci operacyjnej przestanie obowiązywać przy przejściu do systemów rozproszonych) w taki sposób, by z perspektywy innych, współbieżnie wykonywanych wątków nie dało się zaobserwować stanów przejściowych, będących wynikiem wprowadzenia jedynie części zmian realizowanych przez pojedynczą sekcję. Innymi słowy, kod objęty sekcją atomową będzie wykonywany spójnie, tj. tak, by ani w trakcie, ani po jego wykonaniu nie dało się zaobserwować naruszenia warunków (niezmienników, ang. *invariants*) określających, kiedy stan systemu (a dokładnie: współdzielonej pamięci operacyjnej) jest spójny. Ujęcie takie, w świetle problemów poprawnościowych i różnych podejść implementacyjnych, omówionych później można rozszerzyć, zaznaczając, że stany przejściowe, potencjalnie niespójne, mogą być przez wątki obserwowane, ale nie mogą ani wpływać na dalsze przetwarzanie w obrębie tych wątków, ani, tym bardziej, prowadzić do anomalii przetwarzanie uniemożliwiających (nieskończone pętle, błędne odwołania do pamięci, itp.).

Sekcje atomowe same w sobie są deklaratywną konstrukcją językową (z przypisaną jej semantyką), jednak na ich znaczenie w kontekście tego raportu wpływa fakt, że można je w trywialny sposób implementować przy użyciu transakcji, dającej się wprost zastosować do paradygmatu programowania imperatywnego. Przykład przejścia od deklaratywnych sekcji atomowych do transakcji prezentują konstrukcje przedstawione na rysunku 2.1.

Należy podkreślić, że sekcje atomowe, są jedynie koncepcją, definiującą ogólny interfejs programistyczny (początek i koniec sekcji) oraz semantykę (spójne wykonanie wszystkich operacji objętych sekcją, co nie wyklucza współbieżności między różnymi sekcjami w przeciwieństwie do np. *sekcji krytycznych*, implementowanych w oparciu o wzajemne wykluczanie). Oznaczać by to mogło, że transakcje nie są jedynym sposobem ich implementacji. Istnieje jednak silny związek pomiędzy sekcjami atomowymi jako koncepcją i (optymistycznymi) pamięciami transakcyjnymi (sprzętowymi lub programowymi) jako ich implementacją [16]. Stąd taka, a nie inna „definicja" pamięci transakcyjnej wprowadzana na potrzeby tego raportu. Jej dalsza część skupiać się będzie jednak raczej na rozwiązaniach, w których programista korzysta z interfejsu systemu pamięci transakcyjnej bezpośrednio.

Na interfejs programistyczny systemu pamięci transakcyjnej składają się następujące operacje (zapisane w notacji wzorowanej na języku C++):

```
1   atomic {
2       ...
3       ...
4       ...
5       ...
6       ...
7       {polecenia};
8       ...
9       ...
10      ...
11      ...
12      ...
13  }
```

```
1   boolean done = false;
2   while (!done) {
3       start();
4       try {
5           ...
6           ...
7           {polecenia};
8           ...
9           ...
10          done = commit();
11      } catch (Throwable t) {
12          done = commit ();
13          if (done) {
14              throw t;
15          } else {
16              // zatwierdzenie
17              // zakończone
18              // niepowodzeniem
19              // implikuje wywołanie
20              // abort()
21          }
22      }
23  }
```

Rysunek 2.1: Implementacja sekcji atomowej z wykorzystaniem transakcji w języku programowania obsługującym wyjątki. Opracowanie własne na podstawie: [30].

- `void start()` – wywołanie tej procedury wskazuje moment i miejsce rozpoczęcia transakcji. Od tej chwili wszystkie dostępy do współdzielonego zasobu realizowane są w sposób transakcyjny, tj. jeżeli zachodzą za pośrednictwem systemu pamięci transakcyjnej, to zapewnia to pożądaną atomowość.
- `T read(T *addr)` – odczyt wartości spod adresu w pamięci współdzielonej w sposób transakcyjny powinien przede wszystkim gwarantować postrzeganie przez transakcję spójnego stanu systemu. Warto wspomnieć, że wątek może posiadać fragment pamięci współdzielonej, z której korzysta w sposób wyłączny. Odczyt danych z takich zasobów może (choć nie musi) odbywać się z pominięciem pamięci transakcyjnej.
- `void write(T *addr, T value)` – w przypadku modyfikacji zasobu współdzielonego realizowanych w sposób transakcyjny, pamięć transakcyjna powinna zostać powiadomiona o wystąpieniu każdego takiego zdarzenia choćby po to, by mogła wykryć i rozwiązać konflikty przy współbieżnym dostępie do danych. Należy też pamiętać że może zaistnieć potrzeba powtórzenia wykonania transakcji. Być może pamięć transakcyjna powinna zostać powiadomiona o zapisach do puli zasobów prywatnych wątku i, przy wycofywaniu zmian przed powtórzeniem, zapewnić anulowanie także takich modyfikacji.
- `bool commit()` – ta procedura powoduje zakończenie transakcji poprzez zatwierdzenie zmian wprowadzonych w zasobach współdzielonych. W wyniku jej wykonania pozostałe, działające współbieżnie wątki powinny zacząć postrzegać komplet wprowadzonych transakcyjnie zmian w sposób niepodzielny. `commit()` przez zwracaną wartość informuje o tym, czy zatwierdzanie zmian powiodło się, czy też było niemożliwe np. ze względu na wystąpienie konfliktów i przegranie przez transakcję arbitrażu. W wy-

padku gdy `commit()` zwraca wartość `FALSE`, transakcja jest zazwyczaj (tj. w większości implementacji) niejawnie i automatycznie wycofywana.

- `void abort()` – operacja `abort()` wywoływana w trakcie działania transakcji lub po nieudanej próbie jej zatwierdzenia wycofuje zmiany, które zostały przez nią wprowadzone. Pozostałe wątki powinny postrzegać (na potrzeby kontynuowania przetwarzania) stan, który nigdy nie nosił i nie będzie nosił śladów wykonania transakcji. To czy operacja `abort()` jest wywoływana automatycznie po porażce zatwierdzania transakcji (z wewnątrz procedury `commit()` zwracającej `FALSE`), czy też jej wywołanie w takim przypadku jest zadaniem programisty to cecha charakterystyczna implementacji pamięci transakcyjnej.

Niektóre warianty interfejsu programistycznego (np. [61]) opisują także procedurę `bool retry(int maxRetries, long timeout)`, której zadaniem jest powtórzenie wykonania transakcji na życzenie programisty. To udogodnienie wydaje się jednak mało przydatne. Z jednej strony możliwe jest zaimplementowanie bloków atomowych bez użycia tej operacji, co pokazuje rysunek 2.1. Z drugiej – w przypadku udostępnienia operacji programiście – warunkowe powtarzanie transakcji (ponieważ nie sposób wskazać zastosowanie dla bezwarunkowego) mogłoby posłużyć co najwyżej do implementacji *pollingu*, odpytywania w środowisku transakcyjnym. W dalszej części raportu pokazany zostanie przykład, w którym takie podejście może okazać się potrzebne. Przykład ten dotyczy jednak specyficznego problemu aplikacyjnego, wykorzystującego specyficzny system pamięci transakcyjnej. W standardowych systemach wieloprocesorowych polling może śmiało zostać zastąpiony mechanizmami synchronizacji kodu – monitorami, zmiennymi warunkowymi, barierami, itp. Proponowane w ramach tego raportu programy wzorcowe niejako z przymusu, jako kompleksowe i złożone aplikacje, będą łączyć synchronizację dostępu do danych współdzielonych z synchronizacją kodu.

Mając za punkt wyjścia bloki atomowe można wskazać dwa kierunki rozwoju programowania transakcyjnego. Idąc w stronę upraszczania zadań programisty dojść można do nowego paradygmatu programowania, w którym wątki traktowane są jako sekwencje bloków atomowych, a rolą programisty jest jedynie wskazanie (np. poprzez wywołanie odpowiedniej procedury) granic między takimi blokami (por. [31, pkt. 3.4.1]). Kierując się zaś w drugą stronę – udostępniając interfejs transakcyjny programiście, kosztem komplikacji programowania współbieżnego można liczyć na lepszą wydajność przetwarzania spowodowaną np. wczesnym wycofywaniem transakcji nie z racji wystąpienia konfliktów, ale z powodu spełnienia (bądź nie) stawianego przez programistę warunku (tzw. niewymuszone wycofania, ang. *non-forcible rollbacks*).

Koncepcja wątków jako sekwencji bloków atomowych oraz interfejs programistyczny pamięci transakcyjnej, zwłaszcza operacje `read()` i `write()`, które mogą ale nie muszą automatycznie zastępować instrukcje odczytu i zapisu wewnątrz transakcji, wskazują na jeden z wielu wyborów istniejących przy projektowaniu systemów pamięci transakcyjnych, z których kilka opisanych zostanie w dalszej części raportu. Można bowiem zadać sobie pytanie, czy nietransakcyjny dostęp do współdzielonych zasobów jest dopuszczalny, czy też nie. Skrajnym przypadkiem jest tu wątek jako sekwencja bloków atomowych, w którym problem nie występuje (w celu uproszczenia programowania zakładamy, że wszystkie dostępy do pamięci współdzie-

lonej z wewnątrz bloku są transakcyjne, a dodatkowo, skoro wątek jest sekwencją transakcji, nie ma miejsca na podobne dostępy spoza którejkolwiek z nich). Jednak jeżeli w kodzie wątku mogą istnieć odwołania do współdzielonych zasobów nieobjęte żadną transakcją, należy zdecydować czy każdy taki dostęp powinien być automatycznie objęty miniaturową, jednoelementową transakcją (co gwarantuje tzw. *silną atomowość*, ang. *strong atomicity*), czy może należy pozwolić programiście ocenić sytuację. Ten może bowiem, znając algorytm pamięci transakcyjnej, uodpornić kod nietransakcyjny na niespójności stanu pojawiające się w trakcie zatwierdzania transakcji. Może też we własnym zakresie zapewnić, że nigdy nie wystąpi sytuacja, w której pewne odwołanie transakcyjne i inne, nietransakcyjne będzie dotyczyć tego samego zasobu (tj. istnieć będą dwie sekcje pamięci współdzielonej: transakcyjna i druga, z dostępem synchronizowanym w inny sposób lub wcale). Wreszcie programista, wiedząc np. że pewne fragmenty pamięci (komórki, struktury, obiekty, itp.) na czas zatwierdzania transakcji są wyłączane z użycia przy pomocy zamków (dla *pamięci transakcyjnych bazujących na zamkach*, ang. *lock-based STMs*) może we własnym zakresie, tj. pozatransakcyjnie próbować potrzebny mu zamek zamknąć. Jeżeli pojedyncze dostępy do współdzielonych zasobów nie są niejawnie realizowane za pośrednictwem TM, mówi się o *słabej atomowości* (ang. *weak atomicity*). Szczegółowa dyskusja tej klasy problemów jest treścią punktu 2.4.3 w dalszej części raportu.

By można było przejść do opisu implementacji i problemów związanych z pamięciami transakcyjnymi, należy w pierwszej kolejności uściślić nazewnictwo. Dodatkowo wprowadzone tutaj skróty uproszczą wiele prezentowanych dalej opisów. I tak:

- **pamięć transakcyjna**, opisana wcześniej w tym punkcie oznaczana będzie skrótem TM, od angielskiego terminu *transactional memory*. Poprzez **system pamięci transakcyjnej** będziemy rozumieć implementację algorytmu pozwalającego przeprowadzać współbieżne transakcje na puli zasobów współdzielonych (pamięci operacyjnej, zbiorze obiektów, itp.), działającą w określonym systemie komputerowym (wieloprocesorowym lub rozproszonym),
- **sprzętową pamięcią transakcyjną** (ang. *hardware transactional memory*, HTM) nazwiemy taką pamięć transakcyjną, która wszystkie kluczowe dla pamięci transakcyjnej elementy (m.in. detekcję konfliktów, arbitraż) realizuje przy użyciu wsparcia sprzętowego, np. w odpowiedni sposób rozszerzając funkcjonalność protokołu koherencji działającego pomiędzy pamięciami podręcznymi procesorów w systemie wieloprocesorowym,
- **pamięcią transakcyjną hybrydową** (ang. *hybrid transactional memory*, HyTM) nazwiemy pamięć transakcyjną, która w ograniczonym zakresie korzysta ze wsparcia sprzętowego. Pewne kluczowe komponenty algorytmu TM będą musiały zatem być implementowane programowo. Taka sytuacja wystąpi przykładowo wtedy gdy konflikty będą wykrywane przy użyciu pewnych sygnatur, powiązanych z lokalizacjami w pamięci współdzielonej i implementowanych sprzętowo, natomiast algorytm rozstrzygania konfliktów jest na tyle złożony (lub zmienny w czasie/zależny od historii wykonania), że nie może zostać sprzętowo zaimplementowany,
- **programową pamięcią transakcyjną** (ang. *software transactional memory*, STM) będziemy nazywać taką pamięć transakcyjną, której implementacja nie korzysta z żad-

nej specyficznej formy wsparcia sprzętowego. W przypadku STM możliwe, pożądane, a nawet konieczne będzie zmodyfikowanie na jej potrzeby kompilatora języka programowania i/lub środowiska uruchomieniowego,

- **obiekt transakcyjny**, **t-obiekt** (ang. *t-object*) to podstawowa jednostka podziału pamięci współdzielonej (w ogólności: puli zasobów współdzielonych) na potrzeby wykonywania transakcji. Do t-obiektu dostęp może uzyskać jedynie transakcja (jeżeli ten sam element zasobu współdzielonego jest używany raz transakcyjnie, a raz nietransakcyjnie, to będziemy przyjmować, że jest on t-obiektem tylko w kontekście dostępu transakcyjnego). Przykładowo t-obiektem może być pojedyncza komórka pamięci – słowo (*word*), linia pamięci podręcznej, struktura danych lub jej fragment, czy wreszcie obiekt języka programowania obiektowego. Dobór wielkości obiektu transakcyjnego powinien wynikać z przyjęcia kompromisu pomiędzy wielkością narzutów pamięciowych (zapewne z każdym obiektem transakcyjnym związana będzie dodatkowa struktura danych, np. zamek – stąd im mniejsze obiekty tym jest ich więcej, a zatem więcej jest struktur kontrolnych) a ryzykiem występowania fałszywych konfliktów (t-obiekt w całości jest powodem konfliktu, nawet jeżeli zmiana dotyczy tylko jego fragmentu – może się okazać, że jedna transakcja modyfikuje zupełnie inny fragment t-obiektu niż ten odczytywany przez drugą transakcję – gdyby t-obiekt był mniejszy, być może konfliktu dałoby się uniknąć),
- **dostęp transakcyjny** (ang. *t-access*) do pamięci współdzielonej będzie to operacja odczytu lub zapisu zlecana z poziomu kodu objętego transakcją a wykonywana przez system pamięci transakcyjnej na obiekcie transakcyjnym. Zlecenia dostępu do pamięci współdzielonej z wewnątrz transakcji mogą być tłumaczone na dostępy transakcyjne do t-obiektów w sposób automatyczny na etapie kompilacji lub wykonania; w przypadku tłumaczenia na etapie wykonania także jednorazowo (instrumentacja kodu ładowanego do pamięci operacyjnej) lub każdorazowo (mechanizm *reflection*)[1]. Programista może też jawnie wykorzystywać procedury `read()` i `write()` interfejsu TM co ma tę zaletę, że na pewnych etapach przetwarzania fragmenty pamięci współdzielonej mogą występować w roli t-obiektów, a na innych nie (przykładowo mogą być wtedy zawłaszczone przez jeden wątek). Gdy programista wie, że aktualny wątek posiada fragmenty pamięci współdzielonej na własność, a musi z nich korzystać wewnątrz transakcji, może usunąć niepotrzebne narzuty poprzez rezygnację z dostępu do nich za pośrednictwem transakcyjnych operacji `read()` i `write()`,
- transakcja może zostać powtórzona, jeżeli za pierwszym razem nie uda się jej zatwierdzić zmian. **Read-set** będzie zbiorem t-obiektów, które są odczytywane w ramach danego powtórzenia transakcji. Stany t-obiektów z read-setu powinny odpowiadać spójnemu stanowi pamięci współdzielonej. **Write-set** to zbiór t-obiektów, które są modyfikowane w ramach danego powtórzenia transakcji. System pamięci transakcyjnej powinien zapewnić, by wątki współbieżne do tego, który wykonuje transakcję albo nie widziały żadnej ze zmian aktualnie wprowadzanych do t-obiektów z write-setu, albo widziały wszystkie. Jeżeli z pewnych względów nie będzie można odróżnić operacji odczytu t-obiektu od operacji jego modyfikacji, czy to w warstwie interfejsu programi-

[1] porównanie wydajnościowe podejść dostępne jest w [53, punkt 3.3.1].

stycznego pamięci transakcyjnej (`access()` zamiast podziału na `read()` i `write()`), czy też w momencie, w którym system pamięci transakcyjnej operuje na fragmencie pamięci współdzielonej lub innym zasobie (jak w przypadku mechanizmu zdalnego wywoływania procedur, gdzie *gettery* i *settery* są nierozróżnialne), zamiast o read-secie i write-secie będziemy mówić o **access-secie**, a wszystkie zawarte w nim t-obiekty będą potencjalnie modyfikowane przez transakcję. Za moment włączenia t-obiektu do read-, write- lub access-setu przyjmowane będzie pierwsze odwołanie do t-obiektu, chyba, że system pamięci transakcyjnej wymaga określenia zawartości tych zbiorów każdorazowo przed rozpoczęciem transakcji. Wówczas t-obiekty pojawiają się w odpowiednich zbiorach w momencie rozpoczęcia jej wykonywania. Na potrzeby dalszych definicji należy jeszcze wprowadzić pojęcie **rw-setu** jako sumy dwóch zbiorów: read-setu i write-setu lub zbioru tożsamego z access-setem,

- powiemy, że dwie transakcje są w **konflikcie** ze względu na określony t-obiekt jeżeli obie z nich wykonywane są współbieżnie, obie operują na tym samym t-obiekcie oraz co najmniej jedna z transakcji modyfikuje jego stan. Współbieżne wykonywanie transakcji oznacza, że jedna z nich rozpoczyna wykonanie zanim druga zatwierdzi wprowadzone przez siebie zmiany[2]. Jest to podejście intuicyjne. Formalną definicję konfliktu podaje [26, punkt 3.2.3]: transakcje $\mathtt{t}_1$ i $\mathtt{t}_2$ są ze sobą w konflikcie ze względu na pewien t-obiekt `O`, jeżeli $\mathtt{t}_1$ i $\mathtt{t}_2$ są współbieżne i na pewnym etapie wykonania obiekt `O` znajduje się zarówno we write-secie $\mathtt{t}_1$ i w rw-secie $\mathtt{t}_2$ lub odwrotnie – w rw-secie $\mathtt{t}_1$ i write-secie $\mathtt{t}_2$. W ogólności $\mathtt{t}_1$ i $\mathtt{t}_2$ znajdują się w konflikcie, jeżeli są ze sobą w konflikcie ze względu na co najmniej jeden t-obiekt,
- istotą konfliktu jest to, że by zapewnić poprawność przetwarzania, jedna z dwóch transakcji w niego zaangażowanych musi zostać wycofana i wykonana w całości ponownie. Alternatywą jest opóźnienie jednej z nich do momentu, w którym warunki dla powstania konfliktu przestaną istnieć. W szczególności stanie się tak, gdy druga zatwierdzi wprowadzone zmiany. Ponieważ zazwyczaj nie będzie istnieć ścisła preferencja dotycząca tego, którą transakcję wycofać/opóźnić (tj. od tego, że właśnie ta, a nie inna transakcja „przegra” nie będzie zależała poprawność przetwarzania), niezbędne jest obranie pewnej polityki, która, będąc najczęściej algorytmem heurystycznym, udzieli odpowiedzi na pytanie: którą transakcję wycofać lub opóźnić? Politykę taką nazwiemy od angielskiego terminu *polityką zarządzania współzawodnictwem* (ang. *contention management policy*) lub wprowadzonym na potrzeby raportu polskim terminem: **reguła arbitrażu**. Element systemu TM, który z takiej reguły korzysta, nazywany będzie *menedżerem współzawodnictwa* (ang. *contention manager*), *zarządcą konfliktów* (ang. *conflict manager*), a na potrzeby raportu – **arbitrem**. Rozszerzenie pojęcia konfliktu na więcej niż dwie transakcje oraz podanie pożądanych własności reguły arbitrażu nastąpi przy okazji wprowadzenia pojęcia *silnego warunku postępu* (ang. *strong progressiveness*)

[2] przy użyciu modelu procesu sekwencyjnego pochodzącego z [11], a opisywanego szerzej w punkcie 3.3.1 w raporcie, można ująć rzecz w sposób formalny: jeżeli założyć, że każdej operacji wykonywanej w ramach dowolnej transakcji można przypisać unikatowy moment czasu rzeczywistego, w którym operacja ta zachodzi, to dwie współbieżne transakcje: $\mathtt{t}_1$ i $\mathtt{t}_2$ nie mogą być wykonywane sekwencyjnie. Nie są, kiedy następujący warunek jest spełniony: ostatnie zdarzenie w $\mathtt{t}_1$ zachodzi wcześniej niż pierwsze zdarzenie w $\mathtt{t}_2$ wtedy i tylko wtedy gdy relacja pomiędzy pierwszym zdarzeniem w $\mathtt{t}_2$ a ostatnim w $\mathtt{t}_1$ jest dokładnie odwrotna.

w dalszej części raportu,

- niniejszy raport poza standardowymi pamięciami transakcyjnymi dla systemów wieloprocesorowych koncentruje się także na zastosowaniu pochodzących stamtąd koncepcji w systemach rozproszonych. Pamięci transakcyjne dla takich systemów będziemy nazywać **rozproszonymi pamięciami transakcyjnymi**. W literaturze przedmiotu pojawiają się stwierdzenia, że różnorodność i złożoność reguł arbitrażu, wielość topologii systemów rozproszonych, czy też ich możliwy heterogeniczny charakter (wtedy, gdy system rozproszony powstaje z połączenia węzłów o różnych możliwościach) w praktyce wyklucza wykorzystanie wsparcia sprzętowego dla rozproszonych pamięci transakcyjnych. Twierdzenie takie wydaje się tym bardziej uzasadnione, że newralgicznym elementem systemu rozproszonego, implementowanym programowo i odróżniającym takie systemy od np. systemów sieciowych jest warstwa *middleware*, zapewniająca kluczową cechę systemu rozproszonego jaką jest przezroczystość jego użytkowania w różnych aspektach. Rozproszone pamięci transakcyjne albo będące elementem middleware, albo działające na bazie tej warstwy będą to więc raczej programowe pamięci transakcyjne. Stąd skrótem wykorzystywanym w dalszej części raportu dla oznaczenia rozproszonych pamięci transakcyjnych będzie **D-STM**.

Wreszcie by dopełnić prezentowany obraz pamięci transakcyjnej, należy przedstawić jej pożądane własności. Nie (jeszcze nie) w ujęciu formalnym, pod postacią kryteriów poprawności jej implementacji, ale w szerszym i ogólniejszym sensie.

- Pamięć transakcyjna, jako alternatywa dla innych mechanizmów wykorzystywanych w programowaniu współbieżnym i rozproszonym powinna być prosta w obsłudze. Prostota wykorzystania niekoniecznie musi sprowadzać się do ubogiego interfejsu programistycznego (co zostanie pokazane na przykładzie D-STM wymagającej znajomości access-setu *a priori*). Programista powinien móc po prostu stosunkowo łatwo tworzyć programy bez skupiania się na rozwiązaniu problemu synchronizacji współbieżnych dostępów, w więc konieczności podejmowania decyzji w związku ze sterowaniem współbieżnością. Wymaganie to można rozszerzyć zabraniając podejścia zmuszającego programistę do poznania algorytmów leżących u podstaw systemów TM w celu pisania dobrych wydajnościowo albo – co gorsza – poprawnych programów. Jednak w ogólności będzie to dla niektórych pamięci transakcyjnych niemożliwe (por. późniejszy przykład błędnego wykorzystania funkcji wczesnego zwalniania t-obiektu, punkt 2.4.5), a w szczególności – dla tego raportu – niepraktyczne, bowiem jej celem jest wytworzenie takich programów, które mają odpowiednio dociążać pamięci transakcyjne, próbując w ten sposób wyszukać słabe punkty poszczególnych implementacji.
- Pamięć transakcyjna ma zwiększyć wydajność przetwarzania. Oczywiście nie zawsze będzie to możliwe. Można jednak wskazać ogólne kryteria wydajnościowe, które dobry system TM powinien spełniać w określonych okolicznościach. W szczególności system TM, pomimo narzutów czasowych, których obecność jest jego integralną cechą, powinien być wydajnościowo porównywalny (lub lepszy) z rozwiązaniami bazującymi na globalnym zamku współdzielonym, zwłaszcza w przypadku dużej liczby wątków, wysokiego poziomu współzawodnictwa (tu rozumianego jako duża liczba współbieżnych

transakcji, co nie wynika wprost z dużej liczby wątków, ponieważ wątek poza wykonywaniem transakcji może realizować też inne operacje np. na lokalnych danych) i małej liczby konfliktów. Cel ten dobrze ujmuje Scott w [58, punkt 5], stwierdzając, że pojedyncza sekcja atomowa implementowana przy użyciu STM wykonuje się z reguły ok. 3–5 razy dłużej niż odpowiadająca jej sekcja krytyczna (i nie wygląda na to, by wynik ten dało się znacząco poprawić), a celem projektantów systemów pamięci transakcyjnej jest to, by co najmniej zrekompensować to spowolnienie wzrostem współbieżności wykonywania transakcji.

- Bez względu na to, czy w rezultacie służy to zwiększeniu szybkości przetwarzania, czy nie, pamięć transakcyjna powinna, pomimo gwarantowania atomowości wykonania transakcji, umożliwiać ich współbieżną realizację. Jest to podstawową cechą odróżniającą sekcje atomowe, implementowane przy użyciu pamięci transakcyjnej od klasycznych sekcji krytycznych.

2.2 Wybory projektowe i taksonomia systemów pamięci transakcyjnych

Pamięć transakcyjna była i nadal jest tematem licznych badań naukowych [1, 9]. Rezultatem tak wytężonego wysiłku jest m.in. powstanie wielu różnych koncepcji leżących u podstaw różnorodnych algorytmów TM.

O ile można spodziewać się, że ciągły rozwój w tej dziedzinie będzie prowadził do stopniowego eliminowania wad pamięci transakcyjnych, o tyle trudno jest mówić o przyroście przyroście jakości z perspektywy pojedynczego systemu TM. Owszem, pisze się o implementacjach *state-of-the-art* (np. *SwissTM* [22]), które czerpią z najlepszych i najbardziej aktualnych spośród proponowanych rozwiązań, jednak w ogólności jedne rozwiązania niekiedy wykluczają stosowanie innych. Prowadzi to do powstania wielu różnych implementacji TM, przydatnych każda do innych celów. Zjawisko to nasili się jeszcze w przypadku systemów rozproszonych, dla których ze względu na złożoność stosowanych tam rozwiązań i różnorodność właściwości (niech za przykład posłużą tu znowu 32 kombinacje asynchronizmu różnych elementów systemu rozproszonego [11]), nieodłącznym pojęciem będzie *trade-off*, i jego „rozwiązanie”, *kompromis* np. pomiędzy złożonością obliczeniową (czasową) a komunikacyjną (typowe rozgłaszanie, ang. *broadcast*, kontra algorytmy bazujące na plotkowaniu, ang. *gossiping*), złożonością obliczeniową a odpornością na awarie (koszt implementacji i wykonania rozwiązań uodparniających), prostotą topologii a odpornością na awarie (im więcej łączy komunikacyjnych, tym lepsza tolerancja awarii jednego z nich), itp.

Istnienie trade-offów dotyczy nie tylko środowiska, w którym działa pamięć transakcyjna, ale także niej samej. Kompromisy będą tu miały bardziej binarny charakter (nie będą to zatem kompromisy, a wybory jednej ze skrajności, z których każdej będzie można wytknąć wady i zalety), sprowadzone zatem zostaną do roli wyborów projektowych, a same implementacje TM – do zbioru takich wyborów, dokonywanych w oparciu o przewidywania co będzie lepsze, ogólne lub dla określonego zastosowania. Przykład takiego podejścia podaje [53, str. 9]. Saad

przytacza tam koncept budowy rozproszonej pamięci transakcyjnej w oparciu o rozbicie szeregu implementacji wieloprocesorowych na wybory projektowe i wyeliminowanie pewnych rozwiązań na rzecz innych ze względu na rozproszony charakter systemu docelowego.

Zestawienia wyborów projektowych dla wieloprocesorowych i rozproszonych pamięci transakcyjnych można znaleźć odpowiednio w [31, rozdział 2.1] i [53, rozdział 2.3]. Na potrzeby tego raportu wybrane i przedstawione zostaną te wybory projektowe, które 1) są na tyle uniwersalne, że stosują się do większości zastosowań STM/D-STM, 2) są istotne w systemach rozproszonych oraz 3) będą miały znaczenie przy omawianiu wykorzystywanych w raporcie implementacji D-STM. Poza uproszczoną próbą usystematyzowania wiedzy z tej dziedziny, celem takiego zestawienia będzie ponadto pokazanie wzrostu złożoności projektowej D-STM względem STM (część wyborów dotyczyć będzie STM i D-STM, ale część już tylko D-STM).

Za punkt wyjścia można przyjąć dwa intuicyjne sposoby wykonania kodu transakcyjnego z perspektywy pojedynczej transakcji:

1. Transakcja dokonuje modyfikacji bezpośrednio na pamięci współdzielonej. By zapewnić niepodzielność wprowadzanych zmian, transakcja w czasie swojego wykonania instruuje arbitra, by ten opóźniał lub wycofywał inne transakcje będące z pierwszą w konflikcie. Jeżeli jest to niemożliwe, transakcja sama zostaje wycofana. Procedura `commit()` polega wtedy m.in. na dopuszczeniu innych transakcji do modyfikowanych przez siebie obszarów pamięci poprzez ich zwolnienie.
2. Transakcja buforuje dokonane przez siebie zmiany, pozwalając innym transakcjom na współbieżne korzystanie z docelowo modyfikowanych obszarów pamięci. Procedura `commit()` polega tu na wprowadzeniu buforowanych zmian do pamięci współdzielonej. Arbiter, mając do dyspozycji wykazy zmian wprowadzanych przez transakcje mogące być w konflikcie musi ocenić, czy należy którąkolwiek z nich wycofać lub opóźnić (a dokładnie: opóźnić zastosowanie zbuforowanych przez nią zmian do pamięci współdzielonej), a jeżeli tak, to którą.

Takie intuicyjne pojmowanie implementacji systemów pamięci transakcyjnych da nam pole do wprowadzenia bardziej usystematyzowanego opisu.

Eager vs. lazy versioning. Rozróżnienie to jest równoważne wyróżnieniu dwóch, opisanych wyżej podejść. *Eager versioning*, *eager version management*, czy też *wczesne zarządzanie wersjami* sprowadza się do rezygnacji z buforowania transakcyjnych zapisów do momentu zatwierdzenia transakcji. Zmiany dokonywane są bezpośrednio (bezpośrednio w czasie, tj. w momencie gdy wykonuje je transakcja, jednak za pośrednictwem systemu pamięci transakcyjnej) na pamięci współdzielonej. Są to tzw. *direct updates* (w przeciwieństwie do *indirect updates*) lub *in-place updates*. Jeżeli konstrukcja systemu pamięci transakcyjnej przewiduje konieczność wycofania transakcji (w wyniku wystąpienia konfliktów, *forcible rollbacks*) lub jedynie taką możliwość (na życzenie programisty, *non-forcible rollbacks*), TM musi utrzymywać tzw. *undo-log*, zestawienie starych wartości zmodyfikowanych lokalizacji pamięci współdzielonej w celu ewentualnego ich odtworzenia.

W [31] znaleźć można stwierdzenie, że jeżeli w użyciu jest wczesne zarządzanie wersjami, fakt ten wymusza stosowanie określonej formy mechanizmu sterowania współbieżnością, na-

zywanej pesymistyczną.

Dla porównania późne zarządzanie wersjami wymusza stosowanie bufora modyfikacji, *redologu*, który w przypadku wycofania transakcji zostaje po prostu usunięty, natomiast w trakcie zatwierdzania jest atomowo nanoszony na pamięć współdzieloną. W kwestii odczytów system pamięci transakcyjnej musi zapewnić 1) gwarancję *read-your-writes*, co oznacza, że przy odczycie lokalizacji w pamięci współdzielonej należy przeszukać redo-log by odczytana wartość uwzględniała modyfikacje poczynione wcześniej w ramach transakcji, oraz 2) by odczyty, jeżeli pochodzą bezpośrednio z pamięci współdzielonej, odpowiadały jej stanowi spójnemu, tj. wartości lokalizacji będących przyczyną konfliktów pochodziły albo wszystkie sprzed zatwierdzenia tej drugiej transakcji, albo wszystkie były odczytywane po jej zatwierdzeniu.

Pessimistic vs. optimistic concurrency control. Warunki wystąpienia konfliktu pomiędzy transakcjami podano wcześniej, przy okazji wprowadzenia tego pojęcia (punkt 2.1). O konflikcie powiemy, że *zaszedł* w momencie, gdy dwie transakcje odwołują się „w sposób konfliktowy" do lokalizacji w pamięci współdzielonej. Ponieważ do zaistnienia pojedynczego konfliktu pomiędzy dwiema transakcjami potrzebne są dwa takie dostępy, momentem czasu rzeczywistego, w którym konflikt zajdzie będzie czas drugiego z nich. Konflikt nie musi być wykryty w momencie wystąpienia (np. z powodu buforowania modyfikacji). O *wykryciu* konfliktu powiemy gdy system TM sam ustali lub otrzyma informację, że taki konflikt wystąpił (teraz lub w przeszłości, jednak cały czas w ramach pojedynczej transakcji) np. w oparciu o analizę kontrolnych struktur danych skojarzonych z lokalizacjami w pamięci współdzielonej. Konflikt zostaje *rozwiązany*, gdy system TM (wdrażając decyzję arbitra) podejmuje pewną akcję mającą na celu usunięcie przyczyn jego powstania. Może on wycofać jedną z konfliktowych transakcji lub opóźnić jej wykonanie do czasu zatwierdzenia (lub wycofania) drugiej.

Należy zwrócić uwagę na fakt, że trzy zdarzenia związane ze sterowaniem współbieżnością, jakimi są wystąpienie, wykrycie i rozwiązanie konfliktu mogą wystąpić w różnych momentach czasu, na różnych etapach działania transakcji ale nigdy w kolejności innej niż podana.

Definicja *pesymistycznego mechanizmu sterowania współbieżnością* zawarta w [31] podaje, że jest to taki mechanizm, w którym trzy wspomniane zdarzenia zachodzą w tym samym momencie. Analogicznie *optymistyczny mechanizm sterowania współbieżnością* pozwala opóźnić w czasie wykrycie lub rozwiązanie konfliktu. Pesymizm pierwszego rozwiązania miałby objawiać się w podejrzliwym stosunku do każdego dostępu do pamięci współdzielonej będącego potencjalnie źródłem konfliktu. Jeżeli jego wykrycie (przy okazji in-place update) wymaga np. zablokowania lokalizacji w pamięci współdzielonej i utrzymania tej blokady do momentu zatwierdzania, to lepiej jest wykryć konflikt wcześniej i wcześniej zwolnić blokadę. Z drugiej strony jeżeli konflikt nie wystąpi a blokada zostaje utrzymana, dojdzie do ograniczenia współbieżności transakcji. Można też wyobrazić sobie sytuację, w której nie utrzymujemy blokady do momentu zatwierdzenia transakcji. Wtedy, ponieważ stan przejściowy utrzymuje się nie tylko przez czas zatwierdzania, ale przez cały czas wykonania transakcji (dokładnie: od momentu dostępu do zatwierdzenia), pozostałe współbieżne transakcje mogą mieć większy problem z odczytaniem spójnego obrazu pamięci współdzielonej. Warto dodać, że potencjalne rozwiązanie tego problemu – buforowanie modyfikacji nie jest możliwe z definicji mechanizmu pesymistycznego.

Optymistyczny mechanizm sterowania współbieżnością może odroczyć wykrycie lub rozwiązanie konfliktu np. do czasu zatwierdzenia transakcji. Wówczas, jeżeli konflikty nie wystąpiły, możliwy jest większy stopień współbieżności przy wykonywaniu wielu transakcji (stany niespójne występują tylko przez okres zatwierdzania zmian – wykonywania procedury `commit()`). Jednak jeżeli konflikty wystąpią, narażamy się na ryzyko wykonania każdej transakcji do końca, nawet jeżeli już na jej początku (przy pierwszym dostępie do pamięci współdzielonej) było wiadomo, że powinna zostać wycofana.

Lazy vs. eager conflict detection oraz **tentative vs. committed conflict detection**. Jak wspomniano wcześniej, jednym z warunków wystąpienia konfliktu jest to , by co najmniej jedna z operacji, które go powodują, była operacją zapisu. Oznacza to, że konflikt nie musi występować jedynie pomiędzy dwiema operacjami zapisu, a tym samym, że wyróżnienie wczesnego i późnego zarządzania wersjami nie zamyka listy podejść do wykrywania konfliktów. Nawet w przypadku dwóch konfliktowych zapisów i indirect updates możliwe jest podejście, w którym system pamięci transakcyjnej buforuje modyfikacje, jednak już w momencie ich zlecenia sprawdza strukturę kontrolną związaną z zapisywanym adresem by poszukać warunków wystąpienia konfliktu.

Późne wykrywanie konfliktów, charakterystyczne dla optymistycznego sterowania współbieżnością będzie oznaczało wykrywanie konfliktów w momencie zatwierdzania transakcji. W takim ujęciu, gdyby zasoby współdzielone nie były blokowane na czas zatwierdzania (ang. *lock-based STM approach*) w celu zaznaczenia że zmiana została wykonana, zapewnienie by transakcja widziała spójny obraz pamięci byłoby niezwykle trudne (nie byłoby jak sprawdzić takiej spójności).

Wczesne wykrywanie konfliktów zakłada ich poszukiwanie już w momencie, gdy transakcja deklaruje chęć użycia zasobu współdzielonego. Wyżej wskazano, że takie rozwiązanie ma sens dla buforowanych zapisów. Dla odczytów konieczność wykonania takiej operacji jest oczywista w sytuacji w której chcemy przedstawić transakcji spójny obraz zasobu współdzielonego.

Możliwe są też podejścia pośrednie, tj. niekiedy nawet wielokrotnie powtórzone procedury wykrywania konfliktów na różnych etapach życia transakcji (zarówno w trakcie jej wykonania, jak i zatwierdzania). Przykładem algorytmu pamięci transakcyjnej wykonującej wielokrotne poszukiwanie konfliktów jest *Transactional Locking II*, omówiony dalej w tym rapocie (punkt 2.3.2).

Pozostaje pytanie co zrobić z wykrytym konfliktem, a ściślej – kiedy zacząć brać taki konflikt pod uwagę. *Tentative conflict detection*, termin tu przetłumaczony jako *wstępna detekcja konfliktów* zakłada, że sam fakt wykrycia konfliktu (w trakcie wykonywania obu zaangażowanych weń transakcji, a więc przy wczesnym wykrywaniu konfliktów) wystarczy, by zapytać arbitra która transakcja „przegrywa". W podejściu z *późną detekcją konfliktów* (ang. *commited conflict detection*) z odpytaniem arbitra o decyzję czeka się do momentu gdy pierwsza z transakcji rozpocznie zatwierdzanie zmian. Mogłoby się bowiem okazać, że jedna transakcja „przegra" ze względu na decyzję arbitra, druga zaś wykona operację *non-forcible rollback*.

Zarządzanie zmianami lokalnymi. System pamięci transakcyjnej interesuje się w pierwszej kolejności modyfikacjami wykonywanymi na pamięci współdzielonej. Nie zmienia to faktu, że transakcja z perspektywy wątku nie różni się przesadnie od jakiegokolwiek innego wykony-

wanego przezeń kodu. Co za tym idzie, transakcja może modyfikować nie tylko stan pamięci współdzielonej, ale także i zmiennych lokalnych wątku (dla STM) lub globalnych względem węzła (dla D-STM). Z definicji systemu pamięci transakcyjnej wynika, że dzieje się to w sposób nietransakcyjny, a więc bez zapewniania niepodzielności.

W sytuacji, gdy w trakcie przetwarzania wystąpi konflikt uniemożliwiający zatwierdzenie jednej z transakcji, jej wykonanie może zostać powtórzone, nawet bez udziału programisty. O ile przed takim powtórzeniem dotychczasowe zmiany w pamięci współdzielonej są wycofywane, o tyle sposób postępowania ze zmianami lokalnymi może być różny.

Jeżeli programista definiuje sekcje atomowe, to należy się spodziewać, że system TM sam będzie zarządzał modyfikacjami lokalnymi. TM może też udostępniać mechanizm definiowania sekcji atomowych w taki sposób, by wprowadzanie zmian lokalnych o czasie życia dłuższym niż transakcja było niemożliwe (np. oznaczanie metod jako wykonywanych atomowo [53] lub definiowanie transakcji wewnątrz obiektu anonimowego [61]).

Jeżeli programista korzysta bezpośrednio z interfejsu programistycznego systemu TM, system taki może dostarczać specjalne operacje umożliwiające dostęp do danych lokalnych, przy użyciu których modyfikacje o których mowa mogą zostać wycofane razem z transakcją. System TM może też ignorować problem i zrzucać na programistę zadanie wycofania zmian lokalnych, dostarczając odpowiedni mechanizm (np. procedurę `commit()` informującą zwracaną wartością o rezultacie zatwierdzania) lub nie, co w efekcie zmuszałoby programistę do rezygnacji z wprowadzania lokalnych zmian z poziomu kodu transakcyjnego.

Single version STM vs. multiversion STM. Przykładem rozwiązania, które można sklasyfikować jako płynne przejście od STM do D-STM jest pamięć transakcyjna typu *multiversion*. Pamięć wielowersyjna może przechowywać kilka kopii t-obiektu pochodzących sprzed modyfikacji. W sytuacji, gdy przepustowość systemu transakcyjnego (ang. *throughput*) – liczba transakcji zatwierdzanych ogółem w jednostce czasu – jest preferowana względem aktualności danych, na których transakcja operuje, może się okazać, że przechowywane przez TM poprzednie wersje wystarczą do przedstawienia transakcji spójnego, choć niekoniecznie aktualnego stanu pamięci współdzielonej. Można w ten sposób uniknąć szeregu konfliktów i tym samym zwiększyć przepustowość systemu, nawet jeżeli oznacza to brak zachowania porządku czasu rzeczywistego w uszeregowaniu transakcji.

Oczywiście należy zapewnić ograniczenie głębokości takich „odczytów wstecz". Bez tego bowiem łatwo mogłoby dojść do sytuacji, w której każda transakcja zawsze operuje na stanie pamięci z momentu rozpoczęcia wykonania programu. Wszelkie modyfikacje trafiają wtedy w próżnię i pomimo braku zakleszczenia przetwarzanie nie postępuje – mamy do czynienia z *livelockiem*. W [42] znaleźć można stwierdzenie, że kryterium dla zapewnienia postępu przetwarzania jest to, by transakcje czytające dane z przeszłości same wprowadzały modyfikacje „w przeszłości", tj. tak, by inne wątki obserwowały wdrożenie tych modyfikacji przed pojawieniem się nowych wersji t-obiektów (co w praktyce oznacza, że i one będą działać „w przeszłości"). Szczególnym przypadkiem są tu długie transakcje odczytujące dane (*read-only*), które nie muszą wprowadzać żadnych modyfikacji, a działając współbieżnie z krótkimi transakcjami modyfikującymi stan pamięci mogłyby zostać zagłodzone.

Autor [53] powołuje się na wielowersyjny system TM w kontekście pamięci rozproszonych.

Możliwość zastosowania podobnego rozwiązania do wieloprocesorowych systemów TM nie ulega wątpliwości. Ze względu na różnice pomiędzy dwiema klasami docelowych systemów komputerowych, dyskusyjny może być jedynie uniwersalizm przesłanek za nim stojących. Jednak, jak stwierdzono wcześniej, podejście wprowadzające różnorodne rozwiązania w zależności od potrzeb (a więc w większym stopniu koncentrujące się na spełnieniu wymogów pewnego – potencjalnie wąskiego – spektrum aplikacji niż na własnym uniwersalizmie) jest przy pracach nad pamięcią transakcyjną powszechne.

Wielowersyjna pamięć transakcyjna może mieć szczególne znaczenie dla systemów rozproszonych ze względu na możliwą redukcję kosztów komunikacji, które są typową, niepomijalną cechą takich systemów. W niektórych sytuacjach będzie można wykonać transakcję z powodzeniem bez uprzedniego synchronizowania zmian t-obiektu (tj. bez sprowadzania nowszej wersji z innego węzła). Ceną za ten zysk jest narzut pamięciowy na każdym z węzłów.

Control-flow vs. data-flow Wyborem charakterystycznym dla rozproszonych pamięci transakcyjnych jest wybór pomiędzy podejściami *data-flow* i *control-flow*. W oparciu o analizę algorytmów pamięci transakcyjnych można by pokusić się o stwierdzenie, że różnica pomiędzy podejściami sprowadza się do elementu pamięci transakcyjnej, do którego dostęp jest synchronizowany. W podejściu data-flow byłby to sam t-obiekt, zaś w podejściu control-flow – fragment kodu, który stan tego t-obiektu modyfikuje.

Należy jednak pamiętać, że dla systemu rozproszonego kluczową cechą jest mobilność pewnych jego elementów. Saad w [53] definiuje pamięci typu data-flow jako takie, w których to t-obiekty są mobilne, transakcje zaś nie – wykonywane są one w całości na jednym węźle, który sprowadza do lokalnej pamięci potrzebne t-obiekty i, gdy zajdzie taka potrzeba, informuje system D-STM o wystąpieniu konfliktów.

W podejściu control-flow t-obiekty są powiązane z węzłami, transakcja zaś, choć inicjowana i zatwierdzana przez ten sam, pojedynczy węzeł, jest de facto wykonywana na kilku z nich z użyciem mechanizmów umożliwiających przenoszenie wykonania kodu między węzłami. [53] ściśle wiąże pamięci typu control-flow z mechanizmami zdalnego wywoływania procedur, RPC lub RMI, co znajdzie potwierdzenie w przypadku omawianego później algorytmu D-STM tej klasy [61].

Możliwe jest też podejście mieszane, *hybrid-flow*, przy czym [56] określa je jako możliwość definiowania przy pomocy tych samych konstrukcji językowych transakcji przeznaczonych do wykonania w trybie data-flow i control-flow, a następnie wybrania (jednorazowo, na etapie uruchamiania programu), który tryb faktycznie zastosować.

Argumentuje się, że podejście control-flow jest wyjątkowo użyteczne gdy zależności pomiędzy stanami obiektów są złożone i rozbudowane. Wtedy dobrą metodą ich modyfikacji jest kaskadowe zdalne wywoływanie procedur (por. benchmark *loan* opisany w punkcie 2.6.4.3). Bez wątpienia nie jest to jedyny sposób dokonywania zmian, tj. to co można wykonać przy użyciu systemów TM typu control-flow, można też wykonać przy użyciu nieco bardziej złożonych transakcji w systemach data-flow. Jednak ze względu na skalę problemów rozwiązywanych przez systemy rozproszone, wybór projektowy pomiędzy control-flow a data-flow może równać się możliwości lub nie wykonania pewnych zadań w praktyce.

Directory-based D-STM vs. repliacted D-STM Wielość podejść do zarządzania danymi w systemach rozproszonych będzie również integralną cechą systemów D-STM. Przytoczone wcześniej rozróżnienie pomiędzy pamięciami transakcyjnymi wielowersyjnymi i jednowersyjnymi ma za cel podkreślić płynny sposób przejścia od rozwiązań dla systemów wieloprocesorowych do rozwiązań dla systemów rozproszonych.

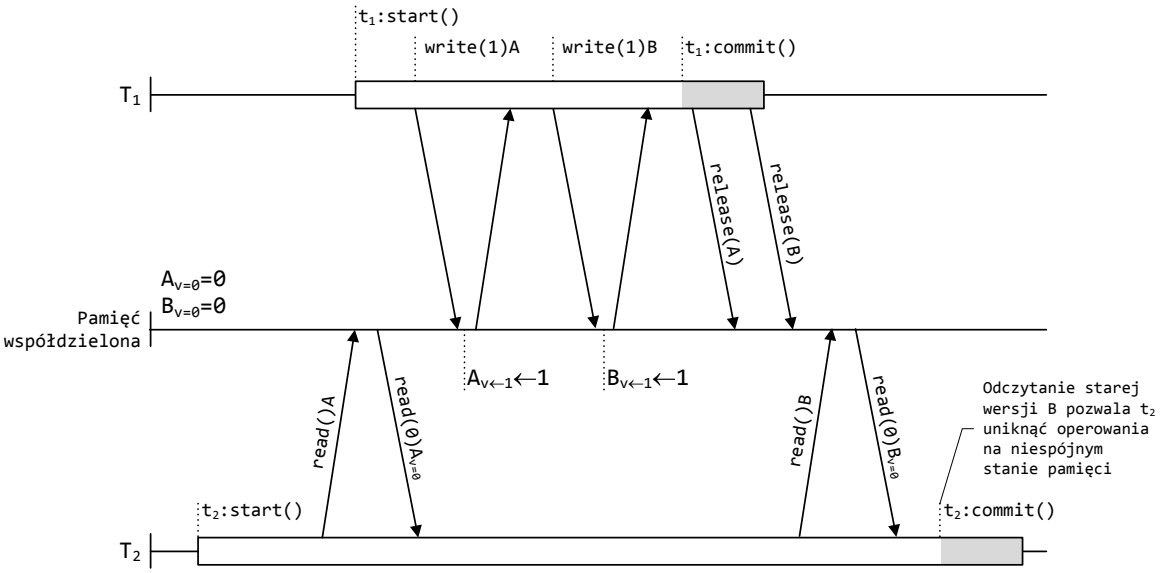
W obrębie rozproszonych pamięci typu data-flow [53] wyróżnia trzy podejścia dot. radzenia sobie z rozproszeniem danych: wykorzystanie (rozproszonych) katalogów obiektów i ich sprowadzenie na potrzeby wykonania transakcji (ang. *directory-based D-STM*), wykorzystanie mechanizmu unieważniania (ang. *invalidation*) do zapewniania spójności kopii i wreszcie replikację danych.

W pierwszym przypadku transakcja pozyskuje obiekty (ich lokalizacje) z centralnego (choć być może ukrywającego rozproszenie, replikację, awarie, itp.) katalogu. Po sprowadzeniu obiektów w czasie dostępu lub zatwierdzania, nowa lokalizacja jest umieszczana w rejestrze, a transakcja operuje wyłącznie na lokalnych kopiach t-obiektów. Jest oczywiste, że podejście takie ma sens (wydajnościowo, w systemach rozproszonych) przede wszystkim dla optymistycznego mechanizmu sterowania współbieżnością, ze względu na ograniczenie czasu, przez który transakcja posiada obiekt na wyłączność.

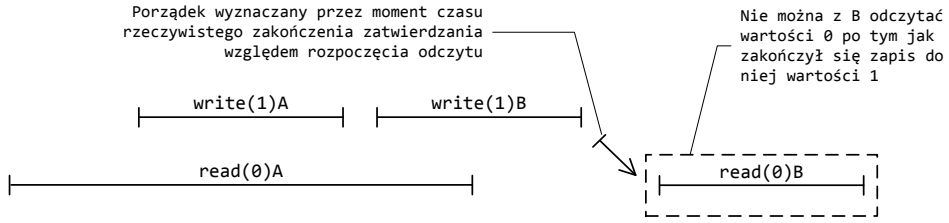
Unieważnianie – rozwiązanie, które znane jest z protokołów zapewniania spójności pamięci podręcznych w systemach wieloprocesorowych, sprowadza się do utrzymywania wielu kopii t-obiektu na różnych węzłach i oznaczania pozostałych w stosunku do tej, do której wprowadzamy zmianę jako nieaktualnych. Sposób sprowadzenia aktualnej kopii lub sprowadzenia i wdrożenia jedynie najnowszych zdalnych zmian zależy od implementacji systemu D-STM. Choć podejście takie intuicyjnie można skojarzyć z systemami wielowersyjnymi, Saad w [53] klasyfikuje je jako jedno z podejść do implementacji systemów typu single version.

Wreszcie replikacja – utrzymywanie wielu kopii t-obiektu na różnych węzłach i uspójnianie zmian np. w momencie ich powstania (inicjowane przez autora zmian) lub w momencie dostępu (inicjowane przez odczytującego zmiany) zyskuje na znaczeniu w kontekście rozproszonych pamięci transakcyjnych, zwłaszcza w połączeniu z mechanizmem dzierżaw (ang. *lease-based replication*) [14, 24]. Wykorzystanie przez TM replikacji jako mechanizmu dobrze znanego w świecie systemów rozproszonych prowadzi do powstania *replikowanych rozproszonych pamięci transakcyjnych* [14]. Ponieważ uspójnienie zmian może sprowadzać się no wykonania operacji modyfikacji na każdej z replik z osobna, replikowane D-STM będą mogły równie dobrze stosować podejście control-flow w miejsce data-flow.

W kwestii wielowersyjnych pamięci rozproszonych Saad wyróżnia takie, które zachowują własność linearyzacji [35] (tj. każda historia wykonania jest dla nich równoważna pewnej legalnej historii sekwencyjnej) i takie, które tej własności nie zachowują. Przykładem drugiego typu mógłby być system, w którym dla *in-place updates* transakcja $\mathtt{t}_2$ odczytuje wartość zmiennej `A`, transakcja `t` zapisuje nowe wartości zmiennych `A` i `B`, po czym zatwierdza zmiany a w końcu $\mathtt{t}_2$ próbuje odczytać zmienną `B`. Gdy system pamięci transakcyjnej zwróci transakcji $\mathtt{t}_2$ wartość `B` sprzed modyfikacji, kosztem nieodczytania zatwierdzonych już zmian, a więc naruszenia legalności sekwencyjnego odpowiednika historii wykonania, $\mathtt{t}_2$ będzie mogła uniknąć konfliktu ze względu na zmienną `B`.



(a) Przykład korzyści z wykorzystania multiversion STM.

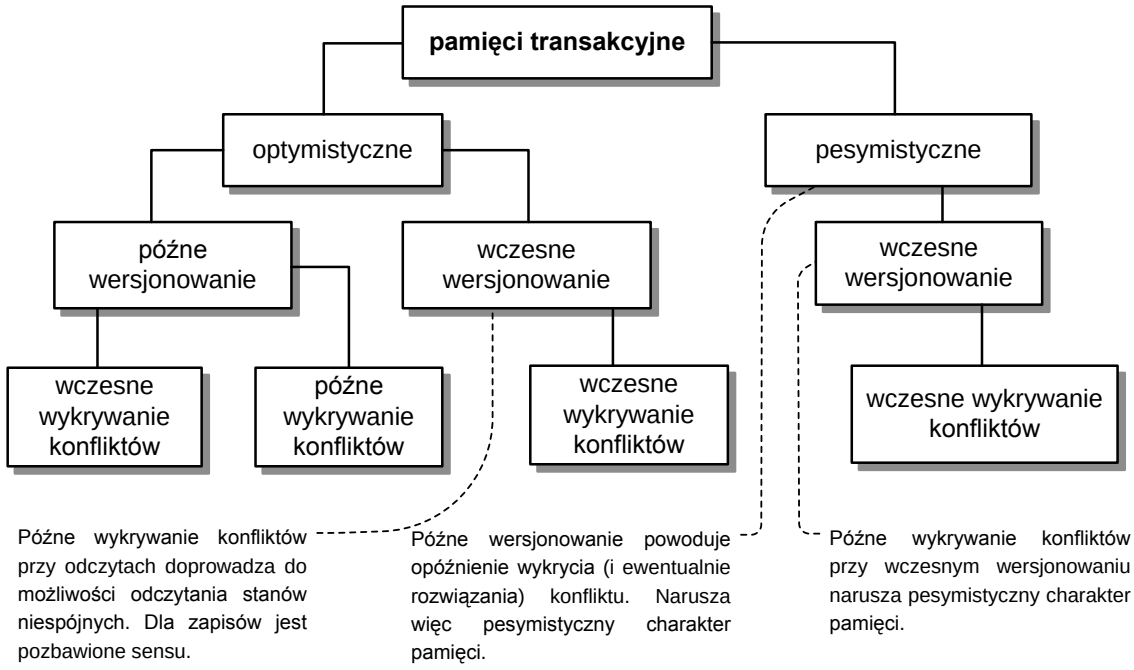


(b) Naruszenie legalności uszeregowania, jeżeli A i B mają zachowywać semantykę rejestru.

Rysunek 2.2: Wielowersyjna pamięć transakcyjna.

Należy też wspomnieć o rozróżnieniu w zakresie interfejsu programistycznego STM. System pamięci transakcyjnej może 1) pozwalać programiście na definiowanie sekcji atomowych lub jedynie na korzystanie z procedur transakcyjnych opisanych w punkcie 2.1, 2) wymagać użycia specjalnych procedur przeznaczonych do odczytu i zapisu pamięci współdzielonej lub automatycznie instrumentować kod objęty transakcją, 3) wymagać zdefiniowania read- i write-setu *a priori* (pamięci *statyczne*) lub budować je w momencie napotkania transakcyjnych poleceń `read()` i `write()` (pamięci *dynamiczne*), 4) ograniczać (pamięci *ograniczone*, ang. *bounded*, cecha charakterystyczna dla HTM) lub nie (pamięci *nieograniczone*, ang. *unbounded*) rozmiar read- i write-setów i liczbę operacji wewnątrz transakcji, 5) udostępniać lub nie specjalne procedury dostępu do danych lokalnych (por. wyżej), czy wreszcie 6) udostępniać lub nie mechanizm *non-forcible rollback*.

Powyższe rozróżnienia prowadzą do powstania taksonomii systemów pamięci transakcyjnej. Niniejszy raport rozróżnia systematykę pamięci dla systemów wieloprocesorowych (z której wiele określeń będzie też stosowalnych do D-STM) i systematykę rozproszonych pamięci transakcyjnych. Pierwszą, skonstruowaną ze względu na mechanizmy sterowania współbież-



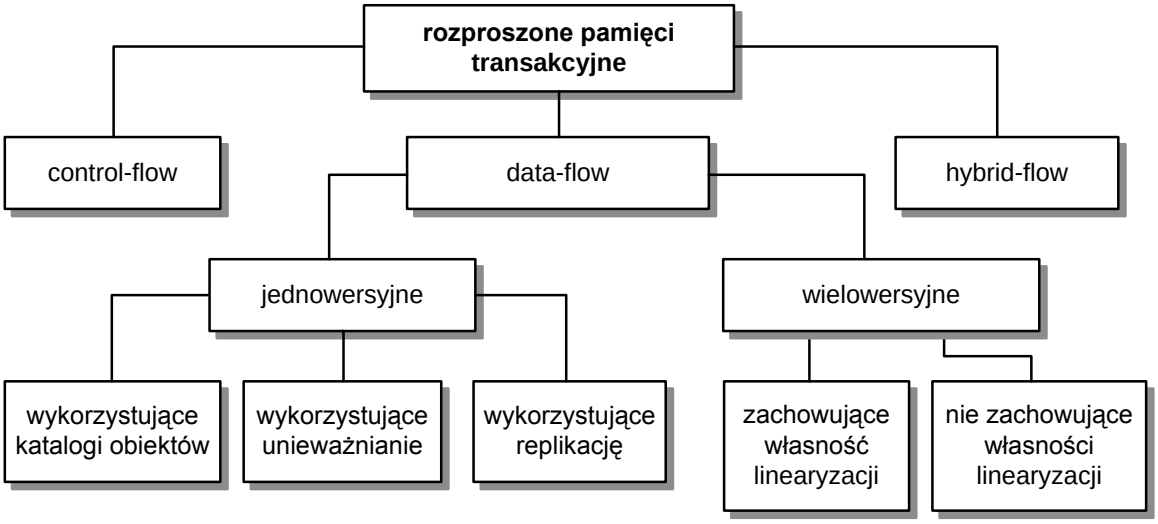
Rysunek 2.3: Taksonomia systemów STM ze względu na aspekty sterowania współbieżnością.

nością, przedstawia rysunek 2.3. Drugą, zapożyczoną od [53] przedstawia rysunek 2.4.

Celem niniejszym raporcie nie jest dokonywanie wyborów projektowych, czyli odnoszenie możliwości w poszczególnych kategoriach do określonych zastosowań i własności systemów komputerowych. Założeniem leżącym u podstaw pracy jest porównanie dwóch różnych systemów D-STM cechujących się skrajnie różnymi podejściami do sterowania współbieżnością. Dlatego też nie zostaną tu przedstawione różne architektury systemów rozproszonych. Jeżeli model systemu zostanie w dalszej części raportu wprowadzony, to jego obranie będzie wynikało raczej z podejścia *bottom-up* – z faktu, że analizowany system D-STM właśnie taki a nie inny model systemu rozproszonego przyjmuje. Ten rozdział, jak już wspomniano, ma za zadanie wpisać prezentowane dalej rozwiązania w szerszy kontekst, a jeżeli uda się wskazać słabe punkty analizowanych implementacji, być może rozważane tutaj koncepcje pomogą je wyeliminować.

2.3 Przykłady implementacji systemów pamięci transakcyjnej

By mówić o wadach poszczególnych rozwiązań projektowych systemów pamięci transakcyjnych omówionych w poprzednim punkcie, należy najpierw przeanalizować zastosowanie tych rozwiązań w praktyce. Niniejszy rozdział przedstawi trzy przykładowe implementacje pamięci transakcyjnych – jedną dla systemów wieloprocesorowych, kolejną, która bazując na algorytmie dla systemów wieloprocesorowych adaptuje go na potrzeby systemów rozproszonych i jedną typowo rozproszoną. Wszystkie są ważne albo ze względów historycznych, albo z racji wykorzystania ich w dalszej części raportu.



Rysunek 2.4: Taksonomia systemów D-STM. Za: [53].

2.3.1 WSTM Harrisa i Frasera

Implementacja pamięci transakcyjnej zaproponowana przez Harrisa i Frasera w [30] znajduje się w tym zestawieniu ze względów historycznych. Można doszukać się informacji o tym, że WSTM (w artykule wprowadzającym nie pada konkretna nazwa implementacji) jest jedną z pierwszych czysto programowych implementacji TM dla systemów wieloprocesorowych oraz że wyniki wydajnościowe osiągane przez tę implementację przy wykonywaniu operacji na specyficznych strukturach danych (tablicach haszowych) były wzorem dla przyszłych systemów STM. Z pewnością można stwierdzić, że Harris i Fraser zaproponowali pierwszy nieblokujący system STM redukujący przy tym liczbę warstw abstrakcji (co miałoby ułatwić integrowanie kodu transakcyjnego i nietransakcyjnego) [31, str. 133], a przyjęta przez nich za sposób wykorzystania transakcji koncepcja warunkowego regionu krytycznego stała się powszechnym sposobem udostępniania mechanizmów transakcyjnych programistom [69]. Dodatkowo wedle słów autorów, WSTM jako pierwsza umożliwia zaimplementowanie warunkowych regionów krytycznych w sposób, który nie bazuje na wzajemnym wykluczaniu [30, punkt 1].

WSTM jest programową pamięcią transakcyjną z optymistycznym zarządzaniem współbieżnością. Zaimplementowana została w języku Java. T-obiektem jest w niej zawsze słowo (jednostka, której modyfikacja jest zawsze atomowa, co wynika z gwarancji udzielanych przez środowisko uruchomieniowe). WSTM dostarcza transakcyjne procedury `read()` i `write()` oraz bazujące na nich konstrukcje regionów krytycznych z warunkiem w oparciu o modyfikacje kompilatorów source-to-bytecode i bytecode-to-native oraz środowiska uruchomieniowego. Polega także na niespecyficznym dla TM wsparciu sprzętowym, dostępnym w większości architektur sprzętowych.

Warunkowe regiony krytyczne, od których wywodzi się koncepcja sekcji (regionów) atomowych to konstrukcja, w której blok kodu przeznaczony do wprowadzania zmian w pamięci współdzielonej w sposób niepodzielny może być opatrzony dodatkowym warunkiem, którego spełnienie powoduje rozpoczęcie wykonania bloku, zaś niespełnienie – oczekiwanie na zmianę stanu systemu i ponowną jego ewaluację. Ponieważ głównym zadaniem WSTM jest

posłużenie za podstawę implementacji warunkowych regionów krytycznych, jej interfejs programistyczny różni się, choć nieznacznie, od typowego, przedstawionego w punkcie 2.1.

```
1  int buffer_get(int[] buffer, int write_pointer) {
2        int result;
3        atomic(write_pointer > 0) {
4             result = buffer[write_pointer--];
5        }
6        return result;
7  }
```

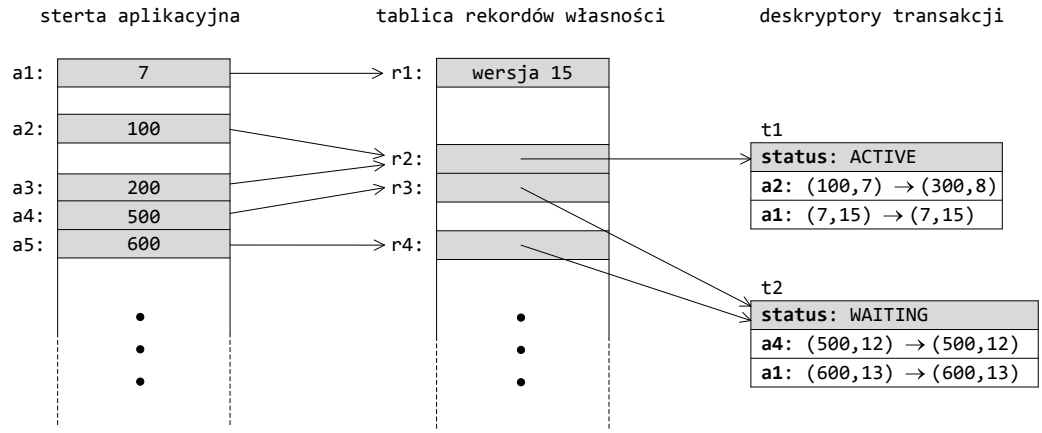
Rysunek 2.5: Przykład zastosowania warunkowego regionu krytycznego.

Podstawą implementacji jest szereg struktur kontrolnych przechowywanych w pamięci współdzielonej, jednak nie związanych bezpośrednio z obszarami mającymi pełnić rolę t-obiektów. Innymi słowy, choć dla t-obiektów pewne dodatkowe struktury będą zawsze potrzebne, pamięć współdzielona będzie mogła być wykorzystywana współbieżnie przez polecenia transakcyjne i nietransakcyjne (powiązanie t-obiektów ze strukturami kontrolnymi jest tworzone dynamicznie). Podejście takie ma na celu ułatwić łączenie nowego kodu, wykorzystującego transakcje ze starymi rozwiązaniami, których uaktualnić nie sposób, np. bibliotekami kodu oraz umożliwić późniejszą zmianę przeznaczenia obszarów pamięci do których dostęp ma początkowo odbywać się transakcyjnie.

Dostęp transakcyjny dotyczy sterty aplikacyjnej (ang. *heap*). Zgrupowanie adresów słów do postaci jednostek, w oparciu o które wykrywane są konflikty odbywa się z wykorzystaniem tablicy rekordów własności (ang. *ownership records, orecs*). Zatem pomimo przyjęcia za t-obiekt relatywnie małej jednostki, jaką jest słowo, możliwe będzie występowanie fałszywych konfliktów. Przyporządkowanie adresów rekordom własności może odbywać się w różny sposób. Rekord może odpowiadać słowu przy założeniu, że dopuszczalne są znaczne narzuty pamięciowe, może też odpowiadać obiektowi/strukturze. Autorzy podają, że w ich implementacji tablica rekordów ma ustaloną wielkość, a przyporządkowanie do nich adresów w pamięci realizowane jest przez funkcję mieszającą. Rekordy przechowują numer wersji, inkrementowany za każdym razem, gdy którakolwiek z lokalizacji spod powiązanych adresów jest modyfikowana, lub wskaźnik na deskryptor transakcji, która z niego aktualnie korzysta.

Deskryptor transakcji (ang. *transaction descriptor, td*) zawiera informację o jej stanie (zatwierdzona: `COMMITTED`, w trakcie wykonania: `ACTIVE`, wycofana: `ABORTED` lub oczekująca: `WAITING` – status wprowadzony z myślą o wykorzystaniu TM do implementacji regionów krytycznych z warunkiem) oraz listę wprowadzanych przez nią modyfikacji w postaci listy par dwójek *(wartość, numer wersji)* (po jednej parze na każdy modyfikowany adres), na której pierwsza dwójka w parze reprezentuje wartość przed wprowadzeniem przez transakcję pierwszej modyfikacji, a druga dwójka – wynik najnowszej modyfikacji. Szczególną formą modyfikacji będzie odczyt, który nie zmienia ani numeru wersji, ani wartości, jednak musi zostać uwzględniony w deskryptorze na potrzeby weryfikacji spójności odczytanego obrazu.

W takim układzie [30] wprowadza pojęcie logicznego stanu adresu (ang. *logical state*, LS) na stercie. Algorytm ustalania LS pozwala powiązać adres w pamięci współdzielonej z przechowywaną pod nim wartością oraz numerem wersji przy uwzględnieniu redo-logów (którymi w praktyce są deskryptory transakcji). Rozróżnienie stanów logicznych pomoże wskazać



Rysunek 2.6: Dodatkowe struktury kontrolne WSTM. Za: [30].

wartość, którą spod adresu zwraca procedura transakcyjnego odczytu `read()`.

Stan logiczny LS1. Jeżeli rekord własności powiązany z adresem posiada wewnątrz numer wersji, to wartość przechowywana pod tym adresem bierze się bezpośrednio ze sterty, zaś numer wersji z rekordu. Przykładem adresu w stanie LS1 na rysunku 2.6 jet `a1`.

Stan logiczny LS2. Jeżeli rekord własności dla podanego adresu zawiera wskaźnik na deskryptor transakcji, a sam deskryptor – wpis dotyczący analizowanego adresu, to wartość i numer wersji należy odczytać z pierwszej dwójki wpisu – wartości i numeru wersji sprzed modyfikacji, jeżeli tylko deskryptor stwierdza, że transakcja jest w stanie `ACTIVE` i z drugiej dwójki – jeżeli deskryptor stwierdza, że transakcja jest w stanie `COMMITTED`. Przykładem adresu znajdującego się w stanie LS2 jest na rysunku 2.6 adres `a2`, z którym związana jest wartość 100 i numer wersji 7.

Stan logiczny LS3. Jeżeli rekord własności dla podanego adresu zawiera wskaźnik na deskryptor transakcji, ale sam deskryptor nie zawiera informacji o adresie, wtedy w deskryptorze szuka się informacji o innym adresie, do którego przyporządkowany jest ten sam rekord. Ponieważ to transakcja wiąże swój deskryptor z rekordem w oparciu o określone kryteria (por. procedura `commit()`), adres taki zawsze będzie istniał. Gdy wpis w deskryptorze zostanie odnaleziony, numer wersji adresu, dla którego ustalany jest stan logiczny odczytuje się podobnie jak w przypadku LS2 (numer sprzed modyfikacji dla transakcji niezatwierdzonej i numer po modyfikacji dla transakcji zatwierdzonej). Wartość natomiast odczytywana jest bezpośrednio ze sterty aplikacyjnej.

Przedstawione na rysunku 2.6 struktury kontrolne pozwalają na zaimplementowanie następujących operacji interfejsu transakcyjnego:

- `start()` – rozpoczyna nową transakcję. Operacja ta sprowadza się do zaalokowania nowego deskryptora transakcji i ustawienia w nim jej stanu na `ACTIVE`. Z początku deskryptor nie jest wskazywany przez żaden rekord własności.

- `abort()` – powoduje oznaczenie odpowiedniej transakcji jako znajdującej się w stanie `ABORTED`. Zmiana jest zapisywana w odpowiednim deskryptorze.
- `read()` – ta procedura musi rozważyć dwa przypadki: 1) jeżeli transakcja już wcześniej zapisywała wartość pod odczytywanym adresem, to by zapewnić gwarancję *read-your-writes* należy odnaleźć odpowiedni wpis w jej deskryptorze i zwrócić wartość po ostatniej modyfikacji, 2) jeżeli zaś deskryptor transakcji nie zawiera informacji o tej lokalizacji, należy wpierw ustalić logiczny stan adresu i użyć go do uzupełnienia deskryptora. Za starą i nową wartość przyjmuje się tę pochodzącą ze stanu logicznego, natomiast w przypadku numeru wersji użycie tego ze stanu logicznego to tylko jeden wariant. Numer wersji ostatecznie trafi do rekordu własności, zatem w pewnym sensie jest on wspólny dla wszystkich powiązanych z nim adresów. Stąd jeżeli tylko deskryptor transakcji zawiera już informację o innym adresie powiązanym z tym samym rekordem co odczytywany, należy z tego wpisu odczytać nowy numer wersji (dla większej liczby powiązanych adresów będzie to maksimum po nowych numerach wersji). Stary numer wersji pochodzi zawsze ze stanu logicznego, tak samo jak nowy numer wersji w przypadku niespełnienia powyższego warunku (oczywiście w tym przypadku jest on równy staremu).
- `write()` – zapis sprowadza się do zapewnienia, że deskryptor transakcji zawiera wpis dotyczący zapisywanego adresu (można to zapewnić np. przez wcześniejsze odczytanie lokalizacji) oraz uwzględnienia zmiany w tym wpisie. Nowa wartość przechowywana pod adresem odpowiada wartości po modyfikacji, natomiast numer wersji po modyfikacji to stary numer wersji zwiększony o 1. Ponieważ, jak wspomniano wcześniej, numery wersji związane są raczej z rekordami własności niż z poszczególnymi adresami, należy też zapewnić, by zwiększenie numeru wersji o 1 dotyczyło wszystkich wpisów w deskryptorze transakcji, które dotyczą adresów z tego samego rekordu co adres zapisywany.
- `commit()` – zatwierdzenie sprowadza się do pozyskania na wyłączność wszystkich rekordów własności, które w oparciu o analizę deskryptora transakcji są potrzebne do wprowadzenia zmian z redo-logu na stertę, faktycznego wdrożenia tych zmian i wreszcie zwolnienia rekordów. Procedura pozyskania rekordu weryfikuje także spójność obrazu pamięci współdzielonej odczytanego podczas wykonywania transakcji.
- `wait()` – ta operacja jest niestandardowym elementem interfejsu programistycznego WSTM. Jej semantykę można określić następująco: „wycofaj transakcję (wykonaj *non-forcible rollback*) i poczekaj na zmianę wartości w co najmniej jednej z odczytywanych lokalizacji". Paradoksalnie działa ona podobnie do operacji `commit()`, pozyskując wszystkie potrzebne rekordy własności. Jeżeli ta operacja się uda i okaże się, że odczytano stan spójny, transakcja, zamiast stosować zmiany do sterty, pozostawia wskaźniki na swój deskryptor (od teraz w stanie `WAITING`) w rekordach, po czym usypia wykonujący ją wątek. Teraz każdy inny wątek, który spróbuje wykorzystać jeden z tych rekordów podczas zatwierdzania otrzyma informację o tym, że w systemie istnieje transakcja, która czeka na zmianę jakiejkolwiek lokalizacji związanej z rekordem. Procedura `wait()` jest wykorzystywana przy implementacji warunkowych regionów krytycznych do ponowienia próby wykonania bloku kodu (po uprzednim odczekaniu na odpowied-

nie modyfikacje) w sytuacji, gdy warunek wejściowy nie jest spełniony.

Jak wspomniano, istotą algorytmu jest sposób pozyskiwania rekordów własności na potrzeby wprowadzenia zmian do sterty. Procedurę instalacji w rekordzie wskaźnika na deskryptor transakcji przedstawia algorytm 2.7. Algorytm ten wykonywany jest dla każdego wpisu (związanego z odczytywanym lub modyfikowanym adresem) w deskryptorze transakcji (indeks takiego wpisu oznaczono na liście parametrów formalnych przez `i`). Poza wydobyciem zawartości wpisu (`te`), pierwszym krokiem jest atomowa podmiana zawartości rekordu własności dla adresu z wpisu (`orec_of(te.addr)`) na wskaźnik na deskryptor transakcji (`td`), ale tylko pod warunkiem, że wcześniej rekord przechowywał numer wersji i to dokładnie taki jak wersja odczytana przez transakcję tuż przed wykonaniem modyfikacji (lub jej szczególnego wariantu – odczytu).

```
1  acquire_result acquire(transaction_descriptor td, int i) {
2       transaction_entry te = td.entries[i];
3       ownership_record seen;
4       seen = CAS(ownership_record_of(te.addr), te.old_version, td);
5       if (seen == te.old_version || seen == td) {
6           return TRUE;
7       } else if (contains_version_number(seen)) {
8           return FALSE;
9       } else {
10          return BUSY;
11      }
12 }
```

Rysunek 2.7: Procedura pozyskiwania rekordu własności. Za: [30].

Semantyka atomowej operacji *compare-and-swap* przedstawia się następująco:

```
old_value = CAS(addr, value_to_compare_to, value_to_replace_with);
```

Jeżeli wartość spod podanego adresu równa się wartości do porównania to należy umieścić pod tym adresem nową wartość. Bez względu na to czy podmiana powiodła się czy nie, `CAS` zwraca zawartość lokalizacji sprzed próby wykonania operacji, tj. starą wartość spod wskazanego adresu, jeżeli ta została zmieniona i aktualną-niezmienioną, jeżeli nie.

W następnym kroku procedury `acquire()` badany jest wynik podmiany. Jeżeli pierwsza część warunku jest spełniona, oznacza to że był spełniony warunek podmiany, zatem została ona wykonana. Jeżeli spełniona jest druga część warunku, podmiana została wykonana wcześniej, przy okazji przetwarzania innego adresu z deskryptora transakcji.

Podmiana może się nie udać. Dzieje się tak w przypadku gdy 1) rekord własności przechowuje numer wersji, ale taki który nie odpowiada informacji przechowywanej we wpisie w deskryptorze lub 2) rekord własności nie przechowuje numeru wersji lecz wskaźnik na deskryptor transakcji (inny niż aktualnie analizowany). W pierwszym przypadku oznacza to, że od momentu pierwszego dostępu transakcji do adresu (tego aktualnie przetwarzanego) w pamięci współdzielonej (odczytu lub zapisu) inna transakcja go zmodyfikowała[3]. Są to przesłanki wystarczające do wykrycia konfliktu i zwrócenia wartości `FALSE`. W drugim – rekord

[3] alternatywnie mogła zmodyfikować inny adres powiązany z tym samym rekordem własności, doprowadzając do powstania fałszywego konfliktu.

znajduje się aktualnie w użyciu, należy więc przeanalizować deskryptor transakcji przez niego wskazywany, by stwierdzić czy druga transakcja, blokująca dostęp jest uśpiona (należy wtedy zatwierdzić własne zmiany i ją obudzić), czy też zatwierdza zmiany (autorzy nie podają sposobu postępowania w takim przypadku, jednak jasne jest, że należy albo poczekać – o ile tylko podajemy indeksy `i` adresów do funkcji `acquire()` w taki sposób, że rekordy własności są zawsze pozyskiwane w określonym porządku by uniknąć zakleszczenia – albo wykonać procedurę zatwierdzania od nowa uprzednio zwalniając pozyskane rekordy (to podejście wydaje się słuszne przy próbie zachowania własności *non-blocking* implementacji TM).

Pozyskanie rekordu jest tożsame ze zmianą sposobu odczytywania wartości spod adresów z nim związanych. Odkąd rekord został pozyskany, wartości te pochodzą z redo-logu zamiast bezpośrednio ze sterty. Zmiana stanu transakcji z `ACTIVE` na `COMMITTED` – zmiana jednego słowa w pamięci, z definicji atomowa, powoduje rozpoczęcie odczytywania wartości z części logu „po modyfikacji". Stąd po skutecznym pozyskaniu wszystkich rekordów transakcja zmienia swój stan na `COMMITTED`, stosuje zmiany z redo-logu do sterty i zwalnia rekordy z wykorzystaniem procedury przedstawionej na rysunku 2.8.

```
1  void release(transaction_descriptor td, int i) {
2      transaction_entry te = td.entries[i];
3      if (td.status == COMMITTED) {
4          CAS(ownership_record_of(te.addr), td, te.new_version);
5      } else {
6          CAS(ownership_record_of(te.addr), td, te.old_version);
7      }
8  }
```

Rysunek 2.8: Procedura zwalniania rekordu własności. Za: [30].

Sposób sprawdzania spójności obrazu odczytywanego przez transakcję (sprawdzanie per rekord własności) może budzić podejrzenia, że jeżeli dla transakcji t_2 odczyt adresu związanego z jednym rekordem nastąpi przed zatwierdzeniem transakcji t_1, a dla innego, związanego z innym rekordem – po, to taka niespójność może pozostać niewykryta. Należy jednak pamiętać, że procedura zapisu powoduje, że numer wersji każdego rekordu powiązanego z zapisywanymi adresami będzie zwiększony o 1. Ponieważ nowe numery wersji są wprowadzane razem ze zmianami transakcyjnymi – niepodzielnie, nie istnieje możliwość, w której taki konflikt nie zostanie wykryty, bowiem zawsze co najmniej jeden rekord nie przejdzie weryfikacji spójności.

Implementacja WSTM ma swoje braki. Najważniejszym z nich jest możliwość operowania kodu na odczytanym stanie niespójnym (ze względu na późne wykrywanie konfliktów), kosztowne wyszukiwanie powiązań pomiędzy adresami, rekordami własności i deskryptorami transakcji oraz brak konkretnych rozwiązań w zakresie reguły arbitrażu. Pomimo tego Harris i Fraser wykazali przydatność ich implementacji podając wyniki pomiarów przedstawione w tym raporcie na rysunku 2.9 (dla pojedynczych operacji na tablicy haszowej, z których 16% to zapisy oraz dla złożonych operacji na tablicy haszowej zawierającej 4096 elementy). W obu przypadkach przyciąga uwagę potężny skok wydajności warunkowych regionów krytycznych (*conditional critical regions*) nawet względem zamków drobnoziarnistych (*fine-grained locks*) i przy stosunkowo dużej liczbie zapisów co zaskakuje w kontekście optymistycznego charak-

teru implementacji.

2.3.2 *Transactional Locking II*

Algorytm STM *Transactional Locking II* (TL2) autorstwa Dice'a, Shaleva i Shavita [21] jest kolejnym, po WSTM, algorytmem z optymistycznym sterowaniem współbieżnością przedstawianym w tym raporcie. Do jego zalet należy przede wszystkim prostota rozwiązania. Dice i współautorzy podają też następujące cechy algorytmu, kluczowe w momencie jego publikacji [21, str. 5]:

- TL2 przy użyciu wielokrotnej weryfikacji spójności stanu postrzeganego przez transakcję zapobiega występowaniu opisanych w punkcie 2.4.1 anomalii związanych z postrzeganiem niespójności. Tym samym eliminuje konieczność modyfikacji środowiska uruchomieniowego na potrzeby wychwytywania błędnych odwołań do pamięci z kodu transakcyjnego i kosztowne (zazwyczaj bazujące na ograniczeniach czasowych, *timeoutach* lub weryfikacji spójności całego read-setu w trakcie wykonania transakcji [21, str. 4]) metody wykrywania *transakcji zombie* – tych, które utknęły w pętlach nieskończonych.
- Z racji luźnego powiązania lokalizacji w pamięci współdzielonej, występujących w roli t-obiektów i ich struktur kontrolnych, TL2 pozwala na zmianę przeznaczenia wybranych obszarów tej pamięci. W efekcie można płynnie zarządzać rozmiarem pamięci przeznaczonej do wykorzystania transakcyjnego nawet w trakcie działania programu.
- Nawet biorąc pod uwagę punkt 1), TL2 pozwala procesom na uzyskanie wydajności porównywalnej nie tylko z uboższymi w gwarancje systemami STM, ale także z rozwiązaniami bazującymi na zamkach drobnoziarnistych.

Ponadto [21] wprowadza kluczową z punktu widzenia tego raportu optymalizację algorytmu, mającą na celu zredukowanie roli struktur danych odpowiedzialnych za synchronizację pomiędzy transakcjami w generowaniu wąskich gardeł przetwarzania (ang. *bottlenecks*). Optymalizacja ta sprowadza się do ograniczenia liczby zapisów do pojedynczej, współdzielonej przez wszystkie wątki struktury danych. Autorzy argumentują jej potrzebę kosztami zapewnienia spójności pamięci podręcznych w systemach wieloprocesorowych (ccNUMA). Optymalizacja ta może mieć też znaczenie przy wykorzystaniu algorytmu w środowisku rozproszonym ze względu na zapewnienie skalowalności systemu.

TL2 jest podstawą dla algorytmu TFA, *Transaction Forwarding Algorithm*[4], będącego kluczowym elementem systemu rozproszonej pamięci transakcyjnej *HyFlow*, na wykorzystaniu którego bazuje dalsza część raportu. TFA również zostanie w tym punkcie przedstawiony.

Pamięć transakcyjną stosuje się m.in. by zwiększyć stopień współbieżności wykonywania bloków atomowych. Zatem niejako wbrew intuicji, TL2 próbuje osiągnąć zamierzony cel

[4] pomimo, że Saad, wprowadzający TFA w [53] nie określa wprost jego związku z TL2, podobieństwa są uderzające. Dodatkowo nazewnictwo pewnych elementów programowych we wprowadzanej przez Saada pamięci transakcyjnej *HyFlow* może już wprost taki związek sugerować (moduł rozproszonej pamięci transakcyjnej typu dataflow został w *HyFlow* nazwany `dtl2`, co miałoby być skrótem od „*Distributed Transactional Locking 2*". Wreszcie [54] dokładnie stwierdza, że implementacja algorytmu DTL2 jest jednym z kluczowych elementów *HyFlow*.

wprowadzając centralny, globalny licznik wersji (zwany dalej gv). Licznik ten jest inkrementowany za każdym razem gdy transakcja wprowadza zmiany do pamięci operacyjnej. Ponieważ TL2 jest algorytmem wykorzystującym późne zarządzanie wersjami i transakcje są w nim wykonywane w sposób spekulacyjny, budując w ten sposób redo-log, zawarte w nich modyfikacje wprowadza się masowo, podczas zatwierdzania (ang. *bulk update*). Co za tym idzie, w praktyce gdy licznik gv został zwiększony o 1, oznacza to, że jakaś transakcja zatwierdziła całość wprowadzonych przez siebie zmian.

Drugim elementem jest zestaw wersjonowanych zamków do synchronizacji zapisów (ang. *versioned write-locks*), po jednym dla każdego t-obiektu. Jednym z przejawów prostoty implementacji TL2 jest wykorzystanie do ich implementacji pojedynczych słów w pamięci współdzielonej i podstawowego wsparcia sprzętowego – instrukcji *compare-and-swap*, CAS, przedstawionej już w punkcie 2.3.1. Pojedynczy bit w słowie przeznaczany jest na reprezentację stanu zamka, pozostałe zaś przechowują numer wersji.

Mając do dyspozycji te dwie struktury kontrolne, algorytm można wyrazić następująco:

- `start()` – powoduje zaalokowanie niezbędnych zasobów (miejsca na redo-log i read-set) i skopiowanie aktualnej wartości gv do zmiennej lokalnej wątku: `rv` (*read-version*). Wartość ta posłuży później do sprawdzenia, czy transakcja odczytała spójny stan pamięci współdzielonej.
- `read()` – w trakcie spekulatywnego wykonania transakcji (a więc takiego, które nie wprowadza zmian do pamięci współdzielonej) odczyty, poza pobraniem wartości i numeru wersji (razem z bitem-zamkiem) z pamięci operacyjnej polegają na wciągnięciu odczytywanego adresu do read-setu. W oparciu o odczytany numer wersji odbywa się sprawdzenie, czy odczyt nie narusza spójności postrzeganego obrazu. Jeżeli numer wersji odczytywanej lokalizacji jest większy niż wartość `rv`, oznacza to, że inna transakcja zatwierdziła zmiany pomiędzy rozpoczęciem wykonania obecnej, a zrealizowaniem odczytu. Jeżeli z kolei zamek związany z odczytywaną lokalizacją jest w chwili odczytu zamknięty, oznacza to, że inna transakcja właśnie zatwierdza zmiany. Wystąpienie dowolnego z tych warunków zmusza transakcję do wycofania się i ponownego wykonania kodu. Oczywiście istnieje też problem zagwarantowania własności *read-your-writes*. Podczas odczytu TL2 poza sprawdzeniem stanu wersjonowanego zamka bada także redo-log pod kątem występowania tam odczytywanego adresu. W tym celu wykorzystuje filtr Blooma [8]. Jeżeli okaże się, że transakcja wcześniej zmodyfikowała odczytywaną lokalizację, wynikiem odczytu jest wartość po modyfikacji brana z redo-logu.
- `write()` – operacja zapisu sprowadza się do przechwycenia przez TM zapisywanego adresu i wartości oraz do umieszczenia złożonej z nich pary w redo-logu.
- `commit()` – zatwierdzenie transakcji rozpoczyna się od zamknięcia zamków dla wszystkich adresów z redo-logu. Proces ten musi się odbywać w oparciu o określoną kolejność zamykania, tak by uniknąć zakleszczenia. Jeżeli któregoś z zamków nie uda się zamknąć, jest to przesłanką do wycofania transakcji; transakcja nie czeka na zwolnienie zamka by zapewnić własność *obstruction-freedom* implementacji. Kolejnym krokiem jest atomowa inkrementacja globalnego licznika wersji, gv. Wykorzystać do tego należy

operację CAS. Zwróconą przez nią wartość (starą wartość gv) wątek zapamiętuje w lokalnej zmiennej wv (*write-version*). Zmienna ta jest inkrementowana po skutecznym zapisie do gv tak, by odzwierciedlała wartość globalnego licznika wersji z chwili zatwierdzenia zmian. Po skutecznym zamknięciu wszystkich zamków, należy zweryfikować cały read-set jeszcze raz w sposób taki jak przy odczycie (dla każdego elementu z read-setu zamek musi być otwarty, a numer wersji co najwyżej równy wartości rv). Jeżeli dla któregokolwiek wersjonowanego zamka numer wersji jest większy niż wartość rv, oznacza to, że inna transakcja zatwierdziła swoje zmiany pomiędzy momentem rozpoczęcia przez aktualną wykonania procedury commit() a chwilą obecną. W specjalnym przypadku, w którym rv + 1 = wv żadna transakcja nie zatwierdziła zmian pomiędzy rozpoczęciem przez obecną działania (odczytaniem gv do zmiennej rv), a zakończeniem zamykania zamków (i dalej odczytem przez CAS gv do zmiennej wv). Wtedy ponowne sprawdzenie read-setu nie jest konieczne. Ostatni krok sprowadza się do przeniesienia zmian z redo-logu do pamięci współdzielonej i zwolnienia zamków. Oczywiście każdemu takiemu zapisowi towarzyszy uaktualnienie wersji t-obiektu do wartości wv.

Należy wspomnieć, że mając na uwadze sposób implementacji operacji read(), dostarczenie mechanizmu wykonywania transakcji, które jedynie odczytują pamięć współdzieloną jest trywialnie proste, a koszt wykonania takich transakcji niski.

Centralna struktura danych, globalny licznik wersji gv, zapisywana jeden raz przy okazji każdego zatwierdzenia transakcji, jest elementem mogącym powodować powstanie wąskiego gardła przetwarzania. By uniknąć tej sytuacji autorzy TL2 wprowadzają optymalizację, która pozwala zredukować liczbę zapisów w najlepszym przypadku tyle razy, ile jest w systemie współbieżnych wątków. Idea rozwiązania polega na połączeniu każdego numeru wersji (zarówno globalnego, jak i tych przechowywanych w zamkach wersjonowanych) z identyfikatorem wątku, który ten numer zmienił jako ostatni. Implementacyjnie rozwiązanie to może się sprowadzać do dalszego podziału słowa przechowującego zamek wersjonowany.

Idea rozwiązania sprowadza się do spostrzeżenia, że skoro wątek może zaznaczyć przy każdym z t-obiektów fakt, że go zmodyfikował, to być może nie będzie musiał nawet zwiększać globalnego licznika wersji. Po uwzględnieniu tego faktu nowe procedury odczytu i zatwierdzania transakcji wyglądają następująco:

- commit() – podczas inkrementowania globalnego licznika wersji wątek sprawdza, czy jego wartość (zwrócona przez CAS a zapamiętywana w wv) zmieniła się w stosunku do tej, którą zapisał przy okazji zatwierdzania swojej poprzedniej transakcji. Schemat uaktualniania gv zakłada odczytanie wartości, następnie zinkrementowanie jej lokalnie i zapisanie przy pomocy operacji CAS po to, by zapewnić, że nikt w międzyczasie wartości gv nie zmienił. Tu pojawia się pole do kolejnej optymalizacji, bowiem jeżeli pierwsza operacja CAS się nie powiedzie, nie trzeba jej ponawiać (wątek ma pewność, że ktoś inny zmodyfikował gv w międzyczasie). Jeżeli więc wartość gv się zmieniła, wątek rezygnuje z uaktualniania (CAS) i zamiast tego przypisuje odczytaną wartość i swój identyfikator zamkom wersjonowanym wszystkich obiektów z redo-logu transakcji. W przeciwnym wypadku poza uaktualnieniem zamków wersjonowanych, wątek – jak wcześniej –

zapisuje zinkrementowany numer wersji i swój identyfikator do gv.

- `read()` – zmianie ulega procedura weryfikacji spójności przed transakcyjnym odczytem lokalizacji w pamięci współdzielonej. Jeżeli w wyniku odczytu zamka wersjonowanego odczytywanego t-obiektu uzyska się numer wersji większy niż `rv` lub też równy `rv` i identyfikator wątku inny niż niż ten odczytany przy okazji odczytu gv do `rv`, to transakcję należy wycofać.

Autorzy określają klasę historii wykonania, dla których powyższa optymalizacja może prowadzić do zwiększonej liczby nieuzasadnionych wycofań transakcji. Jeżeli wszystkie wątki poza jednym zatwierdzą swoje transakcje w ten sposób, że w gv zmianie ulegnie tylko identyfikator wątku, po czym pewna transakcja odczyta gv i będzie chciała odczytać dowolną zmienioną wcześniej przez pozostałe lokalizację, zostanie wycofana mimo iż zmiany tej lokalizacji zostały zatwierdzone przed uruchomieniem transakcji czytającej – nie mogą być więc źródłem konfliktu. Dla porównania w przypadku starego algorytmu transakcja czytająca stwierdzi, że numer wersji zamka wersjonowanego jest równy lokalnej kopii gv, tj. `rv`, co pozwoli na spełnienie warunku poprawności odczytu: `vwl.version` $\leq$ `rv`. Wykonane przez autorów badania wydajnościowe implementacji zdają się jednak marginalizować znaczenie tego zjawiska.

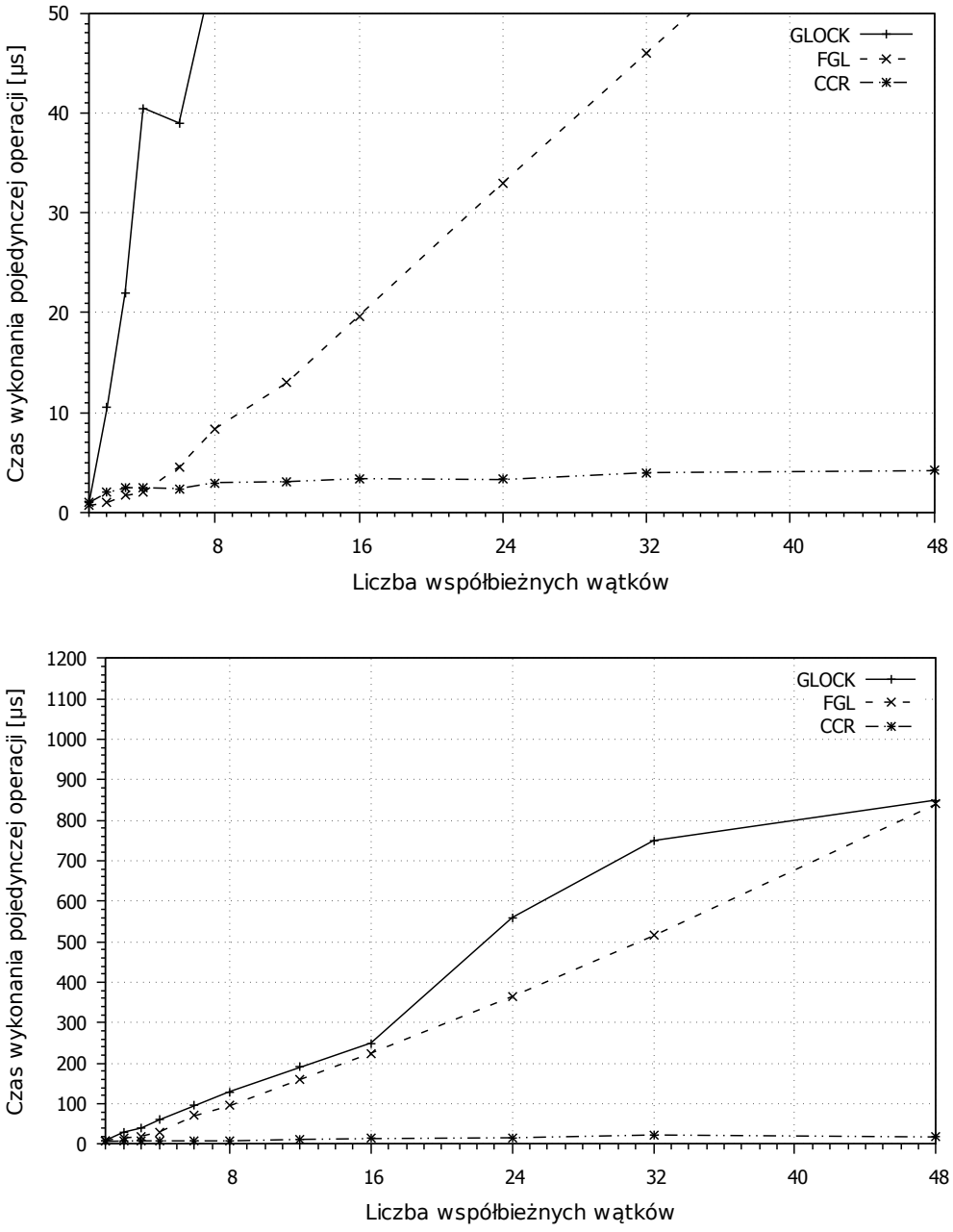
Podsumowując, należy sklasyfikować algorytm TL2 w kwestii różnych wyborów projektowych w następujący sposób: TL2 jest algorytmem optymistycznym (wykrycie i rozwiązanie konfliktu może nastąpić w momencie zatwierdzania transakcji, a wtedy będzie opóźnione względem wystąpienia) z późnym zarządzaniem wersjami (z racji buforowania zapisów) i mieszanym sposobem wykrywania konfliktów (wczesne i późne wykrywanie konfliktów odczyt-zapis i jedynie późne konfliktów zapis-zapis).

2.3.3 *Transaction Forwarding Algorithm*

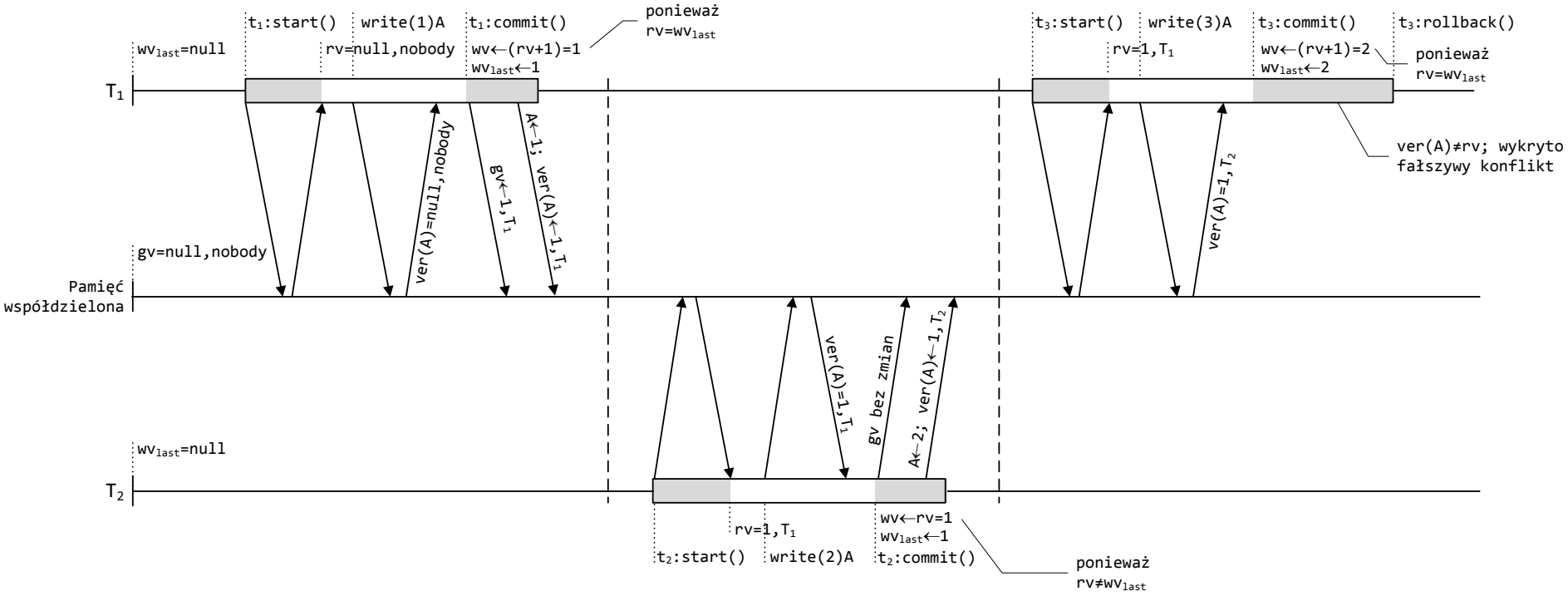
Transaction Forwarding Algorithm to rozwiązanie bazujące na algorytmie TL2, przeznaczone już wprost dla systemów rozproszonych. Elementami modelu systemu, które są w sposób szczególny adresowane przez to rozwiązane jest asynchronizm przetwarzania (asynchronizm lokalnych składowych procesu rozproszonego objawiający się niemożliwością wprowadzenia zegara globalnego) i rozróżnienie (na poziomie *middleware*) lokalnych i zdalnych dostępów do zasobów.

Model systemu rozproszonego przyjmowany przez TFA zakłada, poza asynchronizmem przetwarzania i rozproszeniem zasobów także kilka innych cech. Po pierwsze jednostką dostępu do puli zasobów współdzielonych (a więc t-obiektem) jest zawsze obiekt języka programowania obiektowego. Po drugie lokalizacje wszystkich obiektów współdzielonych są z góry znane i dostępne na żądanie (przy wykorzystaniu np. rozproszonego katalogu obiektów). Lokalizacje obiektów można także zmieniać dynamicznie. Wreszcie wszystkie węzły są połączone, każdy z każdym, kanałami komunikacyjnymi typu *message-passing*, przy czym [53] nie precyzuje, czy kanały takie zachowują porządek FIFO (lub dowolny słabszy).

Jak wspomniano wcześniej, idea algorytmu TFA bazuje na TL2 z kilkoma wyjątkami. W kwestii struktur danych zachowuje zamki wersjonowane skojarzone z obiektami, rezygnuje jednak z globalnego licznika wersji na rzecz liczników lokalnych synchronizowanych przy użyciu



Rysunek 2.9: Wyniki testów wydajnościowych WSTM: dla pojedynczych operacji na tablicy haszowej, przy 16% zapisów (u góry) i dla złożonych operacji na tablicy haszowej (4096 elementów, 2 modyfikacje w jednej transakcji) (na dole). Za: [30, rysunek 7].



Rysunek 2.10: Przykład niepotrzebnego wycofania transakcji w zoptymalizowanej wersji algorytmu TL2.

mechanizmu zbliżonego do zegarów skalarnych Lamporta [41].

Algorytm TFA przedstawia się następująco:

- `start()` – ta procedura alokuje pamięć dla lokalnego redo-logu i read-setu. Ponadto kopiuje zawartość lokalnego licznika wersji, `lc` (*local clock*) do zmiennej lokalnej względem transakcji, tutaj nazywanej `wv` (*write version*).
- `read()` – operacja odczytu może dotyczyć obiektu przechowywanego lokalnie lub zdalnego. W przypadku obiektu lokalnego należy rozróżnić dwie sytuacje. Jeżeli obiekt już znajduje się we write-secie (nawet jeżeli *de facto* byłby zlokalizowany na innym węźle, co sprawdzane jest z użyciem filtru Blooma), to należy zwrócić wartość uprzednio zapisaną przez transakcję. Jeżeli zaś obiekt ma być odczytany bezpośrednio z przechowywanej lokalnie części zasobów współdzielonych, należy go stamtąd pobrać, sprawdzając uprzednio spójność odczytywanego obrazu równoważną spełnieniu warunku `vwl.version` $\leq$ `wv`. W przypadku obiektów zdalnych, przed odczytem należy taki obiekt *transakcyjnie otworzyć* wykorzystując mechanizm przesuwania transakcji opisany dalej. Sam zdalny odczyt sprowadza się do wysłania żądania, zawierającego też wartość zmiennej `lc` (odbiorca do otrzymanej wartości `lc` i własnego lokalnego numeru wersji stosuje mechanizm Lamporta) i odebrania kopii obiektu, razem z numerem wersji i wartością zegara lokalnego jej dotychczasowego posiadacza (z perspektywy odbiorcy odpowiedzi nazywanego `rc`, *remote clock*). Po odebraniu odpowiedzi strona inicjująca odczyt stosuje do wartości `rc` i `lc` mechanizm Lamporta.
- `write()` – procedura zapisu nie różni się od wykorzystywanej w algorytmie TL2. Zapis polega na umieszczeniu identyfikatora obiektu i jego nowej wartości w redo-logu. By jednak mechanizm przesuwania transakcji działał poprawnie, należy uprzednio transakcyjnie otworzyć również i ten, zapisywany obiekt.
- `commit()` – procedura zatwierdzania transakcji w pierwszej kolejności wykonuje zamknięcie zamków wszystkich obiektów wyszczególnionych w redo-logu (w tym – zdalne dla obiektów, które nie są przechowywane lokalnie). Następnie odbywa się sprawdzenie spójności obrazu puli obiektów współdzielonych w sposób analogiczny do TL2: dla każdego obiektu z read-setu sprawdzany jest warunek `vwl.version` $\leq$ `wv` (przy czym wersja odczytywana jest zawsze z zamka wersjonowanego lokalnej kopii obiektu). Przy jego niespełnieniu dla któregokolwiek z obiektów transakcja jest wycofywana. Po skutecznej weryfikacji następuje wdrożenie zmian z redo-logu. Wpierw inkrementowana jest wartość licznika `lc`, następnie do obiektów z write-setu stosowane są zbuforowane wcześniej w redo-logu zmiany. Każdemu takiemu obiektowi przypisywany jest także nowy numer wersji równy `lc`. Ponieważ w wyniku „transakcyjnego otwarcia" węzeł dysponuje lokalną kopią każdego obiektu z write-setu, może stosować zmiany bezpośrednio do tej kopii, a przed zwolnieniem zamka zdalnego obiektu, oznaczyć siebie jako nowego właściciela.
- `abort()` – wycofanie transakcji to nic innego jak zwolnienie pozyskanych do tej pory zamków i wyczyszczenie redo-logu oraz read-setu.

Saad w [53, str. 27] zwraca uwagę na fakt, że o ile zweryfikowanie spójności obrazu odczy-

```
1  object open_transactional(uuid object_id) {
2
3      // ta sama procedura dostępu do obiektów lokalnych i zdalnych
4      node owner =  find_object_owner(object_id);
5      remote_object obj = owner.retrieve_object(this_node, object_id);
6      if (obj.is_remote) {
7          // mechanizm Lamporta
8          if (lc < obj.rc) {
9              lc = obj.rc;
10
11         // przesuwanie wv do wartości zegara poprzedniego
12         // posiadacza obiektu
13         if (wv < obj.rc) {
14             // weryfikacja read-setu względem nowej
15             // wartości wv
16             forall (object rw_obj in transaction.read_set) {
17                 if (rw_obj.vwl.version > obj.rc) {
18                     return rollback();
19                 }
20             }
21             wv = obj.rc;
22         }
23     } else {
24         if (((object) obj).vwl.version > wv) {
25             return rollback();
26         }
27     }
28
29     return (object) obj;
30 }
```

Rysunek 2.11: Algorytm transakcyjnego otwierania obiektu. Algorytm rozróżnia obiekty lokalne i zdalne, nie rozróżnia zaś celu otwierania (odczyt/zapis). Za: [53].

tywanego przez transakcję w zakresie obiektów lokalnych jest proste (sprowadza się bowiem do sprawdzenia pojedynczego warunku – porównania wersji obiektu z `lc`), jest ono bez wątpienia trudniejsze jeżeli transakcja odczytuje także obiekty zdalne. Te bowiem swój numer wersji odnoszą do lokalnego licznika wersji zdalnego węzła, `rc`. W tym celu w TFA wprowadzony jest dodatkowy względem TL2 krok – *przesuwanie transakcji*. Wbrew temu co może sugerować nazwa, nie chodzi o przemieszczanie transakcji między węzłami, co zresztą byłoby sprzeczne z założeniami koncepcji data-flow, ale o jej przyspieszenie (przesunięcie tak, by móc ją odnieść do zegara węzła który zaszedł dalej w przetwarzaniu) tj. przemieszczenie w prawo w historii wykonania[5]. Dla obiektów lokalnych warunkiem zachowania spójności postrzeganego przez transakcję obrazu jest bowiem $\texttt{vwl.version} \leq \texttt{wv} = \texttt{lc}$, natomiast dla obiektów zdalnych odpowiednikiem `wv` będzie pewne `wv'`, być może większe od `wv`, a wartość swoją biorące od `rc`, tj. od `lc` węzła-posiadacza obiektu zdalnego. Sprawdzenie takiego warunku pozwoli odpowiedzieć nie tylko na pytanie czy od momentu rozpoczęcia aktualnej transakcji jakaś inna lokalna transakcja nie zmieniła stanu obiektu, ale też czy nie zrobiła tego żadna transakcja zdalna. Procedury transakcyjnego otwarcia obiektu, czy to odczytywanego, czy zapisywanego oraz odpowiedzi na żądanie udostępnienia obiektu prezentuje odpowiednio algorytm 2.11 i 2.12. W obu algorytmach uwzględniono mechanizm Lamporta.

[5] jeżeli jest tu koniecznie potrzebna jakaś analogia, niech będzie nią funkcja *fast-forward* w odtwarzaczach kasetowych.

```
1  object retrieve_object(node requesting_node, uuid object_id) {
2
3      // mechanizm Lamporta
4      if (lc < requesting_node.lc) {
5          lc = requesting_node.lc;
6      }
7
8      return get_local_object(object_id);
9  }
```

Rysunek 2.12: Algorytm odpowiedzi na żądanie transakcyjengo otwarcia obiektu. Za: [53].

Jak widać, przesunięcie transakcji powoduje synchronizację `wv` węzła wykonującego z wartością `rc` z momentu udostępniania obiektu (po odczytaniu całego read-setu, `wv` to maksimum po `rc`) oraz wymusza przez to w procedurze `commit()` sprawdzenie czy od momentu rozpoczęcia transakcji (zapamiętania wartości `lc` w `wv`, przy czym `lc` jest synchronizowany z zegarem każdego kto żąda od nas obiektu) do momentu odczytu (transakcyjnego otwarcia) stan obiektu nie uległ zmianie.

Prosty przykład działania algorytmu TFA przedstawiono na rysunku 2.13.

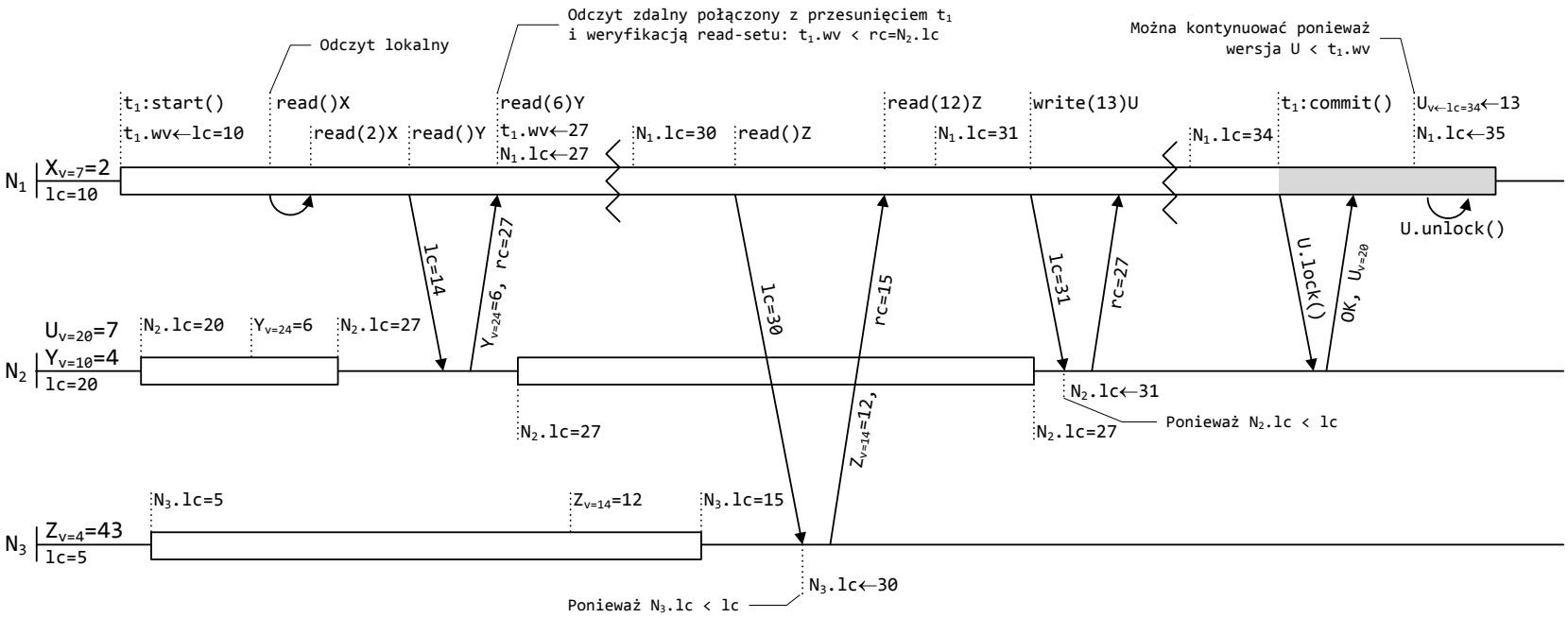
Podsumowując: TFA jest algorytmem optymistycznym, przeznaczonym dla modelu przetwarzania data-flow. Cechują go późne zarządzanie wersjami i wczesne wykrywanie konfliktów. Projekt TFA bierze także pod uwagę wybrane cechy charakterystyczne systemów rozproszonych.

2.3.4 *Supermum Versioning Algorithm*

Supremum Versioning Algorithm i rozproszona pamięć transakcyjna *Atomic RMI* [61], która na nim bazuje zmieniają standardy ustanowione przez przenoszenie rozwiązań w zakresie TM dla systemów wieloprocesorowych do środowisk rozproszonych. Można stwierdzić, że za pewien standard przyjęło się wykorzystywanie podejść optymistycznych typu data-flow (trudno mówić o większym znaczeniu rozwiązań control-flow w systemach wieloprocesorowych), a zwłaszcza replikację rozwiązań lokalnych na węzłach systemu rozproszonego [61, pkt. 2].

W takim ujęciu SVA wydaje się być dość niestandardowym rozwiązaniem. W pierwszej kolejności jest to algorytm pesymistyczny. Konflikty są wykrywane i rozwiązywane dokładnie w momencie ich zajścia. Po drugie konflikty rozwiązywane są nie przez wycofanie jednej z konfliktowych transakcji (poprzez *forcible rollback*), ale przez jej opóźnienie do czasu zatwierdzenia zmian przez drugą, wcześniejszą. Po trzecie, stosując rozwiązanie intuicyjnie bardziej odpowiednie dla systemów rozproszonych (rozwiązanie potencjalnie tańsze komunikacyjnie niż zapewnianie spójności wielu kopii t-obiektów) SVA jest algorytmem typu control-flow. T-obiekty mają w nim trwale przypisane lokalizacje, a fragmenty kodu transakcji mogą być wykonywane tak lokalnie, jak i zdalnie, z użyciem mechanizmu zdalnego wywoływania procedur, RPC. Dla uproszczenia przyjmiemy, że mechanizm RPC gwarantuje semantykę wykonania dokładnie raz.

Model systemu, w którym działa SVA wywodzić się będzie od jego implementacji, *Atomic*



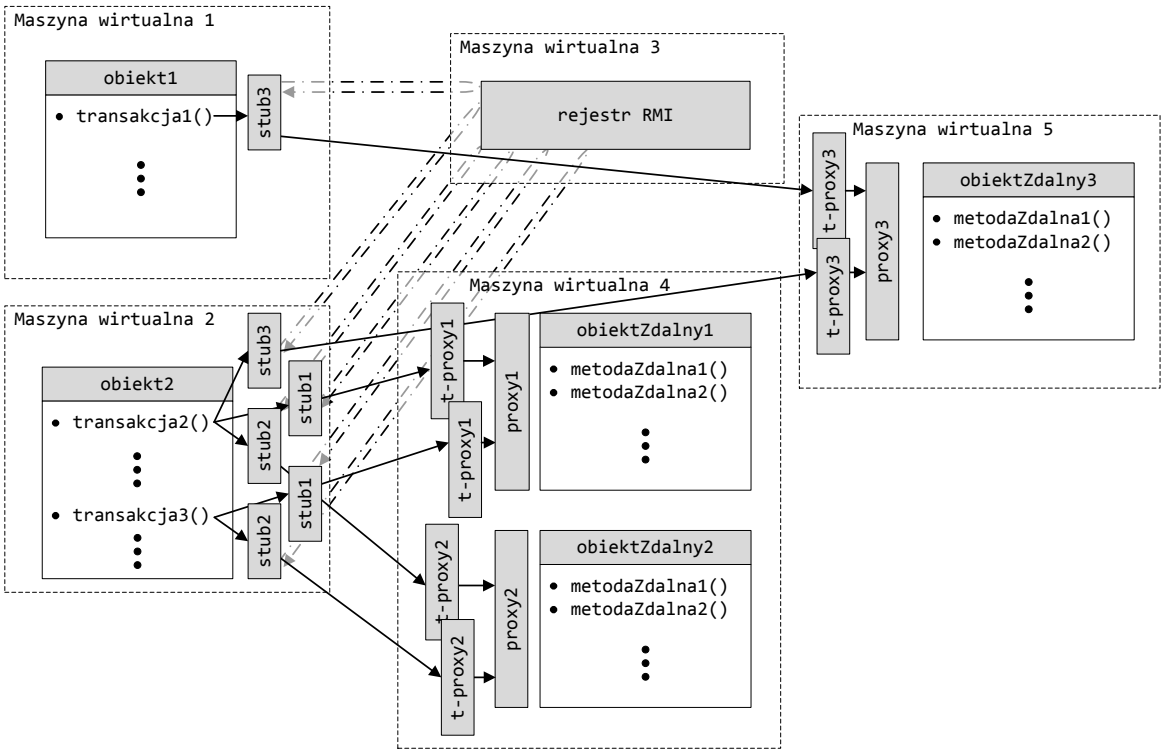
Rysunek 2.13: Przykład działania algorytmu TFA dla trzech węzłów i jednej transakcji. Opracowanie własne na podstawie: [53].

RMI, działającej na bazie mechanizmu RMI (*remote method invocation*) technologii Java. Wybrane (lub wszystkie) węzły przechowują t-obiekty. T-obiekty (obiekty języka programowania obiektowego odczytywane i modyfikowane transakcyjnie) składają się z zestawu pól reprezentujących ich stan oraz metod, które ten stan zmieniają. Niektóre (lub wszystkie) z metod t-obiektów można wywołać zdalnie. Takie wywołanie będzie równoważne dostępowi do puli zasobów współdzielonych, przy czym SVA nie rozróżnia dostępów modyfikujących stan t-obiektu (*setterów*) i jedynie taki stan odczytujących (*getterów*). Transakcja, z perspektywy węzła, który ją inicjuje ma zatem postać szeregu operacji lokalnych i wywołań zdalnych procedur, wykonujących kod transakcji na rzecz t-obiektów posiadanych przez różne węzły. Na potrzeby SVA zakłada się też, że powiązania między t-obiektami mają ograniczony charakter, tj. że zdalna metoda albo ogranicza się do modyfikacji stanu t-obiektu, na rzecz którego jest wykonywana, albo wywołuje zdalne metody innych t-obiektów, uwzględnionych w access-secie transakcji. Problem lokalizowania zdalnych obiektów nie będzie przedmiotem obecnych rozważań (choć jego rozwiązanie nie jest trywialne). Natenczas zakłada się, że każdy węzeł ma dostęp do namiastek (*stubów* – namiastek wykorzystywanych przez klienta wywołującego zdalną metodę, w przeciwieństwie do *proxy* – ewentualnych pośredników po stronie serwera dostarczającego jej implementację) każdego ze zdalnych t-obiektów. Dodatkowo w celu zachowania przezroczystości rozproszenia i uniknięcia omijania TM, dostęp do t-obiektów lokalnych będzie odbywał się również z wykorzystaniem mechanizmu RMI, instrumentowanego przez system D-STM.

SVA wymaga, by zawartość access-setu transakcji była znana przed jej rozpoczęciem. *Atomic RMI* jest więc pamięcią statyczną (w przeciwieństwie do – główne optymistycznych – pamięci dynamicznych, w tym wszystkich przedstawionych wcześniej). Dodatkowo z każdym elementem access-setu skojarzone jest *supremum* liczby wywołań – maksymalna liczba odwołań do t-obiektu w ramach transakcji.

Ideą algorytmu, przy założeniu znajomości access-setu *a priori* jest opóźnienie momentu rozpoczęcia wykonania transakcji do chwili, gdy pierwszy potrzebny jej zasób zostanie zwolniony przez poprzedniczkę. Drugi dostęp opóźniany jest do momentu gdy (być może inna) poprzedniczka zwolni zasób którego on dotyczy, itd. Można więc powiedzieć że transakcje operujące na tym samym t-obiekcie wykonywane są sekwencyjnie. Należy jednak wziąć pod uwagę fakt, że w momencie gdy transakcja wykona sprecyzowaną liczbę wywołań zdalnych metod, t-obiekt jest wyłączany z access-setu bez względu na to czy zatwierdzenie zmian nastąpiło czy też jeszcze nie, co pozwala następnej zakolejkowanej transakcji zrealizować dostęp – rozpocząć wykonanie współbieżne. Ideę działania algorytmu przedstawia rysunek 2.15.

Tym, co wyróżnia SVA wśród algorytmów pesymistycznych jest możliwość wycofywania transakcji na życzenie użytkownika (*non-forcible rollback*). O ile pesymistyczne algorytmy TM, zwłaszcza te które redukują współbieżność do sekwencyjnego wykonania potencjalnie konfliktowych transakcji, nie wymagają mechanizmu wycofywania z racji tego, że jego implementacja jest kosztowna, a konflikty można rozwiązywać bez jego użycia (opóźniając transakcje), o tyle [61] podaje szereg argumentów przemawiających za dostarczeniem takiego mechanizmu w pesymistycznych pamięciach rozproszonych. Poza dopełnieniem interfejsu programistycznego, czyli umożliwieniem programiście wykonywania operacji *non-forcible rollback* według uznania, będzie to zdolność inteligentnego reagowania na częściowe awarie systemu,

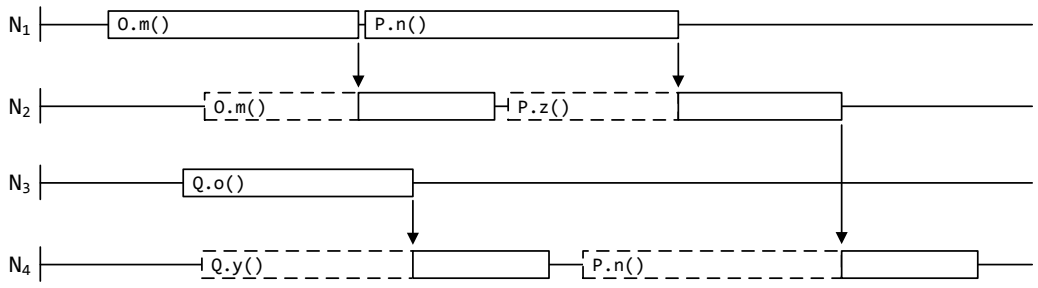


Rysunek 2.14: Poglądowy schemat systemu *Atomic RMI*. Opracowanie własne na podstawie: [47].

charakterystyczne dla systemów rozproszonych. *Atomic RMI* zrzuca ciężar oceny znaczenia awarii na programistę. W przypadku awarii t-obiektu zdalnego, podczas wywoływania metody na jego rzecz wyrzucany jest wyjątek. Programista, przechwytując go, może zdecydować czy awarię należy skompensować, zmieniając scenariusz wykonania transakcji, czy też całą transakcję należy wycofać. Dla porównania, jeżeli to zdalny obiekt straci kontakt z transakcją, jej awarię zakłada się bezwzględnie. Stan obiektu jest wtedy przywracany do tego z momentu tuż przed rozpoczęciem wykonania transakcji ulegającej awarii.

By zagwarantować poprawność uszeregowania transakcji i możliwość ich wycofywania, SVA wprowadza następujące struktury danych:

- `gv` – globalny licznik wersji, definiowany dla implementacji obiektu zdalnego, pozwoli określić ile transakcji zatwierdziło bądź będzie chciało zatwierdzić zmiany tego obiektu. Znając wartość `gv` z momentu rozpoczęcia wykonania, transakcja będzie mogła określić swoją pozycję w kolejce oczekiwania na dostęp,
- `lv` – lokalny licznik wersji, definiowany dla implementacji obiektu zdalnego, będzie zawierał informację o tym ile transakcji zakończyło użytkowanie obiektu (osiągnęło supremum liczby dostępów). Pozwoli więc na określenie warunku wykonania zdalnej metody, jeżeli tylko przypada na to kolej określonej transakcji.
- `ltv` (od ang. *local terminal version counter*) – lokalny licznik wersji końcowej, definiowany dla implementacji obiektu zdalnego, pozwoli na określenie, która transakcja może faktycznie zaprzestać korzystania z obiektu. Transakcja przestaje korzystać z obiektu w mo-



Rysunek 2.15: Uproszczony przykład działania algorytmu SVA z pominięciem obsługi wycofań. Opracowanie własne na podstawie: [62].

mencie, gdy zatwierdza zamiany lub je wycofuje. Ponieważ jednak może odczytywać stan niezatwierdzony (poprzednia transakcja pozwoli na to gdy tylko supremum liczby dostępów zostanie osiągnięte), musi z zatwierdzeniem poczekać do momentu w którym będzie miała pewność, że zmiany, które widziała nie zostaną wycofane. `ltv` będzie w takim momencie zwiększany do wartości gv zaobserwowanej przez transakcję-poprzedniczkę, która ostatecznie zatwierdziła zmiany.

- `cv` (od ang. *current version counter*) – aktualny licznik wersji, definiowany dla implementacji obiektu zdalnego, porównywany z wersją obrazu obiektu przechowywanego przez transakcję w celu ewentualnego odtworzenia (`rv`), posłuży do sprawdzenia w momencie wywoływania zdalnej metody lub zatwierdzania zmian, czy nie należy zaniechać bieżącej operacji z racji tego, że poprzednia transakcja wycofała zmiany zamiast je zatwierdzić.
- `pv` – prywatny licznik wersji, definiowany dla t-obiektu (stuba), przechowuje wartość gv pobraną podczas rozpoczynania transakcji i potrzebną wielokrotnie do sprawdzenia możliwości wykonania metody zdalnej, możliwości zwolnienia obiektu czy możliwości przywrócenia jego wartości po wycofaniu transakcji.
- `cc` (od ang. *call counter*) – licznik wywołań, definiowany dla t-obiektu (stuba), posłuży do sprawdzenia czy supremum liczby dostępów zostało osiągnięte.
- `rv` (od ang. *rollback version*) – licznik wersji do wycofania, definiowany dla t-obiektu (stuba), będzie pamiętał wartość `cv` obiektu z momentu utworzenia dla transakcji jego kopii na wypadek konieczności odtworzenia stanu po wycofaniu.

Z oszczędnym uwzględnieniem powyższych struktur synchronizacyjnych (tj. na wysokim poziomie abstrakcji) można zdefiniować interfejs transakcyjny *Atomic RMI*:

- `start()` – procedura rozpoczynająca wykonanie transakcji inkrementuje gv dla każdego obiektu z access-setu, a w nowo utworzonym stubie każdego takiego obiektu zapamiętuje wartość gv po inkrementacji w liczniku `pv`. Przed rozpoczęciem tej operacji wymagane jest (zdalne) zamknięcie zamków wszystkich obiektów z access-setu (musi się to odbyć w pewnym stałym porządku, by uniknąć zakleszczeń), a po jej zakończeniu – otwarcie zamków.
- `access()` – dostęp do t-obiektu, czyli wywołanie metody zdalnej odbywa się po speł-

nieniu warunku dostępu (`lv + 1 = pv`). W dalszej kolejności, jeżeli jest to pierwszy dostęp do tego t-obiektu w danej transakcji, jego stan jest zapamiętywany. Kolejnym krokiem jest sprawdzenie czy zmiany stanu obiektu, wprowadzone wcześniej, ale jeszcze nie zatwierdzone nie zostały przypadkiem wycofane. Jeżeli tak by się stało, to i bieżącą transakcję trzeba wycofać. Wreszcie odbywa się faktyczne wywołanie metody i inkrementacja licznika wywołań. Jeżeli po inkrementacji okaże się, że supremum liczby dostępów zostało osiągnięte, należy też zwolnić obiekt, pozwalając następnej transakcji na wykrycie spełnienia warunku dostępu.

- `rollback()` – operacja wywoływana czy to na żądanie programisty, czy to z wewnątrz innych operacji interfejsu, dla każdego obiektu z access-setu zwalnia go, a potem przywraca jego stan z przechowywanej lokalnie kopii. Zwolnienie polega na poczekaniu aż wszystkie wcześniejsze transakcje definitywnie zakończą użytkowanie obiektu (tj. zatwierdzą albo wycofają wprowadzone do niego zmiany) oraz umożliwieniu następnym transakcjom dostępu – wykonania pewnej metody zdalnej. Przywrócenie stanu obiektu z kopii zapasowej, poza faktycznym odtworzeniem stanu, odtwarza też wartość licznika `cv` obiektu, a także informuje następne transakcje, że obecna definitywnie zakończyła jego użytkowanie (wycofała zmiany, więc już nic więcej nie może z nim zrobić).
- `commit()` – pierwszym krokiem jest tu zwolnienie obiektów z access-setu w sposób analogiczny do opisanego przy okazji omawiania procedury `rollback()`. Następnie dla każdego obiektu z osobna sprawdzane jest, czy poprzednia transakcja, która wprowadziła obserwowane przez obecną zmiany, ostatecznie ich nie wycofała. Jeżeli choćby dla jednego obiektu tak się stało, transakcję należy wycofać zamiast zatwierdzić, ponieważ operowała na nietrwałym stanie systemu. Gdy zaś weryfikacja powiedzie się, czyszczony jest zbiór kopii zapasowych i wreszcie każdy obiekt z access-setu jest oznaczany jako definitywnie zwolniony.

Rozproszony charakter *Atomic RMI* wymusza zastosowanie specyficznego podejścia architektonicznego. By zapewnić programiście prostotę implementacji obiektów zdalnych, związane z nimi liczniki będą utrzymywane w osobnym komponencie – pośredniku (proxy), na stałe związanym z obiektem zdalnym i przechowywanym w tym samym miejscu co on. Wbrew intuicji jednak liczniki `pv`, `cc` i rv powiązane z t-obiektem występującym w kontekście określonej transakcji nie będą przechowywane w stubie. Należy zwrócić uwagę na to, że m.in. sprawdzenie warunku dostępu wymaga każdorazowo porównania wartości licznika `pv` t-obiektu z wartością `lv` implementacji obiektu zdalnego. By nie odczytywać żadnej z nich zdalnie zdecydowano się na wprowadzenie odpowiednika stuba przechowywanego na tym samym węźle, co zdalny obiekt. Na rysunku 2.14 odpowiedniki te zostały nazwane t-proxy. Utrzymując liczniki przypisane t-obiektowi na tym samym węźle, na którym znajduje się odpowiadający mu obiekt zdalny (razem z jego pośrednikiem), pozwalają one sprawdzić m.in. warunek dostępu bez interakcji z innymi węzłami.

Interakcję transakcji z obiektem zdalnym przy wykorzystaniu t-proxy przedstawia rysunek 2.17. W takim ujęciu należy zaznaczyć, że stub nie musi pełnić żadnej dodatkowej roli względem sposobu jego wykorzystania w podstawowym wariancie mechanizmu RMI.

Kod procedur interfejsu transakcyjnego został przedstawiony na rysunku 2.16. Należy zwró-

```
 1  void start(transaction t, map<remote_object, int> suprema) {
 2      // zgodnie z pewnym porządkiem na zbiorze obiektów w systemie
 3      forall (remote_object obj in suprema.domain) {
 4          obj.lock();
 5      }
 6
 7      forall (remote_object obj in suprema.domain) in parallel {
 8          obj.gv++;
 9          t.proxy(obj).pv = obj.gv;
10      }
11
12      // zgodnie z pewnym porządkiem w zbiorze obiektów w systemie
13      forall (remote_object obj in suprema.domain) {
14          obj.unlock();
15      }
16  }
```

```
17  object call(transaction t, map<remote_object, int> suprema, remote_object o,
              string method_name, object[] arguments) {
18
19      wait until (t.proxy(o).pv - 1 == o.lv); // warunek wykonania
20      checkpoint(t, o); // zachowanie w transakcji kopii obiektu do odtworzenia
21
22      if (t.proxy(o).rv != o.cv) {           // sprawdzenie czy poprzedniczka nie
23          rollback(t, suprema);              // wycofała zmian
24          return TRANSACTION_ROLLED_BACK;
25      }
26
27
28      object result = invoke(o, method_name, arguments); // wykonanie dostępu
29      t.proxy(o).cc++;
30
31
32      if (t.proxy(o).cc == suprema.get(o)) {      // sprawdzenie czy supremum liczby
33          o.cv = t.proxy(o).pv;                   // dostępów zostało osiągnięte
34          o.lv = t.proxy(o).pv;
35      }
36
37      return result;
38  }
```

```
39  void rollback(transaction t, map<remote_object, int> suprema) {
40      forall (remote_object o in suprema.domain) in parallel {
41          dismiss(t, o); // trwałe zwolnienie obiektu o
42          restore(t, o); // przywrócenie stanu o sprzed rozpoczęcia transakcji
43      }
44  }
```

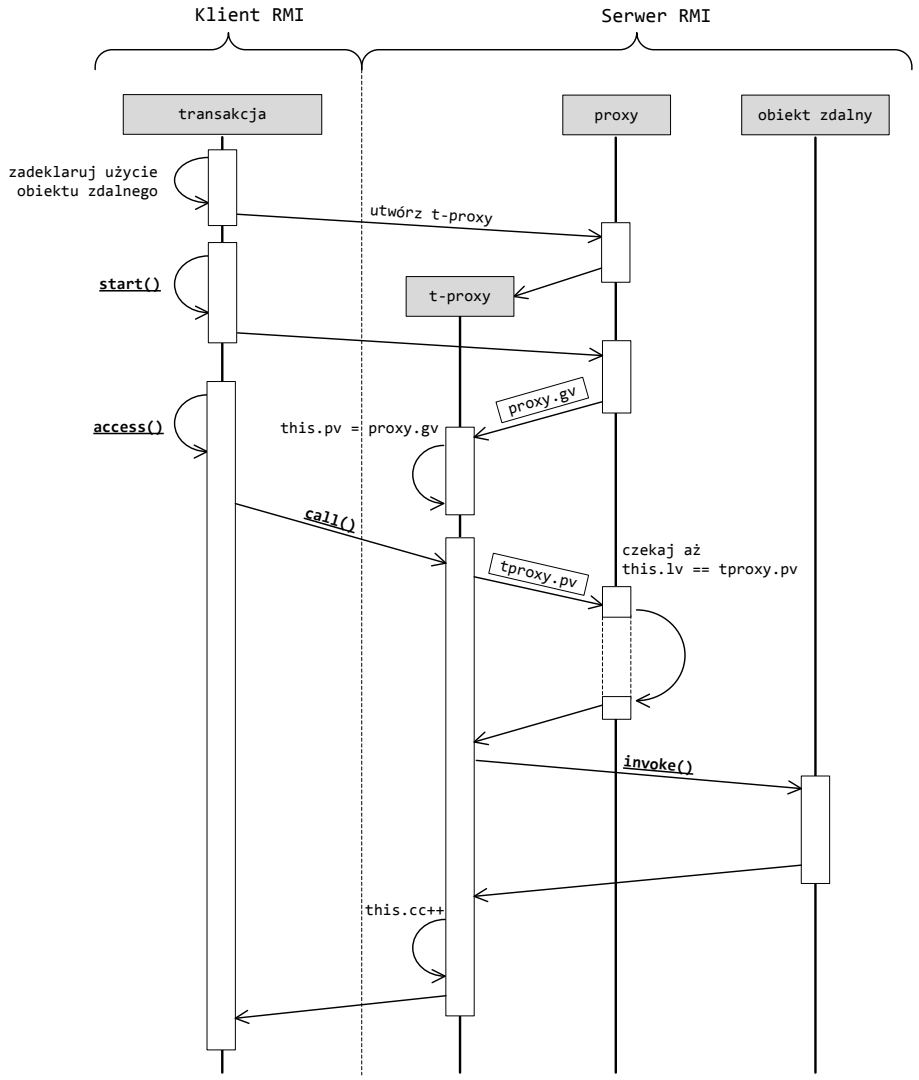
```
45  object commit(transaction t, map<remote_object, int> suprema) {
46      // zwolnienie wszystkich obiektów z access-setu
47      forall (remote_object o in suprema.domain) in parallel {
48          dismiss(t, o);
49      }
50
51      // sprawdzenie czy jakiś obiekt z access-setu został odtworzony po rozpoczęciu
52      // tej transakcji
53      forall (remote_object obj in suprema.domain) {
54          if (t.proxy(obj).rv > obj.cv) {
55              rollback(t, suprema);
56              return TRANSACTION_ROLLED_BACK;
57          }
58      }
59
60      forall (remote_object o in suprema.domain) in parallel {
61          t.remove_copy(o); // usunięcie kopii obiektu zachowanego przez transakcję
62          obj.ltv = t.proxy(obj).pv;          // oznaczenie obiektu jako trwale
63      }                                       // zwolnionego
64
65      return TRANSACTION_COMMITTED;
66  }
```

```
67  void checkpoint(transaction t, remote_object o) {
68      if (t.proxy(o).cc == 0) { // jeżeli nie wykonano jeszcze dostępu
69          t.save_copy(o);                // zachowanie kopii obiektu w transakcji
70          t.proxy(o).rv = obj.cv;        // zapamiętanie wersji kopii obiektu
71      }
72  }
```

```
73  void dismiss(transaction t, remote_object o) {
74      // oczekiwanie na ostateczne zwolnienie obiektu
75      wait until (t.proxy(o).pv - 1 == o.ltv);
76
77      // jeżeli wykonano choć jeden dostęp (i kopię zapasową obiektu przy tej
78      // okazji) oraz poprzednia transakcja nie wycofała zmian...
79      if (t.proxy(o).cc != 0 && t.proxy(o).rv < obj.cv) {
80          o.cv = t.proxy(o).pv;   // dostęp wykonany przez kolejną
81                                  // transakcję nie spowoduje jej wycofania
82      }
83
84      // następna transakcja może wykonać dostęp jeżeli teraz miała do tego prawo
85      // obecna transakcja
86      if (t.proxy(o).pv - 1 == o.lv) {
87          o.lv = t.proxy(o).pv; // równoważne o.lv++
88      }
89  }
```

```
90  void restore(transaction t, remote_object o) {
91      // przywrócenie stanu obiektu z kopii zapasowej jeżeli tylko takowa istnieje
92      // i inna transakcja nie zrobiła tego wcześniej
93      if (t.proxy(o).cc != 0 && t.proxy(o).rv < o.cv) {
94          copy_state(t.get_copy(o), o); // z pominięciem liczników
95          o.cv = t.proxy(o).rv;
96      }
97      t.remove_copy(o);       // usunięcie kopii obiektu o z transakcji
98      o.ltv = t.proxy(o).pv;  // oznaczenie obiektu jako trwale zwolnionego
99  }
```

Rysunek 2.16: Implementacja poszczególnych procedur interfejsu transakcyjnego SVA (w wariancie rozszerzonym – uwzględniającym wycofania). Opracowanie własne na podstawie: [61].



Rysunek 2.17: Schemat interakcji transakcji z obiektem zdalnym w *Atomic RMI*. Na rysunku pominięto blokowanie obiektu na czas pozyskiwania wartości licznika `gv` i inne pomniejsze detale implementacyjne.

cić uwagę na następujące kwestie:

1) Sprawdzenie warunku dostępu/wykonania (linia 19) odbywa się w oparciu o `pv` (dla stuba) i `lv` (dla obiektu). Czekając na spełnienie warunku czekamy właściwie aż poprzedniczka zwiększy `lv` do odpowiedniej wartości. Licznik `lv` jest zwiększany do wartości lokalnej dla transakcji kopii `gv` (czyli `pv`) w momencie zwolnienia obiektu (linia 34, linia 87) w wyniku osiągnięcia supremum (linia 32), ale jeszcze przed (lub w trakcie – por. linia 41, linia 48) zatwierdzeniem lub wycofywaniem zmian.

2) Licznik `rv` jest parametrem stuba (o wartości `cv` obiektu zdalnego z momentu wykonywania jego kopii zapasowej), zaś `cv` – parametrem obiektu zdalnego. Parametrowi `cv` wartość `pv` przypisywana jest w momencie osiągnięcia supremum (linia 33) i (opcjonalnie) w momencie zwalniania obiektu (linia 80) jednak tyko w przypadku gdy wykonano co najmniej jeden

dostęp – jedną potencjalną modyfikację i gdy nikt inny nie wprowadził zmian w widzianym stanie, w szczególności nie wycofał swoich zmian – linia 79). Stąd taki a nie inny warunek operowania na niewycofanych wcześniej zmianach w przypadku dostępu (linie 22) i zatwierdzania (linia 53–58).

3) W momencie osiągnięcia supremum, wykonanie przypisania w linii 33 ma za zadanie przygotować `cv` na potrzeby wykonania kopii zapasowej przez kolejną transakcję, zaś to z linii 34 ma oznaczyć obiekt jako zwolniony przed ostatecznym zatwierdzeniem lub wycofaniem zmian.

4) Spełnienie warunku w linii 54 dla któregokolwiek obiektu oznacza, że jego aktualny stan (`cv`) pochodzi sprzed momentu wykonania przez bieżącą transakcję kopii zapasowej. Któraś z poprzednich transakcji wycofała więc zmiany, zatem i bieżąca, skoro na nich bazuje, musi zostać wycofana.

5) Przypisanie nowej wartości do `ltv` w linii 62 ma za zadanie oznaczenie obiektu jako ostatecznie zwolnionego. W szczególności bieżąca transakcja nie wycofa wprowadzonych do niego zmian bowiem właśnie je zatwierdziła.

6) Warunek w linii 68 zapewnia, że tylko pierwsze odwołanie do obiektu wyzwoli utworzenie jego kopii zapasowej na potrzeby bieżącej transakcji. Oczywiście nie ma potrzeby tworzenia większej liczby kopii, bowiem jeżeli transakcja zostanie wycofana to musi przywrócić stan obiektu pochodzący zawsze z jednego momentu wykonania – tuż sprzed jej rozpoczęcia.

7) Oczekiwanie w linii 75 zapewnia że obiekt zostanie zwolniony dopiero gdy poprzednia transakcja ostatecznie zdecyduje co chce zrobić. Ponieważ dopóki nie dokona wyboru, istnieje ryzyko że wycofa zmiany, `dismiss()` musi spowodować wstrzymanie procedury `commit()` i to jeszcze przed sprawdzeniem czy poprzednia transakcja nie wycofała zmian (linia 53 i następne).

8) Spełnienie warunku w linii 79 oznacza, że poprzednia transakcja nie wycofała zmian, choć już definitywnie zwolniła obiekt (por. zakończenie oczekiwania w linii 75). Można zatem przygotować `cv` do pierwszego dostępu wykonywanego przez następną transakcję, który to dostęp wyzwoli utworzenie kopii zapasowej obiektu.

9) Inkrementacja `lv` w linii 87 ma za zadanie pozwolić na wykonanie pierwszego dostępu opisanego w 8) jednak tylko pod warunkiem, że aktualnie dostępy wykonywać ma prawo bieżąca transakcja. Należy dodać, że wykonanie tego przypisania w ramach operacji `dismiss()` jest niezbędne, gdyż nie zawsze wykona się jego odpowiednik w linii 34. Precyzując, przypisanie z linii 34 nie wykona się w sytuacji, gdy jako supremum dla obiektu wskazano nieskończoność (co jest jak najbardziej dopuszczalne).

10) Pierwsza część warunku w linii 93 ma za zadanie zapewnić, że istnieje kopia zapasowa, z której obiekt jest odtwarzany w linii 94. Jeżeli kopia jest tworzona przy pierwszym dostępie, a dotychczas nie wykonano żadnego, to takiej kopii nie ma. Pierwsza część warunku w linii 79 ma taką samą rolę. W przypadku operacji `rollback()`, która dla każdego obiektu wykonuje najpierw `dismiss()`, a potem `restore()`, modyfikacja `cv` po spełnieniu tego warunku doprowadzi do spełnienia warunku w linii 93.

11) Zmiana wartości `ltv` w linii 98 jest odpowiednikiem dla `rollback()` ostatecznego zwolnienia obiektu w `commit()` (w linii 62).

Bez wątpienia SVA, nie wywodząc się z algorytmów dla systemów wieloprocesorowych, próbuje rozwiązać wiele problemów występujących w środowisku rozproszonym. Potrzeba dostarczenia przez pesymistyczną D-STM mechanizmu niewymuszonych wycofań, choćby jako środka świadomego kompensowania częściowych awarii systemu, nie ulega wątpliwości. Na tym jednak potencjalne zalety stosowania podejścia pesymistycznego w systemach rozproszonych się nie kończą. Zróżnicowane lub duże obciążenie – długie transakcje, z dużymi read- i write-setami oraz wysoki poziom współzawodnictwa, który może być cechą aplikacji wykorzystujących D-STM (rozproszonych) stanowi przesłankę do stosowania mechanizmów stosunkowo odpornych na takie warunki (w podejściu optymistycznym przy dużej liczbie konfliktów wiele transakcji mogłoby być powtarzanych wielokrotnie). W [61] pojawia się sugestia, że wydajność pesymistycznych pamięci transakcyjnych może zależeć od poziomu współzawodnictwa w mniejszym stopniu niż w przypadku ich optymistycznych odpowiedników, co znajduje potwierdzenie w wynikach prezentowanych w [6]. Z drugiej strony, w obliczu możliwych awarii węzłów, *wait-freedom* jest pożądaną cechą rozwiązań, a SVA jako algorytm kolejkujący transakcje tej własności nie zapewnia. Należy tu jednak dodać, że 1) *Atomic RMI* stara się rekompensować tę wadę przy wykorzystaniu detektorów awarii oraz 2) [61] przywołuje udowodnione twierdzenie, mówiące że w systemach z możliwymi awariami węzłów przezroczystość (niezbędny warunek poprawności) i odpowiednik własności *wait-freedom* w systemach rozproszonych, *local progress*, nie mogą być razem zapewnione.

Jednak podstawowym problemem utrudniającym wykorzystanie *Atomic RMI* do tworzenia wydajnych aplikacji rozproszonych będzie statyczny charakter tej pamięci. Szerszą analizę problemu przedstawiono w punkcie 3.4.4, dotyczącym dyskusji wymagań stawianych programom wzorcowym.

2.4 Problemy poprawnościowe systemów pamięci transakcyjnej

Przeniesienie koncepcji transakcji ze świata baz danych, powszechnie wykorzystującego języki deklaratywne – opisujące to jakie zmiany i do jakich danych wprowadzić – do świata języków programowania systemów wieloprocesorowych i rozproszonych, języków imperatywnych – koncentrujących się na sposobie wprowadzania tych zmian niesie ze sobą pewne problemy w kwestii poprawności przetwarzania. Bardziej precyzyjne porównanie dwóch uniwersów znajduje się w sekcji 3.1 jednak już teraz, przy okazji przedstawienia szeregu implementacji STM należy wspomnieć o wybranych aspektach ich działania, mogących powodować błędy przetwarzania, czyli naruszenia warunków poprawności i postępu TM, a będących wynikiem takiej a nie innej historii rozwoju pamięci transakcyjnych.

Dla przedstawionych przykładów naruszenie poprawności przetwarzania będzie oczywiste. Nie zmienia to faktu, że dla wytyczenia dalszej ścieżki rozwoju systemów TM, szczególnie dla przejścia od implementacji eksperymentalnych do „produkcyjnych", wykorzystywanych

w praktyce, niezbędne jest określenie formalnych kryteriów poprawności pod postacią określenia, kiedy dokładnie TM zachowuje własność bezpieczeństwa (lub analogiczną dla przetwarzania, które nie bazuje na sekcjach krytycznych) jak i postępu. Warunki takie zostały już dla pamięci transakcyjnej określone i zostaną przytoczone jako podsumowanie tego punktu.

2.4.1 Niespójne migawki

Przez analogię do baz danych, zbiór stanów t-obiektów należących do read-setu (odczytywanych przez transakcję), określany wcześniej jako obraz stanu systemu przez nią postrzegany nazywany będzie dalej *migawką* (ang. *snapshot, view*).

Migawka może odpowiadać spójnemu stanowi systemu, co będzie pożądaną cechą implementacji TM. Może jednak też odpowiadać stanowi niespójnemu. Pewne specyficzne sposoby implementowania systemów TM mogą dopuszczać odczytanie niespójnej migawki i dalsze wykonanie transakcji na jej podstawie, a sprawdzenie spójności realizować będą dopiero w momencie zatwierdzania. Dotyczyć to może systemów optymistycznych, wykorzystujących undo-logi (in-place updates, co może przyczyniać się do większego prawdopodobieństwa odczytania niespójności) lub redo-logi (commit-time updates) i nie sprawdzające spójności migawki bezpośrednio po odczycie stanu każdego t-obiektu.

Konsekwencje odczytania niespójnej migawki mogą być różne. Oczywiście jeżeli migawka nie przejdzie weryfikacji spójności podczas zatwierdzania transakcji, transakcja zostanie wycofana i ponowiona. Niespójność nigdy nie wpłynie zatem na dalsze („poza transakcyjne") przetwarzanie. Może jednak wpłynąć na sposób wykonania kodu transakcyjnego. Dwa podstawowe problemy, które mogą wtedy wystąpić podaje [21]:

1. Odczytanie stanu niespójnego może wpłynąć na spełnialność warunku zakończenia pętli, potencjalnie prowadząc do powstania *transakcji pasożytniczych* (ang. *parasitic transactions*), „zawieszających się" na wykonaniu pętli nieskończonych i nigdy nie próbujących zatwierdzić zmian. Jeżeli weryfikacja spójności migawki odbywa się właśnie w momencie zatwierdzania, postęp przetwarzania wątku, który taką transakcję wykonuje jest nieosiągalny. W takiej sytuacji pozostaje jedynie liczyć na zachowanie przez system własności *wait-freedom*, a więc na to, że blokada jednego wątku nie spowoduje zablokowania innych.
2. Jeżeli niespójność odczytanych danych dotyczy wskaźników, może to doprowadzić do prób wykonania nielegalnych dostępów do pamięci. Programista z reguły nie będzie mieć podstaw by sądzić, że taka anomalia jest w ogóle możliwa (np. czytając wartość 2-bajtową nie będzie podejrzewał, że pierwszy bajt pochodzi sprzed modyfikacji, a drugi – nie), zatem najprawdopodobniej nie zabezpieczy transakcji w odpowiedni sposób (np. poprzez obsługę odpowiednich wyjątków). W skrajnym przypadku sytuacja taka może doprowadzić do awaryjnego zakończenia procesu wykonywanego w systemie wieloprocesorowym lub awarii fail-stop, a nawet bizantyjskiej węzła systemu rozproszonego (w przypadku której węzeł kontynuuje przetwarzanie, jednak w sposób niezgodny z jego specyfikacją, a więc błędny).

Autorzy [21] argumentują, że podstawową przyczyną, dla której we wczesnych implementa-

cjach (np. WSTM) nie próbowano rozwiązać tego problemu (oczywiście poza brakiem świadomości jego istnienia) jest koszt sprawdzania spójności migawki po każdym odczycie. Kwestie te próbowano zatem rozwiązywać w inny sposób. Problem pętli nieskończonych eliminowany był poprzez sprawdzanie spójności migawki co pewien czas lub pewną liczbę kroków przetwarzania transakcji, jednak nadal w trakcie jej wykonywania. Problem nielegalnych odwołań do pamięci miałby zostać rozwiązany poprzez modyfikacje środowiska uruchomieniowego, w którym pewne pułapki reagujące na wystąpienie błędu zanim jego efekty spowodują awaryjne zakończenie procesu miałyby w ramach takiej reakcji wymuszać wycofanie i powtórne wykonanie bieżącej transakcji (*forcible rollback*). Jest jasne że oba te rozwiązania jako (przynajmniej potencjalnie) kosztowne mogłyby doprowadzić do sytuacji, w której przetwarzanie równoległe byłoby nieopłacalne (lepszą wydajność uzyskiwałoby się rozwiązując problem przy użyciu algorytmu sekwencyjnego i to potencjalnie niezależnie od liczby wątków w przetwarzaniu równoległym).

Nieco bardziej aktualnymi rozwiązaniami są: stosowanie algorytmów pozwalających niskim kosztem sprawdzać spójność migawki przy okazji każdego odczytu pamięci współdzielonej (wkład TL2 w rozwój TM) lub stosowanie algorytmów pesymistycznych, w których naturze leży brak możliwości odczytania niespójnej migawki (tak jak w SVA).

2.4.2 Problem operacji niewycofywalnych

Operacje niewycofywalne (ang. *irrevocable operations*) będą stanowiły problem dla systemów TM, w których bądź to na życzenie programisty, bądź w wyniku wystąpienia konfliktów wymagane będzie powtórzenie wykonania transakcji. Oczywiście by ponowić wykonanie trzeba wcześniej wycofać wprowadzone zmiany. Transakcja może jednak przedtem wykonywać operacje, których wycofać się nie da. Przykładami takich operacji będą wywołania procedur systemu operacyjnego, operacje wejścia-wyjścia, a pewnym szczególnym przypadkiem będą te operacje, które modyfikują stan pamięci prywatnej wątku lub węzła (z racji stosunkowo prostszych rozwiązań w zakresie radzenia sobie z tym problemem, np. definiowania transakcji jako procedur lub funkcji przeznaczonych do wykonania niepodzielnego dalsza część tego punktu dotyczyć będzie dwóch pierwszych grup).

Pojawiły się różne rozwiązania tego problemu. Z jednej strony system TM może całkowicie zabronić używania operacji niewycofywalnych wewnątrz transakcji. Operacje takie mogą też być buforowane do momentu uzyskania pewności, że transakcja zatwierdziła zmiany (o ile tylko nie są potrzebne do wykonania odczytów). Inne systemy mogą pozwalać na definiowanie wprost transakcji wykonujących takie operacje i z racji tego przeznaczonych do wykonania sekwencyjnego względem każdej innej transakcji pojawiającej się w systemie. Wreszcie można użyć systemów TM, w których transakcje nigdy nie są wycofywane (np. systemów pesymistycznych bez wsparcia dla funkcji *non-forcible rollback*).

Krótkie zestawienie podejść do radzenia sobie m.in. z operacjami niewycofywalnymi, razem z przykładami systemów TM przedstawia [61].

2.4.3 Interakcja kodu nietransakcyjnego z STM

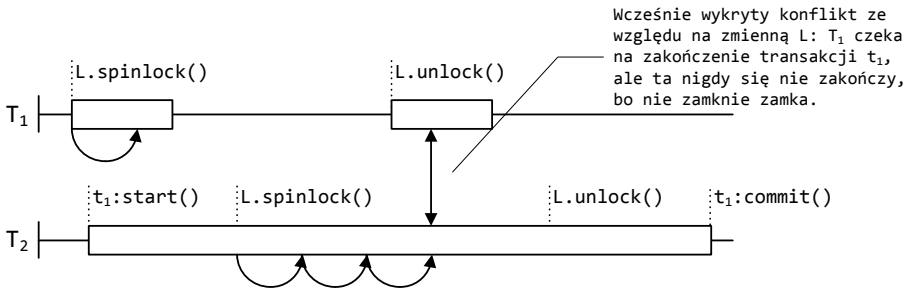
Bez wątpienia nawet przy stosowaniu w aplikacji systemu pamięci transakcyjnej klasy *state-of-the-art* z różnych względów potrzebne może być łączenie kodu wykorzystującego transakcje z kodem używającym innych, bardziej standardowych mechanizmów synchronizacji, np. zamków. Przyczynami mogą być m.in. wykorzystanie w aplikacji kodu, który został stworzony bez użycia transakcji (np. biblioteki kodu) lub brak potrzeby, czy też zasadności (np. w przypadku bardzo krótkich sekcji krytycznych) stosowania transakcji w określonych fragmentach programu.

Analiza anomalii powstających przy łączeniu kodu wykorzystującego transakcje z kodem używającym zamków w celu synchronizacji dostępu do danych dostępna jest w [65]. Autorzy publikacji dokonują jej przy określonych założeniach i wyłącznie dla wieloprocesorowych pamięci transakcyjnych, jednak bez wątpienia co najmniej jeden problem, możliwość wystąpienia zakleszczenia, będzie także udziałem systemów rozproszonych.

Volos i współautorzy zakładają, że w stosie mechanizmów synchronizacji transakcje znajdują się pod zamkami. Zazwyczaj zamki skojarzone z t-obiektami wykorzystywane są przez systemy TM, a rola ich samych jako t-obiektów jest pomijana. W tym przypadku jednak zakłada się, że zamek (raczej będący t-obiektem niż skojarzony z t-obiektem) jest strukturą danych przechowywaną w pamięci współdzielonej i jako taka podlega mechanizmom TM. Podejście to mogłoby sugerować, że pamięciami transakcyjnymi branymi przez autorów pod uwagę są HTM, jednak dla STM rozważania te też będą ważne. Dla każdej patologii autorzy precyzują też charakterystykę systemów TM, których takie patologie dotyczą.

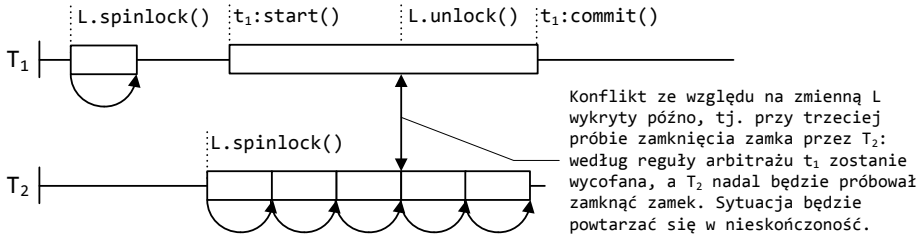
Zakleszczenia (dla systemów z wczesnym zarządzaniem wersjami, wczesnym wykrywaniem konfliktów i regułą arbitrażu, według której transakcja żądająca konfliktowego dostępu jako druga zostaje opóźniona do czasu zatwierdzenia zmian przez pierwszą). Gdy w systemie cechującym się silną atomowością, wewnątrz transakcji nastąpi próba pozyskania zamka (np. w wyniku wywołania procedury z biblioteki kodu) może dojść do następującego scenariusza: wątek T_2 zamyka zamek. Równolegle wątek T_1 rozpoczyna transakcję, w której jedną z pierwszych operacji jest zamknięcie tego samego zamka. Ponieważ ze względu na silną atomowość zamknięcie zamka przez wątek T_2 objęte jest mini-transakcją, dochodzi do konfliktu. Transakcja wykonywana przez T_1 zostaje opóźniona do momentu aż wątek T_2 zakończy zamykanie zamku. Po takim rozwiązaniu konfliktu transakcja sama spróbuje zamknąć zamek, co się jej nie udaje (zostaje zablokowana). Musi poczekać na otwarcie go przez wątek T_2. Jednak wątek T_2 nigdy nie będzie mógł otworzyć zamka. Jego kolejna mini-transakcja – otwierająca zamek – znajdzie się w konflikcie z transakcją wątku T_1, a ze względu na regułę arbitrażu będzie musiała czekać aż wątek T_1 ją zakończy. Jednak wątek T_1 nie może zakończyć transakcji dopóki nie zamknie zamka. Dochodzi do zakleszczenia.

Livelocki (dla systemów z silną atomowością, w których transakcja zatwierdzająca zmiany wygrywa z inną, konfliktową, która dopiero realizuje dostęp do obiektu). Jeżeli procedura zamknięcia zamka jest implementowana w oparciu o aktywne czekanie (ang. *busy waiting*), podczas którego ciągle sprawdza stan struktury współdzielonej, do livelocku doprowadzi następujący scenariusz: wątek T_1 skutecznie zamyka zamek. Wątek T_2 próbuje zamknąć zamek,



Rysunek 2.18: Przykład powstania zakleszczenia podczas interakcji zamków z transakcjami w systemie z wczesnym zarządzaniem wersjami i wczesnym wykrywaniem konfliktów. Opracowanie własne na podstawie: [65].

jednak (ze względu na operację wątku T_1 sprzed chwili) rozpoczyna aktywne czekanie. Wątek T_1 rozpoczyna długą transakcję, w której środku znajduje się instrukcja otwarcia zamka. Ze względu na silną atomowość każda próba dostępu do zamka w celu jego zamknięcia (przez wątek T_2) będzie objęta mini-transakcją. Częste zatwierdzenia tej transakcji spowodują każdorazowo wycofanie transakcji wątku T_1 zwalniającej zamek ze względu na wykrycie konfliktu (T_2 czyta strukturę danych zamka, T_1 próbuje ją zapisać). Pomimo iż nie dochodzi do zakleszczenia, żaden z wątków nie jest w stanie osiągnąć postępu w przetwarzaniu.



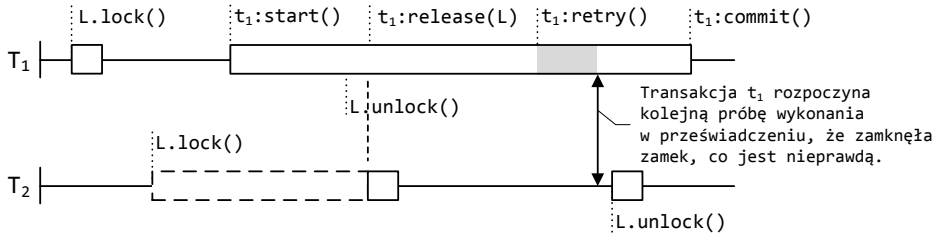
Rysunek 2.19: Przykład powstania livelocku podczas interakcji zamków z transakcjami w systemie zapewniającym silną atomowość. Opracowanie własne na podstawie: [65].

Early release (dla systemów z wczesnym zarządzaniem wersjami, cechujących się słabą atomowością). By zmniejszyć zestaw potencjalnych przyczyn konfliktu, programista wykorzystujący TM może jawnie określić moment, w którym transakcja przestaje wykorzystywać określony t-obiekt. Takie *wczesne zwolnienie* t-obiektu oznaczać będzie w praktyce wyłączenie go z read- lub write-setu transakcji, obserwowane z perspektywy innych wątków. W tej sytuacji nie będzie on mógł być przyczyną konfliktu, w którym omawiana transakcja uczestniczy.

Podręcznikowym przykładem wykorzystania wczesnego zwalniania jest transakcja modyfikująca wybrany element listy jednokierunkowej [31]. Przejście do następnego elementu w poszukiwaniu tego, którego wartość ma zostać zmodyfikowana sprawia, że żadna późniejsza (w tym: zatwierdzona przed uaktualnieniem wartości) modyfikacja bieżącego, nawet związana ze strukturą listy, nie wpłynie już na wynik transakcji. Nie powinna więc być przyczyną konfliktu wykrytego przez którąkolwiek ze stron.

W przypadku prezentowanym w [65] przedwczesne wyłączenie t-obiektu z write-setu może doprowadzić do następującej patologii: wątek T_1 zamyka zamek po czym rozpoczyna trans-

akcję. W jej trakcie będzie ten zamek otwierał i będzie to ostatnie odwołanie do związanej z nim struktury danych – zostanie ona wcześnie zwolniona. Równocześnie z wątkiem T_1 wątek T_2 próbuje zamknąć zamek. Udaje mu się to dopiero gdy wątek T_1 zwalnia go jeszcze przed zatwierdzeniem transakcji i wyłącza z write-setu. Transakcja wątku T_1 może jednak jeszcze zostać wycofana. Wówczas wątek T_2 po zamknięciu zamka nie tylko będzie obserwował niespójności będące skutkiem wycofywania zmian, ale straci też wyłączność na dostęp do sekcji krytycznej. Stanie się tak, ponieważ transakcja wątku T_1 rozpocznie ponowne wykonanie w przeświadczeniu, że to ona zamknęła zamek.



Rysunek 2.20: Przykład naruszenia warunku bezpieczeństwa wzajemnego wykluczania w systemie zapewniającym słabą atomowość i wykorzystującym wczesne zwalnianie. Opracowanie własne na podstawie: [65].

Zamknięcie zamka pozbawione efektu (ang. *invisible locking*) (dla systemów z późnym zarządzaniem wersjami, cechujących się słabą atomowością). Jeżeli transakcja w systemie z późnym wersjonowaniem zamknie zamek, zmiana ta nie będzie widoczna do momentu jej zatwierdzenia. Przy słabej atomowości oznacza to, że po zamknięciu zamka transakcja będzie narażona na współbieżny dostęp do pamięci współdzielonej razem z kodem nietransakcyjnym, który sam zamknął zamek, nie mogąc zaobserwować zbuforowanej przez transakcję zmiany.

Trudno mówić o jednym dobrym rozwiązaniu wspomnianych problemów. Ponieważ dotyczą one systemów o różnych cechach, dobranie takiego, który byłby na wszystkie odporny ograniczałoby skrajnie możliwości dokonania wyborów projektowych. To z kolei mogłoby negatywnie odbić się na wydajności przetwarzania, bowiem pozostałe możliwości nie muszą być w konkretnym zastosowaniu korzystne wydajnościowo.

Tablica 2.1: Patologie interakcji zamków z pamięcią transakcyjną w zależności od rozwiązań przyjętych w ramach jej implementacji. Za: [65].

	Zarządzanie wersjami		Wykrywanie konfliktów		Atomowość
	Wczesne	Późne	Wczesne	Późne	
Zakleszczenia	tak	nie	tak	nie	silna
Livelocki	tak	tak	tak	tak	silna
Problem związany z wczesnym zwalnianiem	tak	nie	tak	tak	słaba
Zamknięcie zamka bez efektu	nie	tak	tak	tak	słaba

W celu rozwiązania problemu interakcji dwóch wspomnianych mechanizmów sterowania współbieżnością [65] wprowadza specjalną implementację zamków, *TxLocks*, w której przy-

kładowo możliwe jest wyłączenie fragmentów kodu objętego transakcją spod nadzoru TM (jeżeli tylko sama pamięć transakcyjna daje taką możliwość). W systemach z późnym zarządzaniem wersjami pozwoli to na pominięcie buforowania zmian przy wykonywaniu operacji na zamkach i w efekcie na uniknięcie zamykania zamków bez widocznego efektu.

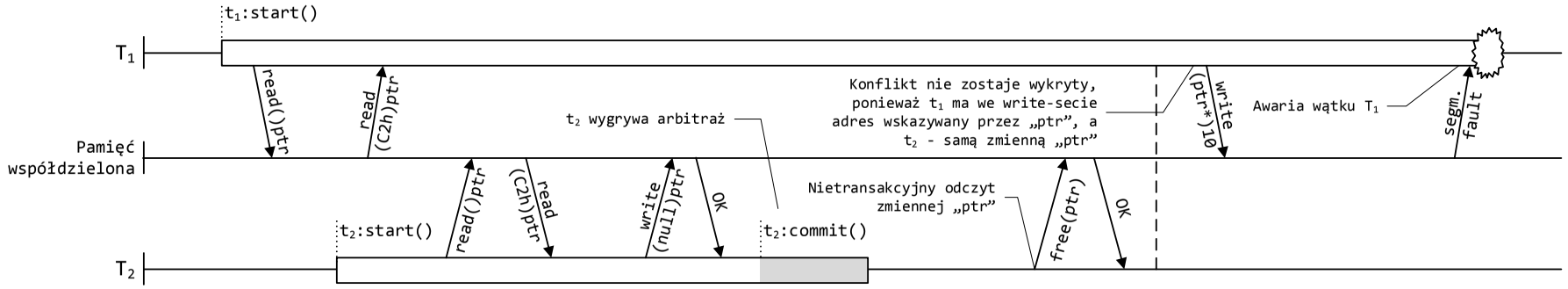
Jak wspomniano wcześniej, podejście, w którym zakłada się, że zamki to struktury danych, do których dostępy realizowane są za pośrednictwem TM, jest podejściem charakterystycznym dla sprzętowych pamięci transakcyjnych, które nie będą wykorzystywane w tym raporcie. Bez wątpienia jednak podobne (jeżeli nie te same) problemy dotyczyć będą implementacji STM bazujących na tych samych zamkach, do których dostęp ma programista (ang. *lock-based STMs*) lub wykorzystujących własny mechanizm blokowania zasobów. Dość wspomnieć o problemie kolejności zamykania zamków dla sekcji krytycznych tworzonych na własny użytek w pierwszym przypadku i brak zachowania własności bezpieczeństwa w drugim. By wyeliminować te problemy, na potrzeby niniejszym raporcie zakładać będziemy, że przygotowujący programy wzorcowe programista aplikacji rozproszonych używać będzie albo transakcji, albo zamków, jednak nigdy obu naraz (w szczególności: w odniesieniu do tego samego obiektu współdzielonego). Ograniczone też zostaną odwołania do procedur pochodzących z zewnętrznych bibliotek kodu, a zwłaszcza tych, które nie dostarczając badanych mechanizmów sterowania współbieżnością implementują algorytmy równoległe i rozproszone (te bowiem mogą być podejrzewane o wykorzystywanie własnych mechanizmów sterowania współbieżnością).

2.4.4 Prywatyzacja współdzielonych struktur danych a transakcje

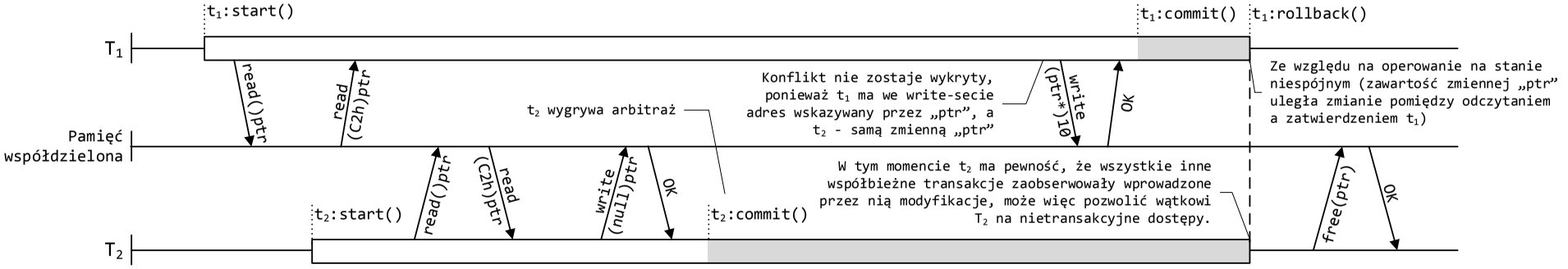
Prywatyzacja (ang. *privatization*) następuje wtedy gdy wątek próbuje na pewien czas zawłaszczyć współdzieloną strukturę danych, po to by móc operować na niej w sposób wyłączny. Podobnie umożliwienie innym wątkom korzystania z tej struktury – jej zwrot do puli zasobów współdzielonych nazywany będzie publikacją (ang. *publication*). W systemie TM prywatyzacja może mieć duże znaczenie z dwóch powodów. Po pierwsze wątki będą mogły chcieć sprywatyzować strukturę danych by operować na niej w sposób wyłączny unikając w ten sposób narzutów związanych z transakcyjnym dostępem do danych. Będzie tak zwłaszcza jeżeli wątek wie, że sprywatyzowanie nie opóźni przetwarzania, bo 1) istnieje małe prawdopodobieństwo, że inny wątek będzie chciał współbieżnie korzystać ze struktury lub też 2) istnieje duże prawdopodobieństwo, że współbieżne korzystanie ze struktury zakończy się powstaniem konfliktu, którego rozwiązanie może okazać się kosztowne. Po drugie, jeżeli prywatyzacja miałaby odbywać się np. poprzez zastąpienie wskaźnika na strukturę pewną wartością wyróżnioną (np. `NULL`), transakcja, jako dostępna od ręki w systemie TM wydaje się najprostszym sposobem podmiany takiego wskaźnika.

Ogólnikowy przykład historii wykonania, w której wymagane jest zapewnienie bezpieczeństwa prywatyzacji (ang. *privatization safety*) podaje [31, str. 136]. Dokładny przykład opracowany na jego podstawie przedstawia rysunek 2.21. Dotyczy on systemów optymistycznych, które nie sprawdzają spójności całej migawki w momencie wykonywania transakcyjnego odczytu (choć mogą sprawdzać przynależność do niej aktualnie odczytywanego elementu).

Problem jest poważny ponieważ stwarza możliwość postrzegania przez transakcje stanu



(a) Błąd segmentacji spowodowany próbą prywatyzacji danych.



(b) Sposób uniknięcia błędu – wydłużenie czasu zatwierdzania transakcji prywatyzującej dane.

Rysunek 2.21: Diagram przestrzenno-czasowy pokazujący zagrożenie spowodowane użyciem prywatyzacji.

niespójnego nawet w systemie, który (w każdych innych okolicznościach) będzie przed taką anomalią zabezpieczony. Przykładowo historia ujęta na rysunku 2.21 jest możliwa do osiągnięcia w systemach działających na bazie algorytmu TL2.

W [70] pojawia się wzmianka o mechanizmie zapewniania bezpieczeństwa prywatyzacji, którego autorstwo przypisywane jest w [31] Hudsonowi – algorytmie wytłumiania (ang. *quiescence*). Zmienna globalna przechowuje w nim numer epoki, inkrementowany w momencie rozpoczęcia każdej transakcji. W założeniu algorytm ten miał zapobiegać anomaliom wynikłym z przedwczesnego zwalniania pamięci przydzielonej t-obiektowi. Jeżeli działoby się to zaraz po zakończeniu transakcji prywatyzującej obiekt a zwolniona pamięć była natychmiast alokowana, istniałoby ryzyko, że inna współbieżna transakcja może postrzegać niespójności wynikłe z użytkowania świeżo zaalokowanego obszaru pamięci (nie zdawałaby sobie sprawy z tego, że w obszarze tym nie ma już t-obiektu). Moment zakończenia zwalniania należy więc opóźnić do chwili, w której numer epoki zapamiętany w momencie żądania zwolnienia pamięci będzie niższy niż aktualne numery epok wszystkich wątków.

Oczywiście z racji ograniczonej współbieżności problem nie dotyczy pesymistycznych algorytmów TM wykonujących transakcje sekwencyjnie.

Zapewnienie bezpieczeństwa prywatyzacji danych będzie, jak widać, kosztownym zadaniem (wprowadzane zostają dodatkowe oczekiwania oraz często zapisywana globalna zmienna, która może stać się źródłem wąskich gardeł). Stanie się ono jednym z elementów dyskusji nad wydajnością pamięci transakcyjnych przedstawionej w punkcie 2.5.1.

2.4.5 Błędne wykorzystanie wczesnego zwalniania obiektów

Wczesne wyłączane t-obiektu z read- lub write-setu, czyli takie które następuje jeszcze przed zatwierdzeniem transakcji to mechanizm, który może pomóc w zredukowaniu liczby konfliktów, a tym samym w zwiększeniu wydajności przetwarzania poprzez zredukowanie liczby spowodowanych nimi wycofań.

W [31] znaleźć można przykład wykorzystania tej możliwości systemu TM, którego poprawność jest uwarunkowana znajomością semantyki t-obiektów i operacji, które system w danej chwili może na nich wykonywać.

Załóżmy, że mamy do dyspozycji listę linkowaną elementów – liczb, posortowaną rosnąco. Wykonywane mogą być na niej dwie operacje: wyszukanie określonego elementu i wstawienie nowego. Przy transakcyjnym wyszukiwaniu elementu, który znajduje się przy końcu listy nie ma potrzeby utrzymywania jej początkowych elementów w read-secie. Druga transakcja może wykonać operację wstawienia elementu gdzieś na początku listy i nie spowoduje to konfliktu.

Z drugiej strony, gdyby była także dostępna operacja usunięcia wszystkich elementów większych niż x, wykorzystanie wczesnego zwalniania początku listy mogłoby doprowadzić do następującej anomalii: podczas gdy jeden z wątków cały czas przegląda koniec listy, drugi ucina ją, pozostawiając jedynie kilka pierwszych elementów. Transakcja ucinająca może zakończyć się jako pierwsza nawet jeżeli transakcja przeszukująca jako pierwsza się rozpoczęła. Konflikt pomiędzy odczytaniem nowego ostatniego elementu listy przez transakcję przeszukującą a jego modyfikacją przez transakcję ucinającą nie zostanie wykryty, ponieważ gdy taki

element jest modyfikowany, transakcja przeszukująca nie ma go już w read-secie. Po zakończeniu transakcji ucinającej kod nietransakcyjny może zacząć zwalniać pamięć przypisaną odciętym elementom współbieżnie z dalszym przeszukiwaniem listy.

Jak widać, przedstawiony tu problem jest blisko związany z prywatyzacją. Należy jednak zwrócić uwagę na to, że nie wystąpiłby gdyby nie użycie wczesnego zwalniania.

Dla odmiany wczesne zwalnianie t-obiektów będzie kluczowym elementem pesymistycznego algorytmu SVA. W sytuacji gdy konfliktowe transakcje są wykonywane sekwencyjnie, zredukowanie liczby konfliktów będzie miało kluczowe znaczenie dla wydajności. Przyspieszy bowiem rozpoczęcie wykonania drugiej z konfliktowych transakcji. Dodatkowo wczesne zwalnianie t-obiektu na żądanie (w przeciwieństwie do tego po osiągnięciu supremum) będzie wyjątkowo przydatne w sytuacji, gdy supremum liczby dostępów nie da się ustalić *a priori* w ogóle (w praktyce będzie ono równe nieskończoności), lub gdy jego wartość zależy od spełnienia, bądź nie, pewnego warunku. Wczesne zwalnianie daje się też w sposób trywialny zaimplementować (choćby pod postacią procedury `release(t-object)` API transakcyjnego) symulując wielokrotny dostęp do obiektu podczas pojedynczego wywołania zdalnej metody.

W SVA transakcja może odczytywać zmiany niezatwierdzone, ale nie może zatwierdzać własnych zmian w oparciu o takie odczyty. Ceną za możliwość wczesnego zwalniania obiektów będzie zatem w przypadku tego algorytmu ryzyko powstania kaskadowych wycofań, wyzwalanych przez pojedyncze wycofanie realizowane na żądanie programisty (*non-forcible rollback*).

2.4.6 Kryteria poprawności implementacji systemu TM

Mając na uwadze te z wymienionych problemów poprawnościowych, które nie dotyczą interakcji pamięci transakcyjnej z innymi elementami systemu (w zasadzie będzie to tylko problem radzenia sobie z niespójnością migawek), należy wprowadzić formalne kryterium poprawności implementacji TM – *przezroczystość* (ang. *opacity*).

Autorzy tego kryterium argumentują w [25], że do określenia kiedy implementacja TM jest poprawna, nie wystarczają standardowe kryteria poprawności, takie jak *linearyzacja* [35] (ang. *linearizability*), *uszeregowalność* [37] (ang. *serializability*, *1-copy serializability* dla systemów rozproszonych [7]), *globalna atomowość* [66] (ang. *global atomicity*), *ścisłe szeregowanie* [10] (ang. *rigorous scheduling*), czy *odzyskiwalność* [29] (ang. *recoverability*) ani jakakolwiek ich kombinacja. Stąd potrzeba wprowadzenia kryterium specyficznego dla pamięci transakcyjnych.

Formalne ujęcie kryterium można znaleźć w [25, 26]. Na potrzeby tego raportu zostanie przedstawiona jedynie jego interpretacja. Implementacja TM jest przezroczysta jeżeli [53, punkt 4.1.2]:

1. z perspektywy każdego wątku zmiany do pamięci współdzielonej wprowadzane są tak jakby transakcje były wykonywane sekwencyjnie (*serializability*),
2. takie sekwencyjne wykonanie zachowuje porządek, który miałby wynikać z rozmieszczenia transakcji w czasie rzeczywistym (w oparciu np. o momenty ich zatwierdzenia) (*preserving real-time order*),

3. żadna inna transakcja nie obserwuje modyfikacji wprowadzanych do pamięci przez transakcję będącą w trakcie wykonania, transakcję, która zakończyła wykonanie, ale dopiero oczekuje na możliwość zatwierdzenia zmian, ani przez transakcję wycofaną (*no inconsistent views*).

Można by pokusić się o stwierdzenie, że konflikt jest relacją binarną, tj. że konflikt sprowadza się do tego, że dwie transakcje pozostają ze sobą w konflikcie ze względu na określony t-obiekt. Oczywiście z powodu jednego t-obiektu w konflikcie może pozostawać więcej transakcji, jednak ten uproszczony wariant posłuży tutaj do określenia zjawiska, któremu należałoby zapobiegać projektując system TM. W sytuacji gdy transakcja $\mathtt{t}_1$ pozostaje w konflikcie z $\mathtt{t}_2$ ze względu na obiekt `x`, $\mathtt{t}_2$ z $\mathtt{t}_3$ ze względu na obiekt `y` i $\mathtt{t}_3$ z $\mathtt{t}_1$ ze względu na obiekt `z`, może się zdarzyć, że dla niskiej jakości reguły arbitrażu każda z transakcji przegra co najmniej raz i w efekcie wszystkie zostaną wycofane. Dobra reguła arbitrażu pozwoli zaś jednej z nich zatwierdzić zmiany. Stąd potrzeba określenia warunku postępu przetwarzania w systemie TM.

Autorzy [53] i [61] wykorzystują w tym celu silny warunek postępu (ang. *strong progressiveness*), który podchodzi do problemu w nieco inny, uproszczony sposób. Znowu, podczas gdy formalne ujęcie kryterium będzie można znaleźć w [25], na potrzeby tego raportu zostanie przedstawiona jego interpretacja z [53]. Implementacja TM spełnia silny warunek postępu jeżeli:

1. ze zbioru współbieżnych transakcji, które nie znajdują się w konflikcie wszystkie zatwierdzają swoje zmiany,
2. ze zbioru współbieżnych transakcji, które znajdują się w konflikcie ze względu na pojedynczy t-obiekt, co najmniej jedna zatwierdzi swoje zmiany.

W tym miejscu należy zaznaczyć, że *strong progressiveness* jest najsilniejszym z możliwych warunków postępu dla TM. W kontekście wieloprocesorowych systemów TM używane są niekiedy także inne warunki, np. *lock freedom*, *wait freedom* czy *obstruction freedom* [34, punkt 3.7]

2.5 Problemy wydajnościowe systemów pamięci transakcyjnej

Pomijając argument na rzecz pamięci transakcyjnych, głoszący, że mają służyć one uproszczeniu programowania aplikacji współbieżnych, można wskazać sytuację, w której wysiłek wkładany w ich rozwój jest sztuką dla sztuki. Jeżeli bowiem okazałoby się, że zyski wynikające ze współbieżnego wykonania transakcji (jeżeli pozwala na to brak konfliktów) są konsumowane przez narzuty czasowe powstające przy wykonaniu dodatkowych operacji zabezpieczających każdy dostęp i przy zatwierdzaniu zmian, stosowanie TM w praktyce byłoby nieopłacalne. Nawet jeżeli wziąć pod uwagę argumenty za stosowaniem pamięci transakcyjnych w celu uproszczenia programowania, aspekt wydajnościowy jest nie do pominięcia (choć może bez stawiania tak wyraźnej granicy opłacalności – w sytuacji gdy rozważamy tylko wydajność, algorytm bazujący na każdej dobrej implementacji TM powinien, wykorzystując

kilka wątków, działać szybciej niż algorytm sekwencyjny, a w sytuacji gdy bierzemy pod uwagę także ułatwianie programowania – zwyczajnie im szybciej tym lepiej).

Wydaje się nieprawdopodobne, by narzut czasowy na każdą instrukcję dostępu do pamięci, wycofywanie całych transakcji w przypadku wykrycia konfliktu i kosztowne oraz wykonywane wielokrotnie procedury weryfikacji spójności odczytywanego obrazu mogły być elementami systemu konkurującego z rozwiązaniami sekwencyjnymi (jeżeli tylko w pierwszym przypadku liczba wątków jest mała) i bazującymi na wzajemnym wykluczaniu (jeżeli nie jest).

Wyniki badań wydajnościowych [21, 30, 53] zdają się jednak zaprzeczać temu podejrzeniu. Wśród czynników mających wpływ na taki stan rzeczy można wymienić:

1. optymalizację systemów pod kątem redukcji kosztów zapewnienia spójności odczytywanego obrazu (jak w przypadku TL2),
2. buforowanie zmian dokonywanych przez transakcje i ich późniejsze grupowe wdrażanie (późne zarządzanie wersjami w WSTM, TL2 i TFA), które redukuje przedział czasu rzeczywistego, w którym pamięć współdzielona znajduje się w stanie niespójnym,
3. zmianę podejścia na pesymistyczne z wykonywaniem transakcji sekwencyjnie (razem z możliwymi optymalizacjami takiego podejścia) w środowisku, w którym występuje wysoki poziom współzawodnictwa w dostępie do zasobów współdzielonych,
4. rezygnację z transakcyjnego dostępu do współdzielonych obszarów pamięci tam, gdzie nie jest to konieczne (por. punkt 2.4.3, 2.5.1, podejście z ręczną instrumentacją kodu transakcyjnego i 2.4.4),
5. rezygnację z kosztownych algorytmów zapobiegania anomaliom, jeżeli nie ma powodów sądzić, że wystąpią (por. punkt 2.5.1, koszt zapewnienia bezpieczeństwa prywatyzacji).

2.5.1 Przykłady problemów

Rozsądnym krokiem następującym po zaproponowaniu na tyle zaawansowanych, że dobrych wydajnościowo algorytmów TM jest próba optymalizacji systemów wykorzystujących transakcje w zakresie kwestii, które z algorytmami nie są bezpośrednio związane. Próbę taką podejmuje [70].

Yoo *et al.* badają w oparciu o zestaw aplikacji wzorcowych wydajność pamięci transakcyjnej McRT-STM, optymistycznej, sprawdzającej spójność odczytywanego obrazu dopiero podczas zatwierdzania transakcji i wykorzystującej wczesne zarządzanie wersjami. Próbują też rozszerzyć funkcjonalność systemu o dodatkowe mechanizmy, które kosztem zwiększonego wysiłku programistów-użytkowników TM mogłyby pozwolić na zwiększenie wydajności przetwarzania (jak choćby opisywane dalej ręczne oznaczanie transakcji prywatyzujących dane). McRT-STM jest systemem TM, który pozwala zarówno na automatyczną jak i ręczną instrumentację kodu transakcyjnego.

Fałszywe konflikty. Specyficzna architektura pamięci podręcznej oraz systemu pamięci transakcyjnej może zwiększać ryzyko wystąpienia konfliktów, których można by uniknąć, gdyby

tylko t-obiekty miały mniejsze rozmiary. Pary odczyt-zapis lub zapis-zapis mogą dotyczyć bowiem dwóch, niezależnych od siebie części t-obiektu. W McRT-STM t-obiektem jest linia pamięci podręcznej.

Architektura, w której wątki mają swobodny dostęp do pamięci współdzielonej (w przeciwieństwie np. do obiektów współdzielonych w systemie rozproszonym) i aplikacje, w których wątki korzystają z dużych ilości pamięci operacyjnej to czynniki mogące razem prowadzić do sytuacji, gdy niektóre obszary pamięci współdzielonej wykorzystywane są w sposób wyłączny, niektóre zaś są faktycznie współdzielone. Fałszywe konflikty mogą się zatem ograniczać wyłącznie do przestrzeni współdzielonej, ale mogą też występować pomiędzy dostępami do niej i do zasobów sprywatyzowanych przez wątek. Występowanie tych anomalii nie jest niczym nadzwyczajnym i jeżeli system TM zakłada stałą wielkość t-obiektu, nie da się ich wyeliminować. Autorzy wprowadzają jednak także trzeci typ: konflikty w wyniku fałszywego współdzielenia. McRT-STM jest systemem o architekturze podobnej do WSTM, co oznacza, że adresy grupowane będą w rekordy własności, pełniące na potrzeby TM rolę t-obiektów, tj. będące jednostką na poziomie której wykrywane są konflikty (pomimo, że dostęp transakcyjny dotyczy zawsze pojedynczego adresu).

Ideą optymalizacji jest tutaj ulepszenie sposobu wiązania adresów z rekordami. Ponieważ w tym celu stosowana jest funkcja haszowa, zwiększenie liczności jej przeciwdziedziny powinno pomóc w zredukowaniu zjawiska.

Jest to optymalizacja specyficzna dla określonej architektury pamięci transakcyjnej. Można z niej jednak wyprowadzić ogólną zasadę projektowania TM tak, by ograniczyć fałszywe współdzielenie: należy zmniejszać rozmiary t-obiektów tak długo, jak długo rosnące równolegle narzuty czasowe i pamięciowe będą akceptowalne.

Zapewnienie bezpieczeństwa prywatyzacji. Mechanizm prywatyzacji i jego problematyczność w kontekście systemów transakcyjnych zostały omówione w punkcie 2.4.4. Autorzy [70] skupiają się na problemie zapewnienia bezpieczeństwa prywatyzacji poprzez odczekanie z rozpoczęciem wykonania kodu nietransakcyjnego tak długo, aż wszystkie wątki zaobserwują zmiany wprowadzane przez transakcję prywatyzującą.

Prywatyzacja nie jest mechanizmem, który musi być powszechnie wykorzystywany. W [22, punkt 2.2] pada stwierdzenie, że nie można doszukać się jego użycia w żadnym z 8 benchmarków zestawu STAMP (por. punkt 2.6.1). Rozwiązaniem, które ma na celu ograniczenie jej negatywnego wpływu na wydajność, przytaczanym w [70] jest wprowadzenie mechanizmu pozwalającego jawnie oznaczyć transakcje prywatyzujące dane.

Nadmierna instrumentacja kodu. U podstaw zaobserwowania tej możliwości optymalizacji systemu TM znowu leży specyfika architektury wieloprocesorowej. Jednak również w systemach rozproszonych, nawet bazujących na współdzieleniu puli obiektów w miejsce pamięci operacyjnej może dojść do sytuacji, gdy na pewnych etapach przetwarzania (czy też w wyniku sprywatyzowania) wątek/węzeł będzie korzystał z pewnego zasobu zazwyczaj współdzielonego w sposób wyłączny.

Optymalizacja miałaby tu polegać na rezygnacji z instrumentacji kodu na rzecz wykorzystywania przez programistę transakcji procedur `read()` i `write()` lub też rozszerzeniu moż-

liwości deklaratywnych konstrukcji definiujących transakcje tak, by programista mógł łatwo określić, które dostępy do zasobów współdzielonych nie powinny być realizowane za pośrednictwem TM. Potencjalnie pozwoli to na zmniejszenie liczby dostępów transakcyjnych, a więc i sumy narzutów czasowych z nimi związanych.

Niska amortyzacja narzutów. By zwiększyć poziom współbieżności aplikacji bazujących na wzajemnym wykluczaniu, sekcje krytyczne konstruuje się zazwyczaj tak, by były jak najkrótsze i występowały rzadko. Tłumaczenie takich sekcji krytycznych bezpośrednio na sekcje atomowe może nie być rozwiązaniem dobrym wydajnościowo.

Zapewnienie atomowego wykonania pojedynczej transakcji jest bez wątpienia bardziej kosztowne niż analogiczna gwarancja dla pojedynczej sekcji krytycznej przy wzięciu pod uwagę stałych narzutów czasowych. By zwiększyć wydajność przetwarzania należałoby być może konstruować długie transakcje, które kosztem nieznacznie zwiększonej liczby konfliktów przy znacząco zwiększonej współbieżności amortyzowałyby stałe koszty związane z ich rozpoczęciem i zakończeniem.

Pewną alternatywą jest tu możliwość wprowadzenia specjalnego trybu wykonywania krótkich transakcji sekwencyjnie, tak by osiągnąć zalety przetwarzania z użyciem sekcji krytycznych (niskie koszty stałe i brak narzutów związanych z dostępami przy ograniczonej, ale na krótko, współbieżności), co jest jak najbardziej możliwe do zaimplementowania (por. punkt 2.3.4).

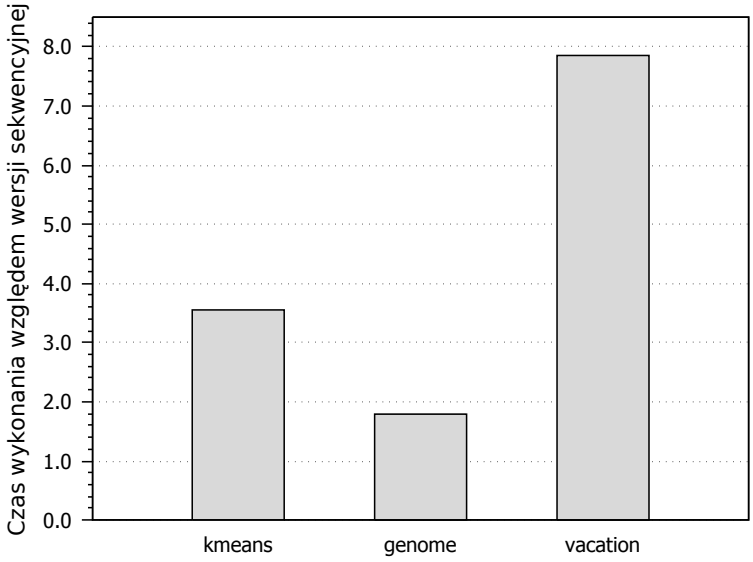
O ile optymalizacje środowiska wykorzystującego transakcje są istotne, o tyle niemniej istotną kwestią jest sposób badania jakości wprowadzonych zmian. Autorzy [70] zwracają uwagę na istotny dla dalszej części tego raportu fakt, że podstawowe sposoby badania wydajności pamięci transakcyjnych, wykorzystujące krótkie, jednorodne transakcje oraz uproszczone i powtarzalne schematy dostępów do zasobów współdzielonych mogą nie reprezentować właściwie przyszłych, „produkcyjnych" sposobów wykorzystywania TM.

Do ewaluacji wymienionych wcześniej ulepszeń wykorzystywane były zaawansowane i rozbudowane aplikacje współbieżne, w tym: silnik fizyki dla gier komputerowych, aplikacja do symulowania dynamiki cząsteczek, narzędzie do rozpoznawania mowy oraz wybrane programy wzorcowe pochodzące z zestawu STAMP, opisywanego szerzej w punkcie 2.6.1.

2.5.2 Dyskusja nad przydatnością systemów pamięci transakcyjnej

Różnorodność wyborów projektowych i szereg kwestii okolicznych, w połączeniu z charakterem pamięci transakcyjnej, wymuszającym obranie pewnych kompromisów oraz wątpliwościami przedstawionymi w punkcie 2.5.1 prowokuje pytanie dotyczące tego na ile poszczególne systemy TM nadają się do tworzenia aplikacji równoległych i rozproszonych, a dokładniej w jakich okolicznościach nawet dobre na ogół rozwiązania osiągają niedopuszczalne wyniki wydajnościowe.

Cascaval ze współautorami przedstawia w [12] wyniki badań wydajności własnej implementacji pamięci transakcyjnej dla systemów wieloprocesorowych, IBM STM oraz odnosi je do dwóch innych implementacji, w tym implementacji algorytmu TL2. Systemy STM badane



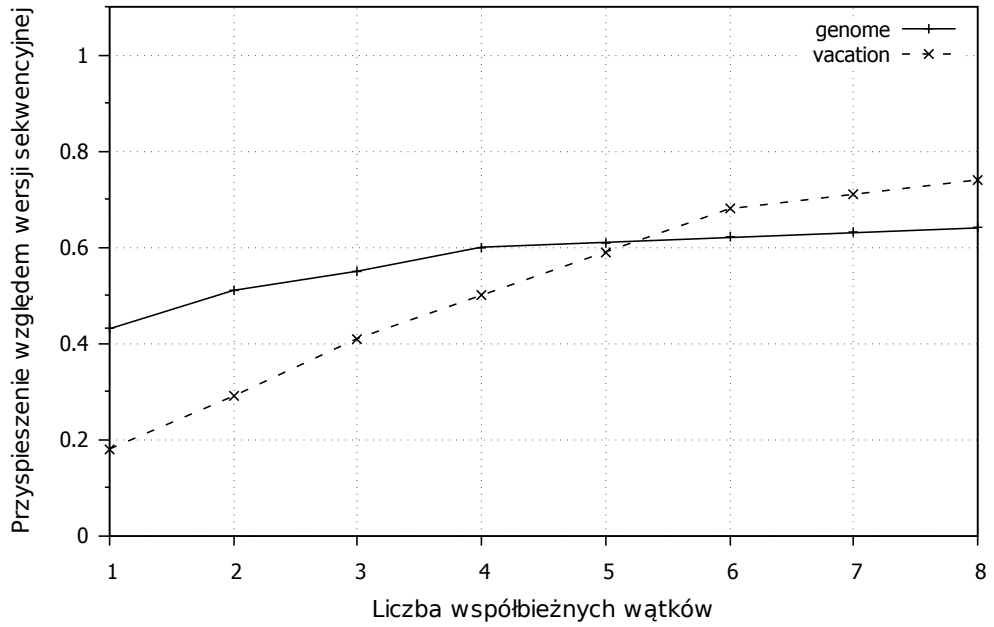
Rysunek 2.22: Narzuty czasowe STM w postaci stosunku czasu wykonania benchmarka transakcyjnego na jednym wątku do czasu wykonywania jego wersji sekwencyjnej dla implementacji TM odpowiadającej funkcjonalnie TL2. Za: [12, rysunek 4].

są z wykorzystaniem różnych benchmarków, w tym – wybranych programów wzorcowych z zestawu STAMP.

Rezultat prezentowany przez autorów przedstawia pamięci transakcyjne w złym świetle: spodziewany wzrost wydajności przetwarzania względem rozwiązań sekwencyjnych nie jest obserwowany. Niejednokrotnie równoległe implementacje programów wzorcowych osiągają czas wykonania współbieżnego przy użyciu znaczącej liczby wątków gorszy niż ich sekwencyjne odpowiedniki. Ponadto rozwiązania wyjątkowo źle się skalują, tj. wzrostowi liczby współbieżnie działających wątków towarzyszy niewspółmiernie mały wzrost wydajności. Kluczowe wyniki badań z [12] przedstawiają rysunki 2.22 i 2.23. Prezentują one odpowiednio czas wykonania transakcyjnych wersji benchmarków na jednym wątku względem wersji sekwencyjnych, tak by uwidocznić narzuty czasowe systemów TM oraz rezultaty skalowania benchmarków wszerz (zwiększania liczby wątków), pokazujące niejednokrotnie załamania wzrostowego trendu wydajności wraz ze wzrostem liczby wątków.

Autorzy dzielą się podejrzeniami dotyczącymi przyczyn tak słabego wyniku, opisanymi już zresztą w tym raporcie (kosztowne sposoby weryfikacji spójności postrzeganego przez transakcję obrazu pamięci, znaczące koszty zapewnienia bezpieczeństwa prywatyzacji, nieoptymalna instrumentacja kodu prowadząca do realizacji dostępu do sprywatyzowanych danych za pośrednictwem TM). Podejmują także próbę wyeliminowania niektórych problemów (ominięcie TM przy dostępach do danych sprywatyzowanych, dostarczenie uproszczonych implementacji operacji `read()` i `write()` wykorzystywanych przy dynamicznym śledzeniu sprywatyzowanych danych, reorganizacja procedury `commit()` tak by uniknąć wąskich gardeł związanych np. z częstą aktualizacją globalnego licznika wersji) jednak bez znaczących rezultatów.

Odpowiedzią na te zarzuty jest publikacja [22], badająca pod kątem wydajności *SwissTM*, system pamięci transakcyjnej, który w ogólnym ujęciu miałby cechować się lepszą wydajno-



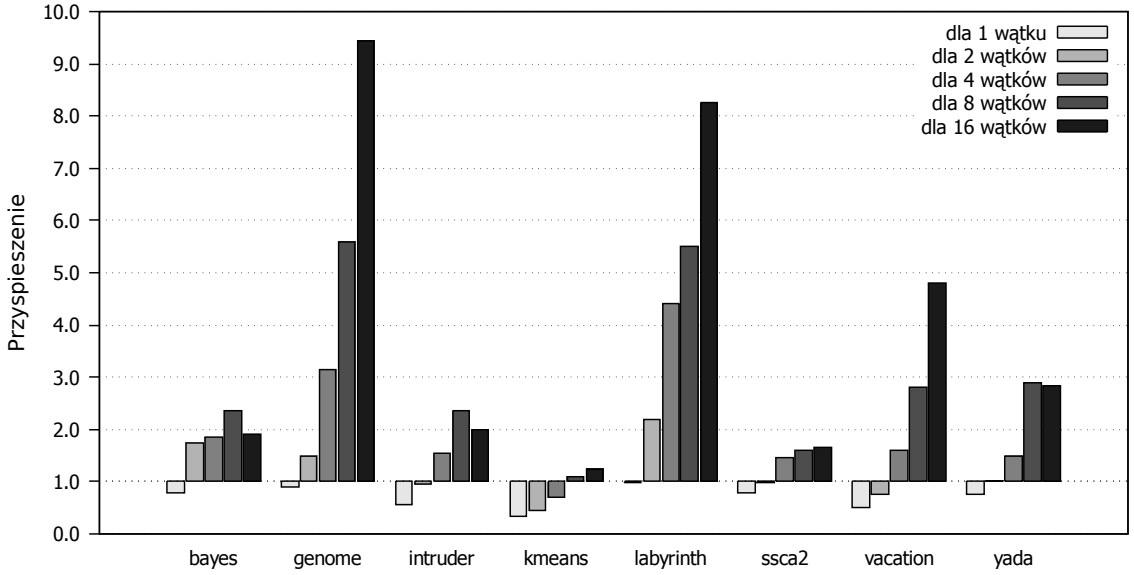
Rysunek 2.23: Przyspieszenie (stosunek czasu wykonania wersji transakcyjnej do wersji sekwencyjnej) wybranych benchmarków. Za: [12, rysunek 3].

ścią (osiąganiem większego przyspieszenia przez programy, które go wykorzystują) niż TL2, porównywalny w tej kwestii z IBM STM [22, str. 6]. *SwissTM* jest pamięcią optymistyczną, zapewniającą postrzeganie spójnego obrazu pamięci współdzielonej w sposób podobny do TL2. *SwissTM* buforuje zapisy, dokonuje odczytów w sposób niewidoczny dla innych transakcji (ang. *transparent/invisible reads*), konflikty odczyt-zapis wykrywa zgodnie z podejściem późnego wykrywania konfliktów, zaś konflikty zapis-zapis – w sposób wczesny (podejście do wykrywania konfliktów znane jako *mixed invalidation*). W przypadku wykrycia konfliktu wycofywana jest transakcja, która do momentu jego wystąpienia wykonała mniej pracy. Powyższe decyzje projektowe były podejmowane z myślą o skonstruowaniu pamięci transakcyjnej zapewniającej dobrą wydajność przetwarzania dla różnego typu obciążeń.

Autorzy rozważają cztery kombinacje dwóch aspektów wskazywanych przez [12] jako przyczyny niskiej wydajności pamięci transakcyjnych: instrumentacji kodu transakcyjnego (automatycznej, wykonywanej przez kompilator lub ręcznej, wykonywana przez programistę stosującego wywołania `read()` i `write()` w miejsce bezpośrednich dostępów) i zapewnienia bezpieczeństwa prywatyzacji (w sposób przezroczysty – przyjęcia, że wszystkie transakcje mogą potencjalnie prywatyzować dane lub oznaczanie przez programistę explicite transakcji prywatyzujących). O ile wszystkie cztery kombinacje zdają się zaprzeczać poprzednim wynikom, podejście z „ręczną instrumentacją” i oznaczaniem transakcji prywatyzujących dane będzie miało szczególne znaczenie, ponieważ odpowiada podejściu z [12].

Wyniki eksperymentów w odniesieniu do programów wzorcowych z zestawu STAMP przedstawiono na rysunku 2.24. Przyspieszenie jest dla *SwissTM* ewidentnie większe niż dla IBM STM. Autorzy podejmują się sformułowania przypuszczeń dotyczących powodów osiągnięcia takich rezultatów:

1. *SwissTM* wykorzystuje algorytm, którego projektanci od początku kładli nacisk na wpływ wybranych rozwiązań na wydajność przetwarzania,
2. Autorzy [12] do testów wykorzystują inny sprzęt. W szczególności procesor użyty do testowania IBM STM umożliwia równoległe wykonanie do 8 wątków przy użyciu jednego procesora czterordzeniowego ze wsparciem hyper-threadingu, podczas gdy dla *SwissTM* system posiada 4 procesory quad-core. Uważa się, że hyper-threading (multipleksacja wątków) ogranicza wydajność w stosunku do tej samej liczby wątków działających na różnych rdzeniach [22, str. 6],
3. Obciążenia generowane przez benchmarki z zestawu STAMP użyte do testowania IBM STM są inne niż te generowane w oparciu o domyślne ustawienia aplikacji i użyte do testowania *SwissTM*. W szczególności trzy benchmarki dla IBM STM pozwalają na podwyższenie poziomu współzawodnictwa przy dostępie do danych współdzielonych. Podejście takie ze względu na zwiększenie liczby konfliktów powoduje obniżenie wyników szczególnie dla optymistycznych pamięci transakcyjnych. Nawet pomimo tego benchmarki z ustawieniami zapożyczonymi z testów IBM STM uruchamiane w oparciu o *SwissTM* dostarczają znacząco lepsze wyniki.



Rysunek 2.24: Przyspieszenie wybranych benchmarków z zestawu STAMP, instrumentowanych ręcznie, przy jawnym oznaczaniu transakcji prywatyzujących dane, wykonywanych na procesorze o architekturze x86. Za: [22, rysunek 3].

2.5.3 Weryfikacja wydajności w praktyce

Wiele z dotychczas przedstawionych problemów razem ze sposobami ich rozwiązania i ewaluacji takich rozwiązań mogłoby już dawać pogląd na to jakie podejście stosować do testowania wydajności systemów pamięci transakcyjnej.

Jak wspomniano we wstępie do tego raportu, podejścia do oceny wydajności pamięci trans-

akcyjnych, bazujące na statycznej analizie kodu są mogą okazać się nieprzydatne (jeżeli da się takowe w ogóle opracować). Dzieje się tak z racji niedeterminizmu przetwarzania, który przykładowo uniemożliwia przewidzenie długości oczekiwania na zwolnienie zasobu przez inny wątek (dla STM bazujących na zamkach). Dodatkowo ani aplikacje mające docelowo wykorzystywać pamięć transakcyjną, ani benchmarki dla niej przeznaczone nie są zazwyczaj pojedynczymi, prostymi algorytmami. Szacowanie złożoności obliczeniowej wydaje się więc w takim ujęciu zadaniem skomplikowanym wręcz do poziomu niemożliwości. A nawet jeżeli w oparciu o znajomość strategii planowania przydziału procesora i ograniczenia czasowe na operacje wejścia-wyjścia można by pokusić się o próbę wykonania takiego zadania dla systemu wieloprocesorowego, to dla systemu rozproszonego, w którym źródła niedeterminizmu wynikają wprost z jego nieodłącznych cech (takich jak losowe powstawanie awarii łączy komunikacyjnych i nieprzewidywalne narzuty czasowe związane z ich ukrywaniem), szacowanie złożoności obliczeniowej jest niewykonalne.

Inne podejście do problemu może polegać na porównywaniu algorytmów TM w oparciu o możliwości, jakie one oferują. Ale i to podejście natrafia na problemy. O ile na przykład algorytmowi zapewniania bezpieczeństwa prywatyzacji można przypisać jednoznaczny, negatywny wpływ na wydajność przetwarzania, a mechanizmowi jego wyłączenia na życzenie programisty – wpływ pozytywny, o tyle nie sposób już jednoznacznie odpowiedzieć czy późne wykrywanie konfliktów będzie wydajnościowo lepsze, niż wczesne. Dodatkowo niezmiernie trudno jest wskazać *a priori* jak często określony mechanizm znajdzie się w użyciu – jaki procent transakcji faktycznie prywatyzuje dane, ile razem powtórzeń transakcji kończy się niepowodzeniem ze względu na konflikty, itd.

W takiej sytuacji dobrym i, jak już wskazano w tym raporcie, wielokrotnie sprawdzonym w praktyce sposobem będzie badanie wydajności pamięci rozproszonych przez ich integrację z odpowiednio przygotowanymi programami wzorcowymi. Programy takie, *benchmarki*[6] powinny cechować się szeregiem specyficznych własności.

- Benchmark nie musi być rzeczywistą aplikacją współbieżną lub rozproszoną. Musi jednak wykorzystywać systemy sterowania współbieżnością w sposób podobny do takich aplikacji.
- Benchmark powinien obciążać system pamięci transakcyjnej w sposób różnorodny. Różne powinny być długości transakcji, rozmiary read- i write-setów oraz ryzyko wystąpienia konfliktów (poziom współzawodnictwa). Klasy transakcji w ramach wykonania algorytmu mogą różnić się charakterystyką lub też benchmark może pozwalać na zadanie parametrów pozwalających później wytworzyć odpowiednie obciążenia[7].
- Pośród generowanych obciążeń powinny się znaleźć szczególnie duże, zarówno pod kątem długości transakcji, jak i poziomu współzawodnictwa.
- Benchmark powinien pozwalać na porównanie różnych metod sterowania współbież-

[6] w zasadzie *benchmarkiem* nazywa się zestaw programów wzorcowych, z których każdy posiada nieco inną charakterystykę, tak by razem były w stanie dogłębnie zbadać wszystkie interesujące cechy systemu. Na potrzeby raportu terminem *benchmark* określany będzie zarówno zestaw takich programów jak i pojedynczy program.

[7] w zależności od typu wykonywanego algorytmu program może albo generować transakcje różnych typów na różnych etapach przetwarzania, w sposób niezwiązany ściśle z parametrami programu albo pozwalać na określenie jawnie za pomocą tych parametrów udziału transakcji określonej klasy w puli wszystkich wykonywanych.

ścią. Musiałby zatem umożliwiać zastąpienie wykorzystywanej pamięci transakcyjnej zamkami drobnoziarnistymi lub globalnym zamkiem współdzielonym.

- Systemy TM powstają dla różnych języków programowania. By możliwe było porównywanie różnych implementacji należy zapewnić przenośność benchmarka (ang. *portability*). Można to osiągnąć na przykład tworząc benchmark łatwy w implementacji.
- Interfejsy programistyczne pamięci transakcyjnej są różne. Jedne implementacje TM mogą udostępniać programiście procedury z API transakcyjnego, podczas gdy inne pozwalają jedynie na określenie sekcji atomowych za pomocą deklaratywnych konstrukcji językowych. Benchmark powinien dawać się łatwo zintegrować z pamięciami transakcyjnymi dostarczającymi różne interfejsy programistyczne.

Pośród benchmarków wyróżnić można dwa podstawowe typy. *Mikrobenchmarki* to proste programy wzorcowe, koncentrujące się zazwyczaj na przetwarzaniu pojedynczej struktury danych. Transakcje będą tam generowane w sposób sztuczny, co oznacza, że ich parametry (długość transakcji, rozmiar read-/write-setu, ryzyko wystąpienia konfliktu) nie wynikają wprost z semantyki rozwiązywanego problemu algorytmicznego i mogą być niemal dowolnie modyfikowane przez uruchamiającego mikrobenchmark. Celem stosowania takiego podejścia będzie zazwyczaj ocena wpływu pojedynczych parametrów transakcji (a co za tym idzie, sposobu implementacji pojedynczych komponentów systemu pamięci transakcyjnej) na wydajność przetwarzania. Sztandarowym przykładem mikrobenchmarka jest *EigenBench*, opisywany dalej w sekcji 2.6. Dla odmiany benchmark pełnowymiarowy często będzie aplikacją, która faktycznie rozwiązuje złożony problem. Generowanie dowolnych zestawów parametrów transakcyjnych – obciążeń – będzie tutaj utrudnione, choć zazwyczaj, w ograniczonym zakresie, cały czas możliwe.

Zasadniczym problemem związanym z mikrobenchmarkami będzie więc takie dobranie parametrów programu, by wygenerowane obciążenia odpowiadały sposobom wykorzystania pamięci transakcyjnej przez rzeczywiste aplikacje. Bez wątpienia przy braku wskazówek, którymi mogłyby być np. pomiary wykonane podczas działania pewnych benchmarków pełnowymiarowych, zadanie to może okazać się trudne. Stąd olbrzymia przydatność tych drugich jako narzędzia badania wydajności TM.

W kontekście systemów wieloprocesorowych można mówić o wielu benchmarkach i mikrobenchmarkach proponowanych w literaturze przedmiotu. Dla rozproszonych pamięci transakcyjnych doszukać się można już jedynie mikrobenchmarków. Analiza przyczyn takiego stanu rzeczy nastąpi w punkcie 3.4.5. W tym momencie dość wspomnieć, że brak odpowiednich narzędzi tego typu, przeznaczonych do wydajnościowego testowania rozproszonych pamięci transakcyjnych motywuje stworzenie niniejszym raporcie.

2.6 Benchmarki dla systemów pamięci transakcyjnej

W dalszej części raportu przedstawiony zostanie szereg benchmarków dla równoległych pamięci transakcyjnych i kilka mikrobenchmarków dla pamięci rozproszonych. W założeniu opisy te mają stanowić punkt wyjścia do analizy możliwości zastosowania benchmarków dla

systemów wieloprocesorowych w systemach rozproszonych oraz możliwości „skomplikowania" mikrobenchmarków dla systemów rozproszonych tak, by mogły pełnić rolę benchmarków pełnowymiarowych.

2.6.1 STAMP

Przy okazji omawiania problemów wydajnościowych dotyczących pamięci transakcyjnych wielokrotnie wspominany był zestaw benchmarków STAMP. Bez wątpienia STAMP (*Stanford Transactional Applications for Multi-Processing*) [13] jest jednym z najpopularniejszych i najbardziej znaczących narzędzi tego typu dla systemów pamięci transakcyjnej.

U podstaw powstania STAMPa leży nieudana próba zebrania zestawu benchmarków spełniających kryteria z punktu 2.5.3. Dodatkowo STAMP dodaje do zbioru kryteriów jeden element: programy wzorcowe wchodzące w skład zestawu benchmarków muszą reprezentować szereg dziedzin informatyki, dla których przypuszcza się, że pochodzące stamtąd przyszłe aplikacje będą wykorzystywać TM. Autorzy formalizują kryteria dla dobrego zestawu benchmarków do postaci trzech punktów:

- **szerokie spektrum testów** (ang. *breadth*) – wykorzystanie TM do budowy aplikacji z wielu różnych dziedzin informatyki,
- **głębokość testów** (ang. *depth*) – wykorzystywanie tych aplikacji do obciążania TM w różny sposób (dostarczenie aplikacji o różnych charakterystykach transakcyjnych),
- **przenośność** (ang. *portability*) – umożliwienie łączenia aplikacji z różnymi systemami pamięci transakcyjnej, tak sprzętowymi jak i programowymi.

Poprzez charakterystykę transakcyjną rozumie się klucz wykorzystywany do opisu i porównywania poszczególnych aplikacji zawierający 1) długość transakcji (w sensie liczby instrukcji nią objętych), 2) rozmiary read- i write-setów, 3) uzyskiwany poziom współzawodnictwa w dostępie do zasobów współdzielonych i 4) procent czasu działania aplikacji poświęcony na wykonywanie transakcji.

Autorzy STAMPa argumentują wprowadzenie do dziedziny własnych rozwiązań nieprzydatnością do oceny wydajnościowej pamięci transakcyjnych powstałych dotąd benchmarków. Z jednej strony w tym celu wykorzystać można by już istniejące benchmarki dla systemów wieloprocesorowych, w których aplikacje używają wzajemnego wykluczania (np. SPLASH-2, NPB OpenMP, SPEComp, PARSEC, por. [13, punkt II.A.]). Jednak obecne w nich sekcje krytyczne są zazwyczaj tak zoptymalizowane, by były jak najkrótsze i występowały stosunkowo rzadko. W efekcie zamienione na transakcje dostarczałyby aplikację, która rzadko z nich korzysta, a jeżeli już to robi, to transakcje są krótkie. Podejście takie nie jest dopuszczalne z dwóch powodów. Po pierwsze wysoki poziom optymalizacji nie reprezentuje sposobu wykorzystania pamięci transakcyjnych (mających ułatwiać programowanie współbieżne) przez niezaawansowanych programistów. Po drugie scenariusz z krótkimi i rzadkimi transakcjami jest tendencyjnie niekorzystny dla systemów pamięci transakcyjnej (por. punkt 2.5.1, niska amortyzacja narzutów).

Z drugiej strony wykorzystać można istniejące benchmarki dla systemów TM (np. STM-

Bench7, RSTMv3, implementacje operacji na drzewach czerwono-czarnych, tworzenie i doskonalenie triangulacji Delaunaya, itd., por. [13, punkt II.B.]). Te jednak zazwyczaj albo przyjmują formę mikrobenchmarków, albo stanowią osobne aplikacje, z których trudno stworzyć jednolitą grupę (co może mieć znaczenie dla przenośności benchmarka). Nawet jeżeli mogą one okazać się przydatne do ewaluacji określonych ulepszeń pojedynczego systemu pamięci transakcyjnej, to z pewnością nie spełniają kryteriów szerokiego spektrum i głębokości testów. W efekcie STAMP dostarcza 8 nowych aplikacji wzorcowych, których zestawienie ujęto w tabeli 2.2. Sumarycznie dostarcza też dla tych aplikacji 30 zestawów parametrów, pozwalających generować różne obciążenia.

STAMP, napisany w języku C, zapewnia przenośność poprzez obudowanie kluczowych punktów przetwarzania – utworzenia i zakończenia nowego wątku, rozpoczęcia przetwarzania, rozpoczęcia wykonywania transakcji, jej zakończenia i każdego transakcyjnego lub nietransakcyjnego dostępu do pamięci współdzielonej wewnątrz niej – makrami preprocesora. Takie podejście pozwoliło na zintegrowanie każdej z aplikacji z symulatorem pamięci transakcyjnej i wyznaczenie charakterystyk ujętych w tabeli 2.2. Z drugiej strony eksperymenty wykonane w oparciu o zintegrowanie aplikacji z właściwymi pamięciami transakcyjnymi pozwoliły zademonstrować użyteczność STAMPa, przykładowo zaprzeczając pewnym stwierdzeniom powszechnie uważanym za prawdziwe.

Powszechnie uważa się, że sprzętowe wsparcie dla pamięci transakcyjnych przyspiesza przetwarzanie. Autorzy STAMPa, testując 6 kombinacji zarządzania wersjami (2 warianty: wczesne lub późne) z poziomem wsparcia sprzętowego (3 warianty: pamięci programowe, sprzętowe i hybrydowe), pokazują na przykładzie benchmarka *bayes* ([13, punkt V.B.1]) i *yada* [13, punkt V.B.8]) że dla bardzo długich transakcji, które budują duże read- i write-sety oraz dla transakcji wykorzystujących małe t-obiekty jest dokładnie odwrotnie. Zarówno transakcje o dużych read-/write-setach jak i małe t-obiekty przyczyniają się do nasilenia występowania fałszywych konfliktów. W pierwszym przypadku ze względu na ograniczoną wielkość transakcji sprzętowych po przekroczeniu pewnej granicy następuje nakładanie się wielu adresów na jednostki, na poziomie których wykrywane są konflikty. W drugim należy brać pod uwagę fakt, że w pamięciach sprzętowych t-obiektem jest zazwyczaj linia pamięci podręcznej (przykładowo 64 bajty), a w pewnych warunkach modyfikacje zmiennych mogą dotyczyć pojedynczych bajtów.

Bez wątpienia STAMP zyskał znaczenie ze względu na trafny dobór problemów rozwiązywanych przez aplikacje składowe. Podczas ich analizy widać wyraźnie, że aplikacje rozwiązują sztandarowe problemy z poszczególnych dziedzin, działają w systemach wieloprocesorowych i mogą skorzystać na zastosowaniu efektywnych metod dostępu do pamięci współdzielonej. W efekcie STAMP jest powszechnie wykorzystywany do oceny wydajności pamięci transakcyjnych [70, 12, 22, 36].

By móc generować różne obciążenia, aplikacje STAMPa muszą umożliwiać skalowanie, np. poprzez dobór liczby współbieżnie działających wątków. Ze względu na założenia dotyczące systemu docelowego (choćby jednorodne koszty dostępu do t-obiektów, czyli brak rozróżnienia dostępów lokalnych i zdalnych) a także czysto równoległy charakter aplikacji, nie będą one się jednak nadawać do ewaluacji rozproszonych pamięci transakcyjnych. Poniższe punkty mają za zadanie przedstawić skrótowo poszczególne aplikacje. Dalsza część raportu (punkt

Tablica 2.2: Benchmarki z zestawu STAMP, ich charakterystyki transakcyjne i dziedziny informatyki, z których pochodzą. Za: [13].

Nazwa	Długość transakcji	Read-/write sety	% czasu w transakcjach	Współzawodnictwo	Dziedzina
bayes	długie transakcje	duże	duży	wysokie	**Uczenie maszynowe**. Benchmark konstruuje sieć bayesowską w oparciu o zestaw obserwacji przy użyciu strategii typu *hill-climbing*.
genome	transakcje średniej długości	średniej wielkości	duży	niskie	**Bioinformatyka**. Benchmark przeprowadza sekwencjonowanie łańcucha DNA.
intruder	krótkie transakcje	średniej wielkości	średni	wysokie	**Bezpieczeństwo sieci**. Benchmark symuluje działanie systemu typu NIDS (ang. *network intrusion detection system Snort*).
kmeans	krótkie transakcje	małe	mały	niskie	**Eksploracja danych**. Benchmark wykonuje algorytm grupowania *k-means* zbioru rekordów w k klastrów.
labyrinth	długie transakcje	duże	duży	wysokie	**CAD**. Benchmark poszukuje ścieżki w trójwymiarowym labiryncie za pomocą algorytmu Lee. Algorytm ten jest wykorzystywany np. do automatycznego wytyczania ścieżek w obwodach drukowanych.
ssca2	krótkie transakcje	małe	mały	niskie	**Zastosowania naukowe**. Benchmark tworzy specyficzną reprezentację grafu w oparciu o reprezentację podstawową.
vacation	transakcje średniej długości	średniej wielkości	wysoki	niskie do średniego	**Relacyjne bazy danych**. Benchmark symuluje działanie relacyjnej bazy danych wykorzystywanej do przechowywania informacji o rezerwacjach przyjmowanych przez biuro podróży.
yada	długie transakcje	duże	duży	średnie	**Zastosowania naukowe**. Benchmark wykonuje algorytm optymalizujący triangulację Delaunaya płaszczyzny w oparciu o zadane kryteria. Triangulacja Delaunaya jest wykorzystywana w symulacjach komputerowych np. dynamiki płynów.

3.5) skupi się na ich dokładniejszej analizie, wskaże dlaczego bez wcześniejszych modyfikacji nie nadają się one dla systemów rozproszonych i przedstawi możliwości wdrożenia takich zmian.

2.6.1.1 Benchmark *bayes*

Benchmark *bayes* buduje dla szeregu zmiennych losowych sieć bayesowską (ang. *bayesian network, belief network*), opisującą warunkowe zależności pomiędzy przyjęciem przez jedne zmienne określonych wartości, a przyjęciem wartości przez inne. Sieć bayesowska to acykliczny graf skierowany (DAG), w którym wierzchołki reprezentują zmienne losowe, a krawędzie – opatrzone prawdopodobieństwem zależności pomiędzy nimi. Innymi słowy, jeżeli pomiędzy zmienną x a zmienną y istnieje krawędź, oznacza to, że gdy x przyjmie pewną wartość, y przyjmie określoną (być może inną) wartość z danym prawdopodobieństwem skojarzonym z krawędzią. W *bayes* zakłada się, że zmienne losowe są binarne, zatem opis taki można uprościć twierdząc, że jeżeli istnieje krawędź z x do y etykietowana prawdopodobieństwem p, to gdy x przyjmie wartość 1, y przyjmie wartość 1 właśnie z prawdopodobieństwem p.

Benchmark *bayes* buduje sieć bayesowską w oparciu o zestaw obserwacji. Każda taka obserwacja, *rekord*, to wektor wartości wszystkich zmiennych losowych z określonej chwili czasu. Proces konstrukcji sieci odbywa się zgodnie ze strategią *hill-climbing*. W praktyce oznacza

to, że rozpoczynając z siecią bez krawędzi (pierwotnym rozwiązaniem problemu), na każdym kroku wyszukiwana jest taka możliwość dodania, zmiany kierunku lub usunięcia połączenia (oczywiście w wypadku usunięcia – poza pierwszym krokiem), by zwiększyć stopień jej adekwatności do zbioru obserwacji. Badanie różnych możliwości ulepszenia sieci może być realizowane współbieżnie. Co warte podkreślenia, przeprowadzenie takiego badania wymaga wprowadzenia do sieci tymczasowej modyfikacji. Objęcie go transakcją jest więc uzasadnione.

Dodatkowo sprawdzenie adekwatności sieci do zbioru obserwacji odbywa się z użyciem pewnej funkcji oceniającej. Ta wykorzystuje do dostarczenia oceny zliczanie rekordów spełniających określone kryteria. By uniknąć pełnego przeglądania zbioru obserwacji za każdym razem gdy do sieci wprowadza się potencjalnie ulepszającą ją zmianę, należy przerobić go na strukturę, która umożliwia zliczanie obserwacji pasujących do zadanych kryteriów mniejszym kosztem.

bayes przed rozpoczęciem konstrukcji sieci przerabia więc zbiór obserwacji na zestaw drzew decyzyjnych (ang. *alternating decision trees*). Ponieważ i ten proces daje się zrównoleglić, transakcje znajdują w nim zastosowanie.

2.6.1.2 Benchmark *genome*

W ogólności problem sekwencjonowania sprowadza się do odtworzenia pełnej, długiej sekwencji symboli z mniejszych jednostek. Jeżeli jednostki te, w oryginalnej sekwencji nakładały się na siebie, problem (przy jej nieznajomości) sprowadza się do ułożenia ich w takiej kolejności, by jak największy końcowy fragment każdej nakładał się na jak największy początkowy fragment następnej. Oznacza to, że by wynikowa sekwencja została uznana za optymalną musi mieć minimalną długość.

Przy sekwencjonowaniu DNA symbolami są *nukleotydy* (`A`, `C`, `T`, `G`), jednostki, z których buduje się sekwencję nazwane są *segmentami*, a sekwencja – *łańcuchem DNA*. Segmenty są znacznie krótsze od sekwencji (przykładowo dla jednego z zestawów parametrów w STAMPie długość segmentu wynosi 8 nukleotydów, zaś długość oryginalnego łańcucha DNA – 32768 nukleotydów). Sposób wytworzenia każdego z nich – wycięcie `n` (np. 8) nukleotydów począwszy od określonej pozycji w łańcuchu zapewnia, że (przynajmniej w teorii) zawsze uda się odbudować oryginalną sekwencję. Segmenty bowiem generowane są tak, by każdy nukleotyd z łańcucha był pokryty co najmniej jednym z nich, a dodatkowo, by każdy segment nakładał się w tym łańcuchu na kolejny co najmniej jednym nukleotydem (liczba segmentów jest więc ograniczona: maksymalnie może być ich tyle, ile jest w łańcuchu możliwych początkowych pozycji – korzystając z wcześniejszego przykładu można obliczyć, że jest ich $32768-8+1$; minimalnie będzie ich tyle, by każdy nukleotyd w łańcuchu był pokryty przez co najmniej jeden segment, a co ósmy – przez 2: $32768/7$).

Proces sekwencjonowania odbywa się w tylu krokach, ile jest możliwych długości nakładających się części segmentów. Dwa różne segmenty o długości 8 nukleotydów mogą nakładać się częściami (końcem jednego na początek drugiego) o długościach od 7 do 1 nukleotydu. Na każdym kroku przetwarzania, iteracji skojarzonej z jedną z długości wyszukiwane są te segmenty, które można dokleić do innych z taką właśnie długością nakładających się

części (nazywanych dalej *overlapami*). Ponieważ iteracje związane z długościami overlapów posortowane są malejąco, najpierw zostaną wyłowione najlepsze dopasowania. Na tym etapie transakcje wykorzystuje się do usuwania segmentów z puli jeszcze nie wykorzystanych, jako że pracę w ramach jednej iteracji można łatwo zrównoleglić.

Samo wyszukiwanie pasujących do siebie nakładających się części jest kosztowne i wykonywane wielokrotnie. By usprawnić ten proces *genome* buduje tablice haszowe, po jednej dla każdej długości takiej części, które zawierają skróty wszystkich możliwych prefiksów każdego segmentu. Wprowadzane jest więc mapowanie:

$$\texttt{overlap\_length} \rightarrow \texttt{hash\_of\_the\_overlap} \rightarrow \texttt{segment}$$

Dodatkowo *genome* konstruuje jeszcze jedną tablicę haszową, która nie dopuszcza duplikatów i nie stosuje funkcji skrótu, by usunąć zduplikowane segmenty. Nawet jeżeli generator segmentów nie zwraca nigdy tego samego fragmentu łańcucha, przez czysty przypadek może się okazać że dwa segmenty z różnych jego miejsc będą identyczne. Występowanie duplikatów poważnie komplikuje proces sekwencjonowania [68].

Do konstrukcji obu typów tablic haszowych wykorzystywana jest pula wątków i transakcje.

2.6.1.3 Benchmark *intruder*

Zapewnienie odpowiedniej wydajności skanerów sieciowych typu NIDS (ang. *network intrussion detection system*) jest problematyczne, zwłaszcza przy rosnących przepustowościach łączy komunikacyjnych. Próbę wykorzystania możliwości wykonywania przez współczesne komputery wielu wątków równolegle, by zwiększyć możliwości takiego skanera – narzędzia *Snort* przedstawia [28]. Autorzy proponują pięć różnych architektur skanera wykorzystujących wątki, co miałoby dobrze rokować choćby z racji tego, że operacje na urządzeniu wejścia-wyjścia jakim jest interfejs sieciowy oraz na repozytorium odebranych pakietów, potencjalnie (ze względu na rozmiar) przechowywanym na dysku twardym mogą wiązać się z opóźnieniami przetwarzania.

„Design 5” jako najbardziej złożona architektura leży u podstaw benchmarka *intruder*. Jego szczegółowy opis będzie przedmiotem dalszej części raportu. Dość wspomnieć, że wielowątkowość jest w nim wykorzystywana przede wszystkim do składania pakietów, które dotarły do stacji poza kolejnością w strumienie danych (ang. *flow reassembly*), normalizacji zawartości tych strumieni do postaci umożliwiającej porównanie z sygnaturami zagrożeń i jej przeszukiwania.

Haagdorens *et al.* prezentują wyniki eksperymentów, z których wynika, że architektura wielowątkowa nie poprawia w znaczący sposób wydajności systemu NIDS. Dla „designu 5” przyczyn takiego stanu rzeczy upatrują oni w nieprzewidywalnym sposobie wykorzystania repozytorium strumieni danych podczas ich odtwarzania z pakietów, co w oryginalnej implementacji skłoniło autorów do stosowania tam mechanizmów synchronizacji dostępu bazujących na globalnym zamku współdzielonym.

STAMPowy *intruder* dostarcza uproszczoną implementację systemu NIDS, w której transakcje wykorzystywane są właśnie do odtwarzania strumieni danych z pakietów oraz do pobierania z repozytorium złożonych już fragmentów celem przeprowadzenia analizy.

2.6.1.4 Benchmark *kmeans*

Algorytm *k-means* (k średnich) [45, str. 75] jest jednym z podstawowych algorytmów grupowania wykorzystywanym w eksploracji danych. Jego zadaniem jest przyporządkowanie `n` elementów posiadających `m` cech o wartościach liczbowych do k grup (klastrów), przy czym to algorytm sam określa te grupy.

W przestrzeni `m`-wymiarowej wyznaczyć można punkty reprezentujące elementy oraz punkty będące środkami klastrów. Dodatkowo jeżeli przestrzeń ta jest przestrzenią euklidesową, można obliczyć odległości między takimi punktami. Początkowe środki klastrów są parametrem zadanym. Alternatywnie mogą też być losowane. Algorytm w szeregu iteracji wykonuje przemieszczenie środków klastrów tak, by przynależność elementów do nich była jak najbardziej wyraźna.

W każdej iteracji do każdego z `n` elementów przypisywany jest jeden z `k` klastrów - ten, którego środek znajduje się najbliżej elementu. Po zakończeniu takiego przypisania oblicza się nowe środki klastrów, o współrzędnych będących średnimi parametrów należących do nich obiektów. Gdy w następnej iteracji okazuje się, że przyporządkowanie elementów klastrom nie zmieniło się (lub też że wykonano zadaną liczbę iteracji), przyporządkowanie uznaje się za ostateczne i algorytm kończy działanie.

kmeans zrównolegla etap obliczania nowych środków klastrów. Każdy wątek otrzymuje pulę elementów do analizy i w pierwszym kroku oblicza, do którego klastra każdy element należy. Ponieważ obliczenie średnich dla każdego wymiaru wymaga najpierw zsumowania pewnych elementów (dodania wartości wymiaru dla każdego elementu przetwarzanego przez wątek do sumy przypisanej wymiarowi klastra), można to wykonać równolegle przy użyciu transakcji. Gdy tylko wszystkie wątki zakończą przypisane im zadania, sumy dla każdego wymiaru każdego klastra dzielone są (sekwencyjnie) przez jego dotychczasową liczność, a następnie są podstawiane w miejsce starych współrzędnych jego środka.

Ponieważ elementów jest generalnie znacznie więcej niż równolegle działających wątków, *kmeans* wprowadza kolejkę zadań, w której wpis określa zakres elementów przetwarzanych jednorazowo przez jeden wątek, a która modyfikowana jest z użyciem transakcji. Zapewne ma to służyć równoważeniu obciążenia wątków.

2.6.1.5 Benchmark *labyrinth*

Benchmark *labyrinth* implementuje algorytm Lee [51] wytyczający ścieżki z określonych punktów początkowych do końcowych w trójwymiarowym labiryncie. Labirynt ma postać kraty (ang. *cuboidal grid*), w której pewne węzły, ściany (ang. *walls*) są wyłączone z użytkowania. Zadaniem algorytmu jest dla szeregu par `(punkt_początkowy, punkt_końcowy)` wyznaczyć ścieżki przejścia cechujące się najniższym kosztem (najmniejszą liczbą mijanych węzłów) oraz brakiem konfliktów (żadne dwie ścieżki nie mogą przechodzić przez ten sam węzeł oraz żadna ścieżka nie może przekraczać ściany).

Algorytm Lee jest algorytmem przeszukiwania grafu wszerz (BFS), który sprowadza się do „zalania" labiryntu znacznikami określającymi odległość węzła od punktu początkowego. Wykonywanie tak rozległych modyfikacji na współdzielonym labiryncie doprowadziłoby do skraj-

nie dużej liczby konfliktów. Stąd w *labyrinth* każdy wątek transakcyjnie kopiuje strukturę labiryntu z wytyczonymi dotychczas ścieżkami do pamięci lokalnej, wykonuje na takiej kopii algorytm Lee, po czym próbuje nanieść wytyczoną ścieżkę z powrotem na współdzielony labirynt (również transakcyjnie). Jeżeli wyszukanie ścieżki w lokalnej kopii nie powiedzie się, należy przejść do następnej pozycji na liście zadań. Jeżeli jednak ścieżka zostanie znaleziona, lecz jej naniesienie na współdzieloną strukturę zakończy się wykryciem konfliktu, należy jeszcze raz wykonać lokalną kopię labiryntu i ponowić wyszukiwanie tej samej ścieżki.

2.6.1.6 Benchmark *ssca2*

SSCA (*Scalable Synthetic Compact Applications*) [5] to zestaw benchmarków dla systemów wieloprocesorowych, badających mechanizmy zapewniania spójności pamięci podręcznych. Zestaw składa się z trzech benchmarków, z których drugi będzie szczególnie interesujący ze względu na wykorzystanie w STAMPie. SSCA2 jest zestawem 4 operacji, które, operując na multigrafach (grafach z dopuszczalnymi wielokrotnymi krawędziami pomiędzy dwoma wierzchołkami), przetwarzać je będą w trybie potokowania.

Każdy kolejny krok potoku nazywany jest kernelem. STAMP, pomimo że implementuje wszystkie 4 kernele SSCA2, do badania systemów pamięci transakcyjnej wykorzystuje jedynie pierwszy, przekształcający reprezentację grafu na taką, która byłaby przyjazna przetwarzaniu w systemach wieloprocesorowych z pamięcią podręczną.

By zmniejszyć opóźnienia związane ze sprowadzaniem danych do pamięci podręcznej (ang. *off-chip access latency*) systemy wieloprocesorowe mogą stosować różne usprawnienia. Jednym z nich jest tzw. prefetching, czyli sprowadzanie danych do pamięci podręcznej z wyprzedzeniem. Ze względu na to, że jest to mechanizm implementowany sprzętowo, nie może być on zbyt skomplikowany. Z drugiej strony prefetching przynoszący pożądany efekt musiałby trafnie przewidywać przyszłe dostępy do pamięci. Na poziomie sprzętu jest to szczególnie trudne, stąd prefetching sprowadza się zazwyczaj do pobierania do pamięci podręcznej wartości spod żądanego przez program adresu i kilku następnych (według logicznej przestrzeni adresowej). Stąd dobry algorytm współbieżny powinien zachowywać czasową i przestrzenną lokalność dostępów do pamięci (ang. *temporal and spatial locality*) tak, by ograniczać liczbę pojedynczych operacji sprowadzenia danych do pamięci podręcznej na żądanie. Gdyby przeanalizować format danych podawanych przez generator instancji na wejście kernela 1 SSCA2, listę trójek `(x, y, w)` w których `x` reprezentuje początek krawędzi, `y` jej końcowy wierzchołek a `w` wagę lub etykietę, mogłoby okazać się, że pod tym kątem jest to jedno z najgorszych możliwych rozwiązań, zwłaszcza że lista nie jest w żaden sposób uporządkowana.

Benchmark *ssca2* buduje reprezentację grafu, która w świetle dostępów realizowanych przez kolejne kernele (częste wyszukiwanie wierzchołków incydentnych z podanym, a dokładnie – następników) jest przyjazna prefetchingowi. Graf jest tam reprezentowany przy pomocy dwóch tablic sąsiedztwa i szeregu tablic pomocniczych. O ile reprezentacja taka pozwoli na łatwe wyszukiwanie krawędzi wychodzących z zadanego wierzchołka (oraz ich wierzchołków końcowych), o tyle kłopotliwe będzie stwierdzenie przy jej pomocy czy określona krawędź nie istnieje. Stąd *ssca2* buduje dwie reprezentacje grafu: standardową i reprezentację dopełnienia grafu (opisującą krawędzie, których w grafie nie ma (ang. *implied edges*)), a dodatkowo także

trzecią pozwalającą dla zadanego wierzchołka wyznaczyć te krawędzie, których wierzchołek jest celem (a tym samym jego poprzedniki). Będzie miało to znaczenie, ponieważ pierwszą reprezentację grafu da się zbudować współbieżnie, nie korzystając prawie wcale z mechanizmów synchronizacji dostępów do danych, podczas gdy kolejnych – już nie.

2.6.1.7 Benchmark *vacation*

Benchmark *vacation* implementuje prostą bazę danych wyposażoną w system przetwarzania transakcyjnego (ang. *on-line transaction processing, OLTP*), której architekturę zapożycza od benchmarka przeznaczonego do ewaluacji środowisk uruchomieniowych technologii Java: SPECjbb2000 [2]. System ten pozwala na dokonywanie rezerwacji w biurze podróży. Struktura danych leżąca u podstaw benchmarka składa się z 4 tabel. Rezerwacje mogą dotyczyć wynajmowanych samochodów, przelotów i pokoi hotelowych. Każda z tych klasa zasobów wymaga osobnej tabeli przechowującej identyfikator, liczbę dostępnych elementów i cenę. Czwarta tabela przechowuje informacje o klientach. Obok identyfikatora klienta zawiera ona kolekcję poczynionych przez niego rezerwacji. By uprościć implementację, związki pomiędzy klientami a rezerwowanymi elementami nie są zawarte w dodatkowej tabeli, lecz są opisywane przy pomocy złożonej struktury danych, umieszczanej w całości w kolumnie tabeli opisującej klientów. W warunkach pamięci transakcyjnej może to być np. wskaźnik na listę linkowaną. Dla odmiany same tabele implementowane są jako struktury drzewiaste (dokładnie: *red-black trees*).

By odróżnić *vacation* od mikrobenchmarka, implementowane są cztery podstawowe sposoby wykorzystania bazy danych – cztery typy transakcji, których charakterystykę można dostroić przy użyciu szeregu parametrów programu. „user tasks" to transakcje, w czasie których użytkownik przegląda fragmenty każdej z trzech tabel z zasobami, po czym dokonuje rezerwacji wybranych spośród przejrzanych elementów. Transakcje „user deletions" sprowadzają się do wystawienia pojedynczemu użytkownikowi rachunku i usunięcia go z systemu (przy jednoczesnym zwolnieniu zarezerwowanych przez niego elementów). „item additions" to transakcje dodające do bazy danych nowe samochody, pokoje i przeloty (transakcje takie mogą też modyfikować już istniejące rekordy). Wreszcie transakcje „item deletions" usuwają losowe elementy z tabel przechowujących zasoby.

2.6.1.8 Benchmark *yada*

Triangulacja Delaunaya pewnej płaszczyzny to taki jej podział na trójkąty, w którym żaden okrąg opisany na trójkącie nie zawiera w sobie punktów-wierzchołków innych trójkątów. Ta metoda podziału płaszczyzny jest często wykorzystywana w symulacjach komputerowych [67]. Triangulacja Delaunaya dopuszcza jednak istnienie pewnych zjawisk, które przy jej wykorzystaniu na potrzeby symulacji komputerowych nie są mile widziane. Płaszczyzna bowiem może zostać podzielona na trójkąty, z których wiele posada jeden lub nawet dwa wyjątkowo ostre kąty (przykładowo mniejsze niż 20 stopni). Trójkąty takie są trudne do poprawnego ujęcia w symulacji. Dodatkowo ich istnienie może zniekształcać własności symulowanego obiektu. Stąd może istnieć potrzeba poprawienia triangulacji tak, by kąty ostre trójkątów, które się na nią składają były zawsze większe od pewnej wartości granicznej.

Algorytmem, który poprawia triangulację Delaunaya do zadanych parametrów jest algorytm Rupperta [52], implementowany przez benchmark *yada*. Działając iteracyjnie usuwa on z triangulacji trójkąty niskiej jakości, zastępując je innymi. Udowodniono, że algorytm zatrzymuje się o ile tylko oczekiwania wobec poprawionej triangulacji nie są zbyt wygórowane. W praktyce oznaczać to będzie kryterium niskiej jakości trójkąta definiowane jako istnienie w nim kąta ostrego mniejszego niż 20,7 stopnia [67, sekcja „Practical usage"].

W naturze problemu, uwzględnianej przez benchmark *yada* leży dodatkowe ograniczenie: wewnątrz dzielonej płaszczyzny mogą znajdować się odcinki, których nie można usunąć, oraz których boki trójkątów nie mogą przecinać. Okręgi opisane na tych odcinkach, tj. takie dla których odcinki te są średnicami również nie mogą zawierać w sobie wierzchołków trójkątów. Mając na uwadze ten fakt, algorytm Ruperta implementowany przez *yada* można wyrazić następująco:

1. dla każdego trójkąta w triangulacji:
   (a) sprawdź czy jego istnienie narusza poprawność triangulacji Delaunaya,
   (b) sprawdź czy któryś z jego kątów ostrych nie jest mniejszy niż wartość graniczna,
   (c) jeżeli a) lub b) jest prawdą, umieść trójkąt w kolejce elementów do przetworzenia,
2. dopóki w kolejce elementów do przetworzenia istnieje co najmniej jeden element `Q`:
   (a) jeżeli `Q` jest odcinkiem, to podziel go na pół; łącząc jego środek z wierzchołkami dwóch trójkątów, których jest bokiem uzyskasz 4 nowe trójkąty, które muszą być dodane do kolejki elementów do przetworzenia,
   (b) jeżeli `Q` jest trójkątem, który ma zbyt mały kąt ostry (lub narusza warunek poprawności triangulacji Delaunaya) to:
      i. jeżeli pewien odcinek `s`, którego nie można usunąć, narusza okrąg opisany na tym trójkącie dodaj taki odcinek do `Q`,
      ii. jeżeli nie to wstaw do triangulacji środek okręgu opisanego na trójkącie, połącz go odcinkami z wszystkimi wierzchołkami trójkąta i wszystkimi innymi wierzchołkami, które możesz osiągnąć bez przecinania boków, usuń te z boków trójkąta, które przecinają nowe odcinki i dodaj wszystkie powstałe w ten sposób trójkąty do `Q`.

Jest oczywiste, że wykonywanie zadań z kolejki `Q` można zrównoleglić. *yada* realizuje dokładnie to podejście, przy czym transakcje są w tym benchmarku wykorzystywane do operowania na kolejce zadań oraz do wprowadzania modyfikacji do triangulacji współdzielonej przez wątki.

2.6.2 Narzędzie *EigenBench*

Typowym zastosowaniem mikrobenchmarków, jako prostych programów, od których nie wymaga się, by korzystały z zasobów współdzielonych w sposób podobny do rzeczywistych, złożonych aplikacji, jest testowanie pojedynczych cech TM (wynikających ze sposobu implementacji jej pojedynczych komponentów) [13]. Bez względu na to, czy stworzenie pro-

gramu wzorcowego wykazującego cechy mikrobenchmarka wynika z intencji twórców, czy też z braku innych, lepszych możliwości, z racji „oderwania od rzeczywistości" niekiedy zarzuca się mikrobenchmarkom, że wyniki wykonywanych przy ich użyciu pomiarów są bezwartościowe, jeżeli nie mylące [6].

Pomimo, że n na mikrobenchmarkach się nie skupia (chyba, że są one elementami badanego systemu TM), przedstawiony w niej zostanie jeden przedstawiciel tego gatunku, *EigenBench* [36]. Jego autorzy, poza samym programem, prezentują bowiem podejście do jego wykorzystania, które mogłoby przeczyć powyższym stwierdzeniom.

Punktem wyjścia przy prezentacji *EigenBench* jest zdefiniowanie pojęcia *charakterystyki transakcyjnej benchmarka* (ang. *workload*[8]) – sposobu w jaki obciąża on system TM, wyrażonego za pomocą pewnych parametrów transakcji. Z założenia mikrobenchmark pozwala dostrajać takie parametry w sposób niemal dowolny, z tym zastrzeżeniem, że mogą one być ze sobą powiązane, tj. że zmiana jednego z nich może nieodwołalnie pociągać za sobą zmianę innego.

Ideą stojącą za *EigenBench* jest wpierw zdefiniowanie takich parametrów transakcji, które byłyby niezależne względem siebie (*ortogonalnych*; stąd użyta w publikacji nazwa *eigen-characteristics* i określenie całej metodyki mianem *analizy ortogonalnej*, ang. *orthogonal analysis*), a dalej opracowanie prostego programu, którego parametry pozwoliłyby każdą z nich z osobna modyfikować. Przy założeniu, że wprowadzone parametry będą wyczerpująco opisywać transakcje (tj. że dwa programy produkujące transakcje o zbliżonych parametrach będą porównywalne w kwestii wydajności), być może mikrobenchmark pozwoli na generowanie obciążeń systemu TM zbliżonych do tych wytwarzanych przez pełnowymiarowe benchmarki.

Tabela 2.3 przedstawia zestaw ortogonalnych parametrów transakcji rozważanych w [36]. Kontrprzykładem może tu być zestaw: rozmiar transakcji (mierzony w liczbie operacji) i rozmiar rw-setu, gdyż oba te parametry w sposób oczywisty są ze sobą powiązane. Rysunek 2.25 przedstawia kod mikrobenchmarka. Wreszcie tabela 2.4 podaje wybrane powiązania parametrów transakcji z argumentami programu wzorcowego.

Ideą programu jest wykonanie szeregu prostych transakcji, z których każda realizuje pewną liczbę odczytów współdzielonych struktur danych, ich zapisów i operacji na danych lokalnych wątku. Dodatkowo operacje na danych lokalnych wplatane są pomiędzy kolejne transakcje. Współdzielonymi strukturami danych są trzy tablice, `Array1`, `Array2` i `Array3`, które otrzymują odpowiednio nazwy: *hot*, *mild* i *cold*. Nazwy te dobrze oddają role poszczególnych struktur w przetwarzaniu współbieżnym (intensywność ich współbieżnego wykorzystania). Tablica *hot* jest współdzielona i współbieżnie wykorzystywana przez wszystkie wątki. Tablica *mild* jest podzielona na rozłączne części, po jednej dla każdego wątku i odczytywana oraz zapisywana transakcyjnie. Wreszcie tablica *cold*, dzielona podobnie jak *mild* posłuży do wykonywania operacji lokalnych z pominięciem TM.

Autorzy udowadniają, że przy pomocy wprowadzonej przez nich metodyki można ustalić charakterystykę wybranych programów z zestawu STAMP (*genome*, *vacation*, *labyrinth*, *ssca2*,

[8] to, że sposób, w jaki program wzorcowy obciąża system TM zależy, poza semantyką rozwiązywanego problemu algorytmicznego, od parametrów wejściowych (jeżeli tylko benchmark takie posiada) nie ulega wątpliwości. Ze względu na ten związek w dalszej części raportu terminem *workload* będzie określać się zarówno samą charakterystykę transakcyjną, jak i zestaw argumentów benchmarka, które ściśle wpływają na jej kształt.

Tablica 2.3: Parametry transakcji stanowiące ortogonalną charakterystykę transakcyjną programu wzorcowego. Za: [36].

Parametr	Definicja
współbieżność (ang. *concurrency*)	Liczba działających współbieżnie wątków.
wielkość *zbioru roboczego* (ang. *working-set size*)	Liczba lokalizacji w pamięci, do których program wzorcowy często się odwołuje.
długość transakcji (ang. *transaction length*)	(Średnia) liczba transakcyjnych odwołań do pamięci współdzielonej w ramach pojedynczej transakcji.
skażenie (ang. *pollution*)	Procent transakcyjnych odwołań do pamięci współdzielonej jaki stanowią zapisy.
czasowa lokalność odwołań (ang. *temporal locality*)	Prawdopodobieństwo z jakim transakcyjny dostęp do pamięci współdzielonej będzie dotyczył lokalizacji, do której transakcja odwoływała się już wcześniej.
poziom współzawodnictwa (ang. *contention*)	Prawdopodobieństwo, z jakim pojedyncza transakcja napotka konflikt.
przewaga (dostępów do pamięci współdzielonej) (ang. *predominance*)	Procent cykli rozkazowych procesora przeznaczonych na realizację dostępów do faktycznie współdzielonych struktur danych względem wszystkich cykli konsumowanych podczas wykonania benchmarka.
gęstość (ang. *density*)	Procent cykli rozkazowych procesora przeznaczonych na realizację dostępów do danych lokalnych wątków poza transakcją względem wszystkich cykli rozkazowych przeznaczonych na realizację dostępów do danych lokalnych.

Tablica 2.4: Wybrane elementy powiązania charakterystyki transakcyjnej mikrobenchmarka *EigenBench* z jego argumentami. Opracowanie własne na podstawie: [36].

Parametr	**Sposób wyliczenia wartości**
concurrency	N
transaction length	$R_1 + R_2 + W_1 + W_2 = T_{len}$
temporal locality	*lct*
working-set size	$A_1 + A_2 + A_3$
pollution	$\frac{W_1 + W_2}{T_{len}}$
contention	$1 - \left(1 - \min\left(1, \frac{(N-1)W_1'(1-lct)}{A_1}\right)\right)^{W_1'+R_1'}$

Gdzie:
N – liczba wątków
R_1, R_2 – liczba odczytów tablic `Array1` i `Array2` w ramach pojedynczej transakcji
W_1, W_2 – liczba zapisów tablic `Array1` i `Array2` w ramach pojedynczej transakcji
A_1, A_2, A_3 – rozmiary tablic `Array1`, `Array2` i `Array3`
R_1', W_1' – liczba unikatowych odczytów i zapisów tablicy `Array1` w ramach pojedynczej transakcji

```
1  static long A1, A2, A3 /* rozmiary tablic */, N /* liczba wątków */;
2  static long *Array1, *Array2, *Array3;
3
4  void init_arrays() {
5      Array1 = malloc(A1 * sizeof(long));
6      Array2 = malloc(A2 * N * sizeof(long));
7      Array3 = malloc(A3 * N * sizeof(long));
8  }
9
10 void test_core(tid /* identyfikator bieżącego wątku */ , loops, lct, R1, W1, R2,
           W2, R3_i, W3_i, Nop_i, k_i, R3_o, W3_o, Nop_o, k_o) {
11     long val = 0;
12     long total = W1 + W2 + R1 + R2;
13     for (int i=0; i<loops; i++) {
14         start();
15         (r1, r2, w1, w2) = (R1, R2, W1, W2);
16         {Wyczyść indeksy zapamiętane dla każdej z tablic.}
17
18         for (int j=0; j<total; j++) {
19             (action, array) = rand_action(r1, w1, r2, w2);
20             index = rand_index(tid, lct, array);
21             if (action == READ)
22                 val += read(array[index]);
23             else
24                 write(array[index], val);
25             if (j % k_i == 0) // by skalować liczbę op. lokalnych wewnątrz trans.
26                 val += local_ops(R3_i, W3_i, Nop_i, val, tid);
27         }
28         commit();
29         if (i % k_o == 0) // by skalować liczbę op. lokalnych na zewnątrz trans.
30             val += local_ops(R3_o, W3_o, Nop_o, val, tid);
31     }
32 }
33
34 (Action, Array) rand_action(r1, w1, r2, w2) {
35     {Z prawdopodobieństwem bazującym na wartościach r1, r2, w1 i w2 wylosuj
        operację spośród (READ, WRITE) i tablicę spośród (Array1, Array2). Zmniejsz
        o 1 wartość odpowiedniej zmiennej.}
36 }
37
38 long rand_index(tid, lct, array) {
39     {Z prawdopodobieństwem lct zwróć indeks zapamiętany dla tablicy array.
        W przeciwnym przypadku wylosuj indeks z zakresu 0..A1 (jeżeli array==Array1)
        lub tid*A2..(tid+1)*A2 (jeżeli array==Array2) i zapamiętaj wynik losowania
        dla tablicy array.}
40 }
41
42 long local_ops(R3, W3, NOP, val, tid) {
43     {Wykonaj R3 odczytów i W3 zapisów wartości w tablicy
        Array3[tid*A3..(tid+1)*A3] w losowej kolejności}
44     {Wykonaj NOP pustych operacji.}
45 }
```

Rysunek 2.25: Kod mikrobenchmarka *EigenBench*. Za: [36].

intruder) po to, by później odwzorować ją na tyle dobrze, żeby uzyskać w sposób sztuczny relację wydajności dwóch systemów TM (*SwissTM* i implementacji algorytmu TL2) zbliżoną do uzyskanej przy użyciu programów z grupy STAMP.

Bez wątpienia przy zastąpieniu pamięci współdzielonej pulą obiektów zlokalizowanych na różnych węzłach systemu rozproszonego i wprowadzeniu do *EigenBench* dodatkowego podziału dostępów transakcyjnych na lokalne i zdalne można by pokusić się o wykorzystanie tego narzędzia do ewaluacji D-STM. W tym raporcie jednak odstępuje się od takiego zadania podobnie jak od rozszerzenia zakresu analizy ortogonalnej (w tym: opracowania mapowania pomiędzy dodatkowymi argumentami mikrobenchmarka a parametrami transakcji roz-

proszonych). Poza pokazaniem, że w ogólności istnieje wykorzystująca je technika ewaluacji STM, mikrobenchmarki są w tym raporcie wprowadzane po to żeby, w miarę możliwości stały się podstawą dla bardziej rozbudowanych programów. *EigenBench* potrafi odtwarzać obciążenia TM generowane przez takie programy, jednak nie wygeneruje ich od podstaw. Stąd, stawiając sobie za cel realizm sposobu wykorzystania pamięci transakcyjnych przez proponowane programy wzorcowe, n nie będzie się już na nim skupiać.

2.6.3 *STMBench7*

Popularność benchmarka STAMP jako narzędzia do oceny wydajności pamięci transakcyjnych nie ulega wątpliwości. Wynikać to może choćby z faktu, że jego autorzy włożyli spory wysiłek w opracowanie takich programów wzorcowych, które, będąc reprezentatywnymi przykładami ze swoich dziedzin, mogą znacząco skorzystać na integracji z odpowiednio dobrymi mechanizmami sterowania współbieżnością. By dopełnić obraz metod wykorzystywanych do ewaluacji TM, należy jednak przedstawić jeszcze analogiczne podejście, wykorzystywane zanim STAMP został opracowany. Dobrym przykładem benchmarka dla systemów pamięci transakcyjnej, który próbuje odwzorować jej zastosowania w rzeczywistych aplikacjach jest *STMBench7* [27].

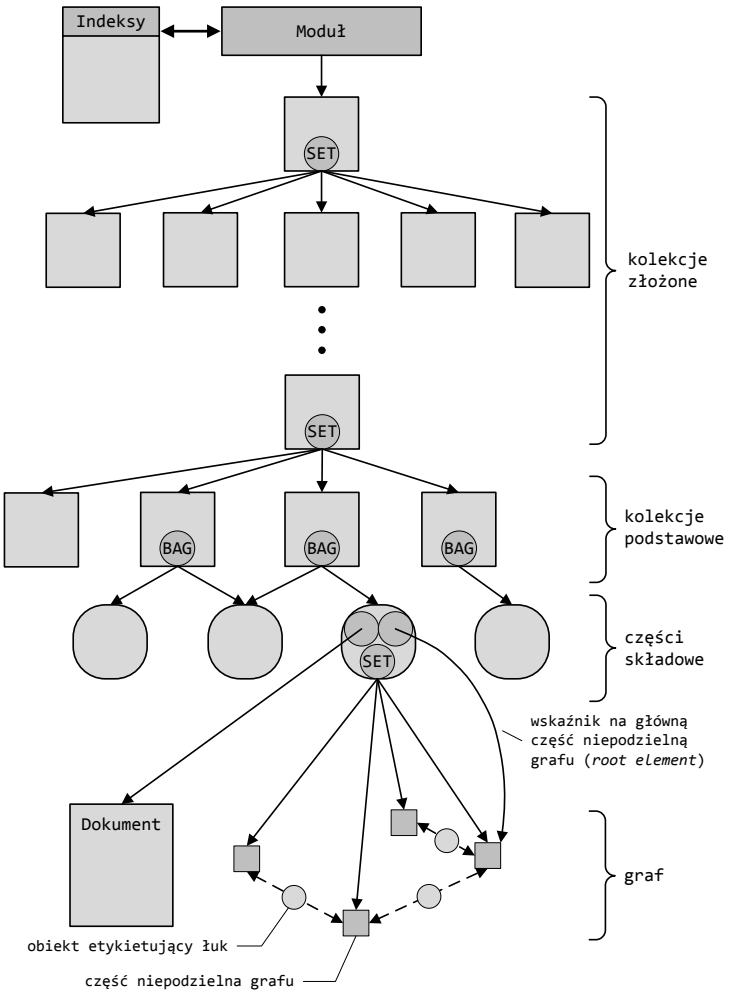
Koncepcją przyświecającą jego autorom było przystosowanie do współpracy z STM rozwiązania, które wykorzystywano dotychczas jako benchmark w innym obszarze informatyki. Możliwości takiej adaptacji dopatrzono się dla benchmarka OO7 przeznaczonego do oceny wydajności obiektowych baz danych.

Współdzielona struktura danych w *STMBench7* została przedstawiona na rysunku 2.26. Na najwyższym poziomie, obiektem, na którym operuje program, jest *moduł*. Poziom ten stanowi pozostałość po OO7, w którym dopuszczalne było istnienie kilku takich obiektów. Autorzy benchmarka dla STM redukują liczbę modułów do jednego tłumacząc to niepotrzebnym ograniczeniem poziomu współzawodnictwa (przez ograniczenie przestrzennej lokalności dostępów) gdy współbieżne operacje działają na wielu z nich. Moduł zawiera drzewo obiektów, z których te będące liśćmi nazywane są *kolekcjami podstawowymi* (ang. *base assembly*), a pozostałe – *kolekcjami złożonymi* (ang. *complex assembly*). Każdej kolekcji podstawowej przypisany jest szereg *części składowych* (ang. *composite parts*), z których każda przechowuje graf i wskazuje na pewien dokument. Graf składa się z *części niepodzielnych* (*atomic parts*) i połączeń, etykietowanych pewnymi obiektami.

Dodatkowo struktura danych wyposażona zostaje w szereg indeksów, pozwalających bezpośrednio, tj. bez jej przechodzenia (ang. *traversal*), odwoływać się np. do części niepodzielnych grafów, części składowych i kolekcji podstawowych oraz złożonych po identyfikatorze, a do dokumentów – po tytule.

Obiekty na każdym poziomie dysponują wskaźnikami na dzieci i rodziców, po to by umożliwić przechodzenie struktury w obu kierunkach.

Struktura taka, według autorów, w połączeniu z odpowiednim zestawem wykonywanych na niej operacji może reprezentować wiele aplikacji rzeczywistych, w tym CAD/CAM/CASE (czego dowodzą twórcy jej pierwowzoru – OO7) czy aplikacji serwerowych, obsługujących jednocześnie wielu użytkowników (serwery gier sieciowych, serwery proxy/cache, itp.).



Rysunek 2.26: Architektura współdzielonej struktury danych, wykorzystywanej w *STMBench7*. Opracowanie własne na podstawie: [27]

STMBench7 udostępnia cztery klasy operacji wykonywanych w ramach benchmarka:

- długie przejścia (ang. *long traversals*), w całości zapożyczone z OO7, operujące na wszystkich kolekcjach i/lub częściach niepodzielnych. Pewien odsetek tych operacji może wprowadzać modyfikacje w częściach niepodzielnych lub w dokumentach.
- krótkie przejścia (ang. *short traversals*), z których część pochodzi z OO7, – wędrówki przez strukturę losową ścieżką, rozpoczynającą się modułem, dokumentem lub częścią niepodzielną. Pewien ich fragment może wspomagać się indeksami. Do krótkich przejść zalicza się także specyficzną operację, która nie przechodzi po strukturze danych, lecz operuje na jej pojedynczym poziomie. Przegląda ona wszystkie kolekcje podstawowe i sprawdza spełnienie pewnego warunku dla niektórych spośród ich części składowych. W ogólności krótkie przejścia mogą wprowadzać modyfikacje w częściach niepodzielnych grafów lub w dokumentach.
- krótkie operacje (ang. *short operations*), również częściowo pochodzące z OO7, spro-

wadzają się do wybrania (losowo lub w oparciu o pewne kryterium; najczęściej przy pomocy indeksów) od jednego do kilku obiektów i wprowadzenia modyfikacji w nich samych lub w ich najbliższym otoczeniu.

- modyfikacje strukturalne (ang. *structure modifications*), wprowadzone specjalnie na potrzeby *STMBench7*. Ich zadaniem jest dodanie lub usunięcie szeregu obiektów (kolekcji podstawowych/złożonych), względnie modyfikacja połączeń między nimi, w sposób losowy. Zakres modyfikacji został ograniczony tak, by struktura nie uległa degeneracji (co miałoby miejsce np. w sytuacji gdy usunięcie wielu połączeń znacząco ogranicza możliwość wykonania krótkich przejść), ani by przesadnie się nie rozrosła.

Operacje te wykorzystywane są do skomponowania workloadów, których krótką charakterystykę zawiera tabela 2.5. Dalszej modyfikacji, poza częstością wykonywanych operacji może podlegać ich zestaw (pewne typy operacji mogą zostać całkowicie wyłączone) i liczba wątków biorących udział w przetwarzaniu.

Tablica 2.5: Domyślne parametry workloadów w *STMBench7*. Za: [27].

Kategoria	**Typ obciążenia**		
	Z przewagą odczytów	Zrównoważony	Z przewagą zapisów
Operacje odczytujące stan obiektów	90	60	10
Warianty operacji modyfikujące stan obiektów	10	40	90
długie przejścia	5		
krótkie przejścia	40		
krótkie operacje	45		
modyfikacje struktury	10		

By można było porównywać wydajność systemów pamięci transakcyjnych nie tylko między sobą, ale także i w odniesieniu do innych mechanizmów sterowania współbieżnością, *STMBench7* implementuje operacje także przy użyciu zamków drobnoziarnistych (dostarczając osobny zamek rozróżniający odczyty i zapisy dla każdego poziomu struktury; jest to strategia *medium-grained locking*, w przeciwieństwie do *fine-grained locking* i podejścia wykorzystującego „grubsze ziarno" blokowania (w tym konkretnym przypadku – pojedynczy zamek dla całej struktury danych, a więc *global lock*.

STMBench7 stanowi próbę wykorzystania TM w sposób zbliżony do jej potencjalnych rzeczywistych zastosowań. Nie jest to być może tak wyraźne, jak w przypadku STAMPa, choć autorzy (m.in. powołując się na analizę realizmu OO7 i wprowadzone do niego modyfikacje) twierdzą, że benchmark odtwarza wiele typowych schematów odwołań do współdzielonej struktury danych, wykorzystywanej w przetwarzaniu współbieżnym.

W kontekście tego raportu *STMBench7* nie będzie wykorzystywany. Przeciwności, jakie można napotkać przy próbie „rozproszenia" tego benchmarka prezentowane są w [6, punkt 2.2].

2.6.4 Wybrane mikrobenchmarki dla systemu *HyFlow*

HyFlow [54], framework pamięci transakcyjnej o budowie modułowej, dostarczający ponadto szereg implementacji modułów, w tym optymistyczną pamięć transakcyjną typu data-flow, bazującą na algorytmie TFA, optymistyczną pamięć transakcyjną typu control-flow (*Snake D-STM* [55]), szereg reguł arbitrażu i moduły odpowiedzialne za komunikację między węzłami, nie może obyć się bez odpowiednich narzędzi weryfikujących wydajność. Jak wspomniano wcześniej, najprawdopodobniej brak jest gotowych rozwiązań w zakresie pełnowymiarowych benchmarków dla rozproszonych pamięci transakcyjnych. Zapewne autorzy *HyFlow*, zdając sobie sprawę z pracochłonności wytworzenia takiego benchmarka od podstaw postanowili ograniczyć wysiłki wkładane w testy wydajnościowe do stworzenia 7 mikrobenchmarków, z których cztery zostaną tu przedstawione. Pierwszy z nich reprezentuje schemat powtarzający się w wielu mikrobenchmarkach dla różnych pamięci transakcyjnych i różnych języków programowania. Drugi demonstruje użyteczność podejścia control-flow do tworzenia rozproszonych aplikacji.

2.6.4.1 Benchmark *DHT*

Rozproszona tablica haszowa (ang. *distributed hash table, DHT*) stała się już swoistym wzorcem projektowym dla systemów rozproszonych. DHT przypomina w kwestii struktury organizacyjnej zwykłą tablicę haszową. Przechowywać w niej można pary *(klucz, wartość)*, przy czym rozmieszczenie par w węzłach wyznaczane jest przez skrót klucza.

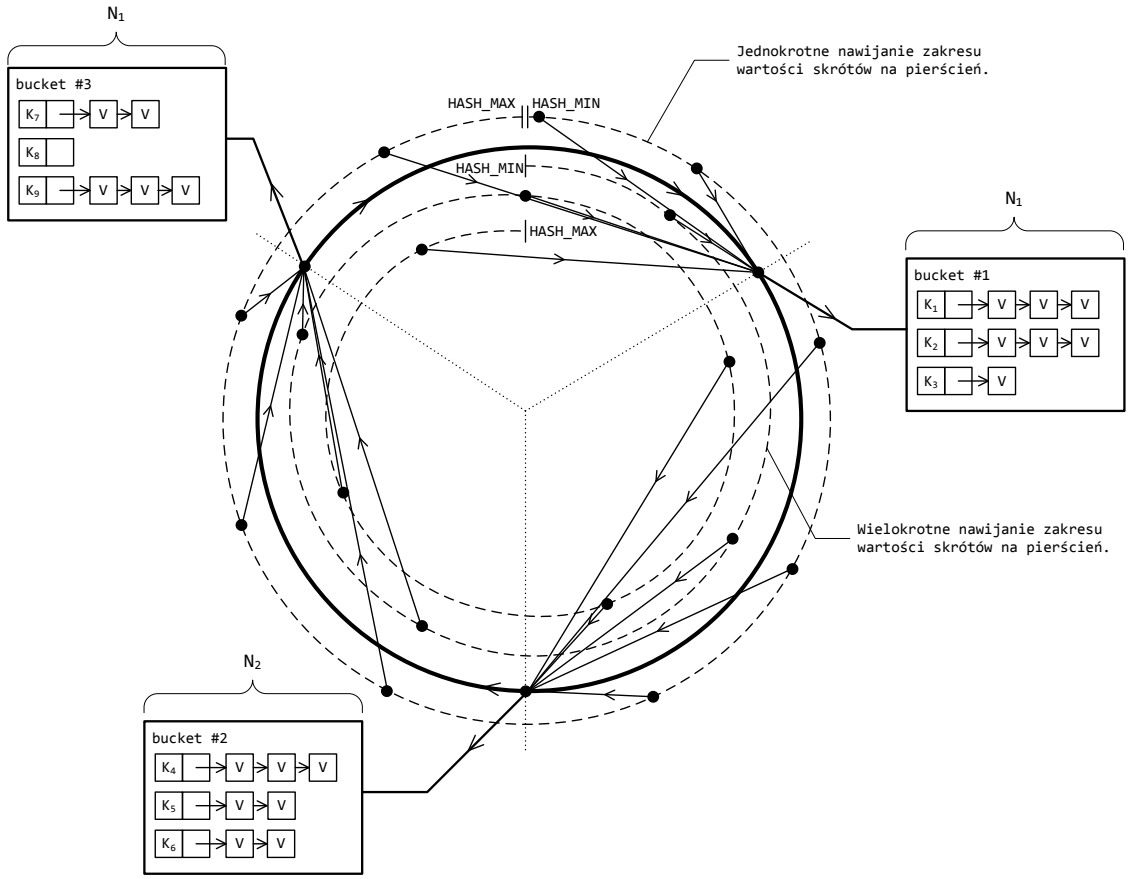
Przeciwdziedzina funkcji haszującej „nawijana” jest na pierścień, być może nawet wielokrotnie, formując spiralę. Na pierścieniu w mniej więcej równych odstępach rozmieszczone są węzły. Skróty, które zgodnie z ruchem wskazówek zegara znajdują się w pierścieniu przed węzłem (ale za poprzednim) wyznaczają (w odniesieniu do skrótu klucza) te pary, za przechowywanie których węzeł jest odpowiedzialny.

Generalnie rozproszona tablica haszowa udostępnia trzy operacje wysokopoziomowe:

- `V get(K key);` – zwraca wartość identyfikowaną przez sprecyzowany klucz jeżeli tylko taki klucz jest elementem pewnej pary znajdującej się w tablicy,
- `void put(K key, V value);` – wstawia parę `(klucz, wartość)` do tablicy. Zależnie od przyjętych koncepcji routingu żądań, klient tablicy haszowej (którym może być np. dowolny węzeł) udostępniający tę metodę programiście skontaktuje się z odpowiednim węzłem (wyznaczanym przez skrót klucza) i zleci mu przechowanie pary. Gdy para o takim samym kluczu już istnieje, zostanie nadpisana,
- `void remove(K key)` – usuwa z tablicy haszowej parę identyfikowaną przez podany klucz.

Istotą rozproszonej tablicy haszowej, poza byciem określonym wzorcem architektury systemu rozproszonego, jest uodpornienie konstrukcji na dołączanie, odłączanie się i awarie węzłów. Z racji złożoności problemu ten aspekt DHT nie będzie dalej rozpatrywany.

Benchmark DHT narzędzia *HyFlow* sprowadza się do udostępnienia dwóch typów transakcji wraz z możliwością określenia jaki ma być ich udział w przetwarzaniu. Transakcje odczy-



Rysunek 2.27: Przykładowa architektura rozproszonej tablicy haszowej.

tujące dane dokonują atomowo określonej liczby wywołań funkcji `get()`, podczas gdy transakcje modyfikujące dane dokonują atomowo szeregu zmian wartości skojarzonych z określonymi kluczami poprzez usunięcie z tablicy starej pary i wstawienie nowej.

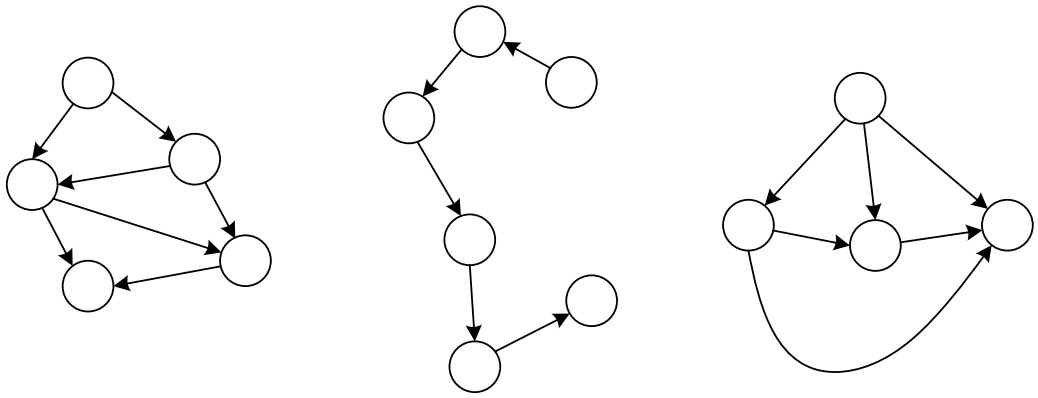
2.6.4.2 Benchmark *bank*

Benchmark *bank* jest podręcznikowym przykładem stosowania transakcji. Szereg kont bankowych (t-obiektów) jest rozmieszczony między węzłami, które pełnią w tym przypadku role oddziałów banku. Transakcje odczytujące dane ustalają sumaryczną wysokość kwot zdeponowanych na zadanej liczbie kont zlokalizowanych potencjalnie w różnych oddziałach, podczas gdy transakcje modyfikujące stan systemu dokonują pojedynczych przelewów bankowych.

Ponieważ liczba kont jest znana z góry, a na każdym początkowo znajduje się ta sama, ustalona wcześniej kwota, w dowolnej chwili wykonać można transakcję sprawdzającą spójność stanu systemu. Transakcja ta ustali sumę kwot zdeponowanych na wszystkich kontach ze wszystkich oddziałów banku i porówna wynik z oczekiwaną wartością.

2.6.4.3 Benchmark *loan*

Benchmark *loan* to program wzorcowy dla pamięci transakcyjnych typu control-flow. Symuluje on działanie systemu transferów pieniężnych, w którym posiadacze zasobów (zdalne t-obiekty) rozdystrybuowani są między węzły. Jednostka otrzymująca prośbę o udzielenie pożyczki próbuję ją zrealizować. Zazwyczaj jej własne środki nie będą w tym celu wystarczające i sama będzie musiała się zapożyczyć. Benchmark pozwala na określenie dwóch parametrów takiego zagnieżdżania pożyczek: `branching`, mówiący o tym u ilu jednostek sam odbiorca prośby może się zapożyczyć i `nesting`, mówiący jak głęboko zapożyczanie może się zagnieżdżać. Przykłady znaczenia tych parametrów przedstawiono na rysunku 2.28.



(a) `branching=2` i `nesting=5` **(b)** `branching=1` i `nesting=6` **(c)** `branching=3` i `nesting=4`

Rysunek 2.28: Przykładowe topologie zagnieżdżonych pożyczek. Za: [57].

Ponieważ *loan* przeznaczony jest dla pamięci typu control-flow, nie można w nim rozróżnić operacji odczytu i zapisu. Nie będą więc w nim istnieć transakcje jedyne odczytujące dane.

Sam benchmark może okazać się przydatny z racji tego, że wraz ze wzrostem wartości parametrów `branching` lub `nesting`, liczba t-obiektów zaangażowanych w wykonanie transakcji (a więc rozmiar access-setu) rośnie wykładniczo.

2.6.4.4 Benchmark *vacation*

Autorzy *HyFlow* wykorzystują benchmark zapożyczony z zestawu STAMP do oceny własnych rozwiązań. Choć STAMPowa wersja *vacation* nie jest nazywana mikrobenchmarkiem, pomimo tego, że implementowany przez nią algorytm pozwala na niemal dowolne sterowanie parametrami transakcji (czy to z racji faktu, że *vacation* jest elementem większej całości, czy z powodu tego, że użyte tam podejście wystarcza dla benchmarka testującego wieloprocesorowe TM), wersja rozproszona została sklasyfikowana jako mikrobenchmark dlatego, że nie wykorzystuje w sposób wyraźny rozproszonego charakteru systemu, w którym miałaby być uruchamiana.

Benchmark *vacation* implementuje prosty rozproszony system rezerwacji wykorzystywany przez biuro podróży. Zbiór rekordów tabeli zawierającej informacje o możliwych do zarezerwowania elementach (przelotach, samochodach do wynajęcia, pokojach hotelowych), po-

dobnie jak zbiór rekordów tabeli opisującej powiązania klienta z rezerwacjami i zbiór opisów samych rezerwacji, jest rozproszony. Jego elementy umieszczone zostają w węzłach systemu rozproszonego według pewnego stałego wzorca. Choć można doszukiwać się w takim podejściu analogii do poziomego partycjonowania danych, to jeżeli wiadomo, że baza danych ma funkcjonować w środowisku rozproszonym, znane są lepsze podejścia, które od początku biorą ten fakt pod uwagę (por. punkt 3.6.2 w tym raporcie lub też [20]).

Benchmark udostępnia cztery typy transakcji wykonywanych z losową częstością na obiektach współdzielonych w systemie rozproszonym:

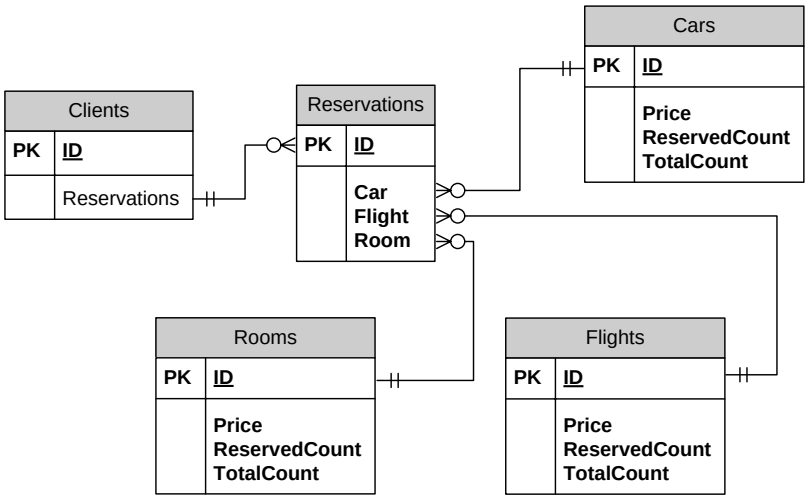
1. Dla klienta o identyfikatorze losowanym z określonego zakresu (od 0 do `query_range`) dokonywana jest rezerwacja stałej liczby (parametr `query_per_transaction`) losowo wybranych elementów różnych typów. Podczas rezerwacji dla każdego z trzech typów wybierany jest ten spośród wylosowanych elementów, który ma najniższą cenę. Jeżeli wreszcie wybrane w ten sposób elementy są dostępne (dla każdego typu istnieje co najmniej jeden, który nie został jeszcze zarezerwowany), rezerwacja jest dokonywana. W przeciwnym przypadku klient rezygnuje.
2. Klient może zostać usunięty z systemu. Ponieważ przypisanych może mu być wiele rezerwacji, z których każda składa się z trzech elementów, proces ich uwalniania objęty został transakcją.
3. Informacje o dostępnych elementach i ich cenach mogą zostać zmodyfikowane. Dla stałej liczby losowo wybranych elementów ((parametr `query_per_transaction`; maksymalną liczbę poszczególnych elementów każdego typu w systemie określa parametr `query_range`) wylosowane zostają nowe ceny i typy operacji. Te ostatnie przechowywane są w zmiennych logicznych, z których każda przyjmuje wartość `TRUE` z prawdopodobieństwem 50%. Modyfikacja każdego z elementów odbywa się w następujący sposób: jeżeli element nie istnieje w systemie, to jest tworzony i przypisuje mu się nową cenę; w przeciwnym wypadku, w zależności od typu operacji może on zostać usunięty, lub zmieniona może być jego cena.
4. Wprowadzona zostaje też długa transakcja odczytująca dane, której zadaniem będzie wyświetlenie wszystkich aktualnych rezerwacji znajdujących się w systemie.

2.6.5 Mikrobenchmarki dla *Atomic RMI*

Jeszcze skromniej w kwestii dostarczanych benchmarków prezentuje się *Atomic RMI*. Poza kilkoma mikrobenchmarkami zapożyczonymi od *HyFlow* (zapewne w celu wstępnego porównania obu systemów), i to na dodatek tylko tymi, w których da się przewidzieć zawartość access-setu przed rozpoczęciem transakcji, *Atomic RMI* dostarcza jeden program, który trudno jest nawet nazwać aplikacją wzorcową.

2.6.5.1 Benchmark *hotel*

Benchmark *hotel* jest ćwiczeniem dla studentów, mającym pokazywać podstawową wadę SVA i demonstrować sposób jej unikania. Jako ćwiczenie, nie doczekał się on nawet oficjalnej im-



Rysunek 2.29: Schemat bazy danych wykorzystywany przez mikrobenchmark *vacation* dla systemu *HyFlow*.

plementacji. Zadanie sprowadza się do zaimplementowania systemu rezerwacji pokoi hotelowych.

Dany jest hotel dysponujący `n` pokojami gościnnymi oraz `m` salami konferencyjnymi. Wyróżnić można też trzy typy klientów. Turysta rezerwuje jeden pokój, po czym po upływie określonego czasu zwalnia go. Organizator spotkania rezerwuje jedną salę konferencyjną, a po upływie pewnego czasu także ją zwalnia. Obaj, gdy w hotelu brak wolnych pomieszczeń odpowiedniego typu, rezygnują z rezerwacji. Dla odmiany organizator konferencji rezerwuje `k` sal i `l > k` pokoi. Po upływie określonego czasu zwalnia je, a gdy w hotelu nie ma dostatecznej liczby miejsc, odczekuje pewien czas i ponawia próbę.

Ideą zadania jest wskazanie możliwości powstania wąskiego gardła przy typowych rozwiązaniach postawionego problemu. Odwoływanie się przez każdą z transakcji rezerwujących pokoje do jakiejkolwiek formy recepcji powoduje podwyższenie poziomu współzawodnictwa w dostępie do niej, a ponieważ transakcje są kolejkowane, wolno działające węzły będą opóźniać przetwarzanie wszystkich innych (każda transakcja musi poczekać na zwolnienie recepcji przez potencjalnie wolno działającą poprzedniczkę). Proponuje się więc rozwiązanie, w którym transakcje sprawdzają dostępność pokoi bezpośrednio w sposób losowy lub wykorzystując uzgodniony wcześniej schemat dostępów.

Rozdział 3

Modelowanie systemu i wybór implementowanych rozwiązań

Niniejszy rozdział poświęcony będzie wszystkim zagadnieniom, które, nie reprezentując aktualnego stanu wiedzy w zakresie pamięci transakcyjnych (w tym: rozproszonych), nie wiążą się też ściśle z implementacją zaproponowanych programów wzorcowych, opisywaną w kolejnym rozdziale.

W pierwszej kolejności należy podkreślić szereg różnic pomiędzy transakcjami wykonywanymi na pamięci operacyjnej, a transakcjami bazodanowymi. Pewne fragmenty tego porównania zostały już przedstawione w punkcie 2.4.1, jednak dokładniejsza analiza jest niezbędna w przypadku, gdyby nie dało się zaadaptować na potrzeby D-STM żadnego z benchmarków dla pamięci transakcyjnych używanych w systemach wieloprocesorowych i przyszło do tworzenia benchmarka od podstaw. Należy też zaprezentować podstawowe możliwości i ograniczenia testowanych w tym raporcie pamięci transakcyjnych w zakresie ich interfejsów programistycznych.

Kolejnym krokiem jest wprowadzenie modelu rozproszonej pamięci transakcyjnej, ujmującego jej aspekty kluczowe dla wykonywania pomiarów wydajności. Istnieje narzędzie, które, stosując podejście podobne do proponowanego w oparciu o ten model daje możliwość testowania własnych implementacji pamięci transakcyjnej. Następnym punktem w tym rozdziale będzie więc przedstawienie tego narzędzia, jego użyteczności dla prowadzonych badań i własności utrudniających jego wykorzystanie.

Na koniec przedstawione i przedyskutowane zostaną wymagania stawiane kandydatom na programy wzorcowe dla rozproszonych pamięci transakcyjnych. Umożliwią one odniesienie rozwiązań współbieżnych przedstawionych w punkcie 2.6.1 (narzędzie STAMP) do cech środowisk rozproszonych.

Podsumowaniem rozdziału będzie wstępne sformułowanie propozycji w zakresie benchmarków dla D-STM.

3.1 Transakcje bazodanowe a transakcje na pamięci operacyjnej

Bez wątpienia jednym z hipotetycznych powodów, dla których pamięć transakcyjna może zyskać popularność jako narzędzie programowania współbieżnego jest wykorzystywanie przez nią powszechnie znanej wśród programistów abstrakcji. Transakcje są powszechnie wykorzystywane do interakcji aplikacji z bazami danych przy użyciu pewnego deklaratywnego języka służącego do specyfikowania modyfikacji oraz kryteriów odczytu/wyszukiwania danych (np. określony dialekt SQL) i interfejsu udostępnianego przez system zarządzania bazą danych (ang. *database management system, DBMS*) dla poszczególnych języków programowania (nazywanego *konektorem*), najczęściej w postaci API i implementującej je biblioteki kodu. Można zapewne pokusić się o stwierdzenie, że na wyższym poziomie abstrakcji, zdominowanym przez mapowanie obiektowo-relacyjne (ang. *object-relational mapping, ORM*) znaczenie transakcji przy interakcjach aplikacji z bazą danych będzie z perspektywy programistów mniejsze, jednak ORM nie jest mechanizmem na tyle uniwersalnym, by wyparł podejścia bazujące na konektorach (jest wolny, pamięciochłonny i zapewne niemożliwy do stosowania w językach nieobiektowych). Dopóki więc programiści mogą korzystać z transakcji bazodanowych istnieje ryzyko, że będą przez ich pryzmat postrzegać pamięć transakcyjną, a to z kolei wymaga podkreślenia różnic pomiędzy tymi dwiema abstrakcjami.

3.1.1 Zestaw własności ACID

ACID (*atomicity, consistency, isolation, durability*) to zestaw własności opisujących transakcje w systemach baz danych. Można je, choć nie bezpośrednio, odnieść do transakcji na pamięci operacyjnej lub zbiorze rozproszonych zasobów.

Niepodzielność, atomowość (ang. *atomicity*). Jest to własność mówiąca o tym, że wszystkie operacje składające się na transakcję zostaną wykonane, albo jeżeli nie jest to możliwe, żadna z nich nie pozostawi po sobie śladu. Nie będzie zatem dopuszczalna sytuacja, w której jedna z operacji składowych nie powiedzie się, a transakcja skutecznie zatwierdzi zmiany. Podobnie transakcja wycofująca wprowadzone zmiany nie może pozostawić po sobie śladu. W odniesieniu do baz danych [31] nazywa tę własność *failure atomicity*.

Spójność (ang. *consistency*). Znaczenie spójności stanu bazy danych jest zawsze definiowane przez aplikację (jest zależne od jej semantyki). Zazwyczaj sprowadza się to do określenia szeregu niezmienników (ang. *invariants*), których naruszenie może odbywać się tylko w wypadku i tylko na czas wprowadzania złożonych modyfikacji stanu. Rolą transakcji bazodanowej jest przeprowadzenie systemu z jednego stanu spójnego w kolejny.

Harris *et al.* w [31, punkt 1.3] odnoszą tak rozumianą spójność do pamięci transakcyjnych, zaznaczając, że ściśle łączy się ona z własnością *failure atomicity*. W oczywisty sposób stan systemu może utracić spójność w sytuacji, gdy w obliczu awarii transakcji część wprowadzonych przez nią zmian nie zostanie wycofana. Guerraoui i Kapałka rezygnują z uwzględnienia w definicji spójności stanu systemu, podając następującą intuicyjną interpretację tej własności dla pamięci transakcyjnych: historia wykonania (por. punkt 3.3.1) produkowana

z wykorzystaniem systemu TM jest spójna, gdy każdy odczyt t-obiektu `A` przez transakcję `t` zwraca albo wartość, którą transakcja `t` ostatnio do niego zapisała, albo wartość zapisaną do tam przez inną transakcję pod warunkiem, że ta już zatwierdziła wprowadzane przez siebie zmiany [26, str. 67].

Izolacja (ang. *isolation*). To, że dwie transakcje znajdują się względem siebie w izolacji oznaczać będzie, że żadna z nich nie jest w stanie dostrzec faktu, że druga aktualnie jest wykonywana. W przypadku baz danych będzie można mówić o różnych poziomach izolacji, od *read uncommitted*, pozwalającego transakcji odczytać zmiany wprowadzone, ale jeszcze nie zatwierdzone przez inną, poprzez *read committed*, pozwalającej transakcji odczytać stan niespójny, ale z modyfikacjami innych transakcji wprowadzanymi niepodzielnie, do *full isolation*, gwarantującego odczyt stanu systemu, który zawsze będzie spójny. W przypadku pamięci transakcyjnych izolacja będzie zazwyczaj rozumiana jako *full isolation*, przy czym w pewnych okolicznościach dopuszczalne stanie się odczytywanie niezatwierdzonych zmian pod warunkiem jednak, że taki odczyt skończy się ostatecznie wycofaniem transakcji czytającej (por. punkt 2.4.1, niespójne migawki).

Trwałość (ang. *durability*). Dla baz danych trwałość oznacza, że zmiany wykonane przez zatwierdzoną już transakcję nie zostaną wycofane, nawet w obliczu awarii. Stosowalność tego kryterium do pamięci transakcyjnych jest kwestią dyskusyjną [31]. Z jednej strony pisze się o tym, że ze względu na ulotny charakter nośnika informacji trudno jest mówić o jakiejkolwiek trwałości zmian wprowadzanych przez transakcje do pamięci operacyjnej. Z drugiej, trwałość zmian mogłaby przejawiać się w tym, że pozostaną one widoczne od momentu wprowadzenia do momentu zakończenia wykonania programu. Podejście takie wydaje się słuszne dla systemów wieloprocesorowych, ale też i dla rozproszonych, w których z racji różnorodności mechanizmów radzenia sobie z awariami (np. poprzez odtwarzanie stanu po jej wystąpieniu) problem znikających modyfikacji nie nie pozostaje bez znaczenia.

Własność *failure atomicity* pamięci transakcyjnych zostaje w [31, punkt 1.2.1] zastąpiona inną cechą. Autorzy argumentują wprowadzenie tej zmiany potrzebą rozszerzenia pojęcia niepodzielności na inne aspekty poza ryzykiem niepowodzenia wykonania pojedynczej operacji. „atomowość w obliczu błędów", mająca nikłe znaczenie dla równoległych (por. tabela 1.1), choć już nie rozproszonych, pamięci transakcyjnych (chyba, że za błąd uznać wystąpienie konfliktu przy dostępie), będzie teraz dodatkowo gwarantować niepodzielność wprowadzania modyfikacji względem współbieżnych odczytów. Powstanie kryterium *atomowego wykonania* transakcji (ang. *atomic execution*), równoważne własności przezroczystości implementacji TM, łączące niepodzielność z zapożyczoną z ACID najsilniejszą spośród gwarancji izolacji.

3.1.2 Deklaratywny a imperatywny charakter transakcji

Różnice (głównie w obszarze podatności systemu pamięci transakcyjnej na błędy) wynikające z objęcia transakcją sekwencji rozkazów, względem deklaratywnej specyfikacji zmian, opisywane były już wcześniej (por. punkt 2.4.1 (niespójność migawek), punkt 2.4.5 (błędy użycia wczesnego zwalniania)). Oczywiście istnieją rozszerzenia języków deklaratywnych pozwala-

Tablica 3.1: Własności ACID w odniesieniu do DBMS i TM.

Własność	**Transakcje bazodanowe**	**Transakcje na pamięci operacyjnej**
niepodzielność	*failure atomicity*, niepodzielność w obliczu awarii przytrafiającej się w trakcie wykonywania transakcji.	*failure atomicity* razem z niepodzielnością wprowadzania modyfikacji z perspektywy jedynie współbieżnych transakcji (*słaba atomowość*) lub wszystkich współbieżnych form odczytu (*silna atomowość*).
spójność	Znaczenie własności definiowane przez aplikację w odniesieniu do stanu systemu.	Gwarancja dotycząca obrazu stanu systemu, postrzeganego przez pojedynczą transakcję.
izolacja	Różne poziomy izolacji, od możliwości odczytywania niezatwierdzonych modyfikacji do gwarancji spójności całego odczytywanego obrazu.	Wymagana pełna izolacja lub ewentualnie odczytywanie zatwierdzanych atomowo zmian.
trwałość	Utrwalenie wyników zatwierdzonych transakcji w sposób odporny na awarie.	Dyskusyjne znaczenie dla pamięci transakcyjnych.

jące wprowadzać do baz danych modyfikacje w sposób proceduralny (np. PL/SQL w dialekcie *Oracle* czy *PostgreSQL*), jednak przede wszystkim pesymistyczny charakter metod sterowania współbieżnością stosowanych w systemach zarządzania bazami danych [31] pozwala uniknąć choćby anomalii spowodowanych niespójnością migawek.

Z perspektywy programisty imperatywny charakter transakcji wykonywanych na pamięci operacyjnej będzie miał znaczenie w dwóch głównych obszarach. Po pierwsze projektujący transakcję musi wiedzieć czy, i ewentualnie jakich, anomalii może spodziewać się w postrzeganym przez nią stanie systemu. Po drugie, musi pamiętać, że transakcje mogą zostać niejawnie powtórzone w następstwie rozwiązania konfliktu. Tę kwestię koniecznie należy uwzględnić przy projektowaniu sposobów interakcji transakcji ze światem zewnętrznym, zwłaszcza jeżeli wprowadza się wewnątrz niej nietransakcyjne modyfikacje współdzielonej lub lokalnej puli zasobów. Problem ten wydaje się prostszy do rozwiązania, gdy zrezygnujemy z instrumentacji kodu transakcyjnego wykonywanej automatycznie. Wtedy bowiem programista jest w stanie dostrzec wspomniane zagrożenie.

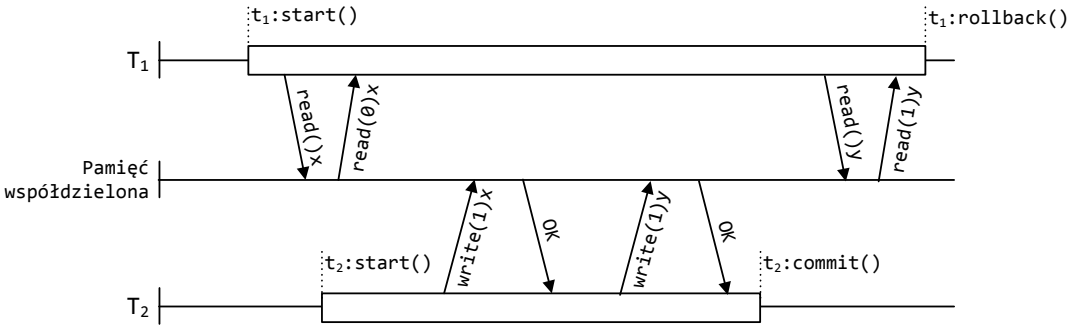
W punkcie 2.4.6 wspomniano także, że kryteria poprawności stosowane zazwyczaj w odniesieniu do transakcji bazodanowych nie są wystarczające w przypadku pamięci transakcyjnych. Ponieważ jedną z przyczyn takiego stanu rzeczy jest różnica pomiędzy charakterem transakcji, imperatywnym z jednej a deklaratywnym z drugiej strony, w tym punkcie przedstawiony zostanie przykład pokazujący, że zachowanie własność uszeregowalności, typowej dla problematyki transakcji bazodanowych, w połączeniu z własnością odzyskiwalności nie wystarczy, by określić, że wykonanie grupy transakcji zachodzi poprawnie.

Zachowanie przez grupę wykonanych transakcji własności uszeregowalności oznacza, że

da się je ułożyć w takiej kolejności, że gdyby według niej zostały wykonane sekwencyjnie (nie nachodząc na siebie w czasie), wpłynęłyby na stan bazy danych w taki sam sposób, w jaki zrobiły to faktycznie.

Zachowanie własności odzyskiwalności oznacza, że żadna spośród zatwierdzonych transakcji nie odczytała zmian, które później zostały wycofane.

Sytuację, w której obie te własności są zachowane przedstawia rysunek 3.1. Wpierw transakcja t_1 odczytuje z t-obiektu x wartość 0. Następnie transakcja t_2 zapisuje wartość 1 zarówno do t-obiektu x, jak i y. Wreszcie t_1 odczytuje z y wartość 1 i zostaje wycofana.



Rysunek 3.1: Wykonanie dwóch transakcji, zachowujące własność *serializability* i *recoverability*, ale nie zachowujące *opacity*. Za [25, prezentacja *PPoPP*].

Własności: uszeregowalność i odzyskiwalność zostają zachowane. Istnieje bowiem takie sekwencyjne ułożenie transakcji, które w efekcie zmienia wartości zapamiętaną w t-obiektach x i y z (0, 0) na (1, 1): wpierw wycofana transakcja t_1, potem t_2. Ponieważ t_2, jedyna wycofana transakcja nie wprowadza zmian, nie ma możliwości by własność odzyskiwalności została naruszona. Mimo to transakcja t_1 postrzega niespójną migawkę i może tym samym np. wpaść w pętlę nieskończoną zanim w ogóle skompletuje zmiany przeznaczone do zatwierdzenia.

Dla wykonania transakcji t_1 i t_2 w sposób oczywisty nie jest zachowana własność przezroczystości, co wynika wprost z trzeciej własności składowej, opisanej w punkcie 2.4.6.

3.1.3 Amortyzacja kosztów synchronizacji dostępu

Koszty wykonania transakcji, choć w ogólności istotne w systemach pamięci transakcyjnej, będą zapewne w pierwszej kolejności interesować projektantów algorytmów pamięci transakcyjnej, a dopiero później jej użytkowników. Bez wątpienia jednak wszystkie systemy, w których wydajność przetwarzania odgrywa kluczową rolę będą podatne na występowanie tradeoffów między wydajnością właśnie a prostotą użytkowania. W sytuacji gdy narzuty związane z przetwarzaniem transakcyjnym trzeba zredukować, będzie to możliwe zapewne w zamian za pozbawienie transakcji pewnych gwarancji, co już może bezpośrednio dotyczyć programistów. Dla przykładu Harris i Fraser nazwą transakcje w systemie WSTM „lekkimi", zapewne z racji tego, że WSTM nie gwarantuje operowania na spójnym stanie pamięci. Być może będzie też istnieć możliwość (a czasem i potrzeba) takiego projektowania transakcji, by niwelować wady algorytmu TM. W kontekście *Atomic RMI* wspominano o możliwości wczesnego

zwalniania obiektów na żądanie, poprzez symulację większej liczby dostępów podczas realizacji jednego (por. punkt 2.3.4). Takie „wymuszone" zwolnienie t-obiektu wiąże się więc z realizacją dostępu, a zatem potencjalnie także i z (kosztownym czasowo) przesłaniem komunikatu przez sieć (jeżeli tylko t-obiekt jest obiektem zdalnym). Formą optymalizacji wykonywanej przez projektanta transakcji – programistę może więc być tutaj możliwie najdokładniejsze specyfikowanie wartości supremum liczby dostępów wszędzie tam gdzie tylko jest to możliwe.

Podstawową różnicą między pamięciami transakcyjnymi a systemami zarządzania bazą danych w kontekście przetwarzania transakcyjnego jest to, że w bazach danych informacje przechowywane są zazwyczaj na dysku twardym. Czas dostępu do danych jest tam zdecydowanie większy niż czas dostępu do pamięci operacyjnej. W efekcie systemy zarządzania bazą danych mogą sobie pozwolić na implementację znacznie bardziej kosztownych czasowo (bowiem współbieżnych z oczekiwaniem na dostęp do dysku) rozwiązań w zakresie synchronizacji dostępu niż systemy pamięci transakcyjnej. Z uwagi na rozmiar i powszechność transakcji bazodanowych (tylko silna atomowość), a także wymóg zapewnienia współbieżności dostępów (bazy danych ze swojej natury, wykorzystywane współbieżnie przez wielu użytkowników) w tej roli nie będzie mógł występować globalny zamek współdzielony. Można już jednak sobie wyobrazić podejście bazujące na (kosztownym) wykrywaniu zakleszczeń, co w praktyce będzie wykorzystywane, nawet w systemach zarządzania rozproszoną bazą danych (por. podejścia do wykrywania rozproszonego zakleszczenia: *path-pushing*, wykorzystywane np. przez algorytm Obermarcka [48] i *edge-chasing*, wykorzystywane przez algorytm Chandyego-Misry-Haasa [15]).

3.2 Model programistyczny systemów pamięci transakcyjnej

By rozpocząć dyskusję o programach wzorcowych dla dwóch systemów D-STM pojawiających się w tym raporcie, *HyFlow* i *Atomic RMI*, należy wpierw pokazać jakie możliwości pamięci te oferują programistom. W tym celu nie wystarczy przedstawić interfejsu programistycznego, bowiem ten, czy to rzeczywisty, czy modelowy (pod postacią wprowadzonego już wcześniej zbioru procedur `start()`, `read()`/`write()`/`access()`, `commit()` i `abort()`/`retry()`) stanowi tylko wycinek funkcjonalności rozproszonego systemu pamięci transakcyjnej. Omawiany zaś w tym punkcie model programistyczny, czy też model programowania (ang. *programming model*) przedstawia programiście spójną i kompletną abstrakcję pamięci transakcyjnej, włączając w to kwestie związane z instrumentacją kodu czy deklaratywnymi konstrukcjami językowymi pozwalającymi definiować transakcje.

Elementami, które nie są ujęte w API, a stanowią część modelu programistycznego mogą być m.in. następujące kwestie:

- jak (co programista ma zrobić by) utworzyć nowy t-obiekt? czy jest to możliwe do wykonania dynamicznie (np. z wewnątrz transakcji)?
- jak (co programista ma zrobić by) uzyskać namiastkę zdalnego t-obiektu?
- w jaki sposób t-obiekty są identyfikowane?
- czy (i jak) można opisać transakcję deklaratywnie? Czy system pamięci transakcyjnej

udostępnia konstrukcje językowe umożliwiające definiowane sekcji atomowych?

- czy da się i w jaki sposób wymusić wycofanie transakcji (wykonać *non-forcible rollback*)?
- jaka jest semantyka powtórzenia transakcji (czy trzeba je zlecić ręcznie po wykryciu konfliktu, a jeżeli nie to które z modyfikacji są automatycznie wycofywane)?
- w jaki sposób D-STM informuje o awarii zdalnego t-obiektu?

W zakresie pamięci transakcyjnych dla systemów wieloprocesorowych modele programistyczne są na tyle podobne do siebie, że autorzy STAMPa (por. punkt 2.6.1) mogli pozwolić sobie na wprowadzenie jednego interfejsu w celu integracji ośmiu aplikacji wzorcowych z symulatorem pamięci transakcyjnych i sześcioma systemami rzeczywistymi.

Wspólne elementy modeli programistycznych rozpoznawane przez STAMP to zastępowane w programach wzorcowych makropoleceniami preprocesora języka C:

- wszelkie operacje wykonywane przed rozpoczęciem i po zakończeniu wykonania zrównoleglonej części programu (*TM startup*, *TM shutdown*),
- wszelkie modyfikacje pozycji na listach parametrów formalnych i aktualnych funkcji wywoływanych wewnątrz współbieżnych wątków, w tym – wewnątrz transakcji
- wszelkie operacje wykonywane przy okazji tworzenia nowego i zakończenia wykonywania istniejącego wątku (*TM thread enter*, *TM thread exit*)
- operacje utworzenia nowych t-obiektów – alokowania pamięci tak wewnątrz wątku, jak i wewnątrz transakcji (*alloc*, *TM alloc*) oraz analogiczne operacje zwolnienia pamięci (*free*, *TM free*)
- operacje rozpoczynania transakcji z wyróżnieniem transakcji jedynie odczytujących współdzielone dane (*TM start*, *TM start RO*) jej ponawiania i zatwierdzania (*TM restart*, *TM end*)
- operacje transakcyjnego odczytu oraz zapisu danych współdzielonych, w tym tych sprywatyzowanych (*TM shared/local write*, *TM shared read*) z wyróżnieniem operacji na wskaźnikach,
- operacja wczesnego wyłączania obiektu z read-setu (*TM early release*).

Dla rozproszonych pamięci transakcyjnych trudno będzie o podobny uniwersalizm, zwłaszcza gdy jeden model ma obejmować pamięci optymistyczne i pesymistyczne, a co za tym idzie – dynamiczne i statyczne. Stąd oba systemy omawiane wtym raporcie wymagają osobnych opisów sposobów, w jakie programiści aplikacji rozproszonych mogą z nich korzystać.

3.2.1 Model programistyczny *HyFlow*

Model programistyczny *HyFlow*, zarówno dla pamięci transakcyjnej typu data-flow (implementującej *Transaction Forwarding Algorithm*), jak i control-flow (*Snake D-STM*) opisany jest w [53].

Podstawowym elementem systemu *HyFlow*, poza implementacją algorytmu pamięci transakcyjnej jest lokalizator obiektów (ang. *locator*). Rola tego komponentu to utrzymywanie

i udostępnianie informacji o powiązaniach obiektów z węzłami, które są ich właścicielami. Sposób implementacji tego rozwiązania nie jest w tym momencie ważny. Lokalizatorem może być na przykład rozproszony katalog obiektów. Dla programisty ważne jest to, jaki interfejs ma do dyspozycji w zakresie przypisywania obiektom właścicieli i jaka jest semantyka poszczególnych operacji wchodzących w jego skład.

Lokalizator w *HyFlow* w ramach swojego interfejsu udostępnia dwie operacje: `lookup(id)` oraz `register(id, object)`. Pierwsza z nich pozwala wyszukać obiekt o podanym identyfikatorze w systemie i, w zależności od przyjętego podejścia uzyskać jego namiastkę (dla control-flow) albo lokalną kopię (dla data-flow). Choć ani architektura *HyFlow* [53, punkt 3.3] ani sposób instrumentacji kodu transakcyjnego [53, punkt 3.3.1] nie będą tu omawiane, wprowadzenie interfejsu lokalizatora będzie potrzebne choćby do płytkiego wglądu w implementację modelu programistycznego.

HyFlow dostarcza rozproszone transakcje dla języka Java. Jego model programistyczny w dużej mierze będzie bazował na adnotacjach wprowadzonych w Java 1.5. Adnotacje to pewne dodatkowe informacje, zazwyczaj udostępniane programowi w trakcie wykonania, pozwalające oznaczyć, a także i przekazać pewne parametry m.in. klasom/obiektom, polom, metodom i zmiennym lokalnym.

By utworzyć t-obiekt, programista musi skonstruować odpowiadający mu obiekt za pomocą operatora `new`. Mechanizm instrumentacji zadba o to, by konstruktor obiektu rejestrował go w lokalizatorze. Oczywiście sam obiekt musi wcześniej zostać odpowiednio przystosowany do swojej roli. W pierwszej kolejności ma implementować interfejs wymuszający na nim posiadanie identyfikatora. Dalej powinien zostać oznaczony odpowiednią adnotacją, tak by silnik instrumentacji wiedział, że obiekt będzie podlegał zdalnym dostępom w modelu control-flow i mógł odpowiednio przetworzyć udostępnianie innym metody.

Dla modelu data-flow nie są wymagane żadne specyficzne akcje w odniesieniu do pól, ich getteów czy setterów, jednak dla modelu control-flow należy dodatkowo oznaczyć odpowiednią adnotacją wszystkie te metody, które mogą być wywoływane zdalnie.

Podstawą dla sekcji atomowych są w *HyFlow* metody opatrzone odpowiednimi adnotacjami. Metoda oznaczona adnotacją `@Atomic`, mogącą ponadto definiować maksymalną liczbę powtórzeń i limit czasowy wykonania transakcji, zawierać będzie kod transakcyjny. Stąd może zostać niejawnie wykonana wielokrotnie, w zamian za gwarancję niepodzielności takiego wykonania. Jedynie dostępy do zdalnych i lokalnych t-obiektów są zastępowanie odpowiadającymi im procedurami transakcyjnymi `read()` i `write()`. Metoda atomowa nie powinna więc m.in. wprowadzać zmian do stanu obiektu (np. jego pól) którego jest częścią, chyba że sam obiekt jest wykorzystywany w roli t-obiektu w transakcji.

Pozyskiwanie t-obiektów wewnątrz transakcji odbywa się poprzez ich „otwarcie”. Lokalizator udostępnia wysokopoziomową procedurę `open(id, mode)`, która może odwoływać się do `lookup(id)` i w połączeniu z pewnym modułem dostępu do obiektów zwracać namiastkę lub lokalną kopię. *HyFlow* rozróżnia trzy tryby otwierania t-obiektów: `read` – tylko do odczytu, `write` – do zapisu (i odczytu) oraz `shared` – do odczytu i zapisu, jednak nie powodujący umieszczenia t-obiektu ani w read- ani we write-secie transakcji.

HyFlow udostępnia pamięć transakcyjną w formie bloków atomowych przy wykorzystaniu instrumentacji. Stąd programista nie będzie miał (przynajmniej wprost) dostępu do procedur

wycofywania transakcji na żądanie (`abort()`) i jej ponawiania (`retry()`). W kwestii powtórzeń wykonania programista może mieć wpływ na wybór reguły arbitrażu wykorzystywanej do rozwiązywania konfliktów między transakcjami. *HyFlow* implementuje m.in. następujące reguły arbitrażu (opisane w [31, punkt 2.3.3]):

- `passive` (na potrzeby HyFlow nazywana default) – transakcja, która napotka konflikt (tj. wykona jako druga operację konfliktową) wycofuje się i rozpoczyna ponowne wykonanie, może też odczekiwać losowy czas przed ponowieniem (mechanizm *backoff*),
- `aggressive` – transakcja, która napotka konflikt zawsze wymusza wycofanie drugiej (zwanej ofiarą[1] – ang. *victim*),
- `polite` – transakcja napotykająca konflikt oczekuje zgodnie z regułą *exponential backoff* przed wycofaniem ofiary. Po zakończeniu każdego oczekiwania sprawdza się czy konflikt nadal występuje,
- `karma` – podstawą do rozwiązania konfliktu są tutaj priorytety odpowiadające liczbie t-obiektów wykorzystywanych przez transakcję (sumarycznie dla wszystkich jej powtórzeń); transakcja napotykająca konflikt, posiadająca wyższy priorytet natychmiast wycofuje ofiarę. Za to gdy posiada niższy priorytet, wykonuje `n` cykli odczekanie-ponowny dostęp do zasobu, gdzie `n` oznacza różnicę w priorytetach obu transakcji[2],
- `timestamp` – arbitraż wygrywa ta transakcja, która rozpoczęła wykonanie jako pierwsza według zegara czasu rzeczywistego (lub dowolnego innego zegara globalnego). Oczywiście w systemie rozproszonym bezpośrednie porównywanie czasu różnych węzłów jest niepraktyczne jeżeli nie bezsensowne. *HyFlow* przypisuje jednak każdemu węzłowi licznik `lc`, który zapewne może pełnić rolę przybliżenia czasu rzeczywistego w ustaleniu relacji pomiędzy transakcjami,
- `kindergarten` – każda transakcja utrzymuje listę innych transakcji, z którymi przegrała arbitraż. Jeżeli transakcja napotykająca konflikt posiada ofiarę na swojej liście, natychmiast ją wycofuje. Jeżeli zaś nie – sama odczekuje pewien (stały) okres i wycofuje swoje zmiany. Ma to zapewnić naprzemienność transakcji w dostępie do t-obiektu powodującego konflikt.

Przykład kodu, jaki programista musi dostarczyć, by zaimplemetnować węzeł listy jednokierunkowej przedstawiony został na rysunku 3.2. Modyfikacje powiązań węzła z innymi elementami listy oraz modyfikacje wartości węzła realizowane są przez metody, które można wywoływać zdalnie. Rysunek 3.3 przedstawia implementację podstawowych operacji wykonywanych na liście linkowanej. Warto dodać, że ze względu na własność komponowalności[3]

[1] w zasadzie ofiarą jest ta transakcja, która przegrywa arbitraż. Jednak dla czytelności opisu reguły arbitrażu będą rozpatrywane z perspektywy drugiej transakcji, napotykającej konflikt, podczas gdy pierwsza weń zaangażowana zawsze będzie nazywana ofiarą, choć w istocie jest nią tylko potencjalnie.

[2] o cyklach odczekiwania i ponownego dostępu pisze Harris i inni w [31]. Wprowadzają oni taki mechanizm zapewne z racji tego, że w ogólności reguły arbitrażu mogą doprowadzić do opóźnienia dostępu zamiast do wycofania jednej z transakcji (zwłaszcza dla wczesnego zarządzania wersjami). W kodzie źródłowym implementacji reguły karma w *HyFlow* brak analogicznego elementu. Jeżeli ofiara operowała dotychczas sumarycznie na mniejszej liczbie obiektów, jest po prostu wycofywana. W przeciwnym wypadku to ofiara prędzej czy później wycofa aktualną transakcję.

[3] własność komponowalności pozwala wykorzystywać jedne abstrakcje do budowy innych, bardziej złożonych.

(ang. *composability*) operacje te, objęte transakcjami mogą zostać użyte do budowy bardziej złożonych operacji (np. sortowania listy), również objętych transakcją, co nie byłoby możliwe np. dla implementacji operacji podstawowych z użyciem zamków.

```
1  /* Węzeł listy jednokierunkowej. */
2  public class Node implements IDistinguishable {
3      private String id; // na potrzeby implementacji IDistinguishable
4      private Integer value;
5      private String nextId;
6
7      public Node(String id, Integer value) {
8          this.id = id;
9          {...}    // w tym miejscu moduł instrumentacji kodu doda polecenie
10                  // umieszczenia obiektu w lokalizatorze
11     }
12
13     @Remote
14     public void setNext(String nextId) { // metoda możliwa do wywołania
15         this.nextId = nextId;            // zdalnie w modelu control-flow
16     }
17
18     @Remote
19     public String getNext() {            // metoda możliwa do wywołania
20         return nextId;                   // zdalnie w modelu control-flow
21     }
22
23     @Remote
24     public Integer getValue() {          // metoda możliwa do wywołania
25         return value;                    // zdalniew modelu control-flow
26     }
27
28     @Override
29     public Object getId() { // implementacja metody interfejsu
30         return id;          // IDistinguishable
31     }
32 }
```

Rysunek 3.2: Implementacja t-obiektu w *HyFlow*. Opracowanie własne na podstawie: [53, rysunek 3.1].

3.2.2 Model programistyczny *Atomic RMI*

Atomic RMI ściśle bazuje na mechanizmie zdalnego wywoływania metod Java RMI. Stąd czynności takie jak tworzenie czy dostęp do t-obiektu będą stanowiły rozszerzenia analogicznych operacji wykonywanych przy użyciu *Remote Method Invocation* (RMI).

By przygotować obiekt do pełnienia roli t-obiektu należy umożliwić wywoływanie wybranych spośród jego metod zdalnie. W RMI sprowadzałoby się to do zaprojektowania interfejsu obiektu uwzględniającego zdalne metody oraz dostarczenia ich implementacji. Obiekt implementujący byłby potem eksportowany (tworzony byłby serwer RMI), a jego namiastka umieszczana w rejestrze.

Te mogą z kolei posłużyć do budowy jeszcze bardziej złożonych. Jeżeli implementacja sekcji atomowych bazowałaby na zamkach drobnoziarnistych, to złączenie dwóch mniejszych sekcji w jedną, większą mogłoby z jednej strony doprowadzić do zakleszczenia, z drugiej zaś naruszyć niepodzielność wykonania (zakładając, że nie wyznaczamy w sposób automatyczny tych z zamków zamykanych zazwyczaj przez wewnętrzne sekcje, które należy zamknąć przy rozpoczęciu wykonania zewnętrznej). Pamięci transakcyjne natomiast pozwalają na taki sposób komponowania bloków atomowych. Temat zagnieżdżania jednych transakcji w innych nie będzie jednak analizowany wtym raporcie.

```
1  /* "wrapper" listy jednokierunkowej. */
2  public class List {
3      String final HEAD = ... ; // identyfikator głowy listy
4
5      @Atomic
6      public static void add(Integer value) {
7          Locator locator = HyFlow.getLocator();
8          Node head = (Node) locator.open(HEAD);
9          String oldNext = head.getNext();
10         String newNodeId = newRandomUniqueString();
11         Node newNode = new Node(newNodeId, value);
12         newNode.setNext(oldNext);
13         head.setNext(newNodeId);
14     }
15
16     @Atomic
17     public boolean delete(Integer value) {
18         Locator locator = HyFlow.getLocator();
19         String next = HEAD;
20         String prev = null;
21         do { // wyszukiwanie usuwanego węzła
22             Node node = locator.open(next, "r"); // otwarcie węzła do
23                                                  // odczytu
24             if (value.equals(node.getValue())) {
25                 Node deletedNode = locator.open(next); // ponowne otwarcie
26                                                        // do zapisu
27                 Node prevNode = locator.open(prev); // ponowne otwarcie
28                                                     // do zapisu
29                 prevNode.setNext(deletedNode.getNext());
30                 locator.delete(deletedNode);
31                 return true; // znaleziono i usunięto węzeł
32             }
33             prev = next;
34             next = node.getNext();
35         } while (next != null);
36         return false;
37     }
38
39     @Atomic
40     public boolean contains(Integer value) {
41         Locator locator = HyFlow.getLocator();
42         String next = HEAD;
43         do { // przeszukiwanie listy
44             Node node = locator.open(next, "r"); // otwarcie węzła do
45                                                  // odczytu
46             if (value.equals(node.getValue())) {
47                 return true; // znaleziono odpowiedni węzeł
48             }
49             next = node.getNext();
50         } while (next != null);
51         return false; // nie odnaleziono węzła
52     }
53 }
```

Rysunek 3.3: Konstruowanie transakcji w *HyFlow*. Opracowanie własne na podstawie: [53, rysunek 3.2].

Atomic RMI wymaga uzupełnienia implementacji obiektu o pewne elementy, jak choćby liczniki `gv`, `lv` i `ltv` oraz mechanizm tworzenia kopii zapasowych. Ich dodanie realizowane jest przez wymuszenie tego, by implementacja obiektu zdalnego dziedziczyła po odpowiedniej klasie. Jej instancja w efekcie pełni rolę pośrednika opisanego w punkcie 2.3.4. By uczynić obiekt dostępnym w systemie rozproszonym, pozostaje potem tylko skojarzyć namiastkę uzyskaną podczas eksportu z nazwą w rejestrze RMI – w niezmieniony sposób.

Proces pozyskiwania t-obiektów niczym nie różni się od pozyskiwania zwykłych namiastek. Oczywiście wszystkie elementy access-setu muszą być znane przed rozpoczęciem transakcji, nie ma więc mowy o tym, by pozyskiwać t-obiekty wewnątrz niej.

Pewne liczniki będą musiały być dodane do namiastki w momencie utworzenia na jej pod-

stawie t-obiektu. Będą to: `pv`, `rv`, `cc`. W momencie deklarowania użycia obiektu zdalnego przez transakcję, w węźle przechowującym jego implementację tworzony jest pośrednik transakcyjny (t-proxy) tak, jak to pokazano w punkcie 2.3.4. Jest on niczym innym jak t-obiektem istniejącym istniejącym tak długo jak transakcja, która go utworzyła, tyle, że przechowywanym zdalnie.

Należy zaznaczyć, że w przeciwieństwie do *HyFlow*, w którym implementacja specjalnego interfejsu zmusza współdzielone obiekty do tego, by same przechowywały swoje identyfikatory, w *Atomic RMI* jedynym miejscem, z którego można uzyskać informacje o identyfikatorach obiektów zdalnych jest rejestr RMI.

Konstrukcja transakcji sprowadza się do jej utworzenia w oparciu o wskazany rejestr RMI, zdefiniowania access-setu razem z supremum liczby dostępów dla każdego umieszczonego w nim obiektu i rozpoczęcia wykonania. Wszystkie te kroki są tożsame z wywołaniami odpowiednich metod API *Atomic RMI*. Alternatywnie można przekazać procedurze uruchamiającej transakcję obiekt (anonimowy) zawierający kod transakcyjny w postaci metody atomowej. Bez względu na sposób wskazania poleceń do niepodzielnego wykonania, transakcję można jawnie zatwierdzić lub wycofać, a w przypadku obiektu anonimowego – także ponowić i pozostawić niezakończoną (zadanie uruchomienia procedury zatwierdzania spadnie wtedy na system pamięci transakcyjnej).

Gdyby okazało się, że zdalny t-obiekt uległ awarii, próba wywołania jego metody zakończy się wyrzuceniem wyjątku. Pozwoli to programiście zdecydować czy należy próbować kompensować taką awarię, czy może należy w całości wycofać transakcję. Gdyby zaś okazało się, że poprzednia transakcja wcześnie zwolniła element access-setu obecnej, a następnie wycofała wprowadzane przez siebie zmiany, wyjątek będzie w obecnej wyrzucany przy okazji wywołania dowolnej zdalnej metody lub przy próbie zatwierdzenia zmian.

Z racji specyficznej konstrukcji algorytmu pewne operacje transakcyjne mogą zajmować znacznie więcej czasu niż inne. W szczególności pierwsze zdalne dostępy do każdego z t-obiektów mogą zostać wstrzymane do momentu zwolnienia go przez poprzedniczkę, a operacja zatwierdzenia zmian będzie musiała poczekać aż poprzedniczka zdecyduje się ostatecznie zatwierdzić (lub wycofać) swoje zmiany.

Rysunek 3.4 oraz 3.5 przedstawiają przykład wykorzystania *Atomic RMI* podobny do przykładu dla *HyFlow* z punktu 3.2.1. Ponieważ lista linkowana jest strukturą danych szczególnie trudną w użyciu podczas stosowania statycznych pamięci transakcyjnych, przykład będzie dotyczył tablicy haszowej.

3.3 Model systemu pamięci transakcyjnej

W ogólności model jest pewną karykaturą[4] rzeczywistego zjawiska. Może przytłumiać pewne jego cechy, a uwydatniać inne, co czyni go mniej lub bardziej trafnym do określonych zastosowań (ale nigdy fałszywym lub prawdziwym) [50]. Model programistyczny pamięci transakcyjnych będzie koncentrował się na aspektach TM istotnych z perspektywy użytkowania tego

[4] zapewne Roy ma tu na myśli pewne starsze przykłady karykaturalnych portretów osób publicznych, cechujące się zazwyczaj nieproporcjonalnie dużymi głowami.

```
1  /* Zdalny interfejs bucketu. */
2  public interface RemoteBucket extends Remote {
3      Object get(Integer key) throws RemoteException;
4      void put(Integer key, Object value) throws RemoteException;
5  }
6
7  /* Implementacja bucketu. */
8  public class Bucket extends TransactionalUnicastRemoteObject  // bucket
           implements RemoteBucket {                            // dziedziczy po
9                                                               // specjalnej
10                                                              // klasie
11     public final int NUMBER;
12     private final Map<Integer, Object> contents
               = new HashMap<Integer, Object>();
13
14     public Bucket(int number) throws RemoteException {
15         super();
16         NUMBER = number;
17     }
18
19     @Override
20     public Object get(Integer key) throws RemoteException {
21         return contents.get(key);
22     }
23
24     @Override
25     public void put(Integer key, Object value) throws RemoteException {
26         contents.put(key, value);
27     }
28
29     public static void allocate(int myId, int numNodes,   // wywoływane przez
               int numBuckets) throws RemoteException {     // węzeł przy starcie
30         Registry registry = LocateRegistry.getRegistry();
31         for (int i = 0; i < numBuckets; i++) {
32             if (i % numNodes == myId) {
33                 Bucket bucket = new Bucket(i);
34                 RemoteBucket bucketStub = (RemoteBucket)
                               TransactionalUnicastRemoteObject
                               .exportObject(bucket, 0);
35                 registry.rebind("bucket" + i, bucketStub);
36             }
37         }
38     }
39 }
```

Rysunek 3.4: Konstruowanie t-obiektu w *Atomic RMI*.

systemu przez programistę aplikacji równoległych/rozproszonych. Algorytmy TM też będą swoistymi modelami. Uwzględniając (znowu) model środowiska, w którym będą działać ich implementacje, prezentują one koncepcje mechanizmów zapewnienia atomowego wykonania bloków kodu. Ten model może posłużyć do wnioskowania o poprawności, czy wydajności rozwiązań, a także o możliwościach ich zaimplementowania w praktyce, jednak nie będzie już dotyczył kwestii wprost związanych z implementacją (np. wyrażenia algorytmu w konkretnym języku programowania).

Na potrzeby te raportu wprowadzony będzie jeszcze jeden model pamięci transakcyjnej. Ten z kolei ma uwzględniać kwestie mające znaczenie dla pomiarów wydajności.

3.3.1 Model zdarzeniowy systemów rozproszonych

Chcąc objąć modelem elementy, w oparciu o które będzie można mierzyć wydajność rozproszonych pamięci transakcyjnych, trzeba zagłębić się w ich implementację. Punktem wyjścia do zbudowania modelu rozproszonej pamięci transakcyjnej będzie pewien ogólny model systemu rozproszonego uwzględniający w pierwszej kolejności proces (program, który w takim

```
1  /* Obudowa (wrapper) zbioru bucketów – tablica haszowa. */
2  public class Hashtable {
3      public final int MY_ID, NUM_NODES, NUM_BUCKETS;
4
5      public Hashtable(int myId, int numNodes, int numBuckets)
               throws RemoteException {
6          MY_ID = myId;
7          NUM_NODES = numNodes;
8          NUM_BUCKETS = numBuckets;
9          Bucket.allocate(myId, numNodes, numBuckets); // alokacja bucketów
10     }
11
12     public Object get(Integer key) throws RemoteException {
13         int bucketNumber = hash(key) % NUM_BUCKETS;
14         RemoteBucket bucket;
15         try {
16             bucket = (RemoteBucket) LocateRegistry.getRegistry()
                       .lookup("bucket"+bucketNumber);
17         } catch (NotBoundException e) {
18             throw new RemoteException("Bucket #"+bucketNumber+" not found!");
19         }
20         return bucket.get(key);
21     }
22
23     /* Zwraca false, jeżeli klucz już jest w tablicy. */
24     public boolean put(Integer key, Object value) throws RemoteException {
25         int bucketNumber = hash(key) % NUM_BUCKETS;
26         Registry registry = LocateRegistry.getRegistry();
27         RemoteBucket bucket;
28         try {
29             bucket = (RemoteBucket) registry.lookup("bucket"+bucketNumber);
30         } catch (NotBoundException e) {
31             throw new RemoteException("Bucket #"+bucketNumber+" not found!");
32         }
33         boolean done = false;
34         boolean duplicateFound = false;
35         while (!done) { // próbuje zatwierdzić transakcję do skutku
36             duplicateFound = false;
37             Transaction t = null;
38             try {
39                 t = new Transaction(registry); // budowanie access-setu
40                 RemoteBucket tBucket = t.accesses(bucket, 2);
41                 t.start(); // wykonanie transakcji
42                 if (tBucket.get(key) != null) {
43                     duplicateFound = true;
44                     t.free(tBucket); // wczesne zwalnianie bucketu
45                 } else {   // operacja która zajmuje dużo czasu, by
46                     {...}  // usprawiedliwić wczesne zwalnianie w przypadku
47                            // spełnienia warunku
48                     tBucket.put(key, value);
49                 }
50                 t.commit();
51                 done = true;
52             } catch (TransactionException e) {
53                 if (t != null && t.getState()
                           != Transaction.STATE_ROLLEDBACK) {
54                     try {
55                         t.rollback();
56                     } catch (TransactionException ee) {
57                         // można zignorować błąd wycofywania
58                     }
59                 }
60             }
61         }
62         return !duplicateFound;
63     }
64 }
```

Rysunek 3.5: Konstruowanie transakcji w *Atomic RMI*.

systemie się wykonuje). Taki model formalny procesu rozproszonego wprowadza [11].

Procesem sekwencyjnym P_i nazywana będzie czwórka:

$P_i = \langle \mathcal{S}_i, \mathcal{S}_i^0, \mathcal{E}_i, \mathcal{F}_i \rangle$

gdzie:
$\mathcal{S}_i$ – jest zbiorem stanów S_i procesu P_i,
$\mathcal{S}_i^0$ – jest zbiorem stanów początkowych procesu, takim że $\mathcal{S}_i^0 \subseteq \mathcal{S}_i$,
$\mathcal{E}_i$ – jest zbiorem zachodzących atomowo zdarzeń E_i, które powodują zmianę obecnego stanu procesu na kolejny,
$\mathcal{F}_i$ – jest funkcją tranzycji (przejścia): $\mathcal{F}_i \subseteq \mathcal{S}_i \times \mathcal{E}_i \times \mathcal{S}_i$, która określa możliwe kroki, jakie proces P_i może wykonać w danym stanie, tj. $\langle S, E, S' \rangle \in \mathcal{F}_i$ jeżeli tylko zajście zdarzenia E jest dopuszczalne w stanie S, a w jego efekcie proces zmienia stan na S'. Zdarzenie E powodujące przejście do stanu S' jest dopuszczalne w stanie S, jeżeli bezpośrednie przejście ze stanu S do S' nie narusza semantyki programu.

Jak widać, poszczególne przejścia wyznaczane przez funkcję tranzycji mogą być łączone w ciągi. Ciągi takie nazywane są *wykonaniami* lub też *realizacjami*. *Częściowym wykonaniem* nazywa się wykonanie $S_i^0, E_i^1, S_i^1, E_i^2, S_i^2, ..., S_i^s, E_i^{s+1}, S_i^{s+1}$, dla którego $\langle S_i^u, E_i^{u+1}, S_i^{u+1} \rangle \in \mathcal{F}_i$ dla każdego naturalnego u takiego że $0 \leq u \leq s$. Formalnie *wykonaniem* nazywane będzie takie częściowe wykonanie, które rozpoczyna się stanem początkowym $S_i^0 \subseteq \mathcal{S}_i^0$. Wykonanie zawsze kończy się stanem. Jeżeli stan S znajduje się na końcu pewnego wykonania, to nazywany będzie *stanem spójnym* lub *osiągalnym*. Gdyby z wykonania usunąć wszystkie zdarzenia, to wynikowy ciąg stanów nazywany będzie *śladem wykonania*. Gdyby z wykonania usunąć wszystkie stany, otrzymany ciąg zdarzeń nazywany będzie *historią wykonania procesu*.

Każdy proces, poza zdarzeniami odpowiadającymi zmianie jego stanu, może sprowokować zajście zdarzenia wykonując operacje komunikacyjne. W szczególności będzie to operacja wysłania wiadomości `M`, `send(Pi, Pj, M)`, gdzie `Pi` jest nadawcą, a `Pj` odbiorcą oraz operacja odebrania wiadomości, `receive(Pi, Pj, M)`, gdzie `Pi` jest nadawcą, a `Pj` odbiorcą[5]. Zdarzenia powiązane z tymi operacjami nazywane będą *zdarzeniami komunikacyjnymi* (w przeciwieństwie do *zdarzeń lokalnych*). Dla operacji `send()` i `receive()` będą to odpowiednio `e_send(Pi, Pj, M)` i `e_receive(Pi, Pj, M)`. Oczywiście fakt, że zdarzenia takie mogą zajść implikuje połączenie procesów kanałami komunikacyjnymi. W tym ujęciu zdarzenia komunikacyjne będą takimi zdarzeniami, które modyfikują stan kanałów komunikacyjnych incydentnych z procesem, podczas gdy zdarzenia lokalne wpływają jedynie na stan takiego procesu.

Opis procesu sekwencyjnego można rozszerzyć tak, by ująć przy pomocy analogicznych konstrukcji istotę procesu rozproszonego.

Procesem rozproszonym $\Pi = \{P_1, P_2, ..., P_n\}$ będzie nazywana czwórka:

$\Pi = \langle \Sigma, \Sigma^0, \Lambda, \Phi \rangle$

[5] a także warianty grupowe tych operacji: m.in. umożliwiający odebranie wielu wiadomości jednocześnie i nadanie wiadomości do wielu adresatów naraz – rozgłoszenie. Celem tego punktu nie jest jednak prezentacja pełnego modelu, a jedynie pokazanie w jaki sposób można wykorzystać pojęcie zdarzenia i jego obsługi do opisu systemów rozproszonych.

gdzie:
Σ – jest zbiorem stanów globalnych procesu rozproszonego: $\Sigma = \mathcal{S}_1 \times \mathcal{S}_2 \times ... \times \mathcal{S}_n$,
Σ^0 – jest zbiorem stanów początkowych, takim że $\Sigma^0 \subseteq \Sigma$,
Λ – jest zbiorem zdarzeń, takim że $\Lambda = \mathcal{E}_1 \cup \mathcal{E}_2 \cup ... \cup \mathcal{E}_n$,
Φ – jest funkcją analogiczną do funkcji tranzycji procesu sekwencyjnego: $\Phi \subseteq \Sigma \times \Lambda \times \Sigma$.

Tak jak stan procesu sekwencyjnego może być utożsamiany z zestawem bieżących wartości jego zmiennych (i ewentualnie zawartością incydentnych kanałów komunikacyjnych[6]), tak stan procesu rozproszonego będzie wektorem stanów składowych procesów sekwencyjnych. Zdarzenie procesu rozproszonego będzie zdarzeniem zachodzącym w jednym z procesów składowych, zmieniającym stan tego procesu i w efekcie stan całego procesu rozproszonego.

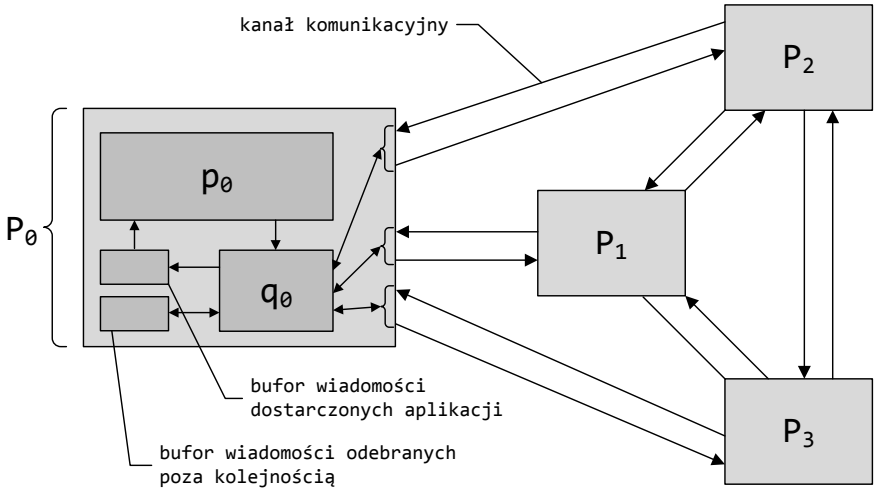
Pomiędzy zdarzeniami zachodzącymi w procesie rozproszonym wprowadzić można *relację poprzedzania*. Para zdarzeń $\langle E^x, E^y \rangle$ znajduje się w tej relacji jeżeli 1) są to zdarzenia zachodzące w tym samym procesie składowym i E^x następuje wcześniej, 2) E^x jest zdarzeniem nadania pewnej wiadomości, a E^y – zdarzeniem jej odebrania lub też 3) w historii wykonania istnieje taka sekwencja zdarzeń, w której E^x jest zdarzeniem początkowym, E^y – końcowym, a dla każdej pary kolejnych zdarzeń pomiędzy nimi zachodzi albo 1) albo 2). Jest jasne, że nie wszystkie pary zdarzeń będą objęte tą relacją. Stąd zdarzenia z niej wyłączone, w postaci par *zdarzeń współbieżnych* będą mogły występować w historii wykonania w różnych kolejnościach, w zależności od czasów trwania poszczególnych stanów procesów składowych. Wybór kolejnej tranzycji ze zbioru możliwych nie jest więc tutaj warunkowany jedynie możliwością (lub niemożliwością) odebrania pewnej wiadomości oraz tym, jaki jest kolejny krok przetwarzania (jak to miało miejsce w przypadku procesu sekwencyjnego). Podejście takie umożliwia zamodelowanie niedeterminizmu przetwarzania w systemach rozproszonych.

W ujęciu uwzględniającym pewne aspekty związane z implementacjami modelu, np. charakterystykę systemu pozwalającego węzłom komunikować się między sobą, lokalna składowa procesu rozproszonego (proces sekwencyjny wykonywany w węźle systemu rozproszonego) dzielona jest na dwa bloki funkcjonalne. Pierwszy z nich, *proces aplikacyjny*, `p_i`, realizuje pewien cel określony przez użytkownika systemu rozproszonego. Drugi, *monitor*, `q_i`, odpowiedzialny będzie za wszystkie inne operacje, w tym wykonanie algorytmów operujących na warstwie middleware takiego systemu. Rozróżnienie to pozwoli m.in. bardziej szczegółowo opisać mechanizm transferu wiadomości (aplikacyjnej) z jednego procesu (aplikacyjnego) do innego.

Gdy proces aplikacyjny chce nadać komunikat, przesyła jego treść oraz informację o odbiorcy monitorowi. Ten umieszcza wiadomość w kanale komunikacyjnym, opatrując ją uprzednio informacjami kontrolnymi, *metadanymi*. Gdy wiadomość dociera do docelowego węzła systemu, otrzymuje ją monitor i przy uwzględnieniu zasad przekazywania wiadomości między procesami aplikacyjnymi (np. konieczności zachowania porządku FIFO w komunikacji na tej warstwie) albo ją buforuje do czasu zajścia pewnego zdarzenia, albo udostępnia procesowi aplikacyjnemu będącemu odbiorcą. Takie udostępnienie wiadomości daje pole do wpro-

[6] nadanie wiadomości wyzwala zdarzenie `e_send`, które powoduje przejście procesu do następnego stanu. Przejście takie nie musi (choć zazwyczaj będzie) modyfikować zawartości zmiennych dla tego procesu.

wadzenia kolejnego zdarzenia komunikacyjnego – `e_deliver(`P_i`, `P_j`, M)`. W zależności od konwencji, jego zajście, wyzwalane przez monitor, może wymusić na procesie aplikacyjnym wykonanie tranzycji, lub jedynie umożliwić mu w przyszłości wykonanie z powodzeniem operacji `receive()`, czyniąc jedną z przyszłych tranzycji możliwą. Należy dodać, że w takim ujęciu operacja `receive()` (sprawdź czy nie dotarła wiadomość od określonego nadawcy) wykonywana jest przez proces aplikacyjny, natomiast zdarzenie `e_receive` (nowa wiadomość dotarła kanałem komunikacyjnym) obsługiwane jest przez monitor.



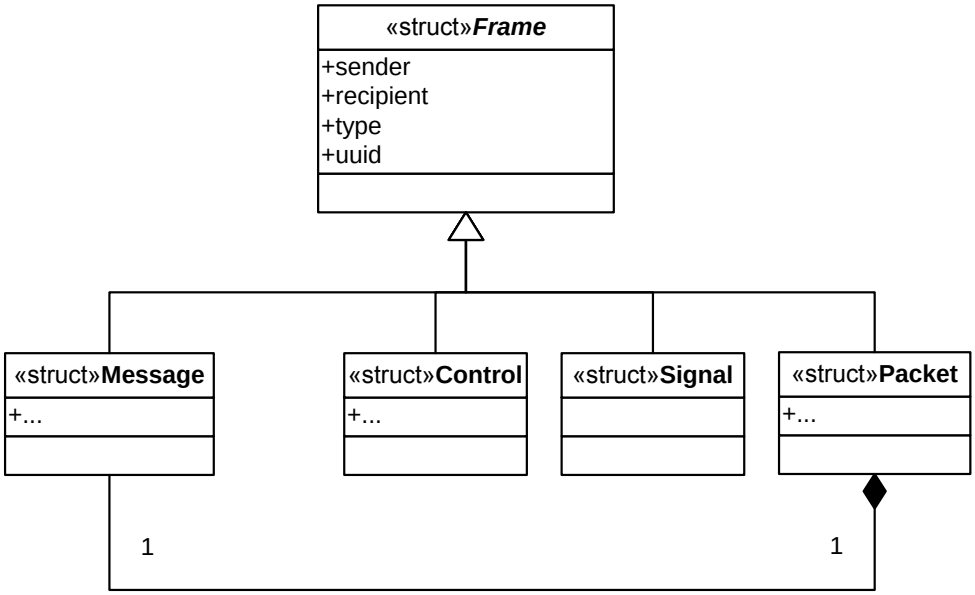
Rysunek 3.6: Podział składowej lokalnej procesu rozproszonego na proces aplikacyjny `p` i monitor `q` w kontekście zapewnienia porządku FIFO wiadomości tam, gdzie nie zapewniają jej kanały komunikacyjne.

Na potrzeby pewnych algorytmów rozproszonych monitory mogą chcieć rozmawiać bezpośrednio ze sobą. W tym celu wprowadza się rozróżnienie typów wiadomości przesyłanych przez kanały komunikacyjne:

- *ramka* (ang. *frame*) – jest to prototyp jakiejkolwiek wiadomości przesyłanej w systemie rozproszonym. Ramka jest strukturą zawierającą pola przechowujące informacje o nadawcy i odbiorcy wiadomości, jej typie i unikatowym identyfikatorze (np. numerze sekwencyjnym). Typ ten nigdy nie jest wykorzystywany bezpośrednio do konstrukcji wiadomości wysyłanej do innego procesu. Stanowi więc rodzaj klasy abstrakcyjnej, grupującej wspólne elementy wywiedzionych z niego pozostałych typów wiadomości.
- *komunikat aplikacyjny* (ang. *message*) – dziedziczy po ramce tak, by umożliwić procesowi aplikacyjnemu odczytanie metadanych wiadomości. Poza polami ramki komunikat aplikacyjny zawiera też inne dane, których znaczenie jest ustalane przez aplikację-nadawcę (przy czym w sposób oczywisty musi być też znane odbiorcy).
- *wiadomość kontrolna* (ang. *control*) – dziedziczy po ramce, jednak w przeciwieństwie do komunikatu aplikacyjnego pozwala na porozumienie się ze sobą monitorów. Dane zawarte w tym typie wiadomości (poza polami ramki) są kwestią specyficzną dla algorytmu realizowanego przez monitory, którego wiadomość kontrolna jest częścią.
- *sygnał* (ang. *signal*) – jest specyficznym typem wiadomości kontrolnej, która nie zawiera

żadnych dodatkowych danych. Jej rolą będzie więc np. powiadomienie monitora o zajściu pewnego zdarzenia w innym.

- *pakiet* (ang. *packet*) – dziedzicząc po ramce kapsułkuje jednocześnie wiadomość aplikacyjną tak, by mogła ona zostać przesłana pomiędzy monitorami. Może też zawierać inne, potrzebne monitorom pola. Warto zwrócić uwagę na nadmiarowość danych – pola ramki zarówno w komunikacie aplikacyjnym, jak i w pakiecie. Monitor podczas odbioru pakietu dekapsułkuje wiadomość aplikacyjną i dostarcza ją aplikacji, więc by proces aplikacyjny mógł np. zidentyfikować nadawcę, niezbędne jest umieszczenie metadanych także w niej. Dla pewnych algorytmów zadaniem monitora będzie też przekazanie pakietu dalej. Wówczas metadane wiadomości aplikacyjnej posłużą np. do zidentyfikowania jej ostatecznego odbiorcy.



Rysunek 3.7: Diagram klas reprezentujących poszczególne typy wiadomości, wyrażony w języku UML.

W kontekście niniejszego raportu celem wprowadzania modelu jest pokazanie, że ze zdarzeniami mogą zostać powiązane pewne procedury ich obsługi. Fakt ten zostanie wykorzystany później, przy próbie zamodelowania w podobny sposób pamięci transakcyjnej. By jednak teraz dopełnić opis modelu, przedstawiona zostanie reprezentacja mechanizmu zegarów logicznych Lamporta wyrażona za jego pomocą.

3.3.2 Model zdarzeniowy a pamięć transakcyjna

Jeżeli by założyć, że w podobny do opisanego wcześniej sposób można zdefiniować zdarzenia związane z operacjami wykonywanymi w powiązaniu z transakcjami i do każdego takiego zdarzenia dołączyć procedurę obsługi, bądź to całkowicie zastępującą oryginalną operację (jak w przypadku instrumentacji), bądź jedynie wykonywaną oprócz niej i mającą wgląd w jej

```
1  struct Packet extends Frame {
2      Message data;       // kapsułkowana wiadomość aplikacyjna
3      int clock;          // wartość zegara nadawcy
4  }
```

Rysunek 3.8: Definicje typów komunikatów na potrzeby realizacji mechanizmu Lamporta.

```
1  /* Obsługa zdarzenia e_send procesu aplikacyjnego. */
2  when e_send(Process sender, RemoteProcess recipient, Message message) {
3      // zwiększenie wartości zegara nadawcy
4      sender.clock += DELTA; // dla uproszczenia zazwyczaj DELTA = 1
5
6      // kapsułkowanie wiadomości aplikacyjnej w pakiecie
7      Packet packet = new Packet();
8      packet.clock = sender.clock;
9      packet.data = message;
10
11     // operacja send() w monitorze umieszcza pakiet w odpowiednim
12     // kanale komunikacyjnym
13     send(sender, recipient, packet);
14 }
```

Rysunek 3.9: Procedura nadania wiadomości o treści otrzymanej od procesu aplikacyjnego przez monitor $q_{\mathtt{sender}}$ procesu składowego $P_{\mathtt{sender}}$.

parametry, byłby to dobry punkt wyjścia do budowy szkieletu (*frameworka*) programu wykorzystującego pamięć transakcyjną jako mechanizm sterowania współbieżnością. Szkielet taki w drugim przypadku mógłby posłużyć do analizy działania implementacji określonych algorytmów TM i wykonywania pomiarów wydajnościowych. W pierwszym zaś, pozwalałby w łatwy sposób łączyć aplikacje z istniejącymi implementacjami pamięci transakcyjnej i tym samym porównywać różne rozwiązania projektowe takich pamięci w warunkach zbliżonych do rzeczywistych.

Podobne podejście stosowane jest w narzędziu STAMP, gdzie programista dostaje do dyspozycji szereg programów wzorcowych, w których każde kluczowe zdarzenie związane z pamięcią transakcyjną opatrzone jest makrem preprocesora języka C. Dzięki temu programista może w prosty sposób zintegrować benchmark z pamięcią transakcyjną o niemal dowolnym API, ale też i np. zdecydować, że przy okazji odwołania do systemu TM w momencie zajścia zdarzenia, jego parametry mają zostać przechwycone i zarejestrowane na potrzeby późniejszej analizy wykonania.

Podejście to okazało się użyteczne dla wieloprocesorowych pamięci transakcyjnych. Może ono odegrać podobną rolę w systemach rozproszonych, z jednym zastrzeżeniem, dotyczącym problemów w odczytywaniu stanu systemu rozproszonego, a opisanym dalej w punkcie 3.3.3. Póki co podejmuje się próbę sformułowania listy zdarzeń, jakie mogą zajść w systemie wykorzystującym rozproszoną pamięć transakcyjną (mając na względzie użycie wykonywanego w nim programu w roli benchmarka) razem z ich parametrami. Lista ta, przy zachowaniu ograniczonej nadmiarowości, powinna dawać punkty zaczepienia zarówno implementacjom pamięci transakcyjnych, jak i systemowi pomiarowemu, traktującemu taką pamięć jako maszynę stanów.

```
1  /* Obsługa zdarzenia e_receive monitora. */
2  when e_receive(RemoteProcess sender, Process recipient, Packet packet) {
3      // uaktualnienie zegara odbiorcy
4      recipient.clock = max(recipient.clock, packet.clock) + DELTA
5
6      // dekapsułkowanie wiadomości aplikacyjnej
7      Message data = packet.data;
8
9      // udostępnienie wiadomości procesowi aplikacyjnemu
10     deliver(sender, recipient, data);
11 }
```

Rysunek 3.10: Procedura dostarczania odebranej wiadomości do procesu aplikacyjnego przez monitor procesu składowego $\mathrm{P}_{\texttt{recipient}}$.

Tablica 3.2: Podstawowe zdarzenia identyfikowane w ramach rozproszonych pamięci transakcyjnych oraz ich parametry.

Zdarzenie	Parametry	Opis
Zdarzenia z ramach programu wykorzystującego D-STM		
`e_program_start`	• liczba współbieżnych jednostek przetwarzania	Zdarzenie rozpoczęcia przetwarzania, razem z informacją o limicie nakładanym na liczbę uruchamianych współbieżnych wątków (parametr tylko w przypadku wieloprocesorowych STM) lub węzłów przetwarzania (D-STM, jeżeli tylko węzły nie dołączają/odłączają dynamicznie). Taki parametr był wymagany przez niektóre STM badane przy użyciu STAMPa.
~~`e_pr_start`~~[7]	• identyfikator etapu przetwarzania • dane przekazywane wątkowi	Zdarzenie rozpoczęcia wykonywania regionu współbieżnego (wątku) na pewnym etapie przetwarzania (w przypadku wieloprocesorowych STM).
~~`e_pr_end`~~[7]		Zdarzenie zakończenia wykonywania regionu współbieżnego (wątku).

[7] zdarzenie nie będzie występowało w rozproszonych pamięciach transakcyjnych. Zostało ono ujęte w tabeli dla dopełnienia obrazu. Nawet jeżeli węzły systemu rozproszonego będą działać wielowątkowo, to będzie się zakładać, że każdy taki wątek odpowiada osobnemu, „logicznemu” węzłowi.

Zdarzenie	Parametry	Opis
`e_stage_start`	• identyfikator etapu	Zdarzenie rozpoczęcia określonego etapu przetwarzania. Podział aplikacji równoległej na wysokim poziomie abstrakcji na (raczej sekwencyjne) etapy przetwarzania danych został zapożyczony z wybranych programów zestawu STAMP (np. *bayes*, *genome*, *ssca2*) i będzie, w miarę możliwości, odtwarzany w benchmarkach rozproszonych.
`e_stage_end`	• identyfikator etapu	Zdarzenie zakończenia określonego etapu przetwarzania.
Zdarzenia w ramach transakcji		
`e_start`	• klasa transakcji • access-set/read- i write-set (jeżeli wymagane) • numer powtórzenia wykonania • czy transakcja *read-only*?	Zdarzenie rozpoczęcia wykonywania transakcji (pierwszej próby) lub jej kolejnego powtórzenia.
`e_commit`		Zdarzenie wywołania przez programistę (lub w wyniku instrumentacji kodu transakcyjnego) procedury zatwierdzania transakcji.
`e_committed`	• czy wykryto konflikt/czy zatwierdzono transakcję?	Zdarzenie zakończenia wykonania procedury zatwierdzania transakcji.
`e_abort`	• powód wycofania (*forcible*/*non-forcible rollback*)	Zdarzenie zlecenia przez programistę lub przez system TM wycofania transakcji.

Zdarzenie	Parametry	Opis
`e_retry`	• powód wycofania przed ponowieniem (*forcible*/*non-forcible rollback*)	Zdarzenie zlecenia przez programistę lub przez system TM ponowienia wykonania transakcji.
`e_access`	• identyfikator t-obiektu • czy jest to odczyt/zapis (jeżeli STM/D-STM rozróżnia)? • czy dostęp został zlecony z wewnątrz transakcji? • czy właścicielem obiektu jest ten sam węzeł, który inicjował transakcję (dla dostępów z transakcji? • czy dostęp odbywa się z pominięciem TM (dla dostępów z transakcji)?	Zdarzenie dostępu do współdzielonego obiektu/t-obiektu wykonywanego przez wątek/węzeł.
`e_accessed`	• czy wykryto konflikt? • czy rozwiązano konflikt?	Zdarzenie zakończenia realizacji zlecenia dostępu do współdzielonego obiektu/t-obiektu. Jeżeli przyjąć że zdarzenie to jest obsługiwane przez system pomiarowy traktujący TM jako maszynę stanów, pozostałe parametry zdarzenia można wydobyć z danych zachowanych przez procedurę obsługi `e_access`.

Zdarzenie	Parametry	Opis
`e_laccess`		Zdarzenie wykonania przez transakcję każdego dostępu innego niż dostęp (transakcyjny lub nie) do t-obiektu (sprywatyzowanego lub nie). Wyróżnienie tego zdarzenia może posłużyć do ustalenia długości transakcji w postaci liczby wszystkich realizowanych w niej operacji.
`e_release`	• identyfikator t-obiektu • zbiór, z którego obiekt jest wyłączany	Zdarzenie wczesnego zwolnienia t-obiektu z read-, write- lub access-setu.
Zdarzenia transakcyjne występujące poza transakcją		
`e_raccessed`	• identyfikator t-obiektu • identyfikator węzła zlecającego • czy jest to odczyt/zapis (jeżeli D-STM rozróżnia)? • czy dostęp został zlecony z wewnątrz transakcji? • czy dostęp odbywa się z pominięciem TM (dla dostępów z transakcji)?	Zdarzenie pojawienia się w węźle lokalnym żądania (zdalnego) dostępu do przechowywanego w nim t-obiektu, które zostało przesłane przez pewien węzeł zdalny.

3.3.2.1 Implementacja modelu – *Deuce*

Ponieważ wiele systemów rozproszonej pamięci transakcyjnej, w tym wszystkie analizowane wtym raporcie, pozwala na tworzenie transakcji w języku Java [18, 54, 62, 64], podejście bazujące na makrach preprocesora *à la* C będzie tu niemożliwe do zastosowania. Przed stworzeniem programów wzorcowych dla tych systemów, należałoby więc rozważyć możliwości w zakresie integracji ich z pamięciami transakcyjnymi i systemem pomiarowym odpowiednie dla języka Java.

Idea szkieletu pamięci transakcyjnej jako narzędzia, które pozwoliłoby dostarczyć procedury obsługi zdarzeń zachodzących w systemach TM z pewnością przyświecała twórcom narzędzia *Deuce* [38, 39]. *Deuce* jest bowiem rozwiązaniem w zakresie instrumentowania kodu transakcyjnego dla języka Java, które pozwala łączyć określone (ale cały czas standardowe)

konstrukcje językowe służące definiowaniu transakcji z implementacjami STM.

W zakresie modelu programistycznego narzędzie to nie różni się znacznie od odpowiadającego mu fragmentu modelu programistycznego *HyFlow*[8]. Kod transakcyjny zawarty jest wewnątrz metody opatrzonej odpowiednią adnotacją. Jest on następnie instrumentowany na etapie ładowania do pamięci operacyjnej tak, by odwołania do obiektów współdzielonych przez wątki z wewnątrz transakcji przekierowywane były do systemu TM. Od strony projektanta pamięci transakcyjnej *Deuce* dostarcza możliwość obsługi następujących zdarzeń[9]:

- „sztuczne" zdarzenie tuż przed odczytem pola obiektu współdzielonego (zapewne umożliwiające opóźnienie samego odczytu jeżeli jet to potrzebne do uniknięcia konfliktu),
- odczyt pola obiektu współdzielonego,
- zapis pola obiektu współdzielonego,
- rozpoczęcie transakcji,
- zatwierdzenie transakcji,
- wycofanie transakcji,

Możliwa jest także implementacja różnych reguł arbitrażu. *Deuce* dostarcza wreszcie w pakiecie przykładowe implementacje algorytmów TM: optymistycznego z późnym zarządzaniem wersjami – *Transactional Locking II* (bez i ze wsparciem dla reguł arbitrażu) oraz optymistycznego z wczesnym zarządzaniem (wieloma) wersjami – *Lazy Snapshot Algorithm* [49] (również bez i ze wsparciem dla reguł arbitrażu).

Na potrzeby realizacji celów stawianych wtym raporcie *Deuce* nie będzie jednak dobrym wyborem. W pierwszej kolejności nie oferuje on tego, czego oczekiwałoby się po mechanizmie integracji programu wzorcowego z pamięcią transakcyjną i systemem pomiarowym. Zakresy odpowiedzialności *Deuce* i modułu instrumentacji kodu *HyFlow* pokrywają się, co w zasadzie wyklucza użycie obu naraz (choć można by pokusić się o zastosowanie *Deuce* w innych pamięciach transakcyjnych, tak by upodobnić je do *HyFlow* w tym aspekcie). Dodatkowo ubogi zestaw zdarzeń, które mogą być obsłużone nie pozwoli np. na zintegrowanie *Deuce* z pamięciami statycznymi, co jest dla tego raportu wymogiem niezbędnym. Wreszcie *Deuce* nie jest szkieletem dla rozproszonych pamięci transakcyjnych, ani też nie pozwala na pełną integrację z rozwiązaniami typu control-flow (chyba, że zdalnymi metodami będą jedynie gettery i settery dla pól w t-obiektach).

W efekcie należy wybrać inny sposób „sprzęgnięcia" programu wzorcowego z systemem pomiarowym. Dla uproszczenia nie będziemy rozważać przy tej okazji metod łączenia programu wzorcowego z samym systemem pamięci transakcyjnej.

[8] zapewne jest tak dlatego, że *HyFlow* wykorzystuje odpowiednio zmodyfikowaną wersję *Deuce* jako moduł instrumentacji kodu transakcyjnego.

[9] por. dokumentacja JavaDoc dla klasy `org.deuce.transaction.tl2.Context`, http://deuce.googlecode.com/svn/trunk/Deuce/doc/, dostęp 5 czerwca 2015 r.

3.3.3 Model zdarzeniowy a rozproszona pamięć transakcyjna

Budowa systemu pomiarowego dla rozproszonych pamięci transakcyjnych, a co za tym idzie – także dobór zdarzeń obsługiwanych przez ten system nie są zadaniami łatwymi. O ile w systemach wieloprocesorowych pomiary mogą odbywać się w oparciu o stan całego systemu, o tyle samo ustalenie bieżącego stanu systemu rozproszonego nie jest trywialne. Trzymając się jednak tego podejścia można wyobrazić sobie implementację systemu pomiarowego w oparciu o globalne repozytorium wyników. W systemie rozproszonym mogłoby ono jednak stać się źródłem wąskich gardeł przetwarzania i tym samym, opóźniając wykonanie procedur obsługi zdarzeń, zniekształcać rezultaty. Stąd wtym raporcie przyjęte zostanie podejście bazujące na pomiarach realizowanych w oparciu o stan lokalny węzła i scaleniu wyników po zakończeniu przetwarzania aplikacyjnego.

To rodzi kolejny problem. Jak bowiem w takim ujęciu wyznaczyć np. poziom współzawodnictwa w dostępie do zasobów? W pamięciach optymistycznych można tego dokonać choćby zliczając wycofania transakcji i obliczając przy scalaniu średnią liczbę powtórzeń przypadających na jedną z nich. W pamięciach pesymistycznych, zwłaszcza control-flow sprawa jest już bardziej skomplikowana. Można próbować mierzyć czas wykonania metody zdalnej (od momentu jej wywołania do zwrócenia wyniku). Nie sposób jednak ustalić jaki ułamek tego czasu przeznaczony jest na oczekiwanie na dostęp do obiektu, a jaki – na faktyczne wykonanie metody bez ingerowania w mechanizmy operujące na warstwie middleware. Pomiary będą więc czasem wymagały dostępu do stanu węzłów zdalnych względem tego, na którym są wykonywane. Znowu w systemie rozproszonym operacja wglądu w stan innego węzła nie jest prosta do zaimplementowania, ze względu na związane z nią narzuty czasowe i możliwość powstania wąskich gardeł przetwarzania.

Można sobie wyobrazić trzy podejścia do rozwiązania tego problemu:

1. rezygnację z wyznaczania miar, do obliczenia których nie wystarcza stan lokalny węzła,
2. rozszerzenie systemu pomiarowego na warstwę middleware. W tym ujęciu system pomiarowy mógłby w razie potrzeby sprowadzać niezbędne parametry ze zdalnych węzłów lub wręcz wyznaczać spójny obraz stanu całego systemu,
3. szerszą integrację systemu pomiarowego z systemem rozproszonej pamięci transakcyjnej. Pamięć transakcyjna niekiedy będzie musiała sama sprowadzić do lokalnego węzła odpowiednie dane, czy to na potrzeby wdrożenia reguły arbitrażu (jak dla *karma* w *HyFlow*), czy też na potrzeby realizacji samego algorytmu TM (jak w *Atomic RMI*).

Podejście pierwsze wydaje się za bardzo uproszczone, podczas gdy drugie zapewne będzie wyjątkowo kłopotliwe w implementacji. W efekcie podejściem stosowanym wtym raporcie będzie scalanie po zakończeniu wykonania programu wzorcowego pomiarów dokonywanych w oparciu o stan węzła i elementy stanu systemu sprowadzone do tego węzła przez pamięć transakcyjną.

3.4 Dyskusja wymagań

Wymagania stawiane programom wzorcowym mającym służyć do wydajnościowego testowania pamięci transakcyjnych zostały już częściowo przedstawione w punkcie 2.5.3. Niniejszy punkt stawia sobie za cel precyzyjne ujęcie kryteriów pochodzących z różnych źródeł. Ich określenie pozwoli wskazać jak wytworzyć nowy lub przystosować istniejący program tak, by pełnił on rolę 1) dobrego benchmarka 2) odkrywającego wydajnościowe wady i zalety różnych implementacji 3) rozproszonej pamięci transakcyjnej, choć w ogólności spełnienie wszystkich warunków przez pojedynczy program będzie niemalże niemożliwe.

3.4.1 Wymagania wywiedzione z koncepcji programu wzorcowego

1) **Prostota**. Algorytm implementowany przez program wzorcowy powinien być stosunkowo prosty. W tym raporcie rozpatrywane są rozproszone pamięci transakcyjne możliwe do wykorzystania w języku Java. Jednak dobry, uniwersalny benchmark powinien łatwo dawać się przetłumaczyć na inne języki, a nawet na inne paradygmaty programowania.

2) **Realność generowanych obciążeń**. Dobry program wzorcowy powinien powinien obciążać badany system pamięci transakcyjnej w sposób możliwie najbliższy rzeczywistej aplikacji rozproszonej. W zasadzie najlepszym rozwiązaniem byłoby użycie w roli benchmarka prawdziwej aplikacji przy zintegrowaniu jej z analizowaną rozproszoną pamięcią transakcyjną. Wynik zastosowania takiego podejścia nie będzie jednak spełniał szeregu postawionych później wymagań. Stąd w roli benchmarka będzie występował raczej model aplikacji rozproszonej, który jednak w żaden sposób nie spłyca i nie ogranicza charakterystyki wykonywanych przez nią transakcji.

3) **Różnorodność generowanych obciążeń**. By uwypuklić różnorodne cechy analizowanej pamięci transakcyjnej w kontekście jej wydajności, benchmark powinien generować transakcje o różnych parametrach. Jak pokazuje punkt 2.5.1 (problem z niską amortyzacją kosztów stałych wykonania transakcji), szczególnie przydatne mogą okazać się transakcje długie. Dodatkowo ze względu na spodziewaną zróżnicowaną zależność wydajności podejść optymistycznego i pesymistycznego od poziomu współzawodnictwa, benchmark powinien generować transakcje cechujące się różnorodnością w tym obszarze ze szczególnym uwzględnieniem wysokiego „zatłoczenia".

3.4.2 Wymagania zapożyczane od narzędzia STAMP

4) **Demonstracja szerokiego spektrum zastosowań pamięci transakcyjnej**. Kryterium *breadth* zapożyczone ze STAMPa zazwyczaj nie będzie możliwe do osiągnięcia w pojedynczym programie wzorcowym. Można wyobrazić sobie program składający się z kliku modułów, przykładowo zorganizowanych w potok, z których każdy wykonuje określony krok obróbki danych (por. punkt 2.6.1.6, benchmark *ssca2*). Jeżeli może się to kłócić z wymogiem 1) oczekującym od benchmarka prostoty, to tym bardziej takie podejście kłóci się z samym wymogiem 4), ponieważ demonstrowane zastosowania pamięci transakcyjnych w różnych dziedzi-

nach informatyki powinny zazwyczaj być na tyle różne od siebie, by nie dać się ująć w jeden proces obróbki danych.

Wymóg *depth*, również zapożyczony ze STAMPa, będzie równoważny wymogowi 3). W STAMPie jest on dodatkowo sformalizowany w następujący sposób: transakcje powinny być różnej długości (w sensie liczby operacji objętych transakcją). W szczególności benchmark powinien dostarczać długie transakcje tak, by umożliwić zamortyzowanie stałych kosztów ich rozpoczęcia i zatwierdzenia. Transakcje powinny posiadać też read- i write-sety różnej liczności. W szczególności w oparciu o transakcje operujące na dużej liczbie t-obiektów możliwa będzie ewaluacja zachowania pamięci transakcyjnej w przypadku przekroczenia granic wyznaczanych przez rozmiar struktur kontrolnych powodującego np. nasilenie występowania fałszywych konfliktów. Wreszcie z racji różnic pomiędzy podejściami pesymistycznym i optymistycznym istotne będzie uzyskanie różnorodnego (w tym – wysokiego) poziomu współzawodnictwa.

5) **Przenośność**. Kryterium to, *portability* w odniesieniu do STAMPa, częściowo pokrywa się z 1). Proste implementacje z pewnością łatwiej jest przenosić pomiędzy językami i paradygmatami programowania. W ogólności jednak chodzi o to, by można było łączyć program wzorcowy z pamięciami transakcyjnymi o różnych interfejsach programistycznych. By to osiągnąć niezbędny jest dostatecznie rozbudowany mechanizm tworzenia takich połączeń o niemałych możliwościach.

3.4.3 Wymagania wywiedzione z rozproszonego charakteru testowanych pamięci

6) **Realność generowanych obciążeń systemu rozproszonego**. W zasadzie wymóg ten mógłby być tożsamy z 2). Jednak ponieważ niniejszy raport koncentruje się na rozproszonych pamięciach transakcyjnych, istotne będzie by obciążać pamięć transakcyjną w mniej więcej taki sposób, w jaki robiłaby to prawdziwa aplikacja rozproszona. To z kolei może mieć kluczowe znaczenie, ponieważ wymóg 2) – realność generowanych obciążeń nie musi kłócić się z 1) – prostotą algorytmu, ale 6) już będzie. Próg złożoności problemów, na którym przestają do ich rozwiązania wystarczać środowiska równoległe, tj. próg komplikacji aplikacji, od którego opłaca się je tworzyć dla systemów rozproszonych jest bowiem wysoki ze względu na trudność zaprogramowania takich systemów.

7) **Możliwość wykonania pomiarów**. Wymóg ten łączy się z wymogiem 5) w tym sensie, że jeżeli benchmark jest przenośny dlatego, że został wyposażony w elastyczny mechanizm łączenia go z implementacją pamięci transakcyjnej, to zapewne połączenie go także z pewnym systemem pomiarowym (który, w przeciwieństwie do TM, nie obsługuje zdarzeń, a jedynie rejestruje ich parametry) nie powinno być skomplikowane. W przypadku rozproszonych pamięci transakcyjnych ma on jednak dodatkowe znaczenie. W punkcie 3.3.3 podano trzy podejścia do radzenia sobie z rozproszeniem danych w oparciu o które obliczane będą metryki (wskazano też to, które będzie używane w dalszej części raportu). W zasadzie nie jest to wymóg kierowany bezpośrednio w stronę benchmarków (chyba, że założymy na przykład, że rozwiązujemy opisany w punkcie 3.3.3 problem z oceną jaka część czasu oczekiwania na wy-

nik metody zdalnej została faktycznie przeznaczona na jej wykonanie poprzez zapewnienie zawsze stałego czasu wykonania). Chodzi o to, by zarówno system umożliwiający obsługę zdarzeń (framework łączący TM z benchmarkiem), jak i sama pamięć transakcyjna dostarczały sposoby wglądu w stan middleware (implementacji tej pamięci) tak, by dało się obliczyć potrzebne metryki bez dodatkowego sprawdzania stanu innych węzłów systemu rozproszonego. Być może pewne elementy takiego mechanizmu wykroczą poza framework i, tym samym, będą musiały zostać uwzględnione na etapie implementacji benchmarka.

8) **Odróżnienie od mikrobenchmarka**. Dla rozproszonych pamięci transakcyjnych istnieje już szereg mikrobenchmarków. Mając na myśli benchmarki dla systemu *HyFlow* (por. punkt 2.6.4) można pokusić się o przypuszczenie, że są to narzędzia do testowania wydajności implementowane *ad hoc*. Takiemu „utylitarnemu" podejściu, polegającemu na użyciu takich narzędzi, jakie można stworzyć przy ograniczonych zasobach i w zadanym czasie zdaje się zaprzeczać *EigenBench* (por. punkt 2.6.2). Oddaje on jednak dobrze tę cechę mikrobenchmarka, której należałoby unikać przy tworzeniu pełnowymiarowych rozwiązań. Programy takie bowiem albo sprawdzają wybrane, pojedyncze aspekty działania TM, albo ich grupy, jednak cały czas zakładając że aspekty te nie są ze sobą powiązane. Innymi słowy mikrobenchmark generuje takie obciążenia systemu pamięci transakcyjnej, które pojedynczy aspekt (względnie ich wąską grupę) uwypuklają. Od pełnowymiarowego benchmarka oczekuje się większego stopnia wszechstronności, objawiającego się np. tym że generowane przezeń obciążenia uwypuklają wszystkie aspekty działania TM i to w takim stopniu, w jakim robiłaby to rzeczywista aplikacja.

Stąd dla rozwiązań bazujących na współdzielonej złożonej strukturze danych należałoby zapewnić szereg różnorodnych schematów jej użycia (a więc i szereg klas transakcji, każda o innej charakterystyce i znaczeniu aplikacyjnym). Natomiast w przypadku mniej jednorodnych benchmarków (tj. takich, w których parametry transakcji nie są określane dowolnie przez programistę, ale wynikają z semantyki rozwiązywanego problemu algorytmicznego) należałoby zapewnić, że na różnych (potencjalnie zrównoleglonych) etapach przetwarzania będą one korzystać z pamięci transakcyjnej w różny sposób (znowu produkując szereg klas transakcji o różnych charakterystykach).

3.4.4 Wymagania narzucane przez modele programistyczne D-STM

9) **Przewidywalne read- i write-sety**. Jedną z rozproszonych pamięci transakcyjnych analizowanych wtym raporcie jest *Atomic RMI*. Ponieważ wymaga ona znajomości zawartości acess-setu przed rozpoczęciem transakcji, jest pamięcią statyczną. Wymaganie to, dosyć specyficzne, odnosić się będzie także do sytuacji, w których w użyciu będzie pamięć dynamiczna, przede wszystkim z tego powodu, że nie sposób go pominąć dla *Atomic RMI*. Stąd chcąc dostarczyć aplikację porównującą ze statycznym dowolne inne podejście, wymóg znajomości read- i write-setów (albo access-setów) *a priori* stworzy wspólny mianownik dla wszystkich testowanych mechanizmów sterowania współbieżnością.

Oczywiście dysponowanie listą obiektów współdzielonych, do których transakcja uzyskuje dostęp przed rozpoczęciem wykonania rodzi podejrzenie, że cały system pamięci transakcyjnej można by zastąpić zamkami drobnoziarnistymi. Znajomość tego jakie zamki trzeba

zamknąć, by pewien blok kodu mógł operować na potrzebnych mu zasobach w sposób wyłączny, pozwala je posortować i zamykać zawsze w określonej kolejności, a to da możliwość uniknięcia podstawowego problemu związanego z tą metodą sterowania współbieżnością – zakleszczeń. Stąd by dać statycznej pamięci transakcyjnej szansę działać w środowisku, w którym może konkurować z innymi metodami sterowania współbieżnością dopuszczalne będzie podejście, w którym access-set konstruowany jest w sposób przybliżony. Po rozpoczęciu transakcji, w tym po zablokowaniu elementów access-setu sprawdzane będzie (w sposób odporny na anomalie niesynchronizowanego współbieżnego dostępu) czy nie uległ on zmianie, tj. czy procedura jego konstrukcji nie zwraca teraz innego wyniku. Jeżeli tak by się zdarzyło, będzie to przesłanką do natychmiastowego wycofania i ponownego wykonania transakcji, z uprzednią konstrukcją access-setu jeszcze raz. Takie podejście rodzi potrzebę wprowadzenia w benchmarku odpowiedniego zestawu klas transakcji, które dokonują małych zmian, lub też dokonują zmian rzadko. W przeciwnym wypadku istnieje ryzyko, że dla pewnych transakcji nigdy nie uda się zakończyć konstrukcji access-setu.

10) **Prywatyzowanie danych**. W punkcie 2.5.1 (nadmierna instrumentacja kodu) wskazano znaczenie wyboru pomiędzy instrumentacją kodu transakcyjnego wykonywaną automatycznie przez pewien moduł TM, a jawnym użyciem przez programistę procedur `read()` i `write()` dostępu do zasobu współdzielonego. Ponieważ jedną z różnic pomiędzy obiema analizowanymi wtym raporcie pamięciami będzie właśnie użycie automatycznej instrumentacji (wykonywanej już nie przez kompilator, ale przez odpowiedni moduł w trakcie ładowania programu do pamięci), benchmark powinien pozwalać na pokazanie różnic pomiędzy tymi podejściami.

Jedną z zalet „ręcznej instrumentacji" kodu transakcyjnego jest możliwość pominięcia pamięci transakcyjnej przy dostępach do danych sprywatyzowanych, które programista zleca z wewnątrz transakcji. Stąd benchmark powinien wykorzystywać mechanizm prywatyzacji i publikacji pewnych współdzielonych obiektów.

3.4.5 Sprzeczność wymagań

Przedstawione wymagania można podzielić na trzy grupy: bezwzględnie obowiązujące, takie dla których stopień realizacji podlega ocenie, która nie może być ścisła (by nie stwierdzić, że stopień realizacji jest kwestią gustu oceniającego) i dodatkowe, których zrealizowanie nie jest obowiązkowe, ale może okazać się przydatne. Do pierwszej grupy zaliczyć można wymagania 5), 7), 8) i 9). W drugiej grupie znalazłyby się wymagania 1) – 4) i 6). Wreszcie wymaganiem z trzeciej grupy będzie 10).

Wymagania z drugiej grupy, jako „płynne" będą podlegały trade-offom między sobą a także niekiedy ich zrealizowanie w dostatecznym stopniu będzie kolidowało z wymaganiami z grupy pierwszej. Kilka z takich konfliktów przedstawiono poniżej. Zapewne z powodu ich istnienia trudno jest po dziś dzień wskazać istniejący już pełnowymiarowy benchmark dla rozproszonych pamięci transakcyjnych.

Sprzeczność wymogu 9) (przewidywalność access-setów) z 3) (*depth*). Generalnie występowanie dużych transakcji (w sensie liczby operacji, czasu wykonania, ale przede wszystkim

wielkości access-setów[10]) przeciwstawić można możliwości przewidzenia zawartości access-setów *a priori*. Nawet wdrażając obejście tego problemu przedstawione w opisie kryterium 9) należy pamiętać, że im większy access-set transakcji tym większe prawdopodobieństwo, że ulegnie on zmianie pomiędzy konstrukcją a rozpoczęciem transakcji, tak że jego weryfikacja zmusi transakcję do restartu. Pogodzenie przewidywalności access-setów z różnymi innymi wymogami będzie główną przeszkodą na drodze do zbudowania pełnowymiarowego benchmarka dla D-STM. Należy też powtórzyć, że znajomość acces-setów *a priori* (gdy nie stosujemy obejścia problemu) stwarza warunki do bezproblemowego użycia zamków drobnoziarnistych, które potencjalnie działać będą szybciej niż pamięć transakcyjna.

Sprzeczność wymogu 9) (przewidywalność access-setów) z 6) (realizm w ujęciu rozproszonym) i 4) (*breadth*). Przewidywalność access-setów będzie też kłócić się z rozproszoną naturą rozwiązywanego przez benchmark problemu. Nie udało się pośród benchmarków ze STAMPa znaleźć takiego konkretnego problemu, którego rozwiązanie wymagałoby użycia systemu rozproszonego i który jednocześnie pozwalałby na ustalenie zawartości access-setów zawczasu. Zestaw problemów generycznych (z transakcjami tworzonymi sztucznie), a więc takich, w oparciu o które można budować mikrobenchmarki, spełniających naraz oba wymagania najprawdopodobniej ogranicza się do wykorzystania architektury bazującej na rozproszonej tablicy haszowej.

Sprzeczność wymogu 9) (przewidywalność access-setów) z 8) (odróżnienie od mikrobenchmarka). Można sobie wyobrazić sposób obejścia problemu przewidywania access-setów poprzez wprowadzanie syntetycznych transakcji, których semantyka nie byłaby związana z rozwiązywanym problemem (o ile taki problem w ogóle by istniał – np. niepodzielne wykonanie 20 losowych modyfikacji współdzielonych struktur danych). Podejście takie znowu jednak zaczyna niebezpiecznie przypominać mikrobenchmark.

Sprzeczność wymogu 1) (prostota) z 6) (realizm w ujęciu rozproszonym). Z rozproszonym charakterem benchmarka kłóci się oczekiwana po nim prostota. Wspomniano już, że poziom skomplikowania problemu, którego rozwiązanie wymagałoby wykorzystania systemu rozproszonego w miejsce systemu wieloprocesorowego, nie jest bez wątpienia niski.

Sprzeczność wymogu 4) (*breadth*) z założeniem, że pojedynczy program ma spełniać wszystkie podane wymagania. Wreszcie niemożliwością będzie zapewnienie własności *breadth* w ramach pojedynczego programu wzorcowego. Jak wspomniano wcześniej, nawet jeżeli program taki miałby składać się z szeregu modułów przetwarzających dane w różny sposób (np. ułożonych w kroki potoku), różnorodność sposobów wykorzystania pamięci transakcyjnych przez programy pochodzące z różnych dziedzin informatyki raczej wyklucza możliwość korzystania przez nie z tej samej puli i struktury danych.

[10] oczywiście nie można we wszystkich przypadkach postawić znaku równości pomiędzy długością transakcji a wielkością jej access-setu. Można bowiem wyobrazić sobie transakcję, która operując na relatywnie małym zbiorze obiektów wykonuje dużą liczbę operacji. Niech przykładem będzie tu niepodzielne sortowanie listy jednokierunkowej. Nie zmienia to faktu, że by dostatecznie utrudnić pamięciom transakcyjnym działanie, potrzebne są transakcje długo operujące na dużych access-setach, a z reguły rozmiar access-setu będzie jednak wprost proporcjonalny do długości działania transakcji.

Sprzeczność wymogu 7) (możliwość wykonania pomiarów) z 6) (realizm w ujęciu rozproszonym) oraz z przyczyną wprowadzenia wymogu 10) (prywatyzowanie danych). Osobną kwestią będzie połączenie rozproszonego charakteru programu wzorcowego z możliwością wykonania pomiarów. Jak wspomniano wcześniej, stan lokalny nie zawsze może wystarczać do obliczenia potrzebnych metryk. Przyjęto sposób radzenia sobie z tym problemem poprzez podglądanie wewnętrznych operacji pamięci transakcyjnej, która dane niezbędne do obliczenia metryk może sprowadzać z innych węzłów na własne potrzeby. Wymusza to znajomość budowy systemu pamięci transakcyjnej (w tym algorytmów TM), co wymaga możliwości wglądu do jej kodu źródłowego. Dodatkowo zarejestrowanie wystąpienia zdarzeń wewnątrz transakcji może być szczególnie trudne w przypadku systemów automatycznie instrumentujących kod transakcyjny, ponieważ faktyczna realizacja transakcji może różnić się od tego co programista umieścił w kodzie źródłowym.

3.5 Analiza przydatności benchmarków z narzędzia STAMP

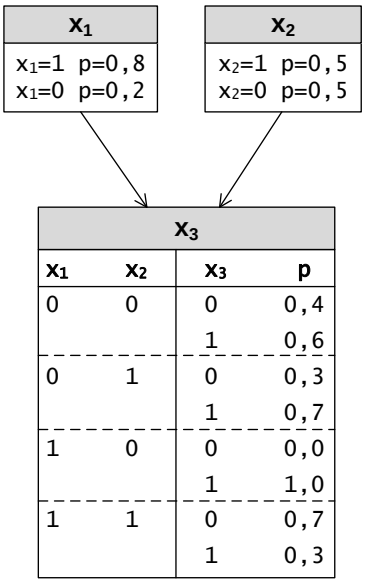
3.5.1 Rozproszona wersja *bayes*

Benchmark *bayes* jest dość skomplikowanym programem, zarówno w zakresie semantyki problemu jak i implementacji rozwiązania. Opis podany w punkcie 2.6.1.1 wymaga rozszerzenia.

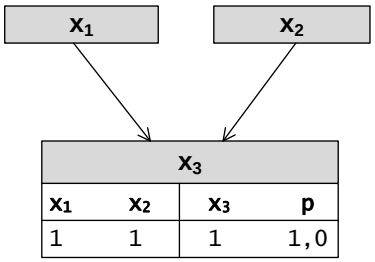
Na wysokim poziomie abstrakcji algorytm *hill-climbing* (algorytm iteracyjnego ulepszania rozwiązania początkowego poprzez wprowadzanie do niego pojedynczych zmian) można wyrazić następująco:

1. Początkowe rozwiązanie – sieć bayesowska bez żadnych krawędzi – jest oceniana względem zbioru rekordów przy użyciu pewnej funkcji oceniającej (ang. *scoring function*).
2. Dla pewnej zmiennej losowej a:
   (a) Obierz pewną zmienną a' taką, że $a' \neq a$ i a' nie jest bezpośrednim ani pośrednim następnikiem a (nie istnieje ścieżka skierowana od a do a'). Rozważ możliwość wstawienia krawędzi $a' \rightarrow a$ poprzez ocenę sieci z wprowadzoną zmianą. Powtórz operację dla każdej możliwej zmiennej a'.
   (b) Obierz pewną zmienną a'' spośród poprzedników (rodziców) a. Rozważ możliwość usunięcia krawędzi $a'' \rightarrow a$ i możliwość odwrócenia krawędzi: $a'' \leftarrow a$. Powtórz operację dla każdego rodzica a'' zmiennej a.
3. Każda z operacji z punktu 2. jest tymczasowo stosowana do sieci by umożliwić ocenę. Jeżeli operacja o najwyższej ocenie pozwoli na poprawienie aktualnej oceny całej sieci, jest ona stosowana na stałe, po czym sieć oceniana jest jeszcze raz.
4. Wróć do punktu 2. przyjmując za a kolejną zmienną losową.

Dwa problemy pojawiają się w przypadku przedstawionego algorytmu: 1) jak ocenić adekwatność skonstruowanej sieci bayesowskiej do zbioru rekordów (najlepiej jeszcze tak, by można było w podobny sposób dokonać wyboru najlepszej operacji modyfikacji krawędzi na każdym kroku)? oraz jak reprezentować zbiór rekordów tak, by ewentualna funkcja oceniająca



(a) Standardowa tabela prawdopodobieństwa.



(b) Tabela implementowana w benchmarku *bayes*.

Rysunek 3.11: Tabele prawdopodobieństwa w sieci bayesowskiej.

sieć pozwalała na szybkie wyznaczenie oceny?

Funkcja oceniająca sieć bayesowską. Należy przypomnieć, że *bayes* zakłada, że zmienne losowe są binarne. Dodatkowo krawędzie sieci bayesowskiej nie są w tym benchmarku etykietowane prawdopodobieństwami. Podczas gdy w typowej sieci bayesowskiej zależność pomiędzy wartością węzła a wartością poprzedników wyrażana jest w postaci przechowywanej w węźle-dziecku tabeli prawdopodobieństwa, w której wiersze odpowiadają na pytanie: z jakim prawdopodobieństwem węzeł przyjmie określoną wartość przy założeniu określonego układu wartości rodziców, *bayes* wykorzystuje podejście, w którym układ wartości rodziców determinuje wartość węzła w sposób pewny, co jest mniej lub bardziej adekwatne do zbioru obserwacji. Porównanie podejść przedstawia rysunek 3.11.

W roli funkcji oceniającej *bayes* stosuje funkcję nazwaną w [17] *log likelihood (of network adequacy to set of records)*. Wyraża się ona wzorem:

$$L(network, records) = -N_{params}\frac{\log R}{2} + R\sum_{a_j \in V}\sum_{asgn \in X_j}\sum_{v \in \{0,1\}} P(a_j = v \wedge asgn)\log P(a_j = v|asgn)$$

gdzie:
N_{params} – to liczba parametrów w sieci bayesowskiej. Bez kary za wstawienie do sieci dodatkowej zależności, krawędzie byłyby do niej wstawiane w sposób nieograniczony. Zazwyczaj wartość ta jest równa sumarycznej liczbie wierszy w tabelach prawdopodobieństwa ze wszystkich węzłów sieci. Ponieważ *bayes* nie utrzymuje tabel prawdopodobieństwa, N_{params} oznacza w nim liczbę wszystkich krawędzi sieci – liczbę wszystkich rodziców,

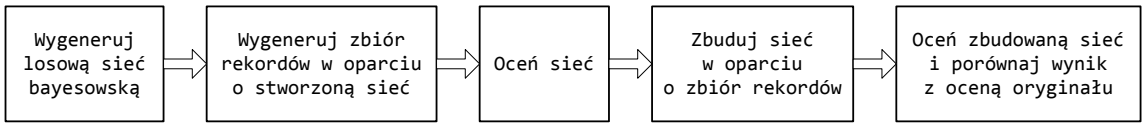
a_j – to j-ta zmienna losowa,
V – to zbiór wszystkich zmiennych losowych,
X_j – to zbiór stanów rodziców zmiennej a_j a *asgn* to pojedynczy stan rodziców (zestaw przypisanych im wartości),
R – to liczba wszystkich elementów w zbiorze rekordów.

Prawdopodobieństwo $P(a_j|asgn)$ jest parametrem zbioru rekordów, dostatecznie dobrze przybliżanym przez $\frac{C(a_j=v \wedge asgn)}{C(asgn)}$, gdzie $C(x)$ oznacza liczbę rekordów pasujących do zapytania (*conjunctive query*) x. Podobnie parametrem zbioru rekordów jest R. Parametrami związanymi z ocenianą siecią są: $P(a_j|asgn)$ i X_j dla każdej zmiennej a_j z V. Nie jest natomiast jasne czy prawdopodobieństwo $P(a_j = v \wedge asgn)$ ma być parametrem sieci, pochodzącym z tabeli prawdopodobieństwa w węźle a_j (zatem dla *bayes* zawsze będzie przyjmować 1), czy też ma być przybliżane w oparciu o zawartość zbioru rekordów przez $\frac{C(a_j=v \wedge asgn)}{R}$. Fakt, że, gdyby pominąć ten parametr, w powyższym wzorze brakowałoby odniesienia do zawartości zbioru rekordów wskazuje na drugą możliwość.

Należy podkreślić, że przy znajomości parametrów sieci z pewnej chwili, takich jak jej aktualna ocena i liczba rodziców można bez odwoływania się do niej ocenić poszczególne operacje modyfikacji krawędzi (dla wstawienia krawędzi między rodzicem a' a zmienną a zmienia się ocena a – składnika sumy we wzorze – i wzrasta o 1 liczba rodziców; dla usunięcia – analogicznie; dla odwrócenia – zmienia się ocena a i jej byłego rodzica, teraz dziecka – a'). Stąd jeżeli dla każdej zmiennej pamiętany będzie odpowiadający jej składnik oceny sieci, całkowita ocena sieci i liczba rodziców, wszystkie z chwili wprowadzenia ostatniej zmiany, nie trzeba będzie wprowadzać zmian do sieci, by wybrać najlepszą na danym kroku przetwarzania modyfikację.

Reprezentacja zbioru rekordów. W zakresie reprezentacji zbioru rekordów jasne jest, że będzie istniał wymóg udzielenia, w oparciu o nią, stosunkowo szybko, odpowiedzi na pytanie: ile rekordów ze zbioru spełnia warunek wyrażony w postaci szeregu porównań wartości połączonych operatorem „i" (ang. *conjunctive queries*), np. w ilu przypadkach $a_1 = 0$, $a_2 = 1$ i $a_3 = 1$, a a_4 i a_5 przyjmują dowolne wartości. Reprezentacją taką są np. drzewa decyzyjne (ang. *alternating decision trees, ADTrees*). *bayes* konstruuje ich wariant na drugim poziomie optymalizacji (*sparse ADTrees without leaf-node lists*, por. [46]).

Jest to dobry moment, by określić poszczególne kroki przetwarzania w ramach benchmarka. Te zostały przedstawione na rysunku 3.12.



Rysunek 3.12: Poszczególne kroki przetwarzania w benchmarku *bayes*.

Równoległa wersja benchamrka. Ideą zrównoleglenia przetwarzania w *bayes* jest to, by każdy wątek na drugim kroku algorytmu przetwarzał pewien wycinek (ang. *shard*) zbioru zmiennych V.

Dla każdej takiej zmiennej (w połączeniu z rodzicem – rzeczywistym lub potencjalnym) dodanie i usunięcie rodzica oraz odwrócenie relacji tworzą zbiór elementarnych operacji ulepszających sieć bayesowską. Jak wspomniano wcześniej, ocena takich operacji może być wyznaczona bez potrzeby przetwarzania całej sieci. Należy jednak zapewnić, by wybrana najlepsza operacja nie spowodowała wystąpienia w sieci cyklu. Stąd dostęp do całej sieci (tylko do odczytu) będzie wymagany, ale tylko przy wdrażaniu operacji.

Wynik produkowany przez *bayes* jest wynikiem przybliżonym. Stąd nie ma znaczenia, kiedy dokładnie operacja polepszająca sieć zostanie wdrożona (względem momentu jej wyznaczenia). Innymi słowy jeżeli na pewnym kroku przetwarzania najlepsza operacja modyfikująca sieć będzie wyznaczona, to nie musi być ona wdrożona zaraz. Nawet wdrożona nieco później powinna podnosić ocenę całej sieci w nie gorszy sposób. W *bayes* istnieć będzie kolejka modyfikacji sieci, wyznaczonych jako najlepsze, ale jeszcze nie wdrożonych (i nie sprawdzonych pod kątem tego czy zamykają jakiś cykl).

Na pierwszym etapie wykonania każdy wątek otrzymuje kilka zmiennych do przetworzenia. Dla każdej takiej zmiennej jej obecny stan (brak rodziców) jest oceniany względem zbioru rekordów (obliczany jest składnik sumy odpowiadający tej zmiennej; suma będzie później składnikiem oceny sieci, L). Wyniki zebrane dla całej sieci pozwolą ustalić podstawę oceny (ang. *base log likelihood*), względem której sieć będzie poprawiana przez wstawianie krawędzi (początkowo; później także ich usuwanie i odwracanie ich kierunku).

Następnie dla każdej zmiennej przypisanej wątkowi i dla każdego jej potencjalnego rodzica (ponieważ początkowo sieć nie ma krawędzi, rodzicem może być każda inna zmienna) obliczany jest lokalny zysk wynikający z wstawienia pomiędzy nimi krawędzi. W ramach zmiennej wybierana jest operacja o największym dodatnim zysku (tj. operacja wstawienia rodzica, która najmocniej podniesie ocenę zmiennej) i ta właśnie ląduje w kolejce zadań. Dodawanie zadań do kolejki jest realizowane wewnątrz transakcji. Gdyby zadania były wdrażane natychmiast po wyznaczeniu i bez odpowiedniej weryfikacji, mogłoby to doprowadzić do powstania w sieci cykli. Stąd potrzeba ich kolejkowania.

W drugim etapie każdy wątek pracuje nad pojedynczym zadaniem z kolejki. W ramach jednej transakcji pobiera takie zadanie, a w ramach kolejnej sprawdza czy wdrożenie zadania nie doprowadzi do powstania cyklu. Jeżeli nie ma takiego ryzyka, wątek wdraża zmianę wewnątrz tej samej transakcji. Kolejnym krokiem jest obliczenie nowych składników oceny całej sieci. W zależności od typu zadania (na tym etapie są to tylko wstawienia krawędzi) modyfikowana jest wartość licznika wszystkich rodziców w sieci, oraz obliczana jest nowa ocena pojedynczej zmiennej[11]. Sumaryczna ocena całej sieci może być w łatwy sposób uaktualniona (transakcyjnie) poprzez modyfikację składnika reprezentującego liczbę wszystkich rodziców (N_{params}) oraz odjęcie starej oceny j-tej zmiennej w postaci: $R\sum_{asgn\in X_j}\sum_{v\in\{0,1\}} P(a_j = v \wedge asgn)\log P(a_j = v|asgn)$ i dodanie nowej, wyrażonej następująco: $R\sum_{asgn\in X'_j}\sum_{v\in\{0,1\}} P(a_j = v \wedge asgn)\log P(a_j = v|asgn)$.

Po uaktualnieniu składników oceny sieci możliwe jest wyznaczenie kolejnej operacji, która

[11] warto wspomnieć, że ocena pojedynczej zmiennej jest elementem współdzielonym. W przypadku gdy później odwracany będzie kierunek krawędzi, zmianie ulegnie nie tylko ocena byłego dziecka, ale też i byłego rodzica. O ile ocena byłego dziecka może być modyfikowana jedynie przez bieżący wątek, o tyle były rodzic może znajdować się w gestii innego wątku. Stąd operacja uaktualnienia oceny pojedynczej zmiennej musi być przeprowadzana transakcyjnie.

w możliwie największym stopniu przyczyni się do polepszenia tej oceny. Odbywa się to zgodnie z algorytmem przedstawionym na początku tego punktu. Sam proces jest podobny do początkowego zapełniania kolejki zadań, przy czym transakcją objęty jest cały punkt 2. Oczywiście stan sieci, na której operuje wątek zmienił się. Sieć nie jest już pusta – zawiera pewne krawędzie i w związku z tym ma inną, wyższą ocenę, nad dalszą poprawą której wątek pracuje.

W oparciu o analizę kodu źródłowego określono następujący przybliżony algorytm działania wątku w odniesieniu do zakolejkowanych modyfikacji: ponieważ wątek dla każdej przetwarzanej zmiennej wyznacza jedno zadanie (dodania pewnego rodzica, usunięcia pewnego rodzica lub zmiany relacji z pewnym rodzicem) i po przetworzeniu wszystkich należących do niego zmiennych nie kończy pracy (ponownie przetwarza pierwszą, drugą, trzecią, itd. tak długo aż dla żadnej z nich nie da się wyznaczyć modyfikacji polepszającej ocenę sieci), po zakończeniu przetwarzania pojedynczej zmiennej, a przed rozpoczęciem przetwarzania następnej wykonuje weryfikację (pod kątem wprowadzania cykli) i wdraża zmiany odczytane z kolejki zadań. Wykonuje tę operację tak długo, aż kolejka nie zostanie opróżniona. Należy zaznaczyć, że współbieżnie pobierać zadania z kolejki mogą też inne wątki.

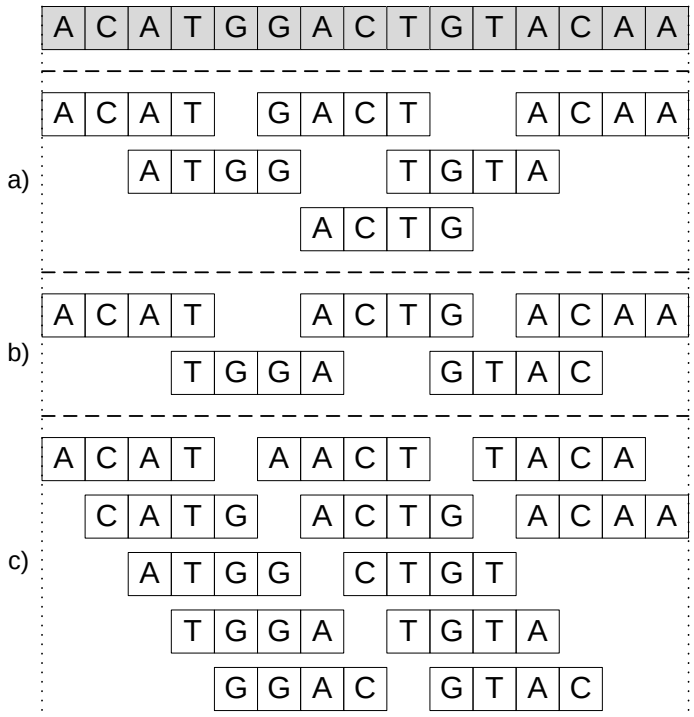
Pomimo tego, że sposób rozwiązania problemu konstrukcji sieci bayesowskiej implementowane przez *bayes* jest złożony, udało się go odtworzyć w stopniu zapewne dostatecznym, by podjąć próbę jego przeniesienia do środowiska rozproszonego. W zakresie motywacji takiego działania można twierdzić, że wolumeny danych przetwarzane podczas konstrukcji sieci bayesowskiej mogą być na tyle duże, że do ich obróbki nie wystarczy pojedynczy, nawet wieloprocesorowy węzeł większego systemu komputerowego. Rozproszenie centralnej struktury danych nie wydaje się też trudne z racji jej rzadkich modyfikacji i niezbyt gęstej sieci połączeń pomiędzy elementami (w przeciwieństwie np. do *labyrinth*). Wreszcie charakterystyka transakcji pozwala sądzić, że nie będzie znacznym problemem wyznaczanie zawartości access-setów statycznych transakcji *a priori*. Transakcje albo operują na pojedynczych wartościach (modyfikacja oceny pojedynczej zmiennej, modyfikacja oceny całej sieci), albo na kolejce zadań, albo też są używane do wprowadzania modyfikacji do sieci przy wcześniejszym zapewnieniu że nie doprowadzi to do powstania cyklu. O ile przewidywalność access-setów dla wszystkich typów poza ostatnim jest oczywista, o tyle transakcje ostatniej klasy wykonują dużo odczytów i mało zapisów. Są to warunki dobre do wprowadzenia obejścia problemu wyznaczania access-setów opisanego w punkcie 3.4.4.

Podsumowując należy stwierdzić, że *bayes* będzie dobrym kandydatem na benchmark dla rozproszonych pamięci transakcyjnych. Z wyjątkiem wymogu prostoty, który spełnia w stopniu dopuszczającym, w pełni odpowiada on kryteriom określonym w sekcji 3.4.

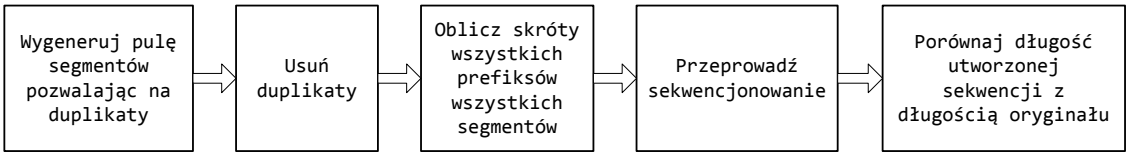
3.5.2 Rozproszona wersja *genome*

Opis programu przedstawiony w punkcie 2.6.1.2 wymaga uzupełnienia. W pierwszej kolejności należy graficznie przedstawić relację pomiędzy sekwencją a segmentami. Czyni to rysunek 3.13.

Po drugie, należy jeszcze raz przedstawić pojawiające się tam etapy przetwarzania. To z ko-



Rysunek 3.13: Segmenty wygenerowane z sekwencji: a) pewna liczba segmentów, b) minimalna liczba segmentów, c) maksymalna liczba segmentów.



Rysunek 3.14: Główne etapy przetwarzania w *genome*.

lei czyni rysunek 3.14 Generowanie instancji nie zostało w oryginale zrównoleglone. Usuwanie duplikatów odbywa się poprzez wstawienie wszystkich wygenerowanych segmentów do współdzielonej tablicy haszowej przy użyciu funkcji tożsamościowej w roli funkcji skrótu. Każdy wątek przetwarza na tym etapie w przybliżeniu równą liczbę segmentów. Transakcją objęte są grupy wstawień o stałej liczności (domyślnie 12 wstawień na transakcję).

Na kolejnym etapie dla każdego segmentu do odpowiednich tablic haszowych wstawiane są pary `(skrót_początku, segment)`. *genome* wykorzystuje schemat Rabina-Karpa wyszukiwania wzorców w ciągach znaków. Elementem tego schematu jest specyficzna funkcja skrótu (np. *sdbm* [59]), która dla podanego skrótu ciągu n znaków i znaku z pozycji $n+1$ potrafi niskim kosztem obliczyć skrót całego ciągu $n+1$ znaków. Schemat ten określa postępowanie wątku obliczającego skróty prefiksów segmentów. Pojedynczy wątek przetwarzając pojedynczy segment o długości x, oblicza i kolejno wstawia do odpowiednich tablic skróty prefiksów segmentu o długościach od 1 do $x-1$. Ponieważ wynikiem przetwarzania na tym etapie jest dwuwymiarowa tablica postaci:

```
segments[prefix_length][prefix_hash],
```

a dokładnie mapa:

```
prefix_length → hashtable(prefix_hash → segment),
```

wątek przetwarzając jeden segment odwołuje się do różnych tablic haszowych. Transakcja na tym etapie obejmuje pojedynczą operację wstawienia skrótu do tablicy haszowej.

Sama tablica haszowa ma postać uporządkowanego zbioru (implementowanego przez standardową tablicę) mniejszych jednostek, *bucketów*, których liczba jest stała. Przy wstawianiu pary `(klucz, wartość)`, skrót klucza jest dzielony *modulo* przez liczbę bucketów, tak by znaleźć odpowiedni element zbioru. Element ten przechowuje listę linkowaną par. STAMP dostarcza implementację tablicy haszowej, w której możliwe jest istnienie na takiej liście dwóch par o tym samym kluczu (równoważne istnieniu duplikatów w całej strukturze danych). Ze względu na drugi etap przetwarzania *genome*, funkcja ta ostaje w jego przypadku wyłączona.

Wreszcie sam algorytm sekwencjonowania sprowadza się do poszukiwania przez każdy z wątków dopasowania do segmentu, za który jest odpowiedzialny (według algorytmu przedstawionego na rysunku 3.15). Poszukiwanie to zorganizowane jest w tury, z których każda odpowiada długości nakładających się części segmentów (długość ta nazywana będzie *długością overlapu*) i kończy się globalną barierą synchronizacyjną, przy czym tury posortowane są według długości overlapów malejąco. Ma to za zadanie doprowadzić do sytuacji, w której najlepsze dopasowania zostaną wychwycone jako pierwsze.

W ujęciu przybliżonym algorytmu sekwencjonowania użyty w *genome* działa w następujący sposób:

- Istnieje współdzielona pula segmentów niedopasowanych, w której początkowo znajdują się wszystkie segmenty.
- Każdy wątek otrzymuje grupę segmentów, za które będzie odpowiedzialny tak, by każdy segment był przypisany tylko do jednego wątku.
- W ramach tury (z przypisaną długością overlapu) dla każdego z przypisanych wątkowi segmentów poszukiwane jest dopasowanie. Wpierw oblicza się skrót sufiksu o odpowiedniej długości. Następnie poszukuje się w odpowiedniej tablicy haszowej pasującego skrótu prefiksu o tej samej długości. W przypadku gdy skrót taki zostanie odnaleziony, sprawdza się czy pod pasującymi skrótami kryją się faktycznie pasujące wartości oraz czy segment powiązany ze skrótem prefiksu cały czas należy do puli segmentów niedopasowanych. Gdyby okazało się to prawdą, wątek „zszywa" oba segmenty (terminologia ta będzie stosowana w przypadku rozproszonej wersji benchmarka) i usuwa drugi z puli niedopasowanych. Od tej pory nie będzie się już zajmował pierwszym segmentem (a drugi wciąż posiada wątek za niego odpowiedzialny). Wszystkie operacje na segmencie, od momentu znalezienia skrótu do zszycia, wykonywane są w ramach pojedynczej transakcji.

Algorytm taki może powodować powstawanie cykli w sekwencji wynikowej. W ogólności nic nie stoi na przeszkodzie by, gdy wątek T_1 zszyje segment S_1 z S_2, wątek T_2 zszył segment S_2 z S_1. Można sobie wyobrazić kilka sposobów rozwiązania tego problemu:

```
// testuj overlapy w kolejności malejących długości
for (int overlap = segment_length - 1; overlap > 0; overlap--) {
    // dla każdego overlapu próbuj znaleźć dopasowanie do każdego posiadanego
    // segmentu
    for (Segment end_to_match in owned_segments_collection) {
        // oblicz skrót końca posiadanego segmentu
        int end_hash = hash(end_to_match.suffix(overlap));

        // wyszukaj za pomocą skrótu segmenty o potencjalnie pasujących początkach
        List<Segment> starts_matched = prefix_hashtables[overlap].get(end_hash);

        // dla każdego znalezionego potencjalnego dopasowania, zbadaj je
        for (Segment start_matched in starts_matched) {
            // sprawdź, czy pod skrótami kryją się pasujące wartości
            if (end_to_match.suffix(overlap) != start_matched.prefix(overlap)) {
                continue;
            }

            // niepodzielnie usuń segment z puli segmentów niedopasowanych
            remove_from_unmatched_segments_pool(start_matched);

            // połącz segmenty
            end_to_match.append(start_matched);

            // nie potrzebujemy już innego dopasowania start_matched do posiadanego
            // segmentu end_to_match
            break;
        }
    }

    // przejdź do sprawdzania następnej długości overlapu tylko wtedy gdy wszystkie
    // testy aktualnej długości w systemie już się zakończyły
    barrier();
}
```

Rysunek 3.15: Algorytm sekwencjonowania w *genome* dla pojedynczego wątku.

1. Po zakończeniu każdej tury powstałe cykle są wyszukiwane i rozcinane.
2. Przed rozpoczęciem przetwarzania każdy wątek otrzymuje tylko jeden segment, który jest usuwany z puli segmentów niedopasowanych. Konstruuje wtedy jedną, własną pod-sekwencję. Po zakończeniu każdej tury, pod-sekwencje wszystkich wątków próbuje się łączyć zszywać między sobą.
3. Wszystkie segmenty posiadają identyfikator pod-sekwencji, początkowo inny dla każdego segmentu. Gdy dwa segmenty są ze sobą zszywane, obu nadaje się ten sam identyfikator. Dodatkowo jeżeli któryś ze zszywanych segmentów jest już elementem pewnej pod-sekwencji, zmianie ulegają identyfikatory wszystkich obecnych w niej segmentów. Przed zszyciem dwóch segmentów wątek upewnia się, że identyfikatory sekwencji dla obu z nich są różne.

Oryginalna implementacja *genome* wykorzystuje pierwszy z nich. Po zakończeniu każdej tury pojedynczy wątek analizuje wykonane zszycia i zatwierdza te, które nie zapętlają sekwencji wynikowej. Ze względu na możliwość powstania wąskiego gardła przetwarzania nie jest to podejście odpowiednie dla systemów rozproszonych Stąd w rozproszonej wersji *genome* wykorzystywany będzie sposób 3., który dodatkowo wzbogaci charakterystykę generowanych

przezeń transakcji.

genome dobrze nadaje się do rozproszenia z kilku powodów. Po pierwsze wolumeny danych przetwarzane przez aplikacje z dziedziny bioinformatyki każą sądzić, że podobny algorytm mógłby być w praktyce wykonywany w środowiskach rozproszonych. Po drugie połączenia między współdzielonymi obiektami (bucketami tablic haszowych oraz segmentami) są na tyle luźne, by można było myśleć o łatwym rozproszeniu struktur danych. Z drugiej strony elementem sekwencjonowania będzie dość często przeglądanie listy (np. przy zmianie numeru pod-sekwencji). Optymalne zastosowanie w tym miejscu statycznej pamięci transakcyjnej będzie kłopotliwe (można oczywiście myśleć o włączeniu do access-setu całej puli segmentów, ale w punkcie 3.4.4 podano potencjalnie lepszy sposób wyznaczania access-setów transakcji *a priori*). Po trzecie wreszcie, algorytm wydaje się na tyle prosty, by mógł pełnić rolę benchmarka dla pamięci transakcyjnych o różnych interfejsach programistycznych dla różnych języków programowania.

3.5.3 Rozproszona wersja *intruder*

„Design 5" systemu NIDS, implementowany przez benchmark *intruder* został przedstawiony na rysunku 3.16.

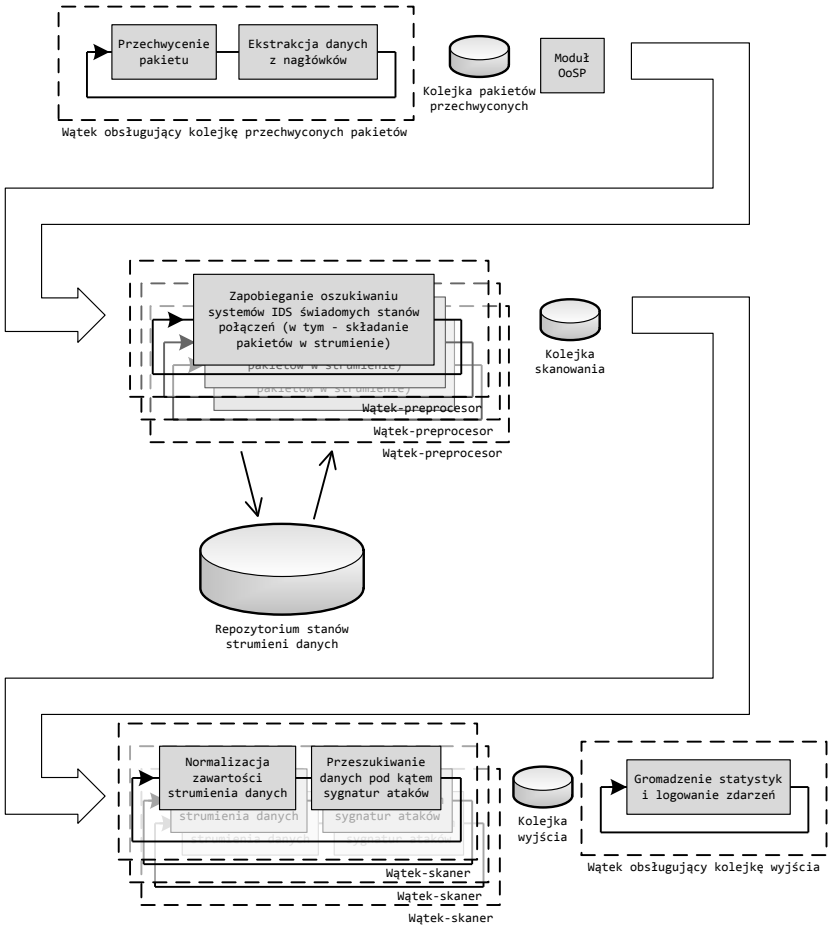
Pewien zewnętrzny (względem NIDS) komponent umieszcza przechwycone pakiety w kolejce wejściowej (ang. *capture queue*). Pojedynczy wątek odpowiada za wstępne przetworzenie tej kolejki – odczytuje dane z nagłówków pakietów i umieszcza je w tej samej kolejce jako metadane jej elementów.

Algorytmy dopasowywania sygnatur bazujące na stanach połączeń wymagają, by pakiety były przez NIDS przetwarzane w określonej kolejności (wynikającej raczej z numerów sekwencyjnych pakietów w ramach połączeń niż z kolejności ich przechwycenia). Jeżeli kolejny etap – poza ponownym składaniem pakietów w strumienie danych przesyłane w ramach połączeń – wykonywałby pewne operacje bazując na stanach powstałych w wyniku odebrania pakietów (np. wstępne ustalanie poprawności *TCP three-way handshake*), należałoby zapewnić określony porządek przetwarzania elementów. Stąd przy pobieraniu pakietów razem z ich metadanymi odczytanymi z nagłówków z kolejki pojawia się moduł OoSP (*Order of Seniority Processing*). W przypadku implementacji proponowanej w [28] zapewnia on, że na następnym etapie żadne dwa wątki nie przetwarzają równolegle dwóch pakietów należących do tego samego strumienia danych.

Kolejny etap sprowadza się do do składania pakietów w strumienie danych (ang. *flows*). Pakiety, które tworzą niekompletny fragment strumienia danych lądują w repozytorium. Gdy tylko pojawi się ostatni pakiet, którego brakuje do kompletu, strumień jest składany i umieszczany w kolejce skanowania (ang. *matching queue*).

Wreszcie kilka współbieżnych wątków wybiera złożone fragmenty strumieni z kolejki skanowania i po normalizacji zawartości przeszukuje je pod kątem występowania sygnatur ataków.

Benchmark *intruder* upraszcza implementowany schemat systemu NIDS. Pierwszym etapem przetwarzania jest wygenerowanie instancji: określonej liczby pakietów podzielonych na



Rysunek 3.16: Architektura „design 5" równoległego systemu NIDS. Opracowanie własne na podstawie: [28].

zadaną liczbę strumieni danych i z określonym procentowym udziałem porcji danych pasujących do sygnatur ataków. Następnie uruchamia się szereg równoległych wątków, które odpowiadają za złożenie strumieni, normalizację ich zawartości i wykrycie ataków. Pojedynczy wątek działa według następującego algorytmu:

1. Pobierz kolejny pakiet z wejścia (przy użyciu pojedynczej transakcji). Zakończ wątek jeżeli w kolejce wejściowej nie ma już żadnych pakietów.
2. Dodaj pakiet do repozytorium stanów. Repozytorium takie to zbiór par `(klucz, wartość)`, w którym kluczem jest identyfikator strumienia danych, a wartością lista dotychczas wstawionych do repozytorium pakietów należących do tego strumienia. Pojedyncza operacja wstawienia objęta jest transakcją. Operacja taka może jednak wyzwolić proces składania pakietów w strumień. Proces składania zachodzi w ramach tej samej transakcji co dodanie pakietu.
3. Jeżeli wstawienie pakietu do repozytorium spowodowało pojawienie się nowego gotowego strumienia, to należy go pobrać z kolejki skanowania. Każde pobranie objęte jest osobną transakcją. W tym kroku następuje próba pobrania tego strumienia, do którego należał pakiet wstawiany do repozytorium w kroku 2.

4. Znormalizuj strumień (np. poprzez zamianę wszystkich liter na małe lub usunięcie znaków niealfanumerycznych) i przeszukaj go pod kątem dopasowań do sygnatur ataków. Informacje o wykrytych atakach umieszczane są na liście wynikowej, lokalnej dla wątku. Ponieważ wątek posiada analizowany strumień na wyłączność, transakcje nie są tu używane.

Krok 2., który opcjonalnie może obejmować składanie pakietów we fragmenty strumienia danych wymaga podania dodatkowego opisu tego procesu:

1. Odczytaj z pakietu podstawowe informacje, takie jak identyfikator strumienia danych, numer części (numer sekwencyjny) i liczbę wszystkich części strumienia.
2. Przeanalizuj listę pakietów należących do tego samego strumienia, które do tej pory zostały wstawione do repozytorium.
    - Jeżeli lista jest pusta, oznacza to, że wszystkie dotychczas przybyłe pakiety zostały złożone w jeden strumień i trafiły do kolejki skanowania (identyfikator strumienia jest ponownie wykorzystywany). Dodaj pakiet do listy.
    - Jeżeli lista nie jest pusta, dodaj pakiet do listy, po czym sprawdź, czy zawiera ona tyle pakietów ile części zadeklarowano w nagłówku. Gdy warunek ten jest spełniony, pakiety należy złożyć w całość (kompletny strumień danych) i usunąć je z repozytorium. Powstałą porcję danych umieszcza się w kolejce skanowania, kolejce fragmentów do dalszego przetworzenia w kroku 3. algorytmu wątku.

Autorzy [28] twierdzą, że dzielenie ruchu sieciowego (ang. *traffic splitting*) na potrzeby przeskanowania go przez rozproszony system NIDS nie jest zadaniem trywialnym. Stwierdzenie to eliminuje możliwość skonstruowania rozproszonego systemu NIDS bez centralnego repozytorium stanów. Z drugiej strony wszystkie pakiety przechodzące przez skaner muszą się ostatecznie choć na chwilę w tym repozytorium znaleźć. Oznacza to, że gdyby było ono elementem systemu rozproszonego, brałoby udział w wielu zdalnych i potencjalnie długich transakcjach. O ile doprowadziłoby to do powstania ciekawej charakterystyki transakcyjnej, w praktyce byłoby zapewne wąskim gardłem przetwarzania. Niemniej jednak w ramach przygotowania tego raportu rozpatrywany był wariant benchmarka *scanner-as-a-service*, który dostarczałby funkcję skanowania ruchu sieciowego w postaci usługi, przy czym system rozproszony, w którym taka usługa działa dbałby o optymalne wykorzystanie własnych zasobów (motywacja rozproszenia NIDSa).

Należy zwrócić uwagę na charakterystykę transakcji z punktu 2. algorytmu wątku. Są one zazwyczaj krótkie, jednak jeżeli dojdzie do scalenia strumienia danych, ich read- i write-sety rozrastają się w znaczący sposób. W efekcie stanowiłyby wyjątkowo trudny przypadek dla statycznych pamięci transakcyjnych. Przy założeniu, że ten wzór zachowania będzie powtarzał się w wersji rozproszonej, *intruder* spełnia większość kryteriów określonych w sekcji 3.4. Oczekiwana prostota implementacji jest jednak tym kryterium dyskwalifikującym, które sprawia, że benchmark w wariancie rozproszonym nie zostanie opracowany.

3.5.4 Rozproszona wersja *kmeans*

Opis *kmeans* z punktu 2.6.1.4 nie wymaga znaczących uzupełnień. W zasadzie należałoby w tym miejscu dodać tylko jedną informację: wątki są tam tworzone jedynie na czas ustalania nowego przydziału elementów do klastrów. Innymi słowy po obliczeniu odległości każdego z przydzielonych mu elementów od środka każdego klastra i transakcyjnym dodaniu wartości każdego z jego wymiarów do odpowiedniej (przypisanej wymiarowi klastra) sumy wątek jest zatrzymywany (o ile rzecz jasna nie uda mu się pobrać kolejnej grupy elementów do przetworzenia z kolejki zadań), a podzielenie sum przez liczność klastra w celu obliczenia jego nowego środka odbywa się już sekwencyjnie.

Podobne podejście dla systemów rozproszonych jest nie do przyjęcia. Węzłów systemu nie powinno się włączać (i wyłączać) na żądanie, zwłaszcza w sytuacji, kiedy potrzeba takiej operacji nie wynika z kompensowania awarii ani potrzeby rozbudowania systemu. Nie należy ich też pozostawiać przez dłuższy czas bez zadania, bowiem oznacza to marnotrawstwo zasobów. Przeniesienie *kmeans* do środowiska rozproszonego oznaczałoby przetwarzanie ujęte w tury, z globalną barierą synchronizacyjną po każdym nowym przydziale elementów do klastrów.

W połączeniu z generowanymi przez wersję równoległą krótkimi transakcjami, perspektywy wykorzystania w systemie rozproszonym sposobu zrównoleglenia algorytmu, który wymaga częstych odwołań do globalnej bariery synchronizacyjnej, nie zachęcają do próby opracowania rozproszonej wersji *kmeans*.

3.5.5 Rozproszona wersja *labyrinth*

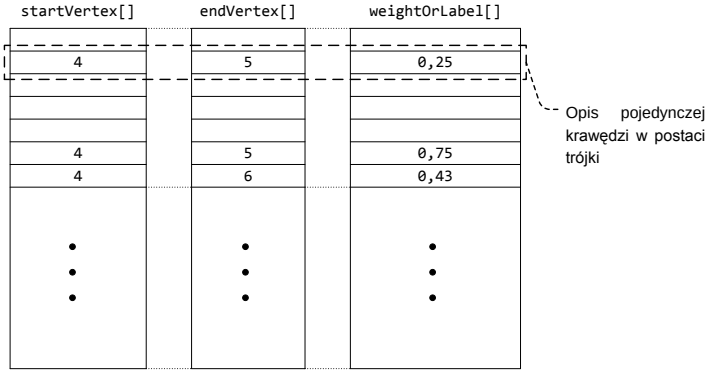
Opis problemu i algorytmu jego rozwiązania został umieszczony w punkcie 2.6.1.5. Zawiera on wszystko to, co udało się ustalić w oparciu o analizę kodu źródłowego. Formalnego ujęcia wymaga być może jeszcze implementowany przez *labyrinth* algorytm Lee. ostał on przedstawiony na rysunku 3.17.

Algorytm Lee stosowany jest w praktyce do wytyczania ścieżek przesyłających sygnały elektryczne w obwodach drukowanych (także wielowarstwowych). Programy pozwalające ręcznie wytyczać ścieżki, wyposażone ponadto w mechanizm automatyzacji tego procesu to typowi przedstawiciele narzędzi z grupy CAD (ang. *computer-aided design*). Jako takie zazwyczaj działają one raczej na wieloprocesorowych stacjach roboczych niż w systemach rozproszonych.

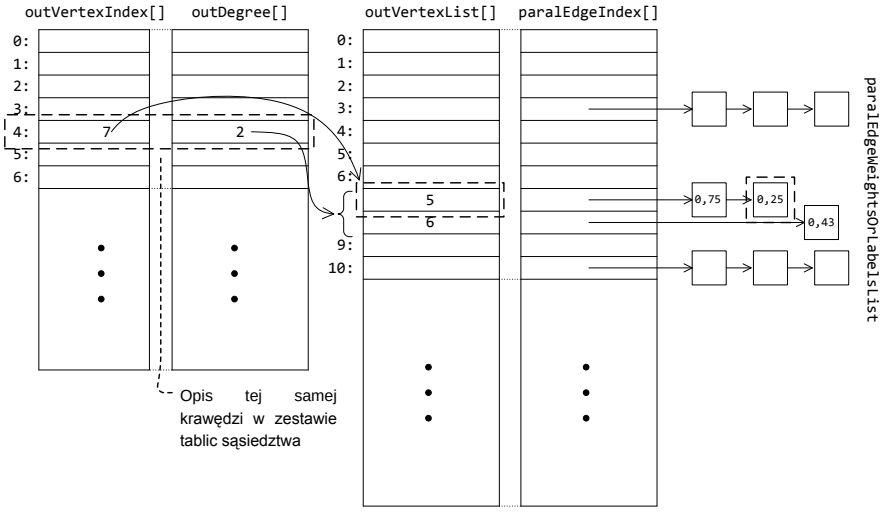
Oczywiście można mówić o rozmiarach instancji problemu na tyle dużych, że usprawiedliwiających użycie systemu rozproszonego. Należy jednak zwrócić wagę na to, że wykonanie algorytmu Lee w takim środowisku wiązałoby się z rozproszeniem centralnej struktury danych, labiryntu. Sam fakt rozproszenia, projektowanego od podstaw, w połączeniu z rozwiązaniem problemu trasowania żądań dostępu do poszczególnych części rozpraszanej struktury danych skomplikowałby benchmark i mógłby negatywne wpłynąć na jego charakterystykę transakcyjną (np. doprowadzając do wykonywania jedynie krótkich transakcji). Do tego należałoby jeszcze zapewnić możliwość przewidywania zawartości access-setów. W obliczu takiego poziomu komplikacji w połączeniu z niepewnym związkiem z systemami rozproszonymi, *labyrinth* nie będzie poddany próbie przeniesienia do środowiska rozproszonego.

```
// faza inicjalizacji - obranie punktu początkowego
Point point = START_POINT;
point.label = 0;

// faza zalewania labiryntu znacznikami (w praktyce - przeszukiwanie grafu wszerz)
Set<Point> justLabelled = new Set<Point>();
justLabelled.add(point);
int i = 1;
boolean endReached = false;
while (true) {
    Set<Point> newJustLabelled = new Set<Point>(); // sprawdź wszystkich sąsiadów
    for (Point point in justLabelled) {            // ostatnio sprawdzanych punktów
        for (Point neighbour in point.getNeighbours()) {
            if (neighbour.label != null) {  // sąsiad do oznaczenia nie może już
                continue;                   // posiadać znacznika
            }
            neighbour.label = i;
            newJustLabelled.add(neighbour);
            if (neighbour == END_POINT) {   // jeżeli osiągnięto punkt końcowy,
                endReached = true;          // zakończ przeszukiwanie
            }
            if (endReached) {
                break;
            }
        }
        if (end_reached) {
            break;
        }
    }
    if (end_reached) {
        break;
    }
    if (newJustLabelled.isEmpty()) {
        break; // nie ma już możliwości zajścia dalej
    }
    justLabelled = newJustLabelled;
    i++;

}

// konstrukcja ścieżki od końca
if (END_POINT.label != null) { // jeżeli poprzednio osiągnięto punkt końcowy
    List<Point> path = new List<Point>();
    path.add(END_POINT);
    for (int i = END_POINT.label; i > 0; i--) {
        for (Point neighbour in path.lastElement().getNeighbours()) {
            if (neighbour.label == i-1) {
                path.add(neighbour);
                break;
            }
        }
    }
    path.revert(); // odwróć kolejność elementów na ścieżce
}
```

Rysunek 3.17: Algorytm Lee implementowany przez *labyrinth*.



(a) Wejściowa struktura danych.



(b) Tablice sąsiedztwa i tablice pomocnicze.

Rysunek 3.18: Tablice: sąsiedztwa i pomocnicze jako struktura danych konstruowana przez *ssca2*.

3.5.6 Rozproszona wersja *ssca2*

Struktura danych konstruowana w *ssca2* przedstawiona została na rysunku 3.18. Dla wierzchołka `i` `outVertexIndex[i]` przechowuje miejsce w którym rozpoczyna się opis krawędzi wychodzących z wierzchołka, a `outDegree[i]` – długość opisu (stopień wyjściowy wierzchołka `i`). Dysponując takimi informacjami można odnaleźć w tablicy `outVertexList` indeksowanej wartościami z zakresu od `outVertexIndex[i]` do `outVertexIndex[i]+outDeggree[i]-1` informacje o tym jakimi wierzchołkami kończą się krawędzie wychodzące z `i`. Te same pozycje w tablicy `paralEdgeIndex` wskazują miejsce w tablicy pomocniczej lub dowolnej innej strukturze danych (na rysunku jest to zbiór list jednokierunkowych), z którego odczytać można wielość krawędzi (*ssca2* operuje na multigrafach, więc dwa wierzchołki mogą być połączone kilkoma krawędziami) i wagę lub etykietę każdej z nich.

Struktury `outVertexIndex`, `outDegree` i `outVertexList` to tablice sąsiedztwa (ang. *ad-*

jacency arrays), natomiast `paralEdgeIndex` oraz tablice zawierające informacje o wielości krawędzi i ich wagach lub etykietach to tablice pomocnicze (ang. *auxiliary arrays*).

Algorytm konstruowania tablic sąsiedztwa razem z powiązanymi tablicami pomocniczymi prezentuje rysunek 3.19. Warto wspomnieć, że aby ułatwić przetwarzanie, stanowiąca wejściową reprezentację grafu lista trójek `(wierzchołek_wejściowy, wierzchołek_wyjściowy, waga/etykie` jest sortowana.

Ponieważ tablice sąsiedztwa, reprezentujące następniki wierzchołków, nie pozwalają w łatwy sposób ustalić, czy określona krawędź nie istnieje w grafie, konstruuje się analogiczną reprezentację dopełnienia grafu (zestawu krawędzi nieobecnych[12] – ang. *implied edges*). O ile można twierdzić, że posortowanie danych podawanych na wejście programu usprawnia konstrukcję pierwszej ze struktur, o tyle przy drugiej znacznie większe będzie ryzyko wystąpienia konfliktów podczas dodawania tam informacji o poszczególnych nieistniejących krawędziach.

By dodatkowo utrudnić przetwarzanie *ssca2* konstruuje trzeci zestaw tablic sąsiedztwa, tym razem reprezentujący poprzedniki wierzchołków. Ten proces nie został ujęty w algorytmie na rysunku 3.19.

Z semantyki problemu w żaden sposób nie można wydostać informacji o granicach rozmiarów instancji. Jak najbardziej możliwe jest więc by przetwarzane grafy były tak duże, żeby musiały być przetwarzane w systemie rozproszonym. Z drugiej strony nie musi być tak zawsze, tj. nie wszystkie grafy przetwarzane przez *ssca2* będą odpowiednie dla takiego podejścia. Dodatkowo rozwiązywany problem jest zbyt ogólny, by można było go powiązać z systemami rozproszonymi w sposób inny niż bazując na rozmiarach instancji. Stąd niniejszy raport nie będzie nie będzie zawierała próby opracowania rozproszonej wersji tego benchmarka.

3.5.7 Rozproszona wersja *vacation*

Uzupełnieniem opisu z punkcie 2.6.1.7 niech będzie prezentacja sposobu przełożenia parametrów programu na generowane transakcje. Dla czterech ich typów, opisanych wcześniej:

1) *user tasks*,
2) *user deletions*,
3) *item additions*,
4) *item removals*,

[12] analiza algorytmu przedstawionego w [4] mogłaby sugerować inną definicję *implied edge*. Byłaby to krawędź, która istnieje w grafie, nazywana tak pod warunkiem, że nie istnieje w nim krawędź skierowana odwrotnie. Innymi słowy, jeżeli w grafie istnieje krawędź $u \rightarrow v$, ale nie istnieje krawędź $v \rightarrow u$, to ta pierwsza powinna znaleźć się na liście krawędzi już nie *nieistniejących* lecz po prostu *implikowanych* (por. [4, strona 9] oraz [4, algorytm 3] – algorytm przedstawiony w nieznacznie zmodyfikowanej formie na rysunku 3.19). Listy tak definiowanych krawędzi implikowanych miałyby być później wykorzystywane przez kernel 2. Termin *implied edge* nie znajduje się w powszechnym użyciu, co pozwala sądzić, że został wprowadzony na potrzeby specyfikacji, na której bazuje *ssca2*. Nie zmienia to faktu, że do ich reprezentacji cały czas wykorzystuje się listy sąsiedztwa.

```
1  // ustal stopnie wyjściowe wszystkich wierzchołków
2  for (int i = 0; i < startVertex.length /* NUM_EDGES */; i++) in parallel {
3      // współbieżna iteracja, osobna dla każdej krawędzi; w tym wariancie każda
4      // powinna być objęta transakcją
5      Vertex start = startVertex[i]; // jeden wierzchołek może być przetwarzany
6                                     // w różnych iteracjach; należy się zabezpieczyć
7                                     // przed takim duplikowaniem pracy
8      int startsOutDegree = 0;
9      {Wyszukaj wszystkie unikatowe krawędzie rozpoczynające się w wierzchołku start.}
10     {Ustal stopień wyjściowy wierzchołka start: startsOutDegree.}
11     outDegree[start.index] = startsOutDegree;
12     if (start.index == 0) {
13         outVertexIndex[start.index] = 0;
14     } else {
15         outVertexIndex[start.index] = outVertexIndex[start.index-1]
                   + outDegree[start.index-1]; // ustal outVertexIndex badanego
16                                             // wierzchołka (przy założeniu, że
17                                             // tablice outVertexIndex i outDegree
18                                             // inicjowane są wartościami 0);
19                                             // operacja konfliktowa
20     }
21
22     // zrób miejsce dla wierzchołka przesuwając outVertexIndex dla każdego
23     // wierzchołka o większym identyfikatorze o outDegree[start.index] komórek
24     for (int j = start.index+1; j < outVertexIndex.length /* NUM_VERTICES */; j++) {
25         outVertexIndex[j] += startsOutDegree; // operacja konfliktowa
26     }
27 }
28 barrier();
29 // UWAGA: w oryginalnym algorytmie wartości w tablicy outVertexIndex uaktualniane
30 // są po zakończeniu wykonywania pętli, w oparciu o kompletne wartości z tablicy
31 // outDegree. Eliminuje to potrzebę objęcia iteracji transakcją. Oryginalny
32 // algorytm byłoby jednak trudno przedstawić w tym miejscu. Poza tym celem te
33 // pracy jest właśnie pokazanie zastosowania dla transakcji w różnych algorytmach.
34
35 // skonstruuj listy sąsiedztwa
36 for (int i = 0; i < outVertexIndex.length /* NUM_VERTICES */; i++) in parallel {
37     Vertex start = getVertexByIndex(i);
38     for (int j = outVertexIndex[i]; j < outVertexIndex[i]+outDegree[i]; j++) {
39         Vertex end;
40         {Pobierz kolejną unikatową krawędź wychodzącą z wierzchołka start. Jej
           koniec zapisz w zmiennej end.}
41         outVertexList[j] = end.index;
42         {Utwórz strukturę danych przechowywaną w paralEdgeIndex[j].}
43     }
44 }
45 barrier();
46
47 // skonstruuj listę krawędzi implikowanych
48 for (int i = 0; i < outVertexIndex.length /* NUM_VERTICES */; i++) in parallel {
49     for (int j = outVertexIndex[i]; j < outVertexIndex[i]+outDegree[i]; j++) {
50         Vertex isNext = getVertexByIndex(outVertexList[j]); // j-ty następnik
51                                                             // wierzchołka i
52         {Sprawdź czy wierzchołek i jest następnikiem wierzchołka isNext. Jeżeli nie
           to krawędź (i, isNext) zapamiętaj w lokalnej liście krawędzi implikowanych,
           a potem uaktualnij impliedDegree[i] i impliedEdgeIndex wierzchołka i oraz
           tych o większych indeksach.}
53     }
54 }
55 barrier();
56
57 {Połącz lokalne listy krawędzi implikowanych w jedną wspólną reprezentację (wypełnij
    tablicę impliedEdgeList).}
```

Rysunek 3.19: Algorytm budowania tablic sąsiedztwa zawierających następniki wierzchołków oraz reprezentacji krawędzi nieobecnych. Opracowanie własne na podstawie: [4, algorytm 3].

znaczenie parametrów dostarczanych do programu przez linię poleceń przedstawia tabela 3.3.

Tablica 3.3: Argumenty linii poleceń *vacation* i ich znaczenie.

`-n` *n*	Dla transakcji klasy 3) i 4) parametr określa liczbę elementów dodawanych lub usuwanych. W przypadku 1) *n* elementów z losowo wybranych tabel z przelotami, samochodami i pokojami jest przeglądanych, po czym te z najwyższymi cenami podlegają rezerwacji przez losowo wybranego użytkownika.
`-q` *q*	By zwiększyć poziom współzawodnictwa można ograniczyć wartości identyfikatorów użytkowników i rezerwowanych elementów, na których operują transakcje. Spośród zakresu identyfikatorów od 0 do *r* tylko *q*% najmniejszych będzie zwracanych przez procedurę generowania losowych identyfikatorów.
`-c` *c*	*c* klientów bazy danych będzie równolegle wykonywało na niej transakcje.
`-t` *t*	Ogółem w ramach programu zostanie wykonanych *t* transakcji.
`-r` *r*	Baza danych będzie posiadała swój początkowy stan. Parametr *r* posłuży do jego wytworzenia, bowiem z początku tabele z przelotami, samochodami i pokojami będą posiadały po *r* rekordów każda.
`-u` *u*	Parametr *u* definiuje podział puli *t* transakcji na klasy w następujący sposób: *u*% stanowić będą transakcje typu 1), $\frac{100-u}{2}$% to transakcje typu 2), natomiast transakcji typu 3) i 4) będzie po $\frac{100-u}{4}$%.

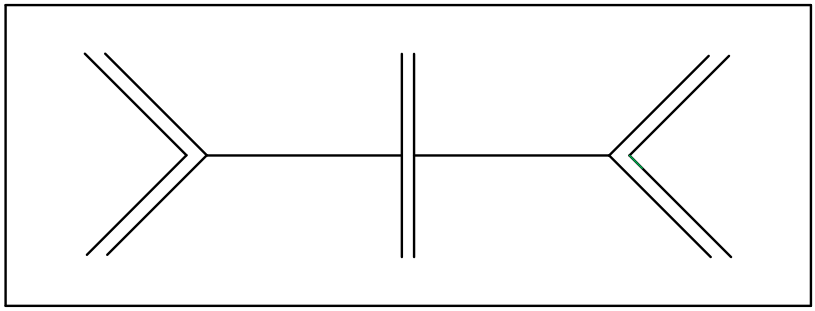
Benchmark *vacation* jest benchmarkiem generycznym w tym sensie, że przetwarzanie koncentruje się w nim wokół pojedynczej centralnej struktury danych, a charakterystyka transakcji nie jest determinowana przez semantykę rozwiązywanego problemu, ale powstaje losowo, przy udziale parametrów precyzowanych przez wywołującego program. Gdyby nie różnorodność transakcji i próba nadania przetwarzaniu praktycznego znaczenia (symulacja systemu rezerwacji w biurze podróży) można by uznać go za mikrobenchmark. Tym co ostatecznie decyduje, że *vacation* nim nie jest będzie jednak brak wyraźnie zarysowanych możliwości takiego generowania obciążeń systemu TM, które skupiałyby się na jego pojedynczych aspektach (zdolności do radzenia sobie z wysokim poziomem współzawodnictwa, wpływem transakcji o dużych rw-setach na wydajność przetwarzania, itd.).

Benchmark symulujący działanie bazy danych jest niemal idealnym kandydatem do przeniesienia na grunt systemów rozproszonych. Abstrahując od semantyki *vacation*, zestaw tabel można podzielić partycjonując je czy to poziomo czy pionowo i rozmieścić razem z klientami na różnych węzłach systemu.

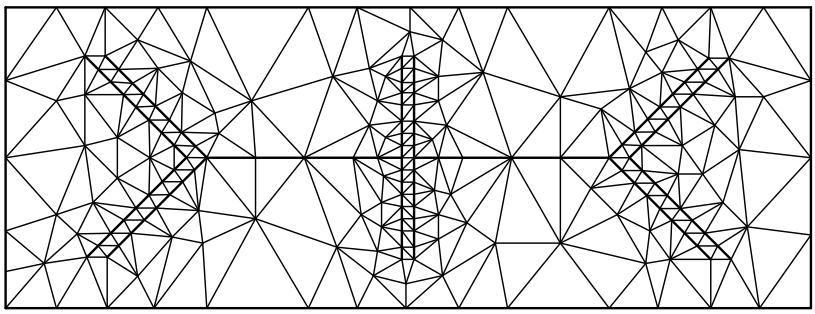
Alternatywnie można pokusić się o symulowanie działania bazy danych, która sama w sobie jest już rozproszona. W takim układzie znowu trzeba by generować szereg klas transakcji o różnych parametrach i próbować dorobić do nich praktyczne znaczenie.

3.5.8 Rozproszona wersja *yada*

Przykład płaszczyzny przeznaczonej do podziału na trójkąty w postaci PSLG (ang. *planar straight-line graph*) oraz samą triangulację Delaunaya przedstawia rysunek 3.20.



(a) Graf PSLG.



(b) Triangulacja Delaunaya.

Rysunek 3.20: Triangulacja Delaunaya płaszczyzny ograniczonej grafem PSLG. Za: [67].

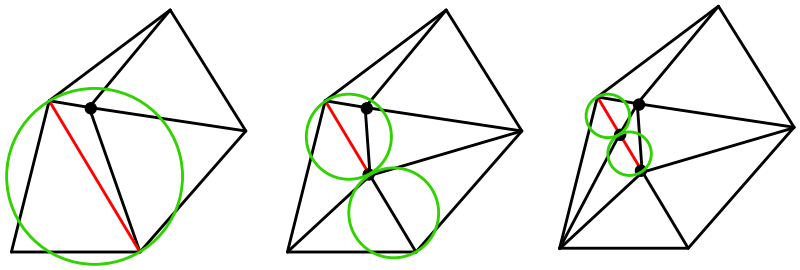
Dokładny opis algorytmu Rupperta, który implementuje ten benchmark znaleźć można w [60]. Jak wspomniano w opisie zawartym w punkcie 2.6.1.8, sposobem dopasowania triangulacji do wymagań użytkownika jest retriangulacja pewnych jej obszarów, zwanych tutaj *regionami*.

O odcinku mówi się, że jest naruszony (ang. *encroached*), jeżeli wierzchołek pewnego trójkąta będącego elementem triangulacji znajduje się wewnątrz okręgu opisanego na tym odcinku. Trójkąt jest zły (ang. *bad*), jeżeli najmniejszy z jego kątów ostrych jest mniejszy wartość graniczna, podawana przez użytkownika. Z faktu, że triangulacja podawana na wejście programu jest triangulacją Delaunaya wynika, że trójkąty (przynajmniej początkowo) nigdy nie będą naruszone w sposób analogiczny do odcinków.

Algorytm Rupperta rozpoczyna działanie od zapełnienia kolejki zadań. Każdy trójkąt należący do triangulacji wejściowej jest oceniany pod kątem tego, czy może zostać nazwany złym i – jeżeli okazałoby się to prawdą – umieszczany w kolejce. Zachowanie własności Delaunaya (dla każdego trójkąta żaden wierzchołek nie leży wewnątrz opisanego na nim okręgu) nie gwarantuje, że triangulacja będzie wolna od odcinków naruszonych. Stąd należy także przeanalizować każdy bok każdego trójkąta pod kątem tego, czy wewnątrz opisanego na nim okręgu nie leży żaden wierzchołek. W [60] znaleźć można informację, że kolejki zadań utrzymywane są osobno dla trójkątów i odcinków, przy czym przetworzenie odcinków otrzymuje wyższy priorytet.

Przetworzenie odcinka sprowadza się do wstawienia nowego wierzchołka dokładnie w jego środku. Sprawia to, że okręgi opisane na dwóch nowo powstałych odcinkach mają znacząco mniejszy rozmiar i w efekcie być może żaden z nich nie będzie obejmował wierzchołka naru-

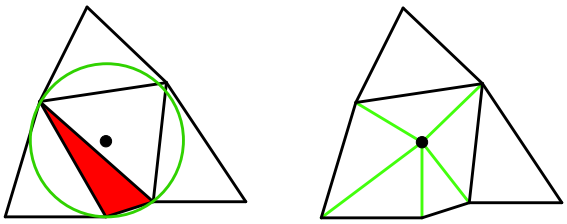
szającego. Jeżeli tak by się nie stało, czynność powtarza się rekurencyjnie. Proces przetwarzania odcinka został przedstawiony na rysunku 3.21.



Rysunek 3.21: Sposób w jaki algorytm Rupperta przetwarza naruszony odcinek. Za: [60].

W wyniku podziału odcinka do triangulacji zostaje wprowadzony nowy punkt, który powinie stać się wierzchołkiem trójkąta, a w zasadzie czterech trójkątów. By takie trójkąty utworzyć, konieczne może być wstawienie krawędzi łączących nowy wierzchołek z już istniejącymi. Pewne krawędzie mogą wtedy także zostać usunięte. Pierwszy podział na rysunku 3.21 powoduje usunięcie jednej krawędzi. Drugi – już nie. To w jaki sposób wybrać jedno z wielu możliwych połączeń nowego wierzchołka z innymi nie jest oczywiste. W [60] znaleźć można informację o algorytmie Lawsona, który miałby wstawić krawędzie w taki sposób, by utrzymana została własność Delaunaya. Alternatywną drogą wydaje się przyjęcie pewnej stałej zasady (np. rezygnacji z usuwania jakichkolwiek krawędzi) i wstawianie nowo utworzonych trójkątów do kolejki zadań, jeżeli tylko zostały ocenione jako złe.

Przetworzenie złego trójkąta, przedstawione na rysunku 3.22 sprowadza się do wstawienia do triangulacji środka opisanego na nim okręgu. Tu już co najmniej jedna krawędź musi zostać usunięta by zły trójkąt został wyeliminowany. Znowu nie jest jasne, czy retriangulacja tak powstałego obszaru powinna odbywać się w oparciu o pewną zasadę gwarantującą zachowanie własności Delaunaya, czy też może być realizowana dowolnie pod warunkiem, że nowo powstałe złe trójkąty trafią do kolejki zadań.

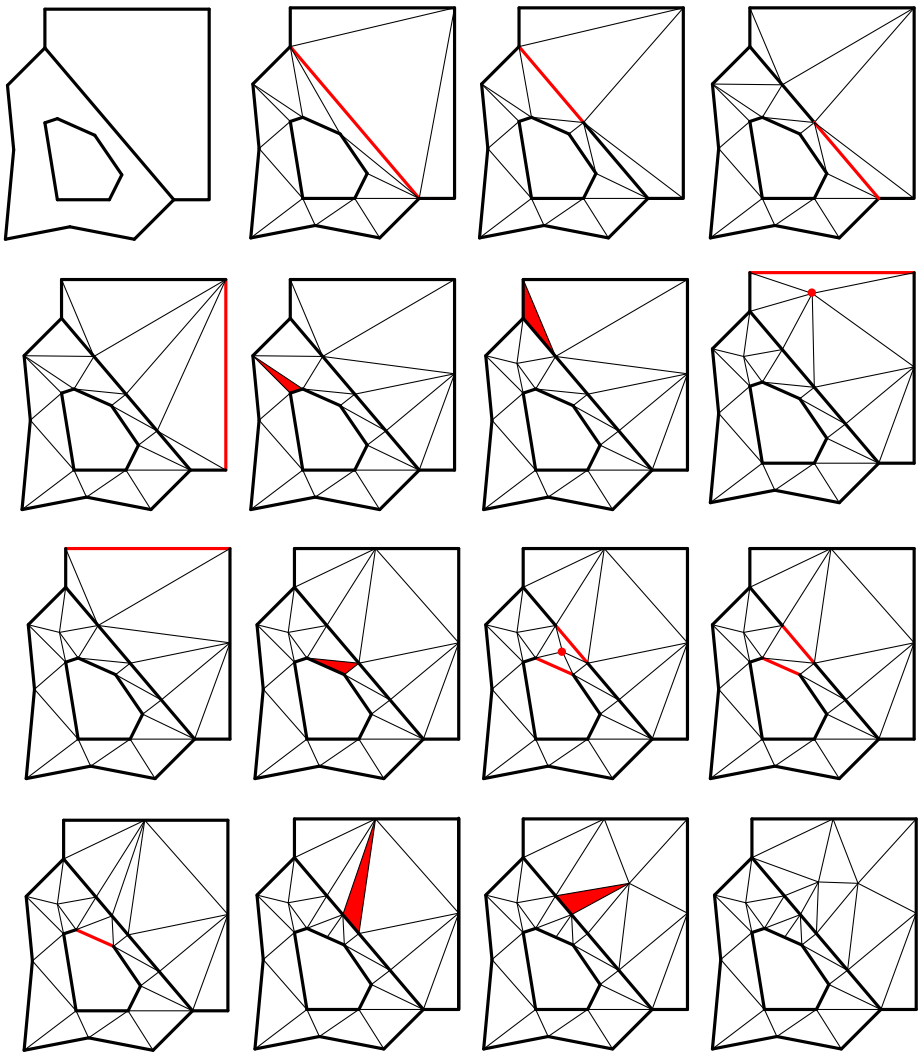


Rysunek 3.22: Sposób w jaki algorytm Rupperta przetwarza zły trójkąt. Za: [60].

Całościowy proces dopasowania triangulacji do wymagań użytkownika przedstawia rysunek 3.23. Na każdym kroku zostały w nim zaznaczone przetwarzane elementy.

Równoległa wersja benchmarka *yada* wykorzystuje do reprezentacji triangulacji następujące struktury danych:

- *punkt* (ang. *coordinate*), który jest parą *(x, y)* współrzędnych na triangulowanej płaszczyźnie,



Rysunek 3.23: Całokształt procesu retriangulacji. W lewym górnym rogu graf PSLG, dalej triangulacja Delauneya podawana na wejście algorytmu Rupperta oraz jego kolejne kroki. Na każdym etapie zaznaczono przetwarzane elementy. Za: [60].

- *krawędź* (ang. *edge*), definiowaną przez parę punktów: początek i koniec,
- *element*: trójkąt lub odcinek, definiowany przez współrzędne początków i końców krawędzi, zawierający także informacje najmniejszym kącie ostrym (jeżeli element jest trójkątem), środku okręgu opisanego oraz o sąsiadach kryjących się za każdą krawędzią,
- *region*, grupujący złe trójkąty i łączący ich zbiór ze środkiem okręgu opisanego na jednym z nich.

Region wprowadzany jest po to, by móc dokonać retriangulacji obszaru większego niż pojedynczy trójkąt. W momencie rozpoczęcia przetwarzania kolejnej pozycji z listy zadań zły trójkąt staje się środkiem regionu, środek opisanego na nim okręgu – nowym wierzchołkiem wprowadzanym do triangulacji, a dodatkowo sprawdza się czy sąsiednie elementy również nie wymagają retriangulacji. Procedura ta, nieobecna w oryginalnym algorytmie Rupperta, została w *yada* wprowadzona zapewne po to, by zwiększyć rozmiary produkowanych transak-

cji. Na potrzeby implementacji nazwano ją rozrostem regionu (ang. *region growth*).

Wprowadzony opis wystarczy by rzucić światło na to, w jaki sposób triangulacja jest reprezentowana w pamięci operacyjnej. By algorytm dysponował pewnym punktem zaczepienia wprowadzona zostaje dodatkowo struktura, *mesh*, która zawiera wskaźnik na wybrany element (tzw. *root element*; do pozostałych można się dostać przechodząc wpierw przez niego, potem przez jego sąsiadów, itd.) i kolejkę złych trójkątów do przetworzenia.

Sposób zrównoleglenia aplikacji sprowadza się do wprowadzenia szeregu wątków, które współbieżnie pobierają zadania ze współdzielonej kolejki i dokonują retrinagulacji wybranych obszarów płaszczyzny. Jest oczywiste, że jeżeli współbieżnie retriangulacji podlegają dwa obszary, które choćby fragmentem na siebie nachodzą, umieszczenie operacji na nich wewnątrz transakcji sprawi, że taki konflikt zostanie wykryty, a jeden z wątków będzie zmuszony sprawdzić po ponowieniu transakcji, czy czy jego zadanie jest jeszcze zasadne, tj. czy zły trójkąt, którego to zadanie dotyczy nie został już z triangulacji usunięty.

Rozproszenie benchmarka *yada* mogłoby polegać na podziale triangulowanej płaszczyzny (zbioru elementów) pomiędzy węzły systemu rozproszonego i przechowywaniu w każdym elemencie lokalizacji sąsiadów uwzględniającej adres posiadacza. Poza wyłącznie dużymi rozmiarami produkowanych transakcji (brakiem różnorodności generowanych obciążeń) i problemem rozproszenia kolejki zadań, nie można dopatrzeć się implementacyjnych przyczyn braku możliwości integracji rozproszonego wariantu *yada* z dynamicznymi pamięciami transakcyjnymi. W przypadku pamięci statycznych kłopotliwa jest niemożliwość przewidzenia w jaki sposób rozrośnie się region przeznaczony do retriangulacji.

Jednak najbardziej znaczącym problemem, który zdecyduje o tym, że próba rozproszenia benchmarka nie zostanie podjęta jest stopień skomplikowania implementowanego algorytmu mogący powodować trudności tak przy łączeniu programu z różnymi implementacjami pamięci transakcyjnych, jak i przy przenoszeniu go pomiędzy językami i paradygmatami programowania.

3.6 Propozycje benchmarków dla systemów rozproszonych

W oparciu o dotychczas przedstawione informacje można już podjąć próbę wybrania tych z problemów rozwiązywanych przez programy składowe STAMPa, które posłużą do stworzenia benchmarków dla rozproszonych pamięci transakcyjnych.

3.6.1 Rozproszony wariant *genome*

Spośród programów rozwiązujących konkretny problem (wyjątkiem jest tu w zasadzie tylko *vacation* i, w pewnym sensie, *intruder*[13]), które są godne rozproszenia ze względu na wo-

[13] jak wspominano wcześniej, w *vacation* (gdyby pominąć parametry wejściowe, opisane w punkcie 3.5.7) można kształtować charakterystykę transakcji w niemal dowolny sposób bez naruszania semantyki problemu algorytmicznego. Własności takiej nie zachowuje np. *genome*, w którym bez względu na dane wejściowe, próby zszycia (co najwyżej zakończone niepowodzeniem) będą następować z podobną częstością. W tym ujęciu *intruder* pla-

lumeny przetwarzanych danych, na uwagę zasługują *bayes*, *genome*, *kmeans* i *yada*. *intruder* należy do innej klasy problemów, ponieważ nie implementuje algorytmu operującego na strukturze o stałej wielkości znanej przed rozpoczęciem wykonania (zawartość kolejki przechwyconych pakietów można uzupełniać nawet w trakcie wykonania programu). Podobnie *vacation*, w którym charakterystykę transakcji można niemal dowolnie modyfikować. *ssca2* rozwiązuje problem zbyt ogólny by w ogóle mówić o celowości jego rozproszenia. Natomiast *labyrinth*, z racji implementowania algorytmu wykorzystywanego m.in. przy wytyczaniu ścieżek w obwodach drukowanych, każe myśleć o skali przetwarzania odpowiedniej raczej dla systemów wieloprocesorowych niż rozproszonych.

Z pozostałych *yada* implementuje algorytm, którego stopień komplikacji jest na tyle duży, że może stwarzać problemy przy próbie przenoszenia benchmarka pomiędzy różnymi językami i paradygmatami programowana (nie doszukano się też dobrego sposobu rozproszenia centralnej struktury danych), a *kmeans*, choć użyty do w oryginale do pokazania pewnych ciekawych własności wybranych implementacji wieloprocesorowych pamięci transakcyjnych (por. [13, punkt V.B.4]) produkuje wyłącznie krótkie transakcje, które same w sobie nie są raczej typowym elementem programu działającego w systemie rozproszonym.

Z dwóch pozostałych wybrano *genome* jako kandydata do implementacji jego wersji rozproszonej z racji tego, że został zbudowany w oparciu o prostszy algorytm niż *bayes*, a wykorzystywane tam struktury danych dają się łatwo rozproszyć. Elementem niepożądanym w kontekście systemów rozproszonych jest obecna w *genome* bariera synchronizacyjna po każdej iteracji związanej z długością nakładających się części segmentów, nadająca przetwarzaniu „turowy" charakter. Spory wysiłek będzie więc włożony w implementację rozproszonej wersji benchmarka, tak by wyeliminować ten element przetwarzania. W przeciwieństwie do *kmeans*, który wymagałby fundamentalnej przebudowy algorytmu, dla *genome* wydaje się to możliwe do osiągnięcia.

Okaże się, że wszystkie etapy przetwarzania, począwszy od generowania instancji, poprzez usuwanie duplikatów i haszowanie prefiksów segmentów, po sam proces sekwencjonowania dadzą się zrównoleglić w środowisku rozproszonym.

Wreszcie sekwencjonowanie, przy dużym uproszczeniu, można sprowadzić do szeregu operacji mających na celu zbudowanie acyklicznej listy linkowanej. Problem wyznaczania optymalnego (więc o minimalnej liczności) access-setu przy operowaniu na takiej liście będzie szczególnie trudny jeżeli w użyciu znajdzie się statyczna pamięć transakcyjna.

3.6.2 Benchmark *cassandra*

Czerpiąc z idei najbardziej podatnego na rozproszenie benchmarka ze zbioru STAMP – *vacation*, do implementacji wyznacza się wariant rozproszonej bazy danych klasy NoSQL, bazującej na rozproszonej tablicy haszowej – narzędzie *Cassandra*.

Wybór ten jest korzystny w wielu aspektach. Po pierwsze *Cassandra* została zaprojekto-

suje się pomiędzy nimi, jednak bliżej *vacation*. Semantyka problemu algorytmicznego składania pakietów w strumienie, jest nieco dokładniej określona niż wykonanie szeregu transakcji na bazie danych, jednak za pomocą danych wejściowych można niemal dowolnie (choć nie niezależnie) sterować kluczowymi elementami charakterystyki transakcyjnej *intrudera*, takimi jak częstość i rozmiar transakcji składających pakiety.

wana i stworzona w określonym celu – by przechowywać zawartości skrzynek odbiorczych systemu wymiany wiadomości przez użytkowników serwisu społecznościowego *Facebook*. [40], wprowadzający *Cassandrę*, podaje gotowy sposób jej wykorzystania.[14] Po drugie ta baza danych jest ze swojej natury systemem rozproszonym, zatem zapewne przy zachowaniu choćby wycinka jej funkcjonalności nadal będzie obciążać rozproszoną pamięć transakcyjną tak jak rzeczywista aplikacja. Po trzecie *Cassandra* (póki co) nie wspiera przetwarzania transakcyjnego. Zatem zintegrowanie jej uproszczonego wariantu z rozproszoną pamięcią transakcyjną może stanowić przyczynek do dalszych prac mających na celu wyposażenie tej bazy danych w mechanizm przetwarzania OLTP.

Rozproszona tablica haszowa jako architektura bazy danych oraz stosunkowo prosty schemat wykorzystania *Cassandry* do przechowywania wiadomości użytkowników (a więc i mało różnych klas transakcji wykonywanych w benchmarku) czyni z programu wzorcowego, który by na niej bazował raczej mikrobenchmark niż benchmark pełnowymiarowy. W przypadku jednak gdyby zestaw sposobów wykorzystania bazy danych udało się rozszerzyć tak, by móc generować dostatecznie dużą liczbę transakcji różnych typów, można obronić się przed takim zarzutem twierdząc, że architektura struktury danych leżącej u podstaw benchmarka, choć prosta, odpowiada rzeczywistej aplikacji rozproszonej, a sposoby jej wykorzystania formują dostatecznie skomplikowany program.

Stąd, podczas gdy [40] określa dwie podstawowe operacje wykonywane na bazie danych (obie są operacjami odczytu), benchmark *cassandra* będzie implementował jeszcze sześć dodatkowych, dostarczając razem 10 klas transakcji o różnych charakterystykach.

3.6.3 Inne możliwości

Pomimo, że STAMP miał być głównym źródłem problemów i ich rozwiązań, w oparciu o które miano budować benchmarki dla rozproszonych pamięci transakcyjnych, rozważyć należy też inne możliwości. Niniejszy punkt prezentuje benchmark, którego specyfikacja została opracowana to tego stopnia, że jako trzeci mógłby zostać zaimplementowany w ramach tego raportu. Ma on uwydatniać zalety *Atomic RMI* względem innych metod sterowania współbieżnością, takie jak sposób radzenia sobie z operacjami niewycofywalnymi i rozproszonym charakterem przetwarzania. Z różnych względów, a w szczególności w wyniku złego oszacowania pracochłonności zaimplementowania benchmarka *genome*, nie wyjdzie on poza etap projektu.

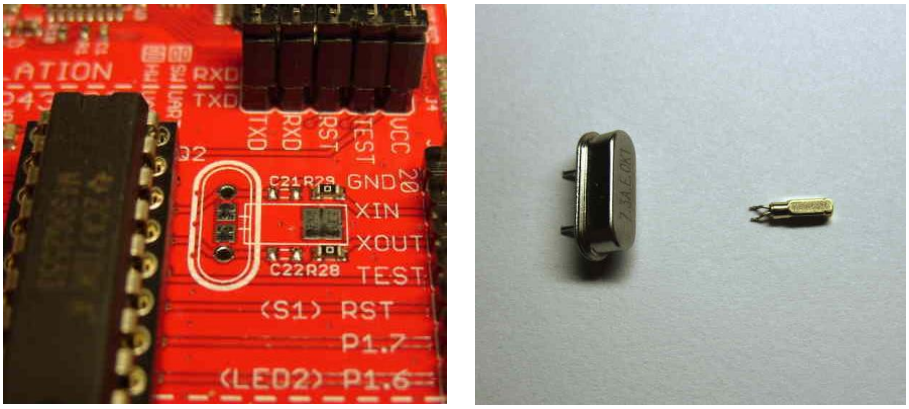
Dodatkowo punkt przedstawia kilka pobocznych kierunków poszukiwania zastosowań dla systemów rozproszonych, które być może byłyby w stanie wykorzystać pamięci transakcyjne.

3.6.3.1 Benchmark *BOM*

Przy projektowaniu obwodów drukowanych przeznaczonych do seryjnej produkcji, dość częstą praktyką jest przystosowywanie pojedynczej płytki z obwodem do zamontowania na niej dwóch lub więcej wariantów części elektronicznych. Wiąże się to zapewne ze znacznymi

[14] nie można twierdzić, że architektura *Cassandry* determinuje jej zastosowanie. Niech zaświadczy o tym fakt, że istnieje kilka innych typowych przykładów wykorzystania tej bazy danych, w tym przykład związany z liniami lotniczymi, przedstawiony w [23].

kosztami stałymi rozpoczęcia produkcji (w tym – z kosztem zaprojektowania obwodu) w połączeniu z częstą praktyką dostarczania identycznych lub bardzo podobnych funkcjonalnie komponentów elektronicznych (w tym – układów scalonych) w różnych obudowach. Rysunek 3.24 przedstawia przykład takiego obwodu – płytkę prototypową *MSP430 LaunchPad* przeznaczoną dla mikrokontrolerów firmy *Texas Instruments*.



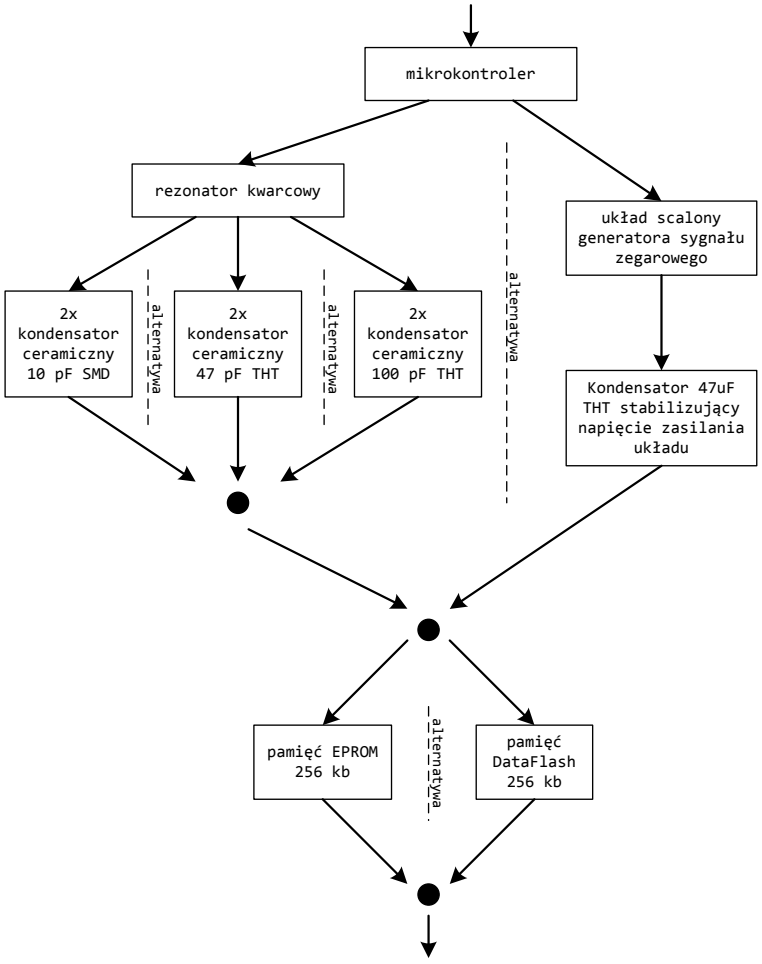
Rysunek 3.24: Fragment płytki prototypowej dla mikrokontrolera *Texas Intruments MSP430* umożliwiający zamontowanie rezonatora kwarcowego montowanego powierzchniowo (SMD) lub przetykanego (THT) oraz przykłady obu wariantów rezonatorów (THT po lewej stronie).

BOM (ang. *bill of materials*) z kolei jest elementem dokumentacji projektu urządzenia elektronicznego – wykazem elementów, razem z ich oznaczeniami wprowadzanymi przez dostawców, które należy pozyskać by móc urządzenie złożyć. Zazwyczaj BOM jest ścisłą specyfikacją, w której próżno szukać alternatyw. Jeżeli jednak weźmie się pod uwagę podejście opisane wcześniej, przydatny okazać się może wykaz części w postaci listy, na której pewne grupy pozycji będą posiadały swoje alternatywy. Przykład takiej listy przedstawia rysunek 3.25.

Niech firma produkująca urządzenia elektroniczne pozostaje w kontakcie z grupą dostawców części, którzy udostępniają mechanizm zamawiania elementów oraz sprawdzania stanów magazynowych za pośrednictwem technologii *Web Services* (np. po jednej usłudze na określony dział magazynu).

Proces zakupu elementów wyszczególnionych w BOM będzie procesem biznesowym angażującym dwóch aktorów, z których każdy ma swoje cele. I tak: zamawiający (producent urządzeń elektronicznych) będzie chciał skompletować zamówienie jeżeli tylko pozwalają na to stany magazynowe oraz mieć na końcu możliwość zweryfikowania listy zamówionych elementów przed ostatecznym zatwierdzeniem zamówienia. Sprzedający (dostawca części) będzie w tym czasie chciał sprzedać towar jeżeli tylko jest to możliwe (tj. uniknąć sytuacji, w której gdy żądany przez zamawiającego towar znajduje się w magazynie, z pewnych względów nie zostaje on sprzedany), jednak przy zachowaniu określonego poziomu jakości oferowanych usług, tak by nie zniechęcać potencjalnych klientów do korzystania z systemu zamówień.

Benchmark BOM powstawał z myślą o porównaniu już nie tyle dwóch charakterów rozproszonej pamięci transakcyjnej: optymistycznego i pesymistycznego, ale dwóch konkretnych implementacji: *Atomic RMI* i *HyFlow* w dowolnym wariancie – data-flow lub control-flow. Dalsza część tego punktu skupiać się będzie na znalezieniu uzasadnienia dla wykorzystania



Rysunek 3.25: Przykład dokumentu BOM zawierającego alternatywy. Generowanie częstotliwości dla mikrokontrolera może być realizowane przez oscylator kwarcowy lub zewnętrzny układ. Przewidziano też dwa warianty wykorzystywanej pamięci trwałej.

mechanizmów dostarczanych przez oba te systemy w procesie kontaktowania się zamawiającego z dostawcami.

Proces taki – składanie zamówienia – będzie polegał na przejrzeniu dokumentu BOM i próbie pozyskania od każdego z dostawców określonego elementu w wymaganej ilości. Jeżeli pewnego elementu w ramach alternatywy nie uda się uzyskać od żadnego z dostawców, wybierana jest inna alternatywa. Jeżeli nie uda się skompletować żadnej alternatywy, zamawiający ma prawo do wglądu w status zamówienia – listę elementów z każdej z alternatyw, które znajdują się w magazynach i w oparciu o ten status decyduje, czy zamówienie ma zostać zatwierdzone czy też wycofane.

Uzasadnienie objęcia realizacji zamówienia transakcją wynika z twierdzenia, że podczas przetwarzania poszczególnych pozycji z listy, raz obrana alternatywa nie powinna być już zmieniana. Należy więc upewnić się, że wszystkie elementy z danej gałęzi są dostępne i wtedy dokonać jej wyboru – zarezerwować elementy. Oczywiście współbieżnie z zamówieniami mogą być także realizowane dostawy do magazynów. Stąd potrzeba zabezpieczenia systemu przed

anomaliami niesynchronizowanego współbieżnego dostępu.

W kwestii modelu systemu rozproszonego, budowanego z myślą o wyborze pomiędzy podejściami control-flow i data-flow należy zaznaczyć, że *Atomic RMI* zyskuje przewagę nad *HyFlow* w wariancie data-flow ponieważ podejście to w sposób naturalny modeluje technologię *Web Services*. Oczywiście w przypadku benchmarka system *Atomic RMI* byłby integrowany z pewną namiastką technologii *Web Services*, która pozwalałaby na tworzenie kopii zapasowych stanu systemu. Natomiast w przypadku rzeczywistych zastosowań nie można mówić o bezpośrednim wykorzystaniu *Atomic RMI*, raczej o stosowaniu implementacji algorytmu SVA, w której np. procedura odtwarzania stanu t-obiektu byłaby tłumaczona na procedurę kompensacji wykonania usługi. Podobnie trudno jest mówić o zastosowaniu *HyFlow* w rzeczywistym systemie, choć można by się silić na tłumaczenie poszczególnych odczytów i zapisów na wywołania usług zdalnych. W takim wariancie *HyFlow* musiałby pozwalać na powiązanie ze zdarzeniem wycofania transakcji odpowiedniej procedury kompensującej zrealizowane dotychczas wywołania usług.

W kwestii celów każdego z aktorów niemałe znaczenie może mieć wybór pomiędzy podejściem pesymistycznym a optymistycznym. W podejściu pesymistycznym (na przykładzie SVA) wycofanie transakcji (a więc i kompensacja wykonania) nastąpi tylko na żądanie użytkownika, podczas gdy w podejściu optymistycznym może być ono jeszcze efektem wystąpienia konfliktów (zatem sumarycznie operacji kompensowania wykonania pewnej usługi będzie więcej). W efekcie system składania zamówień może być przesadnie obciążony, co kłóci się z celem dostawcy. Z drugiej strony jeżeli współbieżnie realizowane jest zamówienie i dostawa, reguła arbitrażu może zapewnić, że to zamówienie zawsze będzie wycofywane i przy ponownym wykonaniu zwiększą się szanse na to by wszystkie potrzebne elementy były dostępne w magazynie. W przypadku *Atomic RMI* jeżeli transakcja-dostawa zostanie rozpoczęta później niż transakcja-zamówienie, nie przeszkodzi ona w jego złożeniu sprawiając, że producent urządzeń być może zrezygnuje ze składania niekompletnego zamówienia. Kłóci się to zarówno z celem producenta, jak i dystrybutora.

Znajomość całego dokumentu BOM w momencie rozpoczynania składania zamówienia stanowi też dobry punkt wyjścia dla przewidywana access-setów *a priori*, choć prawdopodobnie dopóki nie wprowadzi się kilku „terminali usług sieciowych” (osobnych, w pewnym sensie replikowanych t-obiektów) na jeden dział magazynu, skalowalność rozwiązania bazującego na transakcjach pesymistycznych i tak będzie niewystarczająca, niejako dyskwalifikując to podejście.

3.6.3.2 Benchmark *Apache*

W nielicznych komentarzach jakie można znaleźć w kodzie źródłowym narzędzia *HyFlow* doszukano się napomknień dotyczących wykorzystania serwera HTTP *Apache* w roli benchmarka. W istocie jest jeden element tej aplikacji, który potencjalnie mógłby spełniać podstawowe wymagania stawiane programom wzorcowym dla rozproszonych pamięci transakcyjnych.

Elementem serwera *Apache* w wersji 2.2 są tzw. moduły MPM (ang. *multi-processing modules*). Ich rola to umożliwienie współbieżnej obsługi wielu żądań HTTP na różne sposoby. Dla

przykładu *prefork mpm* implementuje politykę, według której serwer uruchamia przy starcie zadaną, stałą liczbę procesów odpowiedzialnych za obsługę żądań, a *worker mpm* dodatkowo dzieli te procesy na wątki, tworząc je dynamicznie w razie potrzeby i nakładając ograniczenie na maksymalną liczbę wątków uruchomionych jednocześnie.

Apache w wersji 2.2 dostarcza też eksperymentalny moduł *event mpm*, który w założeniu ma rozwiązywać występujący w *worker mpm* problem permanentnych połączeń TCP. Mechanizm *Connection Keep-Alive* dostępny w protokole HTTP pozwala zrezygnować z zamykania połączenia TCP po obsłużeniu żądania, by uniknąć kosztów związanych z ponownym jego otwieraniem w momencie gdy od tego samego klienta nadejdzie kolejne żądanie. Problem polega na tym, że tak długo jak długo połączenie jest zestawione, pojedynczy wątek blokuje się w oczekiwaniu na kolejne żądanie, a odstępy pomiędzy dwoma następującymi po sobie żądaniami potrafią być znaczące (np. tak długie jak czas potrzebny internaucie na zapoznanie się z treścią strony i przejście do następnej – jeżeli tylko nie nałożono innych ograniczeń). Wątek, choć uśpiony, zajmuje miejsce w pamięci operacyjnej i obciąża mechanizmy jądra systemu operacyjnego. *event mpm* stawia sobie za cel wprowadzenie „zdarzeniowej" obsługi połączeń. Wątek zamiast biernie oczekiwać na nadejście kolejnego żądania obsługuje zdarzenia nadejścia żądań różnymi połączeniami TCP. W założeniu ma to pozwolić na zmniejszenie liczby wątków potrzebnych by współbieżnie obsłużyć taką samą liczbę klientów.

Być może dobrym pomysłem na benchmark dla rozproszonych pamięci transakcyjnych byłoby utworzenie farmy serwerów HTTP wykorzystującej load-balancing i wykorzystywanie transakcji 1) do utrzymywania puli (np. kolejki) żądań do obsłużenia, 2) zdalnego dostępu do zasobów potrzebnych do ich obsługi, tak by zapewnić przezroczystość rozproszenia (a dokładnie przezroczystość przełączania klienta pomiędzy jednostkami roboczymi), czy wreszcie 3) utrzymywania sesji w serwerach aplikacji zintegrowanych z serwerem HTTP, np. w interpreterze PHP.

3.6.3.3 Distributed Web Crawling

Eksploracja zasobów internetowych [45, punkt 0.1.] jest bez wątpienia jedną z tych dziedzin informatyki, które z racji przetwarzania dużych wolumenów danych podlegają problematyce *Big Data* (a więc wymuszeniu rezygnacji z klasycznych metod przetwarzania danych na rzecz metod dopasowanych do skali takiego przetwarzania – w tym: wykorzystania systemów rozproszonych).

Wśród wielu narzędzi, koncepcji i rozwiązań rozproszonych wykorzystywanych w tej dziedzinie, poza najprostszymi, np. modelem przetwarzania *map-reduce* i jego implementacją – *Hadoop*, wyszukać można i takie, w których znalazłoby się zastosowanie dla transakcji.

Niech za przykład posłuży robot internetowy (ang. *crawler, spider*) działający w środowisku rozproszonym, z jednostkami przetwarzającymi i przechowującymi dane rozlokowanymi na różnych węzłach. Robot taki operuje na dwóch strukturach danych znacznych rozmiarów, z których jedna potencjalnie a druga z całą pewnością jest rozproszona. Pierwsza z nich – wejściowa – to kolejka adresów stron do pobrania, rozbudowywana wraz z postępem przetwarzania (wyszukiwaniem odnośników w kolejnych pobranych dokumentach). Druga – treści pobranych stron – najprawdopodobniej będzie na tyle duża, że w całości nie zmieści się

na żadnym pojedynczym węźle systemu.

Konstrukcja kolejki adresów stron do pobrania nie jest zadaniem trywialnym z racji tego, że na robota internetowego narzuca się szereg ograniczeń [45, punkt 1.2.2.]. Przykładem niech będzie tu zakaz pobierania treści tej samej strony w zbyt krótkich odstępach czasu, tak by administrator strony internetowej, obserwujący dodatkowe obciążenie, nie zdecydował się zablokować możliwości crawlowania. Dodatkowo potrzebne może okazać się przesunięcie terminu crawlowania określonych stron w pory nocne różnych stref czasowych, tj. wtedy gdy najmniejszą aktywność wykazują ich standardowi użytkownicy. W ogólności rozproszone transakcje mogą więc być wykorzystywane do tworzenia, łączenia i przetwarzania kolejek np. poprzez skierowanie zadań pobrania dokumentów spod tego samego adresu na jeden węzeł, usuwanie zduplikowanych adresów (czy to w momencie dodawania nowego, czy w formie inicjowanego zewnętrznie, np. co 10 minut, zadania) lub takie ich ułożenie, by robot mógł spełnić nakładane na niego wymagania.

W kwestii treści pobranych dokumentów często zachodzi potrzeba ich przetworzenia na potrzeby ułatwienia późniejszego wyszukiwania. Najprostszym przykładem będzie tu generowanie indeksów. Ponownie w środowisku rozproszonym istnieć będzie szereg algorytmów „redukujących" dane dostarczone przez robota, w których zastosowanie mogą znaleźć pamięci transakcyjne. Przykładem rozwiązania z tej dziedziny jest algorytm konstrukcji list wystąpień słów (termów) w pobranych dokumentach (ang. *postings lists*), opracowywanych najpierw dla danych lokalnych, a później integrowanych do współdzielonego indeksu np. przy użyciu algorytmu *blocked sort-based indexing* [44, punkt 4.2].

W ogólności zaletą szukania problematyki dla programów wzorcowych testujących wydajnościowo rozproszone pamięci transakcyjne w dziedzinie eksploracji zasobów internetowych jest to, że znaleźć w niej można dużo różnych rozwiązań. Wiele spośród nich to proste algorytmy powstałe z myślą o łatwości ich wykorzystania w środowisku rozproszonym. Ponieważ przyczyna ich powstania, wiążąc się z dynamicznym rozwojem internetu, ma przeważnie praktyczny charakter (są to odpowiedzi na zaistniałe już problemy), udowodnienie, że wykorzystywane są one w rzeczywistych aplikacjach nie powinno stanowić kłopotu.

3.6.3.4 Data mining

Sugestia tego kierunku poszukiwań oparta jest w zasadzie wyłącznie na tytule i streszczeniu rozprawy doktorskiej [19]. Dotyczy ona metod uczenia maszynowego w środowisku cechującym się geograficznym rozproszeniem danych. Na wysokim poziome abstrakcji raport przedstawia szereg rozwiązań: podejść, technik i wreszcie – algorytmów pozwalających na współbieżne „redukowanie" danych przechowywanych lokalnie na różnych węzłach do mniejszych jednostek, które zebrane i połączone później w jednym miejscu dadzą model wiedzy reprezentatywny dla zawartości całego systemu.

Być może proces łączenia takich jednostek, produkowanych na różnych węzłach geograficznie rozproszonego systemu komputerowego, stanowić będzie pole do popisu dla rozproszonych pamięci transakcyjnych.

Rozdział 4

Podsumowanie

W ramach raportu przeprowadzono dogłębną analizę istniejących zestawów benchmarków dla pamięci transakcyjnej (zarówno w przeznaczeniu dla systemów rozproszonych jak i dla tradycyjnych w tym ujęciu systemów wieloprocesorowych). Na podstawie istniejących benchmarków zaproponowano zestaw charakterystyk jakie benchmark dla rozproszonego systemu pamięci transakcyjnej powinien spełniać. Charakterystyki te posłużyły do przygotowania projektów aplikacji wzorcowych pozwalających na dogłębne przebadanie i porównanie systemów pamięci transakcyjnej, które posłużą do implementacji standardowego benchmarku do badania rozproszonej pamięci transakcyjnej w ramach przyszłych badań.

Bibliografia

[1] EuroTM (COST action IC1001) publications. http://www.eurotm.org/publications, 2014. Accessed: 22 September 2015.

[2] Standard Performance Evaluation Corporation, SPECjbb2000 benchmark. https://www.spec.org/jbb2000/, January 2006. Accessed: 24 September 2015.

[3] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January 2004.

[4] David A. Bader and Kamesh Madduri. Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. In *Proceedings of the 12th International Conference on High Performance Computing*, HiPC'05, pages 465–476, Berlin, Heidelberg, 2005. Springer-Verlag.

[5] David A. Bader, Kamesh Madduri, John Gilbert, Viral Shah, Jeremy Kepner, Theresa Meuse, and Ashok Krishnamurthy. Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *Cyberinfrastructure Technology Watch*, 2(4B), November 2006.

[6] Jan Baranowski, Paweł Kobyliński, Konrad Siek, and Paweł T. Wojciechowski. Helenos: A realistic benchmark for distributed transactional memory. Preprint submitted to Future Generation Computer Systems, special issue on Middleware Services for Heterogeneous Distributed Computing, Elsevier, September 2015.

[7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[8] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[9] Jayaram Bobba, Mark Hill, Tim Harris, and Ravi Rajwar. Transactional memory bibliography. http://research.cs.wisc.edu/trans-memory/biblio/, March 2011. Accessed: 16 May 2015.

[10] Yuri Breitbart, Dimitrios Georgakopoulos, Marek Rusinkiewicz, and Abraham Silberschatz. On rigorous transaction scheduling. *IEEE Transactions on Software Engineering*, 17(9):954–960, 1991.

[11] Jerzy Brzeziński. *Ocena stanu globalnego w systemach rozproszonych*. Ośrodek Wydawnictw Naukowych Polskiej Akademii Nauk, 2001.

[12] Călin Caşcaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *ACM Queue*, 6(5):40:46–40:58, September 2008.

[13] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[14] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. Asynchronous lease-based replication of software transactional memory. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, pages 376–396, Berlin, Heidelberg, 2010. Springer-Verlag.

[15] K. Mani Chandy, Jayadev Misra, and Laura M. Haas. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.

[16] Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. *ACM SIGPLAN Notices*, 43(6):304–315, June 2008.

[17] David Maxwell Chickering, David Heckerman, and Christopher Meek. A bayesian approach to learning bayesian networks with local structure. In *Proceedings of the Thirteenth Conference on Uncertainty in Artificial Intelligence*, UAI'97, pages 80–89, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[18] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luís Rodrigues. D^2STM: Dependable distributed software transactional memory. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, pages 307–313, Washington, DC, USA, 2009. IEEE Computer Society.

[19] Ireneusz Czarnowski. Distributed learning with data reduction. In Ngoc Thanh Nguyen, editor, *Transactions on Computational Collective Intelligence IV*, pages 3–121. Springer-Verlag, Berlin, Heidelberg, 2011.

[20] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, October 2007.

[21] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[22] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, 54(4):70–77, April 2011.

[23] Dietrich Featherston. Cassandra: Principles and application. Details available on http://d2fn.com/2010/08/03/cassandra-paper.html, August 2010.

[24] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review*, 23(5):202–210, November 1989.

[25] Rachid Guerraoui and Michał Kapałka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[26] Rachid Guerraoui and Michał Kapałka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[27] Rachid Guerraoui, Michał Kapałka, and Jan Vitek. STMBench7: A benchmark for software transactional memory. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 315–324, New York, NY, USA, 2007. ACM.

[28] Bart Haagdorens, Tim Vermeiren, and Marnix Goossens. Improving the performance of signature-based network intrusion detection sensors by multi-threading. In *Proceedings of the 5th International Conference on Information Security Applications*, WISA'04, pages 188–203, Berlin, Heidelberg, 2005. Springer-Verlag.

[29] Vassos Hadzilacos. A theory of reliability in database systems. *Journal of the ACM*, 35(1):121–145, January 1988.

[30] Tim Harris and Keir Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, October 2003.

[31] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2010.

[32] Richard Henderson, Aldy Hernandez, and Torvald Riegel. Transactional memory in GCC. https://gcc.gnu.org/wiki/TransactionalMemory, April 2015. Accessed: 03 May 2015.

[33] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *ACM SIGARCH Computer Architecture News*, 21(2):289–300, May 1993.

[34] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[35] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[36] Sungpack Hong, Tayo Oguntebi, Jared Casper, Nathan Grasso Bronson, Christos Kozyrakis, and Kunle Olukotun. EigenBench: A simple exploration tool for orthogonal TM characteristics. In *Proceedings of the 2010 IEEE International Symposium on Workload Characterization, IISWC 2010, Atlanta, GA, USA, December 2-4, 2010*, pages 1–11, 2010.

[37] Ekkart Kindler. Serializability, concurrency control, and replication control. In *Transactions and Database Dynamics, Eight International Workshop on Foundations of Models and Languages for Data and Objects, Schloß Dagstuhl, Germany, September 27-30, 1999, Selected Papers*, pages 26–44, 1999.

[38] Guy Korland. Deuce STM – Java software transactional memory. https://sites.google.com/site/deucestm/, 2015. Accessed: 05 June 2015.

[39] Guy Korland, Nir Shavit, and Pascal Felber. Noninvasive concurrency with Java STM. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-2010)*, January 2010.

[40] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *AMC SIGOPS Operating Systems Review*, 44(2):35–40, April 2010.

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

[42] Li Lu and Michael L. Scott. Generic multiversion STM. In Yehuda Afek, editor, *Distributed Computing*, volume 8205 of *Lecture Notes in Computer Science*, pages 134–148. Springer, Berlin, Heidelberg, 2013.

[43] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[44] Cristopher D. Manning, Prabhakar Raghaven, and Hinrich Schütze. *An Introduction to Information Retrieval*. Cambridge University Press, online edition, April 2009. Available at http://nlp.stanford.edu/IR-book/.

[45] Zdravko Markov and Daniel T. Larose. *Eksploracja zasobów internetowych*. Wydawnictwo Naukowe PWN, 2009.

[46] Andrew Moore and Mary Soon Lee. Cached sufficient statistics for efficient machine learning with large datasets. *Journal of Artificial Intelligence Research*, 8(1):67–91, March 1998.

[47] Wojciech Mruczkiewicz, Konrad Siek, and Paweł T. Wojciechowski. Atomic RMI – dokumentacja projektu. https://dsg.cs.put.poznan.pl/gitlab/ksiek/atomic-rmi-sva, November 2014. Accessed: 9 October 2015.

[48] Ron Obermarck. Distributed deadlock detection algorithm. *ACM Transactions on Database Systems*, 7(2):187–208, June 1982.

[49] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC'06, pages 284–298, Berlin, Heidelberg, 2006. Springer-Verlag.

[50] Bernard Roy. *Wielokryterialne wspomaganie decyzji*. Wydawnictwo Naukowo-Techniczne, 1990.

[51] Frank Rubin. The Lee path connection algorithm. *IEEE Transactions on Computers*, C-23(9):907–914, September 1974.

[52] Jim Ruppert. A Delaunay refinement algorithm for quality 2-dimensional mesh generation. In *Selected Papers from the Fourth Annual ACM SIAM Symposium on Discrete Algorithms*, SODA '93, pages 548–585, Orlando, FL, USA, 1995. Academic Press, Inc.

[53] Mohamed M. Saad. HyFlow: A high performance distributed software transactional memory framework. Master's thesis, Virginia Polytechnic Institute and State University, April 2011.

[54] Mohamed M. Saad and Binoy Ravindran. HyFlow: A high performance distributed software transactional memory framework. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 265–266, New York, NY, USA, 2011. ACM.

[55] Mohamed M. Saad and Binoy Ravindran. Snake: Control flow distributed software transactional memory. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 238–252. Springer, Berlin, Heidelberg, 2011.

[56] Mohammed M. Saad and Binoy Ravindran. Distributed hybrid-flow STM. Technical report, Virginia Polytechnic Institute and State University, 2011.

[57] Mohammed M. Saad and Alexandru Turcu. HyFlow TM Wiki. http://old.hyflow.org/hyflow, February 2014. Accessed: 09 March 2014.

[58] Michael Scott. Transactional memory today. *ACM SIGACT News*, 46(2):96–104, June 2015.

[59] Margo I. Seltzer and Ozan Yigit. A new hashing package for UNIX. In *Proceedings of the USENIX Winter 1991 Conference, Dallas, TX, USA, January 1991*, pages 173–184, 1991.

[60] Jonathan Richard Shewchuk. Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In Ming C. Lin and Dinesh Manocha, editors, *Applied Computational Geometry: Towards Geometric Engineering*, volume 1148 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, May 1996.

[61] Konrad Siek and Paweł T. Wojciechowski. Brief announcement: Towards a fully-articulated pessimistic distributed transactional memory. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 111–114, New York, NY, USA, 2013. ACM.

[62] Konrad Siek and Paweł T. Wojciechowski. Atomic RMI: a distributed transactional memory framework. In *7th International Symposium on High-Level Parallel Programming and Applications (HLPP 2014)*, Amsterdam, Netherlands, July 2014.

[63] Dale Skeen and Michael Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, 9(3):219–228, May 1983.

[64] Alexandru Turcu, Binoy Ravindran, and Roberto Palmieri. Hyflow2: A high performance distributed transactional memory framework in Scala. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 79–88, New York, NY, USA, 2013. ACM.

[65] Haris Volos, Neelam Goyal, and Michael M. Swift. Pathological interaction of locks with transactional memory. Technical Report CS-TR-2008-1631, University of Wisconsin–Madison, February 2008.

[66] William E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–283, 1989.

[67] Wikipedia. Ruppert's algorithm. http://en.wikipedia.org/wiki/Ruppert\%27s_algorithm. Accessed: 31 May 2015.

[68] Wikipedia. Sequence assembly. http://en.wikipedia.org/wiki/Sequence_assembly. Accessed: 31 May 2015.

[69] Wikipedia. Software transactional memory. http://en.wikipedia.org/wiki/Software_transactional_memory. Accessed: 20 May 2015.

[70] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 265–274, New York, NY, USA, 2008. ACM.