UNIVERZA V LJUBLJANI
FAKULTETA ZA RACUNALNIŠTVO IN INFORMATIKO

Rok Lenarčič

Programski transakcijski pomnilnik

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2009

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani/-a Rok Lenarčič,

z vpisno številko 63020096,

sem avtor/-ica diplomskega dela z naslovom:

Programski transakcijski pomnilnik

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom
  doc. dr. Boštjana Slivnika
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 9.12.2009 Podpis avtorja/-ice:

Zahvala

Zahvaljujem se mentorju doc. dr. Boštjanu Slivniku za odlično temo, usmerjanje pri raziskovalnem delu, pomoč pri izdelavi naloge, ter za nekatere od najboljših predmetov na tem študiju.

Kazalo

Seznam uporabljenih kratic in simbolov

CMP - chip-multiprocessors
TM - transactional memory
STM - software transactional memory
HTM - hardware transactional memory
HyTM - hybrid transactional memory
CC - concurrency control
LL/SC - load-linked / store-conditional
CAS - compare-and-swap
ACID - atomicity, consistency, isolation, durability
ACI - atomicity, consistency, isolation
RS - read-set
WS - write-set
LVM - lazy version management
EVM - eager version management
MVCC - multi-version concurrency control
LCD - lazy conflict detection
ECD - eager conflict detection
HCD - hybrid conflict detection
RPC - remote procedure call
MWCAS - multi-word compare and swap
CCR - conditional critical region
GC - garbage collector

Povzetek

Diplomska naloga je pregled tehnologij programskega transakcijskega pomnilnika in sorodnih tehnologij. Pisanje programov, ki delujejo v večnitnem okolju, je vedno bila ena izmed težjih nalog programerja. Klasična rešitev je uporaba zaklepanja, s katerim omejujemo sočasne dostope niti do podatkovnih struktur. Na žalost ima zaklepanje veliko število problemov. Ob nepravilni uporabi ali napaki programerja lahko povzroči smrtne objeme niti, zaustavitev programa in nepravilno delovanje. Zaklepanje tudi pretirano omejuje sočasnost dostopov do podatkov. Tehnologija transakcijskega pomnilnika poskuša nadomestiti zaklepanje s tehnologijo, ki je bolj odporna na napake programerja in lažja za uporabo. Omogoča uporabo transakcij, ki so popularen koncept, pri pomnilniških dostopih. Transakcije zagotavljajo atomarnost celih blokov ukazov, ki dostopajo do pomnilnika. Transakcijski pomnilnik je lahko izveden strojno ali programsko.

V tej nalogi so predstavljeni razlogi za razvoj alternative zaklepanju in prednosti transakcijskega pomnilnika. Opisani so tudi drugi mehanizmi za nadzor sočasnosi, splošna struktura programskega transakcijskega pomnilnika in delovanje posameznih komponent, ter hibridni transakcijski pomnilnik. Predstavljeni so problemi, s katerimi se srečujejo razvijalci sistemov programskega transakcijskega pomnilnika, metode testiranja, testna bremena, ter optimizacije delovanja programskega transakcijskega pomnilnika. Naloga se zaključi s pogledom v prihodnost te tehnologije.

Ključne besede:

programski transakcijski pomnilnik, nadzor sočasnosti, slabosti zaklepanja, pregled področja, zagotovilo napredovanja

Abstract

This thesis presents technology overview of software transactional memory and related technologies. Concurrent programming is one of the hardest tasks a programmer can face. The classic solution is the use of locking, which limits concurrent accesses to shared data structures. Unfortunately, locking suffers from a multitude of problems. Incorrect use of locking or programmer error can cause programs to deadlock or malfunction in unexpected ways. Locking also limits the degree of concurrency in a program. The concept of transactional memory is an attempt, to create a concurrency control mechanism, that is less affected by programmer errors and is easier to use. It allows transactions, a popular concept, when making memory accesses. Transactions guarantee atomicity of whole blocks of instructions making memory accesses. Transactional memory can be implemented in software or hardware.

This thesis presents reasons for development of an alternative to locking and the advantages of transactional memory. It describes other mechanisms for concurrency control, software transactional memory general structure and its components. Design problems of software transactional memory, testing methodology and optimizations of software transactional memory are discussed. The thesis wraps up with future outlooks of this technology.

Key words:

software transactional memory, concurrency control, locking downsides, field overview, progress guarantee

Poglavje 1

Uvod

1.1 Multiprogramiranje

V računalništvu je že dolgo v uporabi koncept "*multiprogramiranja*"(angl. multiprogramming). Pri multiprogramiranju teče več procesov navidezno hkrati. Vsak od teh procesov ima lahko tudi več "niti"(angl. threads). Videz hkratnega izvajanja se doseže z izmeničnim zaporednim izvajanjem različnih procesov ali niti na posamezni procesni enoti. Procesorski čas se razdeli na izredno majhne rezine (v operacijskih sistemih se jim reče "kvanti"(angl. quants)), ki jih "*razvrščevalnik*"(angl. scheduler) dodeli enemu ali drugemu procesu. Večprocesorski sistemi, med njih štejejo tudi sistemi z eno večjedrno procesno enoto, so skoraj vedno multiprogramirani in za njih iz vidika teme te naloge veljajo iste ugotovitve kot za enoprocesorske multiprogramirane sisteme (razen, če je drugače določeno). Multiprogramiranje je prisotno v vseh sodobnih računalniških sistemih.

1.2 Nitkanje

"Nitkanje"(angl. threading) oziroma raba večih niti v enem procesu je pogost prijem, ki se ga uporablja predvsem iz dveh razlogov [11]: nitkanje zavoljo priorčnosti in nitkanje zavoljo zmogljivosti.

1.2.1 Nitkanje zavoljo priročnosti

V enoprocesorskih sistemih je velika večina uporabniških programov uporabljala nitkanje samo zaradi lažjega programiranja takih programov. Programerju

ni potrebno skrbeti, da izvajanje enega dela programa ne bi ustavilo izvajanja drugega dela programa. Tipična raba je uporaba ločene niti za uporabniški vmesnik, da ta ostane odziven tudi med procesorsko intenzivnem delovanjem programa [11].

Gre za primere, v katerih nitkanje ne pohitri izvajanja programa. V takih programih je ponavadi ena sama "delovna nit"(angl. worker thread), ki porabi več kot 95% procesorskega časa. V teh primerih skoraj nikoli ni sočasnih pisalnih dostopov večih niti do skupnega pomnilniškega prostora, v nasprotnem primeru pa se uporabljajo preprosti sinhronizacijski mehanizmi.

1.2.2 Nitkanje zavoljo zmogljivosti

Ena nit se ne more izvajati na večih procesnih enotah, zato moramo procesorsko intezivna opravila razbiti na več niti, tipično toliko kolikor je procesnih enot [11], če želimo v večprocesorskem sistemu polno koristiti vse procesne enote. Snop takih niti skoraj vedno dela pisalne in bralne dostope do skupnega medpomnilnika teh niti, kar povzroči sinhronizacijske konflikte.

Običajno se sinhronizacija dostopov do skupnega medpomnilnika izvaja z "zaklepanjem"(angl. locking), kar povzroči povzroči ali padec hitrosti izvajanja ali pa močno povečanje kompleksnosti programa. Z razvojem večjedrnih procesnih enot ta oblika nitkanja *postaja vse pogostejša in pomembnejša.*

1.3 Problem hkratnih dostopov do podatkov

Dostopi neke niti do podatkovne strukture, ki jo druga nit (ali proces) sočasno spreminja, lahko povzroči, da dotična nit vidi podatkovno strukturo v nepravilnem (samo delno urejenem) stanju. Tudi v enoprocesorskem sistemu je ta problem enako prisoten. Ko se razvrščevalnik odloči zamenjati aktivni proces ali nit z drugim, se to zgodi takoj po koncu trenutnega strojnega ukaza (iz perspektive izvajajoče se niti). Torej v takih sistemih je največja izvedljiva enota, ki je sama po sebi "atomarna"(angl. atomic), *en strojni ukaz.* Večina programerjev pa je priučenih predpostavljati, da se bodo vsi ukazi njihovega algoritma izvedli strogo zaporedno.

Tak program v večnitnem okolju ne odpove vedno, kar naredi iskanje takih napak še težavnejše. Klasična razhroščevalna tehnika izvajanja korak za korakom je povsem neuporabna, ker se pri njej, zaradi drugačnih pogojev izvajanja, taki rizični scenariji ne pojavijo. Zato je razvoj pravilno vzporedno izvajajoče kode trenutno en izmed najtežjih problemov na področju razvoja

programske opreme. Rizičnih scenarijev je več [51], najbolj splošni so našteti spodaj.

1.3.1 Umazana branja

"Umazana branja"(angl. dirty reads [51]) je scenarij, v katerem ena nit spreminja podatkovno strukturo, druga nit pa sredi posega podatkovno strukturo uzre v nekonsistentnem ("umazanem") stanju. Pogosto to nekonsistentno stanje tudi krši neke predpostavke programerja o lastnostih in obnašanju programa. Na sliki 1.1 je primer takega programa. V Niti 2 se izvaja koda, ki predpostavlja, da je dolžina seznama konstantna (v tem primeru 100). V niti 1 se izvaja koda, ki zamenja dva elementa seznama. Taka koda sicer ohrani lastnost sistema, da je dolžina seznama konstantna, vendar ob izvajanju obeh niti v vrstnem redu prikazanem na sliki, "Nit 2"vidi seznam z 99 elementi, kar lahko povzroči napačno izvajanje programa.

Nit 1	Nit 2
Na začetku: list.size == 100	

void swap(int first, int second) {
  Object temp = list.remove(second);
 	
  list.put(temp, first);
  list.put(list.remove(first+1), second);
}
	assert(list.size() == 100);

Slika 1.1: Umazano branje

1.3.2 Neponovljiva branja

"Neponovljivo branje"(angl. non-repeatable reads [51]) je ravno obraten scenarij od scenarija "Umazano branje". V tem primeru izvajanje Niti 2 v neprimernem trenutku pokvari izvajanje algoritma v Niti 1. Na sliki 1.2 vidimo, da se spremenljivkama `a` in `b` priredi vrednost iste spremenljike `x` in v nadaljevanju izvajanja Niti 1 potem pričakuje, da vsebujeta isto vrednost. To je razumno pričakovanje, razen če se slučajno koda v Niti 2 izvede ravno med obema prirejanjema v Niti 1.

Nit 1	Nit 2
Na začetku: x == 0	
{	
a = x;	
	x = 10;
b = x;	
assert(a == b);	
}	

Slika 1.2: Neponovljivo branje

1.3.3 Izgubljene spremembe

Drugačen od prejšnjih dveh je scenarij, ki se mu reče "Izgubljena sprememba" (angl. lost update [51]). Če dve niti pišeta sočasno na isto lokacijo, je zapisan podatek ene izmed njih izgubljen, saj je viden samo zadnji zapis na pomnilniško lokacijo. Primer tega je na sliki 1.3, kjer je primer dveh niti, ki povečujeta skupen števec. Podana je tudi razstavitev ukazov na ukaze strojnega jezika (M[] predstavlja dostop do pomnilniške lokacije, R pa procesorski register).

Iz kode strojnih ukazov je razvidno, da lahko, odvisno od medsebojnega zaporedja izvajanja strojnih ukazov, dobimo tri različne končne vrednosti spremenljivke `counter`: "1", "2" ali "3". Končni rezultat, ki bi ga pričakovali od enega poganjanja vsake od niti je vrednost "3". To vrednost bi dobili edino, če se izvajanji obeh niti nič ne prepletata, torej če sta strogo zaporedno izvedeni. Dani primer na sliki tudi odlično kaže, kako so tudi izredno preproste operacije v visokonivojskem programskem jeziku neatomarne in potencialno problematične.

Nit 1	Nit 2
Na začetku: counter == 0	
counter++;	counter += 2;
V zbirnem jeziku:	
R1 <--- M[counter]	R2 <--- M[counter]
R1 <--- R1 + 1	R2 <--- R2 + 2
M[counter] <--- R1	M[counter] <--- R2

Slika 1.3: Izgubljena sprememba

1.4 Kritični odseki

Daleč najpogosteje uporabljana rešitev za zgoraj opisane težave je uporaba kritičnih odsekov. Pri tej rešitvi se opravljajo dostopi do skupnega sredstva (podatkovne strukture ali naprave), ki ga uporabljajo tudi druge niti in procesi, samo v delih kode imenovanih kritični odseki (angl. critical sections). V te dele kode programerji vgradijo sinhronizacijske mehanizme, ki skrbijo za "vzajemno izključevanje"(angl. mutual exclusion), tako da do istega sredstva naenkrat dostopa samo ena nit [48].

Z reševanjem problematike sočasnimih dostopov do naprav se v tej nalogi ne bomo ukvarjali. V praksi se programerji uporabniških programov s tem problemom ne srečujejo, ker dostope do naprav opravlja operacijski sistem.

Do sedaj najpogosteje uporabljani sinhronizacijski mehanizem je "zaklepanje"(angl. locking) z uporabo "ključavnic"(angl. locks). Pri tem mehanizmu vsaka nit ob vstopu v kritični odsek poskusi zakleniti dostop do v nadaljevanju uporabljane podatkovne strukture in si tako zagotoviti ekskluzivni dostop. Če je dostop že zaklenila kakšna druga nit, potem ta nit čaka, da druga nit sprosti dostop in šele nato zaklene dostop do podatkovne strukture. Zatem izvede kodo v kritičnem odseku in sprosti dostop. Sinhronizacija osnovana na podlagi zaklepanja pa lahko vodi do "smrtnega objema"(angl. deadlock), zmanjšane hitrosti sistema, občutljivosti za napake ter drugih problemov [49, 7], zato je programiranje kritičnih odsekov pregovorno težka naloga. Zaradi številnih slabih lastnosti "zaklepanja" so se dolgo iskale alternativne sinhronizacijske sheme.

1.5 Vzpon večprocesorskih sistemov, motivacija za transakcijski pomnilnik

Ideja o transakcijskem pomnilniku ni nova. Zaradi problemov z "zaklepanjem" so se že zelo kmalu pojavile alternative kot je transakcijski pomnilnik, vendar niso zaživele, ker so bile prepočasne ali pa so so privzemale, da ima sistem kakšne lastnosti, ki jih realni sistemi takrat niso imeli. Veliko bolj je dušilo interes za raziskave tudi dejstvo, da so bile alternative tudi deloma nepotrebne.

V preteklih desetletjih je bilo "vzporedno računanje"(angl. parallel computing) nekaj, s čimer se je ukvarjalo le malo programerjev. Večina programerjev ni bila izpostavljena problemom sinhronizacije, ki so prezahtevni za povprečnega razvijalca [53], saj je velika večina računalniških sistemov (namizni in prenosni računalniki ter majhni strežniki) imela samo eno procesno

enoto, torej razvoj uporabniške programske opreme tega znanja ni zahteval. Nitkanje se je uporabljajo predvsem zavoljo priročnosti. S problemi sinhronizacije so se ukvarjali predvsem razvijalci operacijskih sistemov ter razvijalci posebne programske opreme za večprocesorske raziskovalne računalnike, torej specialisti in vse prej kot povprečni programerji. Zato problem prevelike težavnosti sinhronizacije z zaklepanjem v preteklosti ni bil posebno pereč.

Kasneje je razvoj mikroprocesorjev krenil v drugačno smer kot poprej. Študija narejena leta 2000 je ugotovila, da v prihodnosti ne moremo več pričakovati, da se bo hitrost procesnih enot povečevala za 50%–60% na leto kakor v preteklosti [1], ker bo razvoj mikroprocesorske tehnike trčil v povsem fizične omejive, kot so hitrost električnega signala v vezju ter vse večja upornost vse tanjših povezav. Kmalu se je to izkazalo tudi v praksi, kar je povzročilo, da so proizvajalci mikroprocesorjev nadaljevali razvoj v smeri večih procesnih enot, "jeder", na enem samem čipu (angl. chip-multiprocessors — CMP) [39]. Sedaj že skoraj vsak namizni ali prenosni računalnik vsebuje vsaj 2 procesni jedri in zato se morajo sedaj tudi programerji navadnih uporabniških aplikacij priučiti vzporednega programiranja. Spoprijeti se morajo s sinhronizacijskimi problemi, če želijo, da njihova programska oprema dobro izkoristi strojno opremo uporabnikov. Zato se je močno povečalo povpraševanje po preprostejšem načinu sinhronizacije, ki bi približal programiranje aplikacij, ki izkoriščajo vse procesne enote, tudi manj izučenim programerjem, ostalim pa zmanjšal količino porabljenega časa za razvoj in testiranje pravilnosti takih aplikacij. Prihod večjedrnih procesorjev v domačo in pisarniško rabo je torej motiviral obnovljeni interes in raziskave na področju transakcijskega pomnilnika.

Koncept transakcij je preprost in intuitiven ter marsikomu že poznan od podatkovnih baz. Programerju omogoča, da preprosto določi, katere operacije naj se izvedejo atomarno (iz vidika drugih niti), ne da bi moral razmišljati o dejanski izvedbi take sinhronizacije. Za izvedbo poskrbi transakcijski sistem, kar je ravno obratno od zaklepanja. Tam mora, poleg vsebine programa, programer načrtovati tudi samo izvedbo sinhronizacije. Ta lastnost je ena izmed glavnih privlačnosti tehnologije "transakcijskega pomnilnika".

Sledeča poglavja. V poglavju 2 je kratek pregled različnih metod obvladovanja problematike sočasnih dostopov in obravnava slabosti klasičnih prijemov. V poglavju 3 so predstavljeni osnovni deli sistemov programskega transakcijskega pomnilnika, opcijski deli pa so obravnavani v poglavju 4. Operacije programske opreme, ki so problematične za transakcijski pomnilnik, so predstavljene v poglavju 5. Navedene so tudi rešitve, ki so uporabljene za premagovanje teh

ovir. Za popolnejšo sliko so v poglavju 6 prikazane podrobnosti nekaterih izvedb programskega transakcijskega pomnilnika. V poglavju 7 je predstavljena ideja kombiniranja strojnega in programskega transakcijskega pomnilnika. Metode testiranja, rezultati testov ter optimizacijski prijemi so opisani v poglavju 8. Poglavje 9 zaključi diplomsko delo s sklepnimi ugotovitvami.

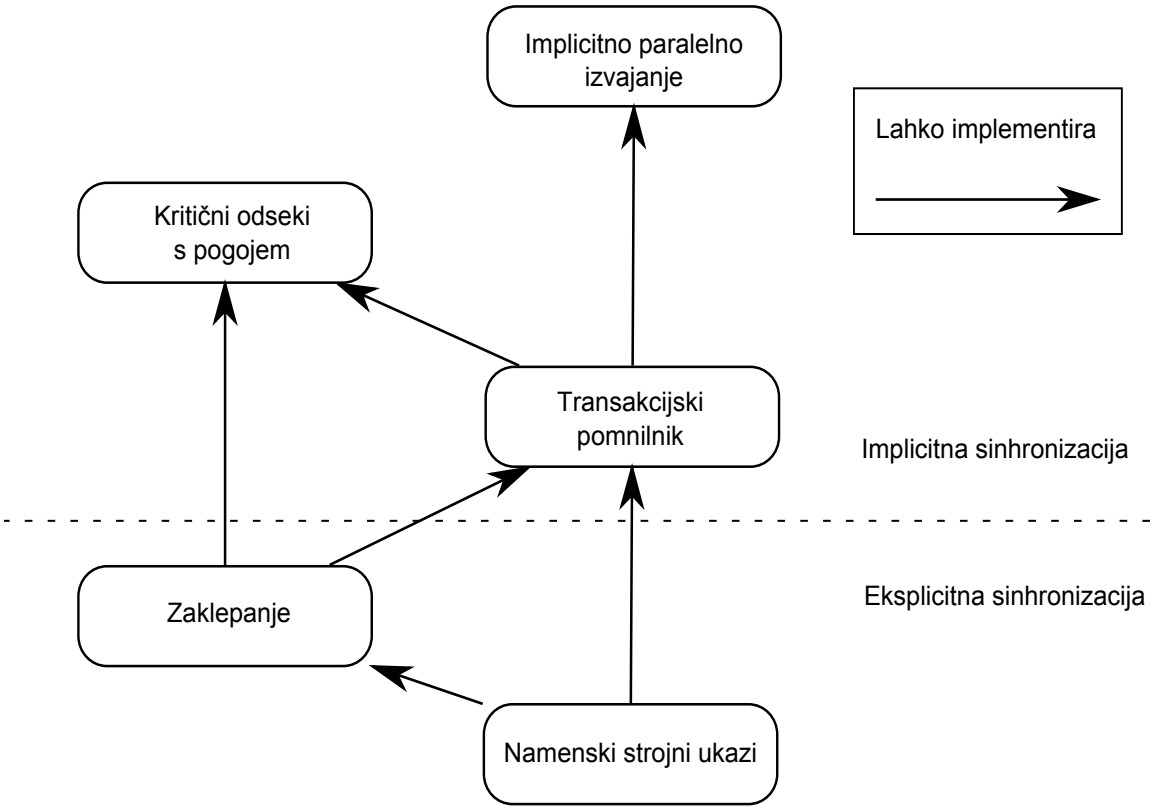
Poglavje 2

Nadzor sočasnosti

Podobno, kot lahko programske jezike razvrstimo po stopnji abstrakcije od izvedbe v strojnih ukazih (strojni ukazi, zbirnik, nizkonivojski jeziki, visokonivojski jeziki, ...), lahko razvrstimo tudi tehnologije za nadzor sočasnosti (concurrency control — CC) in sočasno izvajanje. Njihov odnos je prikazan na sliki 2.1. Tehnologije v spodnjem delu se uporabljajo za eksplicitno sinhronizacijo — programer mora sam napisati njeno izvedbo. V zgornjem delu so tehnologije implicitne sinhronizacije — programer samo določi katere dele kode želi sinhronizirati, izvedba je pa prepuščena izbrani tehnologiji. Sledi kratka obravnava teh tehnologij in njihove povezanosti, na koncu pa še podrobnejša analiza slabosti zaklepanja in pričakovanih lastnosti implementacij transakcijskega pomnilnika.

Optimistične in pesimistične metode. Pogosto se uporablja tudi delitev na optimistične in pesimistične metode CC. Zaklepanje je pesimistična metoda, ker predvideva, da bo najbrž kdo poskusil posegati v zakljenjene podatke med izvajanje kritičnega odseka. Torej, mehanizem ključavnice je uporabljen četudi je trenutna verjetnost takšnega konflikta majhna. To je slabost pesimističnih metod, saj je zaklepanje nepotrebno delo v sistemu, kjer je majhna gostota konfliktnih dostopov [5]. Tudi kadar vsi vpleteni podatke le berejo, se podatki še vedno zaklepajo. Pesimistične metode zato pogosto trpijo zaradi slabe skalabilnosti, ker se ob povečevanju gostote operacij na podatkih povečuje količina časa, ko niti čakajo na sprostitev kjučavnic. Močno je okrnjena sočasnost dostopov, ker je v teh kritičnih odsekih lahko naenkrat samo ena nit.

Optimistične metode pa predvidevajo, da v večini primerov ne bo prišlo do konfliktnih dostopov [8]. Ta predpostavka omogoča, da te metode dopuščajo večjo stopnjo sočasnosti operacij. To je glavni vir prednosti v zmogljivosti za



Slika 2.1: Povezanost tehnologij

transakcijski pomnilnik v primerjavi z metodo zaklepanja [35]. Seveda morajo upoštevati tudi možnost da do takih dostopov pride, za kar transakcijski pomnilnik skrbi s sposobnostjo zaznavanja takih konfliktov in sposobnostjo razveljavljanja transakcij. Za zagotovilo teh dveh sposobnosti je potrebno dodatno knjižiti dostope, kar upočasni take sisteme. To je glavni zmogljivostni problem transakcijskega pomnilnika.

Tu je potrebno omeniti še optimistične implementacije posameznih pogosto uporabljanih podatkovnih struktur. Za podatkovne strukture kot so števec, seznam, polje, sklad obstajajo izvedbe z optimistično metodo sinhronizacije. Imajo pa precejšnje pomanjkljivosti, zaradi katerih je TM včasih še vedno boljša izbira. Najbolj očitna pomanjkljivost je, da so ti algoritmi uporabni samo pri določeni strukturi in jih ne moremo primerjati s splošnimi mehanizmi CC kot so TM in zaklepanje. Pogosto privzemajo, da so na voljo posebni napredni atomarni procesorski ukazi, ponavadi večbesedni atomarni ukazi, ki

jih resnični procesorji nimajo. Poleg tega pogosto odpovejo, če kombiniramo določena zaporedja operacij na njih.

2.1 Namenski strojni ukazi

V sodobnih procesorjih imamo posebne strojne ukaze, katerih glavni namen je omogočenje tehnologij CC višjih nivojev. Gre za ukaze, ki opravijo delo večih standardnih ukazov atomarno. Atomarnost teh operacij je zagotovljena v enoprocesorskem multiprogramiranem sistemu, ker se posamezen strojni ukaz vedno izvrši v celoti ne glede na prekinitve. Ta atomarnost se poruši v večprocesorskih sistemih, kjer lahko drugi procesorji sredi ukaza pišejo v predpomnilnik. Na srečo imajo večprocesorski sistemi vgrajen protokol za konsistenco predpomnilnika (angl. cache consistency protocol), ki skrbi, da se to dogaja samo na pravilen način. Nekateri pogosti ukazi:

- Zamenjaj

  Zamenjaj (angl. swap) je ukaz, ki se uporablja za implementacijo vrteče ključavnice (angl. spinlock), in preprosto zamenja vrednosti dveh registrov v enem koraku.

- Naloži in pogojno shrani

  Par ukazov (angl. load-linked and store-conditional — LL/SC) deluje tako, da LL vrne trenutno vrednost pomnilniške lokacije, SC pa shrani vrednost na to isto pomnilniško lokacijo samo, če ta vmes med ukazoma ni bila spremenjena. LL/SC se lahko uporablja za implementacijo trasakcijskega pomnilnika [50], vendar ni posebno popularna izbira, ker imajo različni procesorji različno močne izvedbe tega para ukazov.

- Primerjaj in zamenjaj

  Ukaz primerjaj in zamenjaj (angl. compare-and-swap — CAS) primerja vrednost pomnilniške lokacije (1. argument) s podano vrednostjo (2. argument) in v primeru ujemanja nadomesti s tretjo vrednostjo (3. argument). CAS se pogosto uporablja v implementacijah programskega transakcijskega pomnilnika (angl. software transactional memory — STM) [25, 34, 27]. CAS je na voljo skoraj na vseh novejših procesorjih, zato je popularna izbira. Obstajajo tudi dvobesedni CAS (angl.

double-word CAS ali DCAS) in večbesedni CAS (angl. multi-word CAS ali MWCAS), ki pa se ne pojavljajo v trenutni generaciji procesorjev.

S temi tipi ukazov lahko torej implementiramo zaklepanje ali pa STM, kot je vidno na sliki 2.1.

2.2 Zaklepanje

Zaklepanje je najbolj razširjena in glavna oblika CC. Veliko se uporablja v operacijskih sistemih, podatkovnih bazah in spletnih strežnikih [48]. Za zaklepanje se uporabljajo različni mehanizmi: semaforji, vrteče ključavnice, monitorji. Pri tem se pogosto uporabljajo posebni strojni ukazi. Pri uporabi v kritičnih odsekih mora nit, ki vstopa, pridobiti lastništvo nad ključavnico, ki ščiti ta kritični odsek. V primeru, da neka druga nit že ima lastništvo ključavnice, ta nit čaka, da se ključavnica sprosti. Ob izstopu iz kritičnega odseka nit sprosti lastništvo ključavnice. Ključavnica je v novejših jezikih pogosto vezana na nek objekt. V programskem jeziku Java je zaklepanje izvedeno z besedo "`synchronized`", ki ima za argument objekt, na katerega je ključavnica vezana. Znotraj posameznega "`synchronized`"bloka je naekrat lahko le ena nit.

Pri programu 1 vidimo lahko vidimo rešitev klasičnega problema proizvajalec — potrošnik z uporabo zaklepanja [7]. Pri tem problemu imamo več niti "proizvajalcev", ki so izvor podatkov in jih dostavljajo, ter več niti "potrošnikov", ki so ponor podatkov, in podatke prevzemajo. Sinhronizacijski problem izhaja iz zahteve, da je vsak dostavljeni podatek prevzet natančno enkrat. Torej proizvajalne niti morajo čakati, da je prejšnji podatek prevzet. Potrošniske niti morajo čakati, da je dostavljen nov podatek, ne pa, da je sočasno s strani več niti prebran isti podatek.

Program 1 ta problem reši tako, da poleg zaklepanja uporabi tudi prijem, da niti spijo, kadar njihovi pogoji za obratovanje niso izpolnjeni. Kadar ena nit spremeni situacijo zbudi vse ostale niti, da se izvajanje nadaljuje v skladu s spremembo.

2.3 Slabosti zaklepanja

Že preprost problem, kot je opisan v programu 1, in relativno preprosta rešitev nam lahko služita za prikaz nekaterih slabosti zaklepanja. Podrobna računlenitev teh in tudi bolj splošnih problemov zaklepanja sledi spodaj [34].

Program 1 Rešitev problema "proizvajalec - potrošnik" z zaklepanjem

```
public int get() {
    synchronized (this) {
        while (!available)
            wait();
        available = false;
        notifyAll();
        return contents;
    }}

public void put(int value) {
    sychronized(this) {
        while(available)
            wait();
        contents = value;
        available = true;
        notifyAll();
    }}
```

2.3.1 Nevarnost smrtnega objema

Eden najbolj perečih problemov zaklepanja je nastanek smrtnega objema pri nepravilno napisanem programu. Primer takega programa je podan na sliki 2.2. Vsaka nit si takoj preskrbi eno ključavnico, kasneje pa zahteva še drugo. V primeru, da se ta dva odseka kode izvajata sočasno, bo nit 1 čakala, da nit 2 sprosti ključavnico nad objektom B, nit 2 pa bo čakala, da nit 1 sprosti ključavnico nad objektom A. V tej situaciji nobena nit ne pride do kode, ki bi ključavnice sprostila in obe niti ostaneta trajno blokirani [26].

Kar dela tak problem posebno pereč, je to da se pojavi samo ob sočasnem izvajanju teh dveh delov kode. Tako se lahko taka napaka pojavlja naključno in zelo redko, uporabniki pa jo zaznajo samo kot ustavitev programa. Posledično jo je zelo težko locirati. Drugi problem je, da se ob povečevanju števila ključavnic povečuje tudi verjetnost, da bosta dve ključavnici v takem odnosu (torej povečuje se verjetnost napake programerja).

Nit 1	Nit 2
{	{
`lock(A);`	`lock(B);`
`...`	`...`
`lock(B);`	`lock(A);`
`...`	`...`
`release(B);`	`release(A);`
`release(A);`	`release(B);`
}	}

Slika 2.2: Problem smrtnega objema

2.3.2 Izgubljeno bujenje

Pri programu 1 lahko vidimo uporabo klica "notifyAll()", s katerim ena nit zbudi druge in jih implicitno opozori na spremembo situacije. Problem je, da ti budilni klici niso povsem zanesljivi. Lahko se izgubijo in kakšna nit ostane neaktivna, kar lahko privede do napačnega izvajanja programa.

2.3.3 Inverzija prioritete

Inverzija prioritete se pojavi, ko si neka nit nizke prioritete lasti ključavnico nad objektom, katerega pozneje poskuša zakleniti nit visoke prioritete. Ta pri tem poskusu zablokira. Ker nit visoke prioritete čaka, niti srednje prioritete nemoteno tečejo, kar povzroči, da nit nizke prioritete ne teče in ne sprosti ključavnice. Zato nit visoke prioritete ostane blokirana. Nit srednje prioritete se izvaja, nit visoke pa ne — inverzija prioritete [26].

Obstajajo rešitve tega problema, vendar niso preproste in imajo druge probleme.

2.3.4 Konvojno izvajanje

Konvojno izvajanje (angl. convoying) je pojav, da veliko niti čaka pred kritičnim odsekom, v njega vstopajo in ga zapuščajo posamično, ena za drugo. Kadar je nit, ki je v kritičnem odseku, zapostavljena (angl. preempted), ostale niti ne napredujejo [26]. Ta pojav je izrazit pri zaklepanju, ker je pesimistična metoda CC in dopušča le eno nit v kritičnem odseku. Kritični odsek je torej kot zožitev na cesti, skozi katero morajo vozila v konvoju.

Negativna posledica je očitna. Ne glede na količino vzporednosti v programu, ne glede na število niti in procesorjev, kritični odseki so ozko grlo, kjer teče ena nit na enem procesorju in to ima močan vpliv na hitrost izvajanja [33].

2.3.5 Zmanjšana robustnost kode

Koda z zaklepanjem je tudi zelo občutljiva na napake programske opreme. Če med tekom izvajanja neke niti ali procesa pride do nepričakovane napake, se lahko zgodi, da ta nit ali proces umre. Ta mehanizem je lahko precej nenevaren za stabilnost sistema in je pogosto povsem dopusten. Povsem drugače je, če nit umre medtem, ko ima ta nit lastništvo nad kakšno ključavnico. Vse ostale niti in procesi, ki bodo potrebovali lastništvo nad to isto ključavnico, bodo trajno blokirani [48].

2.3.6 Težavno združevanje odsekov zaščitene kode

Program 2 Zamenjava dveh elementov seznama

```
Vector vector = new Vector();
...
public void swap(int first, int second) {
    synchronized (this) {
        Object temp = vector.remove(second);
        vector.add(first, temp);
        vector.add(second, vector.remove(first+1));
}}
```

Pogosto programerji uporabljajo podatkovno strukturo, ki uporablja zaklepanje v osnovnih operacijah, da zagotavlja pravilno notranje stanje v večnitnem okolju. Taka koda je lahko dobro preverjena in njena pravilnost zagotovljena. Te lastnosti so povsem brez pomena, če programer hoče uporabiti sestavljeno operacijo. Podan je primer programa v Javi, program 2. V tem primeru je programer uporabil razred "Vector", ki ima z zaklepanjem zavarovane dostope, tako da lahko samo ena nit naekrat ali odstranjuje element ali pa dodaja element.

Problem je, da preprosto kombiniranje teh dveh posamično pravilnih operacij ne da pravilnega rezulata. Tudi, če v tej kompozitni operaciji uporabimo

zaklepanje (kot je v programu 2), še vedno ne dobimo vedno pravilnega programa. V zgornjem primeru je število niti v funkciji "swap"resda omejeno na eno, vendar nič ne preprečuje, da bi kakšna nit sočasno opravljala vstavljanje ali odstranjevanje elementov, kar lahko vodi do napačnih rezultatov.

Če želi programer funkcionalnost razširiti na pravilen način, mora torej posegati v tujo kodo izven funkcije, ki jo piše (in jo tudi poznati). To je izredno nezaželjena lastnost, ker je uporaba že napisane kode že iz ekonomskega stališča nuja.

Problematično je tudi, da zaklepanje ne nudi nobene zaščite pred posegi tuje kode v podatkovne strukture, ki jih z zaklepanjem ščitimo.

2.3.7 Zrnatost

Zrnatost je parameter nekega sistema ključavnic. Ko snujemo sistem zaklepanja, se odločamo med manjšo ali večjo zrnatostjo zaklepanja. Groba zrnatost pomeni, da vse dostope zaščitimo z eno veliko ključavnico. Slabost tega je, da to močno škoduje vzporednosti izvajanja, ker lahko samo ena nit dostopa do velikega kosa programa. Drobna zrnatost pomeni rabo večih ključavnic, ki zaklepajo majhne dele programa. Tak pristop omogoča večjo vzporednost izvajanja, ampak ima slabosti kot sta hitro rastoča kompleksnost in počasnost izvirajoča iz porabe procesorskega časa s strani ključavnic [48].

2.4 Transakcijski pomnilnik

Ideja transakcij kot sinhronizacijski mehanizem je stara, najbolj vidno je uporabljana v podatkovnih bazah. Toda minilo je precej desetletij, preden so se transakcije pojavile v obliki, ki ji rečemo transakcijski pomnilnik. Prva z idejo transakcijskega pomnilnika sta bila Herlihy in Moss leta 1993 [26]. Njuna prva ideja je bil strojni transakcijski pomnilnik, ki je bil precej preprosta razširitev obstoječih protokolov za konsistentnost predpomnilnika (angl. cache consistency protocol). Imel je precej omejitev celo za strojno izvedbo, vendar je bil dovolj obetaven, da je spodbudil dodatne raziskave v smeri tega koncepta. Shavit in Touitou sta prva predlagala povsem programski transakcijski pomnilnik leta 1995 [50]. Le-ta je bil statičen, kar pomeni, da mora uporabnik vnaprej poznati lokacije, ki jih bo spreminjal. Herlihy et al. so pa predstavili prvi dinamični programski transakcijski pomnilnik [27]. Kot je razvidno iz slike 2.1, so transakcijski pomnilniki lahko izvedeni z posebnimi strojnimi ukazi (kot je CAS) ali pa z zaklepanjem.

Koncept transakcij. Da lahko nekemu sistemu rečemo, da uporablja koncept transakcije, mora ta sistem izpolnjevati nekaj osnovnih pogojev. Transakcija je zaporedje ukazov, ki zagotavlja "vse ali nič"(angl. all-or-nothing) rezultate. Če pride do uspešne potrditve (angl. commit), postanejo rezultati vseh ukazov vidni drugim nitim atomarno. Če je razveljavljena, potem pa imamo isto stanje, kot smo imeli pred začetkom transakcije. Transakcija je ločena od okolja z začetno točko in eno ali več končnimi točkami. Vse poti izvajanja od začetne točke do katerekoli končne točke so vsebina transakcije. Transakcija se izvaja znotraj nekega transakcijskega konteksta, ki predstavlja izvajalno okolje za neko določeno transakcijo.

V primeru podatkovnih baz se pogosto opisuje zahtevane lastnosti transakcij s kratico ACID — atomarnost, konsistentnost, izoliranost, vzdržljivost (angl. atomicity, consistency, isolation, durability) [52].

- Atomarnost

  Transakcija se izvrši v celoti, ali pa sploh ne.

- Konsistentnost

  Pred transakcijo in po njej so podatki v konsistentnem stanju.

- Izoliranost

  Ostale niti med izvajanjem transakcije ne morejo videti njenega vmesnega stanja.

- Vzdržljivost

  Ko je transakcija enkrat uspešno zaključena, ne more biti razveljavljena.

Zadnji element, "Vzdržljivost", se na transakcijski pomnilnik ne nanaša, namenjen je zagotavljanju pravilne vsebine podatkovne baze v primeru izpada delovanje med operacijo "potrdi".

2.4.1 Prednosti transakcijskega pomnilnika

Transakcijski pomnilnik rešuje večino zgoraj opisanih problemov zaklepanja.

- Lažje pisanje pravilnih programov

Programerjem ni potrebno poznati kode v drugih delih programa. Poleg tega je združevanje delov kode, ki uporabljajo transakcije, preprosto.

- Prepreči smrtne objeme

  Transakcije se lahko razveljavi kadarkoli, zato se lahko smrtnemu objemu preprosto izognemo tako, da razveljavimo eno od transakcij v konfliktu in jo ponovno zaženemo.

- Lažje doseglivo hitro izvajanje

  Programerjem ni potrebno pisati drobno zrnatih zaklepanj zato, da bi omogočili sočasne dostope, ta funkcionalnost je pri transakcijah samodejna.

- Prepreči inverzijo prioritete

  Če pri izvajanju transakciji preprečuje nadaljevanje konflikt s transakcijo nižje prioritete, se lahko "sovražno"transakcijo razveljavi.

- Odpornost na napake

  Če nit v transakciji izvede napačno operacijo in umre se transakcija avtomatsko razveljavi in se tako podatki ohranijo v konsistentnem stanju.

2.4.2 Zapis operacij trasakcijskega pomnilnika

Transakcijski pomnilnik je v kodi lahko uporabljan z eksplicitnimi klici njegovih funkcij ali pa implicitno. Eksplicitni klici obsegajo vsaj nabor ukazov "začni transakcijo"(angl. begin, start) in "potrdi transakcijo". Primer je podan v programu 3. Mnoge implementacije ponujajo uporabniku tudi druge ukaze. Lahko pa je začetek in konec transakcije označen z začetkom in koncem posebnega bloka kode, kot je v programu 4. V tem primeru se implicitno kliče primerno funkcijo transakcijskega pomnilnika na začetku in koncu bloka. V kolikor implementacija TM dopušča programerju uporabo še kakšnih drugih ukazov (na primer ukaz za razveljavitev transakcije), se ti pojavljajo eksplicitno.

Program 3 Eksplicitni klici

```
Vector vector = new Vector();
...
public void swap(int first, int second) {
    TM_Begin();
        Object temp = vector.remove(second);
        vector.add(first, temp);
        vector.add(second, vector.remove(first+1));
    TM_Commit();
}}
```

Program 4 Implicitni klici

```
Vector vector = new Vector();
...
public void swap(int first, int second) {
    atomic {
        Object temp = vector.remove(second);
        vector.add(first, temp);
        vector.add(second, vector.remove(first+1));
    }
}}
```

2.5 Kritični odseki s pogojem

Kritični odseki s pogojem so že več kot 35 let star koncept [29]. Klasična raba je prikazana v programu 5. Na prvi pogled je podobna implicitnim transakcijam. Vendar je ob besedi "atomic"tudi pogoj. Šele ko je pogoj izpolnjen, se izvrši telo bloka. Telo bloka se izvrši podobno kot pri transakcijskem pomnilniku: atomarno in izolirano, kot da bi se izvrševal v sistemu z eno nitjo [20].

Privlačnost tega mehanizma CC je podobna kot pri transakcijskem pomnilniku. Programer samo pokaže, katere skupine ukazov naj se izvršijo izolirano, izvedbo pa prepusti sistemu [21]. Kljub temu, da gre za tako star in privlačen koncept, se ta do sedaj ni uveljavil. Glavni razlog je bil, da v preteklosti ni bilo nobenih dobrih implementacij tega koncepta [4, 3].

Prve implementacije so bile zgrajene na semaforjih in so dopuščale izvajanje samo enega CCR hkrati. Problemtično je bilo tudi, da so ponovno preverjale vstopne pogoje pri vseh blokiranih CCR preveč pogosto. Rezultat je zelo slaba hitrost CCR zato je bil koncept neuporabljan. S prihodom transakcijskega pomnilnika pa obstaja nov način implementacije CCR (kar je tudi predstavljeno na sliki 2.1).

Medtem ko imajo implementacije z zaklepanjem precejšnje probleme z vprašanjema kako zagotoviti atomarno izvajanje ter kako pogosto preverjati vstopne pogoje (pogostejša preverjanja močno upočasnijo hitrost celega sistema), implementacije s transakcijskim pomnilnikom teh problemov nimajo in se konceptu CCR zelo podajo. CCR s trivialnim pogojem (npr. v skladu s sintakso programa 5 bi trivialni pogoj bil "atomic(true)") je pomensko povsem ekvivalenten transakciji pri transakcijskem pomnilniku. Transakcijski pomnilnik z lahkoto zagotovi atomarnost, ki jo zahteva CCR.

Poleg tega transakcijski pomnilnik rešuje tudi problem preverjanja pogoja. Če je CCR izveden s transakcijskim pomnilnikom in se začne z začetkom transakcije, kateremu sledi preverjanje pogoja, nato pa nit blokira, potem množica prebranih pomnilniških lokacij te transakcije natančno odraža pomnilniške lokacije vseh podatkov uporabljenih pri preverjanju pogoja.

Transakcijski pomnilnik že sam po sebi implementira mehanizme za zaznavanje kdaj je prišlo do pisanj na pomnilniške naslove, ki so v množici branih pomnilniških lokacij neke transakcije. Ker te lokacije v času čakanja obsegajo samo lokacije prebrane pri računanju pogoja, je torej vsak tak konflikt znak, da je potrebno pogoj ponovno izračunati. Torej je pri implementacijah s transakcijskim pomnilnikom število preverjanj pogoja vedno idealno.

Ker je modifikacija iz STM v CCR precej enostavna, CCR pa zelo uporaben koncept, kar nekaj nekaj implementacij STM, kjer implementirajo tudi CCR

[21, 20].

Program 5 Kritični odsek s pogojem

```
atomic (size != 0) {
    return bufArray(size--);
}
```

2.6 Implicitno vzporedno izvajanje

Implicitno vzporedno izvajanje je mehanizem, ki razbremeni programerja, da mu ni potrebno razdeljevati programa na več niti zato, da bi izkoristil več procesorjev. Implicitno vzporedno izvajanje to opravilo avtomatizira, bodisi deloma (samo za določene dele programa) ali v celoti. Primer delnega sistema je naprimer "Atomos" STM [7], ki poleg implemntacije STM omogoča tudi špekulativno vzporedno izvajanje večih iteracij zanke.

Pri vseh teh idejah potrebujemo sinhronizacijski mehanizem, ki je precej splošen, lahek za uporabo in kombiniranje. STM je zato ustrezen in njegov prihod omogoča lažje ustvarjanje takih sistemov.

Poglavje 3

Osnovna struktura in delovanje programskega transakcijskega pomnilnika

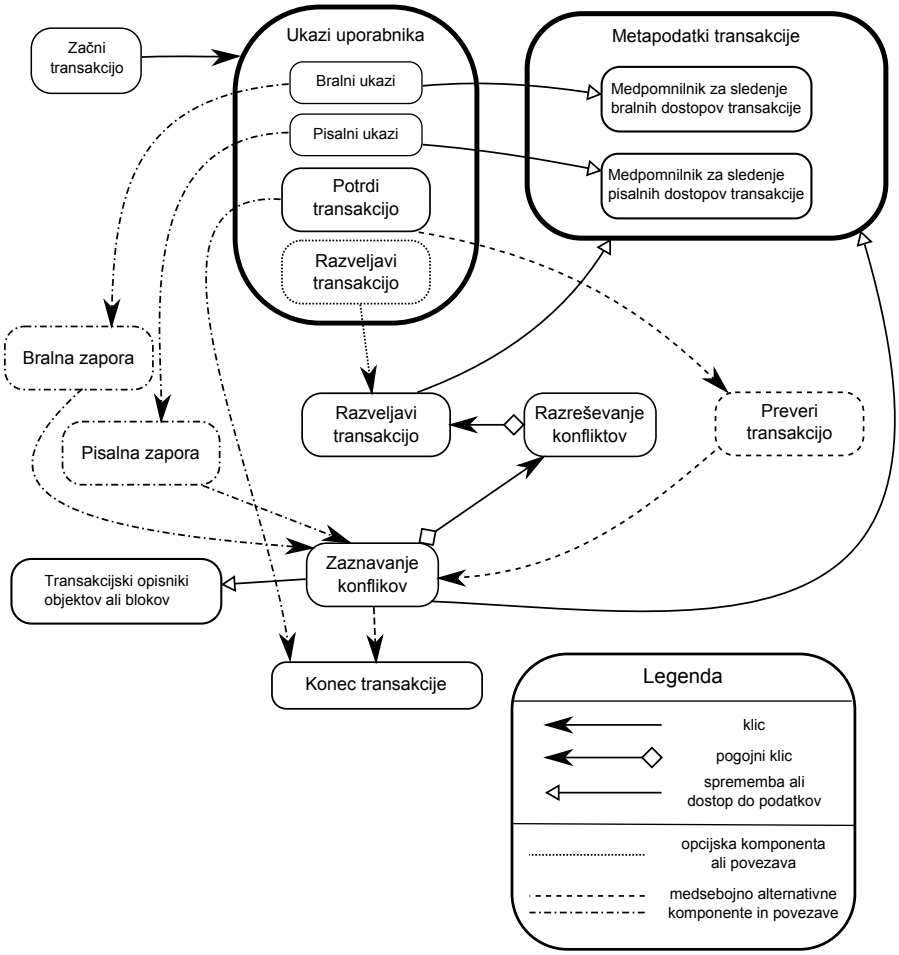
Na sliki 3.1 je prikazana splošna oblika STM, ki zajema najpogostejše ideje in izvedenke. Kljub relativni zapletenosti skice le-ta ne obsega vseh možnih STM izvedb, temveč samo nekatere glavne oblike. Podrobnosti delovanja posameznih sistemov in njihove razne izpeljanke bodo obravnavane v sledečih podpoglavjih. Še pred tem si bodo pa pogledali nekaj lastnosti, po katerih se STM izvedbe razlikujejo.

3.1 Dinamičnost in statičnost

STM izvedbe lahko delimo na statične in dinamične (dynamic software transactional memory — DSTM). Pri statičnem STM je dodana zahteva, da mora programer že vnaprej določiti pomnilniške lokacije, ki bodo uporabljane v transakciji. Pri statičnem STM je transakcija torej neke vrste MWCAS. Ponavadi je transakciji ob začetku kot parameter podan vektor pomnilniških lokacij, ki bodo uporabljane.

Prva izvedba STM, ki sta jo predstavila Shavit in Toitou [50], je bila statična. Kasneje so prevladale dinamične izvedbe STM. Sedaj so statični STM skoraj povsem neobstoječi. Čeprav je mogoče s statičnim STM narediti vse, kar je mogoče z dinamičnimi [50], pa je slednji močno priročnejši za uporabo.

V nadaljevanju o statičnih STM ne bomo govorili, zato bodo dinamični STM označeni preprosto STM namesto DSTM.



Slika 3.1: Shema preprostega programskega transakcijskega pomnilnika

3.2 Zagotovilo napredovanja

Izvedbe STM se razlikujejo med sabo tudi po tem, kakšna zagotovila ponujajo, da bo delo posamezne ali vseh niti opravljeno v omejenem časovnem obdobju.

Razvrstitev nekaterih sistemov STM glede na to lastnost je podana na sliki 3.2.

Zrnatost	STM
Neblokirajoči	OSTM [12], WSTM [21], ASTM [36], DSTM [27], Ananian STM [9], SXM [18], RSTM [37]
Blokirajoči	Ennals STM [11], Bartok STM [24], AUTOLOCKER [38], Haskell STM [47], McRT-STM [49], TL [10]

Slika 3.2: Delitev sistemov STM glede na zagotovilo napredovanja

3.2.1 Neblokirajoči sistemi

Sistemi so neblokirajoči (angl. non-blocking), če neuspešnost poljubnega števila niti ne ustavi napredovanja preostalih niti (kot celote, torej preostalega dela sistema) [21]. Torej v praksi to pomeni, da procesi napredujejo ne da bi blokirali druge procese in da vedno vsaj ena izmed aktivnih transakcij uspe. V to kategorijo ne spada sinhronizacija z zaklepanjem in STM sistemi, ki pri svojem delovanju uporabljajo zaklepanje. Neblokirajoči sistemi se dodatno delijo.

Sistemi brez čakanja

Sistemi brez čakanja (angl. wait-free) so vsi tisti, ki zagotavljajo, da vsaka nit, ki poskuša dostopati do skupnih podatkov, zaključi operacije v omejenem časovnem obdobju merjenem v času te niti same [31]. Tak sistem mora zagotavljati, da vse niti napredujejo, ne glede na odpovedi ostalih niti in konflikte. Ali drugače, vsak proces, ki večkrat zaporedoma poskuša izvesti neko transakcijo, jo uspe zaključiti v končnem številu poskusov [50]. Ponavadi se to zagotovilo doseže z mehanizmi, ki omogočajo zaznavanje problematičnih situacij, in mehanizmi, ki omogočajo posamezni niti, da pomaga drugi niti dokončati delo. Torej namesto da bi v konfliktnih situacijah eno od transakcij samo razveljavili, ji druge niti pomagajo [43].

V praksi te lastnosti pomenijo, da pri sistemih brez čakanja ne more priti do smrtnega objema transakcij, prav tako ne more priti do izstradanja katere od niti. Problem teh sistemov je precejšnja upočasnitev, ki jo prinesejo kompleksni mehanizmi medsebojne pomoči niti, prav tako zagotavljanje teh lastnosti zelo močno omeji možnosti avtorjev STM za optimizacijo učinkovitosti. Izvedbe STM s tako močnim zagotovilom napredovanja so zato izjemno redke.

Sistemi brez zaklepanja

Sistemi brez zaklepanja (angl. lock-free) so vsi tisti, ki zagotavljajo, da v primeru konfliktov in zakasnitev ter odpovedi nekaterih niti vsaj ena nit, ki poskuša dostopati do skupnih podatkov, zaključi v omejenem časovnem obdobju merjenem v času te niti same [25, 12]. Tak sistem zagotavlja, da se smrtni objemi ne bodo zgodili, dopušča pa izstradanje niti (če ena izmed niti nikoli ne konča je zahtevam zadoščeno v kolikor vsaj kakšni drugi niti uspe zaključiti). STM izvedba, ki zadostuje tem zahtevam je lahko preprostejša in hitrejša od izvedbe brez čakanja, in je zato v praksi pogostejša, kljub temu da ima šibkejša zagotovila napredovanja.

Ime "brez zaklepanja"je zelo zavajujoče. V preteklosti se je v literaturi ta izraz enačil z "neblokirajoč"[40]. Treba je poudariti, da niso vsi sistemi, ki ne uporabljajo zaklepanja (angl. locking), sistemi brez zaklepanja po tej definiciji. Lahko spadajo v kako drugo kategorijo neblokirajočih sistemov. Sistemi, ki uporabljajo zaklepanje ne spadajo v nobeno kategorijo neblokirajočih sistemov.

Sistemi brez oviranja

Sistemi brez oviranja (angl. obstruction-free) so vsi tisti, pri katerih je napredovanje zagotovljeno v primeru, da se ta nit izvaja izolirano. Tako kot sistemi v prejšnjih dveh kategorijah v takem sistemu posamezno nit ne ustavijo odpovedi ostalih niti [25]. Sistemi brez oviranja so ranljivi tako za smrtne objeme transakcij kot tudi za izstradanje niti. Na prvi pogled se zdi, da so taki sistemi pretirano ranljivi, da bi bili uporabni, vendar so ti problemi v praksi povsem rešljivi.

Sistemi brez čakanja in brez zaklepanja so rezultat poskusov, da bi razvili sisteme, ki hkrati z matematično dokazljivo pravilnostjo posedujejo tudi dokazljivo sposobnost napredovanja v večnitnem okolju z dostopi do skupnih podatkov. Sistemi brez oviranja pa uberejo drugačno pot in omogočajo programerjem, da rešujejo problem napredovanja ločeno od problema pravilnosti. Vse kar je od sistema brez oviranja pričakovano, je sposobnost napredovanja, ko teče samo ena nit (druge niti so lahko v poljubnem stanju) [31].

Da ne pride do izstradanja niti, se uporabljajo različni mehanizmi za razsojanje sporov. Najpreprostejša rešitev je naprimer mehanizem eksponentnega odmikanja (angl. exponential backoff), pri katerem v primeru konflikta ena od niti razveljavi drugo, ki počaka kratek čas preden se ponovno poskusi izvesti. Če ponovno pride do konflika, se postopek ponovi, le da nit počaka daljši čas preden ponovno poskusi. Ta čas se povečuje eksponentno. Nekatere izvedbe STM uporabljajo posebne module imenovane "razsodniki sporov", ki

omogočajo izvedbo poljubnih taktik spopadanja s konflikti.

Čeprav te rešitve ne prinesejo 100% zagotovila napredovanja, so več kot dovolj dobre, da taki sistemi v praksi odlično delujejo. Šibkejše zagotovilo napredka omogoča več svobode pri optimizacijah izvedb STM, zato taki STM dosegajo večje hitrosti, kot ostali neblokirajoči STM. Ni presenečenje, da je velika večina neblokirajočih STM brez oviranja.

Podrobnosti o različnih razsodnikih sporov bodo obdelane v poglavju o opcijskih komponentah STM.

3.2.2 Blokirajoči sistemi

Sistemi, ki uporabljajo zaklepanje, so blokirajoči. Sem spadajo tudi nekateri STM sistemi, ki uporabljajo zaklepanje. Podobno kot pri sistemih brez oviranja se uporaljajo ločeni mehanizmi za odpravljane slabosti takih sistemov, kot so smrtni objemi.

Čeprav imajo teoretično taki sistemi STM slabosti zaklepanja, se v praksi ti problemi ne pojavljajo. Raba zaklepanja je standardizirana v STM sistemu in je preverjeno pravilna, kar je drugače od navadnega zaklepanja, kjer vsak programer piše svoja zaporedja zaklepanja (možnost napak). Mehanizmi za zaznavanje in preprečevanje smrtnih objemov so že vključeni v izvedbo STM in niso prepuščeni programerju. Pojav smrtnega objema je v STM sistemu preprosto rešljiv z razveljavitvijo ene od vpletenih transakcij. Pri navadnem zaklepanju imamo edino možnost ukinitve ene od niti, kar pa ne odstrani sprememb, ki jih je nit vnesla preden je prišla v smrtni objem. Podobno pri inverziji prioritete lahko transakcijo z nižjo prioriteto preprosto razveljavimo, pri zaklepanju pa moramo uporabiti posebne rešitve, kot je rekurzivno dedovanje prioritete. Območja zaklepanja so tudi razmeroma kratka oziroma ključavnice so drobne, kar se izogne problemu klasičnega zaklepanja, konvojnemu izvajanju.

Nekatere raziskave trdijo, da je lastnost, da je sistem brez oviranja, nepotrebna [11], ker se da pomanjkljivosti blokirajočih STM odpraviti s primernim obnašanjem razvrščevalnika sistema. Blokirajoči STM so vse pogostejša izbira po letu 2006. Izkaže se, da omogočajo prijeme, ki precej pohitrijo STM sistem (večja lokalnost metapodatkov ter manj indirektnih dostopov). Nekatere opcijske funkcionalnosti v STM sistemih zahtevajo blokirajoče obnašanje, npr. klici RPC (angl. remote procedure call). V takih primerih smo prisiljeni v blokirajočo implementacijo STM.

V takih STM se konflikti pokažejo kot sočasne zahteve po istih ključavnicah. Če transakcija poskuša dostopati do podatkov, ki jih je že spreminjala kakšna

druga aktivna transakcija, bo naletela na ključavnico, ki si jo že lasti druga transakcija. Če pa druga transakcija naknadno (toda pred potrditvijo transakcije) spremeni podatke, do katerih je ta transakcija že dostopala, se konflikt pokaže v koraku preverjanja transakcije.

3.3 Začetek transakcije

Vsaka transakcija se začne z operacijo začetka transakcije, ki jo bodisi kliče uporabnik STM (eksplicitno) bodisi pa je točka njenega začetka podana implicitno in klic operacije začetka transakcije v prevedeno kodo vstavi kar prevajalnik. Oba primera sta ista s stališča STM sistema. Ta operacija ustvari novo transakcijo v trenutni niti izvajanja. Transakcija je tudi umeščena v nek transakcijski kontekst (angl. transactional context). Ta pojem ponavadi označuje okolje, v katerem se izvaja transakcija in ki hrani njene zasebne meta-podatke. Zatem se tudi dodeli prostor za meta-podatke transakcije (njen opisnik in razni medpomnilniki). V opisniku se status transakcije postavi v stanje "aktivna".

3.4 Transakcijski opisnik

Transakcijski opisnik objekta ali bloka podatkov (angl. transactional descriptor) so ponavadi podatki, ki opisujejo ali lastnika podatkov (transakcijo) ali pa njihovo trenutno verzijo poleg tega pa vsebujejo tudi kazalec na medpomnilnik, ki hrani spremembe, ki jih je povzročila transakcija, ki je lastnica. Ti podatki se uporabljajo za sledenje spremembam bloka ali objekta za potrebe postopka razveljavitve transakcije ali pa za potrebe izolacije sprememb transakcije lastnice. Pri blokirajočih sistemih je lahko dodana še ključavnica, ki sčiti ta objekt.

3.5 Dostopi uporabnika do podatkov

Jedro transakcije predstavlja koda uporabnika, v kateri se vršijo različni ukazi, ki pa imajo za posledico tudi dostope do pomnilnika. Ob prvem dostopu do nekega objekta ali bloka se pripravi transakcijski deskriptor za ta objekt ali blok. Nekateri STM sistemi zahtevajo od uporabnika, da to signalizira eksplicitno, tako da objekt pred prvim dostopom s posebnim ukazom "odpre"[27]. Dostope, ki jih opravijo ukazi je potrebno beležiti za potrebe zazavanja konfliktov ter za potrebe operacije razveljavljanja transakcije.

Shranjuje se dve vrsti podatkov. Transakcija mora shranjevati podatke, kateri objekti ali bloki so bili dostopani. To je z namenom zaznavanja konfliktov, kjer je potrebno ugotoviti, katere objekte ali bloke je potrebno preveriti. Nekje morajo biti tudi shranjeni podatki o tem, kako so bili podatki spreminjani v transakciji, torej transakcijski opisniki objektov ali blokov. To je lahko shranjeno v posebnem medpomnilniku s povezavo od objekta do medpomnilnika in od medpomnilnika do transakcije, ki je lastnica teh sprememb. Ta medpomnilnik je lahko pridružen objektu samemu.

Pogosto je STM tako narejen, da se ob bralnih dostopih samo zabeleži dostop in ponekod tudi verzija podatkov, ki jih je transakcija uzrla, ob pisalnih dostopih pa se tudi prevzame lastništvo nad podatki. Ta dinamika nam omogoča, da se sočasni bralni dostopi do istih podatkov opravljajo brez problema, sočasni pisalni dostopi do istih podatkov privedejo do konflika (dostop, ki je drugi v verigi naleti na tuje lastništvo objekta), ter v primeru mešanih dostopov lahko zaznamo konflik, da je prišlo do pisalnega dostopa med bralnim dostopom v neki transakciji in njeno potrditvijo.

3.5.1 Lokacija transakcijskih podatkov

Obstajata dva različna pristopa pri izbiri lokacije za podatke, ki nastopajo v transakcijah [13].

Razvrstitev nekaterih sistemov STM glede na to lastnost je podana na sliki 3.3.

Lokacija trans. podatkov	STM
Ločeno	OSTM, WSTM, ASTM, DSTM
Skupaj ter ločeni metapodatki	Bartok STM, RSTM, McRT-STM

Slika 3.3: Delitev sistemov STM glede na lokacijo transakcijskih podatkov

Transakcijski in navadni podatki ločeno

Nekatere izvedbe shranjujejo podatke, ki (lahko) nastopajo v transakcijah v ločenem pomnilniškem prostoru in ne morejo do njih dostopati neposredno [27, 36, 12]. Če hoče transakcija dostopati do enega od teh objektov, ki lahko nastopajo v transakcijah (transakcijski podatki), potem morajo tak objekt "odpreti"(omogočiti dostop s posebnim klicem. Ponavadi takšne izvedbe STM dopuščajo, da klicatelj določi za kakšne dostope bo odprl objekt (samo za

bralne ali pa za bralne in pisalne). To omogoča STM, da optimizira dostope, kadar so le-ti samo bralni, ker lahko več transakcij hkrati dostopa do istega telesa objekta. V primeru, da nekdo odpre objekt za pisalne dostope, se pripravi kopija podatkov, ki je vidna samo tej transakciji in nad njo transakcija izvaja tako pisalne kot bralne dostope.

Takšni STM sistemi imajo najpogosteje dvonivojsko preslikovanje. Od transakcijskega objekta do nekega vmesnega objekta, ki hrani kazalca na staro in novo verzijo telesa objekta in od vmesnega objekta do oprimka objekta (angl. object handle) samega, kar omogča, da sistem najde transakcijske podatke nekega objekta.

Problem teh preslikovanj je, da vsak nivo lahko privede do ene dodatne zgrešitve v predpomnilniku ob vsakem dostopu, kar povzroča precejšnje upočasnitve. ASTM ima zaradi tega pri bralnih dostopih samo en nivo preslikovanja [36].

Transakcijski podatki skupaj z navadnimi podatki z ločenimi metapodatki

Druge izvedbe shranjujejo transakcijske podatke v isti pomnilniški prostor z navadnimi podatki. Tako se jih lahko naslavlja kar z navadnim kazalcem in je zato eno preslikovanje manj. Tak pristop tudi omogoča večjo lokalnost dostopov in zato višje hitrosti.

3.5.2 Upravljanje z verzijami

STM sistemi se ločijo tudi po načinu njihovega ravnanja z različnimi verzijami podatkov. STM mora hraniti tako staro verzijo podatkov (stanje pred začetkom neke transakcije) za potrebe operacije razveljavitve kot tudi nove verzije, ki vključujejo spremembe podatkov s strani ene od niti. Obstajajo trije glavni pristopi do upravljanja z verzijami: leno upravljanje z verzijami (angl. lazy version management – LVM), takojšnje upravljanje z verzijami (angl. eager version management — EVM) in večverzijski nadzor sočasnosti (angl. multi-version concurrency control — MVCC).

Razvrstitev nekaterih sistemov STM glede na to lastnost je podana na sliki 3.4.

Upravljanje z verzijami	STM
Leno	TL, WSTM, ASTM, DSTM, OSTM, RSTM, Haskell STM
Takojšnje	Bartok STM, McRT-STM, AUTOLOCKER
Večverzijsko	Clojure STM [28]

Slika 3.4: Delitev sistemov STM glede na upravljanje z verzijami

Leno upravljanje z verzijami

Pri STM z lenim uporavljanjem z verzijami stara verzija podatkov ostane v pomnilniku na svojem mestu in transakcija hrani svojo novo verzijo podatkov v ločenem medpomnilniku [10, 12, 21, 36, 37]. Ko se transakcija potrdi, nova verzija iz privatnega medpomnilnika zamenja staro verzijo, medpomnilnik se sprosti. Če se transakcija razveljavi, se nova verzija v privatnem medpomnilniku zavrže, stara verzija pa ostane na svojem mestu. LVM je torej pristop, ki pohitri razveljavitve, za ceno dražjih potrditev. Poleg tega LVM omogoča več sočasnih dostopov transakcij do skupnih podatkov, pri čemer vsaka transakcija hrani svojo privatno kopijo objekta. LVM torej dopušča sočasne bralne in pisalne dostope za iste podatke [13].

Takemu pristopu se reče tudi "odloženo posodabljanje"(angl. deferred updating) ali pa "medpomnjenje pisanj"(angl. write buffering) [49].

Takojšnje upravljanje z verzijami

Pri STM s takojšnjim upravljanjem z verzijami nova verzija podatkov (torej spremembe, ki jih povzroča transakcija) takoj nadomesti staro verzijo v skupnem pomnilniškem prostoru transakcij [49, 38, 24]. V posebnem medpomnilniku se shrani stara verzija ali pa podatki, ki omogočajo vzvratno rekonstrukcijo starih podatkov iz spremenjenih. EVM torej prihrani ceno kopiranja objekta ali bloka takoj ob dostopu, saj se vsi pisalni in bralni dostopi vršijo kar neposredno na staro mesto. To seveda pomeni, da se lahko shrani samo ena spremenjena verzija in lahko samo ena transakcija vrši pisalne dostope. Druge transakcije sicer lahko opravljajo bralne dostope, a bodo, če so opravljeni po pisalnih dostopih transakcije-lastnice, zaman, saj bo taka transakcija gotovo razveljavljena pri potrjevanju.

Sočasnost dostopov je bolj omejena. Potrjevanje transakcije pri EVM je preprostejše, saj je spremenjena verzija objekta že na svojem mestu, potrebno je le zavreči razveljavitvene zapise. Glede bralnih dostopov pa je pri potrjevanju potrebno predvsem preveriti, da je verzija prebranega objekta nespre-

menjena, ista v času potrjevanja, kot je bila v času branja. Razveljavljenje transakcije je zahtevnejše kot pri LVM, ker mora transakcija skupne podatke povrniti v prvotno stanje s pomočjo razveljavitvenih zapisov (angl. undo log). EVM je torej primernejši za uporabo kot LVM, kadar lahko pričakujemo več uspešnih potrditev in malo razveljavitev.

Takemu pristopu se reče tudi "neposredno posodabljanje"(angl. direct updating) ali pa "hranjenje razveljavitvenih zapisov"(angl. undo logging) [49].

Večverzijski nadzor sočasnoti

Redkeje se pojavljajo tudi drugačne izvedbe, na primer izvedba pri kateri vsak pisalni dostop proizvede novo kopijo objekta, katera se shrani poleg objekta samega. To seveda predstavlja dodatno obremenitev za sistem, tako časovno (mnogo kopiranj objekta) kot tudi prostorsko. Pridobitev takega pristopa je to, da transakcije, ki samo berejo, nikoli ne morejo biti v konfliktu, saj lahko skozi celotno transakcijo od začetka do konca operirajo z verzijo objekta (ker so vse prejšnje verzije na voljo) kakršen je bil ob začetku dotične transakcije [6].

Takšna izvedba je zelo koristna, če programski jezik že po naravi operira predvsem z nespremenljivimi objekti (angl. immutable objects). Eden takih je programski jezik Clojure, ker je večina struktur nespremenljivih. Poskus dodajanja elementa seznamu bo vrnil kopijo starega seznama z dodanim elementom, pri čemer se uporablja poseben algoritem, ki ima časovno zahtevnost "O(1)". V takem programskem jeziku tak način ravnanja z različnimi verzijami podatkov preprosta razširitev, ki ne povzroča posebnih dodatnih upočasnitev.

3.5.3 Zrnatost dostopov

Različne izvedbe STM hranijo podatke o dostopih z različno ločljivostjo in zrnatost STM govori o enoti podatkov, na podlagi katere STM zaznava konflikte [13]. Gre za ločljivost sledenja dostopov uporabnika STM.

Ena izmed možnosti je ločljivost posamezne besede. Statični STM avtorjev Shavit in Toitou določa, da je hkrati lahko samo ena transakcija lastnica neke pomnilniške besede, in za vsako pomnilniško besedo posebej obstaja zapis o lastništvu. Konflikte se zaznava na podlagi zapisov o lastništvu, torej se ti pojavijo, če druga transakcija dostopa do iste pomnilniške besede.

Drugi avtorji uporabljajo ločljivost bloka, kot na primer Harris v Haskell STM [23, 46, 47]. Haskell STM uporablja večbesedno strukturo TVars, v kateri shranjuje transakcijske spremenljivke in dostop do katerekoli od teh lokacij

sproži konflikt. Harris in Fraser sta predlagala drugačen pristop, ki pa ima v smislu granularnosti podoben efekt. Za sledenje lastništvu se uporablja tabela zapisov lastništva fiksne velikosti (od 4096 do 65536 zapisov). Vsak pomnilniški naslov je preko razpršitvene funkcije preslikan v nek zapis o lastništvu. Več naslovov se lahko preslika v en zapis o lastništvu. V tem primeru do konflika ne privede samo poseg na naslov, ki si ga lasti druga transakcija, ampak tudi posegi na naslove, ki se preko razpršitvene funkcije slikajo v isti zapis o lastništvu. Tak pristop ima torej lahko lažne konflikte.

Tretji avtorji pa uporabljajo ločljivost objekta [27, 36, 37, 12, 18, 24, 9]. V tem primeru ima vsak objekt svoj transakcijski opisnik, ki lahko vsebuje tudi informacijo o lastništvu (v primeru, da gre za sistem, ki uporablja ta pristop). Opisnik je lahko hranjen ločeno od objekta samega ali pa ne. V tem primeru bodo konfliktni vsi posegi različnih transakcij v isti objekt, četudi spreminjajo različne, med sabo neodvisne, dele objekta, kar lahko potencialno pomeni še več lažnih konfliktov, kot pri ločljivosti bloka.

Lažni konflikti niso problematični s stališča pravilnosti izvajanja, ker kvečjemu povzročijo razveljavitev in ponovni zagon transakcije, so pa problem pri hitrosti izvajanja. V primerjav z alternativami, ločljivost posamezne besede omogoča kar največ sočasnosti pri dostopih, saj ne povzroča razveljavitev, če transakcije ne dostopajo do točno istih pomnilniških naslovov. Ta prednost je izrazita, kadar transakcije dostopajo do istega polja, matrike, etc... [13]. Slabost je večja količina meta-podatkov, ki jih mora transakcija vzdrževati in ažurirati, ter večja količina informacije, ki si jo morajo transakcije posredovati med sabo. S povečanjem granularnosti na raven blokov se te slabosti zmanjšajo, poveča pa se število konfliktov. Granularnost na nivoju objektov še potencira ta premik.

Izkaže se, da je granularnost na nivoju objektov povsem primerna za pogoste objektno orientirane programe in dinamične podatkovne strukture (npr. drevesa), ni pa primerna za močno sočasne dostope do polj in sorodnih struktur. Poleg tega je težko ponuditi objektno granularnost v programskih jezikih, ki niso objektni (npr. Haskell) [13].

Leta 2006 so se začeli pojavljati tudi STM z hibridnim pristopom [10, 38, 49], v katerih STM sistem prilagaja granularnost tipu bremena. V primeru, da je več dinamičnih struktur se uporablja granularnost objekta, pri večdimenzionalnih poljih pa granularnost bloka. Tak sistem se najbolje odreže od vseh naštetih.

Razvrstitev nekaterih sistemov STM glede na to lastnost je podana na sliki 3.5.

Zrnatost	STM
Beseda ali blok	Shavit STM [50], WSTM, Haskell STM
Objekt	DSTM, OSTM, Ananian STM, RSTM, SXM, Bartok STM, ASTM
Hibridno	TL, AUTOLOCKER, McRT-STM

Slika 3.5: Delitev sistemov STM glede na zrnatost

3.5.4 Medpomnilnik za sledenje bralnim dostopom transakcij

Ta medpomnilnik je privaten posamezni transakciji. Ponavadi vsebuje zapise v obliki parov:

1. Kazalec na opisnik bloka ali objekta

2. Verzija prebranih podatkov

Ta par dokumentira kateri podatki so bili prebrani in v kakšni verziji, s dodano povezavo na transakcijski opisnik objekta ali bloka. Pri operaciji preverjanja transakcije ta informacija služi kot evidenca objektov ali blokov, katerih verzije je potrebno preveriti, da ostajajo nespremenjene, pred potrditvijo transakcije. Če uporabljamo večverzijski nadzor sočasnosti je ta medpomnilnik nepotreben. Pri dostopih se vedno prebere najnovejšo verzijo objekta s časovno oznako starejšo od začetka transakcije. Ker so pri večverzijskem nadzoru nadzoru sočasnosti bralni konflikti nemogoči, ni potrebno hraniti zapisov zgodovine bralnih dostopov.

3.5.5 Medpomnilnik za sledenje pisalnim dostopom transakcij

Tudi ta medpomnilnik je privaten posamezni transakciji. Najpogosteje vsebuje seznam vseh transakcijskih opisnikov pri katerih ima transakcija registrirano pravico pisalnega dostopa. V primeru, da transakcijski opisnik objekta ne hrani informacije o spremembah samih vrednosti podatkov so lahko taki podatki shranjeni v tem medpomnilniku.

3.5.6 Bralne in pisalne zapore

Zapore so kratki odseki kode, ki se med prevajanjem programov, ki uporabljajo STM, vstavljajo okoli branj in pisanj. Redkeje se zapore vstavlja neposredno v izvorno kodo. Njihov namen je reguliranje dostopov, tako da spoštujejo pravila izvedbe STM, s katerimi se zagotavlja atomarnost in izoliranost transakcij. Če ima STM upravljanje z verzijami EVM se zapore uporabljajo pri vseh pomnilniških dostopih, pri čemer zapora preveri ali ima kdo drug lastništvo nad lokacijo, ki jo želi transakcija dostopati in po potrebi sporoči konflikt.

V primeru, da je upravljanje z verzijami LVM nam zapore ob bralnih in pisalnih dostopih služijo predvsem za zbiranje metapodatkov transakcije.

3.6 Zazanavanje konfliktov

Kadar več transakcij dostopa do istih podatkov imamo opraviti s konfliktom. Pomen izraza "isti podatki"je vezan na granularnost dostopov, le-ta definira katera enota pomnilnika je uporabljana kot ena entiteta za namene zaznavanja konfliktov.

Niso vse kombinacije dostopov konfliktne. Če več transakcij bere iste podatke, potem konflikta nimamo. Če ena transakcija piše, potem so vse transakcije, ki so do istih podatkov opravile bralni dostop, z njo v konfliktu, razen če sistem uporablja MVCC za upravljanje z verzijami (potem konflikta ni). Če več transakcij opravi pisalni dostop do istih podatkov, potem konflikt vedno obstaja. Zagotovo so v medsebojnem konfliktu vse transakcije, ki pišejo, poleg njih so v konfliktu tudi transakcije ki berejo, razen če STM uporablja MVCC.

Ko je konflikt zaznan so podatki o vpletenih dveh transakcijah podani modulu za razsojanje sporov.

Metode zaznavanja konfliktov se razlikujejo po tem kdaj se za konflikte preverja. V osnovi ločimo tri različne načine.

Razvrstitev nekaterih sistemov STM glede na to lastnost je podana na sliki 3.6.

Zrnatost	STM
Leno	TL, SXM, OSTM, DSTM, WSTM, ASTM, RSTM, Haskell STM
Zgodnje	HICKS [41], AUTOLOCKER, Shavit STM
Hibridno	Bartok STM, McRT-STM

Slika 3.6: Delitev sistemov STM glede na tehniko zaznavanja konfliktov

3.6.1 Zgodnje zaznavanje konfliktov

STM z zgodnjim zaznavanjem konfliktov (angl. eager conflict detection — ECD) poskušajo uloviti konflike ob dostopih samih [38, 50, 41]. Ob dostopu se preveri lastništvo dostopanih podatkov v transakcijskem opisniku bloka ali objekta. V primeru, da je prosto, se zaseže lastništvo drugače pa je prišlo do konflikta se takoj kliče modul za razreševanje konfliktov.

Pri STM, ki uporabljajo zaklepanje se konflikt kaže tako, da transakcija naleti na ključavnico, ki si jo lasti nekdo drug. Pri takih sistemih z zaklepanjem vodenje evidence (read-set in write-set) ni vedno potrebno.

Z vodenjem verzij EVM se vedno uporablja ECD ali HCD, ker je potrebno konflikte zaznati takoj. HTM sistemi ponujajo kombinacijo LVM in ECD, katera se pri STM ne pojavlja.

3.6.2 Leno zaznavanje konfliktov

STM z lenim zaznavanjem konfliktov (angl. lazy conflict detection — LCD) si ob dostopih beleži podatke v medpomnilnika za bralne in pisalne dostope. Konflikte se zaznava ob preverjanju transakcije. Takrat se pregleda vso zgodovino dostopov, ki je shranjena v medpomnilnikih in se preda zaznane konflikte modulu za razreševanje konfliktov.

Pri STM, ki uporabljajo zaklepanje, se pri LCD pred potrditvijo transakcije poskuša pridobiti lastništvo vseh ključavnic nad vsemi dostopanimi podatki, kar povzroči, da se vsi konflikti manifestirajo v obliki nepridobljenih ključavnic.

LCD se lahko uporablja z samo vodenjem verzij LVM, ker je za EVM zaznavanje konfliktov tik pred potrditvijo prepozno.

3.6.3 Hibridno zaznavanje konfliktov

Hibridno zaznavanje konfliktov (angl. hybrid conflict detection — HCD) je kombinacija LCD in ECD. Uporablja se v nekaterih novejših STM z zaklepanjem [49, 24]. Ti sistemi uporabljajo upravljanje z verzijami EVM. Pri pisalnih dostopih se uporablja princip ECD in se konflikte takoj zazna. Pri bralnih dostopih pa se samo zabeleži verzijo prebranih podatkov in se konflikte zaznava po principu LCD ob času potrjevanja transakcije. Tak pristop omogoča večjo sočasnost bralnih dostopov.

3.7 Razreševanje konfliktov

Pri večini STM je strategija razreševanja konfliktov zelo preprosta. Najpogostejša izvedba je, da transakcija, ki naleti na drugo transakcijo, ki si lasti objekt ali blok, drugo transakcijo razveljavi. Če STM uporablja zaklepanje lahko transakcija tudi čaka drugo (v tem primeru mora sistem tudi zaznavati smrtne objeme transakcij). Redkeje se konflikt tako rešuje, da se razveljavi transakcijo, ki je konflikt zaznala.

Nekateri STM omogočajo, da se pri razreševanju konfliktov uporabljajo upravitelja sporov (angl. contention manager). Ta opcijska komponenta omogoča bolj zapletene mehanizme razreševanja konfliktov.

3.8 Preverjanje transakcije

Preverjanje transakcije (angl. validating transaction) je opcijski korak pred potrditvijo transakcije in se zgodi ob klicu funkcije za potrditev transakcije s strani uporabnika. Pojavlja se pri sistemih, ki uporabljajo zaznavanje konfliktov LCD. V tem koraku se preverja, če je prišlo med transakcijo do konfliktov.

3.9 Potrjevanje transakcije

Ko uporabnik izda ukaz za potrditev transakcije (angl. commit transaction), se najprej kliče funkcijo za preverjanje transakcije, če STM uporablja zaznavanje konfliktov LCD ali HCD. Ta ob zaključku kliče funkcijo za potrditev transakcije. V primeru da se uporablja ECD je korak preverjanja transakcije pred potrditvijo nepotreben. V primeru, da STM uporablja upravljanje z verzijami EVM in zaznavanje konfliktov ECD je potrditev transakcije trivialna, saj so vsi zapisani podatki že na svojem mestu in vsi bralni dostopi so preverjeno nekonfliktni. Pri uporabi HCD je potrebno preveriti nekonfliktnost vseh bralnih dostopov. V primeru, da se uporablja LVM in LCD, je potrebno ob potrjevanju pridobiti lastništvo vseh transakcijskih opisnikov dostopanih objektov ali blokov (ali v primeru izvedbe STM z zaklepanjem, lastništvo vseh ključavnic) in potem preprisati privatne kopije blokov ali objektov na mesto originala v skupnem pomnilniku.

Zadnji korak, ki je ostaja, je, da se spremeni status transakcije iz aktivne v potrjeno.

3.10 Razveljavitev transakcije

Funkcijo za razveljavitev transakcije (angl. abort transaction, rollback transaction) praviloma kliče funkcija razreševanja konfliktov (te ali druge transakcije) ali pa upravitelj sporov. Nekateri STM omogočajo klic te funkcije tudi s strani uporabnika. Če se uporablja upravljanje z verzijami LVM je ta korak trivialen. Preprosto se zavrže privatne kopije dostopanih objektov ali blokov, ki jih transakcija hrani v zasebnem pomnilniškem prostoru. Dostopane objekte ali bloke imamo evidentirane v medpomnilnikih za pisalne in bralne dostope. V primeru, da se uporablja upravljanje z verzijami EVM pa je potrebno iz razveljavitvenih zapisov (angl. undo logs), ki se hranijo v transakcijskih opisnikih, rekonstruirati originalne podatke (pogosto so razveljavitveni opisi kar dobesedna kopija originalnega objekta ali bloka) ter objekt ali blok z njimi povrniti v stanje enako tistemu ob začetku transakcije. Pred tem je seveda potrebno pridobiti lastništvo nad objektom ali blokom (ali ključavnico).

Po razveljavitvi se transakcija avtomatsko ponovno zažene.

Poglavje 4

Opcijski elementi programskega transakcijskega pomnilnika

Veliko STM sistemov vključuje tudi elemente ali koncepte, ki za osnovno delovanje niso nujni, so pa koristni bodisi zaradi hitrejšega delovanja STM bodisi zaradi lažjega pisanja programov, ki uporabljajo paralelizem.

4.1 Brisanje bralnega dostopa iz medpomnilnika

Nekateri STM dovoljujejo uporabniku, da ukaže, naj se evidenca o določenem bralnem dostopu briše iz medpomnilnika za sledenje bralnim dostopom [27]. Ponavadi se pojavlja kot ukaz "release". Ko je bila evidenca zbrisana, potem dostopi drugih transakcij do istih podatkov niso več v konfliktu s to transakcijo. Tukaj je izredno pomembno poudariti, da ta ukaz omogoča programerju, da poruši garancijo pravilnosti, ki je sicer eno izmed centralnih vodil načrtovanja STM sistemov. Ena izmed prednosti STM pred zaklepanjem je ravno ta garancija pravilnosti, ki vzame odgovornost za pravilnost sinhronizacijskega mehanizma (in posledično izvajanja programa) iz rok programerja.

Ta ukaz omogoča programerju dodatno kontrolo nad delovanjem STM in s tem dodatno optimizacijo. Če na primer pišemo funkcijo, ki v seznamu poišče element, nam ta ukaz pride prav. Pri iskanju elementa v seznamu obiščemo vse elemente seznama zaporedoma, dokler ne najdemo pravega. Taka transakcija ima v medpomnilniku za sledenje bralnim dostopom kazalce na vse te elemente seznama. Pisalni dostop katerekoli druge transakcije do poljubnega od teh elementov seznama pred iskanim elementom bo povzročil konflikt in razveljavitev

transakcije. Taka razveljavitev je na nek način nepotrebna, saj spremembe vrednosti v elementih seznama pred iskanim elementom prav nič ne ogrozijo pravilnosti izvajanja funkcije in tudi, če funkcija vidi polovične spremembe neke druge transakcije, pravilnost v tem primeru ni ogrožena. Tedaj lahko v tej funkciji uporabimo ta ukaz za brisanje bralnega dostopa iz medpomnilnika, da zbrišemo dostope do elementov seznama pred iskanim elementom in se tako izognemo veliko konfliktom in razveljavitvam transakcije.

Kot že omenjeno, ta ukaz prelaga odgovornost v roke programerja, ki ga mora zelo previdno uporabljati. Zato se pojavlja zelo redko [27].

4.2 Pogojno čakanje

Pri STM, ki omogočajo CCR, se pojavlja še ena operacija, ki premošča razliko med STM in CCR. CCR imajo pred atomarnim blokom vstopni pogoj, pri katerem nit čaka dokler ni izpolnjen. Pogosto se v literaturi imenuje "STMWait"ali preprosto "Wait"[21].

Ta ukaz se izvede po tem, ko se prvič preveri vstopni pogoj na začetku, če leta ni izpolnjen. Ko se izvede, transakcija pridobi lastništvo nad vsemi podatki, ki so evidentirani v medpomnilnikih za sledenje bralnim in pisalnim dostopom in spremeni status v "spi"(angl. asleep). Edini do sedaj opravljeni dostopi transakcije so tisti, ki jih je transakcija opravila zato, da je preverila vstopni pogoj, kar so hkrati tudi edini podatki, katerih sprememba lahko potencialno povzroči, da postane pogoj resničen.

Ko druge transakcije poskusijo spremeniti katerega od teh podatkov, naletijo na tuje lastništvo s strani speče transakcije. Potem prevzamejo lastništvo, spremenijo podatek in spečo transakcijo zbudijo. Lastništvo lokacij podatkov, ki so bili uporabljeni pri računanju pogoja, torej služi kot neke vrste sprožilec, da transakcija zazna, kdaj se ji splača ponovno preveriti pogoj. Ravno izguba hitrosti zaradi prepogostih periodičnih preverjanj pogojev je bil problem zgodnejših izved CCR.

Ko je transakcija zbujena, se razveljavi in ponovno požene, kar pomeni ponovno preverjanje pogoja. Razveljavitev je potrebna zaradi možnosti, da izračun pogoja vsebuje tudi pisalne dostope (torej spremembe podatkov).

Ta postopek minimizira število preverjanj pogoja.

4.3 Razsodniki sporov

Razsodniki sporov so moduli, ki se uporabljajo za odločanje katera od dveh transakcij v konfliktu naj razveljavi drugo. V sistemih brez oviranja ali blokirajočih sistemih, ki imajo zaradi šibkih zagotovil napredovanja sistema tveganje živega objema in izstradanja niti, se lahko uporabljajo preproste strategije razreševanja konfliktov, ki naj bi rešili te probleme. Popularna opcija je eksponentno umikanje. Toda z naraščanjem obremenitve sistema so pogosto potrebni bolj zapleteni pristopi. Poleg tega je včasih zaželjeno upoštevanje prioritet niti in transakcij. Razsodnik sporov omogoča razreševanje sporov na podlagi raznih statistik in lastnosti transakcij [31].

Lahko obstaja globalni razsodnik sporov ali pa ima vsaka nit svojega in potem razsodnika sporov obeh niti medsebojno primerjata prioritete in druge statistike. Razsodniki sporov so zelo modularni in ni težko omogočiti uporabniku, da piše lastne razsodnike sporov. Prav tako ni težko omogočiti menjavo razsodnikov sporov med izvajanjem samim.

Kriterij za pravilnost razsodnika sporov je relativno šibek. Neformalno, katerakoli aktivna transakcija, ki zaprosi dovoljkrat, mora sčasoma dobiti dovoljenje, da razveljavi nasprotujočo transakcijo. Vsak klic razsodnika sporov se mora izvesti v končnem času [27]. Te lastnosti so potrebne, da se ohranja lastnost, da je sistem brez oviranja. Transakcija, kateri je vedno zavrnjena prošnja, da bi razveljavila nasprotujočo transakcijo, se nikoli ne bo uspešno potrdila, tudi če se izvaja izolirano.

Te lastnosti razsodnikov sporov še vedno ne garantirajo napredovanja sistema ob pretirano visoki obremenitvi.

Ko transakcija naleti na tuje lastništvo in kliče razsodnika sporov, govorimo o njej kot o “prijateljski”transakciji. Transakcijo, ki poseduje lastništvo nad želenimi podatki, pa imenujemo nasprotujoča transakcija. Sledi pregled nekaterih bolj uspešnih predlaganih razsodnikov sporov.

4.3.1 Karma

Razsodnik sporov “Karma”poskuša oceniti količino dela, ki ga je transakcija do sedaj opravila, ko se odloča, če naj transakcijo razveljavi. Ocena temelji na številu dostopanih objektov glede na posamezno transakcijo, kar naj bi približno vrednotilo investicijo v posamezno transakcijo [32]. Karma uporablja število dostopanih objektov kot prioriteto transakcije. Prioriteta transakcije se poveča vsakič, ko transakcija dostopa do novega objekta.

Tak razsodnik sporov se utemeljuje z idejo, da je boljše za prepustnost

sistema, da se razveljavlja transakcije, ki so se šele začele, namesto tistih, ki so zadnjih korakih pred potrditvijo. Če je transakcija razveljavljena, ima pri naslednjem poskusu izvedbe že na začetku prioriteto, kakršno je imela pri prejšnjem poskusu pri koncu (prioriteta se prenese). Ta akumulacija prioritete omogoča kratkim transakcijam, da si sčasoma, skozi več poskusov, zgradijo prioriteto, kar prepreči, da bi prišlo do izstradanja.

Razsodnik sporov Karma razveljavi nasprotujočo transakcijo, ko število poskusov prijateljske transakcije, da bi dostopala do nekih podatkov, preseže razliko med prioritetama prijateljske in nasprotujoče transakcije. V nasprotnem primeru razsodnik zavrne dovoljenje, da bi prijateljska transakcija razveljavila nasprotujočo ter zahteva, da prijateljska transakcija počaka fiksno količino časa preden ponovno poskusi.

4.3.2 Časovne oznake

Razsodnik sporov "Časovne oznake"(angl. Timestamp) poskuša biti čimbolj pošten do transakcij. Ob začetku vsake transakcije si zapiše čas. Razsodnik dovoli razveljavitev nasprotujoče transakcije, če je ta mlajša. V nasprotnem primeru prijateljska transakcija čaka zaporedje intervalov. Po vsakem intervalu označi nasprotujočo transakcijo kot okvarjeno. Po zaporedju teh intervalov, če je nasprotujoča transakcija še vedno tako označena, razsodnik sporov dovoli prijateljski transakciji, da jo razveljavi. Aktivne transakcije brišejo to oznako. Tak pristop zagotavlja razveljavitev transakcij v nitih, ki jih je razvrščevalnik zapostavil in ostalih transakcij, ki ne delujejo pravilno [32].

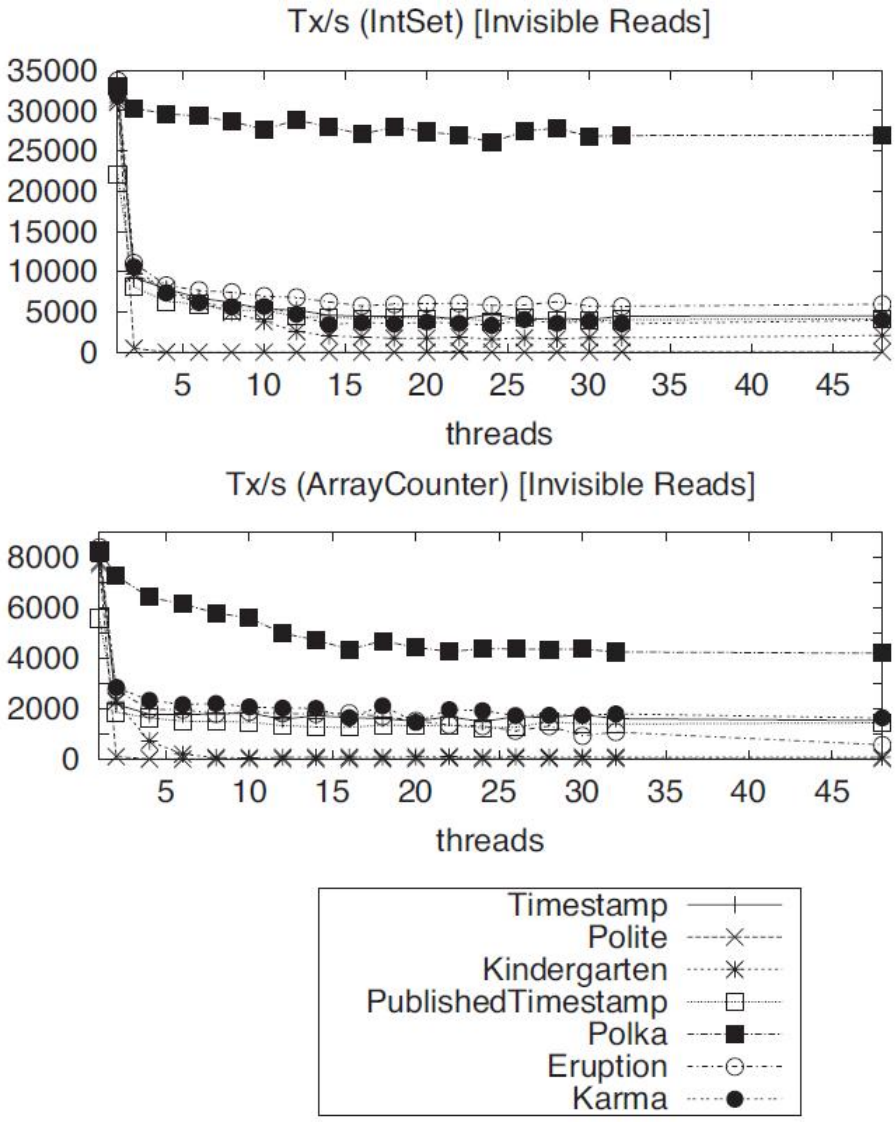
4.3.3 Polka

Razsodnik sporov "Polka"(ime izvira iz besedne kombinacije "polite"in "karma") je podoben upravitelju "Karma" z nekaterimi izboljšavami [32]. Kadar bi razsodnik sporov "Karma"ukazal transakciji, naj čaka fiksno količino časa, "Polka"ukaže transakciji, da naj počaka časovno obdobje, ki je sorazmerno razliki med prioritetama prijateljske in nasprotujoče transakcije. Poleg tega v se pri "Polka"obdobje čakanja pri večih zaporednih čakanjih eksponentno povečuje.

Ker raziskave kažejo, da so pisalni dostopi bolj pomembni za prepustnost sistema, kot bralni, razsodnik sporov "Polka"vedno dovoli razveljavitev transakcije, ki samo bere, če si lasti objekt, ki ga potrebuje transakcija, ki tudi piše.

4.3.4 Primerjava

V praktično vseh testih razsodnik sporov "Polka"dosega daleč najboljše rezultate. Primeri testov so na sliki 4.1.



Slika 4.1: Testi razsodnikov sporov [32]

4.4 Gnezdenje

Zelo pomembno vprašanje je, kako je urejeno gnezdenje transakcij. Koda uporabnika, ki se izvaja znotraj transakcije, lahko naprimer kliče kodo neke knjižnjice, ki tudi sama uporablja STM, česar uporabnik knižnjice niti ne ve. Zato morajo vsi STM imeti strategijo glede gnezdenja. Gnezdenje je pomembno, ker omogoča programerju, da se izogne dragim razveljavitvam dolgih transakcij, če so potencialni konflikti omejeni na majhen del velike transakcije. Klasičen primer potencialne uporabe gnezdenja bi bila izredno dolga transakcija, na koncu katere se dobljen rezulat vstavi v seznam, ki je zelo obiskovan s strani številnih drugih transakcij. Taka transakcija bo najverjetneje razveljaljena ravno v tem zadnjem dejanju, kar bo povzročilo veliko izgubo opravljenega dela. Če pa to vstavljanje rezultatov v seznam zavijemo v gnezdeno transakcijo, nam konflikt povzroči razveljavitev samo te notranje transakcije vse ostalo opravljeno delo pa ostane. Gnezdenje torej prinese precejšnje prihranke ob pravilni rabi.

4.4.1 Spločitev gnezdenja

Najpreprostejša strategija gnezdenja (in hkrati razlog zakaj je gnezdenje v poglavju o opcijskih elementih STM) je, da gnezdenje eliminiramo s sploščitvijo (angl. flattening). To je izredno preprosta rešitev. Ko naletimo na ukaz uporabnika za začetek transakcije znotraj aktivne transakcije samo povečamo števec globine transakcije. Ko naletimo na ukaz za potrditev transakcije, pa samo zmanjšamo števec globine transakcije, razen če je globina enaka nič. Tedaj dejansko sprožimo funkcijo za potrditev transakcije.

S tem pristopomo efektivno eliminiramo vse klice za začetek transakcije za prvim in vse klice za potrditev transakcije pred zadnjim. Notranja transakcija preprosto postane del večje transakcije. Prednost tega pristopa je v njegovi preprostosti. Ni zapleten za izvedbo, poleg tega se izognemu alociranju pomnilnika in inicializaciji struktur transakcije, ki bi jih potrebovali, če bi imeli gnezdeno transakcijo. Slabost je seveda izguba zgoraj omenjenih prednosti gnezdenja.

4.4.2 Zaprto gnezdenje

Zaprto gnezdenje je glavna rešitev problema gnezdenja in nima alternativ. Zaprto gnezdene transakcije so po delovanju zelo podobne navadnim, za uporabnika razlike celo ni, kar omogoča, da uporabnik piše vse transakcije enako

in mu ni potrebno vedeti katere bodo v končanem programu izvedene gnezdeno. Glavne razlika je v ravnanju s transakcijskimi rezultati in metapodatki. Namesto, da bi se ob potrditvi transakcije spremembe podatkov (pisalni dostopi) preslikale v skupni pomnilnik kot pri negnezdeni transakciji, se pri gnezdenih transakcijah spremembe podatkov ob potrditvi samo predajo starševski transakciji. Medpomnilnika, ki hranita podatke o bralnih in pisalnih dostopih se združita z istimi medpomnilniki starševske transakcije z operacijo unije, namesto da bi se zavrgla (kot se zgodi pri negnezdenih transakcijah) [7].

Rezultati zaprto gnezdene transakcije so po končanju vidni samo starševski transakciji. Starševska transakcija poskrbi, da se ob njeni potrditvi zapišejo v skupni pomnilnik tudi rezultati gnezdene transakcije. Starševska transakcija ima tudi vse potrebne podatke, da zaznava konflikte, ki bi izvirali iz dostopov gnezdene transakcije.

4.4.3 Odprto gnezdenje

Zaprto gnezdenje nam sicer omogoča, kadar pride do konfliktov pred potrditvijo gnezdene transakcije, da ne razveljavimo vsega dela starševske transakcije. Problem pa je, da s tem, ko gnezdena transakcija ob končanju preda opravljene dostope starševski transakciji, s tem nje ne zaščiti pred konflikti povezanimi s temi podatki.

Primer take pomanjkljivosti: gnezdena transakcija bere lokacijo A in uspešno zaključi. Sedaj je lokacija A dodana v medpomnilnik za sledenje bralnim dostopom starševske tranaskcije. Neka druga transakcija piše na lokacijo A in se potrdi. Ko uporabnik poskusi potrditi starševsko transakcijo dobimo konflikt. Drugi problem je, če bi gnezdena transakcija si lastila pravico pisalnega dostopa do lokacije A, ter pisala na to lokacijo in se potrdila. Starševska transakcija bi prevzela pravico pisalnega dostopa do lokacije A in, čeprav ne bi nikoli dostopala do lokacije A, nobena druga transakcija ne bi mogla nekonfliktno dostopati do lokacije A.

To je glavna pomanjkljivost zaprtega gnezdenja, da dostopi gnezdene transakcije tudi po njenem končanju še vedno motijo druge transakcije, vse dokler se tudi starševska ne potrdi. Ta problem nima nobene polne rešitve. Obstaja pa delna rešitev, ki se ji reče odprto gnezdenje [7]. Pri odprtem gnezdenju se gnezdena transakcija pri potrjevanju obnaša kot negnezdena. To pomeni, da se po potrditvi medpomnilniki s podatki o dostopih zavržejo, rezultati pisalnih dostopov pa se preslikajo neosredno v skupni pomnilnik. Posledice odprto-gnezdene transakcije so po potrditvi takoj globalno vidne, torej še pred potrditvijo starševske transakcije.

Primer rabe je program 6. Razvidno je, da je "generateID"funkcija, ki jo bo klicalo veliko različnih niti in transakcij. Če bi uporabili zaprto gnezdenje, bi imeli veliko razveljavljenih transakcij v funkciji "createOrder", ker bi vse sočasne izvedbe te funkcije v svoji zgodovini nosile dostop do "id". Z odprtim gnezdenjem so vsi konflikti omejeni na zelo kratko transakcijo, starševska transakcija pa zaradi vsebine gnezdene nima konfliktov.

Program 6 Uporaba odprtega gnezdenja

```
public static open long generateID {
    open {
        return id++;
    }
}

public static void createOrder(...)
    atomic {
        order.setId(generateID());
        .....
    }
```

Pomembno je poudariti, da odprto gnezdenje ni alternativa zaprtemu gnezdenju in zato noben STM ne ponuja samo odprtega gnezdenja. Ali se ponuja samo zaprto ali pa oboje. Odprto gnezdenje ima pri uporabi omejitve, na katere mora uporabnik paziti, če želi previlno delujoč program. V tem je odprto gnezdenje podobno kot funkcija za odstranjevanje pisalnih dostopov iz evidence. Prva omejitev odprtega gnezdenja je, da morata biti množici pisalnih dostopov gnezdene in starševske transakcije disjunktni. V programu 6 je ta pogoj izpoljnjen. Druga omejitev je vezana na razveljavitev starševske transakcije. Ko se starševska transakcija razveljavi, ostanejo posledice odprto-gnezdene transakcije nedotaknjene zato se lahko uporabljajo samo za primere, kjer to ni problematično. V primeru programa 6 bo edina posledica razveljavitev starševske transakcije to, da številke naročil ne bodo povsem zaporedne (nekatere bodo neuporabljene) [7]. Nekateri predlagajo, da bi dovolili uporabniku, da napiše kodo, ki se bo izvršila ob razveljavitvi določene transakcije. Potem bi lahko s takim blokom kode pri starševski transakciji ročno razveljavljali posledice odprto-gnezdene transakcije. Ta rešitev ima svoje probleme, namreč tudi ta koda bi potrebovala sinhronizacijo.

Poglavje 5

Problemi programskega transakcijskega pomnilnika ter rešitve

5.1 Vhodno-izhodne operacije

Ukazi uporabnika STM, ki se končajo v interakciji z vhodno-izhodnimi napravami, porušijo zagotovilo, da razveljavljena transakcija nima posledic oziroma da je za preostanek sistema nevidna. Mehanizem razveljavitve, ki povrne pomnilnik v stanje pred transakcijo, preprosto ne more narediti nič glede že poslanih paketkov preko mreže, zapisanih podatkov na disk etc... Noben STM sistem ne more zagotoviti atomarnosti funkciji, ki zamenja imeni dveh datotek z zaporedjem treh ukazov preimenovanja datoteke. Morda bi lahko zagotovili, da se operacija vedno izvede ali v celoti ali pa nič, nikakor pa ne bi zmogli preprečiti ostalim procesom, da ne bi videli datotek v stanju napol izvedene operacije zamenjave imen. Pri programskem jeziku kot je Java v to kategorijo ukazov spadajo tudi vsi ukazi, ki zahtevajo od JVM, da kliče operacijski sistem (naprimer sprememba sistemskega časa). To so vsi klici označeni z besedo "native"[22].

Veliko STM rešuje to tako, da takih ukazov v transakcijah preprosto ne dovoljuje. Edina druga rešitev je medpomnenje (angl. buffering) vhodno-izhodnih operacij. Pri takem medpomnenju se rezultati vseh vhodnih operacij zapišejo v medpomnilnik in se v primeru razveljavitve transakcije potisnejo nazaj na vhodni tok podatkov. Rezultati izhodnih operacij se prav tako zapisujejo in se ob potrditvi transakcije naenkrat pošljejo v naprave. V primeru razveljavitve se preprosto zavržejo. Problematika take rešitve izvira iz izredne

različnosti vhodno-izhodnih naprav, ki jih uporabnik lahko uporablja. Včasih to medpomnenje preprosto ni mogoče, ali pa je željeno drugačno obnašanje.

Nekateri predlagajo, da bi vhodno-izhodne knjižnjice se registrirale pri STM in bi dobivale obvestila o razveljavitvah in potrditvah [22]. Potem bi lahko na te dogodke reagirale na način primeren napravi. To bi seveda pomenilo, da bi bilo potrebno že obstoječe knjižnice spremeniti.

5.2 Napake in izjeme

Večina sodobnih programskih jezikov sporoča napake v delovanju programa v obliki "izjem"(angl. exceptions). Zagata je v vprašanju kako ravnati ob izjemah.

Prva rešitev je, da se izjema v transakciji obnaša enako, kot se v navadni kodi. Izvajanje transakcije se prekine, njeni delni rezultati ostanejo. Vendar obstaja množica situacij, pri katerih bi bilo bolje, če bi se transakcija razveljavila. Naprimer, da napišemo funkcijo za premik objekta iz ene zbirke v drugo in nam lahko druga zbirka vrže izjemo, da objekta ne more sprejeti. V primeru te prve rešitve potrebujemo dodatno kodo, ki se izvede ob izjemi, ki bo "pospravila"objekt nazaj v prvo zbirko (kar lahko sproži dodatne izjeme). Tukaj je preprosteje, če bi uporabili drugo rešitev in transakcijo preprosto razveljavili [22].

Pri tej drugi rešitvi pa imamo drugačne težave. Nekateri objekti vržejo izjemo, ko dosežejo določeno stanje (npr. poln medpomnilnik), in razveljavitev transakcije bi odstranila posledice ukazov, ki so do takega stanja privedli. Vržena izjema bi torej javljala neobstoječe stanje. Zato je zaželeno, da obstaja v STM način, da uporabnik izbere način spopadanja iz izjemami. Drugo vprašanje kako izjemo spraviti iz transakcije pred razveljavitvijo. Izjema je namreč ustvarjena v transakciji prav tako so vsi podatki, ki jih nosi, bili dodani znotraj transakcije. Razveljavitev bi torej odstranila izjemo, zato se serializira izjemo, razveljavi transakcijo, deserializira izjemo in se jo ponovno vrže [22].

5.3 Šibka atomarnost

V primeru transakcijskih pomnilnikov imamo dve različni meri izolacije transakcij. Močna ter šibka atomarnost. Pri močni atomarnosti delni rezultati, ki nastajajo tekom transakcije, izven nje niso vidni in tuji dostopi do dostopanih podatkov transakcije sprožijo konflikte. Pri šibki atomarnosti to velja samo za transakcije med seboj. Koda, ki se izvaja izven transakcij, v STM s šibko

atomarnostjo ne sproži nobenih konfliktov, če dostopa do skupnih podatkov. Prav tako lahko koda izven transakcij vidi delne rezultate transakcij [7]. Za podrobno obravnavo anomalij, ki jih prinese šibka atomarnost, se obrnite na [51].

Strojni transakcijski pomnilniki večinoma ponujajo močno atomarnost, programski pa na žalost v veliki večini šibko. Razlog, zakaj večina programskih transakcijskih pomnilnikov ne ponuja močne atomarnosti, je zmogljivostne narave. Neoptimizirana močna atomarnost upočasni STM za faktor od 8 do 20-krat. Ta upočasnitev izhaja iz razlike med STM s šibko in močno atomarnostjo: sistem z močno atomarnostjo mora vse pomnilniške dostope v programu zaviti v bralne in pisalne zapore, ne samo tiste, ki so v transakcijah. To prinese precejšnjo upočasnitev v delovanju programske opreme tudi zunaj transakcije.

Poglavje 6

Implementacije programskega transakcijskega pomnilnika

V poglavju o osnovni strukturi in delovanju STM so podani zelo splošni opisi različnih delov STM sistemov, kar otežuje razumevanje kako en cel STM sistem izgleda v praksi. Opisi so zelo splošni, ker so STM sistemi po zasnovi komponent izredno različni in zato opis, ki bi zajemal večino sistemov, ne more biti posebno specifičen.

Zato bodo v tem poglavju predstavljene nekatere dejanske izvedbe.

6.1 DSTM, Herlihy et al

DSTM je prvi dinamični STM in hkrati tudi eden izmed prvih STM [27]. Zaradi tega ima prenekatere pomanjkljivosti, hkrati pa je relativno preprost in pokaže eno izmed osnovnih izvedb STM. Po lastnostih spada med sisteme brez oviranja. Raba STM funkcionalnosti je izredno eksplicitna v kodi, kot je razvidno iz primera programa 7.

Uporabnik izdaja ukaze preko objekta razreda "TMThread", ki je podrazred Java razreda "Thread", ki predstavlja nit. Objekti, ki jih uporabnik želi uporabljati v transakciji so zaviti v objekt razreda "TMObject"in morajo implementirati vmesnik "TMCloneable" s funkcijo "clone()", ki vrne popolno kopijo objekta. Slednje se uporablja za stvaritev kopije objekta za potrebe transakcijskega pomnilnika.

Pred prvim dostopom je potrebno eksplicitno odpreti objekt, kar omogoča STM da pripravi podatkovne strukture za metapodatke. Poleg tega je potrebno tudi določiti vrsto dostopa "WRITE"ali "READ". Več transakcij lahko odpre objekt z bralnim dostopom. Bralni dostop do objekta lahko transakcija poskusi

kasneje nadgraditi na pisalni. Če dostop do objekta ni odobren vrže funkcija "open"izjemo "Denied". V tem primeru je transakcija obsojena na propad. Funkcija "open"uporabi "clone"vmesnika "TMCloneable", da ustvari kopijo in jo vrne uporabniku. Tej kopiji se reče "transakcijina verzija".

Funkcija "commitTransaction"vrne "true", če je transakcija uspešno potrjena. Kot je razvidno iz programa 7, mora uporabnik ročno poskrbeti, da se v primeru razveljavitve transakcije ponovno poskuša.

Takšna izvedba je zelo drugačna od sodobnejših STM, kjer so vse te operacije implicitne in jih STM vstavi med prevajanjem vstavi v izvorno kodo, poleg tega pa ni potrebno programerju ročno pisati kode za kopiranje objekta. Iz takšne izvedbe je preprosto ugotoviti, da so zaščiteni dostopi samo tisti, ki gredo preko teh posebnih objektov STM, in da imamo zato opravka s šibko atomarnostjo.

Program 7 Uporaba DSTM

```
TMThread thread = (TMThread)Thread.currentThread();
Counter counter = new Counter(0);
TMObject tmObject = new TMObject(counter);

while(true) {
    thread.startTransaction();
try {
        counter = (Counter)tmObject.open(WRITE);
        counter.inc();
    } catch(Denied d) {}
    if (thread.commitTransaction()) break;
}
```

Za doseganje atomarnosti brez zaklepanja si DSTM pomaga z dvojno indirekcijo. Za dostop do podatkov moramo iti preko dveh dodatnih referenc, kar ima negativen vpliv na predpomnilnik. "TMObject"vsebuje referenco na transakcijski deskriptor objekta, tako imenovani "lokator"(angl. locator), ki vsebuje kazalec na transakcijo, ki je objekt odprla za pisalne dostope (torej lastnico), poleg tega pa še kazalce na star (originalen) objekt in nov objekt (transakcijina kopija).

Pri branju ali pisanju je postopek dostopa preprost. Če je transakcija navedena v lokatorju kot lastnica, potem se preko reference v lokatorju dostopa do nove verzije objekta. V nasprotnem primeru se pregleda statusno besedo

transakcije, ki je navedena kot lastnica. Če je status, da je transakcija aktivna ali razveljavljena, potem sledimo referenci do stare verzije, v primeru, da je transakcija potrjena, pa sledimo drugi referenci do nove verzije. Na ta način vedno dostopamo do pravilne verzije željenih podatkov.

Medpomnilnik za sledenje pisalnim dostopom je preprosto seznam odprtih objektov v pisalnem načinu. Medpomnilnik za sledenje bralnim dostopom je seznam parov (objekt in verzija). DSTM uporablja LVM. Spremembe se pišejo v kopijo objekta in šele ob potrditvi le-ta postane viden drugim transakcijam. Za zaznavanje konfliktov se uporablja LCD. Konflikti se zaznavajo v posebni fazi preverjanja transakcije pred potrditvijo. Celotna transakcija se preverja tudi ob odprtju vsakega objekta.

Ob preverjanju transakcije je potrebno preveriti vsak bralni dostop v medpomnilniku, če je trenutna verzija dostopanega objekta še vedno ista. Ob potrditvi transakcije se z ukazom CAS atomarno zamenja status iz aktivnega v potrjenega. Vsi nadaljni dostopi bodo v lokatorju ubrali drugo pot in tako dostopali do nove verzije. Naslednja transakcija, ki odpre ta isti objekt za pisanje, ustvari nov lokator in z ukazom CAS atomarno prestavi referenco v "TMObject"iz starega lokatorja na novega.

Za zagotavljanje napredovanja (ker je sistem brez oviranja šibko zagotovilo) uporablja razsodnike sporov. DSTM definira vmesnik za razsodnike sporov preko katerega komunicirajo z STM.

6.2 McRT-STM, Baha et al

McRT-STM je novejši STM in v večini pogledov nasproten DSTM. Je del McRT okolja, narejenega za delo z večjedrnimi procesorji. Napisan je za delo z jezikoma C in C++. McRT-STM uporablja zaklepanje in ima dele kode v funkcijah za potrditev in razveljavitev transakcije, ki uporabljajo striktno dvofazno zaklepanje in so blokirajoči. Uporablja lahko LVM ali EVM, prav tako lahko uporabnik dinamično izbira med granularnostjo objekta ali pa bloka. Uporaba transakcijskih operacij je implicitna, kot je razvidno iz programa 8. Če to kodo primerjamo s kodo pri primeru uporabe DSTM, opazimo veliko razliko v preprostosti uporabe STM.

STM je blokirajoč, toda McRT sistem ima posebni kooperativni razvrščevalnik, ki je v veliko pomoč pri preprečevanju težav z živimi objemi. Hkrati pa lahko STM izkoristi prednosti izvedbe z zaklepanjem, kot so lažje upravljanje s pomnilnikom (ne potrebujemo dvojne indirekcije) in zmanjšano število razveljavitev (transakcija ob konfliktnem dostopu čaka, namesto da bi naprej delala

Program 8 Uporaba McRT-STM

```
transaction {
    counter++;
}
```

in ob potrjevanju se razveljavila). Nekatere tehnologije zahtevajo blokirajoč STM npr. klici RPC (angl. remote procedure call).

Vsaka ta podatkovna enota (objekt ali blok) v uporabi je zaščitena s ključavnico. Ključavnica je ena 32-bitna beseda. Njeni spodnji trije biti imajo poseben pomen.

- Obvestilni bit

  Obvestilni bit "N"(angl. notify bit) označuje, da obstajajo transakcije, ki so te podatke prebrale in čakajo na obvestila. Če je 0, takih transakcij ni.

- Neuporabljeni bit

  Neuporabljeni bit "U"(angl. unused bit) je ostanek starejših verzij.

- Bralni bit

  Bralni bit "R"(angl. reader bit) je 0 kadar si ključavnico lasti neka transakcija (torej le-ta ima pravico pisalnih dostopov), drugače je 1.

Začetna vrednost ključavnice je 4 (bit R postavljen). V takem stanju lahko vsi berejo podatke. Ob branju si vsaka transakcija v svoj medpomnilnik za sledenje bralnim dostopom shrani par, ki označuje ključavnico in verzijsko številko prebranih podatkov. Verzijska številka je ostalih 29 bitov v ključavnici.

Transakcija pridobi pisalni dostop tako, da najprej shrani trenutno verzijsko številko, nato pa v ključavniško besedo shrani kazalec na sebe (na svoj lasten opisnik). Ker so opisniki shranjeni na naslovih, ki so poravnani na 8-bitne meje, je spodnjih sedem bitov vedno 0. Torej je bit R enak 0 in so ostale transakcije obveščene, da je ključavnica pod tujim lastništvom, hkrati pa vrednost ključavnice same služi kot kazalec na lastnika. Ob potrditvi transakcije se ključavnica sprosti tako, da se v ključavniško besedo shrani vrednost, ki smo jo shranili pred pridobitvijo pisalnega dostopa, povečano za 8. To poveča verzijsko številko za ena, hkrati pa ne spremeni spodnjih treh bitov.

Vsaka ključavnica ima poleg tudi čakalni seznam, na katero se dodajo transakcije, ki želijo biti obveščene, če se vrednost podatkov, ki jih ta ključavnica ščiti, spremeni. Poleg tega je potrebno postaviti tudi N bit. Ta mehanizem je uporaben če želimo implementirati CCR. Če je bit N postavljen potem ob potrditvi transakcije, ki je imela lastništvo ključavnice, le-ta zbudi vse transakcije, ki so na čakalnem seznamu.

Vsaka transakcija ima opisnik, ki je ustvarjen ob stvarjenju niti in je shranjen v pomnilniku lokalnemu tej niti. Torej zaporedne transakcije iste niti uporabljajo isti opisnik (ki je seveda vsakič počiščen). Opisnik transakcije vsebuje:

- Statusno besedo

  Statusna beseda pove ali je transakcija aktivna, čaka, razveljavljena, potrjena.

- Globina gnezdenja

- Medpomnilnik za sledenje bralnim dostopom

  Seznam prebranih lokacij in njihovih verzij.

- Medpomnilnik za sledenje pisalnim dostopom

  Seznam ključavnic, ki si jih transakcija lasti.

- Medpomnilnik za hranjenje sprememb podatkov

  Medpomnilnik v katerem se hranijo originalni podatki z lokacij, ki jih ščitijo ključavnice pod lastništvom te transakcije. Beležijo se kot 32-bitne vrednosti. Večje spremembe se zapišejo kot več sprememb.

Zaznavanje konfliktov je hibridnega tipa (HCD). Konflikti tipa branje-po-pisanju (angl. read-after-write) in pisanje-po-pisanju (angl. write-after-write) se manifestirajo kot neuspela pridobitev ključavnice. Pisalni ali bralni dostop naleti na tuje lastništvo ključavnice. Konflikti tipa pisanje-po-branju (angl. write-after-read) pa se zaznavanjo ob preverjanju transakcije. Ob preverjanju se preverijo verzije vseh podatkov, ki so v medpomnilniku za sledenje bralnim dostopom. Če si je kakšna druga transakcija pridobila bralni dostop (lastništvo ključavnice) po bralnem dostopu te transakcije, se verzijske številke ne ujemajo (ali pa je bit R enak 0 kar pomeni, da nasprotujoča transakcija še traja).

Pri razveljavitvi transakcije se podatki tam, kjer ima ta transakcija lastništvo ključavnice, obnovijo iz medpomnilnika za hranjenje sprememb podatkov, kar razveljavi vse spremembe transakcije. Potem se sprostijo vsa lastništva ključavnic, ki so zabeležena v medpomnilniku za sledenje pisalnim ukazom, pri čemer se verzijska številka ne poveča.

Pri potrjevanju transakcije, se najprej izvede preverjanje transakcije, potem se pa sprostijo lastništva ključavnic in se poveča verzijska številka.

McRT-STM tudi omogoča uporabniku, da se registrira za obveščanje o potrditvah in razveljavitvah transakcije.

Poglavje 7

Hibridni transakcijski pomnilnik

7.1 Ideja

Hibridni transakcijski pomnilnik (angl. hybrid transactional memory — HyTM) je ideja, da bi imeli transakcijski pomnilnik, ki bi uporabljal tako STM kot HTM, da bi maksimiziral prednosti vsakega od teh [34]. Sistemi HTM so hitrejši. Začetek transakcije in njeno potrjevanje je samo nekaj strojnih ukazov, kar je velik kontrast sistemom STM, kjer nas ob potrjevanju transakcije čaka precejšnja količina dela. Potrebno je preveriti, da ni konfliktov, in ponekod prenesti rezultate transakcije v skupni pomnilniški prostor.

Sistemi imajo HTM hude pomanjkljivosti. Medpomnilniki, ki jih uporabljajo za shranjevanje metapodatkov, so zelo omejene kapacitete. Čeprav je večina kritičnih odsekov v programih zelo kratkih, je takšna pomanjkljivost za praktično rabo nedopustna. Sistemi HTM imajo tudi fiksno (majhno) število transakcijskih kontekstov s katerimi je omejeno število transakcij, ki se lahko hkrati izvajajo. Te pomanjkljivosti pri sistemih STM niso prisotne. Temeljna ideja HyTM je uporaba HTM z možnostjo izvajanja transakcij z STM v primeru, da so predolge ali pa so vsi transakcijski konteksti HTM zasedeni. Pri tem je zaželjeno, da lahko različne niti tečejo v različnih načinih (nekatere z HTM, nekatere z STM). Nekateri starejši predlogi za HyTM so namreč ob potrebi ene transakcije, da teče z STM, preklopili vse transakcije v STM kar je povzročilo izgubo hitrosti [34].

HyTM ima strojno osnovo v obliki rahlo razširjenega HTM. Za zaznavanje konfliktov uporablja že obstoječe protokole za ohranjanje pravilnosti predpomnilnika, ki so že v današnjih večjedrnih procesorjih. To omogoča hitro zaznavanje konflikov v primerjavi z STM. Vsak procesor je takoj obveščen, če drug procesor poskuša pisati v vrstico predpomnilnika, ki jo je prvi pro-

cesor špekulativno bral ali pisal. Če je konflikt med ukazom v transakciji in takim, ki ni, se transakcija razveljavi. Pri konfliktu med dvema transakcijama HyTM razsodi v korist transakcije, ki trenutno prosi za dostop. Tako razsojanje konfliktov ima to prednost, da ne zahteva nobenih sprememb obstoječih protokolov za ohranjanje pravilnosti predpomnilnika. Ob razveljavitvi transakcije HyTM sprazni strojne medpomnilnike, ki hranijo špekulativne vrednosti vrstic predpomnilnika.

Torej ima HyTM sistem za upravljanje z verzijami LVM in zaznavanje konfliktov ECD.

Pri HyTM se ob začetku transakcije izbere način izvajanja transakcije (HTM ali STM). Preklopi med načinoma delovanja sredi transakcije niso dovoljeni. Izbira načina je odvisna od preteklosti določene transakcije. Vsaka transakcija se najprej poskusi izvesti s HTM. Ob razveljavitvi se v transakcijskem kontekstu postavijo zastavice, ki opisujejo naravo razveljavitve transakcije in HyTM ponovno poskusi izvesti transakcijo. Če je do razveljavitve prišlo zaradi presega kapacitete medpomnilnikov HTM sistema, potem se transakcija v prihodnosti poskuša izvesti z STM sistemom. Ob razveljavitvi zaradi konflikta se pri naslednje poskusu izvajanja transakcije uporabi isti način kot dosedaj. V primeru, da ni na voljo prost transakcijski kontekst v HTM se transakcija izvaja z STM sistemom.

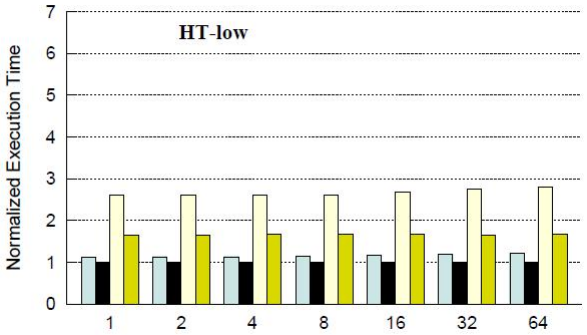
7.2 Rezultati

Sintetični testi sestavljeni iz operacij na razpršenih tabelah in drevesnih podatkovnih strukturah kažejo, da je v testih z nizko tekomvalnosjo med nitmi HyTM dvakrat do trikrat hitrejši od STM in hkrati za pol toliko počasnejši od HTM. Pri testih z visoko tekmovalnostjo med nitmi pa je glavni faktor primerna izbira upravitelja sporov (ki je pogosto zelo preprost in neoptimalen pri HTM), in zato se rezultati STM zelo razlikujejo med posameznimi testi.

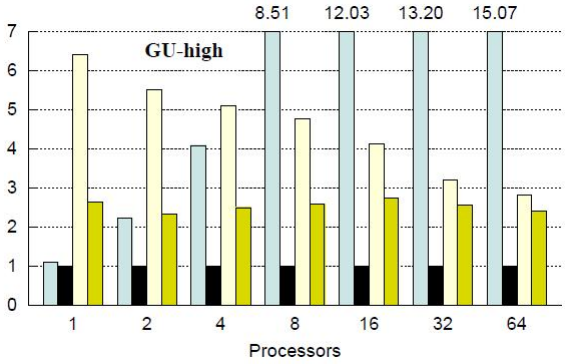
Primeri rezultatov testov so vidni na sliki 7.1(a) in sliki 7.1(b). Grafa predstavljata trajanje izvajanja testa, normalizirano glede na trajanje testa pri HTM. Abscisna os grafa označuje število procesorjev. Pri vsakemu številu procesorjev so štirje stolpci, ki prikazujejo rezultat pri uporabi zaklepanja, HTM, STM in HyTM (v tem vrstnem redu).

Iz slike 7.1(a) je razvidno, da HyTM približno prepolovi upočasnitev sistema STM glede na sistem HTM. Slika 7.1(b) prikazuje drugačen primer pri katerem ima zaklepanje s povečevanjem števila procesorjev vse slabše rezultate, STM pa se približuje HTM.

Ker je strojna razširitev iz HTM v HyTM minimalna je HyTM smiselna ideja, v primeru, da strojna podpora transakcijskemu pomnilniku postane standard.



(a) Test z razpršeno tabelo (nizka obremenitev)



(b) Test s posodabljanjem grafov (visoka obremenitev)

Slika 7.1: Testi hibridnega transakcijskega pomnilnika [34]

Poglavje 8

Testiranja zmogljivosti programskega transakcijskega pomnilnika in optimizacije

8.1 Lastnosti testov in bremen

Ker je transakcijski pomnilnik relativno mlado področje v računalništvu, mu manjkajo standardni testi in bremena, katera mora biti sistem sposoben obvladovati. To pomeni, da ne obstaja standarden nabor merilnih aplikacij, s katerimi bi lahko različne STM sisteme med seboj primerjali. Velika večina prispekov, ki so bili v tej nalogi uporabljeni, uporablja teste, ki so jih napisali avtorji sami. Ti testi skoraj vedno uporabljajo sintetična bremena, ponavadi v obliki izvedbe katere od znanih podatkovnih struktur z uporabo STM.

Testi bi morali biti prilagojeni realnim bremenom. Raziskave testnega paketa aplikacij JavaGrande kažejo, da ima velikost kritičnih odsekov (in s tem transakcij) zelo neenakomerno porazdelitev. Namreč kar polovica kritičnih odsekov je krajših od 150 strojnih ukazov, hkrati pa je 5% kritičnih odsekov daljših od 4200 strojnih ukazov. Najdaljši kritični odseki presegajo 10 milijonov strojnih ukazov [8]. Analiza narejena na jedru Linux 2.4.19 kaže podobno sliko. Večina kritičnih odsekov je kratkih, 99.9% jih opravi dostope samo do 54 različnih vrstic predpomnilnika (posamezna vrstica je 64 bajtov), hkrati pa obstajajo tudi kritični odseki, ki opravijo dostope do 7000 različnih vrstic predpomnilnika [2]. Kratke transakcije predstavljajo izziv za sisteme STM, ker je težko amortizirati (časovne) stroške operacij začetka transakcije in potrditve transakcije. Sintetični testi večinoma zahtevajo izvajanje velikega števila transakcij konstantne dolžine in zato slabo predstavljajo realna bremena.

Zahtevane kapacitete medpomnilnikov za metapodatke so podobno porazdeljene. Transakcije v povprečju dostopajo do nove lokacije vsakih 10 ukazov. Za 95% transakcij zadošča medpomnilnik za sledenje bralnim dostopom velikosti 1000 vnosov ter medpomnilnik za sledenje pisalnim vnosom velikosti 256 vnosov [8]. Ta podatek je sicer bolj pomemben za načrtovalce sistemov HTM in HyTM, vendar tudi za načrtovalce sistemov STM nosi precejšnjo vrednost, ker se v sistemih STM pogosto pridobiva pomnilniški prostor za medpomnilnike v kosih. Ta podatek snovalcem omogoča informirano odločitev o velikosti teh kosov.

8.1.1 Testni programi

Ker so avtorji sistemov STM zaenkrat prepuščeni samim sebi glede testiranja, nam njihova izbira marsikaj razkriva.

V člankih s področja STM je izredno priljubljen test, ki opravlja naključno vstavljanje in brisanje elementov iz razpršene tabele [21, 49, 27, 12, 7, 34]. Ta test je zelo priljubljen, ker ima visoko stopnjo paralelizma, ter majhno verjetnost konfliktnih dostopov. Ob vstavljanju in brisanju elementa iz razpršene tabele se dostopa samo do ciljnega elementa v polju. Zato je ta test idealen za STM, če želimo v študiji prikazati dobre rezultate (predvsem skalabilnist), in je temu primerno popularen.

Drugi izredno popularen test je opravljanje osnovnih operacij na rdeče-črnih drevesih [31, 32, 27, 15, 11]. Popularen je zato, ker je operacije na rdeče-črnih drevesih zelo težko zaščititi z zaklepanjem, ker vrstni red zaklepanja vozlišč ni vnaprej znan. Poleg tega se zaklepanje, kot pesimistična metoda sočasnosti, zelo slabo odreže pri vseh testih, ki uporabljajo drevesne podatkovne strukture (rdeče-črna drevesa, AVL, binarno iskalno drevo), ker mora ob dostopu do lista zakleniti celotno pod od korena do lista drevesa. To povzroči, da je koren drevesa (in druga vozlišča visoko v drevesu) ozko grlo, saj za njegovo ključavnico tekmujejo vse niti. Sistemi STM tu zopet dobijo precej dobre rezultate in so zato ti testi priljubjeni.

Manj popularni testi so testi tipa "proizvajalec-potrošnik", test povečevanja skupnega števca in testi naključnih operacij na skladu. Takšni testi nimajo v sebi možnosti za sočasnost in zato je pesimizem zaklepanja upravičen. Temu primerni so tudi rezultati, pri takih testih sistemi STM nimajo prednosti pred zaklepanjem, hkrati pa še vedno nosijo isto slabost dodatne birokracije ob dostopih. Ogromno je konfliktov in zato je pogosto več kot polovico dela predstavljajo razveljavljene transakcije [32]. V tem primeru lahko sistemi STM v primeru, da imajo dobro strategijo razreševanja konfliktov, pričakujejo v

najboljšem primeru rahlo upočasnitev s številom povečanja niti.

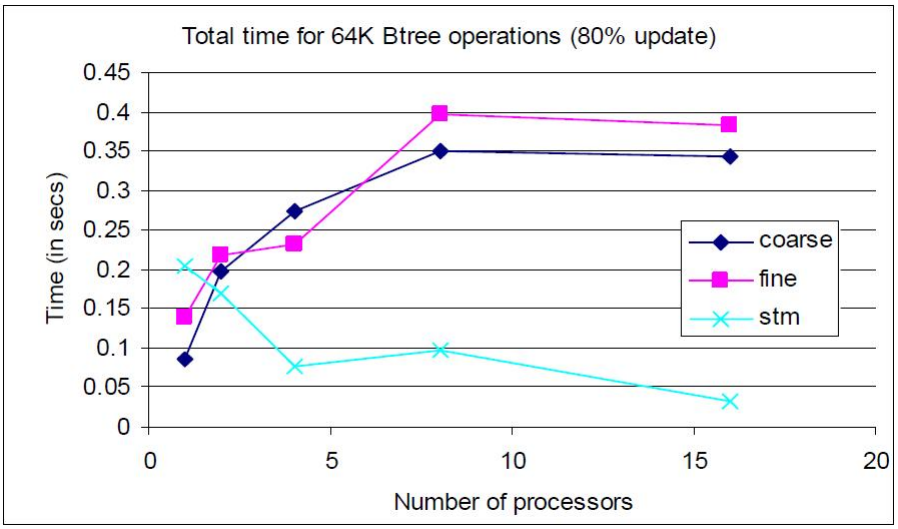
Testiranje na pravih uporabniških aplikacijah se pojavlja predvsem pri sistemih, ki avtomatsko zamenjujejo zaklepanje v programu s transakcijami [19]. Priljubljeni paketi testnih aplikacij so JavaGrande, SPECjbb2000, SPECjvm98. Od posameznih aplikacij se pojavlja predvsem spletni strežnik Apache.

8.1.2 Metrike

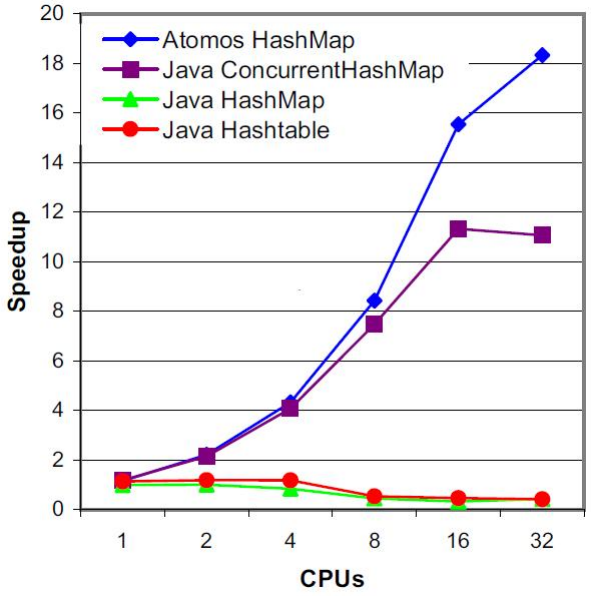
Glede na literaturo ni konsenza, kaj je osnovna metrika, ki beleži zmogljivost sistema STM. Osnovne so: število operacij na časovno enoto in absolutni čas izvajanja testa. Popularne tudi metrike, ki merijo relativni čas izvajanja glede na rezultat testa s samo eno nitjo. Te znajo biti zavajujoče, ker pokažejo samo skalabilnost sistema, ne pa absolutne hitrosti.

8.1.3 Rezultati

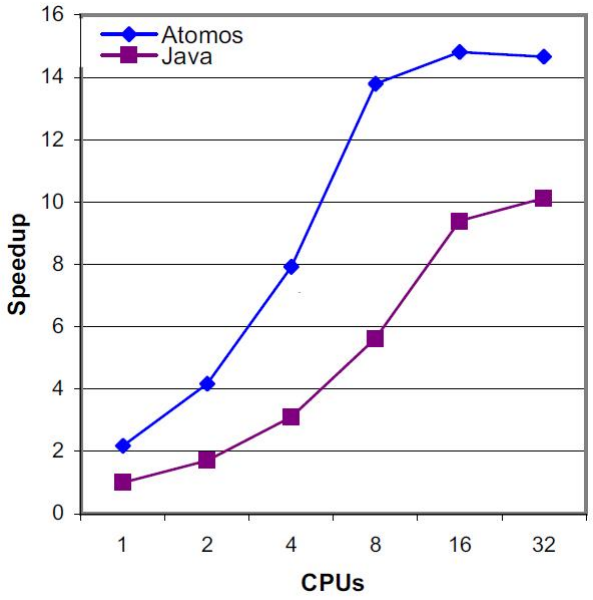
Slika 8.1 prikazuje rezultate testiranj sistema McRT-STM in primerjavo z grobim in drobnim zaklepanjem. Iz grafa je razvidno, da je sistem STM precej hitrejši, predvsem pa bolj skalabilen kot zaklepanje. Ta lastnost je izrazita na testih, ki vsebujejo drevesne podatkovne strukture.



Slika 8.1: Primerjava hitrosti sistema McRT-STM in zaklepanja [49]



(a) Test z razpršeno tabelo



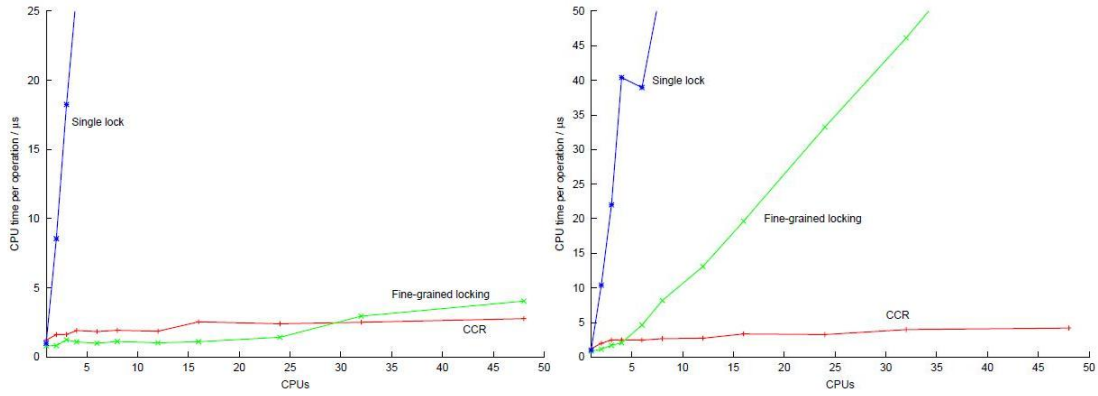
(b) Test z računanjem histograma

Slika 8.2: Primerjava hitrosti sistema Atomos in zaklepanja [7]

Slika 8.2 prikazuje rezultate testiranj sistema Atomos. Vidimo lahko, da so

podobni rezultatom pri McRT-STM, z razliko, da je drobno zaklepanje skoraj enako hitro. Atomos je sistem z močno atomarnostjo (McRT-STM ni).

Pri sistemu WSTM lahko na sliki 8.3 vidimo podobne rezultate. Vsem prikazanim rezultatom je skupno, da povečanje števila niti ne prinese povečanja zmogljivosti, ki bi jo pričakovali. Pri testu McRT-STM povečanje števila niti iz ene na 16 prinese samo petkratno povečanje hitrosti izvajanja testa. Sistemi STM so torej kljub hitrosti, ki je večja od zaklepanja, daleč od idealne hitrosti. Podobno sliko lahko vidimo pri Haskell STM. Testi na sliki 8.4 so bili izvedeni na sistemu s 128 procesorji. Izvedeni so z različnimi števili niti: 1, 2, 4, 8, 16, 32, 64, 120. Vidimo lahko, da je optimalno število niti različno pri različnih testih. To namiguje, da ne bomo iznašli idealne sheme STM, ki bi bila najboljša pri vseh nalogah.

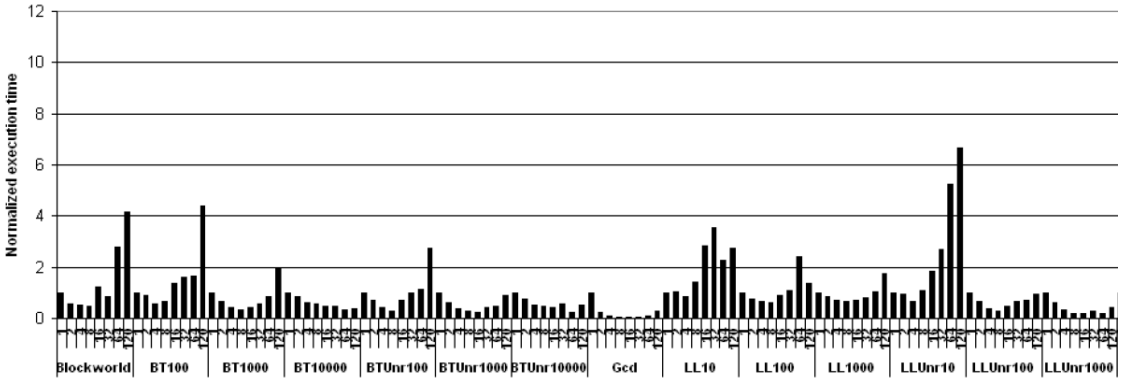


(*a*) Individual hashtable operations, 1% writes (*b*) Individual hashtable operations, 16% writes

Slika 8.3: Primerjava hitrosti sistema WSTM in zaklepanja [21]

8.2 Optimizacije

Glavna slabost ideje transakcijskega pomnilnika je upočasnitev, ki jo povzroči. Zato je minimizacija izgube hitrosti glavna prioriteta razvijalcev sistemov STM. Izgube lahko razdelimo na dve skupini. Izgube, ki nastanejo ob začetku in ob koncu transakcije, in izgube, ki nastanejo ob vsakem dostopu do pomnilnika. Prve so pomembne, ker je večina transakcij kratkih in zato fiksen "strošek"ob vsakem začetku in koncu transakcije predstavlja precejšnjo



Slika 8.4: Hitrost Haskell STM [47]

upočasnitev. Druge pa imajo večji potencial za optimizacijo. Ob dostopih se uporabljajo zapore, ki pa niso potrebne, kadar do podatkov ne dostopa več niti. Tu se da z odstranjevanjem zapor, kjer niso potrebne, veliko prihraniti še posebno pri sistemih z močno atomarnostjo.

8.2.1 Optimizacija začetka in konca transakcije

Nekateri sistemi STM ponovno uporabijo nekatere medpomnilnike v večih zaporednih transakcijah [27]. To sistemu prihrani nekaj operacij zaseganja pomnilnika. Ker ima ena nit lahko samo eno transakcijo hkrati, se po koncu transakcije medpomnilniki za metapodatke lahko ohranijo, in se ponovno uporabijo pri naslednji transakciji.

Najboljša rešitev so seveda HyTM, ki uporabljajo strojno podporo, kar znatno skrajša operacije na začetku in koncu transakcije. Kadar HyTM dela v strojnem načinu ni potrebno zaseganje pomnilnika za shranjevanje metapodatkov, saj se uporabljajo posebni strojni medpomnilniki. Strojni način tudi skrajša začetek in konec transakcije na samo nekaj strojnih ukazov.

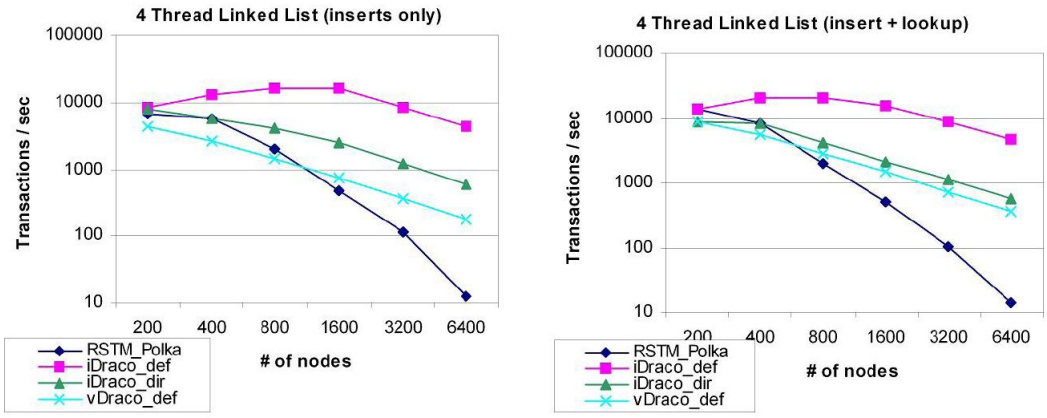
8.2.2 Optimizacija preverjanja transakcije

V svoji osnovni obliki preverjanje transakcije obsega primerjavo vsebine medpomnilnikov za metapodatke z vsebino globalnega pomnilnika. Primerja se verzije pisanih in branih podatkov transakcije z verzijami podatkov v globalnem pomnilniku. Obstaja pa tudi metoda, ki je večinoma hitrejša [16], če je teh podatkov veliko. Preverjanje z označevanjem nepravilnih transakcij je

obraten pristop. Namesto, da bi se preverjalo veljavnost ciljne transakcije, se privzema, da je ta pravilna in so ostale, ki so v konfliktu z njo, napačne. Napačne transakcije so označene in ob začetku njihovega preverjanja so takoj razveljavljene. Pri takem sistemu se v fazi preverjanja vzame vsebino medpomnilnika za sledenje pisalnim dostopom in se za vsak pomnilniški naslov v njem preveri medpomnilnike v vseh ostalih transakcijah. Tiste, ki vsebujejo katerega od teh naslovov se označene za napačne. Tukaj velja omeniti, da se lahko naslove v teh medpomnilnikih hrani v binarnem iskalnem drevesu, kar omogoča logaritmične čase preverjanja, če določena transakcija ima naslov v medpomnilniku. Potem, ko je ta postopek končan, se transakcija lahko potrdi.

Pridobitev te optimizacije je večja, če se transakcije dotaknejo veliko podatkov, in manjša, če je število hkratnih transakcij veliko. Zelo pomembna je tudi lastnost, da je pri metodi z označevanjem nepravilnih transakcij primer transakcije, ki samo bere, povsem trivialen. Takšna transakcija se lahko takoj potrdi ali pa razveljavi, če je označena kot napačna.

Na sliki 8.5 so vidni rezultati testov obeh verzij preverjanja transakcije. Oboje sta izvedbi sistma Draco STM [15]. Test obsega transakcije, ki vstavijo fiksno število elementov (abscisna os) v povezani seznam. Rezultati izvedbe z navadnim preverjanjem so označeni z “vDraco_def”, rezultati optimizacije pa z grafoma “iDraco_dir”ter “iDraco_def”(oznaka “dir”pomeni upravljanje z verzijami EVM, oznaka “def”pa LVM). Ker je namen te optimizacije pohitriti transakcije, ki berejo ali pišejo velike količine podatkov, rezultati niso presenetljivi. Tak test je precej pristranski in, glede na tipična bremena, nere-



Slika 8.5: Testi izvedb z različnim preverjanjem transakcije [16]

alističen. Testni razpon se začne s transakcijami, ki opravijo po 200 vstavljanj v podatkovne strukture, in se nadaljuje vse do transakcij, ki vstavijo 6400 elementov v podatkovno strukturo. Velika večna kritičnih odsekov in transakcij je močno krajših. Tipično manipulirajo samo z nekaj pari objektov. Tak test zato ne da prave slike o uporabnosti te optimizacije pri tipičnem uporabniku.

8.2.3 Optimizacija dostopov

Sistemi STM, ki želijo imeti neblokirajočo izvedbo potrjevanja transakcij, je nujna uporaba večnivojskih preslikav med oprimkom objekta (s katerim razpolaga uporabnik) in dejanskimi podatki. To povzroči precej počasnejše dostope. Pri DSTM je potrebno prehoditi dve referenci za dostop do podatkov. To pa ni edina cena, ki jo plačamo. Vsak nivo preslikovanja nas izpostavi možnosti dodatne zgrešitve v predpomnilniku in zahteva, da naložimo dodatne vrstice predpomnilnika. To dodatno upočasni delovanje. Sistemi STM naj bi torej stremeli k čimvečji lokalnosti podatkov in metapodatkov. Če STM uporablja zaklepanje imamo na tem področju precej bolj proste roke, kot pri neblokirajočih sistemih STM. Ennals STM objektom doda poseben oprimek, ki kaže na metapodatke v posebnem pomnilniškem prostoru, ki je viden samo določeni transakciji. Ker je ta prostor majhen in se uporablja v več zaporednih transakcijah, je skoraj gotovo v predpomnilniku. Oprimek, ki kaže nanj, pa je poleg objekta in zato večinoma ne povzroči dodatnih zgrešitev v predpomnilniku.

Druga vrsta optimizacij je zmanjševanje števila bralnih in pisalnih zapor. Nekateri sistemi STM s šibko atomarnostjo jih nimajo [27]. V osnovni izvedbi so vstavljane okoli vsakega pomnilniškega dostopa v transakciji. Če je sistem z močno atomarnostjo, so vstavljane tudi okoli vseh ostalih pomnilniških dostopov v programu. To privede do močnih upočasnitev, ki jih lahko odpravimo z vrsto optimizacij.

Odstranjevanje zapor

Odstranjevanje zapor (angl. barrier elimination) je proces, ki odstani zapore tam kjer so nepotrebne. Tukaj se kombinira več prijemov. S statično analizo poskušamo določiti, kateri objekti so lokalni glede na nit (angl. thread-local) in kateri niso nikoli dostopani v transakcijah. To sta dve komplementarni analizi, ker velikokrat objekti spadajo v eno skupino in ne v drugo. Pri teh dostopih lahko odstranimo zapore. Učinkovitost kombinacije teh pristopov pri močni atomarnosti je prikazana na sliki 8.6. Tretji stolpec prikazuje število ovir v netransakcijski kodi, zadnji stolpec pa prikazuje število odstranjenih ovir po

program	type	total	barrier removed by		
			NAIT-TL	TL-NAIT	TL+NAIT
JVM98	read	12671	8796	0	12671
	write	9885	7961	0	9885
tsp	read	106	89	0	93
	write	36	16	0	17
OO7	read	300	279	0	292
	write	136	114	2	117
JBB	read	804	364	24	798
	write	621	131	344	575

Slika 8.6: Učinki statične analize [51]

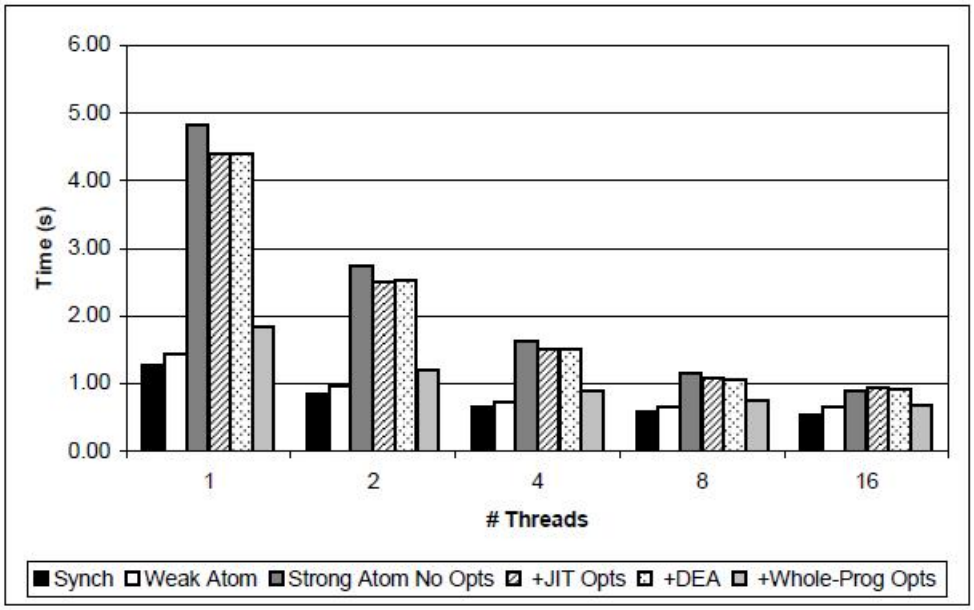
analizi. Prevajalnik lahko tudi odstranjuje zapore pri dostopih do objektov, ki so nespremenljivi.

Združevanje zapor

Prevajalnik lahko zazna, da se izvaja več zaporednih dostopov do istega objekta znotraj enega bloka. V tem primeru lahko izvede združitev zapor (angl. barrier aggregation) v eno samo zaporo. Tako se več sprememb objekta izvede znotraj ene same zapore, kar zmanjša število zapor v programu.

Dinamična ubežna analiza

Dinamična ubežna analiza (angl. dynamic escape analysis) zaznava, ali je objekt viden iz več niti [51]. Ob stvarjenju je objekt zaseben glede na neko nit, javen pa postane, ko se njegova referenca zapiše v kak drug javen objekt ali v statično polje. Analiza je dinamična, ker se izvaja med delovanjem programa. Rezultati optimizacij so na sliki 8.7. Test je problem "potujoči trgovec"(angl. travelling salesman problem), ki se izvaja na računalniku s 16 procesnimi enotami. Prvi stolpec kaže rezultat pri uporabi zaklepanja, drugi kaže rezultat pri uporabi sistema STM s šibko atomarnostjo, tretji kaže rezultat pri uporabi sistema STM z močno atomarnostjo in zadnji kaže rezultat sistema STM z močno atomarnostjo po optimizacijah.



Slika 8.7: Test optimizacij močne atomarnosti [51]

Poglavje 9

Sklepne ugotovitve

Programski transakcijski pomnilnik je precej mlado in, z raziskovalnega stališča, živahno področje. Če gre verjeti avtorjem raziskav, je to naslednik zaklepanja, ki smo ga dolgo čakali. Raziskovalne naloge so polne optimističnih testov, ki pa imajo včasih zelo pristransko izbrane parametre. Vsekakor bo potrebno počakati na konsenz v raziskovalni skupnosti glede standardnih testnih bremen za sisteme STM.

Dan Grossman je podal zanimivo primerjavo med STM in sistemom za čiščenje in strnjevanje pomnilnika (angl. garbage collector — GC) v času preden se je ta uveljavil [17]. Mnogi so trdili, da je GC prepočasen brez strojne podpore. Drugi so trdili, da bo GC postal standard desetletja preden se je to dejansko zgodilo. Mnoge je tudi skrbelo, da je GC v nekaterih primerih premalo natančen in da zato potrebuje nek mehanizem, s katerim bi obšli njegovo delovanje. Zelo podobne stvari lahko danes slišimo o STM. Čeprav je ta analogija zanimiva, po mojem mnenju ni povsem natančna.

Za tehnologijo GC ni stal noben poseben ekonomski interes. Pri tehnologiji STM pa vidimo veliko zanimanja s strani gigantov industrije (Intel in Sun). Proizvajalci strojne opreme, kot je Intel, imajo v svojih laboratorijih procesorje z več ducat jedri, vendar jih ne morejo prodati, saj je izredno malo programov, ki bi izkoristili osemjedrne procesorje kaj šele več. Ta podjetja imajo ekonomski interes, da poskusijo uveljaviti STM in tako olajšati pisanje aplikacij, ki uporabljajo vzporedno procesiranje. Obstoj programov, ki se brez težav prilagodijo poljubnemu številu procesnih enot, pogojuje prodajo njihovih prihajajočih izdelkov. Za ta namen je morda še pomembnejša tehnologija implicitnega vzporednega izvajanja, za katero je STM odličen osnovni gradnik.

Po pregledu različnih vrst STM se postavlja vprašanje, kakšni bodo prihodnji trendi razvoja področja. Na začetku so bili sistemi brez zaklepanja,

kasneje pa brez oviranja. To lahko pripišemo temu, da se je želelo posnemati strojni transakcijski pomnilnik, ki ne uporablja zaklepanja. Drugi razlog je, da se raziskovalci niso želeli vračati k tehnologiji, ki so jo z STM želeli nadomestiti. Toda kmalu se je pokazalo, da sistemi STM z zaklepanjem prinašajo veliko prednosti in predvsem hitreje delujejo. Sistemi STM z zaklepanjem omogočajo večjo lokalnost podatkov in več možnosti pri razsojanju sporov. Če pogledamo raziskovalne naloge objavljene v zadnjih dveh letih, vidimo, da je vse več sistemov STM izvedenih z zaklepanjem. Pri drugih lastnostih se pa uporablja vse več hibridnih pristopov in dinamičnega izbiranja lastnosti STM. V nekaterih primerih je boljša zrnatost objektov, v drugih pa zrnatost blokov. McRT-STM omogoča izbiro te lastnosti za vsako transakcijo posebej [49]. Razsodniki sporov so še en del STM, kjer je uporabniku omogočena izbira neodvisno od ostalega sistema STM. Vse kaže, da ne obstaja en sam nabor implementacijskih izbir (zrnatost, zaznavanje konfliktov, nadzor verzij), ki bi bil optimalen za vse tipe programov.

Zaenkrat je nabor prosto dostopnih izvedb STM omejen. Če želimo delati z nekaterimi manj popularnimi (ali mlajšimi) programskimi jeziki (LISP, Scala, Haskell), izbire praktično ni. Kratek seznam implementacij je na voljo v članku o STM na Wikipediji [54]. Veliko raziskovalnih nalog na univerzah financirajo prej omenjena podjetja (Intel, Sun, Microsoft). Lahko opazimo tudi migracijo raziskovalcev iz univerze k podjetjem (v starejšem članku je neki avtor naveden kot raziskovalec na univerzi, v novejšem pa kot zaposlenec Microsoft Research). Zato je trenutno smiselno uporabljati izdelke teh podjetij, saj imajo vgrajene napredke predstavljene v člankih teh avtorjev, poleg tega pa imajo močno zaledje. Za C# je na voljo "SXM"(Microsoft Research), za C++ obstaja STM razširitev za Intelov prevajalnik in za Javo je na voljo Sunov "DSTM2"[18, 44, 42]. Omeniti se splača še "TBoost.STM", izvirajoč iz "Draco-STM", ki je zelo soliden in predvsem nastavljiv sistem STM za jezik C++ [14]. Kdor si želi preizkusiti veliko izvedb STM, naj si pa ogleda RSTM, ki je paket izvedb STM za C++ (trinajst izvedb) [45]

Vse kaže, da se bodo morali tudi uporabniki STM izobraziti o delovanju in lastnostih STM. Čeprav je osnovna uporaba STM izredno preprosta (in ponavadi so privzete nastavitve sistema STM zadovoljive), bo potrebno vedeti, katere izbire parametrov STM so primerne za določene situacije, če želimo optimalno zmogljivost programa. Potrebno se je tudi zavedati, da so nekatere programske rešitve problematične za uporabo STM. Klasičen primer je program, kjer več niti povečuje en števec. V tem primeru potencialne vzporednosti preprosto ni. Samo ena nit hkrati lahko povečuje tak števec. Uporaba STM na takem primeru bo privedla do ogromno konfliktov ter razveljavitev in je celo

slabša od uporabe preprostega zaklepanja. Tukaj mora programer sam rešiti problem z dodatno kodo, ki problem odpravi z drugačno podatkovno strukturo. Zanimiv primer ponuja tudi testni program pri sistemu STM z močno atomarnostjo, kjer je vse delo potekalo na poljih, ki so definirana statično [51]. Ta test je imel katastrofalno slabe rezulate. V tem primeru je namreč odstranjevanje zapor praktično nemogoče. Takšne stvari bo moral uporabnik STM poznati in se jim izogibati.

S stališča odpornosti na napake uporabnika bi bil tudi smiseln razvoj v smeri močne atomarnosti. Izkazalo se je, da je njene pomanjkljivosti mogoče z optimizacijami v veliki meri odpraviti. Morebitni problem hitrosti sistema STM gotovo ne bo imel takšnega negativnega vpliva, kot bi mu ga pripisali. V industriji načrtovanja programske opreme se je že večkrat pokazalo, da je povečanje produktivnosti programerja pomembnejši faktor od hitrosti izdelka (sicer bi še danes pisali programe v nizkonivojskih jezikih zato, ker so hitrejši). Objektno orientirano programiranje tudi odlično sodeluje s sistemi STM. Pri objektno orientiranem programiranju je ideal pisanje majhnih kosov kode, ki so zaključena celota in ne potrebujejo zunanjih posegov, prav tako pa pri njihovi zasnovi ne potrebujemo podrobnega poznavanja ostalih komponent programa. Zaklepanje je ta ideal rušilo, izolacijska lastnost transakcij pa učinkuje pozitivno.

Splošna uporaba ni daleč. Oddaljena je toliko, kolikor je oddaljena standardizacija področja. Bodoči uporabniki čakajo, da se standardizira uporaba STM in da se pojavi nekaj vodilnih izvedb. Brez dvoma tehnologijo programskega transakcijskega pomnilnika čaka svetla prihodnost.

Literatura

[1] Vikas Agarwal, M.S. Hrishikesh, Stephen W. Keckler, Doug Burger, *Clock rate versus ipc: The end of the road for conventional microarchitectures*, 2000.

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, Sean Lie, "Unbounded transactional memory," v zborniku *11th IEEE International Symposium on High-Performance Computer Architecture*, 316–327, 2005.

[3] Gregory R. Andrews, *Concurrent programming: principles and practice*, Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[4] Jean Bacon, Tim Harris, *Operating systems: Concurrent and distributed software design*, Addison Wesley, 2003.

[5] Philip A. Bernstein, Nathan Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys*, 13:185–221, 1981.

[6] Cachopo Joao, António Rito-Silva, "Versioned boxes as the basis for memory transactions," *Sci. Comput. Program.*, 63(2):172–185, 2006.

[7] Brian D. Carlstrom, Austen Mcdonald, Hassan Chafi, Jaewoong Chung, Chi Cao Minh, Christos Kozyrakis, Kunle Olukotun, *The atomos transactional programming language*, 2006.

[8] Jaewoong Chung, Hassan Chafi, Chi Cao Minh, Austen Mcdonald, Brian D. Carlstrom, Christos Kozyrakis, Kunle Olukotun, "The common case transactional behavior of multithreaded programs," v zborniku *Proceedings of the 12th International Conference on High-Performance Computer Architecture*, 2006.

[9] M. Rinard, C.S. Ananian, "Efficient object-based software transactions," v zborniku *SCOOL '05: Proceedings of Workshop on Synchronization and Concurrency in Object-oriented Languages*, str. 1–10, 2005.

[10] D. Dice, N. Shavit, "What really makes transactions faster?," v zborniku *Proc. of the 1st TRANSACT 2006 workshop*, 2006.

[11] Robert Ennals, *Software transactional memory should not be obstruction-free*, Technical report, 2006.

[12] Keir Fraser, *Practical lock freedom, doktorska disertacija*, Cambridge University Computer Laboratory, 2003.

[13] Chen Fu, Zhibo Wu, Xiaoqun Wang, Xiaozong Yang, "A review of software transactional memory in multicore processors," *Information Technology Journal*, 2009.

[14] Justin E. Gottschlich, "Toward.boost.stm,"2008-2009. `http://eces.colorado.edu/ gottschl/tboostSTM/index.html`.

[15] Justin E. Gottschlich, Daniel A. Connors, "Dracostm: a practical c++ approach to software transactional memory," v zborniku *LCSD '07: Proceedings of the 2007 Symposium on Library-Centric Software Design*, str. 52–66, New York, NY, USA, 2007, ACM.

[16] Justin E. Gottschlich, Daniel A. Connors, "Optimizing consistency checking for memory-intensive transactions," v zborniku *PODC '08: Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, str. 451–451, New York, NY, USA, 2008, ACM.

[17] Dan Grossman, "The transactional memory / garbage collection analogy," v zborniku *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, str. 695–706, New York, NY, USA, 2007, ACM.

[18] Rachid Guerraoui, Maurice Herlihy, Bastian Pochon, "Polymorphic contention management," v zborniku *Proceedings of the 19th International Symposium on Distributed Computing (DISC 2005*, str. 26–29. LNCS, Springer, 2005.

[19] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos

Kozyrakis, Kunle Olukotun, "Transactional memory coherence and consistency," v zborniku *ISCA*, str. 102–113, 2004.

[20] Tim Harris, "Design choices for language-based transactions," v zborniku *In Proceedings of the 18th ACMSIGPLAN conference on Object-Oriented Programing,Systems,Languages,and Applications(OOPSLA03)*, ACMPress, str.388-402, 2003.

[21] Tim Harris, *Language support for lightweight transactions*, 2003.

[22] Tim Harris, *Exceptions and side-effects in atomic blocks*, 2004.

[23] Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy, "Composable memory transactions," *Commun. ACM*, 51(8):91–100, 2008.

[24] Tim Harris, Mark Plesko, Avraham Shinnar, David Tarditi, "Optimizing memory transactions," v zborniku *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, str. 14–25, New York, NY, USA, 2006, ACM.

[25] Maurice Herlihy, "Obstruction-free synchronization: Double-ended queues as an example," v zborniku *In Proceedings of the 23rd International Conference on Distributed Computing Systems*, str. 522–529. IEEE Computer Society, 2003.

[26] Maurice Herlihy, J. Eliot, B. Moss, "Transactional memory: Architectural support for lock-free data structures," v zborniku *Proceedings of the 20th Annual International Symposium on Computer Architecture*, str. 289–300, 1993.

[27] Maurice Herlihy, Victor Luchangco, Mark Moir, *Software transactional memory for dynamic-sized data structures*, str. 92–101. ACM Press, 2003.

[28] Rich Hickey, "Clojure - refs,"2008-2009. `http://clojure.org/refs`.

[29] C. A. R. Hoare, *Towards a theory of parallel programming*, str. 231–244, 2002.

[30] William N. Scherer III, Michael L. Scott, *Contention management in dynamic software*, 2004.

[31] William N. Scherer III, Michael L. Scott, *Contention management in dynamic software transactional memory*, 2004.

[32] William N. Scherer III, Michael L. Scott, *Advanced contention management for dynamic software transactional memory*, 2005.

[33] Alain Kagi, Doug Burger, James R. Goodman, *Efficient synchronization: Let them eat QOLB*, 1997.

[34] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, Anthony Nguyen, *Hybrid transactional memory*, 2006.

[35] H. T. Kung, John T. Robinson, "On optimistic methods for concurrency control," *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[36] Virendra J. Marathe, William N. Scherer III, Michael L. Scott, "Adaptive software transactional memory," v zborniku *In Proc. of the 19th Intl. Symp. on Distributed Computing*, str. 354–368, 2005.

[37] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer III, Michael L. Scott. "Lowering the overhead of nonblocking software transactional memory," *Dept. of Computer Science, Univ. of Rochester*, 2006.

[38] Bill McCloskey, Feng Zhou, David Gay, Eric Brewer, "Autolocker: synchronization inference for atomic sections," v zborniku *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, str. 346–358, New York, NY, USA, 2006, ACM.

[39] Cameron McNairy, "The next product in the Itanium processor family," v zborniku *The 16th Hot Chips Symposium*, 2004.

[40] Maged M. Michael, Michael L. Scott, "Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors," *Journal of Parallel and Distributed Computing*, 51:1–26, 1998.

[41] Polyvios Pratikakis, Michael Hicks, Jeffrey S. Foster, "Lock inference for atomic sections," v zborniku *Proc. of the 1st TRANSACT 2006 workshop*, 2006.

[42] Sun Microsystems, "Product downloads: Dynamic software transactional memory library 2.0,"2009. `http://www.sun.com/download/products.xml?id=453fb28e`.

[43] Mark Moir, "Transparent support for wait-free transactions," v zborniku *In Proceedings of the 11th International Workshop on Distributed Algorithms*, str. 305–319. Springer-Verlag, 1997.

[44] Intel Software Network, "Intel c++ stm compiler, prototype edition 3.0,"2008-2009. `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition-20/`.

[45] Department of Computer Science University of Rochester, "Rochester software transactional memory,"2009. `http://www.cs.rochester.edu/research/synchronization/rstm/`.

[46] Cristian Perfumo, Nehir Sonmez, Adrian Cristal, Osman S. Unsal, Mateo Valero, Tim Harris, "Dissecting transactional executions in Haskell," v zborniku *In The Second ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*, 2007.

[47] Cristian Perfumo, Nehir Sonmez, Srdjan Stipic, Osman Unsal, Adrian Cristal, Tim Harris, Mateo Valero, "The limits of software transactional memory (stm): Dissecting Haskell STM applications on a many-core environment," v zborniku *CF '08: Proceedings of the 5th conference on Computing frontiers*, str. 67–78, New York, NY, USA, 2008, ACM.

[48] Ravi Rajwar, James R Goodman, "Transactional lock-free execution of lock-based programs," v zborniku *In Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, str. 5–17. ACM Press, 2002.

[49] Bratin Saha, Ali Reza Adl-tabatabai, Richard L. Hudson, Chi Cao Minh, Benjamin Hertzberg, "McRT-STM: A high performance software transactional memory system for a multi-core runtime," v zborniku *In Proc. of the 11th ACM Symp. on Principles and Practice of Parallel Programming*, str. 187–197. ACM Press, 2006.

[50] Nir Shavit, Dan Touitou, *Software transactional memory*, 1995.

[51] Tatiana Shpeisman, Vijay Menon, Ali reza Adl-tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, Bratin Saha. "Enforcing isolation and ordering in STM," v zborniku *In the Proceedings of the Conf. on Programming Language Design and Implementation*, str. 78–88, 2007.

[52] Adam Welc Suresh, Suresh Jagannathan, Antony L. Hosking. "Transactional monitors for concurrent objects," v zborniku *ECOOP 2004 - Object-Oriented Programming, 18th European Conference*, 2004.

[53] Herb Sutter, "The trouble with locks," *C/C++ Users Journal, 23(3)*, 2005.

[54] Wikipedia, "Software transactional memory,"2009. `http://en.wikipedia.org/wiki/Software_transactional_memory`.