

Samenvatting

Op moderne computers draaien veel processen gelijktijdig, b.v. de window manager, de mailer, de tekstverwerker, de klok enz. Al deze processen hebben toegang tot het gedeelde geheugen nodig, maar als zij zonder beperkingen dit gedeelde geheugen lezen en wijzigen kunnen de gegevens inconsistent worden. De klassieke manier om dit te vermijden is het afdwingen van synchronisatie door middel van *locks*. Dit kan echter tot een blokkade leiden.

Wegens blokkades kunnen klassieke synchronisatieparadigma's gebaseerd op locks vele problemen ondervinden zoals *convoying*, prioriteits-inversie en *deadlock*. In de afgelopen twee decennia hebben een aantal onderzoekers technieken voorgesteld om *lock*-vrije implementaties te ontwerpen. Deze technieken staan gezamenlijke bewerking van gedeelde datastructuren toe zonder gebruik te maken van kritieke secties die door locks beschermd worden. Essentieel voor dergelijke implementaties is het gebruik van geavanceerde machine-instructies zoals *LL/SC* of *CAS*. Lock-vrije implementaties garanderen dat binnen een eindig aantal stappen altijd één of ander proces dat een handeling op het object probeert uit te voeren zijn taak zal voltooien. Vele problemen die ontstaan door falen en prioriteits-inversie kunnen zij vermijden. Lock-vrije synchronisatie is zeer belangrijk, met name in *real-time* systemen waar het zeer onwenselijk is een geblokkeerd proces te stoppen als deze een belangrijke of real-time taak uitvoert.

De correctheid van een ontwerp in het vroegst mogelijke stadium waarborgen is een belangrijke uitdaging voor elke verantwoorde methode voor systeemontwikkeling. Lock-vrije algoritmen zijn over het algemeen zeer complex en moeilijk om correct te ontwerpen. De enige techniek die wij kennen is het in een formele taal specificeren van het programmeringsmodel van het gedrag van het systeem en het wiskundig verifiëren dat het systeemontwerp en de implementatie aan *safety* en *liveness*-eigenschappen voldoen. In het algemeen er zijn twee verificatie methodes voor systeemontwerp: model checking en stellingbewijzen. Model checking is gebaseerd op automatische en uitputtende verkenning van de bereikbare

toestandsruimte van het systeem. De in dit proefschrift gepresenteerde lock-vrije algoritmen zijn te ingewikkeld en te omvangrijk qua gegevens voor model checking. Stellingbewijzen kan deze zogenaamde toestandsruimte-explosie vermijden door een compacte (of logische) beschrijving van toestanden en toestandsovergangen te gebruiken. Wij hebben daarom gekozen voor de interactieve stellingbewijzer PVS als mechanisch hulpmiddel. Met uitzondering van enkele informele bewijzen voor een paar liveness-eigenschappen, zijn alle algoritmen en bewijzen uit dit proefschrift geformaliseerd en gecontroleerd met PVS.

Het is zeer moeilijk om een eerlijke prestatievergelijking te maken van verschillende, concurrente implementaties van één algoritme, aangezien de prestaties van parallelle verwerking enorm beïnvloed wordt door de machinearchitectuur, de relatieve omvang van de datastructuren in vergelijking met omvang van de caches, en zelfs de verdeling van de processen over de processoren. Elke redelijke vergelijking zal betwistbaar zijn en verouderd raken door de introductie van andere architecturen. We kunnen daarom geen volledig empirische onderbouwing geven voor de gepresenteerde algoritmen.

Hoofdstuk 2 geeft een efficiënt lock-vrij algoritme voor hash tables met open adressering. Het algoritme is dynamisch in de zin dat het, indien nodig, groeien en inkrimpen van de hash table toestaat. Experimenten wijzen er op dat het algoritme lineair schaalt met het aantal processen. Toevoegen, verwijderen of opvragen van elementen blijkt gemiddeld in constante tijd te kunnen. Een schijnbare tekortkoming van ons algoritme is de ruimte-complexiteit welke in het slechtste geval evenredig is met het aantal processen maal de omvang van de hash table. Echter, als alle processen gewoon voortgang boeken en de hash table niet te klein is, is de daadwerkelijke geheugenvereiste evenredig aan de omvang van de hash table.

Alhoewel PVS aanzienlijke hulp biedt voor het beheren en opnieuw gebruiken van bewijzen, moeten wij toegeven dat het controleren van het algoritme, wegens de complexiteit van het algoritme, zeer ingewikkeld was. Het gehele correctheidsbewijs van het algoritme omvat zo'n 200 invarianten. De totale inspanning voor de verificatie kan ruwweg geschat worden op twee manjaar.

Hoofdstuk 3 formaliseert de algemene methodologie van Herlihy voor het transformeren van een sequentiële implementatie van een datastructuur naar lock-vrije synchronisatie, en stelt een lock-vrij patroon als een reductiestelling voor. De reductiestelling stelt ons in staat te redeneren over een lock-vrij programma dat ontworpen wordt op een hoger niveau dan de synchronisatieprimitieven *LL/SC*. De stelling is gebaseerd op *refinement mappings* zoals beschreven door Lamport. Toepassing van deze stelling vereenvoudigt de verificatie voor

lock-vrije algoritmen aangezien er minder invarianten vereist zijn en sommige invarianten gemakkelijker te ontdekken en te formuleren zijn zonder de interne structuur van de definitieve implementatie te beschouwen. Voorts worden twee verbeterde, alternatieve algoritmen voorgesteld waar het onnodige kopiëren van een groot object vermeden wordt in gevallen waarbij slechts een klein deel van het object gewijzigd wordt.

Veel machines bieden *CAS* danwel *LL/SC* aan, maar niet allebei. Hoofdstuk 4 geeft een zelfde lock-vrij patroon dat gebaseerd is op de zwakkere atomaire primitieve *CAS*, zonder dat het zogenaamde *ABA* probleem of problemen met *wrap around* optreden. Het is een variatie op de algemene methodologie van Herlihy voor lock-vrije transformatie.

Hoofdstuk 5 presenteert een lock-vrij parallel algoritme voor mark&sweep *garbage collection* (GC) in een realistisch model met gebruikmaking van de synchronisatieprimitieven *LL/SC* of *CAS*. Een aantal mutators en collectors kunnen gelijktijdig de datastructuur bewerken. In het bijzonder is een strikte afwisseling tussen gebruik en het opruimen niet noodzakelijk, wat bij de meeste andere algoritmen voor garbage collection wel gebruikelijk is. Om het bewijs te vereenvoudigen breiden wij de specificatie uit tot een implementatie op hoog niveau, verifiëren dan de correctheid van de implementatie op hoog niveau, en passen als laatste de reductiestelling uit hoofdstuk 3 toe om atomaire stappen op het hoge niveau te implementeren met behulp van de primitieven van het lage niveau.

Het algoritme gebruikt een procedure *Mark_stack* welke in feite een vorm van graph search is en aanvankelijk ontworpen is als een recursieve procedure. Het is verrassend dat het bewijs mogelijk wordt door de recursie weg te werken ten gunste van een expliciete stapel. Het zou veel moeilijker zijn (als het al mogelijk was) om de correctheid van de recursieve procedure te bewijzen, waarvoor we ons zouden moeten baseren op dekpuntssemantiek van recursieve procedures of een andere denotationele semantiek.

Naast safety eigenschappen, hebben wij ook onderzoek gedaan naar het belangrijke probleem van verificatie van liveness eigenschappen onder aanname van strong fairness. Liveness eigenschappen worden vaak in termen van een “*leads-to*” relatie uitgedrukt. Er wordt algemeen verondersteld dat deze moeilijker zijn te verifiëren dan safety eigenschappen. Wij bemerkten dat de “*steps-to*” (\triangleright) relatie en de “*unless*” (\mathcal{U}) relatie erg nuttig zijn om de “*leads-to*” ($\circ\rightarrow$) relatie te bewijzen, aangezien deze twee relaties slechts één enkele stap omvatten, en direct door PVS met behulp van invarianten kunnen worden gecontroleerd.

Een van de voornaamste observaties is dat het bewijs met PVS opzienbarend ingewikkeld is in vergelijking met de omvang van het bewezen algoritme. Havelund en Shankar, [26],

geven toe dat hun reductie het bewijs niet vereenvoudigde omdat de meeste inspanning ging zitten in het verifiëren van de refinement relatie. Wij hebben de indruk dat in hun geval het gat tussen abstractie en implementatie te groot is. In ons geval van de reductiestelling zijn er slechts zes invarianten en kost het niet veel moeite om de refinement relatie tussen abstractie en implementatie aan te tonen in het lock-vrije patroon. Bij gebruik van de reductiestelling postuleren we in totaal slechts 72 invarianten in het correctheidsbewijs van de implementatie op een hoog niveau van de lock-vrije GC, aangezien wezenlijke delen van het concrete programma als atomaire statements op het hoge niveau kunnen worden behandeld. De totale verificatie duur kan ruwweg geschat worden op een half manjaar. Dit is beduidend minder dan we ons voor de correctheid van de lock-vrije hashtables nodig hadden.