Compactie van programma's na het linken

Bjorn De Sutter

Promotor: prof. dr. ir. K. De Bosschere

Proefschrift ingediend tot het behalen van de graad van
Doctor in de Toegepaste Wetenschappen: Computerwetenschappen

Vakgroep Elektronica en Informatiesystemen
Voorzitter: prof. dr. ir. J. Van Campenhout
Faculteit Toegepaste Wetenschappen
Academiejaar 2001–2002

Aan m'n ouders,
Gerda en Willem.

Woord vooraf

"Ouders begrijpen nooit iets uit zichzelf en voor kinderen is het vervelend hun altijd weer alles uit te moeten leggen."
Antoine de Saint-Exupéry

Het schrijven van een doctoraal proefschrift is een voorlopig eindpunt. Het is het punt waarop een doctorandus denkt een afgewerkt geheel te kunnen neerpennen over een heel specifiek studie-object. Daarbij hoopt hij een originele, relevante en vernieuwende bijdrage geleverd te hebben met zijn onderzoek en zijn pennenvrucht erover. In hoeverre dit met voorliggend proefschrift het geval is moet de lezer van dit werk in het algemeen, en elk lid van mijn jury in het bijzonder, nu maar eens beoordelen. Na vier jaar onderzoek acht ik er de tijd rijp voor.

Dat ik die vier jaar lang de kans gekregen heb met plezier mijn onderzoek te kunnen verrichten komt niet uit de lucht gevallen. Vele mensen hebben hieraan bijgedragen. Hen wil ik dan ook uitdrukkelijk bedanken.

Allereerst mijn ouders en familie, die me altijd de kans gegeven hebben mijn mogelijkheden tot ontwikkeling te laten komen op de manier die ik zelf verkoos. Die steeds een steun geweest zijn en een referentiepunt, maar nooit een belemmering. Van alle "Waarom ...?"-gezaag van een vijfjarige, over die vele opmerkingen in schoolrapporten à la "het is goed dat Bjorn zo kritisch is, maar hij moet er niet steeds de les mee op stelten zetten" tot de eis dat het stil was in huis tijdens de blokperiodes aan de universiteit. M'n zus Anke wil ik bedanken voor de prettige leef- en werkomgeving waarin ik terechtgekomen ben toen ik met haar ging samenwonen bij de aanvang van het neerschrijven van dit proefschrift. Mijn oom Bart tenslotte voor het heel nauwkeurig nalezen van dit werk.

Daarnaast ook vrienden. Dan denk ik vooral aan de mensen waarbij ik terecht kon als het het meest nodig was. Kristl, Reinhilde en Annelies, ik hou van jullie.

Dat mijn blokperiodes steeds succesvol afgerond werden is mee te danken aan enkele collega-studenten. Met Erik, Peter, Francis en vooral met Mark was het een plezier om samen te werken aan projecten, practica en niet in het minst aan ons afstudeerwerk. Als student met een misschien niet al te voorbeeldige studiemethode werd vooral het ter beschikking stellen van notities op prijs gesteld.

Een aantal lessen konden natuurlijk ook mij uitermate boeien, en niet toevallig van die lesgevers die me er ook toe aangezet hebben om me na mijn studies te vervolmaken als onderzoeker. Daarvoor wil ik prof. Van Campenhout en prof. De Bosschere graag expliciet bedanken. Op onderzoeksgebied beschouw ik ze voor een stuk als ouders: altijd een steun en referentiepunt, nooit een belemmering.

Ouders met een druk leven wel, zodat het goed was dat er ook onderzoeksneven en -nichten waren. En zoals in elke familie kom je met bepaalde kanten van de familie meer in contact dan met andere. In mijn geval was dat vooral met Mark, Michiel, Bart, Lieven, Hans en Ronny. Hans, Michiel en ook prof. De Bosschere wil ik bovendien bedanken voor het zorgvuldig nalezen van dit proefschrift om er de tiken andere fouten uit te halen.

Meer nog dan door neven en nichten werd mijn onderzoeksleven aangenamer gemaakt na de geboorte van een broer: Bruno. Na ongeveer twee jaar als jongeling alleen rondgekropen te hebben, was zijn komst een echte verademing. De mogelijkheid om zowel op meta- als op technisch niveau van gedacht te kunnen wisselen is voor mij van onschatbare waarde gebleken. Ik vond het steeds plezant en nuttig om samen te werken, en ik hoop dat Bruno met mij ervaren heeft dat 1+1 soms meer dan 2 kan zijn. Ook Bruno wil ik bedanken voor het nalezen van dit proefschrift.

Dat ik binnen mijn onderzoeksfamilie kon functioneren is voor een groot deel aan een suikeroom te danken, die ik bij deze niet wil vergeten: het Fonds voor Wetenschappelijk Onderzoek - Vlaanderen. Niet alleen heb ik gedurende 4 jaar gewerkt aan een project dat gefinancierd werd door het FWO, bovendien hebben ze me de mogelijkheid geboden gedurende een drietal maanden verre familie op te zoeken aan de andere kant van de wereld. Mijn studieverblijf in Tucson, bij co-ouder prof. Debray en neefje Scott Waterson was zowel op professioneel als op persoonlijk vlak de meest boeiende periode uit mijn leven, die ik voor geen goud had willen missen. Met name het gezin Debray (Saumya, Wendy, Arun, Reena en Kaloo†) ben ik heel dankbaar voor hun

hartelijke ontvangst en gastvrijheid. *Dear Saumya, Wendy, Arun, Reena and Kaloo. I would like to thank you very much for the warm welcome during my stay in Tucson. This stay was without any doubt the most exciting period in my professional and personal life.* Ook aan de samenwerking met Robert Muth, die de familie al een beetje ontgroeid was bij mijn verblijf daar, houd ik goede herinneringen over.

Genoeg over genealogie gepraat nu, terug naar dit doctoraal proefschrift. Dit proefschrift is allereerst opgevat als een weergave van de door mijzelf opgedane kennis en inzichten, die hopelijk een originele, vernieuwende en relevante bijdrage vormen aan de kennis over computersystemen in het algemeen en de generatie van compacte programma's in het bijzonder. Het onderzoek dat gepaard gaat met het opdoen van deze kennis is per definitie beperkt. Ik heb dan ook niet de pretentie op alle vragen in het domein van programmacompactie een antwoord te kunnen geven.

Dit proefschrift is evenmin geschreven als een opeenvolging van tot in de kleinste details beargumenteerde keuzes, algoritmes en resultaten. De betrachting is eerder om een gevoel, een ervaring mee te geven aan de lezer: wat zijn de mogelijkheden, waar moet je rekening mee houden, etc. Daarbij komen zowel meer theoretische overpeinzingen aan bod als implementie-aspecten. Het beoogde doctoraat is immers een doctoraat in de Toegepaste Wetenschappen.

Ik hoop stellig dat u als lezer er iets aan heeft mijn werk te lezen. En dat u, na het als leescommissaris grondig bestudeerd te hebben, het werk als een verrijking beschouwt en niet louter als een administratieve formaliteit.

Veel leesgenot!

Bjorn De Sutter
Gent, december 2001.

Inhoud

Tabellen

Figuren

Woordenlijst

"Om iets te zijn moeten wij Vlamingen zijn.
Wij willen Vlamingen zijn, om Europeeërs te worden."
August Vermeylen, 1900

Op 5 april 1930 werd, na een jarenlange emancipatiestrijd, de taalwet gestemd betreffende de vernederlandsing van de Gentse universiteit. In de woelige zestiger jaren werd één van de laatste Franstalige bastions in Vlaanderen aangepakt, met de vernederlandsing van de Leuvense universiteit. 2001 is het Europese jaar van de talen. Meer en meer komt er kritiek op de op maat van multinationale ondernemingen gesneden globalisering, die op allerlei manieren authentieke culturen onder de mat tracht te schuiven.

En toch komt het gebruik van het Nederlands aan de Vlaamse universiteiten meer en meer onder druk te staan. Het recente Bolognaakkoord lijkt de trend van "functionele verengelsing" zelfs nog te versnellen.

De auteur van dit proefschrift verzet zich sterk tegen het plat op de buik gaan voor de mercantiele lokroep van de economische eenheidsworst. Elke taal is immers een andere bril waardoor iemand de wereld rondom zich kan bekijken, elke taal biedt nieuwe mogelijkheden aan mensen om zich genuanceerd uit te drukken. Meertaligheid kan echter alleen vertrekken vanuit een uitmuntende kennis van de eigen taal. Zonder deze kennis is kruisbestuiving immers uitgesloten.

Vanuit deze visie hebben we in dit proefschrift steeds een Nederlandstalige terminologie gehanteerd. In afwachting van een meer algemene ingang van een Nederlandstalig vakjargon gaat dit al eens gepaard met improvisatie, wat voor een aantal lezers de duidelijkheid en de vlotte leesbaarheid van dit proefschrift misschien niet steeds ten

goede komt. Wij durven deze lezers dan ook aanraden om zich een Nederlandstalige terminologie eigen te maken. Daartoe kan de vertalende en verklarende woordenlijst in dit hoofdstuk misschien een eerste aanzet zijn.

Deze lijst is tot stand gekomen na een jarenlang collegiaal en door professoren Van Campenhout en De Bosschere actief gestimuleerd gebruik van Nederlandstalige terminologie binnen onze onderzoeksgroep. Bij deze wensen we alle steun voor de gevoerde taalpolitiek uit te drukken, evenals onze dank voor het samen improviseren over ons vakjargon!

Engelse term	Nederlandse term
array	rij
backend	achterkant
bottom	bodem
cache	tussengeheugen
callee-saved	achteraf te bewaren
caller-saved	vooraf te bewaren
calling standard	oproepconventie
compiler	vertaler
control flow	controleverloop
data flow	dataverloop
data member	datalid
debuggen	ontluizen
default value	verstekwaarde
delay slot	vertragingssleuf
destination	doel
displacement	afstand
fall-through path	doorvalpad
flow insensitive	verloopongevoelig
flow sensitive	verloopgevoelig
garbage collection	geheugensanering
greedy	gulzig
hardware	apparatuur
hash table	hakseltabel
header	kop
heap	grabbelgeheugen
inheritance	overerving
inlining	proceduresubstitutie
interpreter	vertolker
kernel	kern
offset	afstand
overhead	onkosten
padding	vulling
to parse	ontleden
pointer	wijzer
program slice	programmasnede
programmable logic array	programmeerbare matrix van logische eenheden
reentrant	herbetreedbaar
(to) schedule	inroosteren, inroostering

single assignment	unieke toekenning
software	programmatuur
spill code	overloopcode
splash screen	voorpagina
stack frame	stapelvenster
static single assignment (SSA)	statisch-unieke-toewijzing (SUT)
string	tekenrij
stub	stompje
swap file	wisselbestand
target	bestemming
template	sjabloon
top	top
type checking	typetoetsing
useless code	loze code
variable	veranderlijke
volatile	vluchtig
wafer	plak
window manager	vensterbeheerder
wire code	draadcode
wrapper function	inpakfunctie

Nederlandse term	Engelse term
achteraf te bewaren	callee-saved
achterkant	backend
afstand	offset, displacement
apparatuur	hardware
bestemming	target
bodem	bottom
controleverloop	control flow
datalid	data member
dataverloop	data flow
doel	destination
doorvalpad	fall-through path
draadcode	wire code
geheugensanering	garbage collection
grabbelgeheugen	heap
gulzig	greedy
hakseltabel	hashtable
herbetreedbaar	reentrant
inpakfunctie	wrapper function
inroosteren, inroostering	(to) schedule
kern	kernel
kop	header
loze code	useless code
onkosten	overhead
ontleden	to parse
ontluizen	debuggen
oproepconventie	calling standard
overerving	inheritance
overloopcode	spill code
proceduresubstitutie	inlining
programmeerbare matrix van logische eenheden	programmable logic array
plak	wafer
programmasnede	program slice
programmatuur	software
rij	array
sjabloon	template
stapelvenster	stack frame
statisch-unieke-toewijzing (SUT)	static single assignment (SSA)
stompje	stub

tekenrij	string
top	top
tussengeheugen	cache
typetoetsing	type checking
vensterbeheerder	window manager
veranderlijke	variable
verloopgevoelig	flow sensitive
verloopongevoelig	flow insensitive
verstekwaarde	default value
vertaler	compiler
vertolker	interpreter
vertragingssleuf	delay slot
vluchtig	volatile
vooraf te bewaren	caller-saved
voorpagina	splash screen
vulling	padding
unieke toekenning	single assignment
wijzer	pointer
wisselbestand	swap file

Hoofdstuk 1

Inleiding

"Schat, ik heb ruimte nodig!"
"Dan moet je maar astronaut worden."

In deze inleiding wordt ingegaan op het "waarom" van dit proefschrift en het eraan voorafgaand onderzoek. We kaderen het onderzoek, maken keuzes en trachten die te verantwoorden.

1.1 Kleiner is beter

In moderne computers die gestoeld zijn op de Von Neumann-architectuur worden programma's als data opgeslagen in het geheugen. Met programma bedoelen we hier zowel de code van het programma als de data waarop deze code berekeningen uitvoert.

Voor de meeste programma's wordt de code vastgelegd tijdens het vertalen, linken en eventueel het laden van het programma (als er dynamische gelinkte bibliotheken gebruikt worden of in het geval van JIT (Just-In-Time) compilatie zoals voor Java programma's).

De data van een programma kan grofweg in twee klassen onderverdeeld worden: de statisch gealloceerde data en de dynamisch gealloceerde data. De statisch gealloceerde data wordt eveneens vastgelegd tijdens het vertalen, linken en laden van een programma. De dynamisch gealloceerde data daarentegen wordt pas aangemaakt tijdens de uitvoering van het programma. Dit is vooral de data op de stapel en in het grabbelgeheugen.

Algemeen kan men stellen dat hoe meer functionaliteit een programma aanbiedt, hoe groter het programma zal zijn, zowel qua code als qua data. Dit is niet meer dan logisch. Deze vaststelling moet gekaderd worden in een aantal trends:

1. Consumenten verwachten steeds meer functionaliteit van de programma's die zij gebruiken, ook al hebben ze geen besef van het feit dat ze een programma gebruiken, zoals bv. bij het gebruik van een gsm of een wasmachine.

2. Consumenten verwachten een steeds groeiend gebruiksgemak van de computersystemen waarmee ze werken. Liefst merken ze eigenlijk zelfs niet dat het om een computer gaat. Onder gebruiksgemak wordt bv. de autonomie van draagbare toestellen verstaan, d.w.z. de tijd dat het toestel kan werken zonder dat de batterij opnieuw moet opgeladen worden. Maar ook fysieke kenmerken als gewicht, afmetingen en geproduceerde warmte spelen een rol.

3. Het moet natuurlijk allemaal goedkoop. Dit houdt in dat de producent de apparatuur goedkoop moet kunnen vervaardigen en dat de ontwikkeling van product en programmatuur snel en eenvoudig moet zijn.

Deze vaststellingen zorgen ervoor dat de producenten van consumentgerichte computersystemen met tegenstrijdige vereisten geconfronteerd worden:

- De programma's worden complexer en gebruiken dus meer geheugen. Deze toename in geheugenverbruik is niet enkel aan de toegenomen functionaliteit te wijten, maar eveneens aan het gebruik van allerlei ingenieurstechnieken zoals componentgebaseerd ontwerp en het gebruik van programmabibliotheken. De gebruikte bibliotheken en componenten zijn los van een specifieke applicatie ontwikkeld met het oog op herbruikbaarheid en algemene toepasbaarheid. Zij zijn dus niet specifiek voor een applicatie geschreven en bevatten vaak heel wat functionaliteit die voor een specifieke applicatie nutteloos is. Het proces waarmee programma's samengesteld worden (zie verder in dit hoofdstuk en in hoofdstuk 2) uit hun lossere onderdelen (bronbestanden en bibliotheken), is tot op vandaag vaak niet verfijnd genoeg om

deze voor een applicatie nutteloze functionaliteit weg te filteren bij het samenstellen van het uiteindelijke programma.

- Het fysiek geheugen van de systemen moet zo klein mogelijk gehouden worden, omdat dit het vermogenverbruik laag houdt, en daarmee de bedrijfstemperatuur, en omdat dit voor mobiele systemen de autonomie verlengt. Bovendien kunnen kleinere geheugens sneller in heel-systeem-op-een-chip oplossingen verwerkt worden, wat de productiekosten, het gewicht en de afmetingen ten goede komt.

  Bij moderne ingebedde processors gaat een groot gedeelte van de chipoppervlakte naar geheugen. Als dit geheugen kleiner kan gemaakt worden, verkleint de oppervlakte van de chips. Er kunnen dan meer chips op een siliciumplak geproduceerd worden, wat de opbrengst per plak gevoelig de hoogte injaagt.

Men kan dus stellen dat van programma's verlangd wordt dat ze meer en meer functionaliteit aanbieden en dat ze zo klein mogelijk zijn.

Overigens moet opgemerkt worden dat deze vragen ook meer en meer gesteld worden door producenten van kantoorapplicaties als tekstverwerkers en rekenbladen. Voor deze applicaties is een van de belangrijke criteria de snelheid waarmee de applicatie kan opgestart worden. De gebruiker wil meteen na het aanklikken van een icoontje kunnen beginnen werken. Om alvast de perceptie van het (snel) opstarten van een applicatie te verbeteren zijn de voorpagina's uitgevonden: venstertjes die de psychologisch belangrijke boodschap meegeven dat een applicatie wel degelijk aan het opstarten is. Het moge duidelijk zijn dat de opstartsnelheid afhangt van de grootte van de applicaties. Dit heeft niet alleen te maken met het lezen van de applicatie van schijf, maar ook bv. met het aantal uit te voeren relocaties tijdens het inladen.

Daarnaast is het zo dat de producenten van kantoorapplicaties beginnen in te zien dat consumenten minder en minder geneigd zijn steeds de nieuwste apparatuur aan te schaffen. Dit kan grotendeels verklaard worden door de steeds kortere cyclus waarmee nieuwe, snellere, goedkopere en betere producten op de markt komen. Als de producenten hun applicaties ongebreideld meer en meer geheugen laten gebruiken, sluiten zij zichzelf uit voor een belangrijk deel van de markt. Deze economisch belangrijke vaststelling ligt mee aan de basis van de groeiende interesse die bv. Microsoft heeft in het automatisch kleiner maken van programma's [Deb].

Een extra, even belangrijke reden tenslotte waarom men programma's zo klein mogelijk wil maken is die van de opkomende netwerkcultuur. Meer en meer worden applicaties over netwerken verzonden. Kleinere applicaties nemen minder bandbreedte in bij het verzenden en zijn dus te verkiezen boven grotere applicaties.

1.2 Optimalisatie van hele programma's

Om de door ons onderzochte paden om de tegenstrijdige vragen te beantwoorden te kaderen, is het nuttig kort te beschouwen hoe een programma tot stand komt. De manier waarop een programma opgebouwd wordt is weergegeven in figuur 1.1.

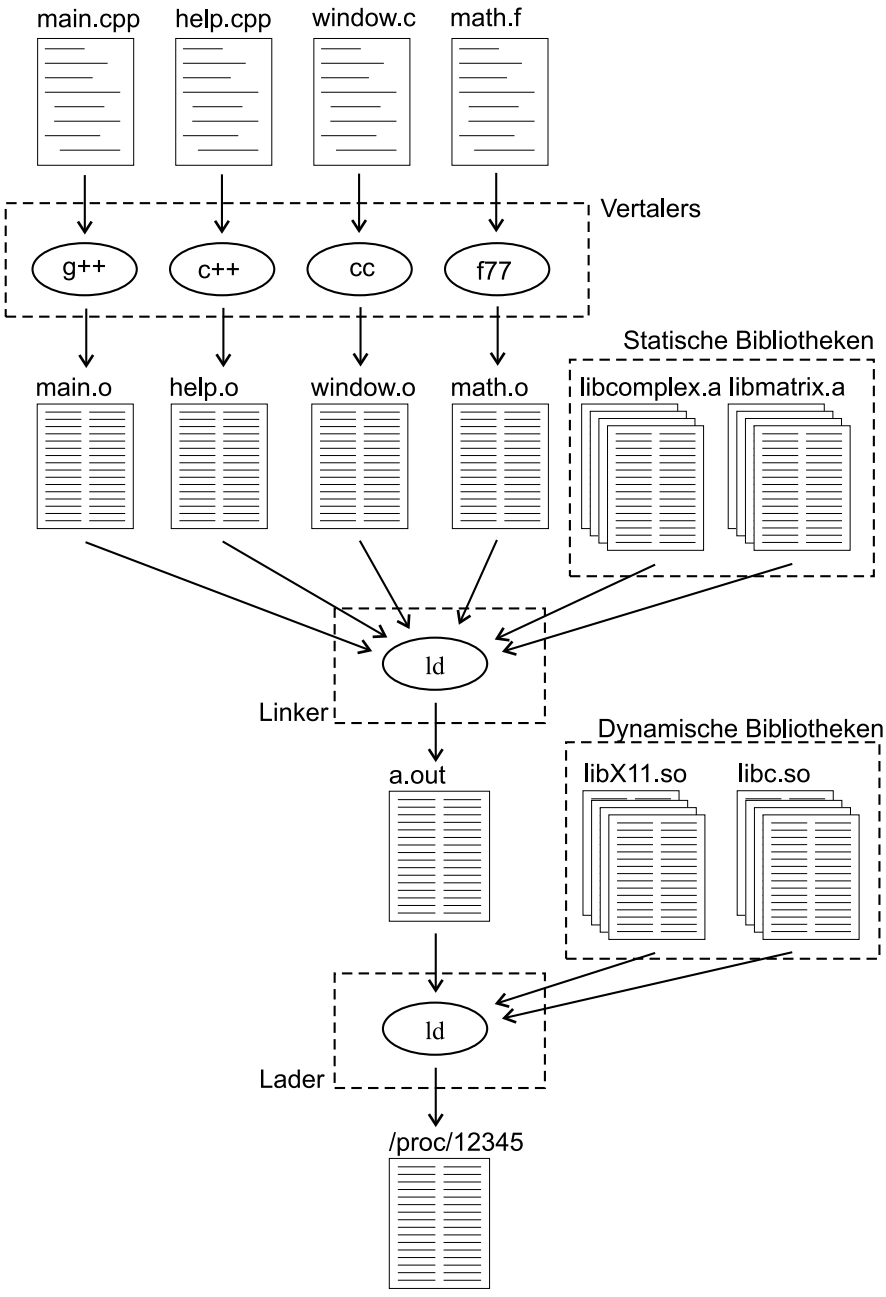
Vertrekkend vanuit bronbestanden en statische bibliotheken wordt een programma gegenereerd door vertalers en een linker. Dit programma `a.out` wordt op schijf opgeslagen. Om het programma uit te voeren wordt het door een lader ingeladen in het geheugen en er eventueel gekoppeld aan dynamische bibliotheken. Het is dan een proces geworden. In hoofdstuk 2 wordt dieper ingegaan op een aantal facetten van deze totstandkoming van een programma.

In deze schematische voorstelling kan men op tal van plaatsen trachten transformaties toe te passen om het programma kleiner te maken. Enkele mogelijkheden die wij niet onderzocht hebben zijn:

- *Compressie van het programma:*
  Men kan het programma zoals het op schijf staat comprimeren, bv. door programma's als gzip, compress, etc. te gebruiken. Om deze programma's te kunnen uitvoeren moeten zij dan wel opnieuw gedecomprimeerd worden, wat onkosten met zich meebrengt en bovendien evenveel geheugen vereist om het gedecomprimeerde programma op te slaan.

  Men kan het decomprimeren evenwel ook dynamisch uitvoeren, waarbij stukken code en/of data gedecomprimeerd worden wanneer ze uitgevoerd en/of opgeladen worden. Dit kan in programmatuur gebeuren (bv. door te werken met stompjes die de decompressie doen of door voor talen zoals Java de klassebestanden te decomprimeren na verzending over het netwerk) of in apparatuur (bv. op het moment dat code en/of data van het hoofdgeheugen naar de tussengeheugens gebracht worden).

  Wij hebben ervoor geopteerd niet naar technieken te kijken die



Figuur 1.1: De werking van een systeemomgeving.

een decompressie vereisen. Dit is een eerste begrenzing aan ons onderzoek.

- *Een betere extractie van bibliotheken:*
  Men kan trachten het extraheren van de benodigde code uit bibliotheken te verfijnen om minder nutteloze code en data mee te nemen in een applicatie. Men neemt dan zijn toevlucht tot methodes die meer informatie gebruiken dan de referenties naar symbolen, wat vandaag in linkers gebeurt (zie hoofdstuk 2 voor een grondiger bespreking van dit proces). De meeste technieken in de literatuur moeten hiertoe (de interface naar) de bibliotheekcode op een hoger niveau analyseren, bv. dat van de broncode, waar ook type-informatie voorhanden is.

  Wij hebben hier niet voor geopteerd, omdat we in ons onderzoek het volledige schema voor de totstandkoming van programma's onveranderd willen laten. We willen bv. onze technieken niet beperken tot programma's waarvan we de volledige broncode ter beschikking hebben. Dit is vaak niet het geval voor de code in systeembibliotheken. Van sommige gebruikte bibliotheken kent men ook de interface in broncode niet. Als men in een applicatie een bibliotheek gebruikt, is de interface ernaar ter beschikking omdat de applicatie anders nu eenmaal niet kan gecompileerd worden. Als die bibliotheek echter om wat voor reden ook gebruik maakt van subbibliotheken, hoeft de interface naar die subbibliotheken niet gekend te zijn op het moment dat de applicatiespecifieke code gecompileerd wordt.

Dat we deze mogelijkheden om tot kleinere programma's te komen niet onderzocht hebben, betekent geenszins dat we niet in de mogelijkheden ervan zouden geloven. Deze worden overigens kort besproken in de sectie over verwant werk op het eind van dit hoofdstuk. We hebben gewoon andere paden willen bewandelen, waarbij we overigens niet de pretentie hebben om de oplossing die we zelf voorstellen als zaligmakend voor te stellen. Soms is onze keuze ook wat arbitrair, men kan immers niet alles onderzoeken.

Heel algemeen kan men stellen dat onze keuze gevallen is op de mogelijkheden tot compactie die voortkomen uit de aparte vertaling van bronbestanden. Daarmee bedoelen we zowel de aparte vertaling van de applicatiespecifieke bronbestanden, als de aparte vertaling van bibliotheekbestanden.

De applicatiespecifieke bronbestanden worden door vertalers vaak elk apart omgezet in objectbestanden. Sommige vertalers kunnen verschillende bestanden gelijktijdig omzetten en bij de omzetting van het ene bestand rekening houden met de code uit andere bronbestanden. Dit kan leiden tot efficiëntere (snellere, kleinere, enz.) programma's, bv. indien hierdoor het gebruik van oproepconventies kan vermeden worden. In het algemeen spreekt men in zo'n geval van intermodulaire optimalisaties, omdat ze de samenwerking (interface) tussen twee programmadelen in verschillende modules optimaliseren.

Intermodulaire optimalisaties zijn vandaag echter nog steeds een uitzondering. Zo kan men ze in ons model niet toepassen op de interface tussen applicatiespecifieke code en bibliotheekcode, simpelweg omdat de bibliotheekcode niet in broncode of op een ander intermediair niveau voorhanden hoeft te zijn. Een andere reden waarom men intermodulaire optimalisaties vaak niet kan toepassen is omdat het programma in meer dan één taal is geschreven.

Over het algemeen kunnen we dus stellen dat vertaaloptimalisaties geremd worden door de modulegrenzen. Willen we deze grenzen overstijgen en tot een hele-programma-optimalisatie komen, dan moeten we ofwel de volledige code tegelijkertijd beschikbaar hebben in een of andere brontaal of een voorstellingswijze op intermediair niveau, ofwel moeten we de optimalisaties op een andere plek in het totstandkomingsproces van de applicatie toepassen.

Wij hebben geopteerd om de tweede mogelijkheid uit te pluizen. We zullen daartoe de volledige code en data in een programma tegelijkertijd beschouwen op het moment dat het programma gelinkt is en trachten het programma te compacteren, d.w.z. kleiner te maken zonder het gedrag van het programma te veranderen. We kunnen er dan van uitgaan dat we alle code en data in het programma ter beschikking hebben.

Voor dynamische gelinkte bibliotheken is dit laatste niet het geval: de context waarin ze uitgevoerd worden is niet op voorhand bekend. Hun reden van bestaan is immers dat ze in verscheidene programma's gebruikt kunnen worden (terwijl ze slechts eenmaal in het geheugen moeten geladen worden). Bij de compactie van één programma willen wij er voorlopig niet van uitgaan dat we alle applicaties kennen die van een dynamische gelinkte bibliotheek zullen gebruikmaken. We beschouwen dus (voorlopig) geen dynamisch gelinkte bibliotheken en zullen alle programma's waarop de door ons ontwikkelde technieken

geëvalueerd worden statisch linken. Dit beperkt ons inziens de algemene toepasbaarheid van deze technieken niet, aangezien ze op een vrij eenvoudige wijze kunnen uitgebreid worden naar dynamisch gelinkte programma's.

Over de programma's die we willen compacteren maken we samenvattend volgende aannames:

- Alle code en statisch gealloceerde data is ter beschikking.
- We beschouwen dus geen dynamisch gelinkte programma's.
- Alle informatie waarover de linker moet kunnen beschikken zoals relocatie- en symboolinformatie hebben ook wij ter beschikking.
- De programma's bevatten geen berekeningen op code-adressen.
- De programma's werken niet met vluchtige data.
- De programma's zijn serieel, het betreft m.a.w. geen parallelle of meerdradige programma's.
- De programma's gebruiken geen excepties zoals aangeboden door het besturingssysteem van ons platform.

Deze aannames worden verder doorheen het proefschrift uitgewerkt en geduid.

1.3 Overzicht van dit proefschrift

In het volgende hoofdstuk worden de gevolgen van de keuze om een heel programma te compacteren na het linken besproken. Aan de hand van een aantal daarmee gepaard gaande vaststellingen wordt besproken welke *informatie* voorhanden is, en hoe we aan de hand van die beperkte informatie een *interne voorstelling* van de te compacteren programma's kunnen opbouwen. Deze voorstelling bevat aanvankelijk enorm veel onbekenden, en een belangrijk onderdeel van dit proefschrift is dan ook het wegwerken van de onbekenden, m.a.w. het verfijnen van de interne voorstelling van een te compacteren programma. Ook dit komt in het volgende hoofdstuk uitvoerig aan bod.

In hoofdstuk 3 wordt in detail ingegaan op de drie volgens ons meest belangrijke analyses waarrond de compactie door hele-programma-optimalisatie gebouwd is. Het zijn *levensduuranalyse*, *constantenpropagatie* en *detectie van dode data* en de ermee corresponderende *onbereikbare code*. Deze analyses worden uitvoerig behandeld omdat ze (1) heel belangrijk zijn; (2) ze complex zijn en de implementatie ervan dus uitgekiend moet gebeuren en (3) de auteur van dit proefschrift er een belangrijke bijdrage in heeft geleverd. Dit laatste is met name het geval voor de constantenpropagatie en de detectie van dode data. Tevens worden de codetransformaties besproken die heel nauw verband houden met deze optimalisaties.

Op het einde van dat hoofdstuk wordt ook kort ingegaan op een aantal andere optimalisaties die ter compactie uitgevoerd worden. Deze worden slechts korter behandeld omdat ze van minder belang zijn voor de eindresultaten en omdat de auteur er ook geen noemenswaardige nieuwe bijdrage in geleverd heeft. Voor de volledigheid van dit werk worden ze evenwel kort aangeraakt.

Waar we het tot dan toe steeds over de agressieve toepassing van analyses en optimalisaties op een heel programma gehad hebben, vatten we in hoofdstuk 4 een heel ander, tevens heel belangrijk, facet van programmacompactie aan: *factorisatie*. Voor verschillende niveaus van granulariteit wordt besproken hoe men kan vermijden dat identieke of gelijkaardige codefragmenten een programma nodeloos groot maken. De belangrijkste bijdragen van de auteur op dit vlak liggen in de factorisatie van volledige procedures en in het hergebruiken van data. Voor de volledigheid wordt ook factorisatie op andere granulariteiten besproken.

In hoofdstukken 2, 3 en 4 zullen af en toe kwalitatieve uitspraken gedaan worden omtrent de efficiëntie en de effectiviteit van de besproken analyses. Een echt empirische evaluatie van de algoritmen wordt er nog niet in gegeven. De reden hiervoor is dat de analyses, optimalisaties en verfijningen van de interne voorstelling zodanig met elkaar verweven zijn, dat het ons inziens vrij nutteloos is de besproken algoritmen kwantitatief te evalueren alvorens men een overzicht heeft gekregen van het geheel. In hoofdstuk 5 wordt daarom eerst ruime aandacht besteed aan het samenbrengen van alle stukjes van de puzzel, waarna we overgaan tot een brede empirische en kwantitatieve evaluatie van de besproken technieken. Daarbij kunnen we nu reeds opmerken dat we zullen trachten de prestaties van de analyses en transformaties naar

ware waarde te schatten: hun rekentijd, geheugenvereisten, invloed op de compactiefactor en op de uitvoeringssnelheid van gecompacteerde programma's spelen daarbij een rol. Kunstmatige criteria worden daarbij niet beschouwd. Een typisch voorbeeld van wat wij als zo'n kunstmatig criterium beschouwen is het aantal aliassen dat een aliasanalyse detecteert of het aantal gevallen waarin die analyses bewijzen dat er zeker geen alias is. Zulke aantallen hebben in wezen geen enkel nut. Het is enkel als men met de resultaten van zulke analyses snellere, kleinere of minder vermogen verbruikende programma's kan genereren, dat de analyses beter worden. We zullen dan ook enkel zulke criteria gebruiken.

Tot slot worden conclusies getrokken en mogelijke richtingen aangegeven voor verder onderzoek.

1.4 Verwant werk

Er zijn verschillende onderzoeksdomeinen waar het in dit proefschrift besproken werk in thuis te brengen valt. Deze zijn met name de zoektocht naar kleinere programma's, algemene programmatransformaties na het linken en transformaties op hele programma's.

1.4.1 Kleinere programma's

De meeste vertaaloptimalisaties hebben naast het versnellen van de gegenereerde programma's als neveneffect dat ze verkleinen. We verwijzen voor een algemene bespreking van optimalisaties naar het standaardwerk bij uitstek [Aho86] en het uitstekende en diepgaande boek van Muchnick [Much97]. Uitzonderingen zijn optimalisaties zoals proceduresubstitutie en het uitvouwen van lussen. Dit zijn voorbeelden van technieken die door het genereren van contextspecifieke instanties van stukken broncode deze code versnellen. Omdat er per context een kopie van de code gegenereerd wordt, leiden ze natuurlijk tot grotere programma's.

Naast vertaaloptimalisaties zijn er een aantal andere manieren bedacht om kleinere programma's te verkrijgen. Het begrip "kleiner" kan daarin allerlei betekenissen hebben: kleiner dynamisch geheugenverbruik, kleinere statische voorstellingswijze, etc. We kunnen de in de literatuur beschreven technieken min of meer onderverdelen in categorieën. Laat het evenwel duidelijk zijn dat de scheidingslijn tussen deze

categorieën niet altijd even eenduidig is.

Voor we de verschillende categorieën aanvatten is het nuttig de later veel terugkomende term compressie- of compactiefactor te definiëren. Het is de verhouding tussen het gecomprimeerde of gecompacteerde programma, naargelang de toegepaste techniek, en het originele programma:

$$\text{compressiefactor} = \frac{\text{Grootte gecomprimeerde programma}}{\text{Grootte originele programma}}$$

Hoe kleiner deze factor, hoe groter dus de bereikte compressie. Om niet steeds deze ietwat logge benaming te moeten gebruiken zullen we vaak schrijven dat een programma X maal kleiner geworden is, of dat er een compressie met een factor X bereikt is. In zulke gevallen geldt:

$$\frac{1}{X} = \text{compressiefactor}$$

Met nog een andere formulering kunnen we zeggen dat een programma $Y\%$ kleiner geworden is of we een compactie met $Y\%$ bekomen hebben. Hierbij geldt:

$$Y = 100(1 - \text{compressiefactor})$$

Instructiesets en vertolking

De grootte van programma's op assemblerniveau hangt natuurlijk af van de instructieset die gebruikt zal worden. Men kan dus processors ontwikkelen met instructiesets die gemiddeld tot kleinere programma's aanleiding geven en gebruikt kunnen worden voor een grote groep programma's.

Omdat verschillende programma's op een andere manier gebruik maken van aangeboden instructiesets kan het nuttig zijn nog verder te gaan en per programma een aangepaste instructieset te gebruiken.

Compacte algemene instructiesets Oudere CISC architecturen met instructies met variabele lengte, zoals de welbekende x86 architectuur [Int99] zijn ten dele ontworpen omwille van een hoge codedensiteit. Codedensiteit betekent het gemiddelde aantal instructies (of nauwkeuriger: operaties) dat per byte kan gecodeerd worden. In zulke instructiesets bestaan er bv. verschillende mogelijkheden om letterlijke

operandi te coderen in instructies, waarbij men minder bytes nodig heeft om kleinere waarden voor te stellen.

Het is algemeen bekend dat RISC architecturen programma's sneller kunnen uitvoeren [Henn90]. De vaste breedte van instructies en het beperktere aantal instructies in zulke instructiesets leidt over het algemeen tot grotere programma's. Meer recentelijk werd onderzoek verricht door Bunda e.a. [Bund92] naar het gebruik van 16-bits instructies i.p.v. 32-bits instructies voor de DLX-instructieset. Omdat men in 16 bits minder mogelijke instructies kan coderen dan in 32 bits, zijn er gemiddeld genomen wel meer instructies nodig als men 16-bits instructies gebruikt. Zij rapporteren dat het snelheidsverlies door de toename van het aantal uit te voeren instructies vaak werd gecompenseerd door het hogere aantal instructies dat door de processor kon opgehaald worden, gegeven een bepaalde ophaalbandbreedte.

Voor ingebedde systemen met kleinere geheugens worden sinds halfweg de negentiger jaren processors op de markt gebracht die zulke 16-bits instructiesets implementeren. Programma's die vertaald worden naar de ARM Thumb architectuur [Turl95, ARM95] zijn gemiddeld tot 30% kleiner dan programma's vertaald naar de standaard ARM instructieset. Ze vereisen gemiddeld 15-20% meer uitvoeringstijd. De Thumb architectuur wordt hierbij aangeboden bovenop de standaardarchitectuur, samen met speciale instructies die de processor van de ene instructieset naar de andere doen omschakelen. Men kan dus voor elk stukje code kiezen voor de meeste efficiënte instructieset.

De MIPS16 [Kiss97] architectuur is een analoge uitbreiding voor de standaard MIPS architectuur. Nog volgens [Kiss97] zijn programma's voor deze architectuur 40% kleiner dan wanneer ze vertaald worden voor de standaard MIPS architectuur.

Applicatiespecifieke instructiesets en vertolking In theorie kan men voor elk programma trachten een aparte instructieset te genereren die de kleinst mogelijke voorstelling van het programma mogelijk maakt. Het probleem is natuurlijk dat men over het algemeen geen instructieset zal implementeren op een processor als die instructieset maar door één programma zal gebruikt worden. Als het gaat om een instructieset die daarentegen vertolkt zal worden, dan zijn er wel mogelijkheden.

Het meest bekende voorbeeld van vertolking is wellicht de Java Virtuele Machine of JVM [Lind99]. Deze vertolkt Java bytecode. Clausen e.a. [Clau00] onderzochten hoe men de JVM kan uitbreiden zodat ze

macrobytecodes aankan. Een programma in de uitgebreide bytecode bestaat dan uit de beschrijving van de macrocodes die specifiek voor het programma zijn en de uitgebreide bytecode zelf, waarin de macro's veel voorkomende sequenties vervangen. Hiermee rapporteren ze een reductie van de codegrootte met gemiddeld 15%.

Ernst e.a. [Erns97] vervangen eveneens vaak voorkomende instructiesequenties door macro-instructies. Zij doen het voor de instructieset van de Omniware Virtuele Machine.

Om programma's geschreven in de C taal naar compacte code te vertalen gebruiken Fraser en Proebsting [Proe95, Fras95] volgend schema: het naar een architectuur X te vertalen programma wordt vertaald naar een architectuur Y en er wordt een vertolker aangemaakt die de taal Y kan vertolken. Die vertolker wordt gegenereerd in de C taal en kan dus met elke standaard C vertaler voor de architectuur X vertaald worden. Als het programma in taal Y en de vertaalde vertolker samen minder plaats innemen dan de rechtstreekse vertaling van naar X, maakt men winst. De taal Y wordt daarom zo gekozen dat er (1) een kleine vertolker voor geschreven kan worden; (2) er een compacte vertaling van het originele programma in mogelijk is. Daartoe gaan ze in een intermediaire representatie op zoek naar gelijkaardige structuren: veel voorkomende structuren krijgen een instructie in instructieset Y. Met dit schema rapporteren ze een compressie met ongeveer 50%.

Hoogerbrugge e.a. [Hoog99] gebruiken een soortgelijke techniek om code voor de TriMedia architectuur [Tri00] van Philips te comprimeren. Enkel de minder vaak uitgevoerde code wordt in een specifieke instructieset vertaald en dan vertolkt tijdens de uitvoering van het programma, om zo geen al te groot snelheidsverlies te boeken. Om meer ruimte te sparen maken ze gebruik van een stapelarchitectuur, waarvoor in de vertolkte instructies minder operandi gecodeerd moeten worden. In tegenstelling tot [Fras95] wordt de specifieke instructieset opgebouwd na een training aan de hand van een verzameling trainingsapplicaties. Eigenlijk is deze dus niet applicatiespecifiek. Met deze techniek werd een compressie met ongeveer een factor 5 behaald, terwijl de programma's gemiddeld ongeveer 8 keer trager uitvoeren.

Aan de hand van herprogrammeerbare logica slagen Larin en Conte [Lari99] er wel in om applicatiespecifieke instructiesets te gebruiken en die te laten uitvoeren in apparatuur. Hun programma's bestaan enerzijds uit een programma in een specifieke instructieset en anderzijds uit de programmatie van een programmeerbare matrix van

logische eenheden die de specifieke instructieset kan uitvoeren.

Programmacompressie

Compressie van data in het algemeen is een onderzoeksdomein op zich, dat buiten het bestek van dit proefschrift valt. Voor een uitstekende algemene uiteenzetting van de principes van datacompressie verwijzen we naar [Hank97].

Programma's kunnen als een specifiek soort data beschouwd worden. De compressie ervan verschilt echter op een aantal belangrijke punten met de meest gangbare compressie van bv. digitale beelden, film of spraak. In dat soort toepassingen worden de gegevens meestal als een stroom beschouwd, waar de decompressor één keer in sequentiële volgorde overgaat.

Als we het benodigde geheugen voor de uitvoering van een programma willen verkleinen, kunnen we het programma natuurlijk niet in één stap volledig decomprimeren. We zullen integendeel steeds slechts een beperkt gedeelte van het programma in gedecomprimeerde voorstelling bewaren voor uitvoering. De decompressie van een programma gebeurt m.a.w. incrementeel: stukken code, op allerlei niveaus van granulariteit, worden gedecomprimeerd op het moment dat ze waarschijnlijk uitgevoerd zullen worden. Een gevolg is dat men op vele verschillende plaatsen in een programma moet kunnen beginnen decomprimeren. Om dit efficiënt te kunnen doen worden er vaak alignementsvoorwaarden aan de gecomprimeerde voorstelling van het programma opgelegd. Zowel de eisen van incrementeel decomprimeren en de alignementsvoorwaarde zijn heel specifiek voor programmacompressie en komen in algemene datacompressie minder voor.

Men heeft dan ook specifieke technieken ontwikkeld om programma's te comprimeren en vooral efficiënt te decomprimeren. Het laatste gebeurt zowel in programmatuur als in apparatuur.

Decompressie in apparatuur De TriMedia [Tri00] architectuur is een zogenaamde VLIW architectuur, waarin elke instructie vijf operaties bevat, die allemaal voorwaardelijk uitgevoerd kunnen worden. De instructies hebben een vaste grootte. Aangezien men vaak niet de vijf operaties in een instructie kan benutten, gaat er veel ruimte verloren, o.a. in het tussengeheugen voor instructies. In dit tussengeheugen wor-

den de instructies daarom gecomprimeerd opgeslagen. Als ze in de processor worden opgeladen voor uitvoering worden ze gedecomprimeerd.

Om de bereikte compressiefactor op te drijven wordt er gecomprimeerd per uitgebreid basisblok: dit is een sequentieel stuk code met een unieke ingang, maar eventueel met verscheidene uitgangen. De (de)compressie van zo'n blok ineens is natuurlijk efficiënter dan bv. een instructiegewijze (de)compressie. Daarbij stelt zich natuurlijk een probleem: waar begint er zo'n blok in de code? Op die punten moet de decompressie immers van nul af aan beginnen, m.a.w. zonder voorgeschiedenis. Om dit op te kunnen vangen bestaan er op de TriMedia geen doorvalpaden: het begin van een basisblok kan enkel bereikt worden via expliciete controletransfers. Die geven dan meteen het sein dat een nieuw blok moet gedecomprimeerd worden. Omdat men toch al met uitgebreide basisblokken werkt, waarin bovendien alle instructies voorwaardelijk kunnen uitgevoerd worden, en men heel veel vertragingssleuven heeft (15 operaties) zijn er nauwelijks onkosten verbonden aan het expliciet moeten toevoegen van een vrij beperkt aantal extra controletransfers.

Bij de Compressed Code Risc Processor (CCRP) [Wolf92, Kozu94] vindt de decompressie plaats bij het invullen van het tussengeheugen voor instructies. De gebruikte compressietechniek is Huffmancodering [Huff52], een wijd verbreide techniek uit de compressiewereld. De processor merkt hier niks van, en adressen in het programma verwijzen naar adressen in het tussengeheugen. Omdat deze kunnen verschillen van adressen in het hoofdgeheugen moet er bij het ophalen van instructies een adresvertaling uitgevoerd worden. Daartoe wordt gebruik gemaakt van een zogenaamde "Line Address Table", die het verband tussen beide stockeert. Voor de MIPS instructieset rapporteren de auteurs een compressiefactor van 0.73. In [Bene98, Bene97] wordt de implementatie van een werkend demonstratiemodel van deze processor beschreven. Codepack [Kemp98, Game98, IBM98, Lefu00] is een analoog compressiesysteem in apparatuur voor de PowerPC architectuur. Men bereikt er een compressiefactor van 0.60 mee, waarbij de uitvoering van programma's gemiddeld met 10% vertraagt.

Decompressie in programmatuur Debray e.a. [Debr01a] en Kirovski e.a. [Kiro97] decomprimeren met behulp van extra programmatuur een stukje code als het uitgevoerd wordt en plaatsen de gedecomprimeerde

code in een proceduregeheugen, van waaruit het uitgevoerd wordt.

Kirovski e.a. [Kiro97] decomprimeren stukken code met de granulariteit van een procedure. Bij elke procedure-oproep wordt nagegaan of de opgeroepen procedure in het proceduregeheugen gevonden wordt. Als dit niet zo is, wordt de procedure door extra programmatuur gedecomprimeerd en in het proceduregeheugen geplaatst. Dit tussengeheugen kan als aparte RAM geïmplementeerd worden. De auteurs beweren hierbij een compressiefactor van 0.60 te halen voor de SPARC architectuur.

Debray e.a. [Debr01a] bestuderen decompressie van meer algemene stukken code van verschillende granulariteiten. Het belangrijkste verschil met Kirovski is dat het proceduregeheugen bij Debray gewoon een statisch gealloceerd stuk geheugenruimte is, dat deel uitmaakt van het uitgevoerde programma. Zijn compressie/decompressieschema vereist dus geen enkele aanpassing aan de bestaande apparatuur waarop het geïmplementeerd werd. Daarnaast is het zo dat Debray de compressie beperkt tot code die zelden of nooit uitgevoerd wordt, zodat de met decompressie gepaard gaande onkosten een minimale invloed hebben op de uitvoeringstijd.

Om zijn systeem te evalueren maakt Debray overigens gebruik van het programmatuurprototype SQUEEZE, de programmacompactor die ook wij gebruiken om de in dit proefschrift besproken technieken te evalueren. Debray vermeldt een extra compressie met 4% bovenop de compactie die met SQUEEZE gehaald wordt, waarbij er gemiddeld ongeveer 15% meer instructies moeten uitgevoerd worden. Kiest men ervoor om nog frequenter uitgevoerde code te comprimeren, dan bereikt men makkelijk 6% compressie. Het aantal uit te voeren instructies is dan echter al verdubbeld. Hierbij moet vermeld worden dat de versie van SQUEEZE die Debray hiervoor gebruikt heeft heel wat ouder was en niet zo geavanceerd als de versie waarin al onze technieken geïmplementeerd zijn. Met name een aantal technieken om onbereikbare code te verwijderen (secties 2.5 en 3.3) zijn niet aanwezig in zijn evaluatieversie. Omdat de winst in compressie die hij haalt precies het grootst is voor code die onbereikbaar is en dus nooit uitgevoerd wordt, zal de behaalde winst verkleinen indien hij zijn systeem toepast bovenop alle in dit proefschrift beschreven technieken.

Draadcodes Draadcodes zijn voorstellingswijzen van programma's die erop gericht zijn programma's of programmabestanden als geheel

te comprimeren, met het oog op bv. het efficiënt verzenden van programma's over een netwerk.

Pugh [Pugh99] bespreekt verschillende methodes om Java klassebestanden of Java archiefbestanden te comprimeren voor verzending. Daarbij gaat de meeste aandacht naar het comprimeren van de symbolische data in deze bestanden, die soms meer plaats innemen dan de bytecodes zelf. Aan zijn werk gingen o.m. de minder sterk comprimerende formaten Jazz [Brad98] en Clazz [Hors98] vooraf.

Ernst e.a. [Erns97] ontwikkelden een draadcode die verderbouwt op de intermediaire representatie van de lcc C vertaler [Fras91], die gebaseerd is op bomen.

Franz en Kistler [Fran94, Fran97] noemen hun gecomprimeerde bestanden "slim binaries" oftewel slanke programma's. Om programma's efficiënter te versturen over netwerken en de verstuurde programma's bovendien platformonafhankelijk te maken versturen ze programma's als gecomprimeerde syntaxbomen. Aan de ontvangerkant moet het ontvangen programma verder vertaald worden, wat dit formaat niet echt geschikt maakt voor ingebedde systemen, ook al bereiken ze er een compressiefactor van 0.33 mee.

Extractie

In plaats van de aanwezige code en data in een programma zo klein mogelijk voor te stellen kan men natuurlijk ook trachten programma's zo klein mogelijk te maken door er zoveel mogelijk onnodige code en data uit te weren. Als men het heeft over het beperken van de code en data die meegelinkt worden vanuit bibliotheken spreekt men over code-extractie.

Grotendeels taalafhankelijke technieken hiertoe zijn ontwikkeld door Agesen en Ungar [Ages94] voor de Self taal. Technieken voor extractie van Java bibliotheken werden o.a. door Tip e.a. [Tip99] ontwikkeld. Hierbij bouwen taalafhankelijke optimalisatietechnieken (bv. ter compactie van de constantensectie in Java) verder op (grotendeels) taalonafhankelijke analyses (zoals bv. voor de opbouw van de oproepgraaf van een programma [Tip00]). Door alle beschikbare code van een programma te analyseren (waaronder de gebruikte datatypes), beperken ze wat er van de algemene uitvoeringsomgeving meegelinkt wordt. Ze bereiken hiermee dat programma's vaak meer dan 50% kleiner worden. Het betreft dan bv. grafische gebruikersbibliotheken die

niet langer meegelinkt worden als het niet nodig is.

Algemenere compactie en factorisatie

Meer algemene vormen van compactie richten zich niet specifiek op bibliotheken, maar trachten code en data in het algemeen te verwijderen uit programma's: uit applicatie- en uit bibliotheekcode.

Sweeney en Tip [Swee98] onderzoeken het elimineren van dode dataleden in C++. Hiermee bereiken ze een compactie van het programma tijdens uitvoering, dus inclusief dynamisch gealloceerd geheugen, met gemiddeld 4.4%.

Ook naar het opbouwen van accurate oproepgrafen wordt veel onderzoek verricht, met het oog op het elimineren van onbereikbare procedures. Meer specifiek voor het opbouwen van de oproepgraaf voor object-geortiënteerde programma's met polymorfe methode-oproepen zijn diverse technieken ontwikkeld [Tip00].

Andere algemene compactietechnieken vallen onder de noemer factorisatie, waarbij men zal trachten het meermaals voorkomen van identieke of nagenoeg identieke instructiesequenties in programma's te vermijden. Een uitgebreide bespreking van de literatuur betreffende factorisatie is terug te vinden in sectie 4.9.

1.4.2 Transformaties op hele programma's

Ook om andere redenen dan compactie van programma's is er onderzoek verricht naar transformaties op hele programma's. Zo is dit bv. gebeurd om de onkosten (qua uitvoeringstijd) te beperken die gepaard gaan met het aanspreken van virtuele methoden in object-georiënteerde talen. Over het algemeen kan men zeggen dat het lokale transformaties betreft, maar dat deze gebaseerd zijn op hele-programma-analyses. Zo analyseren Vortex [Cham96] en mld [Fern96] de klassehiërarchie van volledige Cecile resp. Modula-III programma's. Virtuele methode-oproepen worden door directe oproepen vervangen indien de klassestructuur dit toelaat. De onkosten die met polymorfisme gepaard gaan worden deels opgevangen door op basis van profielinformatie de meest opgeroepen methodes te selecteren en er directe oproepen voor te implementeren.

1.4.3 Transformaties na het linken

Programmatransformaties tijdens of na het linken werden om allerlei redenen onderzocht. De voornaamste redenen zijn binaire vertaling van een architectuur naar een andere, snelheidsoptimalisaties en instrumentatie.

Voor een aantal ingebedde architecturen bestaan ook compacterende linkers. Zo kan de TriMedia [TM] linker in zekere mate onbereikbare code elimineren en identieke codefragmenten hergebruiken door middel van factorisatie.

Binaire vertaling

Als we de term binaire vertaling strikt interpreteren hebben we het over de vertaling van code voor een bepaalde instructieset en uitvoeringsomgeving naar code voor een andere instructieset en uitvoeringsomgeving.

FX!32 [Hook97, Cher98, Dron99] is een systeem dat het mogelijk maakt volledig transparant (d.w.z. zonder dat de gebruiker dit opmerkt) programma's voor de x86 Win32 omgeving uit te voeren op het Alpha/Windows NT platform. Daartoe wordt de x86 code vertolkt en de vaak uitgevoerde code wordt vertaald naar de Alpha architectuur, waarbij ze bovendien geoptimaliseerd wordt.

FreePort Express [Alpa] maakt het mogelijk Solaris/SPARC programma's te draaien op Tru64 Unix/Alpha platformen, terwijl DECmigrate [Alpb, Site92] het mogelijk maakt programma's voor de MIPS of VAX architectuur op Alpha-apparatuur uit te voeren.

Transitives heeft zeer recent een product op de markt gebracht, Dynamite [Dyn] genaamd, dat toelaat om programma's voor enkele ingebedde architecturen binair te vertalen, zodat men code die gegenereerd is voor één bepaald platform op verschillende platformen kan draaien.

UQBT (University of Queensland Binary Translator) [Cifu00] is een algemeen kader om statische binaire vertaling uit te voeren. Het is speciaal gebouwd om vlot overdraagbaar te zijn naar verschillende bronen doelarchitecturen. Uit UQBT is UQDBT (University of Queensland Dynamic Binary Translator) [Ung00] gegroeid, een kader voor dynamische vertaling, d.w.z. tijdens de uitvoering van programma's. Beide systemen werken met een algemene machinespecificatietaal, die het mogelijk maakt op relatief eenvoudige wijze nieuwe architecturen toe

te voegen om als bron- of doelarchitectuur te dienen.

Snelheidsoptimalisatie

Men kan stellen dat het onderzoek naar code-optimalisatie tijdens en na het linken een eerste keer uitvoerig aan bod gekomen is in [Wall86]. Hierin werd door Wall een systeem beschreven waarin de vertaler de linker extra informatie verschaft opdat deze het volledige programma zou kunnen optimaliseren. Meer bepaald wordt informatie meegegeven aan de linker omtrent het verwijderen van overbodige overloopcode. Deze informatie geeft aan welke instructies moeten verwijderd worden als men globaal beslist om een bepaalde veranderlijke niet langer in het geheugen maar in een register te bewaren.

Wall en Srivastava ontwikkelden later OM [Sriv93, Sriv94b], een optimalisator tijdens het linken die invarianten uit lussen haalt en het gebruik van de globale adrestabel optimaliseert voor de Tru64 Unix/Alpha omgeving. Daarnaast wordt onbereikbare code verwijderd, worden procedures gesubstitueerd en wordt de code herordend met het oog op een efficiënter gebruik van de tussengeheugens.

Bij het toenmalige Digital ging men nog een stap verder met Spike [Cohn97, Good97]: deze optimalisator transformeert binaire programma's voor dezelfde omgeving en voor Windows NT/Alpha, waarbij de code herordend wordt om de tussengeheugens en de sprongvoorspelling beter te benutten en waarbij globale registerallocatie toegepast wordt. Spike kan dynamisch gelinkte programma's aan, doordat het zelf op zoek gaat naar de dynamische bibliotheken die door een programma gebruikt worden.

ALTO [Muth01, Muth99] is een snelheidsoptimalisator na het linken voor Tru64 Unix/Alpha programma's. ALTO voert een groot aantal optimalisaties uit die ook in dit werk kort besproken zullen worden. De prototype compactor die we tijdens ons onderzoek ontwikkeld hebben is immers op ALTO gebaseerd. Eén van de belangrijke zaken die niet uitgevoerd worden tijdens compactie maar wel in ALTO is waardeprofilering [Watt01], waarbij stukken code geoptimaliseerd worden voor veel voorkomende waarden van veranderlijken tijdens de uitvoering. Naast snelheidsoptimalisaties kan men met ALTO ook een reductie van het vermogenverbruik van programma's verwachten [Debr01b].

De optimalisators na het linken die we tot nog toe besproken hebben zijn alle statische optimalisators: ze wijzigen het programma vóór

het uitgevoerd wordt. Daarnaast bestaat er sinds kort veel aandacht voor dynamische optimalisatie, waarbij programma's getransformeerd worden tijdens de uitvoering om de uitvoering zo te versnellen. Het reeds besproken FX!32 is hier een voorbeeld van: naast de vertolking van x86 code wordt de vaak uitgevoerde code vertaald naar de Alpha architectuur, waarbij de code verder geoptimaliseerd wordt.

Verreweg de bekendste vorm van snelheidsoptimalisatie tijdens het uitvoeren is JIT (Just-In-Time) vertaling [Yang99, AT98] die gebruikt wordt in Java Virtuele Machines (JVM) [Lind99]. In plaats van keer op keer de frequent uitgevoerde bytecode te vertolken, wordt er een vertaling van gemaakt die rechtstreeks kan uitgevoerd worden op het platform waarop de JVM draait.

Dynamo [Vasa00] leert dat het met het oog op snelheidsoptimalisatie zelfs nuttig kan zijn code te vertolken op de eigen architectuur in plaats van ze gewoon uit te voeren: men heeft de volledige controle over de uitvoering van het programma en kan makkelijk vaststellen welke paden frequent uitgevoerd worden. Door de code op die paden in het programma wel rechtstreeks uit te voeren, na ze geoptimaliseerd te hebben voor die specifieke paden, bereikt men met Dynamo dat programma's die met -O2 vertaald zijn (waarop m.a.w. alle optimalisaties toegepast zijn die het programma niet groter maken) even snel uitgevoerd worden als programma's die met -O4 vertaald werden (waarop m.a.w. alle optimalisaties ten voordele van de uitvoeringssnelheid toegepast werden). Dit gebeurt op transparante wijze voor de gebruiker, die niets van Dynamo zelf merkt.

Instrumentatie

ATOM [Sriv94a] biedt de mogelijkheid om een vertaald programma voor de Tru64 Unix/Alpha architectuur op een eenvoudige en uiterst flexibele manier te instrumenteren. Daarbij wordt er een strikte scheiding gemaakt tussen het eigenlijke toevoegen van de instrumentatiecode en die code zelf. ATOM zelf biedt de gebruiker een interface aan in C die uit twee delen bestaat.

Het eerste deel biedt de programmeur de mogelijkheid in C te specifiëren hoe het programma moet geïnstrumenteerd worden. De programmeur geeft hiermee aan waar het programma moet geïnstrumenteerd worden, bv. aan het begin van basisblokken, bij leesoperaties, etc. Daarbij geeft de programmeur op welke instrumentatieprocedu-

res moeten opgeroepen worden op die plaatsen en welke informatie als argumenten aan die procedures meegegeven wordt. Daarvoor beschikt de programmeur over het tweede deel van de ATOM-interface, die het mogelijk maakt in C allerlei informatie over een programma in uitvoering op te vragen, zoals registerinhouden, opcodes, etc.

De instrumentatieprocedures zelf worden door de programmeur in een ander bestand neergeschreven, eveneens in C. Deze worden door ATOM vertaald en de vertaalde procedures worden toegevoegd aan het programma.

De specificatie van de programmeur, d.w.z. de interface met de instrumentatiecode en de instrumentatiecode zelf, wordt samen met de ATOM bibliotheek omgezet tot een instrumentatieprogramma, dat dan kan gebruikt worden om andere programma's effectief te instrumenteren.

Met ATOM werden o.a. reeds volgende applicaties [Wall92] ontwikkeld:

- Pixie: voor het vergaren van profielinformatie;
- ThirdDegree: voor instrumentatie met het oog op het detecteren van geheugenlekken;
- Hiprof: voor instrumentatie met het oog op prestatie-analyse.

Etch [Rome97] is een programma dat sterk op ATOM lijkt, maar op de Windows NT/x86 architectuur draait. Net zoals Spike kan het werken met dynamisch gelinkte programma's.

EEL (Executable Editing Library) [Laru95, Laru96] is een C++ bibliotheek die de gebruiker de mogelijkheid biedt om nagenoeg platformonafhankelijk binaire programma's te editeren. Daartoe biedt het een interface aan die op een iets lager niveau dan die van ATOM staat, maar daardoor ook krachtiger is.

JiTI [Rons00] is een JIT-instrumentatie-omgeving voor SPARC en x86 programma's. Programma's worden tijdens de uitvoering geïnstrumenteerd met code die de gebruiker opgegeven heeft. JiTI kan o.m. gebruikt worden voor racedetectie in parallelle programma's [Rons98, Rons99].

Dixie [Fern99] is een verzameling hulpprogramma's voor instrumentatie. De nadruk ligt op accurate resultaten, analyse van alle aspecten van architecturen en de mogelijkheid om met verscheidene instruc-

tiesets tegelijkertijd te werken. Hierdoor is het niet alleen een hulpmiddel voor instrumentatie, maar kan het tevens gebruikt worden voor het onderzoeken van instructiesets.

1.5 Voorgeschiedenis van dit onderzoek

Het bredere onderzoek waarin ons onderzoek kadert is halverwege de negentiger jaren opgestart in Arizona door prof. De Bosschere en prof. Debray. Daar hebben zij de eerste implementatie van ALTO [Muth01, Muth99] geschreven, die verder samen met Robert Muth ontwikkeld is. ALTO is een optimalisator gericht op snelheidsoptimalisaties na het linken, voor de Alpha architectuur.

Het onderzoek dat door de auteur uitgevoerd werd, kaderde hier oorspronkelijk ook in. ALTO, met de erin aanwezige datastructuren en de gemaakte keuzes voor o.m. de interne voorstelling, werd verder als ontwikkelomgeving gebruikt.

Uit ALTO is na een tijd SQUEEZE ontstaan, waaraan ontwikkeld werd in Gent en in Arizona. Vanaf dat moment zijn wij ons uitsluitend met compactie gaan bezighouden, inclusief de compactie van data. De reden hiervoor was voornamelijk dat compactie nog veel meer een onontgonnen terrein was dan snelheidsoptimalisatie. Daar waar compactie in Arizona eerder als een extraatje gezien werd, werd het echter door de auteur van dit proefschrift als voornaamste doelstelling beschouwd.

De manier waarop de compactie gebeurt in ons prototype is sterk gerelateerd aan de evolutie in ons gezamenlijk onderzoek met behulp van ALTO en SQUEEZE. In hoofdstuk 2 wordt de interne voorstellingswijze van programma's gekozen. Datastructuren werden waar nodig soms aangepast, maar voor het overige grotendeels overgenomen uit ALTO. Toch blijken ze, mits de nodige aandacht voor verfijningen aan die voorstellingswijze, goed geschikt om aan programmacompactie te doen na het linken.

We zullen op het einde van dit proefschrift, bij het trekken van conclusies en het vooropzetten van mogelijke nieuwe onderzoekspaden, dan ook trachten in te schatten hoezeer deze evolutie en het enigszins vastzitten aan eerder ingeslagen paden belemmerd heeft dat er nog betere resultaten kunnen behaald worden.

Het kan wat vreemd klinken dat we voor de studie van codecompactie en het implementeren van een prototype voor de Alpha archi-

tectuur en de omgeving van een Unix werkstation geopteerd hebben. Waarom niet meteen werken in een omgeving voor ingebedde systemen? De reden is de Alpha architectuur zelf. Dit is een zeer sobere RISC architectuur, die conditiebits noch registervensters gebruikt. Dit zorgt ervoor dat we ons tijdens het onderzoek konden concentreren op de compacterende programmatransformaties zelf i.p.v. op de randtransformaties die elke programmatransformator moet implementeren. Aldus hebben we grotendeels energie- en tijdverspilling kunnen vermijden bij bv. de implementatie van het disassembleren en opnieuw assembleren van programma's.

Hoofdstuk 2

Interne Voorstelling

"Dromen is erg belangrijk. Je kan alleen iets realiseren als je je er een voorstelling van kan maken."

George Lucas

Elke onderzoeker die analyses en transformaties van programma's bestudeert, wordt geconfronteerd met het kiezen van een voorstellingswijze van programma's.

Deze keuze heeft een grote invloed op de haalbare efficiëntie en effectiviteit van de door de onderzoeker bestudeerde technieken. Een goede voorstellingswijze biedt de mogelijkheid efficiënt informatie over een programma te verzamelen, bij te houden en aan te passen nadat de transformaties uitgevoerd zijn. De keuze hangt dus af van de analyses en transformaties die we wensen toe te passen. Welke transformaties we kunnen toepassen zonder het gedrag van een programma te wijzigen hangt dan weer af van de a priori kennis die we hebben over het programma en de informatie die we uit de binaire voorstelling van het programma kunnen extraheren en kunnen uitdrukken in de interne voorstelling.

Daarom is het nuttig eerst in te gaan op de informatie waarover we beschikken bij het compacteren van een te linken of reeds gelinkt programma. Eens deze informatie afgebakend is zullen we de interne voorstellingswijze bespreken waarvoor wij gekozen hebben. Tevens zullen we aangeven hoe gaten in de informatie over het programma op een elegante manier opgevangen kunnen worden.

2.1 Doelprogramma's

De analyses en transformaties die in dit proefschrift besproken worden kaderen in het trachten te verkrijgen van een zo groot mogelijke, automatische compactie van vertaalde programma's. De programma's waarop we deze technieken wensen toe te passen kunnen heel informeel beschreven worden als programma's geschreven in een hogere programmeertaal. Daarbij richten we ons op technieken die bij voorkeur in sterke mate onafhankelijk zijn van de programmeertaal of -talen waarin een programma geschreven is. Meer nog, de technieken moeten kunnen toegepast worden zonder dat de broncode beschikbaar is.

Onze keuze is daarbij gevallen op statisch gelinkte programma's, zoals in de inleiding geduid werd. Dit leidt ons tot volgende aannames:

1. Al die informatie die noodzakelijk is om een programma te linken staat ter onzer beschikking. Aldus stellen we een zo algemeen mogelijk toepasbare compactie voorop.

2. De traditionele programmeeromgeving bestaande uit vertalers die bestand per bestand vertalen en een linker die bovengenoemde informatie in objectbestanden terugvindt, legt een aantal beperkingen op aan code- en datastructuren. We kennen deze beperkingen. Ze gelden voor alle programma's.

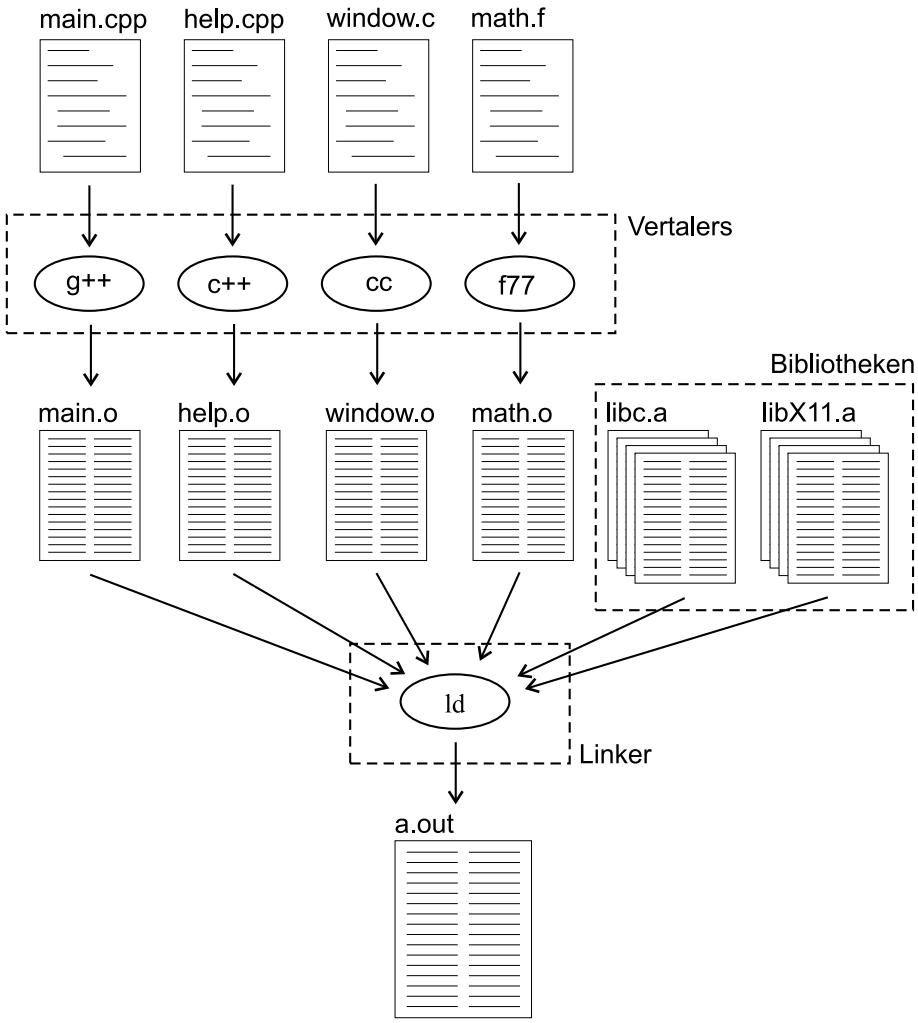
Om de beschikbare informatie meer in detail te kunnen beschrijven is het noodzakelijk eerst even stil te staan bij de werking van een omgeving bestaande uit vertalers, een linker en bibliotheken. Daaruit kunnen we een aantal eigenschappen van programma's afleiden en de beschikbare informatie in meer detail bespreken.

2.2 Werking van een vertaler en linker

In figuur 2.1 is een typische programmeeromgeving afgebeeld. De kerncomponenten van zo'n omgeving zijn de vertalers, de linker(s) en bibliotheken.

2.2.1 De vertaler

De vertaler vertaalt broncode in een hogere programmeertaal naar objectbestanden. In principe gebeurt deze vertaling bestand per bestand:



Figuur 2.1: De werking van een omgeving bestaande uit verschillende vertalers en een linker.

terwijl de vertaler één bestand aan het omzetten is heeft hij geen kennis van de code in andere bronbestanden.

Sommige vertalers kunnen verschillende bestanden gelijktijdig omzetten en bij de omzetting van het ene bestand rekening houden met de code uit andere bronbestanden. Dit kan leiden tot efficiëntere (snellere, kleinere, ...) programma's, bv. indien hierdoor oproepconventies vermeden kunnen worden. In het algemeen spreekt men in zo'n geval van intermodulaire optimalisaties, omdat ze de samenwerking tussen twee programmadelen in verschillende modules optimaliseren.

Wij maken aangaande het al dan niet uitvoeren van intermodulaire optimalisaties geen enkele veronderstelling. Wel zullen we rekening houden met het feit dat intermodulaire optimalisatie vaak niet mogelijk is, zoals bij interfaces tussen applicatiecode en bibliotheekcode of tussen stukken code in verschillende programmeertalen. Dit soort grenzen is dus bij het uitstek een gebied waar een optimaliserende linker extra mogelijkheden biedt.

We gaan er wel van uit dat verschillende vertalers met een linker moeten kunnen samenwerken, en dat dit bovendien het geval moet zijn voor vertalers die onafhankelijk van de linker ontwikkeld zijn. De vertalers hebben met andere woorden geen kennis van de interne werking van de linker (afgezien van de opgelegde conventies natuurlijk).

Afhankelijk van de specifieke omgeving (architectuur en besturingssysteem) zullen code en data in het objectbestand al dan niet door elkaar gealloceerd worden. Wij zullen in dit proefschrift uitgaan van een strikte scheiding tussen code en data, omdat dit de discussie duidelijker maakt en omdat deze veronderstelling geen fundamentele invloed heeft op de in dit proefschrift besproken en geëvalueerde technieken. In omgevingen waarin code en data wel door elkaar staan, zoals in programma's gegenereerd door de Intel C vertalers voor de x86 architectuur, zal men technieken moeten gebruiken die code en data van elkaar kunnen onderscheiden. Alhoewel het hier in theorie een zeer moeilijk probleem betreft [GH00], zal in de overgrote meerderheid van de programma's het onderscheid tussen data en code makkelijk te maken zijn.

2.2.2 De bibliotheken

Sommige onderdelen van applicaties zijn typisch voor verscheidene applicaties nuttig. Dit is bv. het geval voor invoer/uitvoerroutines en

de grafische interfaces van programma's. Opdat deze stukken code niet steeds opnieuw geprogrammeerd zouden moeten worden, worden ze als bibliotheken aan een programmeeromgeving toegevoegd. Er wordt een interface naar de bibliotheken gedocumenteerd en de programmeur kan dan de code uit de bibliotheken gebruiken in zijn applicatie.

Omdat niet iedereen de hele functionaliteit van zulke bibliotheken zal gebruiken, is het nuttig bibliotheken op te splitsen in kleinere delen die apart in een applicatie kunnen opgenomen worden. Een bibliotheek is daarom eigenlijk niet meer dan een verzameling objectbestanden. Dit biedt twee voordelen. Enerzijds is de verwerking van bibliotheekcode door de linker nagenoeg identiek aan de verwerking van de door de vertaler gegenereerde objectbestanden. Anderzijds kunnen bibliotheken net als applicaties aangemaakt worden met een vertaler: het volstaat een aantal objectbestanden te genereren op de klassieke manier en die dan samen te brengen in een bibliotheek.

Welk deel van de code van een bibliotheek zal gebruikt worden voor een bepaalde applicatie, wordt bepaald door de programmeur en de interne opbouw van de bibliotheek. Dit wordt in de volgende paragraaf beschreven.

2.2.3 De linker

Eenmaal de broncode van een applicatie omgezet is in objectbestanden, worden deze door de linker samengevoegd tot een volledig programma en aangevuld met de nodige bibliotheekcode. Dit samenvoegen bestaat uit verschillende stappen. Voor een uitvoerige bespreking van de functionaliteit van moderne linkers verwijzen we naar [Levi00]. We beperken ons hier tot een kort overzicht.

1. De linker zorgt ervoor dat alle code en data uit de objectbestanden een plaats (locatie) in het finale programma krijgen.

2. De linker gaat in bibliotheken op zoek naar ongedefinieerde symbolen (code, bv. procedures, en data, bv. globale veranderlijken) waarnaar in de tot nu toe gelinkte bestanden verwezen werd en die niet door de programmeur zelf gedefinieerd zijn. Bibliotheken zijn zoals gesteld niet meer dan een verzameling objectbestanden, vaak aangevuld met een index die aangeeft welk symbool in welk objectbestand gedefinieerd wordt. Uit de bibliotheken worden de

objectbestanden gehaald die deze symbolen definiëren. Omdat deze objectbestanden op hun beurt naar (nog) niet gedefinieerde symbolen kunnen verwijzen, is dit een iteratief proces, dat symboolresolutie genoemd wordt. Het proces stopt als elk symbool waarnaar gerefereerd wordt op precies één plaats gedefinieerd is.

3. De linker past alle statisch vastgelegde adressen (wijzers) in het programma aan aan de finale plaats van code en data in het programma. Enerzijds worden hierbij hardgecodeerde adressen in de objectbestanden aangepast aan hun nieuwe plaats, anderzijds worden de tot dan toe symbolische verwijzingen naar objecten in andere modules omgezet door de nu bekende adressen van deze objecten hard in te schrijven in het programma waar ernaar verwezen wordt.

4. Extra informatie, zoals bv. deze voor het ontluizen van programma's, wordt eveneens gecombineerd.

Alhoewel de vertaler en de linker vaak met een enkel commando opgestart worden door de programmeur en ze soms nauwelijks te onderscheiden zijn, gaan we ervan uit dat de programmeur noch de vertaler weten hoe de linker intern werkt. Meer concreet betekent dit dat de vertaler tijdens het vertalen van een bronbestand de finale plaats van het corresponderende objectbestand in het volledige programma niet kent. Deze veronderstelling leidt ertoe dat de vertaler zogenaamde relocatie-informatie aan het objectbestand moet toevoegen. Deze zal later overigens voor enkele analyses van uitzonderlijk belang blijken te zijn.

Voorbeeld 2.1 We beschouwen een eenvoudig bronbestand in C code om de (samen)werking van een vertaler en linker te illustreren. Het broncodebestand g.c (figuur 2.2) definieert een zogenaamde inpakfunctie `g()`, d.w.z. een functie die een nieuwe interface naar bepaalde functionaliteit aanbiedt. In dit geval is de nieuwe interface die van `g()` voor de functionaliteit die al in `f()` aanwezig was. De verpakte functie `f()` bevindt zich in een ander bronbestand.

Stap voor stap zullen we de nodige gegevens in het objectbestand (figuur 2.3) invullen. Daarvoor maken we gebruik van het ECOFF formaat [Com00] dat o.a. gebruikt wordt door Compaq Tru64 Unix voor de Alpha architectuur [Com98]. Om de figuur niet te overladen hebben we de voor dit voorbeeld irrelevante

```
int f(int x);

int g(int y)
{
   return f(y);
}
```

Figuur 2.2: Broncode uit het bestand g.c.

details evenwel weggelaten. De gebruikte instructieset is tevens die van de Alpha architectuur. We zullen het objectbestand opbouwen in een volgorde die klaar en duidelijk aangeeft waarom bepaalde informatie in het objectbestand vereist is. Dit is niet de volgorde waarin de vertaler de omzetting doet. Die volgorde is overigens van geen belang voor dit proefschrift.

De procedure-oproep Om de procedure-oproep f(y) te implementeren gebruiken we een indirecte spronginstructie (jsr of *jump to subroutine* (regel 24)), die uit register t12 het adres van de op te roepen procedure haalt en het terugkeeradres in register ra stopt. Het gebruik van een indirecte sprong is nodig omdat het adres van f(), dat op dit moment onbekend is, zowat overal in de (uitgelijnde) 64-bits adresruimte kan liggen, en we dus niet kunnen veronderstellen dat dit adres in een 32-bits instructie kan gecodeerd worden.

Het gebruik van de registers t12 en ra wordt opgelegd door de oproepconventies. Waarom dit gedaan wordt zal enkele paragrafen verder duidelijk worden.

Het bewaren van het terugkeeradres Omdat het terugkeeradres van een oproeper van g() zal overschreven worden door het terugkeeradres van de procedure-oproep f(y), wordt het terugkeeradres op de stapel bewaard: de instructies op regel 21 en 28 (lda of *load address*) alloceren en dealloceren daartoe een stapelvenster door de stapel te verlagen resp. te verhogen. De instructies op regels 22 en 27 schrijven het terugkeeradres weg op de stapel en laden het terug in.

Het opladen van het adres De meest efficiënte manier om een adres (van een procedure) te plaatsen in een register is vaak het opladen van het adres uit het geheugen. Een veelgebruikte techniek is dan ook het aanmaken van een adres-

```
kop:
   gp: 0x8020
   symbolen: 2
   relocaties: 4
   secties: 2
      code: start 0x00 omvang 0x30
      data: start 0x30 omvang 0x10

symbolen:
   0: undefined proc f
   1: 0x00 proc g

relocaties:
   0x00 GPDISP
   0x10 LITERAL
   0x18 GPDISP
   0x30 REFQUAD symbool 0

code:
   0x00 ldah gp, 1(t12)          ; zetten van de gp
   0x04 lda gp, -32736(gp)       ; zetten van de gp
   0x08 lda sp, -8(sp)           ; alloceren stapelvenster
   0x0c stq ra, 0(sp)            ; bewaren terugkeeradres
   0x10 ldq t12, -32752(gp)      ; opladen adres van f()
   0x14 jsr ra, (t12)            ; oproep naar f()
   0x18 ldah gp, 1(t12)          ; herstellen van de gp
   0x1c lda gp, -32760(gp)       ; herstellen van de gp
   0x20 ldq ra,0(sp)             ; ophalen terugkeeradres
   0x24 lda sp,8(sp)             ; dealloceren stapelvenster
   0x28 ret ra                   ; terugkeer naar oproeper
   0x2c no-op

data:
   0x30 0x00000000
   0x34 0x00000000
   0x38 0x00000000
   0x3c 0x00000000
```

Figuur 2.3: Geïnterpreteerde inhoud van het objectbestand g.o. De code is gedisassembleerd en de adressen van de instructies zijn erbij vermeld, net als bij de data. Tevens zijn de instructies becommentarieerd.

tabel waarin alle relevante statische code- en data-adressen opgeslagen worden. Het volstaat dan om een wijzer naar die tabel bij te houden (ook wel de globale wijzer of *global pointer* (gp) genaamd) in een vast register om door middel van indexering t.o.v. die gp elk adres uit de tabel te kunnen opladen.

De vertaler kan het adres van de functie f() evenwel nog niet opslaan in de tabel, aangezien dat adres nog niet bekend is. Daarom voorziet de vertaler voorlopig alleen in de nodige plaats voor dit adres (regels 33-34) en geeft hij de linker de opdracht om daar later het juiste adres in te vullen zodra dit bekend is. Deze opdracht wordt als een relocatiegegeven in het objectbestand opgeslagen (regel 16): het relocatiegegeven bevat de plaats waarop een adres moet komen te staan (d.w.z. de gereserveerde locatie in de adrestabel) en een beschrijving van het adres dat er moet komen te staan.

De enige beschrijving die daarvoor voorhanden is tijdens het vertalen, is een symbolische verwijzing naar procedure f(). Daarom wordt de procedure f() toegevoegd aan de zogenaamde symbooltabel als een niet-gedefinieerde procedure (regel 9), en verwijst het relocatiegegeven naar dit symbool in de symbooltabel. Als de linker een definitie voor f() gevonden heeft tijdens het linken en hij de bijhorende code een plaats in het geheugen gegeven heeft, zal de relocatie uitgevoerd worden: het adres van f() wordt dan ingevuld in de tabel.

De instructie die het adres vanuit de tabel zal opladen gaat in de code de spronginstructie vooraf (regel 23). Ze gebruikt de gp als basisregister en de index in de tabel t.o.v. de gp (in dit geval -32752) wordt als letterlijke afstand in de instructie ingeschreven. De vertaler weet op dit moment echter nog niet wat de waarde van de gp zal zijn in het finale programma. Evenmin kent de vertaler de plaats waar de adrestabel uiteindelijk zal terechtkomen. Opnieuw wordt het doen kloppen van de rekening overgelaten aan de linker: een relocatiegegeven wordt toegevoegd aan het objectbestand. Dit relocatiegegeven (regel 14) bevat het adres van de instructie in het huidige objectbestand waarin de index moet aangepast worden en geeft middels haar type aan om welk type relocatie het gaat. Wanneer de waarde van de gp en de plaats van

de adrestabel vastgelegd zijn door de linker, zal aan de hand van dit relocatiegegeven de letterlijke afstand in de instructie aangepast worden.

Het zetten van de gp Elk objectbestand heeft in theorie een eigen globale wijzer. Dit betekent dat de waarde van de gp tijdens de uitvoering van het programma opnieuw moet gezet worden na elke intermodulaire sprong. Daartoe worden in de code de lda en ldah (*load address high*) instructies toegevoegd. Zij berekenen de waarde van de gp uitgaande van het adres van de procedure g() (regel 19-20, na een oproep van procedure g() zit haar adres in register t12 waar eerst $1 \cdot 2^{16}$ (ldah) en vervolgens -32736 (lda) bij opgeteld worden) en uitgaande van het terugkeeradres na de oproep naar f() (regels 25-26, dit terugkeeradres zit in register ra na de terugkeerinstructie in procedure f()). Omdat het verband tussen de gp en de plaats van de procedure g() in het finale programma niet vastligt, moeten ook deze berekeningen door de linker kunnen aangepast worden eens de definitieve adressen bekend zijn. Daartoe worden opnieuw twee relocatiegegevens toegevoegd aan het objectbestand (regels 13 en 15).

Om deze berekeningen mogelijk te maken moet elke procedure "weten" in welke registers zij adressen kan terugvinden waar tijdens het vertalen al naar kon verwezen worden. Hiertoe wordt de keuze van deze registers (t12 en ra) in oproepconventies vastgelegd.

De eerste twee instructies van deze procedure worden de proloog genoemd. Voor intermodulaire oproepen moet de proloog uitgevoerd worden en zal er dus naar de eerste instructie van de procedure gesprongen worden. Voor intramodulaire sprongen staat de gp al op de correcte waarde en kan de proloog dus overgeslagen worden. Intramodulaire (directe) sprongen zullen dus de procedure binnenkomen na de proloog.

Een symbool voor de procedure g() Net zoals er in dit bestand verwezen wordt naar een niet-gedefinieerd symbool f(), waarvan de definitie gezocht zal worden in andere objectbestanden of bibliotheken, moeten oproepers van g() een definiërend symbool van g() kunnen terugvinden. Daarom wordt g() als procedure toegevoegd aan de symbooltabel,

samen met haar beginadres (regel 10).

Vulling Omdat secties in objectbestanden aan bepaalde alignementsvoorwaarden moeten voldoen, worden zowel de data- als de codesectie aangevuld met vulling bestaande uit nullen (regels 30 en 35-36).

De koppen Opdat de linker al deze informatie in het objectbestand correct zou inlezen en interpreteren, wordt er een beschrijving van in het objectbestand geschreven (regels 1-6). Deze beschrijving in de koppen omvat bv. de voorlopige waarden van de `gp`, de beginadressen en grootte van data en code, de plek waar de relocaties in het bestand beschreven staan, etc.

□

Voorbeeld 2.1 geeft aan dat een linker minstens volgende zaken nodig heeft om een programma te kunnen linken:

- koppen die de inhoud van het bestand beschrijven;
- code en data zelf;
- relocatie-informatie;
- symboolinformatie.

In de volgende sectie wordt verder op deze informatie ingegaan.

2.3 Beschikbare informatie

Om meer concreet en in meer detail de beschikbare informatie te belichten, kiezen we een bestaand objectformaat als voorbeeld. Dit is opnieuw het ECOFF formaat van Compaq Tru64 Unix voor de Alpha architectuur. Dit formaat is tevens het formaat waarvoor een prototype compactor ontwikkeld werd. We wensen op te merken dat dit formaat sterk lijkt op andere courante formaten, zoals ELF, en er dezelfde basiskenmerken mee deelt.

2.3.1 Code en data

In tegenstelling tot het vereenvoudigde voorbeeld 2.1 is het niet zo dat een programma slechts één codesectie en één datasectie bevat.

Statisch gelinkte programma's zijn meestal opgedeeld in drie zogenaamde segmenten. Elk segment bestaat uit een aantal secties met dezelfde kenmerken.

Tekstsegment Dit segment bevat (constante) code en constante data. Constante data wordt bij de code in eenzelfde segment gestopt om redenen van efficiëntie. Zo moeten deze code en data, omdat ze toch constant zijn, nooit weggeschreven worden in het wisselbestand. Men kan ze immers steeds uit het programmabestand op schijf opladen. Zo spaart men ruimte in het wisselbestand van het virtuele geheugensysteem. Sommige besturingssystemen sparen verder ook fysiek RAM-geheugen door voor verschillende gelijktijdige uitvoeringen van een programma slechts één kopie van deze code en data in het geheugen te stockeren.

De secties die men in het tekstsegment terugvindt zijn:

text Deze sectie bevat de code van het eigenlijke programma.

init en fini Deze secties bevatten de code die moet uitgevoerd worden voor en na de uitvoering van het eigenlijke programma. Deze secties zijn met name van belang voor de initialisatie van de C bibliotheek en voor de uitvoering van statische constructoren en destructoren in bv. C++ programma's.

lita De lita of *literal address pool* bevat de adrestabellen met adressen van objecten (code en data) in het programma. Men stockeert deze tabellen in een aparte sectie in de objectbestanden omdat ze dan door de linker kunnen samengevoegd worden tot een of meer grotere tabellen in het finale programma: de globale adrestabel(len).

rdata Deze sectie bevat alle overige constante data, zoals bv. tekenrijen en constante getallen die in het programma voorkomen.

Datasegment Dit segment bevat de schrijfbare datasecties. Meestal wordt er een onderscheid gemaakt tussen kleine en andere data. Klein betekent bv. maximaal 16 bits breed. De voornaamste reden

om dit onderscheid te maken is de optimalisatie van het virtuele geheugensysteem. Door de kleinere, frequenter gebruikte data te bundelen in een aparte sectie (*sdata* of *small data*) kan men het aantal paginafouten verminderen.

Nulsegment Dit segment bevat de op nul te initialiseren data van een programma. Men stopt die data in een apart segment omdat het niet nodig is deze data in het objectbestand op te slaan, in tegenstelling tot de data uit het datasegment. Zodoende kan men schijfruimte sparen. Een lader zal alle data in dit segment op nul zetten van zodra het programma opgeladen wordt voor uitvoering. Zoals in het datasegment zal men ook hier een onderscheid maken tussen kleine en andere data. Dit segment heet ook wel het *bss*-segment (*Block Started by Symbol*). Deze benaming is afkomstig van de IBM 704 architectuur, die een BSS instructie bevatte om geheugen te alloceren.

Voorbeeld 2.2 In figuur 2.4 wordt de samenstelling van drie objectbestanden (`a.o`, `b.o` en `c.o`) getoond. In de objectbestanden volgen de segmenten op elkaar. Het uit deze objectbestanden gegenereerde programma bestaat uit de samengevoegde code en data. Het tekstsegment hoeft in het finale programma niet aan te sluiten op het datasegment.

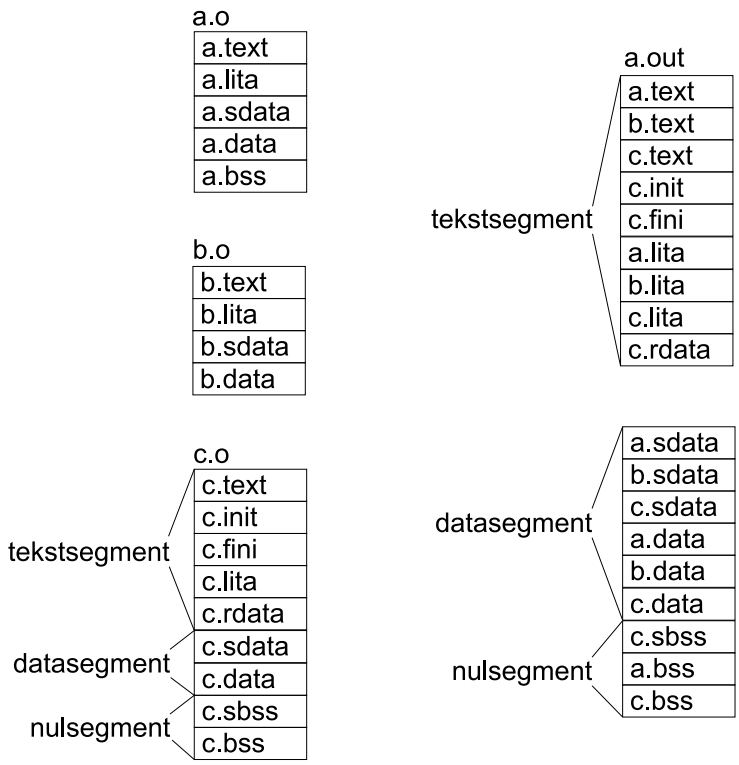
□

2.3.2 Relocatie-informatie

Relocatie-informatie geeft aan op welke plaatsen in een programma aanpassingen moeten doorgevoerd worden als delen van het programma verplaatst worden in het geheugen (gereloceerd).

Vertalers genereren meestal (zoals in voorbeeld 2.1) code die begint op adres nul. De data volgt dan meteen na de code. Wanneer verschillende objectbestanden samengevoegd worden tot een finaal programma, worden de stukken code en data per sectie na elkaar geplaatst, zoals in figuur 2.4. De meeste, zoniet alle, code en data komen dus op een andere plaats terecht dan die plaats die de vertaler hen oorspronkelijk had toegewezen. De relocatie-informatie maakt deze verplaatsing mogelijk.

Merk op dat de relocatiegegevens niet enkel nodig zijn voor het linken, ze zijn tevens noodzakelijk voor compactie na het linken. Aange-



Figuur 2.4: De opdeling van drie objectbestanden en de opdeling van het daaruit gegenereerde programma.

zien de gecompacteerde code kleiner is, zal de meeste code verplaatst worden tijdens de compactie. Samen met zulke verplaatsingen moeten de in het programma opgeslagen adressen aangepast worden. Hiervoor zijn de relocatiegegevens noodzakelijk.

De aanwezige relocaties in het programma vertellen bovendien iets over het gedrag van een programma. Alle plaatsen die vermeld worden in de relocatiegegevens bevatten adressen of relatieve verwijzingen naar adressen. Aan de hand van relocaties kunnen we bijgevolg de hard gecodeerde adressen in een programma identificeren. In voorbeeld 2.1 stellen de relocaties ons in staat om de eerste 8 bytes uit de datasectie als een procedure-adres te herkennen. Zoals verder zal blijken moeten we er dan vanuit gaan dat dit opgeslagen adres kan gebruikt worden om `f()` op te roepen.

2.3.3 Symboolinformatie

Symboolinformatie is vereist om het de linker mogelijk te maken het verband tussen symboolreferenties en symbooldefinities te leggen, en deze verbanden mits relocaties te vertalen naar hard gecodeerde adressen in het programma.

Het enige nuttige gebruik hiervan voor ons en voor compactie tijdens of na het linken is tot nog toe het identificeren van procedures in de code geweest. Proceduresymbolen bevatten immers het startadres van procedures. Dit volstaat als informatie om procedures af te bakenen omdat we er van uitgaan dat procedures als één ononderbroken sequentie van instructies opgeslagen worden. Indien een proceduresymbool bovendien een naam geeft aan een procedure, wordt de procedure geëxporteerd uit zijn objectbestand, om opgeroepen te kunnen worden vanuit andere bestanden. We kunnen er dan van uitgaan dat de procedure de oproepconventies naleeft.

Voor andere symbolen, zoals voor globale veranderlijken, hebben we nog geen gebruik gevonden. Uit de symboolinformatie is immers niet op te maken of symbolen elkaar overlappen, hoe groot de veranderlijken zijn, etc.

2.3.4 A priori kennis over een programma

Waar we tot nu toe informatie besproken hebben die terug te vinden is in een objectbestand, is er ook informatie beschikbaar over eigenschap-

pen van alle programma's. Deze informatie wordt nu kort besproken.

Oproepconventies

Om verschillende, onafhankelijk van elkaar vertaalde programmadelen met elkaar te laten samenwerken, moeten een aantal afspraken gemaakt worden: de gegenereerde code moet aan een aantal oproepconventies voldoen. Voorbeelden van vastgelegde conventies zijn:

1. vaste registers bevatten de stapelwijzer, de globale wijzer, het oproepadres, het terugkeeradres, ...;
2. vaste registers worden gebruikt voor de doorgave van parameters bij een procedure-oproep;
3. een aantal registers zijn vooraf te bewaren, d.w.z. dat de oproeper van een procedure verantwoordelijk is voor het bewaren van de inhoud van deze registers vóór de oproep indien hij ze na de oproep nog wil gebruiken, m.a.w. dat de opgeroepen procedure hun inhoud mag wijzigen. Andere registers zijn bij conventie achteraf te bewaren, d.w.z. dat de oproeper van een procedure mag veronderstellen dat de inhoud van een register onveranderd is na de uitvoering van de opgeroepen procedure, m.a.w. de opgeroepen procedure moet de registerinhoud bewaren.

De kennis van deze conventies is niet noodzakelijk om een programma te compacteren. Zoals echter zal blijken is het een belangrijke extra informatiebron voor tal van analyses.

Beperkingen aan de adresberekeningen

Naast de oproepconventies waarmee een vertaler rekening moet houden bij het genereren van code om intermodulaire samenwerking van code mogelijk te maken, zijn er nog andere beperkingen die voortvloeien uit het apart vertalen van modules op zich.

Eén van de belangrijkste beperkingen is die op de adresberekeningen die een vertaler kan genereren.

Voorbeeld 2.3 Beschouw het stukje C code in figuur 2.5a. Hiervoor kan de vertaler bv. de assemblercode in figuur 2.5b genereren. Merk

```
int x;                  0x00 ldah gp, 1(t12)
int y;                  0x04 lda gp, -32688(gp)
                        0x08 ldq t0, -32752(gp)
void h(void)            0x0c ldq t1, 0(t0)
{                       0x10 addq t1, 1, t1
   x++;                 0x14 stq t1, 0(t0)
   y–;                  0x18 ldq t1, 8(t0)
}                       0x1c subq t1, 1, t1
                        0x20 stq t1, 8(t0)
                        0x24 ret ra
```

(a) C code (b) assemblercode

Figuur 2.5: Een stukje C code en het ermee corresponderend stukje assemblercode ter illustratie van adresberekeningen.

op dat in dit stukje code slechts 1 adres opgeladen wordt uit de globale adrestabel (door de instructie op adres 0x08), zijnde het adres van de globale veranderlijke x. Het adres van y wordt niet apart opgeladen, want de vertaler weet waar hij y t.o.v. x gealloceerd heeft, en kan het adres van x dus hergebruiken om y te laden. Mochten de twee veranderlijken extern gedeclareerd geweest zijn, dan kon de vertaler dit niet en moet het adres van y net als dat van x uit de tabel opgeladen worden. □

Zoals uit het voorbeeld duidelijk wordt, kan de vertaler geen berekeningen (d.w.z. instructiesequenties) genereren op adressen die verwijzen naar data (of code) uit andere modules, eenvoudigweg omdat hij die adressen niet kent.

Analoog daarmee kan een vertaler geen berekeningen genereren die vertrekkend vanuit een adres in één objectbestand een adres uit een ander objectbestand berekenen. Opnieuw is de reden dat de vertaler onmogelijk de relatie tussen de twee adressen kan kennen.

Deze beperking op mogelijke adresberekeningen stelt zich echter nog scherper. Zo kan de vertaler evenmin berekeningen genereren die binnen hetzelfde objectbestand zouden vertrekken van een adres dat in één sectie ligt en uitkomen op een adres dat in een andere sectie ligt. Opnieuw is de reden dat een vertaler tijdens het vertalen het verband tussen beide adressen niet kent.

Als we de secties in objectbestanden voortaan code- en datablokken noemen, dan kunnen we stellen dat er geen blokgrensoverschrijdende

```
addq r1,r3,r2
ldq r1,20[r1]        ldq r1,20[r1]
ldq r0,10[r2]
addq r0,r1,r3        addq r1,r1,r3
```

(a) origineel | (b) na optimalisatie

Figuur 2.6: Links een instructiesequentie met een potentieel redundante leesoperatie, rechts de corresponderende sequentie indien de tweede leesoperatie inderdaad redundant was. (*addq* voert een optelling uit, waarbij de derde operand de destinatie-operand is.)

berekeningen mogelijk zijn, ook niet in finale programma's.

Hieruit kan men afleiden dat er, om toegang tot een blok te hebben, minstens ergens een (reloceerbare) wijzer naar die sectie hard in het programma moet gecodeerd zijn, in de code of in de data. Indien dit niet het geval is, is een blok ontoegankelijk voor het programma en kan men het zonder problemen verwijderen.

2.3.5 Bijkomende beperkingen

Naast de oproepconventies en de beperkingen die volgen uit het apart vertalen en linken, leggen we zelf nog een aantal beperkingen op aan de klasse van programma's die we wensen te compacteren.

Parallellisme en vluchtig geheugen

Een van de compactietechnieken die later aan bod komen is het verwijderen van overbodige schrijf- en leesoperaties.

Beschouw de instructiesequentie in figuur 2.6. Als analyse van het programma ons leert dat register r3 de waarde 10 bevat, kunnen we daaruit afleiden dat beide leesoperaties vanop dezelfde plek in het geheugen zullen lezen. Als we aannemen dat de waarde op die geheugenlocatie niet gewijzigd wordt door "externe factoren", kunnen we de tweede leesoperatie vervangen door een kopieerinstructie die later weggeoptimaliseerd wordt na kopiepropagatie.

Onder "externe factoren" verstaan we alles, behalve het codefragment zelf, wat de waarde opgeslagen in de betreffende locatie kan veranderen tussen de twee leesoperaties door. Enkele mogelijkheden zijn:

1. Gemeenschappelijk geheugen: als de locatie waarvan gelezen

wordt behoort tot het gemeenschappelijk, beschrijfbaar geheugen van een parallelle applicatie kan een andere draad de waarde mogelijkerwijs veranderen. Als dit niet op een gecontroleerde manier gebeurt (m.a.w. met de nodige synchronisatie), zal men dit doorgaans als een programmeerfout beschouwen. Het toepassen van de hierboven besproken transformatie zal in dit geval het (foutieve) gedrag van het programma veranderen, en aldus misschien de programmeerfout verbergen. Om dit soort gevallen niet als een rem op het onderzoek te laten werken, hebben we ons in het onderzoek tot vandaag beperkt tot niet-parallelle programma's.

2. Vluchtige data: een programmeertaal als C kent "volatile" veranderlijken. Dit duidt erop dat de waarde niet in een register mag opgeslagen worden omdat de waarde, bv. de toestand van een I/O-poort, kan gewijzigd worden door externe factoren, zoals andere apparatuurcomponenten van het systeem. Deze kennis die de vertaler bijhoudt tijdens het vertaalproces gaat verloren bij het wegschrijven van de objectbestanden. Vooralsnog hebben we geen methode gevonden om deze informatie terug te achterhalen zonder de vertaler te moeten aanpassen. Om ook door deze mogelijkheid niet geremd te worden in het onderzoek, sluiten we programma's die hiervan gebruik maken voorlopig uit.

   Sommige vertalers en linkers (maar niet deze op ons doelplatform) kunnen een zogenaamde geheugenkaart produceren waarin de adressen van vluchtige data vermeld worden. Deze kaart zou gebruikt kunnen worden om ook programma's met vluchtige data correct te compacteren.

Afhandelen van excepties

Voorlopig werd er geen tijd besteed aan het aanpassen van ons prototype opdat het ook programma's met excepties zou kunnen compacteren. Het probleem van deze applicaties ligt in de manier waarop het exceptiemechanisme werkt. In specifiek daartoe bestemde datasecties ligt informatie opgeslagen over zogenaamde coderegio's en de bijhorende exceptie-afhandelaars. Deze informatie wordt o.a. gebruikt om de stapel te ontrollen. Onder meer door de al te beperkte beschikbare documentatie over dit proces zijn we er nog niet toe gekomen de ket het gecompacteerde programma corresponderende data correct uit te

schrijven. Dit betekent in concreto dat de aldus geproduceerde programma's incorrect werken in het geval er een exceptie optreedt.

Aangezien de aangepaste exceptie-informatie toch nog niet weggeschreven wordt, hebben we vooralsnog geen beperkingen opgelegd aan de programmatransformaties die we uitvoeren. Sommige van die transformaties kunnen er echter voor zorgen dat het onmogelijk wordt om correcte, aangepaste exceptie-informatie weg te schrijven. Dit heeft te maken met dat deel van de oproepconventies dat garandeert dat het exceptiemechanisme correct kan werken. Soms wijken wij momenteel van die standaard af, waar het niet zou mogen als we in staat willen blijven om excepties correct op te vangen en af te handelen.

De fractie van de bereikte compactie die verloren zou gaan als we de betreffende transformaties niet toepassen op code waar exceptieafhandelaars bijhoren, is ons inziens heel beperkt, zoniet verwaarloosbaar. Hierin kan dus geen reden gevonden worden om de resultaten in dit proefschrift in twijfel te trekken.

Zelfwijzigende code

Naargelang de manier waarop zelfwijzigende code geïmplementeerd wordt, zullen onze algoritmen correct werken. Er zijn twee gevallen te onderscheiden:

1. De veranderlijke code ligt opgeslagen in dynamisch gealloceerd geheugen of in statisch gealloceerd beschrijfbaar geheugen. In deze gevallen wordt die code als een zwarte doos beschouwd, waarover conservatieve veronderstellingen gemaakt worden opdat programmatransformaties het gedrag van het programma niet veranderen.

2. De veranderlijke code ligt opgeslagen in het tekstsegment van het programma. Aangezien onze transformaties ervan uitgaan dat deze secties niet veranderen, zullen ze het gedrag van het programma eventueel wijzigen in zo'n geval. Opdat het veranderen van de code echter mogelijk zou zijn, zal er evenwel een systeemoproep in het programma voorkomen die de betreffende pagina's in het geheugen beschrijfbaar zet. Tevens zal het tussengeheugen voor instructies geledigd moeten worden na het veranderen van de code. Deze systeemoproepen kunnen gedetecteerd worden door een compactor, zodat hij de compactie kan stopzetten.

Ook in dit geval wordt het programmagedrag dus niet veranderd, zij het dat er ook geen compactie uitgevoerd wordt.

Berekeningen op code-adressen

Adresberekeningen op code-adressen moeten identificeerbaar zijn aan de hand van de bijhorende relocaties. Indien dit niet het geval is, zullen de berekeningen tussen gewijzigde adressen in de getransformeerde code niet aangepast worden. Er weze opgemerkt dat dit geen vereiste is die enkel gesteld wordt door ons. Ook de meeste instrumentatieprogramma's gaan van deze veronderstelling uit.

We hebben gemerkt dat sommige vertalers voor functionele programmeertalen code genereren waarin wel berekeningen op code-adressen uitgevoerd worden. Dan maakt men bv. geen gebruik van een tabel met bestemmingsadressen, waarin geïndexeerd wordt. In de plaats daarvan gebruikt men een tabel met spronginstructies. Er wordt dan aan de hand van een index op het basisadres van deze sprongtabel naar één van die sprongen gegaan via een indirecte sprong, waarna de directe sprong in de tabel zelf uitgevoerd wordt. Het basisadres van deze sprongtabel is in dit geval een code-adres, waarop dus berekeningen uitgevoerd worden. Deze programma's kunnen niet gecompacteerd worden aan de hand van de in dit proefschrift beschreven technieken, althans niet met het prototype dat de hier besproken technieken implementeert. Merk op dat het het voorbeeld van de sprongtabel geen onoverkomelijke problemen stelt. Men zou bv. de sprongtabel gewoon als een stukje herloceerbare data kunnen voorstellen en niet als code in de ICFG opnemen.

Geheugensanering

In dit proefschrift worden een aantal transformaties besproken die het gebruik van basiswijzers en indexeringsoperaties in een programma kunnen wijzigen. Daardoor worden soms nieuwe wijzers gegenereerd of worden andere wijzers niet meer expliciet gegenereerd tijdens de uitvoering van het gecompacteerde programma, waardoor een conservatieve geheugensanering [Boeh98] in de fout kan gaan. Deze transformaties mogen dus niet toegepast worden op programma's met zulk een geheugensanering.

2.4 Interne voorstelling

Nu we weten welke informatie we ter beschikking hebben, kunnen we een interne voorstellingswijze kiezen, rekening houdend met de transformaties die we wensen uit te voeren.

2.4.1 Voorstelling van instructies

Voor de voorstelling van individuele instructies hebben we een drie-operand voorstellingswijze gekozen. Deze laag-niveau voorstellingswijze sluit zeer dicht aan bij de machinecode voor de Alpha architectuur. Dit is een machinecode die relatief eenvoudig en orthogonaal is, zodat ze op zich al sterk lijkt op de interne voorstelling die vele vertalers gebruiken. Deze keuze vormt ons inziens dan ook geen belemmering op de overdraagbaarheid naar andere architecturen van de in dit werk besproken technieken.

Recentelijk maakt de statisch-unieke-toekenning (SUT) voorstellingswijze veel opgang in het vertaalonderzoek. Wij hebben expliciet niet voor SUT gekozen. In die voorstellingswijze wordt er met symbolische registers gewerkt, die elk slechts op één plaats in het programma gedefinieerd worden. SUT laat toe op bijzonder efficiënte wijze allerlei transformaties uit te voeren en de literatuur over transformaties op SUT voorstellingen van programma's breidt elke dag uit. Het werken met symbolische registers resulteert echter in de noodzaak om opnieuw aan codegeneratie te doen na het transformeren van een programma.

Zo kunnen bv. de vrijheidsgraden die men heeft bij transformaties op SUT code er voor zorgen dat er extra instructies moeten toegevoegd worden (zogenaamde overloopcode) die registerinhouden tijdelijk in het geheugen bewaren (omdat er door het verplaatsen van instructies meer waarden bijgehouden moeten worden dan er registers beschikbaar zijn). Aangezien van vele procedures niet kan uitgemaakt worden of ze al dan niet recursief kunnen opgeroepen worden, en dus herbetreedbaar moeten geïmplementeerd worden, moeten de registerinhouden dan opgeslagen worden op de stapel. Dit laatste is uiterst moeilijk in programmatransformaties na het linken, omdat het zeer moeilijk is het stapelgedrag van een programma volledig te analyseren, laat staan te transformeren.

In het algemeen kunnen we stellen dat voor ons onderzoek alle programmatransformaties in aanmerking komen die de correctheid van

het programma bewaren. Hiermee bedoelen we dat het mogelijk moet zijn na gelijk welke transformatie over te gaan tot het assembleren van de code en het uitschrijven van het gecompacteerde programma. In de praktijk komt dit erop neer dat we ons willen onthouden van expliciete code-generatie, zoals bv. volledige registerallocatie waarbij aan de symbolische registers architecturale registers toegekend worden. Zulke technieken vallen buiten het bestek van dit proefschrift.

Uit deze discussie kan men besluiten dat een voorstelling die heel dicht bij het assemblerniveau ligt de meest geschikte keuze is.

2.4.2 De interprocedurale controleverloopgraaf (ICVG)

Naast de voorstelling van de individuele berekeningen of instructies willen we uiteraard ook structuren in het programma op een hoger niveau kunnen voorstellen. Het doel van die voorstelling is tweeledig:

1. Het efficiënt transformeren van het programma. Dit kan zijn het verplaatsen, verwijderen, toevoegen, dupliceren, etc. van code.

2. Het efficiënt analyseren van het programma. Dit betreft zowel analyse van het dataverloop als van het controleverloop.

Een natuurlijke keuze hiervoor is de controleverloopgraaf (CVG). Omdat de grafen die wij beschouwen grafen zijn die het volledige programma voorstellen, en dus verscheidene procedures bevatten, zullen we het verder over de interprocedurale controleverloopgraaf (ICVG) hebben.

In een controleverloopgraaf stellen de knopen de berekeningen (instructies) voor en modelleren de pijlen de mogelijk uitvoerbare paden. De instructies in drie-operand formaat, en geannoteerd met allerlei informatie zoals bv. het al dan niet constant zijn van hun operandi, vormen dus de knopen van een controleverloopgraaf.

In een controleverloopgraaf wordt elke instructie in principe met een gerichte pijl verbonden met alle mogelijke instructies die meteen erna kunnen uitgevoerd worden. Het geheel vormt dan een CVG die conservatief is: hij bevat de vereiste informatie om er controle- en dataverloopanalyses op uit te voeren, die alhoewel misschien conservatief, zeker correct zullen zijn, in de zin dat zij enkel zullen resulteren in transformaties die het gedrag van het programma niet wijzigen.

Door in de CVG enerzijds gericht artificiële pijlen en knopen in te voeren en anderzijds de knopen te groeperen per basisblok en per procedure kan men evenwel tegelijkertijd de efficiëntie (snelheid) en de effectiviteit (nauwkeurigheid) van een groot aantal analyses verbeteren.

Basisblokken

Voor vele dataverloopanalyses worden er zogenaamde dataverloopvergelijkingen opgesteld die gebruikt worden in een dekpuntanalyse. Zulke analyses hebben er alle voordeel bij het aantal vergelijkingen laag te houden. Daarom zullen vergelijkingen van individuele knopen indien mogelijk samengevoegd worden.

Dit is bij uitstek het geval voor basisblokken. Dit zijn sequenties van opeenvolgende instructies waarin het controleverloop enkel binnenkomt aan het begin van de sequentie en enkel buitengaat achteraan de sequentie. Als er één instructie van een basisblok wordt uitgevoerd, worden ze allemaal uitgevoerd in volgorde van de sequentie. Dit maakt het voor vele analyses mogelijk een samenvattende vergelijking voor het hele basisblok op te stellen en te gebruiken i.p.v. van één vergelijking per instructie.

Omdat basisblokken die uit meer dan één instructie bestaan zo frequent voorkomen, vormen basisblokken de knopen van onze ICVG. Elk basisblok bestaat dan op zich uit een sequentie instructies en over die sequentie als geheel worden tal van eigenschappen bijgehouden, zoals bv. welke registers gelezen of beschreven worden. Een basisblok kan dan tot op bepaalde hoogte als een atomaire rekeneenheid beschouwd worden tijdens de analyse van het programma.

Om het redeneren over basisblokken en ICVG's te vergemakkelijken, krijgen basisblokken ook een type. Zo zijn er bv. oproepblokken, terugkeerblokken, etc.

Procedures

Op een hoger niveau kan men procedures als atomaire eenheid beschouwen. De basisblokken worden daarom gegroepeerd per procedure, zodat per procedure eveneens een aantal algemene eigenschappen kunnen bijgehouden worden, zoals bv. een beschrijving van het stapelgedrag.

Het controleverloop binnen die procedures ligt echter niet eenduidig vast zoals dat in basisblokken het geval is. Bovendien hoeft een procedure geen unieke ingangs- of uitgangsknoop te hebben. Procedureoproepen bv. resulteren in pijlen die van een basisblok in het midden van de oproepende procedure naar een basisblok in de opgeroepen procedure gaan. De terugkeer na een procedure-oproep wordt dan weer voorgesteld door een pijl van een uitgang van de opgeroepen procedure naar een basisblok in het midden van de oproeper.

Omdat het vaak nuttig is deze interprocedurale pijlen op een uniforme manier met de intraprocedurale pijlen te behandelen, zullen we de ICVG niet uit procedures, maar uit basisblokken laten bestaan. We zullen dus geen aparte oproepgraaf opbouwen, waarin de procedures dan uit CVG's bestaan, maar integendeel rechtstreeks met één grote ICVG werken.

Procedures zullen we af en toe echter ook als atomaire eenheid gebruiken in de vergelijkingen voor data- en controleverloopanalyses. Daartoe willen we de deelgraaf van een procedure als apart geheel kunnen analyseren. Omdat gerichte graven makkelijker te bestuderen zijn indien ze een unieke start- en eindknoop hebben, zullen we aan elke procedure zulke knopen toekennen. De ingangsknoop komt overeen met het ingangspunt van een procedure: het punt waarnaar mogelijke procedure-oproepen verwijzen. De zogenaamde uitgangsknoop is een eerder abstracte knoop, die de unieke uitgang van een procedure voorstelt. Alle echte eindpunten van procedures (zijnde de terugkeerinstructies) worden met deze unieke eindknoop verbonden, en vanuit deze eindknoop vertrekken de zogenaamde terugkeerpijlen naar de basisblokken volgend op oproepblokken.

Om in de analyse van een oproeper de opgeroepen procedure als een atomair geheel te kunnen beschouwen, wordt er een pijl toegevoegd aan de graaf die het oproepblok met het daarop volgende blok (op het terugkeeradres) verbindt. De atomaire eigenschappen van de opgeroepen procedure worden dan geassocieerd met deze zogenaamde linkpijl tijdens de analyse van de oproeper.

Voorbeeld 2.4 Beschouw de code in figuur 2.7. Als we die gewoon opdelen in basisblokken en deze verbinden met pijlen voor alle mogelijke uitvoeringspaden, komen we aan een ICVG zoals in figuur 2.8. Alhoewel het stukje C code heel eenvoudig is, valt de CVG op door het grote aantal pijlen en de onduidelijkheid. Met de 2 oproeppijlen (van blokken in `g()` naar blokken uit `f()`) stem-

```
int a,b;

int f(int x)
{
   if (x<15) return x;
   else return -x;
}

g()
{
   a=f(10);
   b=f(20);
   return;
}
```

Figuur 2.7: Een voorbeeld in C code ter illustratie van de ICVG.

men vier terugkeerpijlen overeen (pijlen in omgekeerde richting). Uit de graaf is niet af te leiden welke terugkeerpijl bij welke oproeppijl hoort.

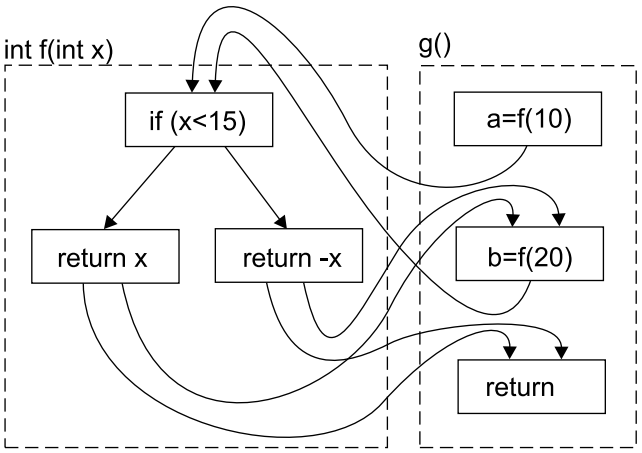
Om dit te vermijden, en het aantal pijlen in de graaf te verminderen, wordt daarom het uitgangsblok toegevoegd voor elke procedure. In figuur 2.9 is het aantal interprocedurale pijlen met 2 verminderd. In die figuur zijn in de procedure `g()` eveneens twee linkpijlen aangebracht. Ze maken het mogelijk om tijdens analyses de oproepen naar `f()` als een atomair gegeven te beschouwen en bedden in de graaf zelf het verband tussen oproeppijlen en terugkeerpijlen in.

□

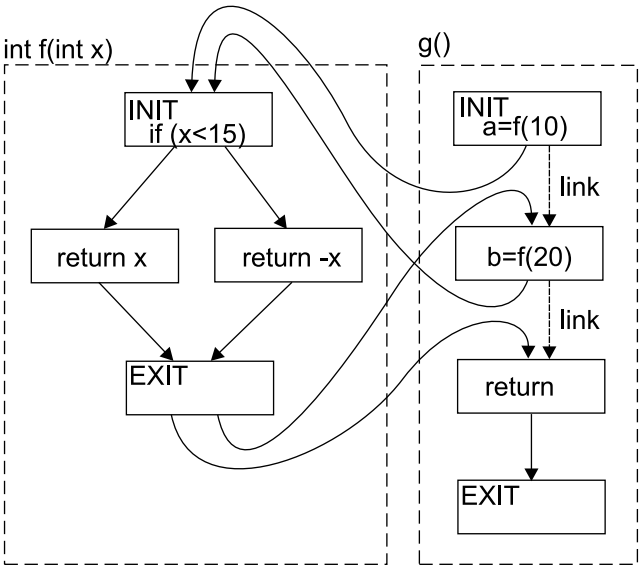
Merk op dat het gebruik van de uitgangsblokken geenszins de resultaten van analyses zal beïnvloeden. Waar de terugkeerpijlen eerst convergeerden in de oproeper, doen ze dit nu in de opgeroepen procedure voor het uitgangsblok. Omdat dit blok zelf geen dataverloopvergelijkingen heeft, wordt het resultaat van de analyses dus niet beïnvloed.

Interprocedurale sprongen

Waar het in programma's op broncodeniveau hoogst uitzonderlijk is dat er, naast de procedure-oproepen, interprocedurale sprongen zijn,



Figuur 2.8: De rechttoe-rechtaan ICVG van de C code uit figuur 2.7.



Figuur 2.9: De met uitgangsblokken en linkpijlen uitgebreide ICVG van de C code uit figuur 2.7.

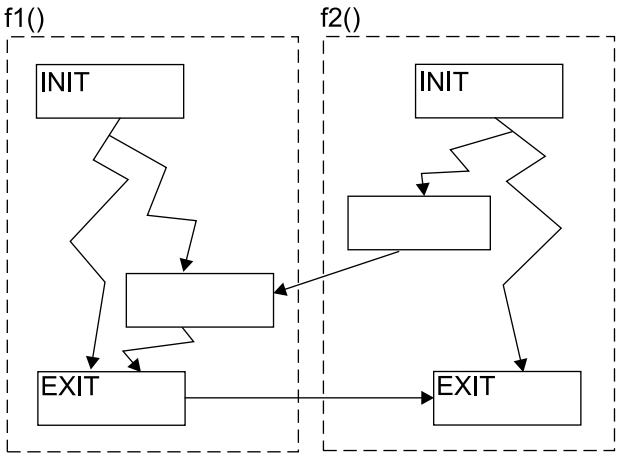
is dit op het assemblerniveau vaker het geval. In de handgecodeerde kernen van bibliotheekcode zijn nogal wat interprocedurale sprongen terug te vinden. Men spreekt van zogenaamde ontsnappende pijlen: afgezien van het feit dat ze interproceduraal zijn hoeft niks hen te onderscheiden van de intraprocedurale pijlen. Toch kan men zo'n pijlen niet zonder meer in de ICVG opnemen.

In principe behoort het stuk code dat bereikbaar is vanaf een ontsnappende pijl tot twee procedures, aangezien het kan uitgevoerd worden tijdens de uitvoering van de procedure waarin het stuk code is ondergebracht en tijdens de uitvoering van de procedure van waaruit de ontsnappende pijl vertrekt. Dit betekent dat het uitgangsblok dat volgt op de ontsnappende pijl ook tot beide procedures zou moeten behoren. Indien we zo'n blok echter als uitgangsblok van meer dan één procedure gaan beschouwen, hebben procedures per definitie weer meer dan één uitgangsblok, wat we precies trachtten te vermijden door de invoering van een uitgangsblok.

Gelukkig kunnen we dit eenvoudig vermijden door het toevoegen van een zogenaamde compenserende pijl: deze verbindt de twee uitgangsblokken met elkaar. Op deze wijze wordt correct gemodelleerd dat een terugkeerinstructie volgend op de ontsnappende pijl optreedt als een terugkeerinstructie van de procedure waaruit de ontsnappende pijl vertrekt.

Voorbeeld 2.5 Beschouw de graaf in figuur 2.10. De ontsnappende pijl van f2() naar f1() zorgt ervoor dat voor elke oproep naar f2() er twee terugkeerpijlen zouden moeten toegevoegd worden: een vanuit het uitgangsblok van f2() en een vanuit het uitgangsblok van f1(). Immers zal, nadat de ontsnappende pijl gevolgd is tijdens de uitvoering, de terugkeerinstructie op het einde van f1() hetzelfde effect hebben als een terugkeerinstructie op het einde van f2(). Dit zou betekenen dat een oproep naar f2() moet bekeken worden als een oproep van een procedure met verscheidene terugkeerpijlen.

Om dit te vermijden voegen we een compenserende pijl toe die de twee uitgangsblokken met elkaar verbindt. □



Figuur 2.10: Een voorbeeld van twee procedures met een ontsnappende pijl en de daarbijhorende compenserende pijl. De pijlen die uit verscheidene lijnstukken bestaan stellen niet verder gespecificeerd controleverloop voor.

Het onbekende: de helleknoop

In de code van figuur 2.3 komt een intermodulaire procedure-oproep voor, die vertaald is naar een indirecte procedure-oproep. Daarvoor is niet uit de gedisassembleerde code alleen af te leiden naar welke procedure zal gesprongen worden. Men moet daartoe ook de data bestuderen. In het voorbeeld is het makkelijk te achterhalen welke procedure opgeroepen wordt.

In andere gevallen zal dit moeilijker zijn. We denken hier bv. aan het gebruik van procedurewijzers, oproepen van virtuele methodes, het gebruik van sprongtabellen voor "switch" constructies, etc. Enerzijds is het in deze gevallen moeilijk, zoniet onmogelijk een beperkte groep van mogelijke bestemmingen te achterhalen naar waar zulke sprongen zich voordoen. Anderzijds is het voor die mogelijke bestemmingen ook moeilijk, zoniet onmogelijk om een beperkte groep mogelijke oproepers af te lijnen. In zulke gevallen moeten we daarom conservatief te werk gaan.

In de eerste plaats moeten we vastleggen om welke programmapunten het gaat. Er stellen zich twee vragen: van welke punten kennen we de opvolgers niet in de ICVG en van welke punten kennen we de voorgangers niet? De eerste vraag valt makkelijk te beantwoorden: alle punten waarop een indirecte controleverlooptransfer plaatsvindt. Het

antwoord op de tweede vraag luidt dan: alle punten waar één van die gevonden punten naar toe zou kunnen springen. Aangezien het om indirecte sprongen gaat zal het adres van de aldus bereikbare punten ergens in een register gestopt moeten worden: of het wordt berekend, of het wordt opgeladen. Beide gevallen zijn terug te vinden door de relocaties in het programma te bekijken: deze geven aan welke code-adressen in het programma gecodeerd zijn. Het zijn precies deze adressen die indirect bereikbaar zijn.

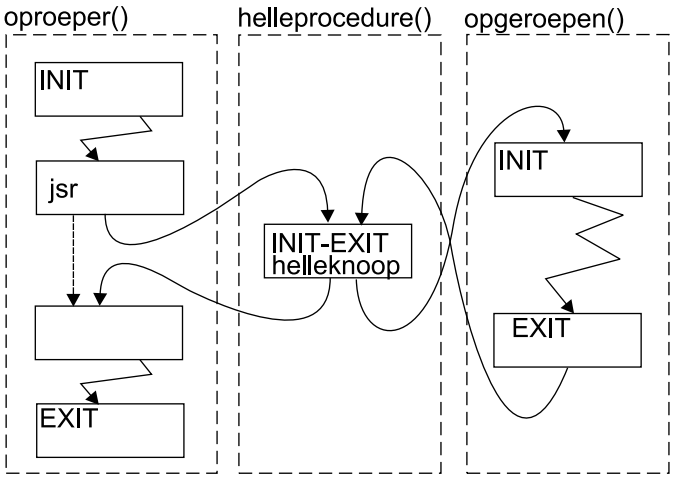
Nadat we deze punten bepaald hebben, moeten we in de analyses en transformaties rekening houden met hun onbekende voorgangers of opvolgers in de ICVG. Hiervoor bestaan twee mogelijkheden:

1. We annoteren de programmapunten en passen alle analyses en transformaties aan zodat ze met onbekende voorgangers en opvolgers rekening houden. Dit biedt de mogelijkheid aan alle analyses en transformaties om op een zeer specifieke manier met het onbekende om te gaan. Uiteraard moeten dan alle analyses en transformaties aangepast worden, wat omslachtig is en aanleiding kan geven tot veel programmeerfouten.

2. We modelleren "het onbekende" in de ICVG zelf, zodat de analyses en transformaties gewoon kunnen toegepast worden alsof er geen onbekenden waren, of slechts een minimale aanpassing vereisen. Deze optie vergt dan wel dat we aan de vereisten van alle analyses en transformaties kunnen tegemoetkomen op een uniforme manier.

Voor onze prototype compactor hebben we voor de tweede mogelijkheid geopteerd. Daartoe wordt aan de graaf een zogenaamde helleknoop toegevoegd [Muth99]. Deze knoop modelleert het onbekende door voor elke analyse een conservatief gedrag te vertonen. Voorbeelden van dit conservatief gedrag zijn:

- de helleknoop gebruikt de waarden in alle registers, m.a.w. alle registers zijn levend aan de ingang van de knoop;
- de helleknoop definieert alle registers, m.a.w. geen enkele waarde blijft bewaard;
- de helleknoop definieert de registers bovendien met niet-constante waarden, m.a.w. er is niks bekend over de waarden van de registers aan de uitgang van de knoop.



Figuur 2.11: Een voorbeeld van twee procedures met onbekend controleverloop dat gemodelleerd wordt aan de hand van een helleknoop en hellepijlen.

Om deze eigenschappen te laten gelden volstaat het meestal om aan het begin van een analyse deze eigenschappen vast te leggen: vele analyses worden immers geïmplementeerd aan de hand van het iteratief oplossing van een stelsel vergelijkingen en het volstaat dan de conservatieve eigenschappen als initiële waarden mee te geven en ze onveranderd te houden tijdens de analyse.

Voor punten waarvan de opvolgers niet bekend zijn in de ICVG volstaat het dan eenvoudigweg een pijl naar de helleknoop toe te voegen. Omgekeerd zal een pijl van de helleknoop naar een andere knoop de onbekende voorgangers van die knoop modelleren.

Voorbeeld 2.6 Beschouw de ICVG uit figuur 2.11. In de procedure oproeper() komt een indirecte procedure-oproep voor. Om de onbekende procedure die er opgeroepen wordt te modelleren voegen we een oproeppijl en een terugkeerpijl toe ter hoogte van de indirecte oproep. Van de procedure opgeroepen() ligt het adres ergens in de datasecties opgeslagen. De helleknoop zal in dit geval de onbekende oproepcontext modelleren, opnieuw door de gepaste oproep- en terugkeerpijl toe te voegen. De oproeppijl vertrekt nu uit de helleknoop en de terugkeerpijl keert ernaar terug. □

Op die manier kan men makkelijk en zonder al te veel analyse van

het programma een correcte ICVG opbouwen. In eerste instantie zal deze graaf uiterst conservatief zijn, maar zoals in sectie 2.5 besproken wordt, zal het mogelijk zijn hem nauwkeuriger te maken naargelang we meer over het programma te weten komen.

Merk op dat het gebruik van helleprocedures ook aangewezen zou zijn bij het compacteren van dynamisch gelinkte programma's: procedures in dynamisch gelinkte bibliotheken kunnen dan d.m.v. deze procedures gemodelleerd worden.

Anomalieën van het type setjmp() en longjmp()

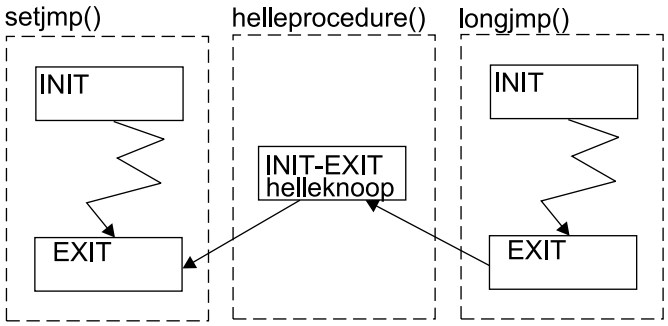
Bij de bespreking van interproceduraal controleverloop hogerop werd enkel relatief eenvoudig interproceduraal verloop in beschouwing genomen, zeker wat betreft de procedure-oproep en de bijhorende terugkeer.

Soms kan het gedrag van procedure-oproepen echter anders zijn, zoals het geval is met de setjmp()/longjmp() procedures uit de standaard C bibliotheek. Bij een oproep van setjmp() wordt de uitvoeringscontext van dat moment bewaard in een buffer. Bij een latere oproep van longjmp() zal die toestand hersteld worden en wordt het programma opnieuw verder uitgevoerd vanaf het punt volgend op de oproep van setjmp(). Een oproep van longjmp() keert dus terug naar het terugkeeradres van een oproep naar setjmp(). Om dit te modelleren maken we gebruik van enkele extra pijlen van en naar de helleknoop, zoals in figuur 2.12 wordt geïllustreerd. De helleknoop vervult hierbij een dubbele functie: vanuit longjmp() gezien modelleert hij alle programmapunten waar setjmp() opgeroepen wordt. De pijl naar het uitgangsblok van setjmp() modelleert dan weer dat dit blok bereikbaar is van op elke plek waar longjmp() opgeroepen wordt.

Om zulke pijlen toe te kunnen voegen moeten we natuurlijk detecteren welke procedures dit gedrag kunnen vertonen. Daartoe gebruiken we volgende heuristieken:

- Indien een procedure het adres waarnaar het terugkeert ergens anders opslaat in het geheugen dan op de daartoe bij conventie voorbehouden plaats in het stapelvenster, is het een procedure van het type setjmp().

- Indien een procedure terugkeert met een terugkeeradres dat van ergens anders opgeladen wordt dan van de bij conventie vastge-



Figuur 2.12: Compenserende pijlen voor het afwijkende gedrag van setjmp() en longjmp().

legde plaats in het stapelvenster is het een procedure van het type longjmp().

Deze heuristieken zijn makkelijk conservatief te implementeren. In theorie kunnen zij al te conservatief zijn, in de zin dat ze foutief een aantal procedures zouden kunnen herkennen als zijnde van één van bovenvermelde types. Dit leidt echter nooit tot een foute compactie. In de praktijk hebben we overigens vastgesteld dat deze eenvoudige heuristiek geen enkele procedure foutief aanduidt.

2.4.3 Opbouwen van de ICVG

Zoals reeds gesteld gaan we ervan uit dat er aparte codesecties zijn, waarin geen data gestockeerd is. Deze secties (text, init en fini) kunnen na het linken evengoed als één sectie beschouwd worden aangezien er dan geen functioneel verschil meer is.

Het opbouwen van de ICVG van het programma verloopt dan in een aantal stappen.

Disassembleren

Op een architectuur met instructies met variabele instructielengte is het decoderen (disassembleren) van instructies ietwat moeilijker, omdat men gezien de toevoeging van alignementsbytes vaak niet weet waar een instructie begint. Het opbouwen van de controleverloopgraaf moet dan vaak samen gedaan worden met het disassembleren. In een

architectuur met instructies van vaste lengte is het decoderen van de instructies kinderspel.

Markeren van procedures

Na het disassembleren beschikken we over een lijst van instructies. Deze zal nu opgesplitst worden in procedures en basisblokken. Hierbij nemen we aan dat procedures uit één ononderbrokensequentie instructies bestaan. Indien dit niet zo zou zijn, verloopt het opbouwen van de ICVG iets complexer, aangezien het groeperen per procedure dan pas samen met het opbouwen van de eigenlijke graaf, d.w.z. het toevoegen van pijlen, kan gebeuren. Fundamenteel is dit echter niet moeilijker.

De instructiesequentie wordt overlopen en hierbij worden alle instructies die de bestemming zijn van een directe procedure-oproep gemarkeerd als het begin van een procedure. Tevens wordt het beginpunt van het programma gemarkeerd. De locatie van dit punt staat beschreven in de koppen.

Markeren van basisblokken

Het standaard algoritme voor de opdeling van een programma in basisblokken [Aho86] wordt gebruikt om het programma verder in basisblokken in te delen. Daartoe wordt in deze stap een eerste verzameling *leiders* opgebouwd, d.w.z. de instructies die een basisblok beginnen. De regels daartoe zijn:

1. Het beginpunt van het programma is een leider.
2. Elke instructie die de bestemming is van een controletransfer is een leider.
3. Elke instructie die volgt op een controletransfer is een leider.

Markeren van reloceerbare adressen

Vervolgens worden de instructies op reloceerbare adressen gemarkeerd als ingangspunten van procedures en als leiders. Dit zijn:

- de eerste instructie van de codesecties;

- alle code-adressen die door de symboolinformatie als het begin van een procedure worden aangeduid;
- alle absolute code-adressen die in de globale adrestabel of datasecties voorkomen en ofwel het begin van een procedure zijn volgens de symboolinformatie, ofwel net voorbij de proloog van een procedure liggen. Afhankelijk van waaruit naar een procedure gesprongen wordt (inter- of intramodulair), moet de proloog immers al dan niet uitgevoerd worden. De reden is dat intramodulaire procedure-oproepen zich niet noodzakelijk aan de oproepconventies moeten houden. Zowel het begin van de proloog als het punt er vlak voorbij kunnen dus als procedure-ingangen beschouwd worden.

Absolute code-adressen die niet overeenstemmen met het begin van een procedure worden wel gemarkeerd als leider, maar niet als beginpunt van een procedure. Zulke punten zullen tijdens latere analyses steeds even conservatief of conservatiever beschouwd worden dan procedure-ingangspunten, waardoor de analyses en transformaties zelf ook conservatief blijven.

Naast de absolute adressen die opgeslagen liggen in de globale adrestabel of andere datasecties, liggen er ook relatieve adressen in opgeslagen.[1] Zo zal men om plaats te sparen sprongtabellen vaak vullen met relatieve i.p.v. met absolute adressen. Dit soort sprongtabellen wordt typisch gebruikt om constructies als een switch-constructie uit de C taal of analoge operaties te implementeren. De waarde van de veranderlijke waarop de switch werkt, wordt dan (onrechtstreeks) gebruikt om een adrestabel te indexeren waaruit een relatief adres geladen wordt om te springen. Met de relatieve adressen in zulke tabel corresponderen relatieve relocaties. Voor deze relocaties worden de instructies die ermee bereikt kunnen worden, gemarkeerd als ingangspunt van een basisblok dat bereikbaar is vanuit onbekende voorgangers, maar niet als ingangspunt van een procedure. Opnieuw zullen we later deze punten conservatiever beschouwen dan de ingangspunten van procedures.

Voor al deze punten wordt ook bijgehouden welke soort relocatie aan de basis lag voor het markeren van de instructies. Deze informatie

[1] Aangezien deze relatieve adressen relatief zijn t.o.v. de globale wijzer, kunnen we er heel eenvoudig de absolute adressen waarnaar ze wijzen uit afleiden. Dit levert geen enkel probleem op, alhoewel het strikt genomen om berekeningen op code-adressen gaat.

is immers van belang voor de latere verfijning van het programma en voor de correcte uitvoering van een aantal transformaties.

Aanmaken van procedures en basisblokken

Eerst en vooral wordt de pseudo-procedure *hel* aangemaakt. Deze bevat één lege pseudo-knoop die de ingangs- en uitgangsknoop van de procedure is.

Daarna worden alle instructies overlopen: per gemarkeerd ingangspunt van een procedure wordt een procedure aangemaakt en per gemarkeerd ingangspunt van een basisblok wordt een basisblok aangemaakt.

De aldus gevormde basisblokken vormen de knopen van de ICVG.

Aanmaken van de pijlen van de ICVG

Eén voor één worden alle basisblokken overlopen. Voor elk basisblok worden volgende pijlen aangemaakt:

1. Als het basisblok een reloceerbare eerste instructie heeft, wordt er een zogenaamde hellepijl naar aangemaakt vanuit de helleknoop.

2. Vervolgens wordt er naar de laatste instructie van het basisblok gekeken. Afhankelijk van het type van deze instructie worden er pijlen aan de ICVG toegevoegd van een bepaald type:

   Niet-conditionele directe sprong Er wordt een normale pijl aangelegd naar de bestemming. Het adres daarvan wordt afgeleid uit de letterlijke afstand die in de instructie is gecodeerd.

   Conditionele directe sprong Er worden twee gewone pijlen aangelegd. Eén naar het volgende basisblok in de code (d.w.z. het zogenaamde doorvalpad) en één naar de bestemming van de sprong.

   Directe procedure-oproep Er wordt een oproeppijl aangemaakt naar het bestemmingsblok. Dit wordt opnieuw afgeleid uit de letterlijk afstand in de instructie. Tevens wordt er een linkpijl aangemaakt naar het blok dat volgt op het oproepblok.

Indirecte procedure-oproep Identiek als bij een directe procedure-oproep, maar de bestemming is in dit geval de helleknoop.

Indirecte niet-conditionele sprong Identiek als bij een directe niet-conditionele sprong, maar de opvolger is nu de helleknoop.

Terugkeerinstructie Er wordt een zogenaamde uitgangspijl aangemaakt naar het uitgangsblok van de procedure horende bij de terugkeerinstructie.

Een systeemoproep Dit is identiek aan een indirecte procedureoproep. Voor de *halt* systeemoproep worden er geen pijlen aangemaakt, aangezien er nooit verdere instructies uitgevoerd zullen worden.

De overige types Er wordt een normale pijl aangemaakt naar het volgende blok, het doorvalblok.

Samen met het aanmaken van deze pijlen worden er een aantal pijlen automatisch aangemaakt: voor elke interprocedurale pijl uit bovenstaand rijtje, wordt er een extra pijl aangemaakt. Is de interprocedurale pijl een oproeppijl, dan wordt automatisch een corresponderende terugkeerpijl aangemaakt. Deze pijl vertrekt uit het uitgangsblok van de opgeroepen procedure en gaat naar het blok volgend op het oproepblok, zoals besproken in sectie 2.4.2. Is de interprocedurale pijl van gelijk welk ander type, dan wordt er een bijhorende compenserende pijl aangemaakt, die de uitgangsblokken van beide procedures met elkaar verbindt zoals in paragraaf 2.4.2 werd besproken.

De compenserende pijlen voor setjmp() en longjmp() en soortgelijke procedures worden toegevoegd als er procedures van zulk type ontdekt worden. Daartoe worden de heuristieken uit paragraaf 2.4.2 gebruikt.

2.5 Verfijning van de ICVG

In deze sectie worden een aantal verfijningen van de ICVG besproken [Debr00, DS00]. Met name wordt besproken op welke manier conservatieve hellepijlen uit de voorstelling kunnen verwijderd worden of vervangen door meer accurate en daarom minder conservatieve pijlen.

Omdat deze verfijningen in zeer sterke mate afhangen van analyses die verder aan bod komen, zullen ze pas later geëvalueerd worden.

Merk overigens op dat er een heel sterke wisselwerking is tussen deze verfijningen en allerlei analyses die later aan bod komen: de analyses ondersteunen deze verfijningen en deze verfijningen leiden op hun beurt tot nauwkeuriger analyses. Het verfijnen van de ICVG zal dan ook afwisselend en iteratief met het analyseren van het programma uitgevoerd moeten worden.

2.5.1 Het onbekende opnieuw bekeken

Zoals in het vorige hoofdstuk besproken modelleren we onbekend controleverloop met een helleknoop. Deze knoop krijgt daartoe een aantal zeer conservatieve eigenschappen toegewezen. Deze eigenschappen zijn zo conservatief gekozen dat ze in alle omstandigheden correcte analyseresultaten met zich meebrengen. Vaak is het echter zo dat we toch enkele minder conservatieve eigenschappen van onbekend controleverloop kunnen veronderstellen.

Indirecte procedure-oproepen Een typisch voorbeeld daarvan is de indirecte procedure-oproep. Alhoewel we bij de initiële opbouw van de ICVG geen idee hebben van de bestemming van deze oproep, weten we wel met zekerheid dat dit een procedure zal zijn. Omdat ook de vertaler in zo'n geval enkel deze kennis heeft, gaat hij er bij het vertalen van de oproeper van uit dat de oproepconventies zullen nageleefd worden door de opgeroepen procedure en implementeert hij de oproep ook volgens de oproepconventies.

Omgekeerd zal een procedure met mogelijke onbekende oproepcontexten conform de oproepconventies geïmplementeerd zijn, en gaat de vertaler er bij het genereren van zulk een procedure van uit dat oproepers ook conform de oproepconventies zullen werken.

Een aantal van deze oproepconventies kan echter niet accuraat gemodelleerd worden aan de hand van één enkele helleknoop. Het probleem komt voort uit de dubbele functie die de helleknoop moet vervullen: hij moet zowel onbekende oproepende als onbekende opgeroepen procedures modelleren. Om deze functies op te splitsen en aldus een nauwkeuriger modellering mogelijk te maken, worden daarom twee nieuwe pseudo-knopen ingevoerd: de oproepers- en de opgeroepenenhelleknoop, met bijhorende pseudo-procedures. De oproepershelleknoop modelleert een onbekende oproeper, terwijl de opgeroepenenhelleknoop een onbekende opgeroepen procedure modelleert.

Voorbeeld 2.7 Beschouw de graaf in figuur 2.11 die eerder in voorbeeld 2.6 werd besproken. Nemen we als voorbeeld van dataanalyse de levensduuranalyse. Een waarde (in een register) wordt als levend beschouwd op een bepaalde plek in het programma indien die waarde nog gebruikt kan worden verder in het programma. Levensduuranalyse tracht te achterhalen welke registers op welke plaatsen levende waarden bevatten.

Voor de helleknoop moeten we daartoe vastleggen welke registers levende waarden bevatten bij het binnenkomen in de knoop. Aan de uitgang van alle knopen met een pijl naar de helleknoop bevatten dan diezelfde registers levende waarden.

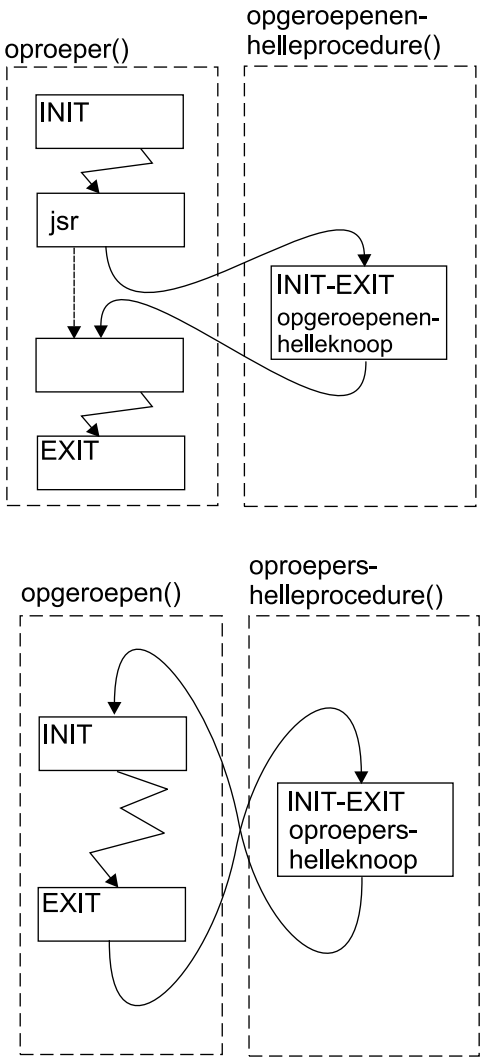
In het voorbeeld zijn er twee pijlen naar de helleknoop: een oproeppijl vertrekkend vanuit oproeper() en een terugkeerpijl vertrekkend vanuit de uitgangsknoop van opgeroepen().

Veronderstellen we dat de oproepconventie stelt dat er drie argumentregisters zijn en dat die vooraf te bewaren zijn, m.a.w. de oproeper is verantwoordelijk voor het tijdelijk opslaan van de waarden indien hij deze waarden na de procedure-oproep nog wenst te gebruiken. De vraag stelt zich dan of deze argumentregisters levende waarden bevatten aan de ingang van de helleknoop? Als men de helleknoop bekijkt als opgeroepen procedure, dan is het antwoord ja: de argumenten voor een procedure moeten per definitie levend verondersteld worden aan de ingang van die procedure.

Bekijkt men de helleknoop als de knoop die volgt op de uitgangsknoop, dan is het antwoord echter neen: omdat de argumentregisters vooraf te bewaren zijn, kan opgeroepen() de waarden in de argumentregisters vrij overschrijven. De oproeper weet dat dit kan gebeurd zijn en hij zal de waarden in die registers dus niet gebruiken.

Deze registers als levend bestempelen in de helleknoop resulteert dus in een te conservatieve veronderstelling wat betreft de procedure opgeroepen(). In figuur 2.13 echter kunnen deze eigenschappen wel correct gemodelleerd worden, aangezien beide pseudo-knopen nu verschillende waarden kunnen aannemen tijdens de analyse.

□



Figuur 2.13: Een indirecte procedure-oproep gemodelleerd met verscheidene helleknopen.

Systeemoproepen Net als voor procedure-oproepen zijn er ook voor de zogenaamde systeemoproepen allerlei conventies uitgeschreven die sterk verschillen van de conventies voor gewone procedure-oproepen. Daarom wordt er naar analogie van de opgeroepenenhelleknoop tevens een systeemhelleknoop ingevoerd.

2.5.2 Verfijnen van indirecte procedure-oproepen

Zoals uit voorbeeld 2.1 gebleken is, zullen heel wat directe intermodulaire procedure-oproepen in de broncode geïmplementeerd worden a.d.h.v. een indirecte oproep waarvoor het adres opgeladen wordt uit de globale adrestabel. In zulke gevallen is het niet moeilijk te achterhalen wat de opgeroepen procedure is.

Constantenpropagatie brengt aan het licht welke constante opgeladen wordt, en hiermee kan de opgeroepen procedure achterhaald worden. In de graaf wordt de oproeppijl naar de opgeroepenenhelleknoop vervangen door een oproeppijl naar de opgeroepen functie. Dit is een triviale verfijning van de graaf.

Indien het programma klein genoeg is, zodat de afstand tussen opgeroepen procedure en oproepende instructie zeker klein genoeg zal zijn om in een instructie te coderen als letterlijke afstand, wordt zo'n indirecte oproep bovendien vervangen door een directe oproep. Ten gevolge daarvan wordt het opladen van het procedure-adres vaak loos, en kan dit opladen verwijderd worden uit het programma.

2.5.3 Onbekende oproepcontexten

Bovenstaande vervanging van oproeppijlen en van indirecte sprongen is slechts de helft van het verhaal. Met het converteren van indirecte naar directe oproepen wordt ook het opladen van de corresponderende procedure-adressen overbodig. Indien dit voor alle oproepers van een procedure het geval is, worden de opgeslagen adressen dood: ze worden nergens meer gebruikt in het programma.

Voor deze opgeslagen adressen hadden we in de initiële graaf een oproeppijl vanuit de oproepershelleknoop toegevoegd. Als alle adressen van een procedure dood worden, betekent dit dat de ermee corresponderende pijl eveneens dood wordt: hij modelleert een uit het programma verwijderde indirecte transfer. In dat geval kunnen we dus ook die pijl verwijderen uit het programma.

Het detecteren van dode data wordt uitvoerig besproken in sectie 3.3. Eens de dode data verwijderd is uit het programma gaan we als volgt te werk:

1. markeer de procedures waarvan adressen opgeslagen liggen in de levende data;

2. verwijder de oproeppijlen vanuit de oproepershelleknoop naar de ingangsknopen van alle niet-gemarkeerde procedures.

Een tweede manier om procedure-oproepen vanuit de oproepershelleknoop te verwijderen is het gebruik van de relocatie-informatie. Bij elke instructie die een adres oplaadt uit de globale adrestabel geeft een relocatie aan waarom deze waarde wordt opgeladen. Hieruit kunnen we achterhalen of de opgeladen adressen enkel door indirecte procedure-oproepen gebruikt zullen worden of ook nog door andere instructies. Stel dat het adres van een procedure enkel in de globale adrestabel ligt opgeslagen. Als het na het opladen enkel door procedure-oproepen wordt gebruikt, kunnen we ervan uitgaan dat constantenpropagatie ervoor zal zorgen dat voor die oproepen een oproeppijl naar de opgeroepen procedure zal worden aangemaakt. Zelfs al blijft het adres van die opgeroepen procedure levend in de globale adrestabel (omdat de indirecte oproepen niet in directe oproepen kunnen omgezet worden), dan nog mag de oproeppijl van de oproepershelleknoop worden verwijderd na de constantenpropagatie, aangezien voor alle oproepers van de opgeroepen procedure een oproeppijl zal aangemaakt zijn. Om deze tweede manier te implementeren gaan we als volgt te werk:

- Voor de aanvang van de codecompactie worden alle procedures gemarkeerd waarvan het adres enkel in de globale adrestabel voorkomt en niet in de overige datasecties, en waarvan dit adres enkel opgeladen wordt om er indirecte procedure-oproepen mee uit te voeren. Daartoe overlopen we alle instructies die adressen opladen uit de globale adrestabel en kijken we naar de bijhorende relocaties.

- Eens we van al deze indirecte procedure-oproepen de opgeroepen procedure kennen (en de nodige oproeppijlen aangemaakt zijn in de graaf), verwijderen we de oproepen vanuit de oproepershelleprocedure naar de gemarkeerde procedures.

Deze heel eenvoudige verfijningen van de interne voorstelling van het programma blijken van uitzonderlijk belang te zijn voor de compactie van programma's. Voor een heel groot aantal procedures (met name de procedures die oproepbaar zijn van buiten hun eigen module), wordt er immers een oproep verwijderd uit de graaf. Deze oproep vanuit de oproepershelleknoop kon verschillende optimalisaties in de weg staan. In volgorde van aflopend belang zijn dit:

- Als de oproeppijl vanuit de oproepershelleknoop de enige oproep naar een procedure is, en deze oproep kan met bovenstaand algoritme verwijderd worden, dan kan ook die procedure verwijderd worden, aangezien er geen oproepers meer over blijven.
- Als er naast deze oproeppijl slechts één andere oproeppijl overblijft, kan de opgeroepen procedure gesubstitueerd worden, aangezien hier nu geen codeduplicatie meer aan te pas komt en het programma dus per definitie kleiner wordt bij het uitvoeren van de proceduresubstitutie. Bovendien kan de procedure dan geoptimaliseerd worden naar die ene context waarin ze uitgevoerd wordt.
- In de overige gevallen verdwijnt er in ieder geval een heel conservatieve oproepcontext voor de betreffende procedure als de oproeppijl verwijderd wordt. Soms kan dit aanleiding geven tot verdere optimalisatie.

Eigenaardig genoeg is het echter in sommige gevallen nuttig om oproeppijlen vanuit de oproepersknoop in de ICVG te laten staan. De reden hiervoor is dat deze pijl ervoor zorgt dat de oproepconventies gerespecteerd zullen blijven[2]: net omdat ook de compactor de uitvoeringscontext niet kent, kan hij niet raken aan die oproepconventies. Daaruit volgt dat voor procedures die bereikbaar zijn uit de oproepershelleknoop, de oproepconventies kunnen verondersteld worden.

Zo kan men ervan uitgaan dat de vooraf te bewaren registers niet van inhoud veranderen over een oproep naar zo'n procedure heen, ook al kan men dit niet afleiden uit de analyse van die procedure, bv. omdat het stapelgedrag niet accuraat genoeg kan geanalyseerd worden.

[2]In theorie kan het zo zijn dat de vertaler ergens het adres van een procedure alloceert, terwijl hij toch afziet van de naleving van oproepconventies omdat hij alle oproepers van de procedure kent. In zulke gevallen zal de procedure niet als proceduresymbool geëxporteerd worden uit de module. Zulke gevallen kunnen dus makkelijk gedetecteerd worden.

Omdat deze eigenschappen voornamelijk voor levensduuranalyse een rol spelen, zullen we de criteria om oproeppijlen te verwijderen in die context bespreken. Zie daarvoor sectie 3.1.

2.5.4 Verfijnen van indirecte sprongen

Naast indirecte procedure-oproepen zijn er ook indirecte sprongen die het adres van de bestemming uit een register halen. Ze verschillen van elkaar doordat de laatste geen terugkeeradres op de stapel zetten.

Een typisch gebruik van zo'n indirecte spronginstructie is de implementatie van een switch-constructie uit de C taal of soortgelijke programmaconstructies in andere talen. In zo'n constructie bepaalt een argument welk stukje code zal uitgevoerd worden. Vaak kan men dit argument herleiden tot een numerieke waarde. Deze waarde kan dan gebruikt worden om een tabel met adressen te indexeren en er het adres van het uit te voeren codefragment uit op te laden. Naar dit adres wordt dan gesprongen. Omdat een switch-constructie normaal gezien niet over proceduregrenzen heen gaat, volstaat het relatieve adressen op te slaan in de adrestabel i.p.v. absolute adressen. Zo kan men plaats sparen.

Voor de implementatie van zo'n switch-constructie worden typisch volgende berekeningen gegenereerd door vertalers:

Normalisatie van de index Typisch liggen de waarden van het argument van de switch-constructie waarvoor code uitgevoerd moet worden in een interval. Dit interval wordt verschoven, zodanig dat de indexering begint met waarde 0. Deze normalisatie kan echter ingewikkelder vormen aannemen, zoals het veranderen van teken, het selecteren van bepaalde bits uit het argument, etc. Voor deze normalisatie wordt typisch gebruik gemaakt van optel-, aftrek- en schuifoperaties.

Controleren van de bovengrens Er wordt gecontroleerd of het (genormaliseerde) argument binnen de toegestane grenzen valt, zodat er niet buiten de adrestabel zal gelezen worden. Dit gebeurt typisch aan de hand van een vergelijkingsoperatie en een conditionele sprong, die ofwel uit de switch-constructie springt ofwel de code horend bij de verstekwaarde zal uitvoeren.

Opladen Het absolute of relatieve adres wordt uit de adrestabel opgeladen. Indien het een relatief adres betreft, wordt dit opgeteld bij

het basisregister. Hiervoor volstaan een leesoperatie en eventueel een optelling.

De sprong Een indirecte sprong springt naar het adres dat opgeladen of berekend is.

Hierbij dient opgemerkt dat met name de eerste twee onderdelen van zo'n implementatie vrij veel variatie kunnen vertonen, zowel wat betreft de mogelijke berekeningen zelf als wat betreft de volgorde waarin ze uitgevoerd worden. Ondanks de vrijheid die vertalers hiervoor genieten hebben we vastgesteld dat, in ieder geval voor het doelplatform van onze prototype compactor, de verschillende vertalers nagenoeg identieke sequenties genereren voor soortgelijke structuren.

Indien men de bestemmingen horende bij zo'n sprong wenst te weten te komen, is het nodig de plaats en grootte van de adrestabel te achterhalen. De plaats ervan kan middels het basisadres van de leesinstructie teruggevonden worden. Dit basisadres wordt normaliter gevonden tijdens een constantenpropagatie. De grootte van de tabel kan men afleiden uit de normalisatieberekeningen en de controle van de bovengrens.

Voor beide gegevens is het nodig een aantal instructies te identificeren die aan de sprong voorafgaan. Daartoe nemen we de zogenaamde programmasnede [Weis84] van de sprong, m.a.w. tot op zekere hoogte die instructies in het programma die voorafgaan aan de sprong en er de uitkomst van bepalen. Deze programmasnede wordt dan middels patroonherkenning vergeleken met veel gebruikte instructiesequenties waarvan we uit de letterlijke operandi de grootte van de tabel kunnen afleiden.

Dit is niet triviaal omdat we rekening moeten houden met:

Codeduplicatie Het kan zijn dat bepaalde stukken uit het instructiesequentie gedupliceerd zijn om uitvoering vanuit verschillende voorgangers efficiënter te laten verlopen.

Inroostering De programmasnede horende bij de sprong kan ingeroosterd zijn tussen allerlei andere berekeningen waaronder procedure-oproepen, of andere conditionele sprongen (die verward kunnen worden met de controle van de bovengrens). Tevens kunnen met name de instructies voor normalisatie en grenscontrole herordend zijn en kunnen tussenresultaten van de berekeningen tijdelijk op de stapel bewaard worden.

Optimalisaties De vertaler kan op de normalisatie-instructies eveneens kijkgatoptimalisaties uitgevoerd hebben.

Dit alles maakt deze patroonherkenning verre van triviaal. Toch slagen we erin van een heel groot aantal van de indirecte sprongen de mogelijke bestemmingen te achterhalen. Dit is belangrijk ter verfijning van de ICVG.

Enerzijds wordt in zo'n geval immers de pijl naar de helleknoop vervangen door een reeks pijlen naar de opvolgers van de indirecte sprong in de ICVG. Dit is belangrijk voor bv. de levensduuranalyse ter hoogte van de indirecte sprong.

Anderzijds stelt het ons in staat om, eens alle indirecte sprongen in een procedure op bovenstaande manier geanalyseerd zijn of verwijderd zijn uit het programma, de pijlen van de helleknoop naar de bestemmingen van de indirecte sprongen in die procedure te verwijderen. Dit is slechts een heuristiek, die in theorie aanleiding kan geven tot incorrecte programma's: het zou immers kunnen dat er een nog niet geïdentificeerde indirecte sprong naar zo'n bestemming bestaat. In de uitgebreide set programma's die met ons prototype gecompacteerd werden is dit echter nooit voorgevallen. Dit hoeft ook niet te verbazen: daar waar er indirecte sprongen voorkomen die we niet met onze methode kunnen analyseren, betreft het vooral manueel geschreven assemblercode in bibliotheken. Deze procedures zullen dan typisch geen door vertalers gegenereerde implementaties van switch-constructies bevatten.

2.5.5 De procedure exit()

Procedures zoals exit() uit de C bibliotheek hebben een speciale eigenschap: er wordt nooit uit teruggekeerd, aangezien ze bedoeld zijn om de uitvoering van een programma te beëindigen.

Voorbeeld 2.8 Beschouw het korte stukje C code in figuur 2.14. De CVG horende bij de procedure veilig_deref is weergegeven in figuur 2.15. Merk op dat de interprocedurale oproep- en terugkeerpijlen niet opgenomen zijn in de figuur.

In de graaf is er een linkpijl te zien die correspondeert met de oproep naar exit(). Deze pijl modelleert een uitvoeringspad gaande van de oproep, door de procedure exit() naar de operatie return

```
int veilig_deref(int* x)
{
   if (!x)
   {
      printf("Fout wijzer x == 0");
      exit(-1);
   }
   return *x;
}
```

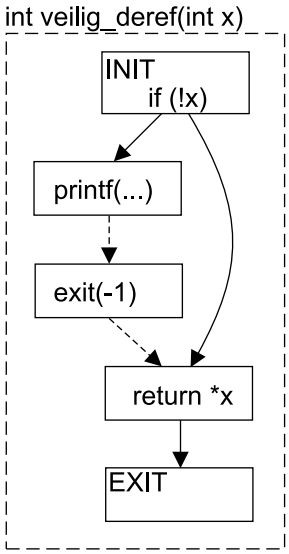
Figuur 2.14: Broncode met een oproep naar exit().

*x. Dit verschilt niet met andere linkpijlen. Dit specifieke uitvoeringspad is in de praktijk echter niet realiseerbaar (m.a.w. kan nooit uitgevoerd worden), omdat er nooit zal teruggekeerd worden uit de procedure exit().

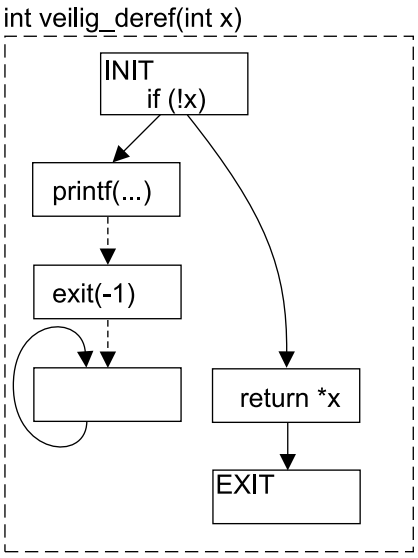
Om de aldus niet-realiseerbare paden niet te laten interfereren met realiseerbare paden tijdens de analyses worden zulke niet-realiseerbare linkpijlen omgezet in linkpijlen naar lege knopen die enkel zichzelf als opvolger hebben. Dit is te zien in figuur 2.16. □

Dit voorbeeld geeft aan hoe we kunnen vermijden dat niet-realiseerbare paden onze analyses in negatieve zin beïnvloeden. Dit kan in belangrijke mate gebeuren als een oproep naar een exit()-achtige procedure door de vertaler helemaal achteraan een procedure geplaatst wordt. Als de vertaler na die oproep geen terugkeerinstructie meer plaatst, wat frequent het geval is omdat ook de vertaler weet dat er niet teruggekeerd wordt, zullen de terugkeerpijl en de linkpijl horende bij de oproep naar de exit()-achtige procedure wijzen naar de volgende procedure in het programma. Het zijn als het ware doorvalpijlen geworden, die nodeloos extra uitvoeringscontexten aan die volgende procedure toevoegen.

De vraag stelt zich dan hoe we deze paden kunnen detecteren. De procedures als exit() van waaruit niet kan teruggekeerd worden kunnen ontdekt worden aan de hand van de systeemoproep die het afsluiten van het programma aangeeft. Indien deze systeemoproep het uitgangsblok van de procedure waarin het voorkomt domineert en het



Figuur 2.15: De graaf horende bij de code uit figuur 2.14.



Figuur 2.16: De verfijnde graaf horende bij de code uit figuur 2.14.

ingangsblok postdomineert[3], kan uit deze procedure niet meer teruggekeerd worden.

Eens zulke procedures gemarkeerd zijn, kan men iteratief verder zoeken doorheen het programma: in de plaats van naar de systeemoproepen zelf, gaan we nu op zoek naar procedure-oproepen naar als niet-terugkerend gemarkeerde procedures. Op deze manier worden exit()-achtige procedures transitief afgesloten.

2.6 Statisch gealloceerde data

Tot nu toe hebben we het over de code en haar interne voorstelling gehad. Daar waar we de code op verschillende niveaus willen kunnen analyseren en transformeren, is dit ook voor de data het geval. De te onderscheiden niveaus zijn secties, datablokken en individuele bytes.

Individuele bytes De statisch gealloceerde data van een programma is beschikbaar tijdens de compactie. Van die data is het nuttig enkele eigenschappen te kennen.

Als gegevens niet overschreven kunnen worden tijdens de uitvoering van het programma (m.a.w. ze zijn constant), dan kunnen we de waarden van die gegevens gebruiken bij het optimaliseren of compacteren van het programma.

Als gegevens niet opgeladen worden tijdens de uitvoering van een programma, kunnen we ze als dood beschouwen en eventueel uit het programma verwijderen. Gegevens als dood beschouwen kan op zich al nuttig zijn, zo kunnen we bv. hellepijlen verwijderen als we weten dat corresponderende code-adressen nooit uit de datasecties opgeladen worden.

We kunnen echter niet zomaar dode data verwijderen uit een programma. Dode data verwijderen betekent immers het verband tussen de locaties van andere data wijzigen. Dit kunnen we enkel doen als er met zekerheid geen code is die, zonder relocaties, de oude verbanden hard in zich gecodeerd heeft.

[3]Een basisblok A domineert een blok B als een uitvoering van blok B steeds voorafgegaan wordt door een uitvoering van blok A (met eventueel nog andere blokken er tussenin). A is dan een dominator van B. Omgekeerd postdomineert B A als na elke uitvoering van A ook zeker B zal uitgevoerd worden. B is dan een postdominator van A.

Datablokken Behoudens analyses die de in de vorige alinea vermelde vereiste verifiëren, kunnen we er enkel van uitgaan dat er geen blokoverschrijdende verbanden tussen datalocaties hard in het programma gecodeerd zullen zijn. De granulariteit waarmee we dus data uit het programma kunnen verwijderen (en herordenen indien gewenst) is die van het datablok. Daarom worden de individuele dataelementen (bytes) per datablok gegroepeerd. Zoals later zal blijken, zullen ook een aantal eigenschappen van de data op het blokniveau gebruikt worden.

Overigens dient opgemerkt te worden dat men niet enkel onze vorm van datablokken (zijnde secties uit objectbestanden) kan beschouwen. Elke vorm waarvan men de zekerheid heeft dat er geen blokgrensoverschrijdende berekeningen op adressen in het programma aanwezig zijn, voldoet.

Een voorbeeld van zo'n andere vorm is de globale adrestabel van een programma. Precies omwille van zijn eigenschap dat elk element ervan enkel rechtstreeks vanaf de `gp` kan gelezen worden, kan men ervan uitgaan dat er geen berekeningen op adressen wijzend naar de tabel uitgevoerd zullen worden: deze zouden immers enkel een verspillend gebruik van registers met zich meebrengen. In concreto betekent dit dat elk adres in de globale adrestabel een apart datablok is.

Secties Omdat er een aantal algemene eigenschappen per sectie vastgelegd zijn in het finale programma, zoals het al dan niet beschrijfbaar zijn van geheugenlocaties in een sectie, is het nuttig de datablokken ook nog eens per sectie te bundelen. Dit is bovendien handig omdat een aantal relocaties adressen betreft die relatief t.o.v. het begin van een sectie beschreven worden.

Hoofdstuk 3

Analyses en Optimalisaties

"Laten we ons niet te druk maken over het leven,
we komen er toch niet levend uit."
Ciora

"De enige constante in het leven is: de verandering."
La Rochefoucauld

"Gebruiksaanwijzing bij een nieuw apparaat:
Punt 1: neem een stevige borrel."
W. van Broeckhoven

In dit hoofdstuk worden in extenso een aantal algemene analyses besproken die de fundering vormen van de algemene optimalisaties. Het zijn meer bepaald de analyses die van heel groot belang gebleken zijn voor compactie, en waar de grootste persoonlijke bijdrage van de auteur in gevonden wordt: *levensduuranalyse*, *constantenpropagatie* en de *analyse van het gebruik van statisch gealloceerde data*.

De efficiëntie en effectiviteit van de analyses zal hierbij aangeraakt worden. Voor een echte evaluatie van de analyses wachten we echter tot het einde van dit proefschrift: de enige echte criteria van belang zijn immers uitvoeringstijd, geheugenverbruik en invloed op de compactiefactor van de analyses. Een analyse waarvan de resultaten veel nauwkeuriger zijn, maar waarvan de extra nauwkeurigheid niet kan uitgebuit worden, is immers nutteloos.

De eerste twee analyses, levensduuranalyse en constantenpropagatie, komen uit de vertalerswereld. De toepassing ervan na het linken zorgt er echter voor dat we ze vaak niet zomaar kunnen overnemen. We zullen dus met name aandacht schenken aan de verschilpunten met analoge of identieke analyses tijdens het vertalen. De analyse van het gebruik van statisch gealloceerde data is specifiek voor de transformatie van volledige programma's, dus na het linken.

In een laatste sectie wordt voor de volledigheid, doch korter, ingegaan op een aantal analyses en optimalisaties die eveneens nuttig zijn voor compactie na het linken. De auteur van dit werk heeft er behoudens waar dit expliciet vermeld wordt, geen bijdrage toe geleverd, en deze analyses worden dus enkel voor de volledigheid meegegeven.

3.1 Levensduuranalyse

Levensduuranalyse tracht te achterhalen of waarden die op een bepaald moment tijdens de uitvoering van een programma in een veranderlijke opgeslagen liggen, later tijdens de uitvoering van het programma nog kunnen gebruikt worden. Indien dit het geval is, wordt de waarde als levend beschouwd, in het andere geval is ze dood. Het is een goed gekende analyse [Much97], die in zowat alle vertalers geïmplementeerd wordt.

3.1.1 Inleiding

In het kader van compactie na het linken zullen we de waarden in registers beschouwen i.p.v. die in veranderlijken. Er is immers geen notie van veranderlijken zoals dit in vertalers het geval is. De voornaamste programmatransformaties die verderbouwen op de resultaten van levensduuranalyse zijn

- het verwijderen van loze instructies, d.w.z. instructies die dode waarden produceren;
- het vinden van vrije registers die men kan gebruiken om er andere waarden in te plaatsen;
- het aanwenden van levensduurinformatie ter verfijning van andere analyses zoals de detectie van dode data.

.

Omdat we het volledige programma ter beschikking hebben, beschouwen we enkel interprocedurale levensduuranalyses. Deze technieken kunnen handig gebruik maken van het feit dat het aantal registers sterk beperkt is (in tegenstelling tot het aantal veranderlijken in een programma) en dat er geen aliassen mogelijk zijn tussen de registers. Er bestaan immers geen wijzers naar registers[1].

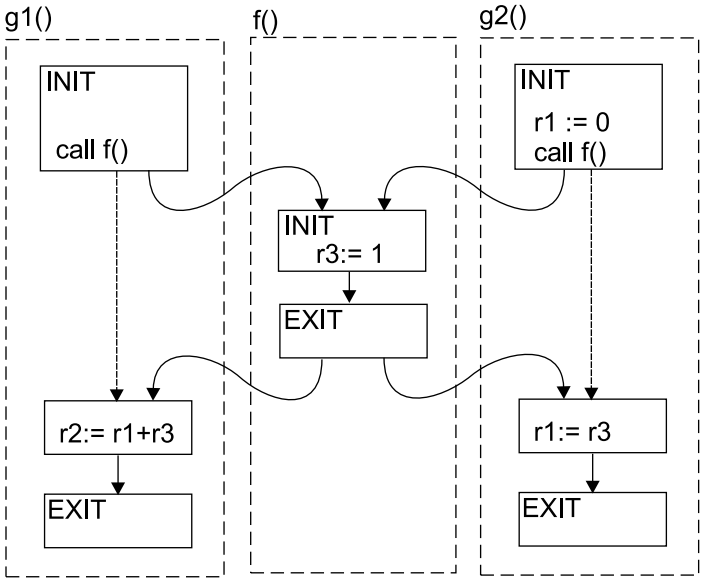
Tevens zullen we de levensduur van statisch gealloceerde data achterhalen. Dit laatste vereist, net door de aanwezigheid van wijzers, echter een heel andere techniek. Aangezien die techniek heel nauw samenhangt met constantenpropagatie, zullen we de levensduuranalyse van statisch gealloceerde data pas bespreken nadat we constantenpropagatie besproken hebben.

De levensduuranalyses die we in dit proefschrift bespreken zijn verloopongevoelig [Marl95]: tijdens het propageren van gegevens doorheen het programma gaan we ervan uit dat conditionele sprongen in beide richtingen kunnen gaan, m.a.w. we evalueren ze niet tijdens de levensduuranalyse. Sommige van de uitvoeringspaden die we aldus meerekenen zullen in realiteit nooit gevolgd worden, wat resulteert in een ietwat conservatieve schatting van de levensduur. Merk op dat ook wanneer we de conditionele sprongen op de een of andere manier zouden kunnen interpreteren tijdens de levensduuranalyse, er zogenaamde niet-realiseerbare paden [Land91] zouden in beschouwing genomen worden. Statische analyses kunnen in het algemeen immers geen volledig uitsluitsel geven over welke paden gerealiseerd zullen worden en welke niet (het probleem is m.a.w. onbeslisbaar).

Anders is het voor de niet-realiseerbare paden in de ICVG met betrekking tot procedure-oproepen. Indien een procedure meerdere oproepers heeft, zullen verscheidene oproeppijlen de procedure binnenkomen. Verscheidene terugkeerpijlen zullen vertrekken vanuit de uitgangsknoop. Met elke oproeppijl correspondeert slechts één terugkeerpijl. In dit geval is het herkennen van de realiseerbare combinaties kinderspel: ze worden geïdentificeerd a.d.h.v. de linkpijlen. We hebben dus geen bijkomende analyses nodig om de aldus niet-realiseerbare paden te herkennen.

Een analyse die geen rekening houdt met het feit dat er niet-realiseerbare interprocedurale paden zijn, wordt een *contextongevoelige*

[1] Een uitzondering hierop zijn architecturale registers die verwijzen naar geheugenlocaties, zoals bij de PDP-11 architectuur [Ston75].



Figuur 3.1: Voorbeeld ter verduidelijking van het begrip contextgevoelige analyse.

analyse [Marl95] genoemd: bij het propageren van informatie uit de procedure wordt er geen rekening gehouden met de context waaruit de informatie in de procedure gepropageerd is. Houdt men er wel rekening mee, en vermijdt men dus dat de niet-realiseerbare paden een bijdrage leveren aan het resultaat van de analyse, dan spreekt men over een *contextgevoelige analyse*: informatie die uit een procedure gepropageerd wordt, wordt enkel gepropageerd naar plaatsen corresponderend met de context waaruit de informatie in de procedure binnengebracht werd.

Voorbeeld 3.1 Beschouw de graaf in figuur 3.1. De procedure f() wordt opgeroepen vanuit twee contexten. Vóór de oproep in procedure g2() wordt register r1 gedefinieerd met waarde 0. Wanneer men blindelings de pijlen g2() $\rightarrow$ f() $\rightarrow$ g1() volgt zou men kunnen concluderen dat deze waarde levend is, omdat ze zal gebruikt worden na de terugkeer in procedure g1(). Met behulp van een contextongevoelige analyse zal de definitie van r1 dus niet kunnen verwijderd worden uit het programma. Een contextgevoelige analyse zal daarentegen dat pad buiten beschouwing laten en correct concluderen dat de waarde 0 toegewezen aan r1 dood

is, omdat die waarde voor ze gebruikt kan worden overschreven wordt na de terugkeer in `g2()`. □

Al is het triviaal om de aldus niet-realiseerbare paden te herkennen, een contextgevoelige levensduuranalyse is heel wat complexer dan een contextongevoelige. In de rest van deze sectie zullen we eerst voor beide analyses het vertrekpunt belichten. Daarna gaan we in op verdere verfijningen van de contextgevoelige analyse. We zullen het hebben over verfijningen van de effectiviteit (nauwkeurigheid) en ook van de efficiëntie (uitvoeringssnelheid en geheugenverbruik).

3.1.2 Contextongevoelige analyse

De contextongevoelige interprocedurale levensduuranalyse is een rechtstreekse uitbreiding van de intraprocedurale analyse. Oproeppijlen, terugkeer- en compensatiepijlen worden als alle andere pijlen beschouwd. Linkpijlen worden niet in beschouwing genomen.

De dataverloopvergelijkingen [Much97] voor elke knoop in de graaf zijn als volgt:[2]

$$\mathrm{Uit_k}[k] = \bigcup_{l \in \mathrm{Opv}[k]} \mathrm{In_k}[l], \tag{3.1}$$

$$\mathrm{In_k}[k] = \mathrm{Cons_k}[k] \cup (\mathrm{Uit_k}[k] \setminus \mathrm{Prod_k}[k]) \tag{3.2}$$

De in deze formules voorkomende verzamelingen zijn als volgt gedefinieerd:

- $\mathrm{In_k}[k]$ is de verzameling registers die een levende inhoud hebben aan de ingang van knoop k.
- $\mathrm{Uit_k}[k]$ is de verzameling registers die een levende inhoud hebben aan de uitgang van knoop k.
- $\mathrm{Cons_k}[k]$ is de verzameling registers die geconsumeerd worden in knoop k voor ze gedefinieerd worden. Daarin zitten de registers waarvan de binnenkomende waarde in de knoop gebruikt wordt door een instructie (de consument).

[2] We gebruiken de subnotaties 'k', 'p' en 'i' doorheen dit hoofdstuk om aan te duiden of de betreffende verzamelingen bij knopen, procedures of instructies horen. De eerste 'k' uit $\mathrm{Uit_k}[k]$ is dus geen index of parameter, maar verduidelijkt enkel dat deze verzameling bij een knoop hoort. De tweede 'k' is het nummer (de identificatie) van een knoop.

- $\text{Prod}_k[k]$ is de verzameling registers waarin instructies waarden produceren in knoop k. Dit zijn m.a.w. de registers die als doeloperand in instructies in knoop k voorkomen.

- $\text{Opv}[k]$ is de verzameling van alle knopen die k opvolgen in de ICVG, met uitzondering van de opvolgers langs linkpijlen (de linkknopen).

Formule 3.1 moet als volgt geïnterpreteerd worden: een waarde is levend aan het eind van een knoop als ze levend is bij het begin van minstens één knoop die erop volgt in de graaf. Deze vergelijking is een zogenaamde achterwaartse vergelijking, net als vergelijking 3.2 die zegt dat wat levend is aan het begin van een knoop wordt gedefinieerd aan de hand van eigenschappen van de knoop zelf en aan de hand van wat levend is aan het eind van de knoop. De interpretatie van de formule is als volgt: een waarde is levend aan het begin van een knoop als ze ofwel gebruikt wordt in de knoop zelf en/of ze wordt gebruikt voorbij de knoop. Dit laatste kan enkel als ze niet overschreven wordt in de knoop zelf.

Deze combinatie van vergelijkingen zal vaak terugkomen: één formule beschrijft de verloopvergelijkingen voor bewerkingen, een tweede formule de vergelijkingen voor controleverloop in de graaf.

Iteratieve oplossing

De oplossing van bovenstaande vergelijkingen gebeurt conceptueel aan de hand van het zoeken van een dekpunt van een functie in een tralie. Vooraleer het concrete algoritme voor het oplossen van deze vergelijkingen te geven gaan we kort in op deze (eerder theoretische) begrippen.

- Een tralie $L, \sqsubseteq$ is een partieel geordende verzameling waarvoor voor elk koppel elementen x en y uit L een unieke kleinste bovengrens (genoteerd $x \sqcap y$) en een unieke grootste ondergrens (genoteerd ($x \sqcup y$) bestaan met betrekking tot de partiële ordening. Meestal bevat L twee speciale elementen, $\top$ en $\bot$, waarvoor geldt dat

$$\forall x \in L : x \sqcup \top = \top \text{ en } x \sqcap \bot = \bot \tag{3.3}$$

  De hoogte n van een tralie is de lengte van de langste sequentie

elementen uit L waarvoor geldt dat

$$\bot \sqsubset x_0 \sqsubset x_0 \sqsubset x_1 \sqsubset ... \sqsubset x_n \sqsubset \top \tag{3.4}$$

- Een functie f binnen de tralie L is niks anders dan een functie die elementen uit L op elementen uit L afbeeldt: $f : L \to L$. Een functie f wordt monotoon genoemd indien geldt dat

$$\forall x, y \in L : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y) \tag{3.5}$$

- Een dekpunt x van een functie f is een element uit de tralie waarvoor geldt: $f(x) = x$.

In analyses van programma's die aan de hand van tralies opgelost worden, worden de eigenschappen die men wenst te analyseren afgebeeld op elementen van de tralie. In het geval van levensduuranalyse bestaat de tralie uit de machtsverzameling ($\mathcal{P}(R)$) van de verzameling van alle registers (R): elk element van de tralie is een deelverzameling van deze verzameling. $\top$ komt overeen R en $\bot$ met $\emptyset$. $\sqsubseteq$ is niks anders dan $\subseteq$.

De relaties tussen de geanalyseerde eigenschappen op verschillende programmapunten worden vertaald in functies binnen de tralie. Voor de levensduuranalyse worden deze functies verkregen door in de vergelijkingen 3.1 en 3.2 $\cap$ door $\sqcap$ te vervangen en $\cup$ door $\sqcup$. De uitkomst van de analyse is dan het minimale dekpunt van alle vergelijkingen voor alle programmapunten samen.

Deze uitkomst wordt vaak verkregen door de vergelijkingen binnen de tralie iteratief op te lossen om zo het dekpunt te benaderen vanaf de onderkant. Startend met een initiële waarde voor elke veranderlijke worden de functies iteratief toegepast tot men een dekpunt bekomt, m.a.w. tot de waarden van alle veranderlijken geconvergeerd zijn. Indien de functies monotoon zijn en de hoogte van de tralie eindig, dan zal het iteratief oplossen zeker convergeren binnen een eindige tijd. Voor de levensduuranalyse is dit het geval, aangezien het aantal elementen in L eindig is en aangezien alle functies monotoon zijn: de vereniging van twee verzamelingen is immers steeds groter dan de twee verzamelingen afzonderlijk.

Het iteratief oplossen van de vergelijkingen voor levensduuranalyse gebeurt dan met het algoritme in figuur 3.2. Hierin is K de verzameling van alle echte knopen in de graaf, K^+ is dezelfde verzameling

uitgebreid met de helleknopen. In de verzameling M zitten die knopen waarvoor een nieuwe iteratie vereist is.

We itereren over de knopen zolang er veranderingen optreden aan hun verzameling uitgaande levende registers. Telkens er zo'n verandering heeft plaatsgevonden (dan is $k \in M$, regels 2 en 3), worden alle voorgangers l bekeken (regel 5) en wordt er gekeken of ook hun verzameling uitgaande levende waarden moet aangepast worden (regels 6-7). Indien dit zo is wordt die voorganger voor een volgende iteratie opgenomen in M (regel 8).

Dit dekpuntalgoritme moet natuurlijk op een correcte manier geïnitialiseerd worden. Dit gebeurt in de pseudo-code van figuur 3.3. Voor alle knopen wordt de beginwaarde op de lege verzameling gezet (regel 2). De oplossing van het stelsel vergelijkingen gebeurt m.a.w. op optimistische wijze: we gaan er initieel van uit dat geen registers levende waarden bevatten aan het einde van een blok. Tijdens de iteratieve oplossing zal deze optimistische schatting herzien worden waar nodig. Het is dus noodzakelijk verder te itereren tot de verzamelingen convergeren om een correct resultaat te verkrijgen (dit is altijd het geval bij optimistische oplossingstechnieken). Tevens moet worden vastgelegd welke registers geproduceerd en geconsumeerd worden door elke knoop, door de instructies in die knoop af te lopen in omgekeerde volgorde (regels 3-6). Tenslotte worden ook de registers vastgelegd die de verschillende helleknopen in de graaf consumeren (regels 7-10). Merk op dat dit volstaat om de analyse conservatief te maken met betrekking tot de helleknopen: gelet op formule 3.2 volstaat het de verzameling geconsumeerde registers voldoende groot te kiezen. Deze keuze wordt grotendeels bepaald door de oproepstandaard. We hebben de volledige verzameling registers dan ook opgedeeld in verschillende groepen, overeenstemmend met verschillende onderdelen van een mogelijke oproepstandaard. Deze opdeling is weergegeven in tabel 3.1.

Voor een meer formeel bewijs van het algoritme in figuur 3.2 verwijzen we naar bijlage A.

3.1.3 Contextgevoelige analyse

Zoals voor heel wat contextgevoelige analyses moeten we afwegingen maken over de complexiteit van de analyse en de nauwkeurigheid. De mogelijke oplossingen bevinden zich in een breed spectrum liggende tussen twee uitersten:

```
1  | M := K+
2  | while (M ≠ ∅)
3  |    M := M \ {k}
4  |    In_k := Cons_k[k] ∪ (Uit_k[k] \ Prod_k[k])
5  |    for all l ∈ Voorg[k]
6  |       if ((Uit_k[l] ∪ In_k) ≠ Uit_k[l])
7  |          Uit_k[l] := Uit_k[l] ∪ In_k
8  |          M := M ∪ {l}
```

Figuur 3.2: Iteratief algoritme voor contextongevoelige levensduuranalyse. Hierin is Voorg[k] de verzameling van alle voorgangers van knoop k in de ICVG.

```
1  | do for all k ∈ K+
2  |    Uit_k[k] := Cons_k[k] := Prod_k[k] := ∅
3  |    do for all instructies i in k in omgekeerde volgorde
4  |       Prod_k[k] := Prod_k[k] ∪ {doeloperandi van i}
5  |       Cons_k[k] := Cons_k[k] \ {doeloperandi van i}
6  |       Cons_k[k] := Cons_k[k] ∪ {bronoperandi van i}
7  | Cons_k[helleknoop] := R
8  | Cons_k[oproepershelleknoop] := R_glob ∪ R_achteraf ∪ R_func
9  | Cons_k[opgeroepenenhelleknoop] := R_glob ∪ R_achteraf ∪ R_arg
10 | Cons_k[systeemhelleknoop] := R_glob ∪ R_achteraf ∪ R_arg
```

Figuur 3.3: Initialisatie voor de contextongevoelige levensduuranalyse.

R	alle registers
R_{glob}	globale registers, zoals stapelwijzer, globale wijzer, enz.
R_{achteraf}	achteraf te bewaren registers
R_{func}	register dat de terugkeerwaarde van een functie bevat
R_{arg}	argumentregisters

Tabel 3.1: Opdeling van de registers volgens de oproepconventie.

- De meest nauwkeurige analyses voeren een aparte, en daarom heel nauwkeurige, analyse uit voor elke context waarin een bepaalde procedure kan uitgevoerd worden. Het is duidelijk dat zulk een aanpak een hoge complexiteit bezit en het geheugenverbruik en het tijdsverbruik navenant zullen zijn.
- De minst nauwkeurige analyses maken gebruik van samenvattende, contextonafhankelijke vergelijkingen die procedures als een atomaire operatie beschrijven: de samenvattende informatie beschrijft wat de eigenschappen aan de ingang van een procedure zijn i.f.v. de eigenschappen aan de uitgang. Door deze contextonafhankelijke vergelijkingen toe te passen voor verschillende contexten (uitgangen corresponderend met ingangen volgens de realiseerbare paden) voor elke procedure tijdens het iteratief oplossen, wordt de analyse contextgevoelig. Dit soort analyses is veel sneller, aangezien voor elke procedure slechts eenmaal de samenvattende vergelijkingen moeten opgesteld worden.

Wij hebben in eerste instantie voor de tweede manier gekozen. Naar analogie met formule 3.2 die de verloopvergelijking voor een knoop weergeeft, wordt een procedure-oproep gemodelleerd met een verloopvergelijking [Good97]. Voor een oproepknoop k die procedure p oproept en een linkknoop k' heeft, geldt nu:

$$\mathrm{Uit_k}[k] = \mathrm{Cons_p}[p] \cup (\mathrm{In_k}[k'] \setminus \mathrm{Prod_p}[p]) \tag{3.6}$$

Hierin is $\mathrm{Cons_p}[p]$ de verzameling registers die in procedure p kunnen geconsumeerd worden alvorens ze gedefinieerd worden. Dit is m.a.w. die verzameling registers die altijd levend is bij een oproep naar p, ongeacht wat er levend is bij het terugkeren na de oproep. $\mathrm{Prod_p}[p]$ is de verzameling registers die op alle paden doorheen procedure p gedefinieerd worden. Het is enkel voor die registers dat een waarde die vóór de oproep geproduceerd is nooit kan gebruikt worden na het terugkeren uit p, aangezien ze op alle paden doorheen p overschreven wordt.

Om redenen die later duidelijk zullen worden, gaan we echter niet bovenstaande vergelijking gebruiken, maar wel een equivalente vergelijking. Men ziet makkelijk in dat tevens geldt

$$\begin{aligned} \mathrm{Uit_k}[k] &= \mathrm{Cons_p}[p] \cup (\mathrm{In_k}[k'] \cap \overline{\mathrm{Prod_p}[p]}) && (3.7)\\ &= \mathrm{Cons_p}[p] \cup (\mathrm{In_k}[k'] \cap \mathrm{Door_p}[p]) && (3.8) \end{aligned}$$

$\overline{x}$ is het complement van x, zodat $\mathrm{Door_p}[p]$ die registers zijn die, als ze levend zijn vlak na de terugkeer naar de oproeper, dan ook levend

zijn vlak vóór de oproep: de levensduurinformatie gaat onbelemmerd doorheen de opgeroepen procedure, omdat er geen definities zijn die het verloop ervan tegenhouden. De vraag is hoe we $\text{Cons}_\text{p}[p]$ en $\text{Door}_\text{p}[p]$ op een efficiënte manier kunnen berekenen.

Berekening van $\text{Door}_\text{p}[p]$

Deze verzamelingen kunnen aan de hand van volgende vergelijkingen berekend worden:

$$\begin{aligned}
\text{DoorUit}_\text{k}[k] &= \bigcup_{l \in \text{Opv}[k]} \text{DoorIn}_\text{k}[l] && (3.9)\\
&\quad \text{als } k \text{ geen oproep- of uitgangsknoop is} \\
&= \text{DoorIn}_\text{k}[\text{Init}[p]] \cap \text{DoorIn}_\text{k}[k'] && (3.10)\\
&\quad \text{als } k \text{ een oproep is naar } p \text{ met linkknoop } k' \\
\text{DoorIn}_\text{k}[k] &= \text{DoorUit}_\text{k}[k] \setminus \text{Prod}_\text{k}[k] && (3.11)\\
\text{Door}_\text{p}[p] &= \text{DoorIn}_\text{k}[\text{Init}[p]] && (3.12)
\end{aligned}$$

met als initiële waarden:

$$\begin{aligned}
\text{DoorUit}_\text{k}[k] &= R \text{ als } k \text{ een uitgangsknoop is} \\
&= \emptyset \text{ voor alle andere knopen } k && (3.13)
\end{aligned}$$

We veronderstellen bij de initialisatie dat alle registers levend zijn aan de uitgangsknoop van een procedure, m.a.w. we veronderstellen ze levend over de terugkeerpijlen die vertrekken vanuit de uitgangsknoop. Door deze registers achterwaarts te propageren zolang ze nergens gedefinieerd worden, krijgen we aan de ingangsknoop uiteindelijk de verzameling registers waarvan de waarde niet op alle paden overschreven wordt. Dit is m.a.w. de verzameling registers waarvan de waarde vóór de oproep kan gebruikt worden na het terugkeren uit de opgeroepen procedure.

We zijn echter niet gebonden aan deze intuïtieve betekenis van $\text{Door}_\text{p}[p]$. Dit wordt duidelijk als we vergelijking 3.8 herschrijven als volgt:

$$\text{Uit}_\text{k}[k] = (\text{Cons}_\text{p}[p] \cup \text{In}_\text{k}[k']) \cap (\text{Cons}_\text{p}[p] \cup \text{Door}_\text{p}[p]) \qquad (3.14)$$

We zien dat we het niet te nauw moeten nemen met de originele betekenis van $\text{Door}_\text{p}[p]$ wat betreft registers die ook in $\text{Cons}_\text{p}[p]$ zitten [Muth99]: het speelt geen enkele rol of we die registers in $\text{Door}_\text{p}[p]$

steken of niet. Daarom kunnen we vergelijking 3.11 vervangen door

$$\text{DoorIn}_k[k] = \text{Cons}_k[k] \cup (\text{DoorUit}_k[k] \setminus \text{Prod}_k[k]) \tag{3.15}$$

Dit maakt de vergelijkingen analoog aan wat we in de rest van de berekeningen zullen tegenkomen.

Berekening van $\text{Cons}_p[p]$

Nadat we alle verzamelingen $\text{Door}_p[p]$ berekend hebben, kunnen we de $\text{Cons}_p[p]$ verzamelingen berekenen met gelijkaardige vergelijkingen:

$$\begin{aligned}
\text{ConsUit}_k[k] &= \bigcup_{l \in \text{Opv}[k]} \text{ConsIn}_k[l] && (3.16)\\
&\quad \text{als } k \text{ geen oproep- of uitgangsknoop is} \\
&= \text{ConsIn}_k[\text{Init}[p]] \cup (\text{Door}_p[p] \cap \text{ConsIn}_k[k']) && (3.17)\\
&\quad \text{als } k \text{ een oproep is naar } p \text{ met linkknoop } k' \\
\text{ConsIn}_k[k] &= \text{Cons}_k[k] \cup (\text{ConsUit}_k[k] \setminus \text{Prod}_k[k]) && (3.18)\\
\text{Cons}_p[p] &= \text{ConsIn}_k[\text{Init}[p]] && (3.19)
\end{aligned}$$

Deze worden geïnitialiseerd met

$$\text{ConsUit}_k[k] := \emptyset \tag{3.20}$$

Door nu aan de uitgangsknopen geen enkel register levend te veronderstellen, zullen na de achterwaartse propagatie enkel die registers die kunnen gebruikt worden door de procedure zelf (of een erdoor opgeroepen procedure) in de ConsIn_k verzameling van de ingangsknoop $\text{Init}[p]$ van p zitten.

Het totaalbeeld

Als we de vergelijkingen waarmee Cons_p, Door_p en Uit_k berekend worden naast elkaar leggen, zien we dat deze heel gelijklopend zijn. Dit vergemakkelijkt aanzienlijk de implementatie, aangezien we maar één dekpuntberekening hoeven te implementeren, die drie maal uitgevoerd wordt met nagenoeg dezelfde vergelijkingen, zij het op verschillende verzamelingen. Dit is de reden waarom de berekening van Door_p verkozen werd boven die van Prod_p en ze bovendien afwijkt van de intuïtieve betekenis die we langs Prod_p om aan Door_p hadden gegeven.

Met betrekking tot de helleknopen kunnen we zeggen dat het, zoals we in het contextongevoelige geval deden met Prod_k en Cons_k, volstaat om Cons_p en Prod_p vast te leggen:

$$\begin{aligned}
\text{Door}_p[\text{helleproc.}] &:= R && (3.21)\\
\text{Cons}_p[\text{helleproc.}] &:= R && (3.22)\\
\text{Door}_p[\text{opgeroepenenhelleproc.}] &:= R_{\text{glob}} \cup R_{\text{achteraf}} \cup R_{\text{func}} && (3.23)\\
\text{Cons}_p[\text{opgeroepenenhelleproc.}] &:= R_{\text{arg}} \cup R_{\text{glob}} && (3.24)\\
\text{Door}_p[\text{systeemhelleproc.}] &:= R_{\text{glob}} \cup R_{\text{achteraf}} \cup R_{\text{func}} && (3.25)\\
\text{Cons}_p[\text{systeemhelleproc.}] &:= R_{\text{arg}} \cup \{\text{stapelwijzer}\} && (3.26)
\end{aligned}$$

De toekenning zoals hier weergegeven is degene die in onze prototype compactor gebruikt wordt. Uiteraard is dit sterk afhankelijk van de oproepstandaard. Dit zal dus van architectuur tot architectuur verschillen. Merk op dat voor de oproepershelleprocedure deze waarden niet moeten vastgelegd worden. Er komt immers geen enkele oproeppijl in het programma naar deze procedure voor.

3.1.4 Verfijning met betrekking tot de effectiviteit

Hier zullen we ingaan op enkele verfijningen die een dekpuntoplossing van bovenstaande vergelijkingen accurater en dus minder conservatief maken.

Achteraf te bewaren registers

De bij conventie achteraf te bewaren registers mogen slechts door een procedure beschreven worden als die procedure de oorspronkelijke waarde terug herstelt. Meestal zal in zo'n geval de oorspronkelijke waarde eerst op de stapel weggeschreven worden. Net voor het terugkeren uit de opgeroepen procedure laadt men dan de waarde op van de stapel.

Als blijkt dat geen enkele van de oproepers van deze procedure de aldus herstelde waarde nog gebruikt na de oproep, zijn de voor het achteraf bewaren gebruikte schrijf- en leesoperaties overbodig en kunnen ze dus geschrapt worden. Nochtans zal de hierboven beschreven analyse deze waarde als levend aanduiden voor elke oproep. De schrijfoperatie die de waarde op de stapel zet zorgt er immers voor dat het betreffende register in de Cons_p verzameling van die procedure terechtkomt.

Om dit te vermijden volstaat het de op de stapel bewaarde registers te identificeren en de vergelijkingen voor Door_p en Cons_p aan te passen. Noemen we de verzameling op de stapel bewaarde registers van een procedure p $\text{Stapel}_p[p]$, dan worden vergelijkingen 3.12 en 3.19 vervangen door

$$\text{Door}_p[p] = \text{DoorIn}_k[\text{Init}[p]] \cup \text{Stapel}_p[p] \quad (3.27)$$

$$\text{Cons}_p[p] = \text{ConsIn}_k[\text{Init}[p]] \setminus \text{Stapel}_p[p] \quad (3.28)$$

De belangrijkste vraag is nu natuurlijk hoe we deze registers kunnen identificeren. Daarvoor hebben we twee opties: we kunnen de procedures zelf analyseren, of we kunnen verder bouwen op de kennis van de oproepstandaard.

Analyse van achteraf te bewaren registers Voor veel procedures kan men de op de stapel bewaarde registers makkelijk terugvinden. Het volstaat in het ingangsblok van een procedure op zoek te gaan naar schrijfoperaties naar de stapel en dan te verifiëren of de aldus bewaarde waarden opgeladen worden vlak voor elke terugkeerinstructie. In onze prototype compactor hebben we zulk een analyse geïmplementeerd met goede resultaten.

Soms is het echter niet zo eenvoudig, bv. als de vertaler bij de optimalisatie van de procedure de schrijf- en leesoperaties verplaatst heeft zodat ze slechts uitgevoerd worden op die uitvoeringspaden waarop de betreffende registers effectief overschreven werden. Hoe geavanceerder de vertaler, hoe kleiner de kans dat een relatief eenvoudige analyse de bewaarde registers zal kunnen achterhalen.

Verder bouwen op de oproepconventies Waar de eigen analyse niet tot goede resultaten leidt, kan men in vele gevallen voortgaan op de oproepconventies: als een procedure kan opgeroepen worden vanuit de oproepershelleknoop, betekent dit dat deze opgeroepen procedure verondersteld wordt de oproepconventies na te leven.[3] Van zo'n opgeroepen procedure weten we dan bv. dat de Cons_p verzameling zeker niet de achteraf te bewaren registers kan bevatten, maar dat die daarentegen in Door_p opgenomen moeten worden.

Hierbij dient opgemerkt te worden dat deze extra informatie slechts nut heeft als er ook andere oproepers van de procedure zijn dan de op-

[3] Zie voetnoot in sectie 2.5.3.

roepershelleknoop. Het is immers enkel voor die andere oproepcontexten dat de extra informatie nuttig aangewend wordt. Als er andere oproepcontexten zijn kan het dus voor een levensduuranalyse nuttig zijn een oproeppijl vanuit de helleknoop in het programma te hebben, ook al weten we dat deze oproeppijl niet meer realiseerbaar is.

Voor procedures die geen andere oproepers hebben, is de extra informatie van geen belang. Voor deze procedures is de nauwkeurigheid van de levensduuranalyse dus geen argument om niet-realiseerbare oproeppijlen vanuit de oproepershelleknoop in de graaf te houden. In dat geval zullen niet-realiseerbare oproeppijlen, zoals in sectie 2.5.3 besproken, eerder een belemmering voor de compactie van een programma vormen, aangezien ze het verwijderen of het substitueren van procedures in de weg kunnen staan.

We hebben vastgesteld dat het gemiddeld genomen voor procedures met één bekende oproeper en een niet-realiseerbare oproeppijl vanuit de oproepershelleknoop nuttiger is de dode oproeppijl te verwijderen en de procedure dan te substitueren i.p.v. de dode oproeppijl te laten staan om de op de stapel bewaarde registers te kunnen herkennen. Het criterium dat we geïmplementeerd hebben om dode oproeppijlen vanuit de oproepershelleknoop (waarvan corresponderende code-adressen gestockeerd in de data dus dood zijn) te verwijderen is dan ook:

- het totale aantal oproepers is maximaal 2, of
- we hadden alle levensduurinformatie die deze pijl ons opleverde zelf al achterhaald met de analyse besproken in de vorige paragraaf.

De helleprocedures van dichterbij bekeken

Naast het verwijderen van loze instructies is de levensduuranalyse van belang voor het vinden van vrije registers, waarin tijdelijk waarden kunnen opgeslagen worden. Zulke registers behouden slechts hun waarde over een procedure-oproep heen als ze zeker niet overschreven worden door de opgeroepen procedure.

Daarom moet men dus van procedures kunnen achterhalen welke registerinhouden zij in elk geval ongemoeid laten (of terug herstellen). Voor een procedure is dit de verzameling van alle registers die nergens in de procedure gedefinieerd worden, uitgebreid met de verzameling

Stapel_p. Noemen we deze verzameling Invar_p.

Ook om deze verzameling te bepalen kunnen we ofwel een (triviale) analyse, ofwel de oproepstandaarden gebruiken. Zoals de helleprocedures tot nu toe gebruikt worden, leidt dit echter tot problemen.

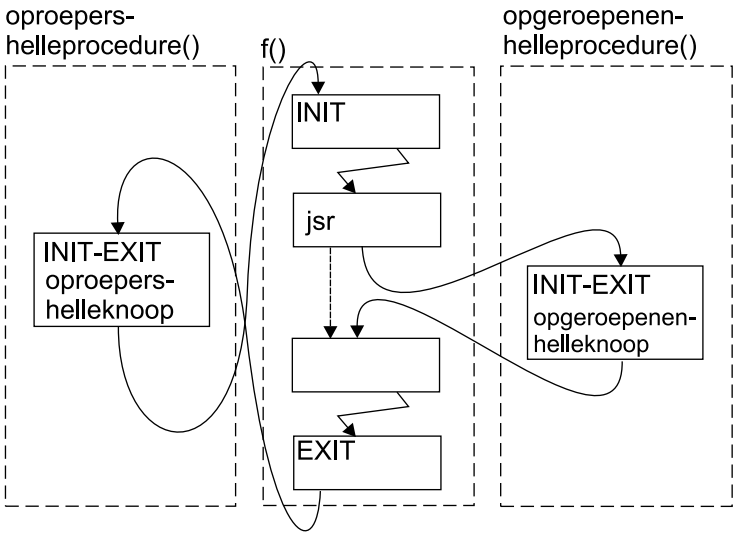
Voorbeeld 3.2 Beschouw procedure f() in figuur 3.4. Ze wordt opgeroepen vanuit de oproepershelleknoop en roept zelf een onbekende procedure op, gemodelleerd middels opgeroepenenhelleknoop(). De vraag is nu in welk register we over die oproep heen tijdelijk een waarde kunnen opslaan. Beschouwen we even die registers die helemaal niet voorkomen in de procedure f(). Daarvan komen die registers in aanmerking die bij conventie vooraf te bewaren zijn: die kunnen we in f() vrij gebruiken zonder dat we ons om de onbekende oproepcontext van f() moeten bekommeren. Jammer genoeg geldt dit ook voor de onbekende procedure die door f() opgeroepen wordt. Ook zij kan ervan uitgaan dat ze precies deze registers mag gebruiken. Vanuit f() kunnen we er dus absoluut niet van uitgaan dat ze bewaard worden over de procedure-oproep heen. Van de vooraf te bewaren registers komen er dus geen in aanmerking.

Voor de achteraf te bewaren registers geldt een soortgelijke redenering: hun inhoud zal wel bewaard blijven over de oproep heen, maar we kunnen er nu niet van uitgaan dat we die registers zomaar mogen gebruiken, omdat we ze levend moeten veronderstellen doorheen f().

Van alle niet in f() gebruikte registers kunnen we dus vooraf noch achteraf te bewaren registers gebruiken. □

Het in voorbeeld 3.2 besproken probleem lijkt vergezocht, maar is het niet: de redenering die er gemaakt wordt geldt immers voor alle code die via eender welk pad bereikbaar is vanuit de oproepershelleknoop en van waaruit de opgeroepenenhelleknoop bereikbaar is en voor alle registers die op die paden niet gedefinieerd worden. Als ze wel gedefinieerd worden geldt bovenstaande redenering niet meer, maar dan zijn ze per definitie ook vaak niet meer bruikbaar voor nieuwe doeleinden.

Een groep registers waarvoor bovenstaande redenering bij uitstek opgaat zijn de registers die door het programma helemaal niet gebruikt worden. Dit zijn er vaak geen of toch niet veel, maar in een aantal niet-



Figuur 3.4: Een procedure die zowel een onbekende oproeper heeft als een onbekende procedure oproept.

wetenschappelijke programma's is dit toch regelmatig het geval voor de registers voor vlottende-kommagetallen.

Als men deze registers, zoals op de Alpha, als tijdelijke registers kan gebruiken i.p.v. overloopcode te moeten toevoegen, is het zonde deze omwille van bovenstaande redenering niet te kunnen gebruiken.

Een gedeeltelijke oplossing voor dit probleem bestaat eruit de oproepershelleknopen en -procedures geen vaste eigenschappen te geven, maar die integendeel ook uit het programma te halen. Zo kan men bv. de verzameling onveranderde registers van de opgeroepenenhelleknoop berekenen als de doorsnede van de verzamelingen van onveranderde registers van alle procedures die door de oproepershelleknoop opgeroepen worden:

$$\mathrm{Invar_p}[\mathrm{Opgeroepenenhelleproc.}] = \bigcap_{k \in \mathrm{Opv}[\mathrm{Oproepershelleknoop}]} \mathrm{Invar_p}[\mathrm{Proc}[k]] \quad (3.29)$$

De opgeroepenenhelleprocedure modelleert nu niet enkel meer de onbekende opgeroepen procedures, haar eigenschappen worden ook afgeleid uit de procedures die daarvoor in aanmerking komen.

Soortgelijke verfijningen voor andere eigenschappen van de helleknopen zijn mogelijk. Het blijven echter slechts gedeeltelijke oplossingen. In het licht van de zoektocht naar vrije registers bv. zal het zo zijn dat een tot dan toe ongebruikt register door deze verfijning wel gebruikt kan worden, maar het zal bij een eenmalig gebruik blijven: vanaf dan duikt het betreffende register gewoon weer overal in de analyses op.

Merk bovendien op dat zulke verfijningen in de praktijk een hele groep controleverloopvergelijkingen toevoegen, waardoor de analyse behoorlijk vertraagd kan worden. Mocht het in de toekomst echter mogelijk zijn om de onbekende oproepcontexten en onbekende opgeroepen procedures te partitioneren en voor elke partitie verschillende helleknopen te gebruiken, dan zal deze verfijning wel degelijk haar nut hebben.

3.1.5 Verfijning met betrekking tot de efficiëntie

Het is zonder meer duidelijk dat een contextgevoelige analyse veel complexer is dan een contextongevoelige, ook al vertonen de vergelijkingen en dus de iteratieve oplossingsmethode veel overeenkomsten. Het verbaast dus niet dat we zowel meer geheugen als meer rekentijd zullen nodig hebben: er zijn immers veel meer vergelijkingen en tussenresultaten.

De basisvergelijkingen en hun oplossing

De basisvergelijkingen van de contextgevoelige analyse werden tot nu toe een beetje apart behandeld in drie groepen (Door_p, Cons_p en Uit_k), wat de indruk kan wekken dat ze achtereenvolgens in verschillende fasen moeten uitgevoerd worden. Dit is niet zo, aangezien alle vergelijkingen die in een iteratieve oplossing voorkomen monotoon zijn. De drie groepen vergelijkingen voor Uit_k, Door_p en Cons_p kunnen dus tegelijkertijd opgelost worden.

Als we deze vergelijkingen echter in drie fases oplossen, kunnen we het geheugenverbruik sterk beperken. Alle drie de eigenschappen die per blok bijgehouden worden tijdens de fases (DoorUit_k, ConsUit_k en Uit_k) kunnen immers in hetzelfde geheugen bewaard worden.

Bovendien kan men uit de vergelijkingen afleiden dat $\text{ConsUit}_\text{k}[k]$ een deelverzameling van $\text{Uit}_\text{k}[k]$ is voor elke knoop k. Men kan in de

derde fase dus $\text{Uit}_k[k]$ initialiseren met $\text{ConsUit}_k[k]$.

Naast de snelheidswinst die men verkrijgt door deze initialisatie, kan men de derde fase, zijnde de berekening van Uit_k, gevoelig versnellen door op te merken dat niet alle veranderingen tijdens het itereren steeds aanleiding geven tot nieuwe noodzakelijke iteraties, naar analogie met wat we doen op regel 10 van. Beschouw daartoe volgende eigenschappen van de hierboven besproken vergelijkingen:

$$\text{Uit}_k[k] \subset (\text{ConsUit}_k[k] \cup \text{DoorUit}_k[k]) \tag{3.30}$$

$$\text{ConsUit}_k[k] \subset \text{Uit}_k[k] \tag{3.31}$$

$$(\text{DoorUit}_k[k] \cap \text{Uit}_k[\text{Exit}[\text{Proc}[k]]]) \subset \text{Uit}_k[k] \tag{3.32}$$

Omwille van de laatste twee vergelijkingen kan men de laatste fase (d.w.z. de berekening van Uit_k) implementeren aan de hand van de pseudo-code in figuur 3.5. De snelheidswinst wordt met dit algoritme geboekt doordat de nieuwe waarde aan de uitgangsknoop van een procedure meteen door kan gepropageerd worden naar elke knoop in de procedure, zonder dat daar een iteratieve berekening binnen de procedure voor nodig is. Volgens [Muth99] levert dit een snelheidswinst op van ongeveer 25%.

Deze aldus verkregen snelheidswinst is echter verwaarloosbaar klein in vergelijking met de snelheidswinst die op de contextgevoelige analyse gehaald wordt door de lokaliteit van het geïmplementeerde algoritme te verhogen en het aantal berekeningen per iteratie zo klein mogelijk te houden. Dit is meteen het onderwerp van de volgende sectie.

Implementatie-aspecten

Met het algoritme uit figuur 3.2 of de daar sterk op lijkende algoritmen uit de eerste en tweede fase van de contextgevoelige analyse kan men nog vele kanten uit wat betreft een implementatie. We bespreken hier dan ook enkele afwegingen betreffende uitvoeringssnelheid en geheugenverbruik van mogelijke implementatievormen. Enkele voorafgaande bemerkingen hieromtrent zijn:

- Om een globale analyse als levensduuranalyse te kunnen uitvoeren binnen een aanvaardbare tijdsspanne is het noodzakelijk met vergelijkingen per basisblok te kunnen werken i.p.v. met vergelijkingen per instructie. Als gevolg daarvan moeten we in geheugen voorzien voor de verzamelingen $\text{Cons}_k[k]$ en $\text{Prod}_k[k]$.

```
0   for all knopen k
1     Uit_k[k] := ConsUit_k[k]
2     In_k[k] := ConsIn_k[k]
3   wijziging := waar
4   while (wijziging)
5     wijziging := onwaar
6     for all procedures p
7       NieuwUit_k = ∅
8       for all k ∈ (Opv[Exit[p]]
9         NieuwUit_k := NieuwUit ∪ In_k[k]
10      if (NieuwUit_k ≠ Uit_k[Exit[p]])
11        Uit_k[Exit[p]] := NieuwUit_k
12        wijziging := waar
13        for all knopen l in procedure p
14          Uit_k[l] := ConsUit_k[l] ∪ (DoorUit_k[l] ∩ NieuwUit_k)
15          In_k[l] := ConsIn_k[l] ∪ (DoorUit_k[l] ∩ NieuwUit_k)
```

Figuur 3.5: Geoptimaliseerde laatste fase contextgevoelige levensduuranalyse.

- Daarnaast volstaat het echter om enkel aan de uitgangen van basisblokken de verzamelingen van levende registers op te slaan. Waar we die informatie nodig hebben in of vooraan een basisblok kunnen we ze makkelijk afleiden uit de verzameling levende waarden aan de uitgang van het blok.

Beide opmerkingen zijn al gehonoreerd in het algoritme in figuur 3.2 zelf. Daarnaast moeten er echter nog een aantal belangrijke vragen beantwoord worden, zoals over de voorstellingswijze van de verzamelingen. Omdat elke architectuur een vast registerbestand heeft, is een bitvector [Aho86] een uiterst geschikte keuze. Elk bitje in de vector correspondeert met een bepaald register. Zit een register in een verzameling, dan is het corresponderend bitje een 1, anders is het een 0. Operaties als vereniging, doorsnede en verschil van verzamelingen kunnen efficiënt met de bitsgewijze logische operaties geïmplementeerd worden.

Een andere belangrijke vraag is de volgende: in welke volgorde selecteren we knopen uit de verzameling M van gemarkeerde knopen? Experimenteel hebben we vastgesteld dat mogelijke volgordes die theoretisch [Aho86] in aanmerking komen om een graaf te doorlopen, in

de praktijk verre van optimaal zijn. Deze volgordes, zoals bv. diepte-eerst, geven in theorie aanleiding tot het minimaal gemiddelde aantal iteraties van de while-lus in het algoritme. In de praktijk is het aantal iteraties echter niet de enige maat voor uitvoeringssnelheid. De gemiddelde uitvoeringstijd van een iteratie zelf is ook van belang, en deze tijd wordt eveneens grotendeels bepaald door de volgorde waarin de knopen doorlopen worden. Het verband tussen beide zit in het gedrag van de tussengeheugens tijdens de analyse. Het is van primordiaal belang deze zo optimaal mogelijk te benutten, m.a.w. potentieel aanwezige lokaliteit te benutten.

Een eerste belangrijke bijdrage tot het uitbuiten van lokaliteit is het splitsen van de while-lus (figuur 3.2) in twee lussen: een buitenste die over de procedures itereert waarin blokken gemarkeerd zijn en een binnenste die over de blokken binnen zo'n procedure itereert totdat alles binnen die procedure geconvergeerd is. De keuze van procedures als granulariteit voor de buitenste lus ligt voor de hand. Niet alleen vraagt het qua implementatiewerk weinig moeite, bovendien is het een natuurlijk eenheid, aangezien enerzijds programmeurs nu eenmaal code bijeenstoppen in een procedure die bij elkaar hoort en aangezien anderzijds de vertaler per procedure code genereert.

De volgorde waarin gemarkeerde procedures geselecteerd worden is die waarin ze in het originele programma voorkomen. Gelet op de hoeveelheid hellepijlen die aanwezig is in de grafen van niet-triviale programma's is het vastleggen van andere volgordes die enige gefundeerde criteria in aanmerking nemen immers uiterst moeilijk, zoniet onmogelijk.

In onze prototype compactor is het zo dat procedures bestaan uit dubbel gelinkte lijsten van basisblokken. De blokken zitten in deze lijsten in de volgorde dat ze aangemaakt zijn bij het initiële opbouwen van de graaf en verder naargelang er transformaties op de graaf zijn uitgevoerd, zoals het toevoegen en verwijderen van knopen. Zoals in sectie 2.4.3 gesteld werd, gebeurt het initiële opbouwen van de graaf door de instructies sequentieel te doorlopen en per zogenaamde leiderinstructie een basisblok aan te maken. In theorie hoeft de volgorde waarin de blokken aangemaakt zijn geen enkele correlatie met mogelijke volgorden in de CVG van de procedure te vertonen. In de praktijk is de correlatie wel heel groot. Dit is o.a. het geval omdat de vertaler zal getracht hebben het aantal genomen sprongen (conditioneel of niet) te minimaliseren. Deze minimalisatie is immers een uiterst belangrijke

optimalisatie voor heel wat onderdelen van moderne architecturen. Zo bevordert ze speculatief ophalen van instructies, de prestaties van het tussengeheugen voor instructies, het aantal instructies dat moet uitgevoerd worden, enz. Het rechtstreekse gevolg van deze optimalisatie is dat de blokken grotendeels in uitvoeringsvolgorde in het programma voorkomen: de gerichte pijlen (behalve die van lussen) in de graaf verbinden stukken code met oplopende adressen. En zelfs indien de vertaler deze optimalisatie niet uitvoert zal er een zekere correlatie zijn: de aangegeven volgorde is immers de natuurlijke volgorde waarop vertalers code genereren en plaatsen in het geheugen.

Om de aldus aanwezige correlatie uit te buiten, rekening houdend met het feit dat levensduuranalyse een achterwaartse analyse is (de verzamelde informatie wordt achterwaarts doorgegeven door het programma, d.i. van opvolgers naar voorgangers), volstaat het de gelinkte lijst van basisblokken achterwaarts te doorlopen bij het selecteren van gemarkeerde blokken. Als alle blokken aldus overlopen zijn, en er zijn er nog gemarkeerd, wordt de volledige lijst opnieuw doorlopen. Omdat de blokken dan grotendeels in de volgorde doorlopen worden waarin we ze een plaats in het geheugen gegeven hebben, benut deze methode opnieuw vrij veel lokaliteit.

Ze heeft bovendien als positief neveneffect dat we slechts een bit moeten bijhouden die aangeeft of een blok al dan niet gemarkeerd is. We moeten geen aparte stapel of lijst van gemarkeerde blokken bijhouden, wat geheugen spaart. Alhoewel we dus maar een bit nodig hebben om aan te duiden of een blok al dan niet gemarkeerd is, is het toch nuttig in een byte per blok te voorzien. Zo'n byte kan met één enkele leesoperatie gelezen worden en met één schrijfoperatie op nul of 1 gezet worden. Indien we slechts één bit gebruiken per blok, zijn er allerlei maskeeroperaties nodig om het ene bit te verzetten.

Er weze overigens opgemerkt dat deze aspecten van de implementatie grotendeels ook terugkomen bij andere analyses. Dezelfde redeneringen zijn er dan ook voor geldig.

Het kiezen van procedures als granulariteit van een buitenste lus en de omgekeerd-gealloceerde-volgorde voor het itereren binnen een procedure versnelt de levensduuranalyse met meer dan een factor 5. Voor grotere programma's kan dit oplopen tot meer dan een factor 20. De snelheidswinst die kan gehaald worden uit het optimaliseren van de derde fase zoals in de vorige sectie uiteengezet vervalt hierbij in het niets.

3.1.6 Levensduuranalyse en loze-code-eliminatie

Vergelijking 3.2 geeft een eenvoudig verband tussen de Uit_k verzameling van een blok en de In_k verzameling. Volgens dit verband wordt er geen onderscheid gemaakt tussen loze instructies, die dode waarden produceren, en nuttige instructies, die levende waarden produceren: het verband stelt dat een waarde levend is als ze gebruikt wordt door eender welke instructie. Dit leidt tot nauwkeurigheids- en efficiëntieproblemen.

Voorbeeld 3.3 Beschouw het basisblok met volgende instructies:

$$\begin{aligned} \mathsf{r2} &:= \mathsf{r0} + \mathsf{r1} \\ \mathsf{r2} &:= \mathsf{r0} - \mathsf{r3} \\ \mathsf{r1} &:= \mathsf{0} \end{aligned}$$

Zulk een blok zal een vertaler niet genereren, maar soortgelijke instructiesequenties komen wel voor na allerlei transformaties die we ter compactie van een programma uitvoeren. Het is duidelijk dat de eerste instructie loos is. Zolang deze instructie echter niet verwijderd is uit het programma, zal register r1 als levend beschouwd worden aan het begin van dit blok.

De levensduuranalyse zou in dit geval dus moeten gevolgd worden door een loze-code-eliminatie en een nieuwe levensduuranalyse om nauwkeurige resultaten te verkrijgen.

□

Zoals uit het eenvoudige voorbeeld blijkt is het nodig loze-code-eliminatie en levensduuranalyses iteratief uit te voeren totdat er geen instructies meer verwijderd worden door de loze-code-eliminatie.

Loze-code-eliminatie voor een basisblok is eenvoudig: een algoritme daartoe is weergegeven in figuur 3.6. Met neveneffecten wordt bijvoorbeeld bedoeld het wegschrijven van data naar het geheugen.

Het iteratief moeten afwisselen van analyse en eliminatie is jammer genoeg niet erg efficiënt. Men zou daarom kunnen trachten de vergelijkingen aan te passen om meteen rekening te houden met loze instructies. In plaats van een vergelijking per blok beschouwt men daarvoor

```
1  | Lev = Uit_k[k]
2  | do for all instructies i in k in omgekeerde volgorde
3  |   if {doeloperandi van i} ∈ Lev ∨ i heeft neveneffecten
4  |     Lev := Lev \ {doeloperandi van i}
5  |     Lev := Lev ∪ {bronoperandi van i}
6  |   else
7  |     verwijder i uit het programma
```

Figuur 3.6: Pseudo-code voor loze-code-eliminatie.

een vergelijking per instructie. Voor een instructie i geldt dan

$$\begin{aligned} \mathrm{In_k}[i] &= \mathrm{Cons_k}[i] \cup (\mathrm{Uit_k}[i] \setminus \mathrm{Prod_k}[i]) && (3.33)\\ &\qquad \text{als } \mathrm{Prod_k}[i] \in \mathrm{Uit_k}[i] \\ &\qquad \text{of als } i \text{ neveneffecten heeft} \\ &= \mathrm{Uit_k}[i] && (3.34)\\ &\qquad \text{anders} \end{aligned}$$

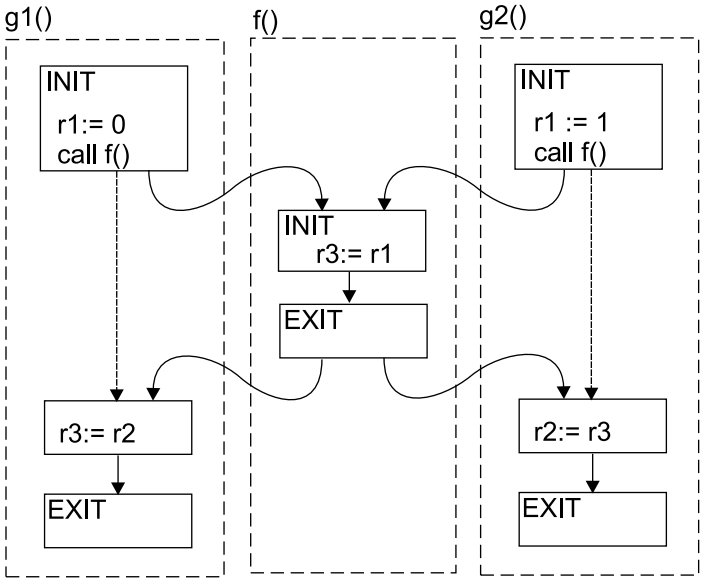
Het evalueren per instructie zal echter veel meer tijd in beslag nemen en is bovendien voor 99% van de geëvalueerde blokken niet nodig. Gelukkig kan men eenvoudig testen of het nodig is om per instructie te evalueren: het volstaat te zien of er in $\mathrm{Prod_k}[k]$ registers zitten die niet in $\mathrm{Uit_k}[k]$ voortkomen.

Alhoewel deze aanpassing eenvoudig lijkt kan men ze niet zomaar toepassen. Wat moet men immers met $\mathrm{Cons_p}$ doen?

Voorbeeld 3.4 Beschouw het programmafragment in figuur 3.7. Het resultaat van de instructie in `f()` wordt niet gebruikt in procedure `g1()` en dus hoeft ook de operand er niet voor klaargezet te worden in `g1()`. De instructie in `f()` is echter niet dood, aangezien haar resultaat wel degelijk gebruikt wordt in `g2()`. Zit `r1` in $\mathrm{Cons_p}$[`f()`]? □

Zoals het voorbeeld laat zien is het niet mogelijk om $\mathrm{Cons_p}$ onafhankelijk van de context waarin een procedure opgeroepen wordt te analyseren als men een instructie loos dan wel levend wil kunnen beschouwen naargelang de oproepcontext.

Daarom hebben we een totaal andere aanpak bekeken, die een contextongevoelige component ($\mathrm{Uit_k}[k]$) bevat en een contextgevoelige



Figuur 3.7: Interprocedurale loze-code-eliminatie.

($\text{Uit}_k[k, c]$) voor een knoop k in procedure p met oproepknoop c. De vergelijkingen zijn:

$$\text{Uit}_k[k] = \bigcup_{c \in \text{Oproepers}[p]} \text{Uit}_k[k, c] \tag{3.35}$$

$$\text{Uit}_k[k, c] = \text{In}_k[k, r] \tag{3.36}$$

als k een uitgangsknoop is
met r de linkknoop van c

$$= \text{In}_k[\text{Init}[f], k] \tag{3.37}$$

als k een oproepknoop is naar proc. f

$$= \bigcup_{l \in \text{Opv}[k]} \text{In}_k[k, c] \tag{3.38}$$

voor alle andere knopen k

$$\text{In}_k[k, c] = \text{evaluatie blok } k \text{ voor } \text{Uit}_k[k, c] \tag{3.39}$$

De evaluatie van een blok k gebeurt nu door de instructies van het blok in omgekeerde volgorde te doorlopen met als regel:

$$\begin{aligned} \mathrm{In_i}[i,c] &= \mathrm{Cons_i}[i] \cup (\mathrm{Uit_i}[i,c] \setminus \mathrm{Prod_i}[i]) && (3.40)\\ &\qquad \text{als } \mathrm{Prod_i}[i] \in \mathrm{Uit_i}[i,c] \\ &\qquad \text{of als } i \text{ neveneffecten heeft} \\ &\qquad \text{of als } i \text{ excepties kan veroorzaken} \\ &= \mathrm{Uit_i}[i,c] && (3.41)\\ &\qquad \text{anders} \end{aligned}$$

Het verschil tussen vergelijkingen 3.33 en 3.40 zit in de instructies die excepties kunnen veroorzaken. Zo zou het best kunnen dat een leesoperatie dood is voor een bepaalde oproepcontext, omdat het resultaat van de leesoperatie toch niet gebruikt wordt na die oproep. Als die leesoperatie niet kan verwijderd worden omwille van andere oproepcontexten, moeten we ervoor zorgen dat in alle contexten toch een geldig basisadres gegenereerd wordt, zodat er bv. geen segmentatiefouten optreden. Deze aanpassing van de vergelijkingen betekent dat voor leesoperaties die wel dood zijn op alle paden, er toch geïtereerd zal moeten worden over levensduuranalyse en loze-code-eliminatie.

Merk op dat we in bovenstaande vergelijkingen geen rekening gehouden hebben met andere interprocedurale sprongen dan procedureoproepen. De knopen waaruit die vertrekken kunnen echter op identiek dezelfde manier als oproepknopen behandeld worden, waarbij dan niet de linkknoop van de oproeper, maar de uitgangsknoop van de procedure waaruit de pijl vertrekt, gebruikt wordt. Bij die pijl hoort immers een compenserende pijl naar dit uitgangsblok, die op dezelfde manier kan behandeld worden als een terugkeerpijl die hoort bij een linkknoop.

Merk tevens op dat alle verfijningen die we tot nu toe hebben voorgesteld voor de oplossing van het contextgevoelige probleem ook hierop kunnen toegepast worden. Zo is het ook bij het oplossen van deze vergelijkingen nuttig mogelijke lokaliteit uit te buiten, en dus per procedure te werken. Daarbij kan men overigens vermijden plaats nodig te hebben voor elke contextgevoelige $\mathrm{Uit_k}[k,c]$ verzameling van een knoop. Men doet dit als volgt:

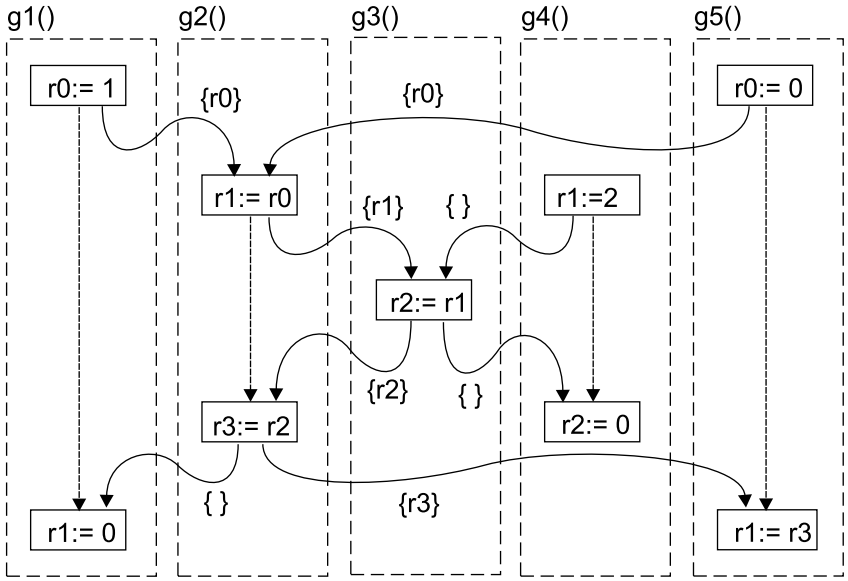
- Gedurende de volledige dekpuntoplossing worden enkel de verzamelingen levende registers over interprocedurale pijlen bijgehouden.

- Als aan één van die verzamelingen horend bij een terugkeerpijl (of een compenserende pijl) iets verandert, wordt deze verandering in de betreffende procedure (d.w.z. waaruit de pijl vertrekt) gepropageerd. Daarbinnen wordt deze verzameling iteratief gepropageerd en de verzamelingen levende registers over inkomende interprocedurale pijlen van die procedure worden aangepast voor de inkomende oproeppijl corresponderend met de terugkeerpijl waarvoor de propagatie werd uitgevoerd, en voor alle compenserende en terugkeerpijlen die deze procedure binnenkomen. Dit is met andere woorden de propagatie van de contextgevoelige component.
- Als aan één van deze verzamelingen horend bij een oproeppijl iets verandert, wordt deze verzameling in de oproepende procedure gepropageerd, daarbinnen iteratief gepropageerd, en tenslotte wordt de intraprocedurale dekpuntoplossing gepropageerd naar alle inkomende pijlen in de oproepende procedure. Dit is m.a.w. de contextongevoelige component.

Men noemt zulk een stelsel van vergelijkingen en de iteratieve oplossing ervan ook wel contextgevoelige oplossingen met diepte 1. Immers, aangezien er voor elke interprocedurale pijl slechts 1 verzameling van levende registers wordt bijgehouden, reikt de contextgevoeligheid maar 1 procedure-oproep ver.

Voorbeeld 3.5 Beschouw de graaf in figuur 3.8. In deze figuur zijn de oproepinstructies weggelaten, evenals de uitgangsknopen, om de figuur niet te overladen. Alle interprocedurale pijlen in de graaf zijn oproep- en terugkeerpijlen. De enige instructie die loos zal zijn volgens de besproken analyse is de definitie van r1 in g4(). Alhoewel het resultaat van de definitie van r0 in g1() eigenlijk dood is op het pad g1() $\rightarrow$ g2() $\rightarrow$ g3() $\rightarrow$ g2() $\rightarrow$ g1(), zal deze analyse de definitie als levend beschouwen. Dit komt doordat de verzameling levende registers $\{$r1$\}$ over de pijl g2() $\rightarrow$ g3() de levende registers voor zowel de oproep uit g1() als die uit g5() moet bevatten. De contextgevoeligheid reikt m.a.w. slechts 1 oproep diep.

Merk op dat het bij de analyses uit sectie 3.1.3 geen zin heeft om over diepte te spreken: voor de graaf in figuur 3.8 zijn $\text{Door}_p[\text{g3()}]$ en $\text{Cons}_p[\text{g3()}]$ onafhankelijk van de overige procedures. $\text{Door}_p[\text{g2()}]$ en $\text{Cons}_p[\text{g2()}]$ hangen dus enkel af van g3()



Figuur 3.8: Contextgevoelige levensduuranalyse met diepte 1.

en $\text{Door}_p[\texttt{g1()}]$ en $\text{Cons}_p[\texttt{g1()}]$ hangen enkel af van `g2()` en `g3()` en niet van `g4()` of `g5()`. □

Omdat de analyses uit 3.1.3 niet beperkt zijn tot diepte 1 zullen ze in heel wat gevallen betere resultaten opleveren dan de analyse met diepte 1. Om dit op te vangen en toch de voordelen van diepte-1 analyse te kunnen benutten, voeren we daarom deze analyse slechts uit na de analyses uit sectie 3.1.3, waarvan we de resultaten permanent als bovengrens gebruiken tijdens deze analyse: een register dat niet levend was tijdens de eerste analyse, kan het ook nooit worden tijdens de tweede analyse. Wel worden er nieuwe dode registers gevonden.

3.1.7 Verwant werk

Intraprocedurale levensduuranalyse is één van de oudste dataverloopanalyses [Aho86, Much97].

Interprocedurale levensduuranalyse toegepast op volledige programma's werd voorgesteld in [Sriv93] door Srivastava en Wall. Ook zij maken gebruik van samenvattende vergelijkingen voor procedures, maar kiezen voor Door_p een andere invulling dan de onze. Zij kiezen

namelijk voor $\text{Door}_p = \overline{\text{Dood}_p}$ waarin Dood_p de verzameling van registers is die zeker dood zijn aan de ingang van een procedure. Dit zijn de registers die in de procedure op alle paden gedefinieerd worden alvorens gebruikt te worden. Deze definitie van Door_p leidt tot dezelfde finale oplossing, doch ze heeft als minpunt dat er een mutuele afhankelijkheid tussen Door_p en Cons_p bestaat. Dit compliceert het oplossen van de vergelijkingen [Muth99].

Goodwin [Good97] gebruikt de oorspronkelijke keuze voor Door_p, en zijn versie is dus verkiesbaar. Muth [Muth99] heeft die verder aangepast zoals ook in dit werk uiteengezet omwille van de uniformiteit van de vergelijkingen. De versnelde derde fase die hij voorstelt en die we ook hier besproken hebben, blijkt geen enkel nut te hebben als we de implementatie-aspecten uit sectie 3.1.5 beschouwen. De verfijningen i.v.m. de helleknopen zijn evenzeer nieuw in dit werk.

3.2 Constantenpropagatie

Net als levensduuranalyse is constantenpropagatie een welbekende analyse in de vertalerswereld [Much97]. We zullen ze vanuit die context inleiden en meteen de verschillen met constantenpropagatie tijdens het linken aangeven.

3.2.1 Inleiding

De bedoeling van constantenpropagatie is om er achter te komen of er veranderlijken zijn die op bepaalde programmapunten een constante waarde hebben en wat die waarden dan wel zijn.

Wanneer men van een veranderlijke in een expressie de waarde kent, zal men de expressie kunnen vereenvoudigen. Kent men de waarden van alle veranderlijken die voorkomen in een expressie, dan zal de expressie zelf herleid kunnen worden tot een constante waarde. De berekening van de expressie wordt dan eenmalig uitgevoerd tijdens het vertalen en moet niet bij elke uitvoering van het betreffend stukje code herhaald worden.

Een andere optimalisatie die voortbouwt op de resultaten van een constantenpropagatie is het gebruik van letterlijke operandi tijdens de registerallocatie en instructieselectie. Als we weten dat een bepaalde veranderlijke op een bepaalde plek een kleine constante waarde heeft,

kunnen we die waarde vaak als letterlijke operand coderen in de instructies die de waarde consumeren. We moeten er m.a.w. geen register voor reserveren.

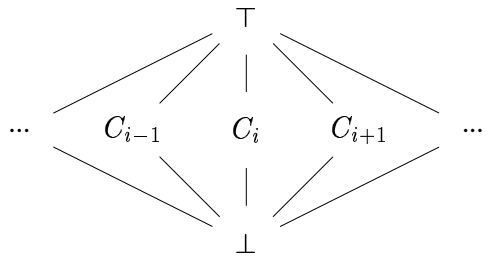
Constantenpropagatie wordt over het algemeen uitgevoerd met een dekpuntalgoritme. De algoritmen waarmee wij zullen werken zijn opnieuw optimistisch: we veronderstellen eerst dat alle veranderlijken een constante waarde kunnen hebben op alle programmapunten, en pas wanneer we ontdekken dat het niet zo is, passen we onze veronderstelling aan.

De tralie die meestal gebruikt wordt, is afgebeeld in figuur 3.9. Ze bestaat uit alle voorstelbare getallen C_i, en de elementen top ($\top$) en bodem ($\bot$). Als een veranderlijke op een programmapunt met een waarde C_i geassocieerd is, betekent dit dat de veranderlijke op dat programmapunt de constante waarde C_i heeft. Is de veranderlijke er met $\top$ geassocieerd, dan betekent dit dat we nog geen definities van die veranderlijke gezien hebben en er dus nog geen bepaalde waarde mee kunnen associëren. Een associatie met $\bot$ betekent dan weer dat we verscheidene producenten gezien hebben die de veranderlijke een verschillende waarde geven die dat programmapunt bereiken, zodat we kunnen besluiten dat de veranderlijke geen constante waarde zal hebben op dat punt.

De tralie-elementen zullen tijdens het dekpuntalgoritme voorwaarts gepropageerd worden, van producenten naar consumenten in het programma dus. Daar waar verschillende waarden samenkomen (d.i. waar verscheidene pijlen in de ICVG samenkomen), worden de waarden samengevoegd volgens de regels die eveneens in figuur 3.9 weergegeven zijn. Zij geven meer formeel de betekenis van $\bot$ en $\top$ aan.

Bij een optimistisch algoritme worden alle veranderlijken initieel op alle programmapunten met $\top$ geassocieerd, behalve op het beginpunt van het programma: daar hebben ze allemaal de waarde $\bot$.

De algoritmen die we geëvalueerd hebben in het kader van dit proefschrift zijn conditionele constantenpropagaties [Wegm91]. Dit betekent dat bij conditionele sprongen de tralie-elementen enkel gepropageerd worden in de richting die de sprong kan uitgaan. Kan men uit de waarde van de veranderlijken in de conditietest niet afleiden welke richting de sprong zal uitgaan, dan wordt in beide richtingen gepropageerd. Het is algemeen bekend dat dit nauwkeuriger resultaten oplevert dan de toepassing van gescheiden onbereikbare-code-



$$\bot \sqcap C_i = \bot$$
$$\top \sqcap C_i = C_i$$
$$C_i \sqcap C_j = \bot \Leftrightarrow i \neq j$$
$$C_i \sqcap C_j = C_i \Leftrightarrow i = j$$

Figuur 3.9: De tralie gebruikt bij constantenpropagatie.

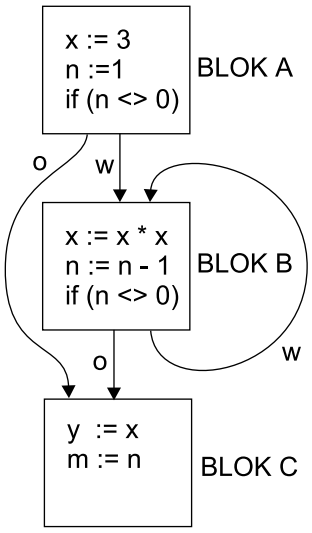
eliminaties en eenvoudige constantenpropagaties [Clic95].

Voorbeeld 3.6 In figuur 3.10 is een kleine graaf met enkele instructies weergegeven. Met de hierboven vermelde evaluatie van conditionele expressies wordt bedoeld dat de over de pijlen die met o (onwaar) geannoteerd zijn de gepropageerde waarde van n 0 (nul) is. Uit de testen kan men immers afleiden dat dit de enige waarde van n is waarvoor die pijlen gevolgd worden.

Als men simpele constantenpropagatie toepast, zal de waarde van y op het einde onbekend zijn. De reden is als volgt: de waarde 3 die uit blok A gepropageerd wordt voor x, geeft aanleiding tot de waarde 9 die voor x terug naar de ingang van blok B wordt gepropageerd. Daar botst ze echter op de oorspronkelijke waarde 3, dus vanaf dan wordt ⊥ voor x in blok B gepropageerd, en verder naar blok C.

Past men echter conditionele constantenpropagatie toe, dan zal voor de inkomende waarde 1 voor n in blok B de sprong op het einde van blok B niet genomen worden. Nu houden we hier rekening mee en we propageren dus niks meer over de pijl naar boven, waardoor de finale inkomende waarden in blok B de uitgaande waarden van blok A zijn. De waarden zijn dan bekend aan het eind van blok B en dus ook in blok C, aangezien ook niks gepropageerd is over de o-pijl uit blok A. □

Net als in [Wegm91] worden conditionele expressies bovendien geëvalueerd: als een conditionele sprong bv. enkel genomen wordt indien een bepaald register de waarde 0 bevat, dan wordt over de corresponderende pijl de waarde 0 gepropageerd, of we de inhoud van het register vóór die sprong nu kennen of niet.



Figuur 3.10: Voorbeeld ter verduidelijking van conditionele constantenpropagatie.

3.2.2 Constantenpropagatie tijdens het linken

Dit verschilt op een aantal vlakken van constantenpropagatie tijdens het vertalen. We werken immers niet meer met veranderlijken, maar met registers, data-adressen en statisch gealloceerde data. Daarom gaan we eerst wat dieper in op de verschillen, waarbij we meteen de doelstellingen van constantenpropapagatie tijdens het linken kunnen duiden.

Aliassen Wanneer er in de broncode van een programma wijzers voorkomen, moet men rekening houden met mogelijke aliassen, d.w.z. dat verschillende wijzers die gebruikt worden om te lezen en te schrijven naar hetzelfde adres kunnen wijzen. De literatuur over aliassen bestrijkt reeds verscheidene boekenplanken, omdat aliassen op heel veel optimalisatietechnieken invloed hebben.

Na het linken is de situatie iets anders: daar waar wijzers in de broncode van een programma expliciet en dus herkenbaar voorkomen naast het gewone gebruik van veranderlijken, is dit na het linken niet meer het geval: de enige manier om data aan te spreken in het geheugen is via wijzers, m.a.w. adressen die in registers opgeladen worden. Waar vertalers vaak een onderscheid kunnen maken tussen indexbe-

werkingen en echte wijzerberekeningen, kan dit bovendien na het linken nog nauwelijks, aangezien ook de indexbewerkingen door de vertaler geïmplementeerd zijn als wijzerberekeningen.

Hierdoor is het zeer moeilijk, zo niet onmogelijk een nauwkeurige aliasanalyse uit te voeren tijdens of na het linken (zie ook sectie 3.4.1). Voor een constantenpropagatie heeft dat als gevolg dat we zullen veronderstellen dat alle geheugentoegangen (behalve die naar vaste locaties in het geheugen) alle geheugenlocaties kunnen aanspreken of beschrijven. In concreto betekent dit dat we niet weten welke waarden door welke instructies weggeschreven worden in het geheugen. We kunnen een door een instructie weggeschreven waarde dus ook niet propageren naar de instructies die die waarde opladen (als we die instructies al zouden kunnen identificeren). We zullen daarom alleen maar registerinhouden propageren. Daartussen zijn zeker geen aliassen te vinden, aangezien er geen wijzers naar registers bestaan.

Een uitzondering is natuurlijk het tekstsegment: aangezien we daarvan op voorhand weten dat de statisch gealloceerde data niet zal overschreven worden, kunnen we ze in het programma propageren als ze opgeladen wordt door instructies die laden van op een door ons bekend (en dus constant) adres.

Een tweede uitzondering is de stapel. Net zoals bij levensduuranalyse kunnen we in zekere mate achterhalen welke registers onveranderd doorgegeven worden over procedure-oproepen heen. Als die registers een constante waarde bevatten voor de oproep, doen ze dit ook na de oproep, ongeacht het feit of de waarde tijdelijk op de stapel is bewaard. We kunnen dit makkelijk modelleren door die waarden langs de linkpijlen te propageren i.p.v. langs de oproep- en terugkeerpijlen.

Adresberekeningen Constantenpropagatie tijdens het vertalen is erop gericht waarden van veranderlijken te propageren. De propagatie gebeurt immers op het intermediaire codeniveau, waar nog steeds symbolische namen voor veranderlijken gebruikt worden. Ze hebben met andere woorden nog geen plaats in het geheugen gekregen.

Constantenpropagatie tijdens of na het linken werkt met code op een veel lager niveau, waarin ook de berekeningen op adressen van data opgenomen zijn. Deze berekeningen zijn net zo goed als de “echte” berekeningen op data kandidaten voor constantenpropagatie. Meer nog, het zijn uitstekende kandidaten. Na het linken kennen we immers, in tegenstelling tot de vertaler, de adressen van statisch

gealloceerde objecten, en kunnen van deze kennis gebruik maken om de adresberekeningen te optimaliseren. In het bijzonder zullen relaties tussen adressen die de vertaler nog niet kende nu nuttig aangewend kunnen worden. Wanneer in een klein stukje code ogenschijnlijk onafhankelijk van elkaar gegenereerde adressen dicht genoeg bij elkaar liggen in de adresruimte, zal het vaak mogelijk zijn adressen op eenvoudige wijze af te leiden uit elkaar, zodat ze niet allemaal vanaf nul opnieuw gegenereerd moeten worden.

Dit is met name het geval voor adressen die dicht bij de globale wijzer liggen, maar, omdat de vertaler dat niet wist, toch nog opgeladen worden uit de globale adrestabel. Dit opladen kan dan vermeden worden door het opgeladen adres rechtstreeks te berekenen aan de hand van de globale wijzer. Vaak zal zelfs helemaal geen berekening van het adres meer nodig zijn: bv. als voor alle consumenten van het adres de letterlijke afstand die in de instructies gecodeerd is kan omgezet worden in de corresponderende afstand tot de globale wijzer. Dit is overigens ook mogelijk voor relatieve adressen t.o.v. de programmateller. Waar de afstand tussen oproeper en opgeroepene in een directe oproepinstructie kan gecodeerd worden, moet het adres van de opgeroepen procedure niet meer opgeladen of berekend worden.

Om de mogelijkheden hiertoe te ontdekken volstaat het de opgeladen adressen door te propageren naar hun consumenten: de jsr-instructies, de indirecte leesinstructies, etc.

Merk op dat de omzetting van indirecte naar directe procedureoproepen niet alleen een optimalisatie van het programma inhoudt, maar dat dit bovendien van kapitaal belang is voor onze compactie in het algemeen: deze omzetting biedt ons immers zoals in sectie 2.5.2 aangehaald de mogelijkheid om de ICVG te verfijnen.

Dode waarden Tijdens een constantenpropagatie in een vertaler kan het gebeuren dat een veranderlijke die niet gebruikt wordt in een stuk code (en er ogenschijnlijk niks mee te maken heeft) een waarde bevat, waaruit heel makkelijk een andere waarde kan berekend worden, een waarde die wel vereist is in het stukje code. De vertaler zal van deze mogelijkheid geen gebruik maken, omdat hij bijzonder moeilijk de kost kan inschatten van het gebruik van die eerste veranderlijke. Zoals reeds gezegd voeren vertalers immers traditioneel een constantenpropagatie uit op het intermediaire codeniveau. Dit gebeurt vóór het inroosteren en vóór registerallocatie en instructieselectie [Aho86] uitgevoerd wor-

den. De vertaler weet dus niet of hij de te gebruiken waarde in een register of in het geheugen zal vinden. Indien die waarde in het geheugen opgeslagen ligt, zal hij een dure leesoperatie moeten gebruiken om ze op te laden, en bovendien moet er dan een register opgeofferd worden om er de waarde in op te laden. Of het gebruik van een extra register de overige berekeningen moeilijker implementeerbaar zal maken kan de vertaler evenmin inschatten.

Voor een vertaler is het dan ook niet nuttig dode waarden te propageren. Het volstaat waarden te propageren van producenten naar consumenten, die dus per definitie levend zijn. Tijdens of na het linken kan het propageren van dode waarden wel nuttig zijn: aangezien we enkel registerinhouden propageren, weten we dat de aldus gepropageerde waarden in registers zitten, en dus zeer goedkoop hergebruikt kunnen worden waar dat nuttig is.

Daarom willen we alle registerinhouden overal propageren, ook die van dode waarden. Dit zal ons meer geheugen kosten tijdens de compactie omdat we meer waarden moeten bijhouden. Maar als we dan toch al van meer registers de waarde bijhouden, is de extra kost om voor een vaste, direct adresseerbare voorstelling van de registerinhouden te kiezen, die de propagatie zal versnellen, relatief klein. We kunnen dan bv. een rij van registerinhouden bijhouden per programmapunt i.p.v. een gelinkte lijst.

3.2.3 Contextongevoelige constantenpropagatie

Anders dan bij de levensduuranalyse volstaat het nu niet een verzameling registers bij te houden voor elk programmapunt. We kunnen dus niet eenvoudig met een bitvector werken. We moet integendeel voor elk register een waarde (een tralie-element) bijhouden. De geheugenvereisten zijn dus aanzienlijker dan bij de levensduuranalyse. Om hieraan tegemoet te komen zijn er verschillende mogelijkheden:

- We beperken het aantal registers waarvan we de waarde bijhouden tijdens de propagatie. Daartoe propageren we op elk tijdstip bv. slechts één register doorheen het hele programma. Als er na zo'n propagatie instructies zijn die dat register gebruiken en waarvan dat register een nieuwe ermee geassocieerde waarde (tralie-element) gekregen heeft, zullen verdere constantenpropagaties uitgevoerd worden, nu voor de doelregisters van deze instructies. Dit gaat voort tot er geen operandi van instructies

van waarde veranderen. Het nadeel van deze methode is dat we vele propagaties moeten uitvoeren, die telkens over het hele programma gaan. Het voordeel is dat we per basisblok slechts één gepropageerde waarde moeten bijhouden, plus de waarden van de operandi van alle instructies. Aangezien we de laatste toch al moeten bijhouden om allerlei transformaties te kunnen uitvoeren, vraagt deze methode dus bijzonder weinig geheugen.

- We beperken het aantal programmapunten waarvoor we de waarden bijhouden tijdens de propagatie. Zoals we met de levensduuranalyse voor loze-code-eliminatie gedaan hebben, kunnen we bv. enkel de waarden over interprocedurale pijlen gedurende de hele propagatie bijhouden. Telkens er aan de waarden over een inkomende interprocedurale pijl iets verandert, zullen die waarden dan doorheen de hele procedure gepropageerd worden, en zullen de waarden over de uitgaande pijlen aangepast worden. Deze methode heeft als voordeel de lokaliteit, aangezien we nu ook vele propagaties zullen moeten uitvoeren, maar dat dit telkens per procedure gebeurt. Het nadeel van deze methode is evenwel het hogere geheugenverbruik.
- We combineren beide voorgaande opties.

Beide extreme opties hebben we geïmplementeerd in ons prototype. Daaruit bleek de tweede optie veruit verkiesbaar als men de uitvoeringstijd wil beperken: ze is tot 5 maal sneller! Net als bij levensduuranalyse blijkt het opsplitsen van de propagatie per procedure een enorme snelheidswinst met zich mee te brengen, omwille van de lokaliteit. Bovendien blijkt het qua tijdsgebruik veel voordeliger om dan meteen alle registers tegelijkertijd door een procedure te propageren dan dit register per register te doen. Per blok van een procedure moet er dan meer data in de tussengeheugens opgeladen worden, maar er moet minder over de blokken geïtereerd worden.

Daarnaast kunnen we nog volgende praktische opmerkingen maken omtrent het implementeren van zulk een constantenpropagatie:

- Omtrent de voorstellingswijze van de tralie-elementen kunnen we kort zijn: indien men alle registers tegelijkertijd propageert, is een rij van waarden die bestaat uit getallen met eenzelfde woordbreedte als de breedte van de registers het meest efficiënt. Daarnaast houdt men twee bitvectoren bij die de registers aangeven waarvan de waarde $\bot$ resp. $\top$ is. Vaak kunnen dan registers

parallel behandeld worden bij het toepassen van de samenvoegingsregels door bitsgewijze operaties op de bitvectoren toe te passen. Deze bitvectoren ontslaan ons ook van het kiezen van een expliciete voorstelling voor de waarden $\bot$ en $\top$.
Indien men register per register propageert is het minder efficiënt om voor het register bitjes bij te houden die aangeven of de waarde $\bot$ of $\top$ is. Een veel efficiëntere voorstellingswijze kan men bereiken door $\top$ en $\bot$ voor te stellen door twee getallen die relatief weinig kans hebben om voor te komen in een programma. Indien die getallen toch voorkomen (men kan daarop testen na het uitvoeren van de propagatie door te kijken of er instructies zijn die die getallen produceren), dan kiest men gewoon andere zeldzame getallen. Op deze manier is het niet nodig extra bits te lezen of te schrijven.

- Omtrent de volgorde waarin procedures en basisblokken gekozen worden tijdens het itereren kan een identieke redenering als bij de levensduuranalyse gehanteerd worden. Wel zullen de basisblokken van een procedure nu van begin naar eind doorlopen worden, aangezien constantenpropagatie een voorwaarts probleem is.

- Net zoals bij de levensduuranalyse volstaat het per blok één verzameling waarden bij te houden. Aangezien het hier een voorwaarts probleem betreft, zullen we dus de ingangsverzamelingen bijhouden en er tijdens het intraproceduraal itereren de uitgangsverzamelingen uit berekenen.

- Voor de evaluatie van instructies tijdens de constantenpropagatie bestaan verschillende mogelijkheden. Men kan ze bv. interpreteren. Indien dan de bronoperandi van instructies bekend zijn en constante waarden hebben, zal ook de doeloperand een constante waarde hebben. Een uitzondering is natuurlijk de leesoperatie: daar moet niet alleen het adres waarop gelezen wordt bekend zijn, ook de data op die locatie moet constant zijn en bekend. Zulk een aanpak vereist de implementatie van nogal wat interpretatiewerk: voor elke instructie moet code geschreven worden die exact berekent wat de instructie berekent.

  Om dit te vermijden kan men voor een andere oplossing kiezen: directe uitvoering via zelfwijzigende code. Daartoe implementeert men in de compactor een procedure beschikken die pre-

cies één instructie uitvoert (naast de terugkeerinstructie natuurlijk). Vooraleer deze procedure op te roepen overschrijft men die ene instructie met de opcode van de uit te voeren instructie. Deze opcode is zo gekozen dat de bronoperandi van de instructie de argumentregisters zijn en de doeloperand het register waarin de functiewaarde wordt opgeslagen. De instructie wordt dan geëvalueerd met een eenvoudige procedure-oproep met als argumenten de waarden van de bronoperandi. Het nadeel hiervan is dat voor de evaluatie van elke instructie het tussengeheugen voor instructies moet geledigd worden.
Om dit te vermijden, hebben we alle instructies in een tabel opgeslagen. Voor elke instructie wordt bij de aanvang van de compactie precies één procedure aangemaakt die de instructie uitvoert.

- Conditionele constantenpropagatie wordt behalve om constanten te vinden, ook gebruikt om onbereikbare code te detecteren. Dit wordt gedaan door alle code behalve het beginpunt van het programma initieel als onbereikbaar te beschouwen, en deze veronderstelling pas op te geven als er waarden naar een stuk code gepropageerd worden: de code wordt hierdoor meteen bereikbaar.

  Daarbij moet men natuurlijk wel oppassen met de helleprocedures. Sommige procedures kunnen in de graaf enkel door de oproepershelleprocedure opgeroepen worden. Om te vermijden dat we die procedures als onbereikbaar gaan beschouwen, volstaat het om de helleprocedures initieel als bereikbaar te markeren. Met betrekking tot de waarden die uit de helleknopen gepropageerd worden kunnen we kort zijn: alle waarden die uit deze knopen gepropageerd worden zijn op $\perp$ geïnitialiseerd: we kennen deze waarden immers niet en moeten dus conservatief veronderstellen dat ze niet constant zullen zijn.

- Bij de propagatie van achteraf te bewaren registers moeten we voorzichtig te werk gaan. Bij een contextgevoelige levensduuranalyse worden deze registers over de linkpijlen gepropageerd i.p.v. doorheen de opgeroepen procedures (zie sectie 3.1.4). Stel dat we de constante waarde van een dood register dat in een procedure op de stapel bewaard wordt wel in de procedure propageren en de procedure daarna optimaliseren door van die constante waarde gebruik te maken. Vanaf dan moeten we het register als levend beschouwen, en mogen we het register niet meer als achteraf te bewaren beschouwen, waardoor we alle verfijningen aan

de diverse analyses op basis van het achteraf te bewaren zijn niet meer kunnen toepassen. Om dit te vermijden propageren we de op de stapel bewaarde registers van een procedure altijd over de linkpijlen van de oproepers van die procedure. Zelfs onze contextongevoelige constantenpropagatie is dus gedeeltelijk contextgevoelig. Deze gedeeltelijke contextgevoeligheid komt de uitvoeringssnelheid van de constantenpropagatie gelukkig alleen maar ten goede, omdat deze constanten aldus over minder code moeten gepropageerd worden.

3.2.4 Contextgevoelige constantenpropagatie

Net zoals bij de levensduuranalyse, kunnen we ook een constantenpropagatie contextgevoelig maken. De terugkeerwaarden[4] van een procedure-oproep zullen dan slechts in beperkte mate vertekend worden door niet-realiseerbare paden. Net als bij levensduuranalyse zullen de niet-realiseerbare paden toch nog enige invloed hebben, omdat we ook hier de diepte van de contextgevoelige analyses noodgedwongen tot 1 zullen moeten beperken. Diepere analyses zouden zowel qua geheugen als qua rekentijd te hoge eisen stellen.

In tegenstelling tot de levensduuranalyse echter zijn een aantal binnenwegen niet voorhanden:

- Als we de contextongevoelige analyse nauwkeuriger willen maken, kunnen we enkel steunen op zogenaamde symbolische uitvoering van opgeroepen procedures. Het is m.a.w. niet mogelijk om zoals we voor de levensduuranalyse gedaan hebben samenvattende vergelijkingen per procedure op te stellen. Niet alleen zouden zulke vergelijkingen bijna even complex zijn om te evalueren als de symbolische uitvoering van de procedure zelf, bovendien zouden ze de detectie van onbereikbare code belemmeren door conditionele propagatie.

- Om die laatste reden kunnen we evenmin de te propageren informatie splitsen in een contextongevoelige component en een contextgevoelige component. De contextgevoelige component die doorheen een procedure moet gepropageerd worden zou nu onder meer uit de waarden over binnenkomende terugkeerpijlen

[4] Dit zijn er verscheidene, aangezien we alle waarden in registers als terugkeerwaarden kunnen beschouwen.

bestaan. Een vraag die zich dan stelt is naar welke uitgaande terugkeerpijlen we deze informatie moeten propageren. De binnenkomende terugkeerpijl correspondeert met een uitgaande oproeppijl, en de uitgaande terugkeerpijlen corresponderen met binnenkomende oproeppijlen. Aangezien we een conditionele constantenpropagatie wensen uit te voeren, mogen we er niet van uitgaan dat de uitgaande oproeppijl een realiseerbare pijl is voor alle binnenkomende oproeppijlen. We weten dus niet naar welke uitgaande terugkeerpijlen we de informatie die over een terugkeerpijl binnenkomt moeten door propageren.

We hebben dan ook gekozen voor een tragere, zij het nauwkeuriger implementatie, waarin enkel contextgevoelige propagaties doorheen procedures uitgevoerd worden. Per binnenkomende pijl in een procedure waarop een waarde verandert tijdens de propagatie zal dus een volledig aparte intraprocedurale propagatie uitgevoerd worden.

Uit metingen blijkt dat de contextgevoelige constantenpropagatie geen extra compactie met zich meebrengt. Dit lijkt eigenaardig, en is het in zekere mate ook. De reden is niet dat de contextgevoelige constantenpropagatie niet meer constanten zou vinden. Er worden integendeel wel meer constanten gevonden, en het programma wordt extra geoptimaliseerd aan de hand van de extra gevonden constanten. Dit gaat gepaard met een klein aantal extra geëlimineerde instructies. Jammer genoeg bemoeilijkt de optimalisatie van de code op die manier het factoriseren van identieke stukken code (zie hoofdstuk 4). Daar waar een vertaler identieke code had gegenereerd, leiden de optimalisaties aan de hand van de contextgevoelige resultaten tot verschillende stukken code. Codefactorisatie presteert hierna minder goed en dit geeft uiteindelijk aanleiding tot een grotere compactiefactor. Merk op dat het hier een fundamenteel dilemma betreft: specialistische code is sneller maar minder goed factoriseerbaar.

Omdat een contextgevoelige constantenpropagatie uiteraard wel aanleiding geeft tot een grotere rekentijd kunnen we voorlopig besluiten dat ze niet nuttig is voor programmacompactie tijdens het linken.

3.2.5 Propagatie met laat samenvoegen

Naast het inbrengen van contextgevoeligheid in de propagatie, kan men nog op een andere manier de nauwkeurigheid van een analyse trachten te vergroten.

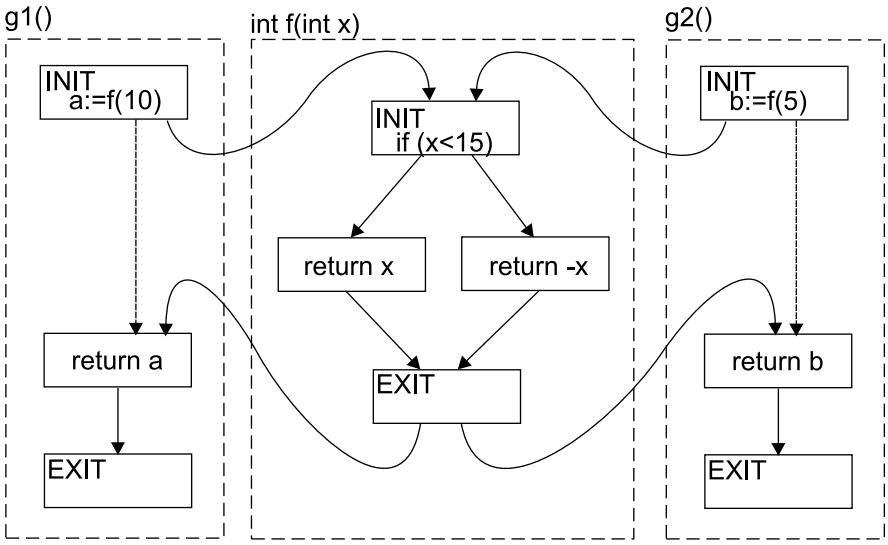
Voorbeeld 3.7 Beschouw de graaf in figuur 3.11. Bij een contextgevoelige propagatie zullen de waarden van a en b op het eind van g1() en g2() bekend zijn. Bij het propageren van de waarden 10 en 5 doorheen f() is telkens hetzelfde pad gevolgd. Om nu te weten te komen wat er constant is binnen f() zelf, moeten we de waarden 10 en 5 beide door de procedure propageren.

De makkelijkste manier is om deze waarden aan de ingang van de procedure samen te voegen met de samenvoegregels uit figuur 3.9: $10 \sqcap 5 = \bot$. We zien dat x bij het binnenkomen van f() geen constante is, en dus zal in de procedure f() nergens een constante gevonden worden. Aangezien we aan de conditionele sprong de waarde van x niet kennen, zal deze niet omgezet worden tot een gewone sprong. Na de convergentie van de intraprocedurale propagatie kunnen we de waarden van de operandi van alle instructies op $\bot$ zetten voor verder gebruik tijdens de compactie.

We kunnen het vastleggen van de waarden van de operandi van instructies ook anders aanpakken. Eerst propageren we bv. enkel de waarde 5 doorheen de procedure. Met het resultaat van de intraprocedurale propagatie leggen we de waarden van de operandi in elke instructie vast. We houden voor de conditionele sprong ook bij in welke richting hij geëvalueerd is (bv. 0 voor niet genomen, 1 voor wel genomen, waarbij we de richting van de sprong zelf m.a.w. ook een te evalueren constante maken). Daarna propageren we de waarde 10 doorheen de hele procedure. Als die propagatie geconvergeerd is, passen we voor elke operand van de instructies de geassocieerde waarde als volgt aan: voeg de oude waarde (in het voorbeeld overal 5) samen met de nieuwe waarde (10) volgens de regels uit figuur 3.9. Pas deze regel ook toe op de richting waarin de sprong genomen werd. Het eindresultaat hiervan zal zijn dat we nu evenmin de waarde van x kennen doorheen de procedure, maar dat we nu wel weten dat de conditionele sprong altijd in dezelfde richting zal genomen worden. We kunnen hem dan elimineren.

□

In het algemeen kunnen we de contextgevoelige analyse verfijnen door de determinatie van constanten binnen een procedure met verschillende oproepcontexten met late samenvoeging te doen: i.p.v. de binnenkomende waarden samen te voegen aan de ingang van de pro-



Figuur 3.11: Voorbeeld ter verduidelijking van het begrip late binding.

cedure, worden ze apart doorheen de procedure gepropageerd en de resulterende waarden worden samengevoegd bij elke instructie.

Merk op dat men deze verfijning ook kan toepassen op een context-ongevoelige constantenpropagatie: nadat de gepropageerde waarden over de interprocedurale pijlen geconvergeerd zijn naar hun dekpunt-oplossing, voert men dan één propagatie per inkomende pijl uit. Hoe men daarvoor de waarden over de interprocedurale pijlen laat convergeren (contextgevoelig of niet) staat er volledig los van.

Deze verfijning zal eveneens extra tijd kosten, maar veel minder dan wat een contextgevoelige analyse nodig heeft, aangezien er voor elke procedure-oproep hoogstens één aparte intraprocedurale propagatie moet plaatsvinden. De vereiste rekentijd is echter toch nog zo groot, dat die niet verantwoord is, gelet op de beperkte extra compactie die ermee gehaald kan worden.

3.2.6 Optimalisaties met constanten

De resultaten van de beschreven constantenpropagaties worden in diverse programmatransformaties benut. De belangrijkste zijn:

- de verfijning van de verloopgraaf (sectie 2.5);

- de detectie van dode statisch gealloceerde data in het programma (sectie 3.3);
- de omzetting van indirecte datatoegangen naar directe toegangen;
- de daarbij horende optimalisatie van het gebruik van de globale wijzer.

Merk op dat we verder in deze sectie besproken transformaties nooit op berekeningen met code-adressen uitvoeren. Zoals in sectie 2.3.5 geduid is, kunnen niet al onze technieken zomaar omgaan met berekeningen op code-adressen. We hebben toen programma's met zulke berekeningen uitgesloten. Aangezien de hier besproken transformaties precies de productie en consumptie van constanten optimaliseren door ze van elkaar af te leiden, zouden we zelf berekeningen op code-adressen invoeren indien we de transformaties toepassen op instructies die code-adressen opladen. Op de door ons gebruikte architectuur voor de prototype compactor is het overigens triviaal om na te gaan of een constante een code-adres kan zijn of niet: kijk of zijn waarde binnen de codesecties valt. Indien dit zo is, beschouwen we de waarde als een code-adres. Het is natuurlijk mogelijk dat er toevallig een constante die geen adres voorstelt binnen dit interval valt. Die kans is echter relatief klein en resulteert steeds in conservatieve resultaten. Bovendien zullen we op die groep adressen toch weinig optimalisaties kunnen uitvoeren, aangezien het tekstsegment in de systeemomgeving van ons prototype (Tru64 Unix voor de Alpha architectuur) op adres 0x120000000 begint. Indien de codesecties op adres nul beginnen ligt het moeilijker, aangezien dan de verwarring tussen mogelijke letterlijke operandi en code-adressen maximaal zal zijn. Dit kan men dan oplossen door code-adressen, die sowieso herkenbaar zijn a.d.h.v. de relocatiegegevens, een annotatie mee te geven en deze annotatie mee te propageren.

Producenten van constanten

Een van de resultaten van de constantenpropagatie is de detectie van instructies die constante waarden in registers produceren. Heel vaak betreft het hier adressen van data. Deze adressen worden zelf vanuit de globale adrestabel opgeladen.

Indien mogelijk proberen we deze instructies te vervangen door goedkopere instructies. Daarbij trachten we in de eerste plaats de re-

gisters met een vaste inhoud te gebruiken. Dit zijn de globale wijzer of het register dat steeds de waarde nul bevat. Vaak volstaat een operatie van het type lda waarbij een afstand bij een getal in een ander register kan opgeteld worden. De enige voorwaarde hiertoe is dat de afstand klein genoeg is om letterlijk in een instructie zoals lda te coderen.

Deze optimalisatie werd al door Muth [Muth99] beschreven. Hij heeft bij de optimalisatie van de generatie van constanten echter enkel lokale informatie ter beschikking: hij gebruikt enkel die waarden opnieuw die al in een blok gebruikt worden. De reden is dat hij na zijn constantenpropagatie niet op een makkelijke manier kan beschikken over de binnenkomende waarden in een blok. De enige constantenpropagatie die hij geïmplementeerd en geëvalueerd heeft propageert immers slechts één register tegelijkertijd: er is dus nooit een globaal overzicht van alle waarden die een blok binnenkomen. Met de techniek waarin alle registers tegelijkertijd gepropageerd worden is het triviaal alle waarden die een blok binnenkomen te achterhalen na het convergeren van de finale intraprocedurale propagatie: het finale resultaat bestaat immers net uit de verzameling binnenkomende waarden.

Indien er verscheidene registers zijn waaruit we makkelijk de geproduceerde waarde kunnen berekenen kiezen we bij voorkeur de registers die sowieso levende waarden bevatten. Dan verlengen we de levensduur van dat register niet, en zullen we dus ook geen potentiële eliminatie van loze code in de weg staan.

Merk op dat het vervangen van leesoperaties door een "goedkopere" instructie niet rechtstreeks een compactie meebrengt. Het zal normaal gezien het programma wel versnellen (zoals in [Muth99] aangetoond). En ook onrechtstreeks kan het een programma kleiner maken: indien men alle operaties die bepaalde statisch gealloceerde, constante data inlezen kan verwijderen, wordt ook die data dood, en kan men ze eventueel uit het programma verwijderen.

Idempotente instructies

Soms kan men instructies gewoon weglaten: namelijk als zij een waarde in een register genereren die er gegarandeerd al in zat. We noemen zulke instructies *idempotente instructies*. De geproduceerde waarde hoeft geen constante te zijn. In het geval van late samenvoeging kan het zijn dat voor al de oproepcontexten een instructie een waarde overschrijft met dezelfde waarde, die echter afhankelijk is van

de oproepcontext. Het volstaat zulke instructies te markeren tijdens de late samenvoeging om ze te detecteren en eventueel te verwijderen. Bij het verwijderen van idempotente, constanten genererende instructies moet men echter zorgvuldig te werk gaan.

Voorbeeld 3.8 Beschouw de graaf in figuur 3.12. % is de modulo-operator. De originele graaf staat bovenaan, de door eliminatie van idempotente instructies geoptimaliseerde graaf onderaan. Alle instructies in blok E zijn idempotent. Laten we één voor één de eliminatie van die instructies bekijken:

w := w % 8 Deze idempotente instructie kan men elimineren. Als men dat doet, moet men echter de definities van w in blokken B en C laten staan, wat dus 1 instructie meer zou opleveren. Beter is het dus deze instructie te vervangen door een andere instructie, die de definities van w in B en C dood maakt.

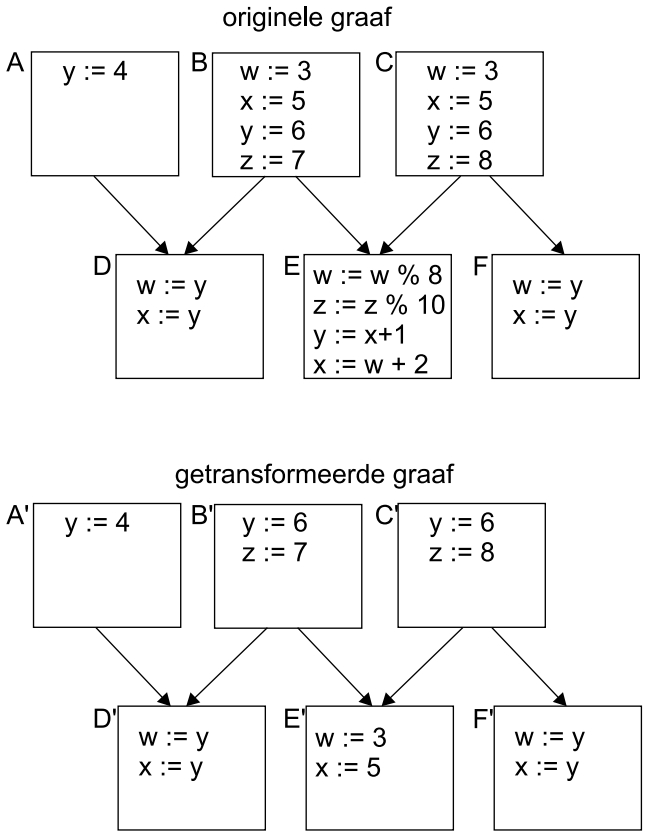
z := z % 10 Deze idempotente instructie lijkt sterk op de voorgaande. In dit geval is het echter onmogelijk ze te vervangen door een eenvoudige instructie die een constante genereert. We elimineren ze dus.

y := x + 1 Deze instructie is idempotent, en we kunnen ze vervangen door een eenvoudiger instructie. Er worden echter geen instructies loos als we hier een eenvoudiger instructie gebruiken, aangezien de definities van y in B en C geconsumeerd worden in D en F. We kunnen deze instructie dus beter elimineren.

x := w + 2 Deze instructie is eveneens idempotent en we kunnen ze vervangen door een eenvoudiger instructie. Als we ze vervangen door een eenvoudiger instructie, kunnen we de definities van x in B en C echter elimineren. In dit geval opteren we dus opnieuw voor het vervangen van de idempotente instructie door een andere instructie.

□

Zoals in het voorbeeld, is het mogelijk om door de eliminatie van een idempotente instructie andere instructies nuttig i.p.v. loos te maken: de waarde die de idempotente instructie overschreef blijft nu immers langer levend. In onze prototype compactor hebben we dit probleem opgevangen door van meer dan één constantenpropagatie uit



Figuur 3.12: Voorbeeld van optimalisatie van idempotente instructies.

te gaan. Zoals in hoofdstuk 5 zal blijken is dit geen vreemd uitgangspunt. Voor een goede compactie moeten we de diverse analyses sowieso meermaals (iteratief) toepassen.

De oplossing is dan als volgt: als we een idempotente instructie tegenkomen die een constante waarde genereert, proberen we die instructie eerst te vervangen door een goedkopere instructie. Indien dit niet lukt elimineren we ze. Er zijn twee mogelijkheden:

1. We hebben een eenvoudiger instructie kunnen genereren. Een loze-code-eliminatie die uitgevoerd wordt voor de volgende constantenpropagatie zal nu ofwel de producenten elimineren die de constante genereren die door de idempotente instructie overschreven wordt, of niet. In het eerste geval zijn er oorspronkelijke producenten geëlimineerd en is de nieuwe instructie dus geen idempotente instructie meer. In het andere geval is ze nog wel een idempotente instructie, omdat de andere producenten van de oorspronkelijke waarde toch niet konden geëlimineerd worden. Tijdens de volgende constantenpropagatie komen we dan vanzelf in geval 2 terecht.

2. We hebben geen eenvoudiger instructie kunnen genereren. De idempotente instructie wordt geëlimineerd.

Consumenten van constanten

In heel wat architecturen kunnen kleine constanten in de opcodes van instructies gecodeerd worden, zodat we de waarde niet in een register moeten stoppen. Men noemt deze operandi dan letterlijke operandi. Meestal kan dit slechts voor één van de bronoperandi van instructies.

Waar mogelijk vervangen we geconsumeerde kleine waarden in registers door letterlijke operandi. Indien we een kleine constante waarde terugvinden voor een operand waarvoor geen letterlijke operandi mogelijk zijn, proberen we de operandi om te wisselen voor commutatieve instructies. Indien de instructie niet commutatief is proberen we twee zaken:

- Indien mogelijk wisselen we de operandi om en inverteren we de instructie. Een voorbeeld is de instructie *compare-less-or-equal*:

```
C := 10 <= B
```

Deze instructie is equivalent met

$$C := B >= 10$$

Dan moet er natuurlijk wel een inverse instructie bestaan, zoals in het voorbeeld *compare-greater*.

- Indien er geen inverse instructie beschikbaar is, is het soms toch nog mogelijk om de operandi gewoon om te wisselen, zoals in volgende sequentie:

$$\begin{aligned} C &:= 10 - B \\ C &:= D - C \end{aligned}$$

In de eerste instructie kunnen de operandi omgewisseld worden als we alle consumenten van C ook kunnen aanpassen. De aangepaste sequentie wordt:

$$\begin{aligned} C &:= B - 10 \\ C &:= D + C \end{aligned}$$

Daarnaast proberen we het gebruik van basisadressen in lees- en schrijfoperaties te optimaliseren. Door de in de instructies gecodeerde afstanden aan te passen is het vaak mogelijk een ander basisregister te gebruiken dan het oorspronkelijke. Zo kunnen basisadressen hergebruikt worden voor verscheidene operaties en wordt vermeden dat er telkens nieuwe basisadressen moeten gebruikt en dus gegenereerd worden.

Merk op dat het daardoor (en door het optimaliseren van de producenten van constanten) mogelijk wordt dat er toch adresberekeningen over de grenzen van datablokken heen gaan, of dat instructies adressen uit een bepaald blok gebruiken om via letterlijke afstanden toegang tot een ander blok te krijgen. De beperking die de vertaler hierbij had valt nu dus weg. Voor de analyse van het datagebruik zoals die in sectie 3.3 wordt beschreven maakt dit echter geen verschil, aangezien hier geïntroduceerde berekeningen over blokgrenzen heen alleen voorkomen waar het constante en dus bekende adressen betreft.

3.2.7 Verwant werk

Het vergelijken van constantenpropagatie tijdens het linken met de bestaande literatuur over constantenpropagatie is niet eenvoudig, aangezien nagenoeg alle bestaande literatuur handelt over constantenpropagatie op een veel hoger niveau van programmavoorstelling.

De enige uitzondering ons bekend is [Muth99]. Hij beschrijft de basis van ons werk en de optimalisaties op constante waarden in het kader van snelheidsoptimalisatie na het linken. De voornaamste verschillen met de bespreking in dit proefschrift zijn dat

- Muth evalueert enkel de oplossingsmethode waarin slechts 1 register tegelijkertijd wordt gepropageerd;
- zijn implementatie voert een simpele constantenpropagatie uit maar geen conditionele constantenpropagatie;
- Muth bestudeert geen contextgevoelige of laat-samenvoegende technieken;
- hij heeft bij de optimalisatie van de productie van constanten enkel lokale informatie ter beschikking.

In OM [Sriv93, Sriv94b] wordt het gebruik van de globale wijzer geoptimaliseerd. Daar wordt deze optimalisatie als een apart geval beschouwd, aangezien men er helemaal geen constantenpropagatie uitvoert. Wij kaderen deze optimalisaties in de algemenere optimalisatie van constanten.

De overige ons bekende literatuur over interprocedurale constantenpropagatie focust op de detectie van constante parameters of constante globale veranderlijken aan de ingang van procedures. Daarbij wordt gebruik gemaakt van zogenaamde sprongfuncties [Call86, Grov93, Cari95, Autr94, Sagi95, Metz93]. Sprongfuncties worden in vele algoritmen voor constantenpropagatie gebruikt om de relaties te modelleren tussen de formele parameters van een procedure en de actuele parameters van procedure-oproepen binnen die procedure. Ze zijn een soort samenvattende vergelijkingen. Verschillende vormen van zulke functies werden reeds voorgesteld. Deze gaan van eenvoudige functies die enkel letterlijke (constante) parameters modelleren tot sprongfuncties die gebaseerd zijn op symbolische uitvoering van een procedure. Hoe complexer de sprongfuncties, hoe meer constanten er doorgaans gevonden worden. Sprongfuncties worden meestal gecombineerd met MOD en REF informatie [Call86] die modelleren hoe procedures omgaan met globale data.

Voor compactie tijdens het linken hebben parameters natuurlijk geen betekenis: één van de belangrijkste doelstellingen van compactie tijdens het linken is net het kunnen afstappen van conservatieve

en inefficiënte oproepconventies waar ze niet nodig blijken. Het is dus niet nodig/nuttig registers speciaal te behandelen volgens hun rol in de oproepstandaard (behalve waar het de nauwkeurigheid van de analyses kan verhogen natuurlijk). Omdat we (voorlopig) het geheugen zien als een zwarte doos, is er evenmin sprake van globale veranderlijken. De registers waarvan we de inhoud propageren kunnen wel als globale veranderlijken beschouwd worden. Deze hebben dan nog het belangrijke voordeel dat er geen aliassen tussen optreden. De door ons besproken technieken voor constantenpropagatie maken dus geen gebruik van sprongfuncties. Onze directe uitvoering van instructies en het intraproceduraal propageren van alle binnenkomende constanten tot aan de uitgang leunt echter heel dicht aan bij de symbolische uitvoering als sprongfunctie. Men kan onze verzameling van door procedures op de stapel bewaarde registers dan als MOD en REF informatie beschouwen.

Wegman en Zadeck [Wegm91] stellen voor hun intraprocedurale conditionele constantenpropagatie over een SUT-representatie uit te breiden naar een interprocedurale propagatie over een SUT-voorstelling van het hele programma. Daartoe moeten zij o.a. pseudo-operaties toevoegen die aliassen modelleren. Daarnaast voegen ze oproep- en terugkeerpijlen toe om procedurale SUT-voorstellingen tot een grote graaf samen te voegen. Ook moeten ze, afhankelijk van de methode waarmee parameters doorgegeven worden in de brontaal, pseudo-operaties toevoegen die actuele parameters aan procedure-oproepen binden met de formele parameters van de opgeroepen procedures.

Aangezien we geen rekening hoeven te houden met aliassen, heeft onze methode geen nood aan pseudo-operaties voor aliassen. En aangezien de registers dezelfde blijven over procedure-oproepen heen hoeven we ook geen operaties toe te voegen om actuele met formele parameters te binden. Dit zou wel het geval zijn op architecturen die werken met registervensters, zoals de SPARC architectuur [Weav94].

3.3 Analyse van datagebruik

In deze sectie zullen we het gebruik van statisch gealloceerde data in een programma analyseren [DS01]. We zullen trachten dode data en constante data te detecteren. Dit gebeurt aan de hand van een uitbreiding van de hiervoor besproken constantenpropagatie. Eerst zullen we

echter het belang van deze analyse trachten duidelijk te maken in een inleidende paragraaf.

3.3.1 Inleiding

Naast nutteloze code uit bv. bibliotheken wordt er natuurlijk ook nutteloze data meegelinkt in programma's. Bovendien, zoals in de sectie over constantenpropagatie al aangehaald werd, kan sommige data dood worden als we alle leesoperaties naar die data elimineren uit het programma.

Dode, statisch gealloceerde data kan in principe uit programma's verwijderd worden. Daardoor kan op zijn beurt weer onbereikbare code ontdekt worden als de geëlimineerde data wijzers naar code bevat zoals adrestabellen, virtuele methodetabellen, enz.

Men kan stellen dat het verwijderen van onbereikbare code en dode data dus hand in hand gaan.

Voorbeeld 3.9 Beschouw de C code in figuur 3.13. Het programma bestaat uit twee bronbestanden, main.c en pointer.c. Het laatste bestand bevat enkel onbereikbare code: een procedure die haar eigen adres print maar die nergens opgeroepen wordt. Afhankelijk van welke objectbestanden op de commandolijn meegegeven worden aan de vertaler/linker, worden verschillende bestanden gegenereerd. De grootte van deze bestanden is weergegeven in tabel 3.2. Het meelinken van het nutteloze pointer.o resulteert in een initieel programma dat meer dan 2 maal zo groot is. Het objectbestand pointer.o zelf was nochtans slechts 1 KB groot. De enige reden voor deze toename is het meelinken van code uit de C bibliotheek. De oproep naar `printf()` in de dode procedure resulteert immers in het meelinken van code om hexadecimale getallen af te drukken. Naar deze functionaliteit wordt niet verwezen vanuit het hoofdprogramma.

Als we enkel codecompactie toepassen worden de programma's aanzienlijk kleiner. De ene versie is echter nog steeds veel groter dan de andere. De reden is dat het adres van `a()` in de datasecties ligt opgeslagen om af te printen. Indien we niet achterhalen dat dit de enige manier is waarop het adres zal gebruikt worden, moeten we ervan uitgaan dat het adres zal gebruikt worden om `a()` op te roepen. We kunnen de nutteloos meegelinkte onbereikbare code dus niet helemaal verwijderen.

main.c

```
#include <stdio.h>

main() {
   printf("hello world");
}
```

pointer.c

```
#include <stdio.h>

void a(void){
   printf("%lx",(void*)&a);
}
```

Figuur 3.13: C code voor dode-data-eliminatie.

Objectbestanden	main.o pointer.o		main.o	
	data	code	data	code
voor compactie	70848	182432	46704	58432
na codecompactie	68032	102016	45680	24640
na code- en datacompactie	43408	20864	43408	20864

Tabel 3.2: Grootte (in bytes) van de code- en datasecties voor en na code-compactie en na gecombineerde code- en datacompactie. De gegevens zijn weergegeven voor beide versies van het programma: links zijn beide object-bestanden meegelinkt, rechts enkel main.o.

Als we de onbereikbare code en data gelijktijdig detecteren, kunnen we dit wel. Beide versies zijn dan even groot, wat er op wijst dat onze prototype compactor erin geslaagd is alle nutteloos meegelinkte functionaliteit (door dat ene objectbestand pointer.o) te elimineren.

Merk overigens ook op dat de gecombineerde aanpak de kleine versie van het programma nog ietsje kleiner maakt. Dat de data-secties ook kleiner geworden zijn wanneer enkel code-compactie toegepast wordt is te wijten aan het hergebruik van data. Dit wordt besproken in sectie 4.7 en heeft niets met de detectie of het verwijderen van dode data te maken. □

Dit voorbeeld is natuurlijk geen realistisch voorbeeld. Het toont ons inziens wel aan hoe een heel klein stukje nodeloos meegelinkte functionaliteit op zijn beurt een heleboel onnodige code en data kan meelinken. Het geeft ook aan dat een gecombineerde aanpak nodig is.

De doelstellingen van de in deze sectie besproken algoritmen zijn dan ook om tegelijkertijd

- onnodige datatoegangen te elimineren;
- dode data te detecteren en eventueel te verwijderen;
- een meer nauwkeurige onbereikbare-code-eliminatie uit te kunnen voeren.

We zullen dit doen door de bestaande constantenpropagatie aan te passen. Om vlot over de nodige uitbreidingen te kunnen redeneren is het nuttig eerst nog eens terug te grijpen naar het verschil tussen simpele en conditionele constantenpropagatie.

Eén manier om naar de uitbreiding van simpele naar conditionele constantenpropagatie te kijken is vanuit het oogpunt van de slechtstegeval-veronderstelling (SGV) [dEUS94] die we maken voor en tijdens de uitvoering van het algoritme. Voor elke analyse worden zulke veronderstellingen gemaakt voor gevallen waarin we het exacte gedrag van een programma niet kennen. Deze veronderstellingen dragen er dan toe bij dat de analyses toch nog correcte, conservatieve resultaten opleveren.

Bij simpele constantenpropagatie wordt er een globale a priori veronderstelling gemaakt:

SGV 0 : Beide paden volgend op alle conditionele sprongen zijn realiseerbaar.

Bij conditionele constantenpropagatie wordt deze veronderstelling niet op voorhand gemaakt, maar wordt het maken van SGVen naar later opgeschoven. Door op het goede moment tijdens de analyse SGVen te maken kan men zich beperken tot veronderstellingen die een meer lokaal karakter hebben. Meer concreet zal men dan voor elke conditionele sprong apart, op het juiste moment voor die bepaalde sprong, een SGV maken:

SGV 0′ : Beide paden volgend op een conditionele sprong zijn realiseerbaar indien de conditie van de sprong naar $\perp$ evalueert.

Deze benadering van het in de tijd verplaatsen en verfijnen van SGVen zullen we gebruiken in de rest van deze sectie om de algoritmen te beschrijven waarmee we de bovenvermelde doelstellingen willen bereiken. Uitgaand van een aantal SGVen die gemaakt worden voor de eerder besproken constantenpropagatie zullen we komen tot een constantenpropagator die in belangrijke mate die doelstellingen verwezenlijkt.

3.3.2 Veronderstellingen voor constantenpropagatie

Een van de SGVen die we in de bespreking van de constantenpropagatie gemaakt hebben is dat alle lees- en schrijfoperaties onderling aliassen kunnen zijn:

SGV 1 : Alle lees- en schrijfoperaties kunnen onderling aliassen zijn.

Over de statisch gealloceerde data werden impliciet volgende veronderstellingen gemaakt:

SGV 2 : Code-adressen die opgeslagen liggen in levende, statisch gealloceerde geheugenlocaties resulteren in bereikbare code op die adressen.

SGV 3 : Alle geheugenlocaties met statisch gealloceerde data zijn levend.

SGV 4 : Alle statisch gealloceerde data in overschrijfbare datasecties (data- en nulsegment) wordt overschreven gedurende de uitvoering van het programma.

SGV 2 is een gevolg van het feit dat we, eens we weten dat code-adressen eventueel kunnen opgeladen worden, niet weten waarvoor die code-adressen zullen gebruikt worden. We moeten er dus van uitgaan dat ze onder meer gebruikt worden om naar te springen via indirecte sprongen of oproepen.

Omdat SGV 3 stelt dat alle statisch gealloceerde data levend is, waaronder dus ook alle erin opgeslagen code-adressen, resulteert dit in een groot aantal programmapunten die initieel als bereikbaar gezien worden. Niet alleen zijn deze punten per definitie bereikbaar (m.a.w. de oproepershelleknoop is bereikbaar), we weten niet van waar ernaar gesprongen zal worden. We moeten dus niet-constanten naar deze punten propageren (d.w.z. dat $\perp$ uit de oproepershelleknoop gepropageerd wordt). Het is precies SGV 3 die heel conservatief is. Hoe we ze kunnen verfijnen is precies het onderwerp van de volgende paragrafen.

SGV 4 resulteert in het feit dat we geen data uit beschrijfbare datasecties in het programma zullen propageren als we leesoperaties evalueren die van constante adressen in die secties lezen. Dat doen we wel voor instructies die laden vanuit het tekstsegment.

3.3.3 Globaal-uniforme constantenpropagatie

In deze sectie wordt de constantenpropagatie uitgebreid door de veronderstellingen te verfijnen. SGVen 0', 1 en 2 blijven echter onveranderd van kracht, aangezien we nog geen andere, minder conservatieve veronderstellingen gevonden hebben om ze te vervangen. SGV 1 ligt aan de oorsprong van de naam die we aan de aangepaste constantenpropagatie gegeven hebben: de globaal-uniforme constantenpropagatie. De statisch gealloceerde data in het programma wordt ofwel constant verondersteld gedurende de hele uitvoering van het programma, of ze wordt als niet-constant (en dus onbekend) beschouwd. Dit lijkt sterk op de uniforme divisie van veranderlijken in de theorie over partiële evaluatie [Jone93].

Het verfijnen van SGV 3 en SGV 4 gebeurt in twee stappen: eerst worden ze opgesplitst in nauwkeuriger veronderstelingen die tijdens de propagatie zelf gemaakt moeten worden. Daarna zullen we enkele van die veronderstellingen terug naar voor schuiven in de tijd.

Stap 1: verfijnen van SGV 3 en 4

SGV 3 wordt vervangen door drie minder verregaande veronderstellingen:

SGV 3.1 : Een geheugenlocatie wordt levend als er een instructie leest van op die locatie.

SGV 3.2 : Alle geheugenlocaties worden levend als er een leesoperatie is die leest van op een onbekend adres.

SGV 3.3 : Een data-adres dat statisch gealloceerd is op een locatie die niet-constante levende data bevat of op een locatie die om andere redenen dan SGV 3.1 levend wordt, maakt het hele datablok dat dat adres bevat levend, en niet-constant indien het in een overschrijfbare sectie ligt.

SGV 3.1 spreekt voor zich. Als er leesoperaties zijn in het programma die we niet kunnen evalueren tijdens het programma (omdat het basisadres geen constante is), gaan we er met SGV 3.2 conservatief van uit dat deze leesoperatie van op alle statisch gealloceerde locaties kan lezen.

SGV 3.3 is vergelijkbaar met SGV 2. Net zoals code-adressen die opgeslagen liggen in levende locaties resulteren in bereikbare code omdat we ervan uitgaan dat ze gebruikt worden voor indirecte controletransfers, zullen data-adressen waarvan we het gebruik niet kunnen analyseren aanleiding geven tot andere levende en eventueel niet-constante data. We gaan ervan uit dat die data-adressen voor alles zullen gebruikt worden waarvoor ze kunnen gebruikt worden. Gelukkig is dit beperkt tot het eigen datablok, aangezien er geen berekeningen over een grens tussen twee blokken kunnen uitgevoerd worden, zoals in sectie 2.3.4 werd uiteengezet. Als er een locatie levend wordt omwille van SGV 3.1, zullen de daarvoor verantwoordelijke leesoperaties geëvalueerd worden en zal de constante data op die locatie in het programma gepropageerd worden en het gebruik ervan dus verder geanalyseerd, waarbij we eventueel verdere veronderstellingen zullen maken indien nodig. Dit geval sluiten we dus uit voor SGV 3.3.

SGV 4 wordt op een analoge manier als SGV 3 verfijnd:

SGV 4.1 : Een geheugenlocatie bevat niet-constante data als er een instructie schrijft naar die locatie.

SGV 4.2 : Alle geheugenlocaties in overschrijfbare secties bevatten niet-constante data als er instructies schrijven naar onbekende adressen.

SGV 4.3 : Als er een bekend (constant) data-adres wordt weggeschreven door een instructie naar een al dan niet bekende locatie, wordt het hele blok rond dat data-adres levend, en indien het blok in een overschrijfbare sectie ligt, bevat het enkel niet-constante data.

SGV 4.4 : Wordt onbekende (niet-constante) data weggeschreven naar een al dan niet bekende locatie, dan worden alle locaties levend en alle locaties in overschrijfbare secties bevatten niet-constante data.

De eerste drie verfijningen zijn zeer analoog aan de verfijningen van SGV 3. De redenering achter 4.4 is eenvoudig: de onbekende waarde die weggeschreven wordt kan potentieel elk data-adres zijn. Van elk van die adressen zullen we niet analyseren hoe ze gebruikt worden, aangezien we het gebruik ervan niet analyseren waar ze mogelijk ingeladen worden. Voor elk van deze adressen moet dus een veronderstelling in de zin van SGV 4.3 gemaakt worden.

Waren de oorspronkelijke SGVen 3 en 4 a priori veronderstellingen, dan is dit niet meer het geval voor hun vervangers: die moeten slechts gemaakt worden indien zich gedurende de propagatie bepaalde zaken voordoen. Daardoor kan het voorkomen dat op een bepaald moment tijdens de dekpuntberekeningen de statisch gealloceerde data op een bepaalde locatie als constant beschouwd wordt en ergens bij een leesoperatie in het programma gepropageerd wordt, terwijl later blijkt dat deze data toch niet constant is. De leesoperaties die de data op die locatie in het programma doen propageren, moeten dan opnieuw geëvalueerd worden, waarbij nu $\perp$ in het programma gepropageerd wordt. Om op een efficiënte manier deze instructies op te kunnen sporen houden we per locatie waarvan de data constant verondersteld wordt een lijst bij van instructies die de data op die locatie in het programma gepropageerd hebben. Telkens we een leesinstructie kunnen evalueren voegen we die instructie toe aan de lijst instructies horende bij de locatie waarvan gelezen wordt. Als de data op die locatie later niet-constant wordt, volstaat het de instructies uit de lijst te halen en te markeren voor herevaluatie. Merk op dat elke instructie maar in één lijst kan zitten, aangezien een instructie die we kunnen evalueren slechts van één adres kan laden.[5]

Het belangrijkste resultaat van deze verfijningen is dat bij de initialisatie van de constantenpropagatie alle statisch gealloceerde data als dood beschouwd wordt. Dit betekent ook dat we enkel het ingangspunt van het programma als bereikbaar moeten beschouwen bij de initialisatie. Tijdens de propagatie zullen daar opvolgers van de oproepershelleknoop bijkomen naargelang er code-adressen in de datasecties levend worden. Dan zullen we nog steeds $\perp$ naar deze opvolgers propageren.

Het moge echter duidelijk zijn dat deze verfijning absoluut niet volstaan om in niet-triviale programma's ook maar enig verschil te maken. SGVen 3.2, 4.2 en 4.4 zullen altijd wel ergens moeten gemaakt worden tijdens de dekpuntberekeningen voor niet-triviale programma's. Het is immers ijdele hoop in programma's alle lees- en schrijfoperaties volledig te kunnen evalueren.

Gelukkig kunnen we de heel conservatieve SGVen die gepaard gaan met het consumeren van onbekende data (adressen) vermijden.

[5]Voor een contextgevoelige constantenpropagatie kan elke leesinstructie per binnenkomende interprocedurale pijl van een ander adres lezen, en kan de instructie dus in verscheidene lijsten voorkomen. Dan nog is het aantal lijsten waarin een instructie kan voorkomen sterk beperkt.

In plaats van veronderstellingen te maken als de onbekende data geconsumeerd worden, maken we ze daartoe beter op het moment dat ze geproduceerd worden.

Stap 2: van consumenten naar producenten

Op het punt waar tijdens de dekpuntberekeningen een niet-constante waarde wordt geproduceerd uit één of meer constante adressen, dus waar we m.a.w. het spoor bijster raken van constante adressen, kunnen we de veronderstellingen over het gebruik van die adressen beperken tot de datablokken van de betrokken adressen[6]. Het aantal gevallen waarin adressen verloren gaan tijdens de berekeningen is bovendien beperkt:

- als een levend (volgens de levensduuranalyse van sectie 3.1) constant adres samengevoegd moet worden met een niet-constante of een andere waarde;
- als een instructie een niet-constant adres berekent uit een constant adres, bv. door bij een constant basisadres een onbekende afstand op te tellen. Het resultaat wordt dan verder gepropageerd als niet-constante.
- als bij een indirecte sprong of procedure-oproep de doelen van de controletransfer niet bekend zijn, en er een levend adres naar de helleknopen wordt gepropageerd.

Als de correcte SGVen worden gemaakt voor deze gevallen, moeten we geen verdere veronderstellingen meer maken wanneer onbekende data (adressen) geconsumeerd worden door lees- of schrijfinstructies. SGVen 3.2, 4.2 en 4.4 kunnen dan ook vervangen worden door één SGV:

SGV 5 : Wanneer een constant data-adres niet verder gepropageerd wordt als een constante (m.a.w. een niet-constante wordt verder gepropageerd), worden alle locaties in het datablok van dat adres levend, en, indien het blok in een overschrijfbare sectie ligt, niet-constant.

Hiermee zijn eindelijk alle SGVen omgezet in veronderstellingen die beperkt zijn tot slechts één datablok.

[6] Dit is eigenlijk een vorm van abstracte interpretatie [Cous77]: het adres zelf wordt geabstraheerd naar het blok van het adres.

Discussie

Als we de aangepaste constantenpropagatie zoals ze er nu voor staat evalueren, dan merken we dat ze eigenlijk niet goed presteert.

De reden is dat SGV 5 heel conservatief is wat betreft het gebruik van de verloren adressen. We veronderstellen dat deze adressen gebruikt zullen worden voor lees- en schrijfoperaties na berekeningen op de adressen, m.a.w. dat ermee in heel hun blok zal gelezen en geschreven worden. Vaak worden ze enkel voor leesoperaties gebruikt, in andere gevallen worden er geen berekeningen meer op de adressen uitgevoerd en worden ze dus enkel gebruikt om van/naar een beperkt aantal vaste locaties in hun blok te lezen of te schrijven.

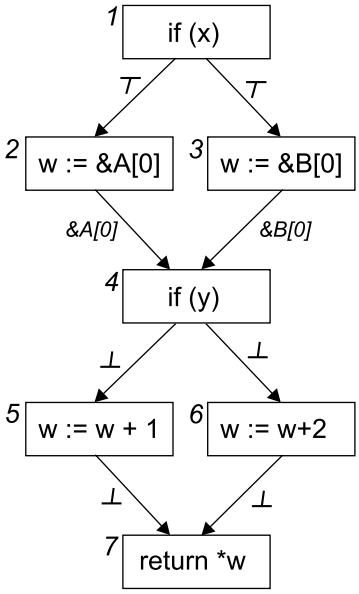
Voorbeeld 3.10 Beschouw de graaf uit figuur 3.14. Veronderstel daarbij drie rijen (A, B en C) die elk een apart datablok vormen in overschrijfbare secties. De waarden van x en y zijn onbekend. De pijlen van de graaf zijn geannoteerd met de met w geassocieerde dekpuntwaarden die erover gepropageerd worden.

Als we met de originele SGVen 0', 1, 2, 3 en 4 werken, is het eindresultaat dat A, B en C levend zijn en niet-constante data bevatten. Dit was immers de beginveronderstelling.

Als we met de verfijningen van stap 1 werken, dan is het eindresultaat dat A, B en C levend zijn, maar constante data bevatten. De enige SGV die we moeten toepassen is immers 3.2 op het moment dat $\perp$ in basisblok 7 gepropageerd wordt: de dereferentieoperatie op w is een leesoperatie vanop een onbekend adres.

We hebben dus nog niet ontdekt dat rij C dood is voor het stukje code. Als we echter met de SGV na stap 2 werken, moeten we SVG 3.2 niet meer maken. Daarentegen moeten we wel aan de ingang van blok 4 SGV 5 aannemen: de adressen &A[0] en &B[0] botsen er op elkaar en worden dus niet meer afzonderlijk verder gepropageerd. Het eindresultaat is nu dat enkel A en B als levend beschouwd worden, maar jammer genoeg ook als niet-constant. □

Algemeen kunnen we stellen dat de globaal-uniforme constantenpropagator zoals we die tot nu toe beschreven hebben vergelijkbaar is met een monovariante partiële evaluatie [Jone93] (er wordt slechts één geassocieerde waarde bijgehouden per veranderlijke per programmapunt). Het is algemeen bekend dat een polyvariante partiële evaluatie



Figuur 3.14: Globaal-uniforme constantenpropagatie.

(waarin verscheidene waarden kunnen bijgehouden worden) accuratere resultaten oplevert. Een polyvariante analyse is jammer genoeg ook complexer en moeilijker implementaarbaar omwille van efficiëntie- en terminatieproblemen.

Gelukkig zou een polyvariante evaluatie in ons geval niet het hele programma moeten evalueren, we zijn immers enkel in het gebruik van adressen geïnteresseerd. En wat het terminatieprobleem betreft: aangezien adressen slechts kunnen gebruikt worden binnen hun datablok, stelt het terminatieprobleem zich hier ook in veel mindere mate dan bij een algemene partiële evaluatie. Een eenvoudig terminatiecriterium voor wijzigende data-adressen is immers dat ze niet buiten hun blok mogen gaan.

3.3.4 Partiële evaluatie van adresberekeningen

Als we de vergelijkingen van de globaal-uniforme constantenpropagatie eens opnieuw bekijken, kunnen we met betrekking tot het levend worden van data twee vergelijkingen terugvinden:

SGV 3.2 : Als er een leesoperatie leest op een onbekend adres, worden

alle geheugenlocaties levend.

deel van SGV 5 : Wanneer een constant data-adres niet verder gepropageerd wordt als een constante, worden alle locaties in het datablok van dat adres levend.

Deze twee vergelijkingen zouden we graag combineren, zodat we iets krijgen in de zin van:

SGV : Als er een leesoperatie leest op een onbekend adres, worden alle locaties in het corresponderende datablok levend.

Dit gaat natuurlijk niet zomaar. Het probleem zit in de discrepantie tussen de woorden "onbekend" en "corresponderende".

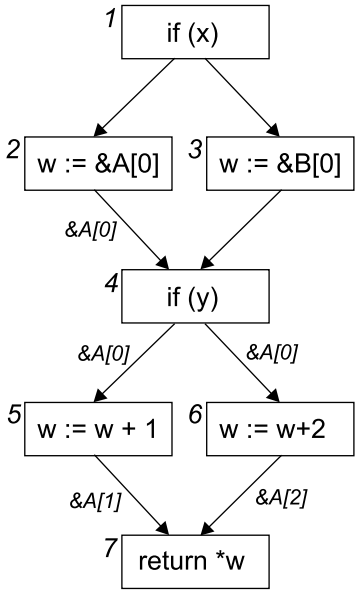
Er bestaat evenwel een correcte, vrij gelijkaardige SGV:

SGV : Als er een leesoperatie leest op een onbekend adres dat afgeleid is van een bekend adres, worden alle locaties in het corresponderende datablok levend.

Deze SGV is correct. Immers, in sectie 2.3.4 hebben we gesteld dat om toegang te kunnen hebben tot de data in een blok, er ergens een wijzer naar dat blok expliciet in het programma moet voorkomen. Alle onbekende adressen worden m.a.w. berekend uit bekende adressen en blijven binnen hetzelfde datablok. Als men deze SGV kan evalueren en eventueel toepassen voor alle geproduceerde bekende adressen, zijn meteen alle adresgebruiken geëvalueerd. Niet-constante adressen kunnen immers enkel berekend worden met berekeningen die als startpunt een constant adres hebben.

Is het makkelijk om deze SGV neer te pennen, ze in de praktijk brengen in een constantenpropagator is verre van triviaal. Bijhouden van welke adressen een onbekende waarde kan afgeleid zijn, komt immers neer op een multivariante propagatie. Alhoewel dit in theorie zoals hierboven geschetst geen probleem hoeft te zijn, is een implementatie ervan toch bijzonder complex.

Daarom hebben we geopteerd om een multivariante analyse te benaderen door het uitvoeren van verschillende monovariante analyses: voor elk geproduceerd adres wordt een aparte constantenpropagatie uitgevoerd, die nagenoeg identiek is met de globaal-uniforme propagatie zoals hierboven beschreven. Het enige verschil is dat we vermijden dat tijdens zo'n aparte propagatie andere geproduceerde constante adressen meegepropageerd worden.



Figuur 3.15: Partiële evaluatie van adressen.

Voorbeeld 3.11 Beschouw dezelfde graaf als in het vorige voorbeeld, die we opnieuw afgebeeld hebben in figuur 3.15. Veronderstel eventjes dat we de instructie in blok 2 als het begin van het programma beschouwen en er een constantenpropagatie voor uitvoeren. In blok 7 zal w opnieuw geassocieerd zijn met $\perp$. Als we er echter in geslaagd zijn om geen andere adressen mee te propageren, kunnen we ervan uitgaan dat die $\perp$ enkel adressen berekend vanaf &A[0] kan voorstellen, en we moeten dus enkel het datablok met rij A erin levend maken. Een analoge propagatie waarin we de instructie in basisblok 3 als begin van het programma beschouwen resulteert in het feit dat het het blok met rij B levend wordt. Geen van beide datablokken wordt echter niet-constant. Door in aparte propagaties de waarde $\perp$ door te propageren en opnieuw pas veronderstellingen te maken op het moment dat die $\perp$ geconsumeerd wordt, zijn we erin geslaagd het gewenste resultaat te bereiken, althans voor dit voorbeeld. □

Aangezien die aparte propagaties nog steeds monovariant zijn, zullen er nog steeds veel onbekende waarden als $\perp$ gepropageerd worden. Deze onbekende waarden kunnen berekend zijn uit de waarde die geproduceerd werd aan het beginpunt van zo'n aparte propagatie (het

startadres), of ze kunnen van heel andere waarden komen (maar geen andere adressen). In het laatste geval moeten we geen SGV maken op het moment dat die $\perp$ opgeslagen wordt door een instructie. Om het verschil te kunnen maken tussen beide gevallen, krijgen de gepropageerde waarden een extra annotatie mee, die aanduidt of de (bekende of onbekende) waarde berekend is uit het startadres.

De partiële evaluatie van adresberekeningen werkt dan als volgt:

1. Alle data wordt initieel als dood en constant beschouwd. Markeer alle instructies die direct bereikbaar zijn vanuit het beginpunt van het programma.

2. Voor elke aldus gemarkeerde instructie die een constant adres produceert volgens de globaal-uniforme constantenpropagatie wordt een partiële evaluatie uitgevoerd, beginnend bij die instructie.

3. Gedurende elk van die partiële evaluaties zullen een aantal SGVen moeten gemaakt worden. Hoe deze verschillen van de veronderstellingen tijdens de globaal-uniforme constantenpropagatie wordt verder besproken.

4. Als er gedurende zo'n partiële evaluatie instructies geëvalueerd worden die constante adressen produceren (volgens de globaal-uniforme constantenpropagatie), propageren we die adressen niet, maar propageren we $\top$ in hun plaats. Dit vermijdt dat de samenvoegingsregel nodeloos moet toegepast worden. Voor zulke instructies worden de nodige veronderstellingen wel gemaakt bij de aparte partiële evaluatie die er toch voor uitgevoerd wordt.

5. Gedurende een partiële evaluatie kan het voorkomen dat er leesinstructies geëvalueerd worden waarvan we nu het basisadres kennen, terwijl de globaal-uniforme constantenpropagator het adres niet kende. Dit kan voorkomen omdat we bv. niet met alle realiseerbare paden die naar een leesinstructie leiden rekening hoeven te houden tijdens een bepaalde partiële evaluatie. In zulke gevallen wordt de instructie enkel geëvalueerd indien we de a priori kennis hebben dat de locatie waarvan gelezen wordt constante data bevat. Anders propageren we $\perp$ in het programma. De reden is dat we omwille van tijds- en geheugenefficiëntie slechts één partiële evaluatie per geproduceerd constant

adres kunnen uitvoeren. Daardoor kunnen we uiteraard niet veronderstellen dat data constant is tot het tegendeel aangetoond wordt.

6. De analyse van het gebruik van adressen hield in de globaal-uniforme constantenpropagatie op waar de adressen verloren gingen, d.w.z. waar $\perp$ geproduceerd werd. Tijdens een partiële evaluatie duurt dit echter zolang als er geannoteerde adressen gepropageerd worden: zolang kunnen er immers nieuwe consumenten ontdekt worden. Dit is de belangrijkste reden waarom we deze analyse een partiële evaluatie noemen i.p.v. een constantenpropagatie. De annotaties worden meegepropageerd zoals de annotaties statisch/dynamisch [Jone93] tijdens klassieke partiële evaluaties, en terminatie hangt af van de gepropageerde waarden en de annotaties. Overigens vertoont onze analyse ook veel verwantschap met abstracte interpretatie [Cous77].

7. Gedurende één partiële evaluatie worden een aantal geheugenlocaties op levend gezet. Code-adressen die op die locaties statisch gealloceerd zijn geven zoals bij de globaal-uniforme constantenpropagatie aanleiding tot nieuwe bereikbare instructies. Deze worden toegevoegd aan de initiële verzameling bereikbare punten (uit item 1), en ook voor de aldus toegevoegde instructies die een constant adres produceren zal een aparte partiële evaluatie uitgevoerd worden. Dit geeft uiteraard aanleiding tot een iteratief proces, waarin de verzameling bereikbare punten steeds groter wordt.

Hoe moeten de SGVen aangepast worden? Het is duidelijk dat alle veronderstellingen die gemaakt moeten worden tijdens een aparte partiële evaluatie enkel het datablok kunnen beïnvloeden dat correspondeert met het startadres van die aparte propagatie. SGV 3.2, 4.2 en 4.4 moeten dus niet meer vervangen worden door SGV 5. We vervangen ze daarentegen door:

SGV 3.2' : Alle geheugenlocaties in het datablok van het startadres worden levend als er een leesoperatie is die leest vanop een onbekend, van het startadres afgeleid adres.

SGV 4.2' : Alle geheugenlocaties in het datablok van het startadres bevatten niet-constante data als er instructies schrijven naar on-

bekende adressen die van het startadres zijn afgeleid. Dit kan enkel als het startadres in een overschrijfbare sectie ligt.

SGV 4.4' : Wordt onbekende (niet-constante), maar van het startadres afgeleide data weggeschreven naar een al dan niet bekende locatie, dan worden alle locaties in het datablok van het startadres levend en bevatten ze niet-constante data indien ze in een overschrijfbare sectie liggen.

Die gevallen waarin onbekende, afgeleide adressen een lees- of schrijfinstructie bereiken zijn daarmee afgehandeld. De afhandeling in het geval van bekende (constante) adressen gebeurt nog steeds door SGV 3.1 en SGV 4.1 toe te passen. Doordat dit nu enkel gebeurt als de onbekende waarden geconsumeerd worden i.p.v. als ze geproduceerd worden, worden de te conservatieve SGVen van de globaal-uniforme constantenpropagatie vermeden. Merk op dat het voorkomen van blokoverschrijdende berekeningen na de optimalisaties na constantenpropagatie geen probleem stelt, aangezien het in zulke gevallen precies om bekende adressen gaat, die steeds onder SGV 3.1 en SGV 4.1 vallen.

Wat overblijft zijn de gevallen waarin afgeleide, al dan niet constante adressen niet meer verder gepropageerd worden om bepaalde redenen. Als we het lijstje uit paragraaf 3.3.3 bekijken, zien we dat enkel het laatste item niet door SGV 3.2', 4.2' of 4.4' wordt behandeld. We hebben dus nood aan een extra SGV voor dat geval:

SGV 5' : Als een afgeleid adres (constant of niet) gedurende de partiële evaluatie niet verder kan gevolgd worden omdat we de doelen van een indirecte controletransfer niet kennen, wordt alle data in het blok van het startadres levend en, indien de data in een overschrijfbare sectie ligt, niet-constant.

Merk op hoe dit nog steeds sterk op SGV 5 lijkt.

3.3.5 Combinatie van de twee analyses

Beide tot hier toe besproken analyses (globaal-uniforme constantenpropagatie en partiële evaluatie van adresberekeningen) resulteren in conservatieve benaderingen van de verzamelingen van dode en constante statisch gealloceerde data.

De globaal-uniforme constantenpropagatie resulteert in een te conservatieve schatting omdat ze sterk belemmerd wordt door de al te

conservatieve veronderstellingen als niet-constante adressen geproduceerd worden. De partiële evaluatie hangt dan weer af van de prestaties van de constantenpropagatie: hoe meer instructies de constantenpropagatie vindt die constante adressen produceren, hoe meer aparte, en dus minder conservatieve, evaluaties er uitgevoerd worden, en hoe beter de resultaten van de volledige partiële evaluatie.

Beide analyses leveren dus correcte, zij het verschillende resultaten op. Men kan die resultaten samenvoegen: als één van de analyses aangeeft dat een geheugenlocatie dood is, is ze immers zeker dood. Hetzelfde geldt voor het al dan niet constante karakter van de data op een locatie.

Omdat er ook andere redenen zijn waarom we in de praktijk meer dan een constantenpropagatie wensen uit te voeren, stellen we daarom voor beide analyses iteratief uit te voeren, waarbij alle te maken veronderstellingen aangepast worden door toevoeging van het zinnetje "als dit niet in tegenstrijd is met de a priori kennis". Dit zinnetje is eigenlijk de uitbreiding van wat al in de SGVen stond met betrekking tot de overschrijfbare secties. Enkel data in die secties kan niet-constant zijn. Dit is één vorm van a priori kennis, maar er kunnen er ook andere zijn, meer bepaald door de toevoeging van andere, nieuwe analyses, of door in ons geval de beschreven analyses iteratief uit te voeren.

Na elke analyse wordt de "a priori kennis" uitgebreid met de nieuwe kennis: de verzamelingen van a priori dode en constante data groeien aldus gedurende het iteratief uitvoeren van de analyses.

3.3.6 Gebruik van de resultaten

De resultaten van de besproken analyses worden voor twee groepen optimalisaties gebruikt: enerzijds voor de verfijning van de ICVG en anderzijds voor het verwijderen van dode data en instructies die naar die locaties schrijven.

Verfijning van de ICVG Hiervoor overlopen we zoals bij de initiële opbouw van de graaf de relocatiegegevens. We berperken ons echter tot relocatiegegevens over levende data. Dit levert ons alle levende code-adressen op die opgeslagen liggen in de data-secties. De instructies en basisblokken op deze adressen worden gemarkeerd. Daarna worden de pijlen vanuit de helleknopen verwijderd naar niet gemarkeerde basisblokken, waarbij rekening wordt gehouden met de criteria

die in sectie 3.1.4 besproken werden.

Verwijderen van de dode data In tegenstelling tot wat men op het eerste zicht kan denken, volstaat het jammer genoeg niet dat alle data in een datablok dood is opdat men het blok zou mogen verwijderen.

Een extra voorwaarde is nog dat er geen instructies in het programma overblijven die data wegschrijven naar het datablok. Indien dit wel het geval is, en het blok wordt toch verwijderd, zal zo'n schrijfinstructie dan immers ofwel in een ander blok schrijven ofwel ergens schrijven waar het niet toegestaan is, wat tot een foutief programma zal leiden. Beide gevallen zijn onaanvaardbare wijzigingen in het gedrag van het programma.

Natuurlijk kunnen we schrijfinstructies die schrijven naar dode locaties verwijderen uit het programma. Het probleem ligt echter bij de instructies die naar een dood blok (kunnen) schrijven, maar die we niet kunnen detecteren of niet kunnen verwijderen omdat ze ook naar andere, levende blokken kunnen schrijven.

Om te zien of we dode blokken al dan niet kunnen verwijderen moeten we dus kunnen inschatten of er schrijfoperaties naar die blokken bestaan die we al dan niet kunnen verwijderen. Gelukkig is dit eenvoudig: het volstaat een onderscheid te maken tussen het niet-constant zetten van één locatie en het niet-constant zetten van een heel datablok: alleen in het laatste geval bestaat immers de kans dat we niet weten welke instructie de oorzaak is van het niet-constant zetten. Merk op dat we een analoog onderscheid ook al moeten maken i.v.m. het levend zijn van data: SGV 3.3 vereist dit immers.

Eens dit onderscheid gemaakt is gaan we als volgt te werk:

1. Als er in een blok geen enkele data levend is, mogen alle schrijfoperaties die enkel naar dat blok schrijven geëlimineerd worden uit het programma.

2. Zijn er bovendien geen onbekende schrijfoperaties, dan kan het blok verwijderd worden uit het programma.

3. Is een blok niet 'en bloc' levend gezet, dan kan het zijn dat er dode locaties in een blok zijn. In dat geval verwijderen we de schrijfoperaties die enkel naar dode locaties schrijven. Bovendien zetten we de data op dode locaties op nul. Code- of data-adressen op die locaties worden met andere woorden gewist, zodat ze geen

negatief effect meer kunnen hebben op verdere analyses of verfijningen.

4. In het nulsegment worden schrijfoperaties die nul wegschrijven naar locaties waar geen andere data naar kan geschreven worden eveneens geëlimineerd.

Het effectief verwijderen van dode data uit het programma gebeurt nu nog steeds per blok. Uit blokken waarvan we alle schrijf- en leesinstructies kennen kunnen we in principe ook aparte data-elementen verwijderen. Dit hebben we nog niet geprobeerd.

Het verwijderen van datablokken uit de datasecties brengt natuurlijk met zich mee dat de data-adressen waarmee lees- en schrijfinstructies werken, veranderen. Zoals we berekeningen op code-adressen niet kunnen toestaan omdat de adressen wijzingen tijdens de compactie, stelt ook het veranderen van de data-adressen ons voor problemen. Uiteraard kunnen we geen programma's uitsluiten die berekeningen op data-adressen uitvoeren, aangezien elk niet-triviaal programma dit doet.

Een mogelijke oplossing hiervoor is het annoteren van alle data die in het programma gepropageerd wordt als adres of als niet-adres. Telkens een data-adres dan verandert, moet ook de code aangepast worden die het adres gebruikt of ernaar verwijst. Dit vraagt een aanpassing aan heel wat transformaties en analyses en is moeilijk implementeerbaar. Bovendien kan men er tijdens de constantenpropagatie en de daarmee gepaard gaande optimalisaties dan niet meer van uitgaan dat data-adressen constant zijn.

We hebben dit in ons prototype als volgt vermeden: na een eerste compactie wordt een lijst uitgeschreven van datablokken die elimineerbaar zijn. Daarna voeren we een tweede, identieke compactie uit. Het enige verschil is dat we de elimineerbare datablokken voor we aan de tweede compactie beginnen samengebracht hebben en geheralloceerd hebben in een apart stuk geheugen. We halen ze m.a.w. van tussen de levende data, die achter elkaar kan geplaatst worden. Het volstaat daarbij om bij die herallocatie de relocatiegegevens aan te passen om het gedrag van het programma niet te veranderen.

Eens die verandering doorgevoerd, voeren we gewoon dezelfde compactie uit. Op het eind van die compactie zijn normaal gezien alle blokken die apart gezet zijn opnieuw dood en nu volstaat het om die niet mee uit te schrijven in het finaal gecompacteerde programma. Enig

nadeel van deze methode is dat de compactietijd verdubbelt.

Men moet er dan wel rekening mee houden dat transformaties waarvan de uitvoeringsvolgorde bepaald wordt door de volgorde waarin data in het programma voorkomt, op de oorspronkelijke volgorde van de data in het programma werken, en niet op de volgorde van de verplaatste data. Dit is gelukkig heel makkelijk implementeerbaar.

Zelfs dan nog bestaat er evenwel een kleine kans dat data die volgens de eerste compactie dood was, de tweede keer levend blijft. De afstanden tussen data-adressen zijn immers veranderd door de herlocatie en sommige transformaties kunnen enkel uitgevoerd worden als de afstand tussen verschillende adressen klein genoeg is. Voor zo'n gevallen volstaat het echter na de tweede compactie na te gaan of alle data die dood moet zijn wel echt dood is: is dit niet het geval dan markeren we dat blok als niet verwijderbaar en beginnen we gewoon opnieuw.

3.3.7 Bespreking

Voor een cijfermatige discussie van de besproken analyses wachten we tot de algemene evaluatie van compactie na het linken. Hier kunnen we toch al een aantal zaken meegeven.

De geheugenvereisten zijn aanvaardbaar. Gelet op het feit dat we geen contextgevoelige constantenpropagatie uitvoeren is het additioneel benodigde geheugen lineair evenredig met de grootte van het te compacteren programma: per geheugenlocatie volstaan immers enkele bits om de veronderstellingen i.v.m. het dood of levend zijn en het al dan niet constant zijn van de data bij te houden. Daarnaast kan elke instructie slechts in één verzameling van te reëvalueren instructies opgenomen worden.

Merk op dat de analyses op verschillende manieren kunnen uitgebreid worden:

- Men zou de analyses zelf kunnen verfijnen. Daartoe zien wij echter niet veel mogelijkheden.
- Men kan de analyses waarop deze analyses steunen verfijnen. Zo zou men bv. het stapelgedrag beter kunnen trachten te analyseren, om meer waarden via de stapel te kunnen propageren en dus het "echte" gebruik ervan te kunnen analyseren.
- Men kan de granulariteit van de datablokken verfijnen. De datablokken waarmee wij werken zijn de datasecties uit de oorspron-

kelijke objectbestanden. Men kan proberen deze blokken te verkleinen door vertalers op een andere manier objectbestanden te laten genereren. Of men kan proberen de blokken a.d.h.v. nieuwe analyses na het linken te verfijnen. In principe komen alle mogelijke vormen van blokken in aanmerking die voldoen aan de vereiste dat geen berekeningen uitgevoerd worden die van één blok naar een ander leiden.

3.3.8 Verwant Werk

Voor zover ons bekend zijn er geen analyses die dezelfde doelstellingen hebben als de analyses die besproken zijn in deze sectie. Dit hoeft niet te verbazen, aangezien het nut in eerste instantie toch beperkt blijft tot compactie na het linken, wat een smal onderzoeksgebied is.

Zoals al hier en daar opgemerkt, is er echter conceptueel wel enige overlap met de wereld van partiële evaluatie [Jone93] en van abstractie interpretatie [Cous77].

Wat het elimineren van dode data uit programma's betreft stellen we vast dat zowat alle bekende technieken (gedeeltelijk) afhankelijk zijn van de programmeertaal waarin een programma geschreven is en bovendien werken op de broncode van het programma. Bij vele technieken is het verwijderen van dode data overigens enkel een neveneffect van het verwijderen van onbereikbare code uit de broncode. Voor die verwijderde code moet de vertaler immers in geen geval nog in statisch gealloceerde data voorzien. Tip en Sweeney [Swee98] hebben daarentegen wel de verwijdering van dode data op zich bekeken, meer bepaald het verwijderen van ongebruikte dataleden uit klassen in objectgeoriënteerde talen. Ze rapporteren een mogelijke compactie van de objectruimte (d.i. de maximale ruimte die dynamich gealloceerde objecten innemen tijdens de uitvoering van het programma) met gemiddeld 4.4%. Deze compactie is het gevolg van het elimineren van de 12% dode dataleden die hun methode gemiddeld ontdekt.

Het enige ons bekende onderzoek naar het verwijderen van dode data tijdens of na het linken is dat van Srivastava en Wall [Sriv94b]. Als neveneffect van de optimalisatie van indirecte datatoegangen en het gebruik van de globale wijzer, worden adressen in de globale adrestabel immers dood. Deze adressen verwijderen zij, wat de tabel kleiner maakt. Dit kan op zich leiden tot verdere optimalisaties, zeker als het aantal globale wijzers in een programma erdoor kan afnemen. Zij ver-

wijderen dus enkel de meest triviale gevallen van dode data, met heel specifieke technieken.

Er zijn ons voor het overige geen technieken bekend die gebruik maken van de eigenschappen van datablokken zoals het niet overschrijden van hun grenzen door berekeningen op adressen.

3.4 Overige analyses & optimalisaties

In deze sectie zullen we een aantal optimalisaties bespreken met eventueel de bijhorende analyses. We bespreken ze evenwel minder uitgebreid dan de voorgaande analyses en daarbij aansluitende optimalisaties. Daar zijn twee redenen voor:

- ze zijn minder belangrijk voor compactie tijdens het linken;
- we hebben minder werk verricht met betrekking tot deze analyses en er bijgevolg ook minder nieuwe zaken over te vertellen.

3.4.1 Aliasanalyse

Aliasanalyse of meer algemeen het analyseren en disambigueren van geheugenreferenties is waarschijnlijk een van de meest onderzochte en in de literatuur beschreven onderzoeksdomeinen. Op PLDI 2001 (*ACM Conference on Programming Language, Design and Implementation*) alleen al waren er bv. opnieuw 4 bijdragen over dit onderwerp.

De vragen waarop deze groep analyses een antwoord trachten te vinden zijn tweeërlei:

- Naar welke data kan een wijzer of een geheugenreferentie wijzen?
- Wat kan men zeggen over twee wijzers of geheugenreferenties met betrekking tot de data waar beide naar kunnen wijzen: wijzen ze zeker, misschien of zeker niet naar dezelfde data?

Afhankelijk van de uit te voeren optimalisaties waarvoor de antwoorden op deze vragen van belang zijn, zal men een of andere vorm van analyse kiezen. In de literatuur zijn er dan ook tal van criteria gebruikt om de nauwkeurigheid van analyses uit te drukken. Voorbeelden zijn het aantal koppels wijzers die zeker niet naar dezelfde data

kunnen wijzen, de gemiddelde groottes van de zogenaamde "wijst-naar" verzamelingen, d.w.z. het aantal locaties waarnaar een wijzer kan wijzen, enz.

Wij zijn van mening dat deze criteria heel artificieel zijn en dat men meer realistische criteria moet gebruiken om analyses te vergelijken. In ons geval zal dit dus het aantal instructies zijn dat geëlimineerd kan worden uit het programma door een aliasanalyse uit te voeren. Een uitstekende vergelijking tussen verschillende algoritmen met het oog op snelheidsoptimalisatie tijdens het vertalen vindt de lezer terug in [Ghiy01].

Aliasanalyse door code-inspectie

De algoritmen die we gebruiken voor aliasanalyse na het linken van een programma zijn uitvoerig beschreven in [Debr98] en [Muth99]. Ze vallen onder de noemer aliasanalyse door code-inspectie. Daarbij wordt getracht aan de hand van de programmasnedes van twee geheugenreferenties uit te maken of ze naar dezelfde locatie kunnen wijzen of niet. Indien de programmasnedes een gemeenschappelijk programmapunt hebben kunnen we door de berekeningen te vergelijken die na dat gemeenschappelijk punt uitgevoerd worden, afleiden of twee geheugenreferenties naar dezelfde locaties kunnen verwijzen.

Voorbeeld 3.12 Beschouw volgend stukje code:

$$
\begin{aligned}
\mathsf{r1} &:= \mathsf{r0} + 16 \\
\mathsf{r2} &:= \mathsf{r0} - 8 \\
\mathsf{r3} &:= \mathsf{r1} + \mathsf{r5} \\
\mathsf{r4} &:= \mathsf{r2} + \mathsf{r5} \\
\mathsf{r5} &:= \text{load byte } 0(\mathsf{r3}) \\
\mathsf{r6} &:= \text{load byte } 24(\mathsf{r4})
\end{aligned}
$$

De twee leesoperaties in deze figuur laden vanop dezelfde plek. De adressen van waarop ze laden zijn immers

$$
\begin{aligned}
\mathsf{r4} + 24 = \mathsf{r2} + \mathsf{r5} + 24 &= \mathsf{r0} + \mathsf{r5} + 16 \\
\mathsf{r3} + 0 = \mathsf{r1} + \mathsf{r5} + 0 \ \ &= \mathsf{r0} + \mathsf{r5} + 16
\end{aligned}
$$

□

Deze analyse wordt aangevuld met een aantal heuristieken als volgt:

- Indien er in een procedure geen aliassen van de stapelwijzer voorkomen, kunnen geheugenreferenties met de stapelwijzer als basisregister geen alias zijn van geheugenreferenties met een ander basisregister of van geheugenreferenties naar constante adressen.
- Een geheugenreferentie naar een constant adres waarvan men zoals in sectie 3.3 weet dat er enkel bekende geheugenreferenties naar zijn, kan geen alias zijn van een geheugenreferentie naar een niet-constant adres.

Deze laatste heuristiek is nieuw aangezien de analyses in sectie 3.3 nieuw zijn.

De analyses die uitgevoerd worden door vertalers zijn over het algemeen veel geavanceerder. Ze hebben dan ook veel meer informatie ter beschikking, zoals de datatypes van wijzers, en moeten veel minder rekening houden met allerlei wijzermanipulaties die enkel voorkomen in code op een laag niveau, zoals assemblercode. In [Wils95] probeert men hieraan gedeeltelijk tegemoet te komen omdat ook in C broncode veel wijzermanipulaties kunnen voorkomen.

Elimineren van geheugenoperaties

De voornaamste consument van de resultaten van de aliasanalyse is de eliminator van geheugenoperaties. De mogelijkheden om geheugenoperaties te verwijderen zijn:

1. Als er data weggeschreven wordt die nooit opgeladen zal worden kunnen we deze bijhorende schrijfoperaties elimineren. Het geval van dode statisch gealloceerde geheugenlocaties is uitvoerig besproken in sectie 3.3.
2. Een tweede geval is dat van loze-code-eliminatie van stapeloperaties. Indien blijkt dat het herstellen van een achteraf te bewaren register loos is, en de leesoperatie dus geëlimineerd wordt, kan men ook de bijhorende schrijfoperatie verwijderen.
3. Indien twee leesoperaties lezen vanop hetzelfde adres, zonder dat ernaar kan geschreven zijn tussen de leesoperaties door, kan men trachten de waarde die de eerste maal opgeladen is te bewaren in een register, zodat men ze geen tweede maal hoeft op te laden. Indien men de waarde daartoe niet in een tijdelijk register moet

kopiëren leidt dit tot de eliminatie van 1 instructie. Indien men toch een kopieerinstructie naar een tijdelijk register moet toevoegen, kan men hopen dat de eliminatie van kopieerinstructies ervoor zal zorgen dat men uiteindelijk een netto eliminatie van één instructie verkrijgt.

4. Indien twee instructies schrijven naar dezelfde geheugenlocatie zonder dat tussen de schrijfoperaties door vanop die locatie gelezen wordt, dan kan de eerste schrijfinstructie verwijderd worden.

Punten 2, 3 en 4 en de algoritmen om ze te benutten zijn uitvoerig besproken in [Muth99]. Deze mogelijkheden ontstaan doordat we (1) het volledige programma kunnen analyseren of (2) met andere transformaties vrije registers gecreëerd hebben waar de vertaler eerst zijn toevlucht moest nemen tot overloopcode.

Merk op dat deze optimalisatiemogelijkheden zowat het startpunt vormen van onderzoek naar optimalisaties na het linken. In [Wall86] bespreekt David Wall hoe een vertaler annotaties aan objectbestanden kan toevoegen waardoor de linker na een analyse van het volledige programma nodeloos toegevoegde overloopcode kan verwijderen.

3.4.2 Eliminatie van kopieerinstructies

Een algemeen bekende vertaaloptimalisatie is de propagatie van kopieën [Much97]: als de waarde in een register gekopieerd wordt naar een ander register kan men zolang ze beide dezelfde waarde hebben elke referentie naar het register met de kopie vervangen door een referentie naar het originele register. Indien op deze manier alle referenties kunnen omgezet worden, kan de kopieeroperatie verwijderd worden.

Ook na het linken kan deze situatie zich voordoen, o.m. na het toepassen van andere optimalisaties waardoor bv. de levensduur van registers verandert. In sectie 4.4 zullen bovendien kopieeroperaties toegevoegd worden in een belangrijke factorisatietechniek. Een gedeelte van deze kopieeroperaties kan achteraf terug verwijderd worden.

We verwijzen naar [Muth99] voor een bespreking van de mogelijke eliminatie van kopieerinstructies.

3.4.3 Proceduresubstitutie

Als een procedure slechts vanop één plek opgeroepen wordt, kan men de procedure-oproep beter substitueren door de opgeroepen procedure. De originele opgeroepen procedure kan dan verwijderd worden uit het programma. Ook indien het lichaam van een procedure kleiner is dan het aantal instructies dat nodig is om een procedure-oproep te implementeren, is het nuttig proceduresubstitutie toe te passen.

Soms is het mogelijk om de stapelvensters van oproeper en opgeroepen procedure samen te voegen na proceduresubstitutie. Daarmee spaart men dan meteen ook instructies uit voor de allocatie en deallocatie van het stapelvenster.

Men moet echter zorgvuldig te werk gaan bij het substitueren van procedure-oproepen: aangezien het lichaam van de gesubstitueerde procedure nu niet meer als het volledige lichaam van een procedure beschouwd wordt, zal dit stuk code door de verschillende analyses niet meer als een aparte procedure beschouwd worden. Op sommige analyses, zoals de analyse van het stapelgedrag, kan dit een slechte invloed hebben. Andere optimalisaties worden dan weer wel mogelijk omdat de gesubstitueerde procedure nu volledig deel uitmaakt van de context waaruit het oorspronkelijk opgeroepen werd.

3.4.4 Kijkgatoptimalisaties

Met het toepassen van alle programmatransformaties die tot nu toe besproken zijn is het logisch dat er zich nieuwe mogelijkheden tot kijkgatoptimalisaties voordoen, zoals sterktereductie.

3.4.5 Eliminatie van onbereikbare code

Eens we met constantenpropagatie ontdekt hebben dat pijlen uit de ICVG kunnen verwijderd worden, kunnen blokken in de graaf onbereikbaar worden: als geen enkele pijl er binnenkomt zal de code in het blok nooit uitgevoerd worden. We kunnen het blok m.a.w. verwijderen, waardoor ook misschien opvolgers van het blok onbereikbaar worden.

Ook in het initieel programma zit er echter meestal al een flinke hoeveelheid onbereikbare code. Dit is voornamelijk het geval bij nodeloos door de linker meegelinkte code.

Het detecteren van onbereikbare code gaat als volgt: markeer de ge-

wone helleknoop en de oproepershelleknoop en de ingangsknoop van het programma. Markeer dan iteratief alle opvolgers van gemarkeerde knopen. Knopen die na dit iteratief proces nog niet gemarkeerd zijn, zijn onbereikbaar.

Hoofdstuk 4

Factorisatie

"Wie spreekwoorden citeert krijgt wat hij wil."
Spreekwoord uit Zimbabwe

De programmatransformaties die we tot nu toe besproken hebben, zijn allemaal vrij algemene optimalisaties: zij komen gemiddeld genomen de uitvoeringssnelheid van een programma, de grootte van een programma, het vermogenverbruik van een programma en dergelijke meer ten goede. In dit hoofdstuk kijken we naar technieken die heel specifiek het compacteren van programma's beogen: codefactorisatie.

Factorisatie van code is het omgekeerde van proceduresubstitutie. Instructiesequenties die meermaals voorkomen in het programma worden eruit gelicht, en vervangen door één instructiesequentie die dan in de plaats van de originele sequenties uitgevoerd wordt. Opdat deze sequentie vanuit verschillende plaatsen zou kunnen opgeroepen worden, wordt ze vaak in een aparte procedure gestopt. Men spreekt dan ook wel van procedurale abstractie.

Om factorisatie toe te passen moeten volgende vragen beantwoord worden:

- Wat verstaan we onder identieke stukken code? Moeten die enkel functioneel equivalent zijn? Speelt de inroostering van instructies bv. een rol?

- Wat is de granulariteit van de identieke stukken code waar we naar op zoek gaan? Zijn het instructiesequenties, basisblokken, procedures of nog iets anders? Hoe lager de granulariteit, hoe

meer met elkaar te vergelijken stukken code men zal moeten bekijken, maar hoe makkelijker over het algemeen het vergelijken van twee stukken code is. Hoe lager de granulariteit, hoe groter ook de kans om identieke stukken code te vinden, maar hoe kleiner ook de opbrengst van factorisatie.

- Wanneer en hoe kunnen we de identieke stukken code vervangen door één stuk code, en hoe zorgen we ervoor dat dit stuk code zal uitgevoerd worden binnen de contexten van de originele stukken code? Welk controleverloop moet er m.a.w. toegevoegd worden?
- Hoe ver willen en kunnen we gaan om identieke stukken code te verkrijgen? Hernoemen we registers? Voegen we extra parameters toe?

In dit hoofdstuk worden op deze vragen antwoorden geformuleerd. Deze antwoorden zijn geordend volgens de granulariteit van de identieke stukken code die gezocht worden. Voor de verschillende granulariteiten zullen we een aantal antwoorden trachten te formuleren.

Daarnaast zullen we het hebben over de factorisatie van statisch gealloceerde data, wat een ons inziens nieuwe techniek is die als doelstelling heeft statisch gealloceerde data te hergebruiken waar mogelijk. Dit leidt tot compactere datasecties én indirect tot compactere codesecties.

Zoals in één van de vragen al gesuggereerd is, gaat codefactorisatie gepaard met het invoegen van controletransfers, die ervoor zorgen dat, alhoewel de statische hoeveelheid instructies in een programma daalt, het dynamisch aantal uitgevoerde instructies toeneemt. Dit vertraagt met andere woorden de uitvoering van gecompacteerde programma's, in tegenstelling met de in de vorige hoofdstukken besproken optimalisaties. We zullen daarom ook kort ingaan op mogelijke compromissen tussen snelheidsoptimalisatie en compactie.

4.1 Factorisatie van procedures

Het eerste niveau van granulariteit dat we bekijken is het procedureniveau. Op het eerste zicht lijkt het niet evident dat er verschillende instanties van identieke procedures in een programma voorkomen.

Toch blijkt dit voor te komen in programma's in het algemeen, en in programma's in object-georiënteerde talen in het bijzonder. Voor die talen is het gebruik van taalconstructies als sjablonen en overerving een

belangrijke bron van identieke procedures. Daarvoor bestaan twee belangrijke redenen:

- Uit een sjabloon wordt een zelfde concrete klasse geïnstantieerd in verschillende bronbestanden. Alhoewel er verschillende technieken bestaan om te vermijden dat deze identieke instanties allemaal in het programma terechtkomen, glippen er soms toch nog door de mazen van het net.
- Uit sjablonen worden verschillende concrete klassen geïnstantieerd op het niveau van de broncode. Vaak zullen methodes uit die concrete klassen er op assemblerniveau toch identiek uitzien. Dit is bv. typisch het geval voor zoek- en sorteermethodes in de zogenaamde containerklassen. Deze methodes verschillen in de broncode vaak enkel wat betreft de types van de wijzers die gebruikt worden. Omwille van typetoetsing moet dit onderscheid gemaakt worden tijdens het vertalen. Wijzers zijn echter vaak typeloos in de assemblercode die gegenereerd wordt voor de methodes. Het komt dus voor dat verschillende methodes op bronniveau aanleiding geven tot identieke procedures op het machineniveau.

Het komt er dan natuurlijk in de eerste plaats op aan deze identieke procedures te herkennen. Een gedetailleerde paarsgewijze vergelijking is niet erg efficiënt, dus daarom partitioneren we de procedures vooraf aan de hand van een vingerafdruk. Deze bestaat bv. uit het aantal instructies in de procedure, het aantal basisblokken in de procedure, het type van de basisblokken, enz. Procedures met een identieke vingerafdruk worden gegroepeerd, bv. door de procedures te sorteren op de vingerafdruk of door de vingerafdruk (of een afgeleide ervan) als een sleutel voor een hakseltabel te gebruiker. Een gedetailleerde vergelijking van procedures is slechts nodig binnen elke groep.

De procedures waarop we ons richten zijn die procedures die omwille van de hierboven vermelde redenen identiek zijn. Daarbij veronderstellen we dat de vertaler voor (op types van wijzers na) identieke procedures in broncode identieke procedures in assembler zal genereren. We beschouwen dan ook enkel echt identieke procedures: basisblokken moeten in dezelfde volgorde staan, instructies moeten in dezelfde volgorde staan, ze moeten identiek dezelfde registers gebruiken, etc. We proberen dus geen transformaties uit om procedures identiek te maken. Merk overigens op dat identiek zijn ook inhoudt dat behalve

voor compenserende en terugkeerpijlen die uit de procedures vertrekken, de doelen van interprocedurale controletransfers uit de procedures dezelfde moeten zijn. Dit geldt voor procedure-oproepen, maar evengoed voor andere interprocedurale sprongen[1].

Eens we twee identieke procedures gevonden hebben, worden alle verwijzingen naar één van beide procedures in het programma omgezet naar verwijzingen naar de andere procedure. Dit houdt o.a. in dat we alle code-adressen uit die procedure in de datasecties aanpassen en dat we oproep- en terugkeerpijlen in de ICVG verplaatsen. Er worden dus geen onkosten in het programma geïntroduceerd bij deze vorm van factorisatie, aangezien we nergens instructies toevoegen.

Merk op dat het factoriseren van procedures een iteratief proces is: soms zijn procedures identiek, op de procedures die zij oproepen na. Indien de opgeroepen procedures echter dezelfde worden door factorisatie, worden ook deze procedures identiek en kunnen we ze ook factoriseren.

Zoals we in sectie 2.3.5 gesteld hebben, hebben we voorlopig in onze algoritmen geen rekening gehouden met het afhandelen van excepties. Uiteraard moeten we dit voor programma's in object-georiënteerde talen wel doen. Wat deze vorm van factorisatie betreft kunnen zich problemen stellen. We gaan er hier kort op in, maar willen nu al duidelijk stellen dat we bij de evaluatie van onze algoritmen geen rekening gehouden hebben met deze problematiek.

Er zijn algemeen genomen twee mogelijkheden om een exceptie op te werpen en af te handelen:

1. De excepties die opgeworpen of opgevangen worden, worden volledig bepaald door de opcodes van de instructies uit de procedure en de registerinhouden. In dit geval is er geen probleem: als de procedures identiek zijn, zijn ook de op te werpen excepties identiek.

2. De excepties die opgevangen worden, hangen ook af van de locatie van de code in het geheugen. Aan de hand van het adres van de instructie die een exceptie opwerpt zal dan een bepaalde

[1] Een uitzondering maken we voor oproepen naar de procedures die vergeleken worden zelf. Tijdens het vergelijken van twee procedures worden zulke oproepen vanuit beide procedures als identiek beschouwd, of ze nu dezelfde van de twee vergeleken procedures oproepen of niet. Op deze manier worden identieke recursieve procedures gedetecteerd, evenals identieke wederzijds recursieve procedures.

afhandelaar uitgevoerd worden. Daartoe worden de excepties en hun afhandelaars beschreven in aparte datasecties, die een link leggen tussen code-adressen en excepties. In dit geval kan een probleem ontstaan als de excepties die door de identieke procedures opgevangen worden, verschillen.

Een mogelijke oplossing bestaat eruit extra onkosten toe te voegen in het programma: men kan van de originele procedures inpakprocedures rond de gefactoriseerde procedure maken. In de exceptiebeschrijving moeten dan ook de excepties ingepakt worden. De gefactoriseerde procedure doet dan niks meer dan de op te vangen excepties doorgeven aan zijn oproepers, zijnde de inpakprocedures, die de correcte afhandeling verzorgen.

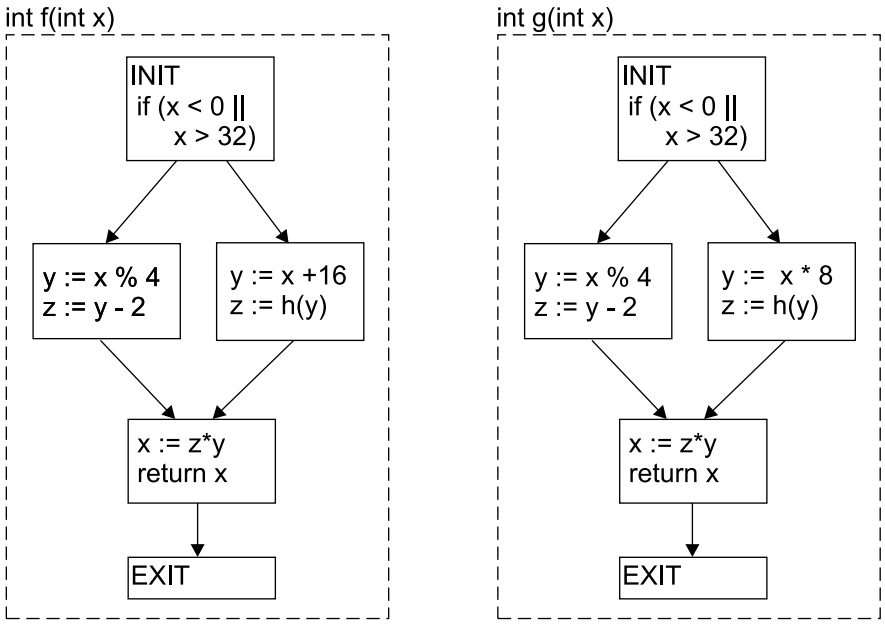
4.2 Parametrisatie van procedures

Om dezelfde redenen dat identieke procedures voorkomen in een programma, kunnen er ook bijna identieke procedures in voorkomen. Het verschil tussen twee procedures die uit sjablonen komen, kan bv. beperkt zijn tot het oproepen van verschillende subprocedures. Of als bv. in de broncode objecten met een verschillende grootte gemanipuleerd worden op voor het overige identieke wijze, zullen indexoperaties verschillen in de procedure.

Men kan dan twee kanten op:

- Ofwel factoriseert men de identieke stukken uit de procedures zoals in de volgende secties besproken wordt. Het nadeel is dan dat er voor deze identieke stukken steeds onkosten zullen toegevoegd worden om die stukken uit te voeren.

- Of men probeert de procedures samen te voegen tot één procedure, waarin voor verschillen in de originele procedures beide versies in de samengevoegde procedure zitten. Tijdens de uitvoering van de samengevoegde procedure zal men dan aan de hand van een extra parameter de correcte code uitvoeren. De parameter wordt gezet in de originele procedures die verder enkel als inpakprocedure fungeren. Hier zal de geïntroduceerde onkost niet voorkomen waar de procedures identiek zijn, maar waar er verschilpunten zijn.

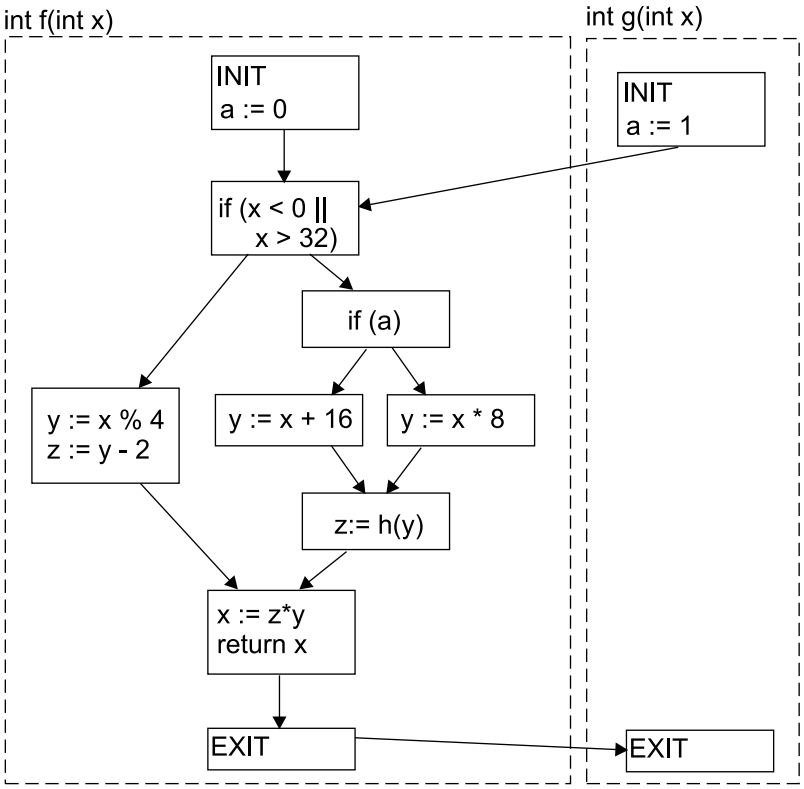


Figuur 4.1: Kandidaten voor parametrisatie.

Als het aantal verschilpunten dus beperkt is, kan men beter de procedures trachten te factoriseren door parametrisatie.

Voorbeeld 4.1 Beschouw de procedures f() en g() in figuur 4.1. De procedures zijn heel gelijkaardig, afgezien van één operatie: in de ene procedure is het een vermenigvuldiging, in de andere een optelling. Men kan trachten de identieke basisblokken te factoriseren zoals in sectie 4.4 wordt besproken. Men bekomt echter een betere compactie indien het lichaam van procedure f() geparametriseerd wordt en indien dat lichaam ook vanuit g() uitgevoerd wordt. Dit is afgebeeld in figuur 4.2 □

Om zulke kansen op te sporen werken we ongeveer zoals bij de factorisatie van identieke procedures. Voor elke procedure wordt dus opnieuw een vingerafdruk opgesteld. Dit is in dit geval een tekenrij waarin eerst het aantal basisblokken van de procedure staat en dan voor elk basisblok in de procedure het type aangegeven wordt door een letter. Dit zijn bv. 'O' voor een oproepblok, 'C' voor een blok dat eindigt met een conditionele sprong, etc. Alle procedures worden diepte-eerst doorlopen, waarbij de types van de blokken tot een tekenrij samenge-



Figuur 4.2: Geparametriseerde factorisatie.

voegd worden. De procedures worden vervolgens gepartitioneerd op basis van de vingerafdrukken. Binnen een groep gelijkaardige procedures wordt vervolgens nagegaan of ze een identieke structuur hebben, waarbij vooralsnog de instructies buiten beschouwing worden gelaten. Zijn de procedures niet identiek qua structuur, dan proberen we ze niet samen te voegen.

Zijn ze echter wel gelijk van structuur, dan wordt geschat of het nuttig zou zijn de procedures samen te voegen. Voorlopig beperken we ons tot twee gelijkaardige procedures, we voegen m.a.w. niet meer dan twee procedures samen. Daartoe overlopen we alle paren van corresponderende basisblokken in de twee procedures, waarbij we "equivalentiepunten" accumuleren als volgt:

- We overlopen voor elk basisblok de instructies van begin tot einde. Voor elke identieke instructie winnen we 1 punt. Als de

instructies niet allemaal identiek zijn, beginnen we achterwaarts te vergelijken, waarbij elke identieke instructie opnieuw 1 punt waard is. In dat geval trekken we van het totale aantal punten ook de kost af van het toevoegen van een conditionele sprong op basis van de parameter. Deze kost is het aantal instructies dat toegevoegd moet worden bij verschillen tussen beide procedures om tijdens de uitvoering het correcte stuk code te selecteren op basis van de parameter.

- Van het aantal punten wordt eveneens de kost afgetrokken van het zetten van de parameter aan het begin van beide procedures.
- Het aantal equivalentiepunten is dus finaal het aantal instructies dat uitgespaard wordt door de betrokken procedures samen te voegen en te parametriseren. Als dit aantal positief is, wordt de parametrisatie uitgevoerd.

Alhoewel de vooropgestelde methode uiterst eenvoudig is, worden er toch heel wat kandidaten voor factorisatie mee gevonden. Tot nu toe lijken we er dus uitstekend in te slagen de grotere onkost van het apart factoriseren van basisblokken uit gelijkaardige procedures te ontwijken.

Er is echter één groot probleem waar we vooralsnog geen bevredigende oplossing voor gevonden hebben: waar wordt de parameter bijgehouden? Idealiter is dit in een register, maar als er geen vrij register voorhanden is kunnen we de parameter in het geheugen opslaan. Daartoe kunnen we dan zelf een nieuw blokje statisch geheugen alloceren in de datasectie. Voor recursieve procedures moeten lokale, dynamische (in de betekenis van niet-*static* lokale veranderlijken zoals in de C taal) echter op de stapel bewaard worden. Ook de parameter moeten we dus op de stapel plaatsen, indien het geparametriseerde lichaam van de procedures recursief kan uitgevoerd worden.

We moeten dus (1) nagaan of die nieuw gevormde procedure recursief kan opgeroepen worden, en (2) nagaan hoe we de parameter dan op de stapel kunnen bewaren. Omwille van het grote aantal pijlen vanuit de oproepershelleknoop en naar de opgeroepenenhelleknoop lijkt het er meestal op dat nagenoeg alle geteste procedures recursief zijn. Wat problematisch is, aangezien we nog geen veilige (conservatieve) manier gevonden hebben om het stapelvenster van procedures te vergroten en er bv. een parameter te kunnen in opslaan. De voornaamste reden is dat we vaak conservatief (moeten) veronderstellen dat er er-

gens in de oproepketting vanuit de geparametriseerde procedure verwezen wordt naar het stapelvenster ervan, om eruit te lezen of erin te schrijven. Dan moeten we die lees- en schrijfoperaties aanpassen aan de gewijzigde grootte van het stapelvenster, wat we (voorlopig) nog niet kunnen. Wanneer er in die oproepketting een oproep naar de opgeroepenhelleprocedure zit weten we zelfs niet of en waar er naar het gewijzigde stapelvenster zal verwezen worden. Toekomstig onderzoek zal moeten uitwijzen of hier al dan niet meer stappen in gezet kunnen worden.

Merk op dat het bijhouden van de parameter op de stapel overigens niet meer instructies zal vereisen dan wanneer de veranderlijke in statisch gealloceerd geheugen wordt gestopt. Zoals de evaluatie ons zal leren, is deze techniek zeer beloftevol, maar vooralsnog kunnen we hem dus niet toepassen. We zien jammer genoeg ook niet meteen een manier om ze wel te kunnen toepassen zonder terug te moeten vallen op extra extern beschikbare informatie i.v.m. de oproepgraaf van een programma.

4.3 Factorisatie van regio's

Indien er op procedureniveau niet kan gefactoriseerd worden omdat procedures niet volledig identiek zijn of indien er niet kan geparametriseerd worden omwille van problemen met het opslaan van de parameter, moet men zijn toevlucht nemen tot het factoriseren van kleinere stukken code. Een mogelijkheid is de coderegio: dit is een stuk code met een uniek ingangspunt en een uniek uitgangspunt. Men ziet makkelijk in dat een coderegio gekarakteriseerd wordt door een paar basisblokken (d, p) waarbij d een dominator is van p, en p een postdominator van d. Omgekeerd zal ook elk zulk paar eenduidig een regio bepalen, gegeven de ICVG. Identieke regio's kunnen we in een nieuwe aparte procedure onderbrengen, en de originele regio's worden vervangen door een procedure-oproep naar de nieuwe procedure.

Basisblokken zijn uiteraard de eenvoudigste vorm van regio's. Omwille van hun specifieke eigenschappen zullen we basisblokken in de volgende sectie apart behandelen. Hier zullen we het dus hebben over regio's die uit verscheidene basisblokken bestaan.

Om regio's te vergelijken met elkaar maken we opnieuw gebruik van vingerafdrukken, die nu bv. het aantal basisblokken en het aantal instructies van een regio bevatten en gebruikt worden om regio's op

voorhand te partitioneren. Omdat er veel meer regio's voorkomen in programma's dan procedures, hebben we onze exploratie van de factorisatie van regio's voorlopig beperkt tot identieke regio's. We zullen dus niet naar analogie met procedures een eventuele parametrisatie proberen.

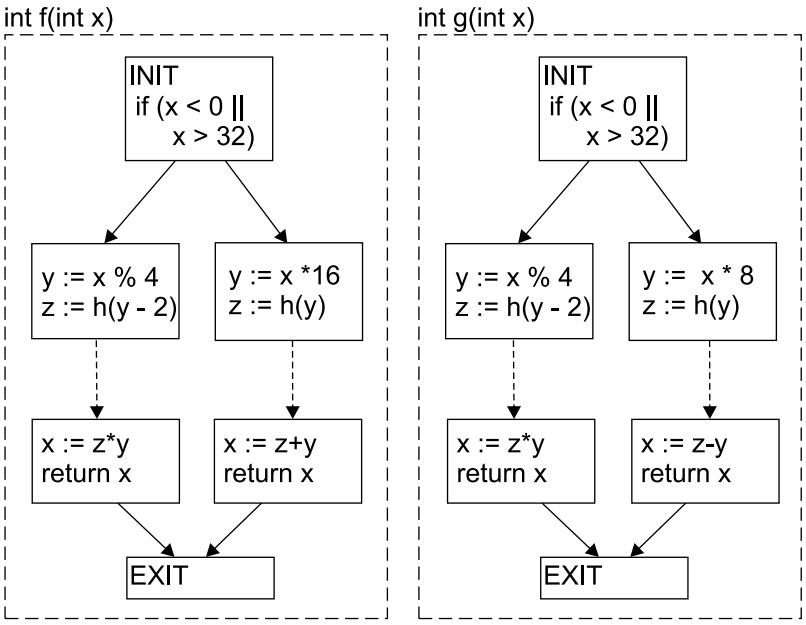
Desalniettemin moeten we toch een plaats vinden om een specifiek soort parameter op te slaan, nl. het terugkeeradres om terug te keren naar de programmapunten volgend op de oorspronkelijke regio's. En net als bij de parametrisatie van procedures levert dit problemen op: meestal zullen we een locatie op de stapel moeten gebruiken om dit terugkeeradres op te slaan.

In een aantal gevallen kunnen we die problemen echter vermijden:

- Als de identieke coderegio's eindigen met een terugkeerinstructie, volstaat het alle binnenkomende pijlen in alle regio's naar de ingang van één enkele regio te laten wijzen. Het terugkeeradres voor de terugkeerinstructies van de originele regio's was bij wijze van spreken al een soort parameter. We hergebruiken die gewoon. Een voorbeeldje van dergelijke factorisatie is afgebeeld in figuren 4.3 en 4.4.

- Identieke regio's waarin geen procedure-oproepen voorkomen, kunnen geen aanleiding geven tot een recursieve gefactoriseerde versie. In zulke gevallen kan men de parameter dus gewoon in een register bewaren. Daartoe moet men natuurlijk wel over een register beschikken dat vrij is in de hele regio's. Dit verifiëren is triviaal.

- In een aantal gevallen kunnen we veilig een extra locatie in het stapelvenster voorzien. Daartoe moet het stapelgedrag binnen de regio en binnen de erin opgeroepen procedures aan allerlei eisen voldoen. Het formuleren van deze eisen hangt heel sterk samen met de interne voorstelling van het stapelgedrag van bv. procedures in een programma. Er hier in detail op ingaan zou ons dan ook te ver leiden. We beperken er ons toe te stellen dat we in ons prototype een uiterst conservatieve analyse hebben geïmplementeerd die het in een aantal gevallen effectief mogelijk maakt extra plaats op de stapel te gebruiken. Ze is echter niet toepasbaar in de context van parametrisatie van procedures. De reden is heel concreet dat de analyse o.a. nagaat of de stapelwijzer bewerkt wordt in de dominator of de postdominator van de



Figuur 4.3: Kandidaten voor factorisatie van een regio eindigend met een terugkeerinstructie.

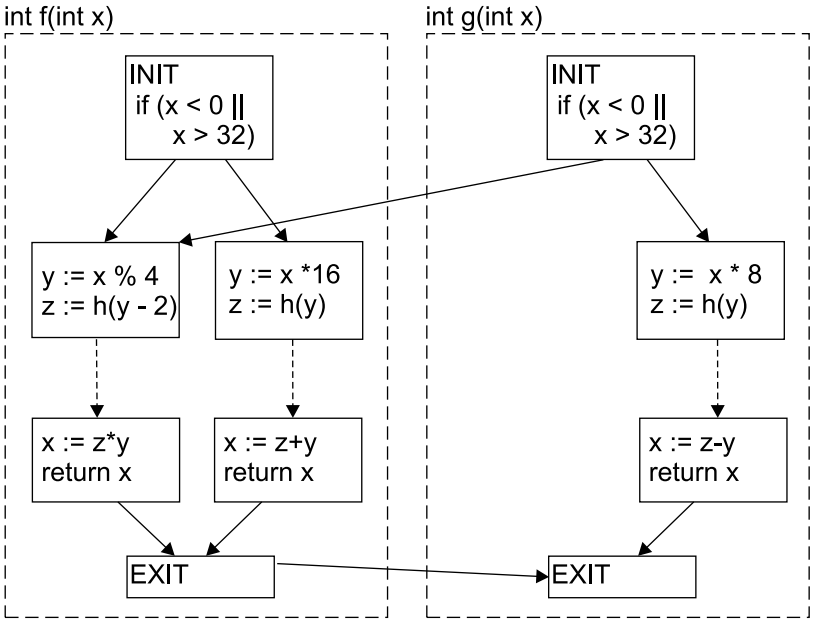
regio. Indien dit het geval is, kunnen we niet veilig het stapelvenster vergroten. Aangezien voor alle niet-triviale procedures waarvan we het stapelgedrag enigszins kunnen analyseren in het ingangsblok (d.i. de dominator) het stapelvenster wordt gealloceerd, voldoet een volledige procedure (als regio bekeken) niet aan deze vereiste.

4.4 Factorisatie van basisblokken

De eenvoudigste vorm van coderegio's zijn basisblokken: er is geen controleverloop binnen een basisblok, afgezien van de sequentiële uitvoering. Dit zorgt ervoor dat het relatief makkelijk is om na te gaan of twee blokken functioneel equivalent zijn, d.w.z. dat ze dezelfde berekeningen uitvoeren.

Basisblokken kunnen functioneel equivalent zijn, maar toch niet identiek aan elkaar, omwille van volgende oorzaken:

- De berekeningen kunnen in een andere volgorde uitgevoerd wor-



Figuur 4.4: Een gefactoriseerde regio eindigend met een terugkeerinstructie.

den. Dit kan een gevolg zijn van de inroostering door de vertaler, die voor elk basisblok anders kan geweest zijn, of van onze andere optimalisaties die vertrekkende van functioneel verschillende blokken functioneel equivalente blokken gemaakt hebben.

Over de gevolgen van deze oorzaak hebben we ons niet druk gemaakt tijdens het onderzoek. Experimenten met ons prototype hebben immers aangetoond dat het opnieuw inroosteren van instructies binnen basisblokken, zodat mogelijke verschillen in instructievolgorde weggewerkt worden, geen enkele invloed heeft op de compactiefactor, d.i. dat er niet meer basisblokken gevonden worden om te factoriseren. Let wel, dit zou best aan de achterkant van de vertaler kunnen liggen waarmee we de programma's vertaald hebben: indien we bv. een vertaler zouden gebruikt hebben waarin meer globale inroostertechnieken gebruikt worden, zou dit resultaat best anders kunnen uitvallen. In de rest van deze sectie gaan we dus steeds uit van de (toevallige) instructievolgorde van onze interne representatie, die meestal nauwelijks afwijkt van de volgorde die de vertaler gegenereerd heeft.

- De berekeningen worden op andere data uitgevoerd. In ons geval

slaat dit op registers: lokale tijdelijke registers kunnen verschillen, net zoals de registers waarin de invoerdata zit voor de lokale berekeningen of de registers waarin de resultaten zitten op het einde van het blok. Een voorbeeld hiervan is te zien in blokken A en B in figuur 4.5. Aan deze verschillen zullen we trachten te verhelpen door registers te hernoemen.

Een andere mogelijkheid is dat de in de instructies gecodeerde letterlijke operandi verschillen. Hieraan zullen we gedeeltelijk tegemoetkomen met een oplossing die dicht aanleunt bij het hernoemen van registers.

- De opvolgers in de ICVG verschillen. Om problemen hieromtrent te vermijden splitsen we elk basisblok op tot een zogenaamd "normaal" blok en een transferblok: het transferblok bestaat uit enkel de controletransferinstructie, het normaal blok bestaat uit de overige instructies en heeft als enige opvolger het transferblok. Omdat de opvolgers van identieke normale blokken dan nog steeds verschillen, laten we gewoon de onvoorwaardelijke sprongen uit onze intermediaire voorstelling weg.

  Merk op dat dit geen invloed heeft op andere optimalisaties of transformaties, integendeel. Aan welke blokken onvoorwaardelijke spronginstructies moeten toegevoegd worden hangt immers enkel af van de codelay-out. Het is dus de evidentie zelve dat we met zulke sprongen geen rekening wensen te houden zodra de ICVG gevormd is.

  Merk ook op dat dit opsplitsen ook voor de factorisatie van coderegio's uitgevoerd wordt.

- Soms kan het voorkomen dat een identieke operatie met verschillende instructies kan geïmplementeerd worden. Een groot aantal algemeen-toepasbare architecturen hebben bv. instructies die optellingen met een constant getal kunnen uitvoeren met behulp van de lees/schrijfeenheid op de processor, om de aritmetische functionele eenheid gedeeltelijk te ontlasten. De constante die bij een registerinhoud dient opgeteld te worden wordt dan als letterlijke afstand gecodeerd in de instructie. Daarnaast kan men in gewone optellingsinstructies ook vaak een letterlijk operand coderen.

  Ook aan de abstractie van dit soort verschillen hebben we gedacht tijdens ons onderzoek. In ons prototype hebben we daartoe een

normaliseringsfase geïmplementeerd, die voor operaties met verschillende implementatiemogelijkheden, er een vaste kiest. Net zoals met het herordenen van instructies kan ook hiermee echter geen grotere compactie bereikt worden. Dat dit niets opbrengt ligt aan de consequente keuzes die zowel onze transformaties als de vertaler maken bij het kiezen van een instructie om een elementaire operatie te coderen. Op een vrij orthogonale architectuur als de Alpha ligt dit eigenlijk voor de hand. We kunnen ons echter voorstellen dat dit op minder orthogonale architecturen niet zo is, en dat een vertaler die daarvoor heel verfijnd instructies selecteert en inroostert tot andere resultaten aanleiding kan geven.

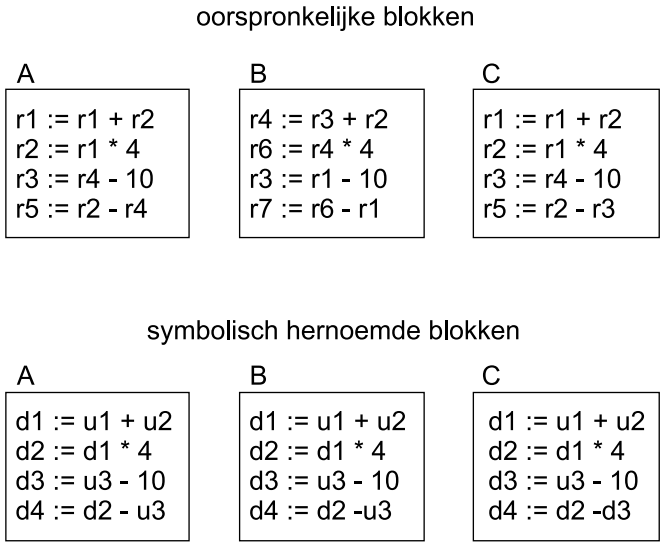
Samenvattend moeten we dus enkel nog op zoek naar blokken die alleen wat betreft de operandi van de instructies kunnen verschillen, maar voor het overige dezelfde berekeningen uitvoeren. Dit betekent dat de opcodes van de instructies dezelfde moeten zijn (en in dezelfde volgorde moeten voorkomen) en de gerichte acyclische afhankelijkheidsgraaf (GAAG) van het basisblok er identiek moet uitzien. Om het laatste te evalueren zullen we symbolische registerhernoeming gebruiken.

4.4.1 De opcodes

Zoals in de voorgaande secties zullen we de vergelijking van paren versnellen door de blokken op voorhand te partitioneren. De vingerafdruk daarvoor halen we uit het aantal instructies in een blok en de opcodes van de instructies in een basisblok. Als de opcodes niet allemaal volledig samen in de sleutel kunnen, zal een verfijnder vergelijking toch nog nodig zijn na het sorteren of het groeperen. Deze verfijnder vergelijking zal in ieder geval niet zo vaak meer moeten uitgevoerd worden.

4.4.2 De symbolisch hernoemde registers

Het is niet omdat basisblokken eenzelfde sequentie van instructies bevatten dat ze functioneel equivalent zijn. Functionele equivalentie wordt typisch nagegaan door de GAAGen van twee blokken te vergelijken. Op een intermediaire drie-adresrepresentatie kan men dit praktisch uitvoeren door registers symbolisch te hernoemen en dan de hernoemde code letterlijk te vergelijken.



Figuur 4.5: Enkele basisblokken met hun symbolische hernoeming.

Voorbeeld 4.2 Beschouw de basisblokken in figuur 4.5. Alhoewel deze dezelfde instructies bevatten, in dezelfde volgorde, zijn ze toch niet allemaal functioneel equivalent. Blok A is functioneel equivalent met blok B, maar niet met C. De verschillen tussen A en B onderling lijken nochtans groter dan die tussen A en C.

Onder de originele blokken hebben we de symbolisch hernoemde blokken geplaatst. Deze staan in een SUT notatie, waarbij registers in lexicale volgorde hernoemd zijn. Registers die lokaal gedefinieerd worden zijn hernoemd naar symbolische 'd' registers, registers die niet lokaal gedefinieerd zijn of gebruikt worden voor ze lokaal gedefinieerd zijn, zijn naar 'u' registers hernoemd.

Na de hernoeming blijkt duidelijk dat A en B functioneel equivalent zijn met elkaar, C daarentegen niet met de andere.

□

Daarnaast is het zo dat functionele equivalentie geen strikte vereiste is, maar dat het volstaat dat het blok waarnaar hernoemd zal worden functioneel superieur is aan het te hernoemen blok. Een blok A is functioneel superieur aan een ander blok B als A dezelfde berekeningen dan B kan uitvoeren, maar dit omgekeerd niet het geval is.

Voorbeeld 4.3 Beschouw de basisblokken in figuur 4.6. De symbolisch

hernoemde versies van de blokken laten uitschijnen dat ze niet functioneel equivalent zijn. Dit is ook zo.

Toch kunnen we de berekeningen uit blok B gebruiken om de berekeningen uit blok A uit te voeren, mits het toevoegen van kopieeroperaties. Dit is weergegeven in de registerhernoemde blokken. Om deze mogelijkheid voorlopig open te laten terwijl we de mogelijke equivalentie van blokken nagaan, kunnen we op een andere manier de registers symbolisch hernoemen, waarbij we (nog) geen onderscheid maken tussen extern gedefinieerde registers. Dit is onderaan in figuur 4.6 weergegeven.

Merk overigens op dat we omgekeerd blok A niet kunnen gebruiken om de berekeningen van blok B uit te voeren.

□

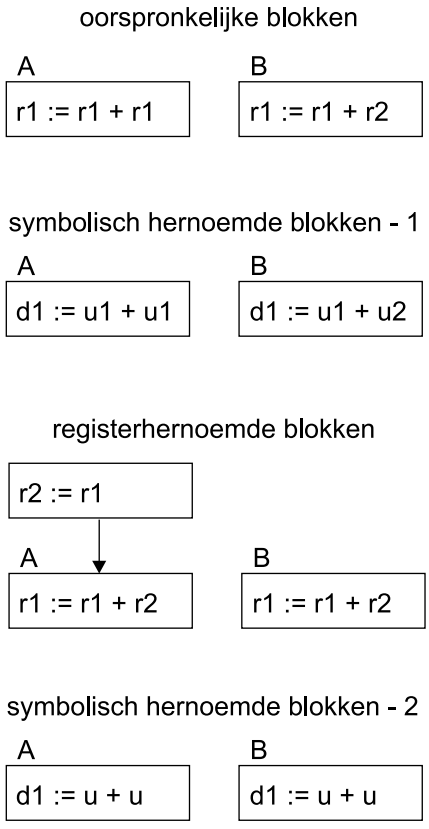
Door alle extern gedefinieerde registers te hernoemen tot hetzelfde symbolische register laten we vooralsnog alle externe pijlen van de GAAG buiten beschouwing. We zullen deze in beschouwing nemen bij het effectief trachten te hernoemen van registers, wat in de volgende sectie aan bod komt.

Merk op dat bij het vergelijken van de symbolisch hernoemde code van twee blokken de volgorde waarin operandi voorkomen in instructies een rol speelt. Dit hoeft echter geen rol te spelen voor de zogenaamde commutatieve instructies. Voor zulke instructies zullen we de bronoperandi dan ook omwisselen indien dit nuttig is om een blok te kunnen factoriseren.

4.4.3 De eigenlijke registerhernoeming

Als twee blokken bestaan uit identieke instructies, is factorisatie ervan vrij eenvoudig. Het volstaat een gezamenlijk vrij register te vinden om het terugkeeradres in op te slaan en de oorspronkelijke blokken te vervangen door procedure-oproepen naar de gefactoriseerde code.

Als de registers waarmee de twee blokken rekenen niet dezelfde zijn, moeten we eerst in een blok de registers hernoemen om de factorisatie te kunnen uitvoeren. Zoals in het voorbeeld in de voorgaande sectie getoond, zullen we dit doen door instructies toe te voegen die registerinhouden kopiëren voor en na het te factoriseren blok. Er zijn vier redenen waarom we kopieeroperaties moeten toevoegen:



Figuur 4.6: Enkele basisblokken met hun gedeeltelijke symbolische hernoeming.

1. Er worden andere extern gedefinieerde registers geconsumeerd. Om dit op te vangen voegen we kopieeroperaties toe voor het te factoriseren en te hernoemen blok.

2. De registers die gedefinieerd worden in het te herbenoemen blok en levend zijn op het einde van dat blok verschillen van de ermee corresponderende registers in het doelblok (d.i. het blok waarnaar het blok hernoemd moet worden). Dit wordt opgevangen door kopieeroperaties na het te hernoemen blok toe te voegen.

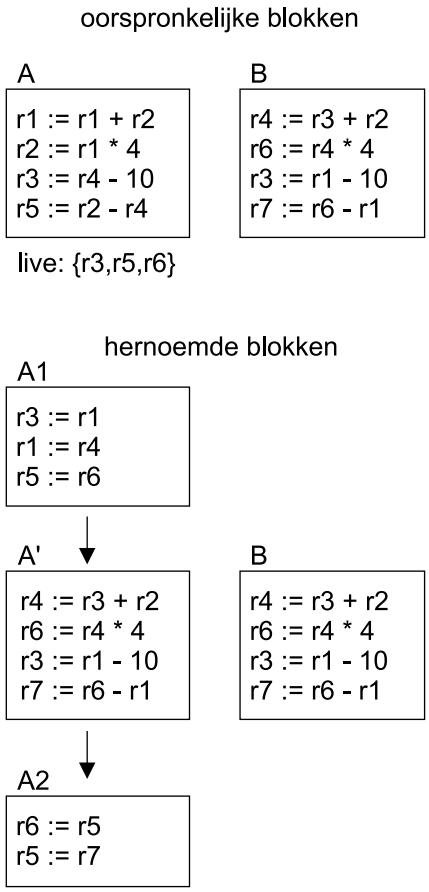
3. In het doelblok worden registers gedefinieerd die levend zijn voor en na het te hernoemen blok en bewaard worden doorheen het hele te hernoemen blok. In zo'n geval moeten kopieeroperaties toegevoegd worden die de te bewaren registerinhouden tijdelijk in andere registers opslaan. Er worden dan kopieerinstructies voor en na het te hernoemen blok toegevoegd.

4. Indien er levende registers zijn voor en over het te hernoemen blok, die overschreven worden door in punt 1 toegevoegde kopieeroperaties, moeten ook hun registerinhouden tijdelijk ergens anders bewaard worden. Dat gebeurt opnieuw door vooraan en achteraan kopieeroperaties toe te voegen naar en vanuit een vrij register.

Voorbeeld 4.4 Bekijk nogmaals de blokken A en B uit figuur 4.5. In figuur 4.7 hebben we blok A hernoemd naar blok B door er de benodigde kopieeroperaties aan toe te voegen. Deze hangen sterk af van de levensduurinformatie op het einde van het te hernoemen blok. We hebben dan ook een (willekeurige) verzameling levende registers gekozen om het voorbeeld uit te werken.

De eerste twee kopieeroperaties blok A1 zijn toegevoegd omwille van de eerste regel in het lijstje hierboven. Merk op dat de volgorde waarin deze instructies toegevoegd worden een rol speelt. De laatste operatie in blok A1 en de eerste uit A2 zijn nodig omdat de waarde in register r6 bewaard moet blijven, en ze in blok B overschreven wordt. De laatste operatie in blok A2 tenslotte is nodig omdat de waarde die in het originele blok A in register r5 werd weggeschreven nu in r7 wordt weggeschreven. □

Nagaan welke kopieeroperaties moeten toegevoegd worden is heel eenvoudig: voor gevallen 1 en 2 uit bovenstaand lijstje overlopen we



Figuur 4.7: Enkele basisblokken met hun uiteindelijke registerhernoeming.

de instructies in beide blokken en houden bij welk register naar waar moet gekopieerd worden voor en na het blok. Indien er op een bepaald moment twee registers naar hetzelfde register moeten gekopieerd worden, zowel voor als achter het te hernoemen blok, is hernoeming onmogelijk in de richting waarvoor we de hernoeming proberen. Daarna kijken we welke registerinhouden tijdelijk moeten bewaard worden in andere registers en voegen we ook daarvoor kopieeroperaties toe. Indien het aantal toe te voegen instructies kleiner is dan de winst die geboekt wordt door de basisblokken te factoriseren, voeren we de hernoeming ook effectief door, in het andere geval doen we dit niet.

Net zoals er kopieeroperaties toegevoegd worden om registers te hernoemen, zouden we instructies kunnen toevoegen om letterlijke operandi naar een register te "hernoemen". We voegen dan voor de naar elkaar te vernoemen blokken instructies in die de letterlijke waarde in een register schrijven en vervangen de letterlijke waarde in het blok door het gebruik van dit register. Dit komt overeen met het omgekeerde van constantenpropagatie. We hebben evenwel gemerkt dat deze extra hernoemingsmogelijkheid geen extra compactie oplevert.

4.5 Factorisatie van instructiesequenties

In deze sectie zullen we naar nog een kleinere granulariteit afdalen: instructiesequenties binnen basisblokken. We zullen dit doen voor willekeurige instructiesequenties en voor een aantal specifieke instructiesequenties die vaak terugkomen omwille van het naleven van oproepconventies.

4.5.1 Willekeurige instructiesequenties

Indien basisblokken niet kunnen hernoemd worden naar elkaar, kan het wel zijn dat er nog delen van de blokken gelijk zijn. Omdat er echter veel meer instructiesequenties zijn dan basisblokken, zou het te veel tijd in beslag nemen om ook equivalente instructiesequenties te herkennen en naar elkaar trachten te hernoemen. Bovendien zitten we met het bijkomend probleem dat instructies tot meer dan één instructiesequentie behoren. We zullen dus enkel volledig identieke sequenties beschouwen.

We werken daarom als volgt:

1. Beschouw de verzameling van instructiesequenties. Elk element stemt overeen met een instructiesequentie en bestaat uit een tekenrij van de aaneengeplakte opcodes van de instructies in de sequentie, het aantal instructies in de sequentie en het nummer van de eerste instructie van de sequentie. Deze verzameling wordt opnieuw eerst gepartitioneerd.

2. Als er twee of meer identieke sequenties gevonden worden, worden deze sequenties in aparte blokken ondergebracht door basisblokken te splitsen. Deze aparte, nieuwe blokken worden gemarkeerd zodat ze later zelf niet meer opgesplitst worden. Zodoende worden grote identieke instructiesequenties bevoordeeld. Dit is een gulzig selectiemechanisme, dat niet altijd optimaal hoeft te zijn, maar dat efficiënt is wat betreft tijdsgebruik.

3. Eens de blokken gesplitst zijn, passen we opnieuw de factorisatie van basisblokken toe. De nieuwe gecreëerde blokken zullen geen registerhernoeming behoeven, maar de resten van de gesplitste blokken kunnen er misschien wel nog baat bij hebben. We hergebruiken op deze manier ook de infrastructuur voor de factorisatie van basisblokken, zoals bv. de code om een register voor het terugkeeradres te vinden.

Merk overigens op dat we in een latere transformatie paren basisblokken die de unieke voorganger en opvolger van elkaar zijn, zullen samenvoegen tot één basisblok, zodat eventueel nutteloos opgesplitste blokken terug samengevoegd worden.

4.5.2 Specifieke instructiesequenties

Vaak terugkerende instructiesequenties zijn de sequenties die de achteraf te bewaren registers op de stapel zetten aan de ingang van procedures en ze er terug afhalen aan de uitgangen.

Of deze sequenties makkelijk kunnen gefactoriseerd worden hangt af van de oproepconventies: indien de locatie in het stapelvenster voor de achteraf te bewaren registers vastligt t.o.v. de stapelwijzer of de stapelvensterwijzer en bv. niet afhangt van het aantal lokale veranderlijken dat op de stapel moet bewaard worden, dan zal het aantal identieke sequenties groter zijn.

Dit aantal wordt ook beïnvloed door de volgorde waarin achteraf te bewaren registers gebruikt worden door het programma. Stel dat regis-

ters r9 t.e.m. r15 achteraf te bewaren zijn. Indien de standaard oplegt dat deze registers in oplopende volgorde moeten geselecteerd worden door een registerallocator, zullen alle procedures die 1 register nodig hebben register r9 op de stapel bewaren, alle procedures die 2 registers nodig hebben zullen registers r9 en r10 bewaren, enz. Merk overigens op dat dit eigenlijk geen vereiste hoeft te zijn van de oproepconventies, maar dat het volstaat dat de vertaler op een consequente manier uit de vijver van achteraf te bewaren registers selecteert. Er is ons inziens geen reden om hieraan te twijfelen.

De belangrijkste verschillen met alle voorgaande factorisatie-algoritmen zijn:

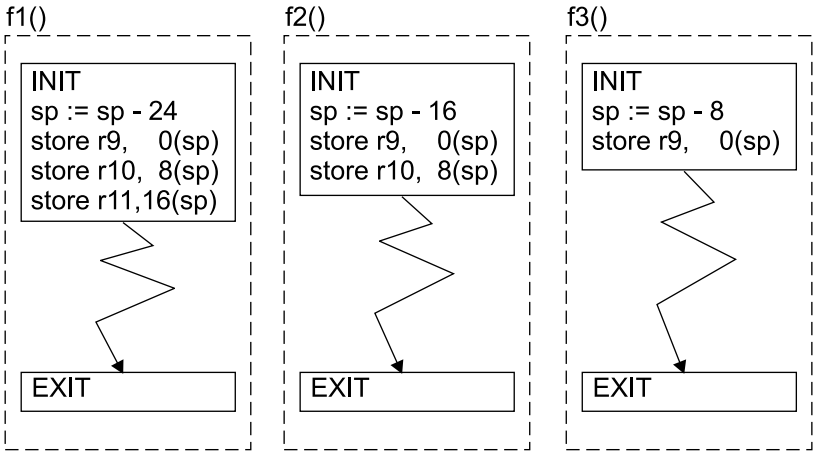
- We gaan nu op zoek naar identieke instructiesequenties los van de manier waarop ze ingeroosterd zijn. We zullen m.a.w. instructies die niet noodzakelijk op elkaar volgen in een basisblok samenbrengen en factoriseren. We kunnen ons dit permitteren omdat de locaties waar we de sequenties trachten te vinden beperkt zijn tot de ingangen en uitgangen van procedures.
- Er worden specifieke schema's voorgesteld om de gefactoriseerde code te implementeren. Dit is mogelijk omwille van het beperkte aantal variaties dat optreedt in de code die we hiermee willen factoriseren.

Het bewaren van registers op de stapel

Als we ervan kunnen uitgaan dat de achteraf te bewaren registers inderdaad steeds in dezelfde volgorde aangesproken worden, kunnen we de factorisatie uitvoeren aan de hand van procedures die als het ware in elkaar doorvallen.

Voorbeeld 4.5 Beschouw de procedures f1(), f2() en f3() uit figuur 4.8. Alhoewel de sequentie schrijfinstructies in elke procedure anders is, zullen we toch alle instructies factoriseren. Voor elke locatie in het stapelvenster moet bovendien slechts één schrijfoperatie in de gefactoriseerde code voorkomen. Daartoe werken we met een soort doorvalprocedures waarin telkens slechts 1 instructie staat. Voor de voorbeeldprocedures is dit afgebeeld in figuur 4.9. □

De vraag die zich natuurlijk stelt is welk register (in het voorbeeld was dit ra) we in zo'n geval gebruiken om het terugkeeradres in op te



Figuur 4.8: Kandidaten voor factorisatie van het wegschrijven van achteraf te bewaren registers.

slaan. Daartoe overlopen we eerst van alle procedures de ingangsblokken. Aan elk register kennen we punten toe die overeenstemmen met de bereikte compactie in aantal instructies als we dat register zouden gebruiken om het terugkeeradres op te slaan.

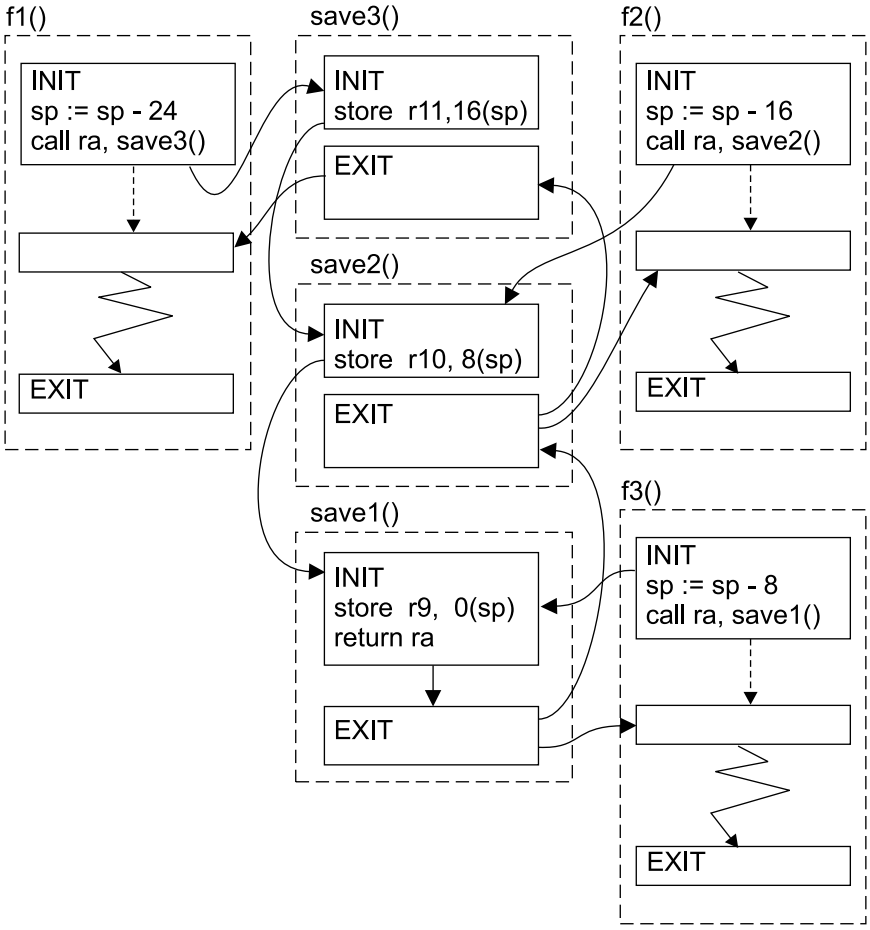
Het register dat het meeste punten gekregen heeft wordt als eerste gebruikt om de doorvalprocedures op te zetten. Daarna doen we hetzelfde voor alle procedures waarvoor de eerste poging niet slaagde omdat het gekozen register er niet vrij was, enz.

Het herstellen van de registers van op de stapel

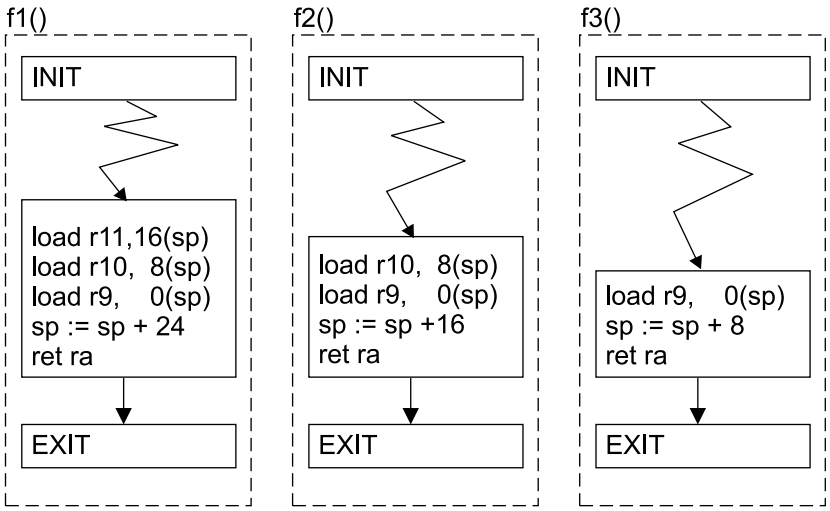
Vlak voor het terugkeren vanuit procedures worden de waarden terug van de stapel gehaald en in hun oorspronkelijke registers gezet. Ook dit komt dus veel voor, zelfs nog meer dan het wegschrijven van de achteraf te bewaren registers. Vele procedures hebben immers meer dan één terugkeerblok waarin de registerinhouden hersteld worden.

Daarbij kunnen we op volledig analoge manier werken als bij het bewaren van de registerinhouden.

Voorbeeld 4.6 Naar analogie met het voorbeeld uit de vorige sectie hebben we drie procedures afgebeeld in figuur 4.10. Het herstellen van de achteraf te bewaren registers is gefactoriseerd in figuur 4.11. □



Figuur 4.9: Gefactoriseerd wegschrijven van achteraf te bewaren registers.

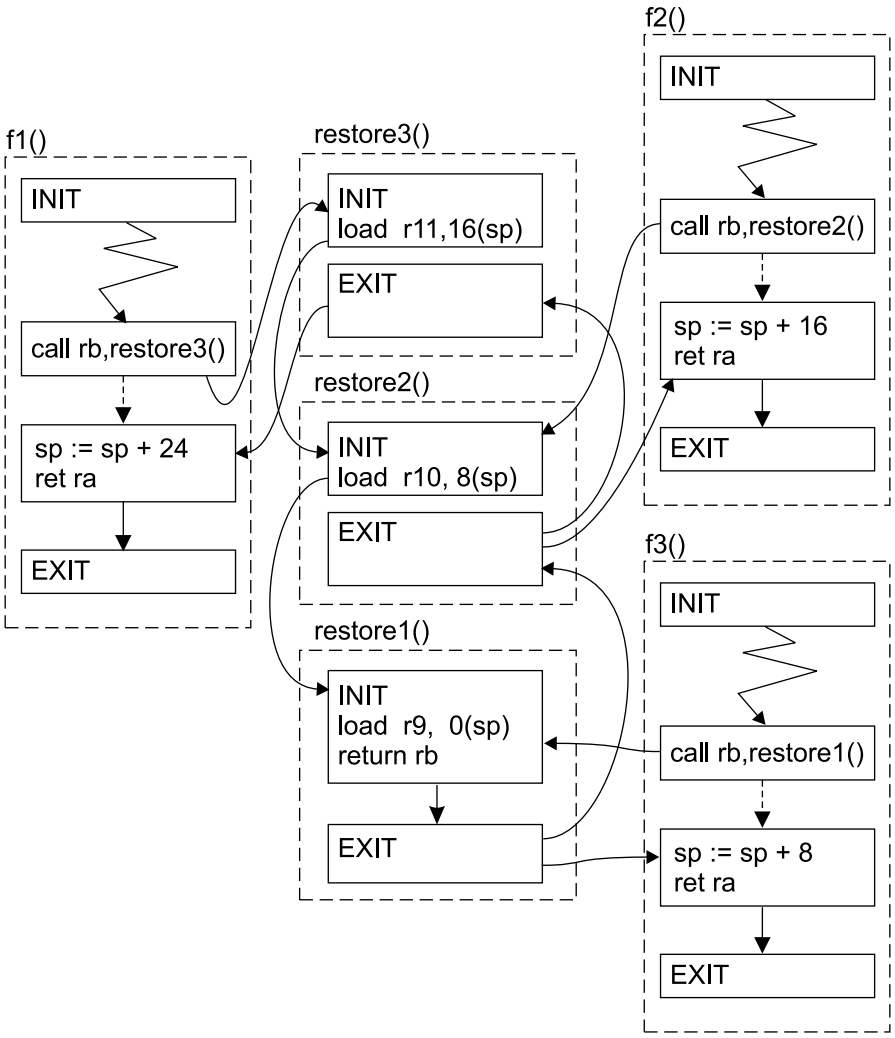


Figuur 4.10: Kandidaten voor factorisatie van het herstellen van achteraf te bewaren registers.

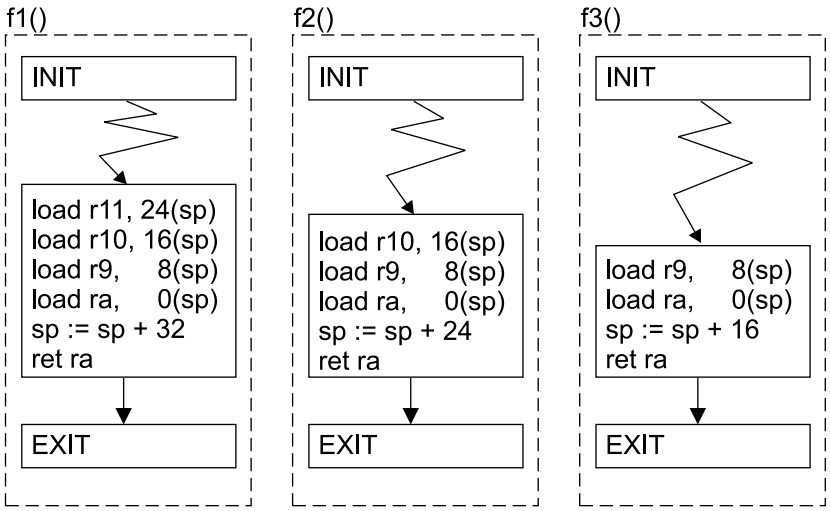
Ook nu moeten we op zoek gaan naar registers waarin we het terugkeeradres kunnen plaatsen. In het voorbeeld hebben we dit register rb genoemd. Ook deze zoektocht naar vrije registers kan analoog gebeuren aan wat beschreven werd in de vorige sectie. Over het algemeen zal het echter moeilijker zijn om vrije registers te vinden.

In veel gevallen kan men gelukkig zijn toevlucht nemen tot een andere optimalisatie. Aangezien de instructies die deze registers herstellen allemaal in terugkeerblokken zitten, zouden we een gelijkaardige optimalisatie kunnen proberen als bij regio's die eindigen op een terugkeerblok: daarvoor was het niet nodig in een extra register voor het terugkeeradres te voorzien, aangezien we daar een staartoproepoptimalisatie uitvoerden.

Het probleem hier echter is, zoals in het voorbeeld, de aanwezigheid van de optellingsinstructies die de waarde van de stapelwijzer herstellen vlak voor de terugkeer uit een procedure. Indien we deze instructie in de gefactoriseerde code willen plaatsen, zodat we effectief de staartoproepoptimalisatie kunnen uitvoeren, zullen we ze moeten parametriseren, aangezien op de grootte van de stapelvensters van procedures wel veel variatie kan zitten. Voor deze parametrisatie zullen we dus eveneens een register nodig hebben. We hebben in dat geval geen vrij register voor het terugkeeradres meer nodig, maar wel voor de grootte van het stapelvenster als parameter. Dit lijkt niet echt een



Figuur 4.11: Gefactoriseerd herstellen van achteraf te bewaren registers.



Figuur 4.12: Kandidaten voor factorisatie van het herstellen van achteraf te bewaren registers, inclusief het terugkeeradres.

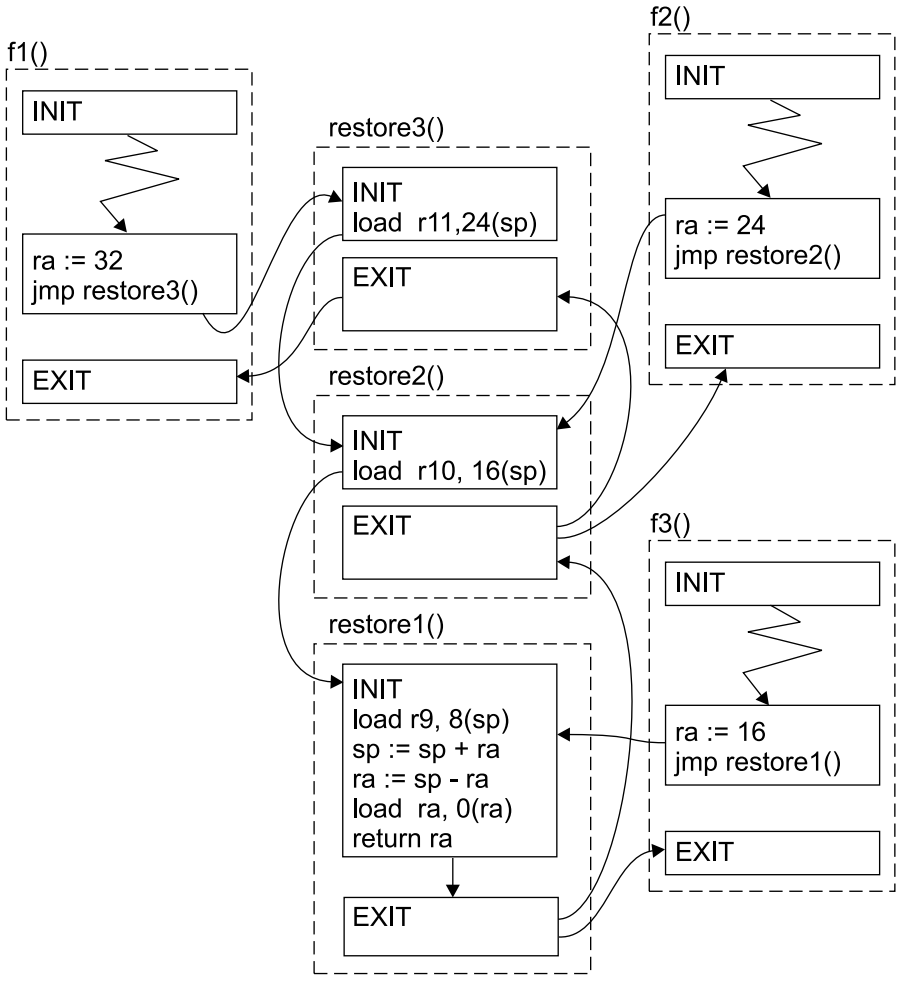
vooruitgang.

In vele gevallen zal gelukkig ook het terugkeeradres onderdeel uitmaken van de achteraf te bewaren registers. Ook het terugkeeradres wordt dus van de stapel gehaald, vlak voor de terugkeerinstructie. Tot dan kunnen we dus vrij over dit register beschikken om er de parameter in op te slaan.

Voorbeeld 4.7 In figuur 4.12 hebben we de procedures uit het vorige voorbeeld hernomen. Het enige verschil is dat hier ook het terugkeeradres van de stapel wordt geladen. Het register waarin het terugkeeradres zit kan daarom gebruikt worden tot vlak voor het terugkeeradres geladen wordt. In figuur 4.13 hebben we hier op ingenieuze wijze gebruik van gemaakt. □

Merk op dat indien het terugkeeradres door de oproepconventies aan een vast register is toegewezen of indien de vertaler hier consequent hetzelfde register voor kiest, de gefactoriseerde code niet meer afhangt van een vrij register in de procedures. We zullen dus ook slechts één instantie van de gefactoriseerde code nodig hebben, in tegenstelling tot wat we voor het bewaren van de registers moesten doen.

Tenslotte willen we nog wijzen op een laatste verfijning. Soms is het zo dat de verzameling op de stapel bewaarde registers geen verzame-



Figuur 4.13: Geoptimaliseerd gefactoriseerd herstellen van achteraf te bewaren registers.

ling is met opeenvolgende registers. Dit kan met name het geval zijn als andere programmatransformaties ervoor gezorgd hebben dat sommige registers niet meer moesten bewaard worden en de bijhorende lees- en schrijfinstructies geëlimineerd zijn. Stel bv. dat de verzameling bewaarde registers {ra,r9,r10,r12} is. In zo'n geval kan enkel het bewaren en herstellen van registers ra, r9 en r10 gefactoriseerd worden volgens de tot hier toe besproken schema's. Voor het herstellen van de registers kan men echter soms verder gaan: men kan de verzameling te herstellen registers immers uitbreiden met registers die dood zijn. Bovenstaande verzameling zou bv. uitgebreid kunnen worden met register r11 indien het dood is na de terugkeerinstructie. De te herstellen registers zijn dan opnieuw een opeenvolgende rij, en dus kunnen we alle leesinstructies factoriseren.

4.6 Lokale factorisatie

Een gemeenschappelijk kenmerk van alle tot nu toe besproken factorisatietechnieken was dat ze uitgevoerd worden op stukken code die identiek of equivalent zijn, maar voor de rest helemaal los staan van elkaar. In deze sectie bekijken we twee vormen van lokale factorisatie. Identieke of equivalente instructies of instructiesequenties in elkaars buurt worden verplaatst, zodat er slechts één instantie van nodig is.

Verplaatsen van code Soms komen er identieke instructies voor op alle paden die volgen op een bepaald blok (de dominator) of op alle paden die leiden naar een blok (de postdominator). Als we die instructies op alle paden kunnen verplaatsen tot net na of voor dat blok, kunnen we ze eveneens vervangen door één instructie aan het einde of het begin van dat blok. Of instructies kunnen verplaatst worden hangt af van de data-afhankelijkheden tussen de te verplaatsen instructies en de instructies waarover ze verplaatst moeten worden.

Zowel het vinden van identieke instructies als het verplaatsbaar maken van instructies kan bevorderd worden door registerhernoeming, aangezien hierdoor anti-afhankelijkheden kunnen vermeden worden. Dit kan op tijdsefficiënte wijze gebeuren door bv. alleen maar lokale hernoemingen te proberen, m.a.w. hernoemingen waarbij geen registers hernoemd worden die levend zijn over basisblokgrenzen heen.

Samenvoegen van identieke staarten Soms eindigen twee blokken op een identieke instructiesequentie, maar hebben ze geen unieke opvolger naar waar de instructies kunnen verplaatst worden. Indien de opvolgers van beide blokken dan toch gelijk zijn, kan men de identieke staarten van de blokken samen in één apart blok onderbrengen, dat eveneens diezelfde opvolgers heeft. Dit nieuwe blok wordt daarbij de unieke opvolger van de oorspronkelijke blokken.

Ook hier kan men de resultaten verbeteren door lokale registerhernoeming toe te passen.

4.7 Hergebruik van data

Net zoals men code kan hergebruiken op verschillende plaatsen, kan men dit ook met statisch gealloceerde data doen. Uiteraard is dit beperkt tot constante data. We bespreken drie manieren om data te hergebruiken en aldus tot extra compactie te komen.

4.7.1 Compactie van de globale adrestabel

In voorbeeld 2.1 hebben we gezien hoe in de globale adrestabel van een objectbestand in plaats voorzien wordt om een adres van een extern gedefinieerd globaal object op te slaan. Als een globaal object dus vanuit verschillende bronbestanden (of bibliotheekbestanden) aangesproken wordt, zal het adres van dat object meermaals in de globale adrestabel voorkomen.

Omdat we na het linken exact de lay-out van data in het geheugen kennen, en de adressen op de gereserveerde locaties ingeschreven zijn, kunnen we dubbels uit de globale adrestabel verwijderen.

Dit gebeurt eenvoudigweg door alle verwijzingen naar verschillende locaties die hetzelfde adres bevatten om te zetten in verwijzingen naar één van die locaties, en de globale adrestabel te herordenen, zodanig dat alle levende data erin gegroepeerd wordt.

Voor een aantal grotere applicaties heeft dit als gevolg dat de globale adrestabel klein genoeg wordt om slechts één globale wijzer nodig te hebben, waar er vóór deze vorm van compactie verscheidene wijzers nodig waren. Daardoor kan dus ook het zetten van de globale wijzer na intermodulaire sprongen geëlimineerd worden.

Let wel, men moet op voorhand testen of men het met één of meer

globale wijzers zal moeten doen. Indien het immers met meer wijzers moet, moet men ervoor zorgen dat alle code horend uit een module dezelfde wijzer blijft gebruiken. Na intramodulaire sprongen heeft de vertaler immers geen aanpassing van de globale wijzer gegenereerd, en we moeten dus vermijden dat er over zulke sprongen heen plots wel een andere wijzer zou moeten gebruikt worden.

Merk op dat we, gebruik makend van de relocatiegegevens, deze vorm van compactie kunnen uitvoeren vóór elke andere analyse van het programma dat we wensen te compacteren.

4.7.2 Hergebruik van datablokken

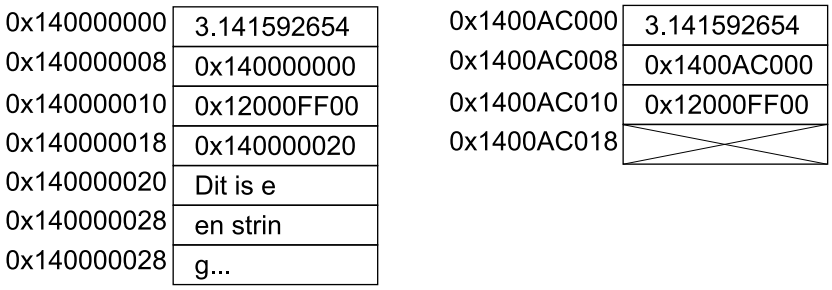
Adressen in de globale adrestabel die er meermaals in voorkomen zoals in de vorige sectie besproken is, kunnen als een speciaal geval van een algemenere datacompactie gezien worden: als volledige (constante) datablokken identiek zijn, kunnen we eveneens alle verwijzingen naar elk van de blokken naar één van de blokken laten wijzen, om zo volledige blokken dood te maken.

De blokken hoeven zelfs niet volledig identiek te zijn. Als er locaties dood zijn in een bepaald blok, speelt het geen rol wat er op de overeenkomstige locaties binnen een ander blok staat, en of die corresponderende locatie ook al dan niet dood is. Bovendien, als er in die blokken adressen staan die ergens in het eigen blok wijzen, hoeven die alleen relatief gelijk te zijn. Ze moeten m.a.w. naar dezelfde positie t.o.v. het begin van het blok wijzen. Het hoeft zelf niet zo te zijn dat blokken dezelfde grootte hebben, men kan perfect verwijzingen naar kleinere blokken vervangen door verwijzingen naar grotere blokken.

Voorbeeld 4.8 Beschouw de datablokken die weergegeven zijn in figuur 4.14. Het is evident dat alle verwijzingen naar locaties in het rechter blok kunnen gewijzigd worden in verwijzingen naar dezelfde locatie in het linker blok, en dit ondanks het verschil in data in de blokken en de verschillende grootte. □

Merk overigens op dat deze vorm van compactie iteratief is: door verwijzingen naar blokken aan te passen in andere blokken, kunnen deze laatste eveneens identiek worden.

We hebben vastgesteld dat deze vorm van compactie geen goede resultaten oplevert. Als men programma's bekijkt is het duidelijk dat er



Figuur 4.14: Kandidaten voor het hergebruik van datablokken.

zich mogelijkheden voordoen om de compactie toe te passen. Dit is met name het geval voor datablokken die komen uit bronbestanden waarin enkel een sjabloonklasse geïnstantieerd wordt. Als men dit in een programma verschillende keren doet voor dezelfde sjabloonklassen, durven de datablokken al eens heel gelijklopend zijn. Het probleem daar zit in het geharrewar van wederzijdse verwijzingen in de datablokken komende van één bronbestand. Als men deze verwijzingen als pijlen in een graaf zou beschouwen, met datablokken als knopen, dan blijken deze grafen vaak cyclisch te zijn. Men vindt dus geen initieel identiek paar blokken om het hergebruik in gang te zetten. In veel andere gevallen komen er wijzers voor naar lichtjes verschillende procedures horende bij de verschillende instanties. Omdat we deze nog niet kunnen parametriseren, blijven deze wijzers naar procedures een rem op het factoriseren van datablokken.

4.7.3 Hergebruik van individuele data

Levert het trachten te factoriseren van gehele datablokken weinig tot niks op, dan kunnen we natuurlijk nog altijd trachten onderdelen van blokken te hergebruiken.

Een mogelijke manier is het aanpassen van leesoperaties waarvan men weet wat ze inlezen, maar die men niet kan elimineren, om welke reden dan ook (bv. omdat ze code-adressen of vlottendekommagetallen opladen). Indien verschillende instructies dezelfde data inlezen, maar vanop verschillende adressen, trachten we de instructies zo aan te passen dat ze alle vanop hetzelfde adres lezen. Dit heeft twee mogelijke voordelen:

- De verschillende locaties waarvan oorspronkelijk gelezen werd,

kunnen misschien dood worden en zo tot een betere datacompactie leiden.

- Stukken code die bijna identiek zijn en identieke berekeningen uitvoeren op identieke data, maar die data vanop andere geheugenlocaties opladen, kunnen hierdoor identiek worden, wat de codecompactie ten goede komt.

4.8 Factorisatie versus snelheidsoptimalisatie

Terwijl de programmatransformaties uit hoofdstukken 2, 3 en 4 vooral optimalisaties waren, die gemiddeld gezien zowel de uitvoeringstijd als de programmagrootte ten goede komen, is dit bij factorisatie niet het geval.

Met name bij codefactorisatie van basisblokken en instructiesequenties, wordt er nogal wat extra controleverloop toegevoegd aan de programma's. Men kan trachten de negatieve invloed hiervan op de uitvoeringstijd van programma's te verminderen zonder compactie door factorisatie op dat niveau van granulariteit helemaal uit te sluiten. Hiertoe kan men bv. deze vormen van factorisatie enkel toepassen op basisblokken die niet heet zijn, d.w.z. die niet heel frequent uitgevoerd worden. Daartoe moet men dan natuurlijk wel over de nodige profielinformatie beschikken.

In hoofdstuk 5 zal nagegaan worden in hoeverre men hier eventueel een gulden middenweg in moet en/of kan zoeken.

4.9 Verwant werk

Allereerst zullen we het hebben over enkele belangrijke technieken om te vermijden dat er op grote schaal codeduplicaten in een programma voorkomen. Daarna gaan we in op factorisatietechnieken zoals die door anderen geïntroduceerd werden.

4.9.1 Vermijden van codeduplicatie

Factorisatie van programmastructuren zoals procedures is nuttig als er om de een of andere reden identieke procedures in een programma terechtkomen. Parametrisatie van procedures zou dan weer nuttig kun-

nen zijn als er nagenoeg identieke procedures in een programma voorkomen.

De kans dat dit gebeurt is met name groot als er gewerkt wordt met sjablonen, zoals in C++ of andere object-georiënteerde talen. Men kan stellen dat hergebruik van code op het bronniveau tot hergebruik van code op het assemblerniveau aanleiding geeft. Voor zulke gevallen probeert men eerst en vooral te vermijden dat identieke instanties van procedures meermaals in een programma terecht komen.

Al in de tachtiger jaren werd er veelvuldig onderzoek gedaan naar codegrootte-efficiënte implementaties voor sjablonen in de Ada programmeertaal [Rose83, Bray84]. Meestal neemt men er zijn toevlucht tot het genereren van geparametriseerde code, waarbij de types van de objecten waarvoor klassen geïnstantieerd worden dus ook in de assemblercode als parameter blijven bestaan.

Een probleem waarmee talen als C++ geconfronteerd worden is het volgende: stel dat er in twee bronbestanden een identieke instantie van een bepaald sjabloon gebruikt wordt. Als de twee bronbestanden apart vertaald worden, zal het sjabloon twee keer op identieke wijze geïnstantieerd en geïmplementeerd worden in objectcode. Het komt er dan op aan slechts één van de twee implementaties mee te linken met het programma. Hiervoor bestaan verschillende technieken.

Een ervan is vertaling met terugkoppeling van de linker. Initieel genereert men dan geen code voor sjablonen. Bij het linken van het programma zal dit foutmeldingen opleveren, omdat naar bepaalde methodes en procedures gerefereerd wordt zonder dat er een definitie wordt voor gevonden. Deze foutmelding wordt teruggekoppeld naar de vertaler, die objecten waarnaar gerefereerd werd zal aanmaken door nu wel de nodige sjablonen te instantiëren en te implementeren. Dit proces gaat door tot alle gerefereerde symbolen gedefinieerd zijn. Merk op dat dit enkel vermijdt dat identieke instanties van sjablonen op hoog niveau vertaald zouden worden in verscheidene instanties op laag niveau. Dit vangt bv. niet de triviale variaties op waarbij datatypes enkel een nieuwe naam krijgen maar voor het overige identiek zijn: de gerefereerde symbolen zijn dan verschillend, en dus wordt de codeduplicatie niet opgevangen.

De GNU C++ vertalers en linkers werken dan weer op een andere manier samen: de vertaler markeert elk mogelijk stuk redundante code, bv. door ze in aparte secties te stoppen waarvan de naam met .gnu.linkonce.d. begint. De GNU linker zal aan de hand van die naam

op zoek gaan naar identieke secties en er dan slechts één instantie van overhouden in het finale programma. Een andere manier om hiervoor te zorgen is het gebruik van een zogenaamde "repository". Dit is een tabel, waarin alle instanties van sjablonen opgeslagen worden die nodig zijn voor een volledig programma. Door ervoor te zorgen dat elke instantie slechts eenmaal in de tabel voorkomt, vermijdt men duplicaten. Men moet dan natuurlijk alle broncode tegelijkertijd ter beschikking hebben.

Alle tot hiertoe besproken technieken gaan op basis van datatypes en namen op zoek naar identieke stukken code om ze te vermijden. De technieken die wij besproken hebben in sectie 4.1 zoeken naar identieke procedures door de code erin te vergelijken op een laag niveau. Dit zorgt ervoor dat verschillen in code die enkel op het broncodeniveau zichtbaar zijn onze technieken niet remmen. Volgens [Levi00] gaan ook sommige linkers voor Microsoft Windows op zoek naar functionele gelijkenissen op het objectniveau. Ook de linker uit de ontwikkelongeving [TM] voor de Philips TriMedia doen dit.

4.9.2 Factorisatie van objectcode

In de al wat oudere literatuur over de identificatie van identieke stukken objectcode beschouwt men een programma als een lineaire sequentie van instructies. Fraser e.a. [Fras84] bouwen een zogenaamde suffixboom op om terugkerende instructiesequenties te identificeren. Terugkerende instructiesequenties worden gefactoriseerd in aparte procedures. Daarmee konden ze een aantal hulpprogramma's voor een VAX systeem met gemiddeld 7% verkleinen. Door gebruik te maken van suffixbomen, kunnen alleen identieke instructiesequenties gedetecteerd worden.

Daaraan werd eerder al op verschillende manieren tegemoetgekomen door tot op zekere hoogte variaties in de vergeleken stukken code toe te staan. Baker [Bake93] doet dit door geparametriseerde suffixbomen te beschouwen, Cooper en McIntosh [Coop99] door registerhernoeming toe te passen en Zastre [Zast95] doet het middels parametrisatie van gefactoriseerde code.

Cooper en McIntosh gebruiken evenwel een andere techniek om registerhernoeming toe te passen: i.p.v kopieeroperaties toe te voegen zoals wij doen, proberen ze registers globaal (dus over basisblokgrenzen heen) te hernoemen. Ze bereiken er een compactie van gemiddeld

5% mee. Hun techniek heeft als voordeel dat er geen kopieeroperaties moeten ingevoegd worden, maar als nadeel dat een globale registerhernoeming voor de factorisatie van één basisblok kan interfereren met mogelijke registerhernoemingen voor blokken in de buurt.

We geloven dat met de eliminatie van kopieeroperaties na factorisatie een gelijkaardig resultaat kan gehaald worden, zonder geconfronteerd te worden met de nadelen van een globale registerhernoeming. Er is immers een grote overeenkomst tussen het zoeken naar mogelijke globale registerhernoemingen en het zoeken naar mogelijkheden om kopieeroperaties te elimineren. Indien men in het ene ver raakt door complexe zoekalgoritmen te gebruiken, zal dit ook voor het andere kunnen.

Zastre bereikt een compactie van gemiddeld 2.7% door in de suffixbomen geen rekening te houden met verschillen in de operandi. Op die manier worden kandidaten gezocht voor procedurale abstractie, waarbij de operaties identiek zijn en de verschillen in operandi door middel van parameters opgevangen worden.

In [Muth99] en [Debr00] worden de factorisatie van regio's, basisblokken en instructiesequenties besproken. Het werk dat in dit hoofdstuk gepresenteerd werd overlapt daar in grote mate mee (voor die granulariteiten). Het belangrijkste verschil zit in de hernoeming van registers voor basisblokken. Het algoritme dat in [Debr00] besproken wordt, gaat in één fase na welke kopieeroperaties moeten toegevoegd worden om een blok te hernoemen naar een ander blok en of beide blokken dezelfde GAAG hebben. Daardoor worden de interne afhankelijkheden en de externe afhankelijkheden op dezelfde manier behandeld, en zullen blokken zoals in voorbeeld 5.3 nooit naar elkaar hernoemd worden.

4.9.3 Factorisatie van andere coderepresentaties

Baker [Bake93] past een gelijkaardige parametrisatie toe als Zastre door geparametriseerde suffixbomen te beschouwen op de intermediaire representatie vlak na het inlezen en ontleden van de broncode in de C taal.

Andere manieren om een programma te compacteren door abstractie van veel voorkomende instructiesequenties is het gebruik van macro's of superoperatoren [Proe95]. Men zal dan nieuwe "basisoperaties" aan een bestaande set van operaties toevoegen, waarbij de nieuwe superoperaties een veelvoorkomende sequentie vervangen.

Omdat men natuurlijk niet zomaar de instructieset van een architectuur kan aanpassen aan de veelvoorkomende instructiesequenties uit een bepaald programma, moet men hierbij zijn toevlucht nemen tot andere uitvoeringswijzen dan directe uitvoering op een processor.

In [Proe95] en [Fras95] wordt het idee van superoperatoren uitgewerkt, waarbij voor een programma vertaald wordt naar een nieuw tweedelig programma voor een bepaalde computerarchitectuur. Het eerste deel van het nieuwe programma is de vertaling van het originele programma naar een instructieset die zo opgebouwd is dat de voorstelling van het programma zo klein mogelijk wordt. Om die instructieset op te bouwen past men een techniek toe op syntaxbomen die sterk vergelijkbaar is met procedurele abstractie: men gaat er op zoek naar veel voorkomende subbomen en zal daar dan een aparte instructie voor toevoegen in de instructieset.

Het tweede deel van het programma bestaat dan uit een vertolker die het eerste deel kan vertolken en die kan uitgevoerd worden op de doelarchitectuur. Ze rapporteren hiermee een compressiefactor van ongeveer 0.50 t.o.v. gewoon vertaalde programma's voor de SPARC architectuur. Over de uitvoeringssnelheid van de gecomprimeerde programma's zwijgen ze. Omdat de vertolker die ze genereren in de C taal gegenereerd wordt, is deze techniek uiterst makkelijk overdraagbaar naar andere architecturen.

Hoogerbrugge e.a. [Hoog99] gebruiken een soortgelijke techniek om code voor de TriMedia architectuur [Tri00] van Philips te comprimeren. Enkel de minder vaak uitgevoerde code wordt in een specifieke instructieset vertaald en dan vertolkt tijdens de uitvoering van het programma, om zo geen al te groot snelheidsverlies te boeken. Om een zo compact mogelijk programma te verkrijgen maken ze gebruik van een stapelarchitectuur, waarvoor in de instructies minder operandi moeten gecodeerd worden. In tegenstelling tot [Fras95] wordt de specifieke instructieset opgebouwd na een training aan de hand van een verzameling trainingsapplicaties. Uit de beslissingsbomen die gebruikt worden als interne representatie voor programma's, worden de zogenaamde superinstructies gehaald die vaak voorkomende sequenties in de programma's vervangen. Met deze techniek werd een compressie met ongeveer een factor 5 behaald, terwijl de programma's gemiddeld ongeveer 8 keer trager uitvoeren.

Clausen e.a. [Clau00] passen de JVM aan (Java Virtuele Machine), zodat ze applicatiespecifieke macro's kan decoderen en uitvoeren. De

macro's worden zodanig geselecteerd dat ze de vaak voorkomende bytecodesequenties in een programma kunnen vervangen. Voor het programma opgestart wordt, leest de JVM de definities van de nieuwe macro's in. Hiermee rapporteren ze een reductie van de codegrootte met gemiddeld 15%.

Hoofdstuk 5

Evaluatie

"Alles is in het getal."
Pythagoras

"The proof of the pudding is in the eating."
Te mooi om vertaald te worden Engels gezegde

In dit hoofdstuk worden de algoritmen geëvalueerd die in de vorige hoofdstukken beschreven zijn. Daartoe hebben we ze geïmplementeerd in een prototype compactor. We zullen eerst beschrijven hoe we dit gedaan hebben. Daarbij gaat de aandacht voornamelijk naar de wijze waarop we de verschillende analyses, optimalisaties, factorisaties en verfijningen aan de ICVG optimaal met elkaar kunnen laten interageren.

Daarna wordt de verzameling programma's beschreven die we gebruikt hebben om de compactie te evalueren. We beschrijven ze kort en geven er een aantal karakteristieken van.

In het derde en belangrijkste deel van dit hoofdstuk worden de verschillende algoritmen numeriek geëvalueerd. De hoofdbrok bestaat uit de studie van de bijdrage van de verschillende algoritmen aan de bereikte compactiefactoren en aan de uitvoeringstijd van de compactor. We bestuderen ook, zij het in beperktere mate, hun invloed op de uitvoeringssnelheid van de gecompacteerde programma's.

5.1 SQUEEZE: een prototype compactor

Aan de hand van een prototype compactor worden de algoritmen uit dit proefschrift geëvalueerd. We beschrijven daarom eerst hoe we het prototype opgebouwd hebben en er de verschillende algoritmen in geïmplementeerd hebben.

5.1.1 Afhankelijkheden tussen verschillende algoritmen

Opdat we tot een implementatie van de beschreven algoritmen zouden kunnen komen, is het noodzakelijk eerst stil te staan bij de afhankelijkheden tussen de besproken analyses, optimalisaties, factorisaties en verfijningen aan de ICVG. Hoe beïnvloeden ze m.a.w. elkaars bijdragen tot het eindresultaat? We bepreken daarom eerst kort deze afhankelijkheden.

Verfijningen aan de ICVG

- *Omzetten indirecte procedure-oproepen* - sectie 2.5.2
  De belangrijkste analyse waarop deze verfijning steunt is constantenpropagatie. Immers, pas als duidelijk is dat een indirecte sprong een constant bestemmingsadres heeft zal hij omgezet worden in een directe sprong.

- *Onbekende oproepcontexten* - sectie 2.5.3
  Hiervoor is de belangrijkste analyse de detectie van dode data. Het is immers pas als code-adressen in de statisch gealloceerde data dood worden dat er pijlen vanuit de helleprocedures zullen verwijderd kunnen worden. Daarnaast hebben we de analyse van het stapelgedrag nodig om uit te maken of de levensduuranalyse gebaat zou kunnen zijn bij het behouden van een oproep vanuit de oproepershelleknoop.

- *Verfijnen indirecte sprongen* - sectie 2.5.4
  Deze verfijning steunt eigenlijk alleen maar op constantenpropagatie. We moeten voor "switch"-achtige structuren immers het constante adres kennen waar de adrestabel opgeslagen ligt.

- *De procedure* `exit()` - sectie 2.5.5
  Deze analyse is afhankelijk van de omzetting van indirecte procedure-oproepen naar directe oproepen. Het is immers daarmee dat

we achterhalen dat exit()-achtige procedures opgeroepen worden vanuit oproepblokken.

Analyse

- *Analyse van het stapelgedrag* - sectie 3.1.4
  Uiteraard is deze analyse gebaat bij een zo groot mogelijke verfijning van de ICVG. De analyse van het stapelgedrag wordt moeilijker na proceduresubstitutie omdat het na substitutie vaak zo is dat de stapelwijzer niet enkel meer in ingangs- en terugkeerknopen gemanipuleerd wordt.

- *Levensduuranalyse* - sectie 3.1
  Uiteraard is ook deze analyse gebaat bij een zo groot mogelijke verfijning van de ICVG. Levensduuranalyse hangt verder nauw samen met de analyse van het stapelgedrag. De resultaten van de analyse hangen evenzeer af van alle transformaties die het gebruik van registers veranderen.

- *Constantenpropagatie* - sectie 3.2
  Ook deze analyse is gebaat bij een zo groot mogelijke verfijning van de ICVG.

  Net als levensduuranalyse wordt constantenpropagatie moeilijker na proceduresubstitutie, omwille van het moeilijker analyseren van het stapelgedrag. Substitutie kan echter ook de contextgevoelige constantenpropagatie ten goede komen. De contextgevoelige constantenpropagatie heeft slechts diepte 1. Waar bij een procedure-oproep deze diepte meteen bereikt wordt, is dit niet meer het geval bij een gesubstitueerde procedure.

  Omdat de constantenpropagatie gebruik maakt van a priori kennis over statisch gealloceerde geheugenlocaties waarvan enkel gelezen kan worden en deze kennis door de analyse van het datagebruik kan uitgebreid worden, kunnen de resultaten van de constantenpropagatie ook verbeterd worden door ze na de analyse van het datagebruik uit te voeren.

  Na factorisatie is het zo dat daar waar oorspronkelijk verschillende stukken code gebruikt werden in verschillende contexten, die contexten nu hetzelfde stuk code gebruiken, namelijk de gefactoriseerde code. De kans dat gepropageerde constanten met

elkaar botsen is dus groter en er zullen minder constanten gevonden worden.

- *Analyse van datagebruik* - sectie 3.3
  Deze analyse bestaat uit twee delen: een aangepaste constantenpropagatie, met uiteraard dezelfde afhankelijkheden, en een partiële evaluatie van adresberekeningen. Dit laatste wordt uitgevoerd voor elke instructie die een constant adres produceert. Deze partiële evaluatie is dus duidelijk afhankelijk van de constantenpropagatie.

  De uitgebreide constantenpropagatie wordt nu evenwel ook afhankelijk van de levensduuranalyse: dode waarden worden wel nog gepropageerd, maar als ze samengevoegd moeten worden tijdens de propagatie geven ze geen aanleiding tot slechtste-geval-veronderstellingen.

- *Aliasanalyse* - sectie 3.4.1
  Aliasanalyse is een vrij onafhankelijke analyse, omdat onze analyse door code-inspectie een vrij lokale analyse is. Het detecteren van lees- en schrijfinstructies die constante adressen gebruiken is natuurlijk nuttig voor de aliasanalyse.

Optimalisaties

- *Loze-code-eliminatie* - sectie 3.1.6
  Loze-code-eliminatie bouwt uiteraard voort op de levensduuranalyse.

- *Optimalisaties met constanten* - sectie 3.2.6
  Het optimaal gebruik maken van constanten aanwezig in het programma steunt natuurlijk op de detectie van constanten door de constantenpropagatie. Deze optimalisatie kan er bovendien toe leiden dat er code loos wordt.

- *Verwijderen van dode data* - sectie 3.3.6
  Dit hangt helemaal samen met de analyse van het gebruik van data. Zoals reeds gesteld heeft het verwijderen van dode code-adressen in de data een gunstige invloed op het verfijnen van de ICVG.

- *Eliminatie geheugenoperaties* - sectie 3.4.1
  Om geheugenoperaties te verwijderen, moeten we soms vrije re-

gisters vinden. Deze analyse is dus gebaat bij een goede levensduuranalyse.

- *Eliminatie kopieerinstructies* - sectie 3.4.2
  Omdat het al dan niet kunnen elimineren van kopieerinstructies in heel grote mate afhangt van de levensduur van de registers die erbij betrokken zijn, is een goede levensduuranalyse van groot belang.

  Een heel belangrijke kans voor de eliminatie van kopieerinstructies doet zich voor na het hernoemen van registers voor de factorisatie van basisblokken. Deze hernoeming vindt immers plaats door kopieerinstructies toe te voegen na en voor de hernoemde blokken.

- *Proceduresubstitutie* - sectie 3.4.3
  Om procedures te kunnen substitueren moeten we weten waar ze opgeroepen worden. Dit hangt dus in sterke mate af van de verfijningen van de ICVG. Daarnaast kunnen allerlei optimalisaties procedures zo klein maken dat we ze willen substitueren, ook al worden ze op verscheidene plaatsen opgeroepen.

- *Kijkgatoptimalisaties* - sectie 3.4.4
  Kansen voor deze optimalisaties doen zich enkel voor ten gevolge van andere wijzigingen aan het programma. Met name constantenpropagatie en de optimalisatie van het gebruik van constanten levert mogelijkheden op voor sterkte-reductie.

- *Eliminatie onbereikbare code* - sectie 3.4.5
  Deze eliminatie is voornamelijk afhankelijk van de voorgaande verfijningen en de omzetting van conditionele naar gewone sprongen, die dan weer steunt op de resultaten van de constantenpropagatie.

Factorisatie

- *Procedures* - secties 4.1 en 4.2
  Procedures kunnen (nagenoeg) identiek zijn omdat de vertaler ze zo gegenereerd heeft. Het is dan nuttig ze te factoriseren of te parametriseren alvorens andere transformaties ze gespecialiseerd hebben voor hun specifieke oproepcontexten, waardoor er extra verschillen kunnen geïntroduceerd worden.

Anderzijds zijn factorisatie en parametrisatie van procedures gebaat bij het hergebruiken van data: als voor het overige identieke procedures dezelfde data opladen van op verschillende plaatsen kunnen ze door het hergebruik van die data identiek worden. Ook andere optimalisaties zoals het elimineren van instructies en van onbereikbare paden kunnen er echter voor zorgen dat procedures (nagenoeg) identiek worden.

Voor parametrisatie is het voornamelijk van belang over een nauwkeurige ICVG te beschikken omdat de parameter niet op de stapel moet opgeslagen worden als we voor de geparametriseerde procedure kunnen aantonen dat deze niet recursief kan opgeroepen worden.

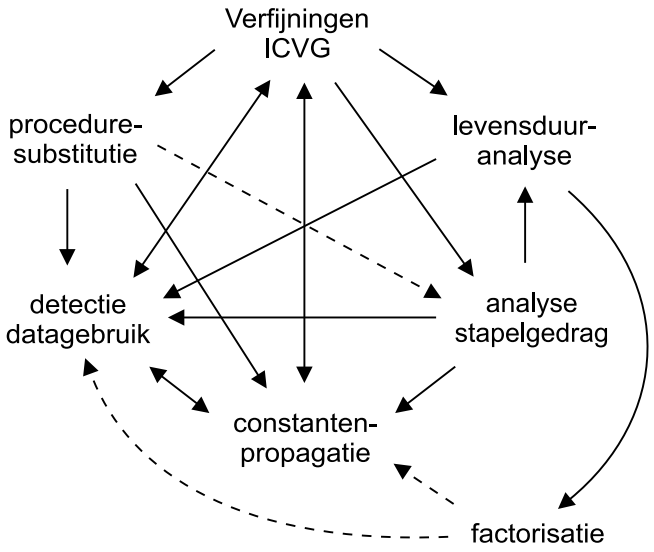
- *Regio's* - sectie 4.3
  Voor regio's is het voornamelijk van belang om over vrije registers te kunnen beschikken om het terugkeeradres in op te slaan. Dan moeten we het niet op de stapel opslaan. Een zo ver mogelijk doorgedreven voorafgaande optimalisatie van het programma aan de hand van de optimalisaties beschreven in dit proefschrift zal extra vrije registers opleveren.

- *Basisblokken* - sectie 4.4
  Voor het hernoemen van basisblokken is het eveneens belangrijk om over vrije registers te beschikken. Ook het terugkeeradres moet in een vrij adres opgeslagen worden.

- *Instructiesequenties* - sectie 4.5
  Dit geldt in mindere mate voor de factorisatie van instructiesequenties, aangezien we daarbij geen hernoeming proberen.

- *Lokale factorisatie* - sectie 4.6
  Opdat we instructies zouden kunnen verplaatsen in de ICVG is het van belang zo weinig mogelijk gehinderd te worden door afhankelijkheden tussen instructies. Sommige optimalisaties, zoals het gebruik van constanten, zorgen er voor dat er minder afhankelijkheden zijn. Andere, zoals de optimalisatie van het genereren van constanten, zorgen er weer voor dat er afhankelijkheden bijkomen.

- *Hergebruik van data* - sectie 4.7
  Opdat we statisch gealloceerde data zouden kunnen hergebruiken moeten we weten hoe de data gebruikt wordt en welke data



Figuur 5.1: De belangrijkste afhankelijkheden tussen verschillende analyses en programmatransformaties. Volle pijlen duiden op positieve afhankelijkheden: de (nauwkeuriger) toepassing van de analyse of transformatie aan het hoofd van de pijl leidt tot een betere prestatie van de analyse of transformatie aan de staart van de pijl. Gestreepte pijlen duiden op een negatieve afhankelijkheid. Na toepassing van proceduresubstitutie of factorisatie presteren andere analyses minder goed.

eventueel dood is. Dit hangt dus af van de analyse van het gebruik van data.

Het wijzigen van gegevens in de datasecties brengt met zich mee dat er datablokken identiek kunnen worden, wat meer hergebruik mogelijk maakt.

In figuur 5.1 zijn de belangrijkste afhankelijkheden grafisch weergegeven. We hebben er alle verfijningen aan de ICVG samengevoegd, net als alle factorisatietechnieken. Optimalisaties die enerzijds enkel een invloed hebben op het resultaat van analyses, zonder de nauwkeurigheid ervan te beïnvloeden, en anderzijds direct gebruik maken van de resultaten van een specifieke analyse hebben we niet opgenomen in de figuur. Men kan zien dat er heel wat cycli in de figuur zitten.

5.1.2 Een implementatie: SQUEEZE

Het prototype dat we opgebouwd hebben aan de hand van alle besproken verfijningen, analyses, optimalisaties en factorisatietechnieken heet SQUEEZE. Gelet op de afhankelijkheden die in de vorige sectie besproken werden, werkt SQUEEZE volgens het skelet in figuur 5.2.

Voor de generatie van de basisversies van de evaluatieprogramma's hebben we tevens een basisversie van SQUEEZE gebruikt. Het skelet daarvan is in figuur 5.3 weergegeven.

We bespreken nu de verschillende onderdelen en kaderen ze in de gehele werking van SQUEEZE.

Reloceren

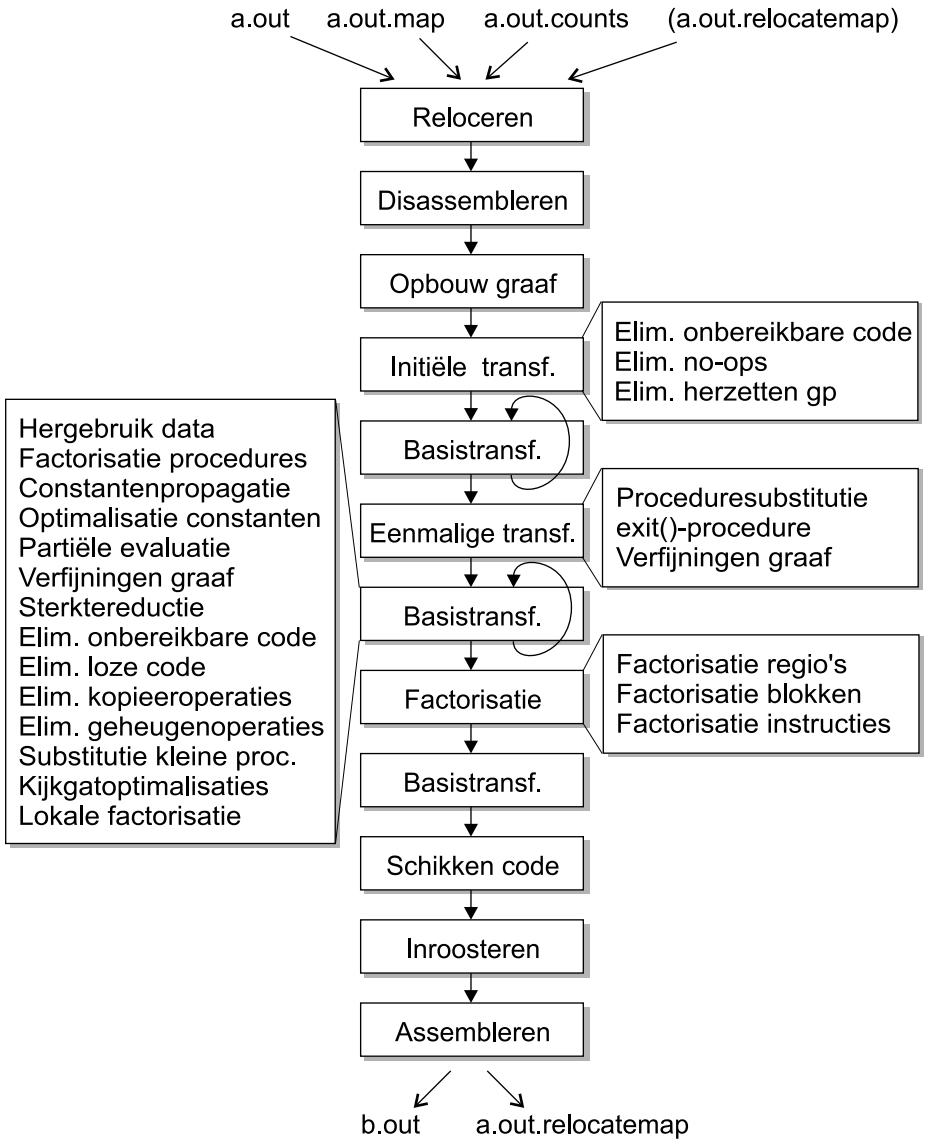
De data waarover SQUEEZE dient te beschikken om een programma te compacteren bestaat uit het statisch gelinkte programma a.out, inclusief relocatie- en symboolinformatie, en de opdeling van data in blokken. Deze data laten we door de linker produceren in a.out.map.

Aan de hand van de relocatie-informatie en a.out.map wordt de data en de code herordend, zodanig dat de code op het einde van het tekstsegment komt te staan. Hierdoor zullen de data-adressen in het programma niet meer wijzigen als er code verwijderd wordt en kunnen we data-adressen in de rest van SQUEEZE als gewone constanten beschouwen.

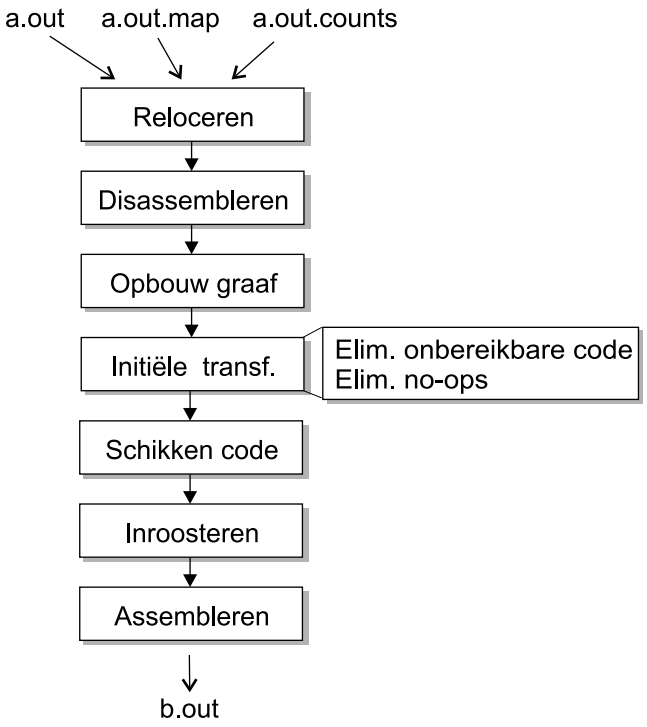
Om dezelfde reden wordt in deze fase ook de globale adrestabel reeds gecompacteerd door er dubbels uit te verwijderen. Mocht de tabel verkleinen tijdens de andere fases, dan moeten de overige datasecties verschoven worden, wat hun adressen zou wijzigen.

Voorbeeld 5.1 Beschouw de opdeling van een mogelijk programma in segmenten, secties en blokken in de linker helft van figuur 5.4. Na herordening ziet het programma eruit als in de rechter helft. De globale adrestabel (.lita) is gekrompen omdat de dubbels eruit zijn en de code staat nu op het einde van het tekstsegment, zodat de codesectie mag krimpen zonder dat er data-adressen veranderen. □

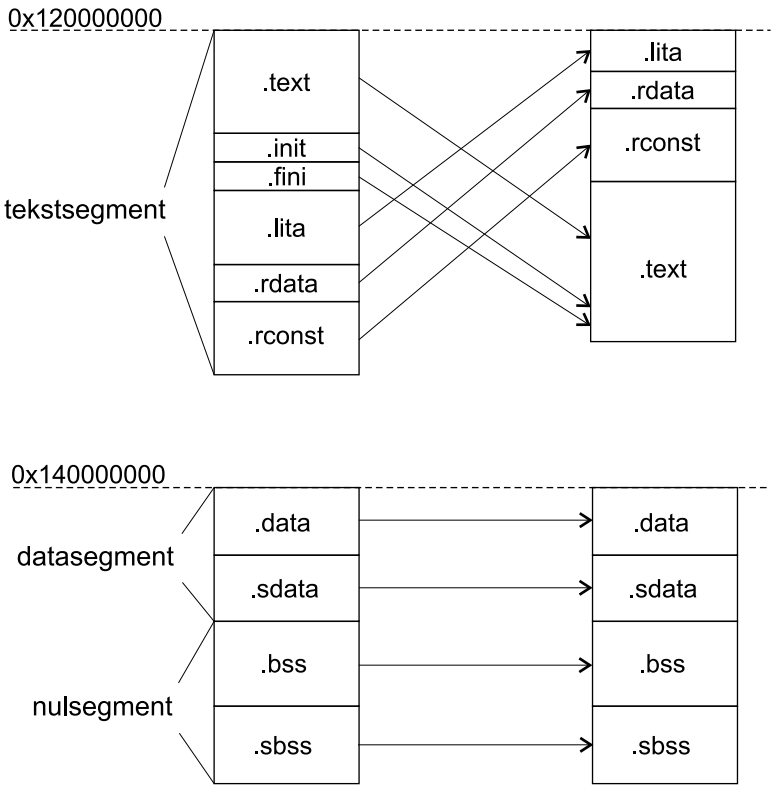
Na de eerste uitvoering van SQUEEZE wordt de informatie uit a.out.map opnieuw uitgeschreven in a.out.relocatemap, waarbij bovendien per blok wordt aangegeven of het blok dood of levend is na



Figuur 5.2: Het skelet van SQUEEZE.



Figuur 5.3: Het skelet van de basisversie van SQUEEZE.



Figuur 5.4: Relocatie na het inlezen.

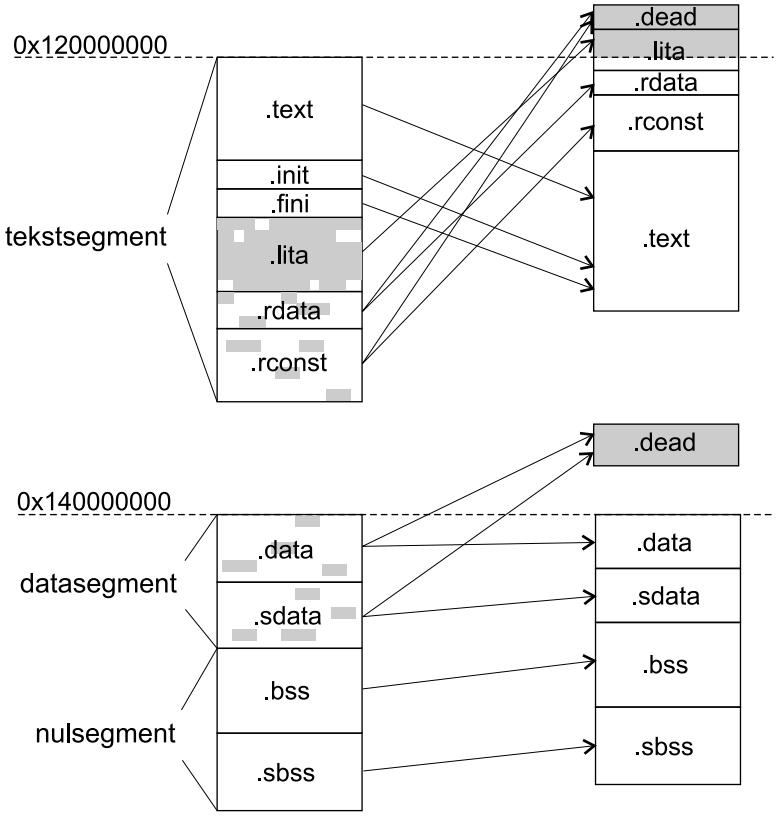
```
.rconst 120005c70 500 /usr/ccs/lib/libc.a(cvtas_pow_ten_128.o) 0
.rconst 120006170 d0 /usr/ccs/lib/libc.a(ieee.o) 0
.rconst 120006240 60 /usr/ccs/lib/libc.a(Tgettimeofdy.o) -1
.rconst 1200062a0 a0 /usr/ccs/lib/libc.a(ldr_load.o) 0
.rconst 120006340 110 /usr/ccs/lib/libc.a(localtime.o) 0
.rconst 120006450 50 /usr/ccs/lib/libc.a(asctime.o) -1

.data 140000000 8d90
.data 140000000 10 /usr/lib/cmplrs/cc/crt0.o 0
.data 140000010 30 rawdaudio.o 0
.data 140000040 1b0 adpcm.o 0
.data 1400001f0 3d0 /usr/ccs/lib/libc.a(data.o) 0
.data 1400005c0 1580 /usr/ccs/lib/libc.a(malloc.o) 0
```

Figuur 5.5: Fragment uit a.out.relocatemap.

compactie. In figuur 5.5 is een stukje uit een a.out.relocatemap bestand weergegeven bij wijze van voorbeeld. Het enige verschil met a.out.map is overigens het laatste getal op elke lijn: 0 geeft aan dat een blok levend is, -1 geeft aan dat het dood is.

Tijdens de tweede uitvoering wordt dit bestand ingelezen in de plaats van a.out.map en worden de blokken op basis van de extra informatie herordend in de relocatiefase. Merk op dat de data die dood is na de eerste uitvoering van SQUEEZE nog niet dood zal zijn bij het begin van de tweede uitvoering: het is precies door de transformaties opnieuw uit te voeren dat ze opnieuw dood wordt. Deze dood-te-worden data moet dus nog steeds in het programma opgenomen worden. We halen ze evenwel van tussen de levend-te-blijven data en groeperen ze in aparte secties. Doordat we alle dood-te-worden data nu gegroepeerd hebben, zullen we ze op eenvoudige wijze kunnen verwijderen uit het programma: na de tweede uitvoering wordt die data gewoon niet meer weggeschreven in het gecompacteerde programma. Opnieuw zullen dus geen data-adressen meer wijzigen na de relocatiefase, zodat we ze tijdens constantenpropagatie als gewone constanten kunnen beschouwen, en alle mogelijke optimalisaties op de berekeningen en het gebruik van data-adressen kunnen toepassen alsof het gewone waarden zijn.

Voorbeeld 5.2 Figuur 5.6 geeft aan hoe de data gereloceerd wordt aan



Figuur 5.6: Relocatie van dood-te-worden blokken na het inlezen.

de hand van a.out.relocatemap. De grijze blokjes zijn de dode data na de eerste uitvoering van SQUEEZE. Het is daarbij van belang de globale adrestabel als een aaneensluitend geheel van dode en levende blokken te blijven behouden, aangezien men op die wijze het kleinste aantal globale wijzers nodig heeft. Dit zijn er precies evenveel als tijdens de eerste uitvoering. □

Merk op dat de data-adressen tijdens de tweede uitvoering verschillen van die in de eerste uitvoering. Het is dus zo dat de gecompacteerde programma's na de tweede uitvoering verschillen van die na de eerste uitvoering van SQUEEZE. Hierdoor bestaat er ook een kleine kans dat data die na de eerste uitvoering dood is, tijdens de tweede uitvoering levend blijft. Men kan dit makkelijk detecteren door de geproduceerde bestanden a.out.relocatemap van de eerste en de tweede

uitvoering met elkaar te vergelijken. Eventueel kan men een derde en nog meer uitvoeringen doen tot dit proces van dode-data-detectie convergeert. Dit is voor geen van de geëvalueerde programma's vereist geweest.[1]

Omdat er eens het programma gelinkt is geen reden meer is om een onderscheid te maken tussen de verschillende codesecties worden .text, .init en .fini samengevoegd.

Disassembleren en opbouw graaf

Na het reloceren van de code en data wordt de code gedisassembleerd en wordt de initiële ICVG opgebouwd zoals in secties 2.4.3 en 2.4.3 besproken is.

Initiële transformaties

Nu we over een ICVG van het programma beschikken kunnen we er enkele triviale transformaties op toepassen. Dit zijn de eliminatie van nuloperaties (no-ops) en van initieel onbereikbare code. Dit zijn procedures en knopen die niet direct vanuit het ingangspunt van het programma of de helleknopen bereikbaar zijn.

Deze eliminaties verwijderen code die onterecht door de vertalers en linker aan het programma is toegevoegd: no-ops dienen enkel de uitvoeringssnelheid van programma's en een specifiek naar codegrootte optimaliserende vertaler zou er dan ook geen gegenereerd hebben. De onbereikbare code kon heel eenvoudig vermeden worden door bv. de bibliotheken op te splitsen in meer objectbestanden, zodat deze via directe sprongen onbereikbare code initieel niet meegelinkt wordt.

We hebben deze transformaties (samen met de compactie van de globale adrestabel, die triviaal is voor een linker) dan ook uitgevoerd om de basisprogramma's te genereren waarmee de gecompacteerde programma's vergeleken worden in de rest van dit hoofdstuk.

[1] Voor één enkel programma blijven er tijdens de tweede uitvoering items uit de globale adrestabel levend die vooraf als dood-te-worden beschouwd werden. Omdat deze enkel rechtstreeks vanaf de globale wijzer worden aangesproken kan dit op het einde van de tweede uitvoering opgelost worden door deze data te verplaatsen naar dode locaties in de levende datablokken en de leesoperaties horend bij die data aan te passen.

Verder worden onnodige berekeningen om de globale wijzer te zetten verwijderd uit het programma indien blijkt dat er slechts één globale wijzer nodig is voor het programma (na compactie van de globale adrestabel). Dit bleek bij alle gecompacteerde programma's het geval te zijn. Deze optimalisatie hebben we evenwel niet uitgevoerd om de basisversies van de evaluatieprogramma's te genereren, omdat we dit als een echte optimalisatietechniek tijdens of na het linken beschouwen. Merk bovendien op dat deze initiële verwijdering evengoed kan gebeuren na constantenpropagatie: de herberekeningen van de globale wijzer zijn dan immers idempotente berekeningen.

Basistransformaties

Zoals in sectie 5.1.1 besproken werd zijn er een heleboel analyses, verfijningen en optimalisaties die voor elkaar nieuwe mogelijkheden scheppen. We voeren er daarom iteratief een aantal van uit, zowel vóór als na enkele eenmalige transformaties die er zowel een positieve als negatieve invloed op kunnen uitoefenen. Ook na factorisatie worden ze nog eenmaal herhaald. Om welke transformaties het gaat is weergegeven in figuur 5.2. Ze omvatten o.m. alle verfijningen aan de ICVG, behalve die uit de eenmalige transformaties.

Omdat procedures soms initieel identiek zijn en na optimalisatie niet meer, of net omgekeerd, proberen we ook tijdens elke iteratie procedures te factoriseren.

Eenmalige transformaties

De belangrijkste transformaties die we slechts eenmaal hoeven uit te voeren zijn proceduresubstitutie van procedures die slechts één oproeper hebben en het verfijnen van de ICVG, met name met betrekking tot exit()-achtige procedures en door het verwijderen van oproeppijlen vanuit de oproepershelleknoop op basis van de relocatie-informatie over het gebruik van uit de globale adrestabel opgeladen adressen.

Factorisatie

Na de algemene optimalisatie van het volledige programma proberen we de resterende code te factoriseren. Daartoe beginnen we met de grootste granulariteit die ons rest, zijnde regio's. Vervolgens factorise-

ren we blokken, specifieke instructiesequenties (die te maken hebben met het bewaren en opladen van achteraf te bewaren registers) en algemene instructiesequenties.

Schikken code

Een extra optimalisatie die een hele-programma-optimalisator kan uitvoeren is het herschikken van de code op basis van profielinformatie. Deze informatie, zijnde het aantal keren dat de verschillende basisblokken uitgevoerd worden, is beschikbaar in het bestand a.out.counts. SQUEEZE kan mits het meegeven van een vlag op de commandolijn het te compacteren programma instrumenteren, zodat men de profielinformatie kan verzamelen door het programma te trainen.

Door de code in het resterende programma te herordenen kan het ophalen van instructies en het gebruik van het tussengeheugen voor instructies gevoelig verbeterd worden. Het algoritme dat we gebruikt hebben om de code te herordenen is een variant van het algoritme van Pettis en Hansen [Pett90]. Omdat dit los staat van compactie en ook kan uitgevoerd worden op niet-gecompacteerde programma's gaan we er hier niet verder op in. Het gebruikte algoritme wordt in meer detail beschreven in [Muth99]. Wel belangrijk om op te merken is dat we geen no-ops aan de code toevoegen om een beter alignement van de code te verkrijgen. Dit zou immers volledig haaks staan op het trachten te verkrijgen van zo compact mogelijke programma's.

Om een faire vergelijkingsbasis te hebben tussen basisprogramma's en gecompacteerde programma's hebben we het herschikken van de code op basis van profielinformatie ook toegepast om de basisversies van de evaluatieprogramma's te genereren.

Inroosteren

Omdat het programma na compactie sterk veranderd is (er zijn instructies geëlimineerd, vervangen en toegevoegd) is het van belang de code opnieuw in te roosteren om de gecompacteerde programma's zo snel mogelijk te laten uitvoeren. We roosteren de programma's daarom opnieuw in, waarbij we evenwel geen no-ops tussenvoegen. Het gebruikte algoritme is opnieuw van minder belang, zodat we er hier niet dieper op ingaan.

Wel van belang is het om op te merken dat we ook de basisver-

sie van de evaluatieprogramma's met dezelfde algoritmen ingeroosterd hebben. We hebben dit opnieuw gedaan om een faire vergelijkingsbasis te hebben. De vertalers waarmee de oorspronkelijke programma's gegenereerd zijn beschikken immers ongetwijfeld over betere inroosteringsalgoritmen dan de algoritmen in SQUEEZE.

Assembleren

Als de volledige code herschikt en ingeroosterd is, wordt ze tenslotte geassembleerd en samen met de overblijvende data uitgeschreven in het finale, gecompacteerde programma.

5.2 De evaluatieprogramma's

De evaluatieprogramma's die we gebruiken om de verschillende technieken te evalueren zijn in vier categorieën onder te verdelen. Tabel 5.1 bevat een beschrijving van alle evaluatieprogramma's. Tabel 5.2 bevat de vertalers en bibliotheken waarmee we de programma's vertaald en gelinkt hebben.

De vlaggen waarmee vertaald en gelinkt is zijn in tabel 5.3 opgenomen. De vlaggen -O1 en -O2 werden gekozen om de vertalers die optimalisaties te laten uitvoeren die het programma zo snel mogelijk maken zonder optimalisaties uit te voeren die de grootte van het programma doen toenemen, bv. door proceduresubstitutie of het uitvouwen van lussen. De vlag -arch ev67 zorgt ervoor dat de vertaler code genereert die specifiek geoptimaliseerd is voor ons doelplatform, dat bestaat uit het besturingssysteem Compaq Tru64 Unix 5.1 en draait op Alpha 21264 EV67 processors. De vlaggen -D_FASTMATH en -ffast-math werden meegegeven om versies van de bibliotheekprocedures zoals exp() en log() mee te linken die niet steunen op het gebruik van excepties. Hierdoor worden in geen van de evaluatieprogramma's nog excepties gegenereerd. De -r -d -z vlaggen zorgen ervoor dat de linker de relocatie- en symboolinformatie opneemt in het gelinkte programma. De vlag -m zorgt ervoor dat de linker de a.out.map bestanden genereert. De vlag -non_shared tenslotte geeft aan dat de programma's statisch moeten gelinkt worden. Daarnaast zijn per programma uiteraard een aantal vlaggen aan de vertaler meegegeven die nodig zijn om een correcte vertaling voor ons platform te verkrijgen. Het zijn voor de SPEC2000 evaluatieprogramma's bv. -DSPEC_CPU2000 en -DALPHA.

Programma	Beschrijving	Taal
	MediaBench	
adpcm	Spraakcompressie	C
epic	Beeldcompressie	C
gsm	GSM-spraaktranscodering	C
mpeg2decode	MPEG decoderen	C
mpeg2encode	MPEG coderen	C
	SPECint2000	
164.gzip	Datacompressie	C
175.vpr	Plaatsing en routering voor FPGA circuits	C
176.gcc	Gnu C vertaler	C
181.mcf	Combinatorische optimalisatie	C
186.crafty	Schaakprogramma	C
197.parser	Ontleden van Engelse tekst	C
252.eon	Beeldvisualisatie	C++
253.perlbmk	Vertolker van PERL code	C
254.gap	Vertolker van specificaties in groepentheorie	C
255.vortex	Object-georiënteerde databank	C
256.bzip2	Datacompressie	C
300.twolf	Simulator voor plaatsing en routering	C
	SPECfp2000	
168.wupwise	Quantumchromodynamica (fysica)	Fortran 77
178.galgel	Vloeistoffendynamica	Fortran 90
	C++ programma's	
blackbox	Vensterbeheerder	C++
deltablue	Oplossen incrementele dataverloopvergelijkingen	C++
fpt	Parallelliseren van Fortran programma's	C, C++
gtl	Testprogramma van de "Graph Template Library" (bewerkingen op grafen)	C++
lcom	Vertaler "L" apparatuurbeschrijvingstaal	C++
richards	Simulator besturingssystemen	C++

Tabel 5.1: De evaluatieprogramma's.

Programma	Compaq vertaler	Compaq Biblioth.	Gnu vertaler	Gnu Biblioth.
MediaBench				
adpcm	cc	c m	gcc	c m
epic	cc	c m	gcc	c m
gsm	cc	c m	gcc	c m
mpeg2decode	cc	c m	gcc	c m
mpeg2encode	cc	c m	gcc	c m
SPECint2000				
164.gzip	cc	c	gcc	c
175.vpr	cc	c m	gcc	c m
176.gcc	cc	c m	gcc	gcc c m
181.mcf	cc	c m	gcc	c m
186.crafty	cc	c m	gcc	c m
197.parser	cc	c m	gcc	c m
252.eon	cxx	cxx c exc	-	
253.perlbmk	cc	c m	-	
254.gap	cc	c m	gcc	c m
255.vortex	cc	c m	gcc	c m
256.bzip2	cc	c	gcc	c
300.twolf	cc	c m	gcc	c m
SPECfp2000				
168.wupwise	f77	for Futil m c_r	g77	g2c m c
178.galgel	f90	for Futil m c_r	-	
C++ programma's				
blackbox	cxx	cxx c exc	-	
		X11 dnet_stub Xext	-	
deltablue	cxx	cxx c exc	-	
lcom	cxx	cxx c exc	-	
fpt	cc, cxx	cxx c exc m cxxstd	-	
gtl	cxx	cxx c exc	-	
		GTL cxxstd	-	
richards	cxx	cxx c exc	-	

Tabel 5.2: De evaluatieprogramma's en de gebruikte vertalers en bibliotheken. Om de tabel enigszins compact te houden zijn de gebruikte vertalers en bibliotheken cryptisch omschreven. Zo betekent een omschrijving 'm' in de tabel dat het programma gelinkt werd met de optie '-lm'. De bibliotheek die dan meegelinkt wordt is libm.a. De enige bibliotheken van Gnu zijn libgcc.a en libg2c.a. De overige bibliotheken zijn geleverd door Compaq bij Tru64 Unix 5.1. Meer details over de vermelde vertalers vinden we in tabel 5.3.

Vertaler	Versie	Vlaggen
cc	Compaq C V6.3-025 1	-arch ev67 -O1 -D_FASTMATH
cxx	Compaq C++ V6.3-002	-arch ev67 -O1 -D_FASTMATH
f77	Compaq Fortran X5.3	-arch ev67 -O1 -D_FASTMATH
f90	Compaq Fortran X5.3	-arch ev67 -O1 -D_FASTMATH
gcc	Gnu CPP 2.95.2	-O2 -D_FASTMATH -ffast-math
g77	Gnu CPP 2.95.2 Voorkant 0.5.25	-O2 -D_FASTMATH -ffast-math
ld	Compaq ld versie 5.1	-r -d -z -m -non_shared

Tabel 5.3: De gebruikte vertalers en linker, en de vlaggen die we meegegeven hebben.

Het porteren van de Gnu vertalers naar ons doelplatform is voorlopig slechts gedeeltelijk gebeurd. Zo is er nog steeds geen Gnu assembler of Gnu linker beschikbaar voor dit platform. Om die redenen zijn de Fortran 90 en C++ programma's enkel geëvalueerd voor de Compaq vertalers. Voor 253.perlbmk genereerde gcc een fout programma, zodat we ook voor dit programma enkel de resultaten voor de C vertaler van Compaq meegeven.

5.3 Numerieke evaluatie

In deze sectie evalueren we eindelijk onze algoritmen. We zullen hierbij als volgt te werk gaan.

- Eerst bespreken we kort de belangrijkste kenmerken van de basisversies van de programma's.

- Met de basisversies vergelijken we vervolgens de door ons gekozen optimale compactie. Hiertoe zetten we in SQUEEZE alle analyses en transformaties aan. Van sommige transformaties kiezen we echter niet noodzakelijk de krachtigste variant, omdat gebleken is dat de krachtigste variant toch slechts een heel marginale extra compactie oplevert, hoewel die heel wat tijd in beslag neemt.

  We bekijken de resulterende (optimale) compactiefactoren en gaan na hoe de uitvoeringssnelheid van de programma's beïnvloed wordt door de compactie.

- Vervolgens gaan we de verschillende analyses en transformaties nauwkeuriger bekijken. Omdat ze onderling verweven zijn, zullen we dit doen door het verlies aan compactie t.o.v. de optimale compactie te beschouwen als we de analyses en transformaties niet uitvoeren of een eenvoudiger variant gebruiken. Op deze manier verschaffen we de lezer een impressie van het aandeel in de kost en het belang voor de totale compactie van de analyses.

  Men moet zich wel realiseren dat de evaluatie uitgevoerd wordt met een prototype compactor. De duurste algoritmen die geëvalueerd worden (zowel qua geheugenkost als qua uitvoeringstijd) zijn tot op enige hoogte geoptimaliseerd met het oog op het inkorten van ontluiscycli tijdens de ontwikkeling van het prototype. Ze zijn echter geenszins uitgevlooid om er elke seconde snelheidswinst uit te puren. De absolute en relatieve compactietijden die in dit hoofdstuk gegeven worden moeten dan ook met de nodige zin voor relativering beschouwd worden.

  Er is een erg goede reden om de kost en de invloed op de totale compactie van diverse technieken op een "subtractieve" manier te beschouwen. De technieken zijn vaak zo nauw verweven met elkaar en steunen in zo'n mate op elkaar dat een analyse van de compactie die ze apart bereiken nutteloos is. Bovendien is het zo dat de toepassing van een compactietechniek er voor kan zorgen dat andere technieken sneller uitgevoerd worden, en zo de totale uitvoeringstijd van de compactie afneemt. Het meest typische voorbeeld is de eliminatie van onbereikbare code. Uiteraard neemt deze optimalisatie enige tijd in beslag. Doordat de overige technieken hierdoor echter op een kleiner programma kunnen uitgevoerd worden, versnelt deze techniek de totale compactie.

- Op het einde van dit hoofdstuk proberen we tenslotte de lezer nog wat additionele inzichten in de bereikte resultaten aan te bieden, zoals bij aparte compactie van van bibliotheekcode afgescheiden applicatiecode.

De analyses en transformaties die in sectie 3.4 besproken worden zullen we niet apart evalueren. Waar ze kunnen toegepast worden hangt nauw samen met de meer fundamentele analyses. Zo hangt de eliminatie van onbereikbare code bv. volledig af van het verwijderen van pijlen uit de ICVG. Dit laatste gebeurt in de conditionele constantenpropagatie en bij de detectie van dode code-adressen in de data en

de ermee geassocieerde pijlen vanuit de helleknopen. De prestaties van onbereikbare-code-eliminatie zijn dus bij de evaluatie van die fundamentele analyses in rekening gebracht.

5.3.1 De basisversies van de evaluatieprogramma's

In tabel 5.4 wordt de grootte van de basisversies van de evaluatieprogramma's weergegeven. Voor één keer geven we de absolute cijfers. Verder zullen we enkel relatieve getallen gebruiken.

Het zijn deze basisversies die we verder gaan gebruiken voor onze evaluatie. In de cijfers voor de grootte van de statische gealloceerde data zijn de op nul geïnitialiseerde data niet opgenomen, aangezien die in het programmabestand toch geen plaats innemen. Ook de data die de procedures beschrijft (en gebruikt zou kunnen worden voor de afhandeling van excepties) is niet opgenomen in deze cijfers, aangezien we deze data niet aanpassen na compactie. Voor programma's die geen excepties genereren is deze data overigens niet van belang. Als er in het programma code opgenomen is om excepties af te handelen uit de libexc.a bibliotheek blijft deze wel in de code zitten. Uit de tabel kunnen we grofweg twee zaken afleiden:

- De programmabestanden bevatten veel meer code dan data.
- De grootte van de programma's hangt nauwelijks af van de gebruikte vertaler. Dit komt natuurlijk grotendeels doordat dezelfde bibliotheken gebruikt werden bij het linken. Enkel bij 168.wupwise zijn de verschillen zeer groot. Daar zijn dan ook andere bibliotheken gebruikt. Verder in dit hoofdstuk zullen we de opdeling tussen applicatie- en bibliotheekcode maken.

Om de uitvoeringssnelheid van de gecompacteerde programma's te evalueren maken we gebruik van de SPEC evaluatieprogramma's. Deze zijn steeds getraind met de trainingsdata en we meten de uitvoeringstijd voor de referentiedata. Om een idee te geven van de uitvoeringstijden hebben we de absolute tijden opgenomen in tabel 5.5. Voor de overige programma's hebben we geen snelheidsmetingen gedaan, voornamelijk omdat we niet over (gestandaardiseerde) data beschikken waarop een nauwkeurige snelheidsmeting kan uitgevoerd worden.

Alle snelheidsmetingen zijn uitgevoerd op een Compaq ES20 werkstation met twee 21264 EV67 processors van 667 MHz met elk 2 MB

Programma	Compaq code	Compaq data	Compaq totaal	Gnu code	Gnu data	Gnu totaal
MediaBench						
adpcm	138368	67088	205456	138496	67088	205584
epic	224000	80640	304640	224704	80960	305664
gsm	186176	86176	272352	180992	86656	267648
mpeg2decode	209344	91904	301248	209216	92176	301392
mpeg2encode	275904	100784	376688	279552	101248	380800
SPECint2000						
164.gzip	180480	78544	259024	184832	80096	264928
175.vpr	337472	125584	463056	342528	127664	470192
176.gcc	1536128	370768	1906896	1535552	360448	1896000
181.mcf	207552	72480	280032	208000	72560	280560
186.crafty	383936	127104	511040	380096	129920	510016
197.parser	297408	94432	391840	298816	93696	392512
252.eon	544832	188960	733792			
253.perlbmk	770752	201424	972176			
254.gap	706432	141968	848400	657024	136288	793312
255.vortex	658560	218656	877216	688064	226096	914160
256.bzip2	168128	75232	243360	168064	75696	243760
300.twolf	425664	115184	540848	422080	111184	533264
SPECfp2000						
168.wupwise	715200	166192	881392	249664	94224	343888
178.galgel	893568	177120	1070688			
C++ programma's						
blackbox	1101696	212560	1314256			
deltablue	237120	86816	323936			
fpt	2123008	912880	3035888			
gtl	647616	381104	1028720			
lcom	396608	177824	574432			
richards	185152	80336	265488			

Tabel 5.4: Grootte in bytes van de basisversies van de evaluatieprogramma's.

Programma	Compaq uitvoeringstijd	Gnu uitvoeringstijd
SPECint2000		
164.gzip	592	572
175.vpr	427	460
176.gcc	264	276
181.mcf	458	466
186.crafty	255	250
197.parser	781	794
252.eon	379	
253.perlbmk	426	
254.gap	518	414
255.vortex	541	592
256.bzip2	568	456
300.twolf	798	851
SPECfp2000		
168.wupwise	444	514
178.galgel	1527	

Tabel 5.5: Uitvoeringstijden in seconden van de basisversies van de evaluatieprogramma's voor de referentiedata van SPEC2000.

niveau-2-tussengeheugen. In totaal beschikt de machine over 1.5 GB RAM geheugen. Het gebruikte besturingssysteem is Compaq Tru64 Unix 5.1. Alle programma's zijn 5 keer uitgevoerd op een, op het besturingssysteem na, onbelaste machine. Na de traagste en snelste uitvoeringstijden uitgesloten te hebben, werd het rekenkundig gemiddelde van de overige drie metingen berekend.

5.3.2 Optimale compactie met SQUEEZE

In figuren 5.7, 5.8 en 5.9 zijn de bereikte compactiefactoren weergegeven voor de diverse evaluatieprogramma's gecompacteerd door SQUEEZE. Hierbij hebben we alle optimalisaties, verfijningen aan de graaf en factorisatietechnieken toegepast, uitgezonderd de parametrisering van nagenoeg identieke procedures. Omdat bij sommige analyses het resultaat nauwelijks verbeterde door het gebruik van een krachtiger analyse, hebben we een minder krachtige variant van die analyses gekozen. Dit is het geval voor constantenpropagatie, waarbij we voor de contextongevoelige propagatie geopteerd hebben en geen laat samenvoegen toepassen, en voor de loze-code-eliminatie, waarbij

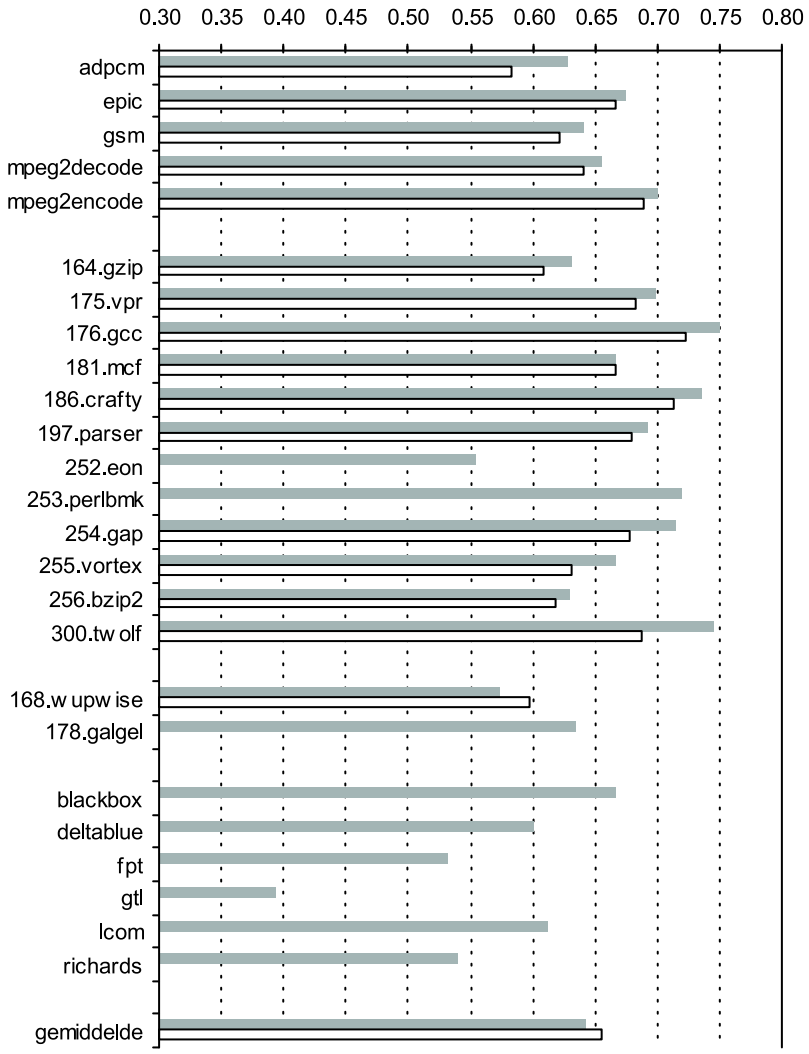
we de contextgevoelige levensduuranalyse met diepte 1 niet toegepast hebben.

Optimale compactiefactoren voor code

Wat betreft de compactiefactoren voor code kunnen we samenvattend stellen dat de (rekenkundig) gemiddelde compactiefactor voor de programma's gegenereerd met de Compaq vertalers 0.64 bedraagt. Hoe kleiner deze factor, hoe beter natuurlijk. De compactiefactoren schommelen tussen 0.39 en 0.75. Opmerkelijk daarbij is dat deze voor C++ programma's gemiddeld merkelijk beter zijn. De reden hiervoor is tweevoudig: allereerst leent C++ code zich beter tot factorisatie: vooral de specialisaties van sjabloonklassen vertonen heel veel gelijkenissen en soms zijn ze zelfs identiek: waar het type verschilt op broncodeniveau, hoeft dit niet altijd tot verschillende asseblercode aanleiding te geven. Anderzijds is het zo dat deze applicaties meer bibliotheken gebruiken. Dit is normaal, aangezien één van de belangrijkste doelstellingen van object-georiënteerde talen net het stimuleren van hergebruik van code is. Men kan verwachten dat er meer onbereikbare code in bilbiotheekcode te vinden is binnen een specifieke applicatie dan in de applicatiespecifieke code ervan.

De programma's die vertaald zijn met de Gnu vertalers kunnen gemiddeld gezien iets beter gecompacteerd worden. Dat het totale gemiddelde voor deze programma's hoger ligt dan bij de door de Compaq vertalers gegeneerde programma's valt te verklaren door het feit dat de C++ en Fortran 90 programma's niet met de Gnu vertalers vertaald zijn. Dat de Gnu versies beter gecompacteerd kunnen worden kan grotendeels toegeschreven worden aan een betere factorisatie van de programma's. De reden is dat de Gnu vertaler minder sterk lokaal optimaliseert (geselecteerde codesequenties om broncodefragmenten te implementeren worden na de selectie minder sterk gespecialiseerd binnen hun uitvoeringscontext) en er dus minder variatie in de vertaalde code voorkomt. Merk op dat dit enkel geldt voor de applicatiespecifieke code, aangezien de gebruikte bibliotheken nagenoeg niet verschillen.

Het meest opvallende resultaat is dat voor 168.wupwise. Alhoewel de Gnu basisversie iets minder dan 3 keer kleiner is dan de Compaq basisversie halen we toch een zeer vergelijkbare compactiefactor. Een eenduidige verklaring hebben we hier niet voor. Waarschijnlijk is het verschil te wijten aan het gebruik van andere bibliotheken. Het duidt



Figuur 5.7: De optimale compactiefactoren voor code. De compactiefactoren voor de met Compaq vertalers gegenereerde versies zijn met grijze balkjes weergegeven, die voor de Gnu versies met witte balkjes.

er wel op dat we ook bij het gebruik van een reeds kleinere bibliotheek, waaruit dezelfde functionaliteit gebruikt wordt, nog een fikse compactie kunnen bereiken.

Optimale compactiefactoren voor data

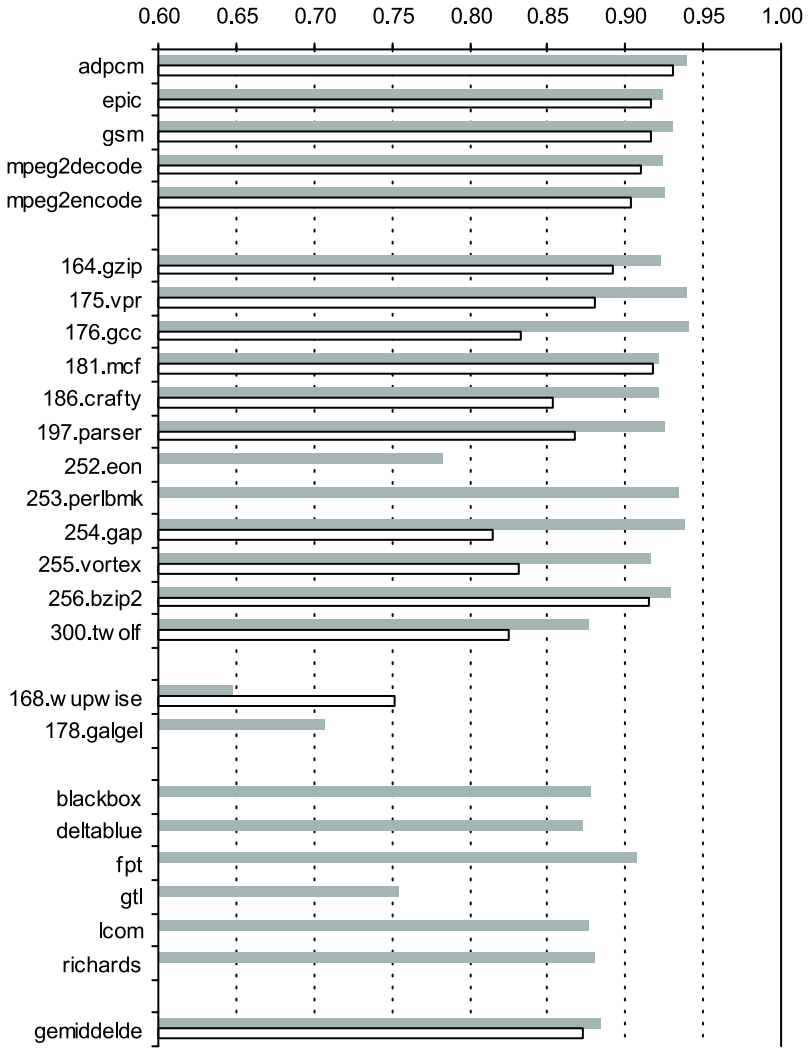
Over de compactiefactoren voor statisch gealloceerde data kunnen we samenvattend stellen dat deze voor de Compaq versies schommelen tussen 0.65 en 0.94, met een rekenkundig gemiddelde van 0.88. We kunnen dus heel wat minder data dan code uit de programma's verwijderen. Uit de Gnu versies kunnen we voor de meeste programma's ietsje meer data verwijderen uit de programma's, al is het verschil meestal veel kleiner dan de verschillen voor codecompactie.

Wat wel opvalt is dat datacompactie voor het Fortran 77 programma merkelijk beter presteert. De reden hiervoor is te zoeken in het gebrek aan dynamische geheugenallocatie: alle data wordt statisch gealloceerd, bij onbereikbare code hoort dus gemiddeld meer dode statisch gealloceerde data. Voor de Gnu versie, waarbij de code eerst van Fortran 77 naar C vertaald werd en verder met de C bibliotheken gelinkt werd, is dit al veel minder het geval, net zoals voor het Fortran 90 programma.

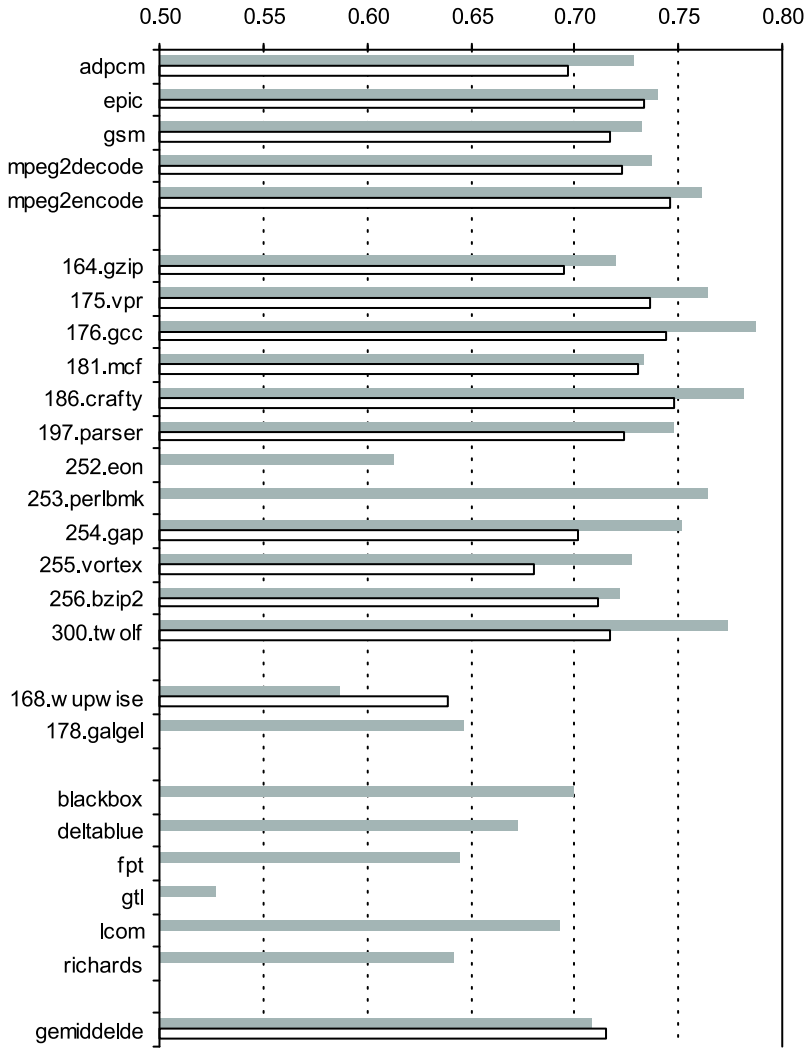
Optimale compactiefactoren voor de volledige programma's

Als we compactie van code en data samen bekijken, zien we dat we voor de Compaq versies gemiddeld een compactiefactor van ongeveer 0.71 halen. We kunnen de programma's dus gemiddeld net geen 30% kleiner maken.

De evaluatieprogramma's gtl en 252.eon zijn hiervan de meest uitgesproken illustratie. Terwijl voor de andere programma's de grootte van de dynamisch gelinkte programma's aanzienlijk kleiner is dan die van de gecompacteerde statisch gelinkte programma's, is dit bij deze programma's niet het geval. De door ons gecompacteerde versie van 252.eon is ongeveer 19% kleiner dan de dynamisch gelinkte versie van hetzelfde programma. Voor gtl loopt het verschil in het voordeel van de statisch gelinkte en gecompacteerde versie zelfs op tot 30%! Daarbij moeten we wel opmerken dat we ook bij de dynamisch gelinkte versie van gtl de GTL bibliotheek statisch meegelinkt hebben. Deze bibliotheek biedt immers zo'n specifieke functionaliteit aan dat niet



Figuur 5.8: De optimale compactiefactoren voor data.



Figuur 5.9: De optimale compactiefactoren voor de code en data samen.

te verwachten valt dat men hem dynamisch zou meelinken. De belangrijkste reden waarom deze twee gecompacteerde statisch gelinkte programma's kleiner worden dan de dynamisch gelinkte is, naast de compactie door SQUEEZE uiteraard, de aanwezigheid van dynamische symbooltabellen in de dynamisch gelinkte programma's. Deze zijn nodig om de dynamisch meegelinkte procedures en data te kunnen aanspreken.

Invloed van compactie op uitvoeringssnelheid

Kleinere programma's worden soms met een tragere uitvoering geassocieerd. Dit beeld komt voort uit optimalisatietechnieken als het uitvouwen van lussen en proceduresubstitutie. Deze technieken voegen code toe om die dan te specialiseren voor de context waarin ze uitgevoerd wordt, zijnde een bepaalde iteratie of een oproepcontext.

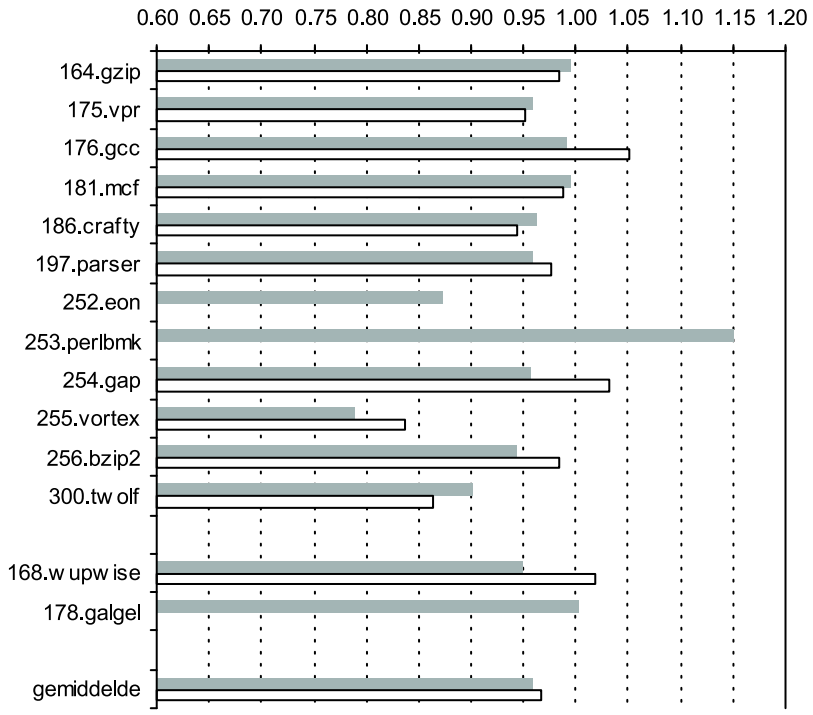
Men moet zich de vraag stellen wat er gebeurt met de uitvoeringstijd van programma's als ze gecompacteerd worden. Enerzijds verwacht men snelheidswinst van de algemene optimalisaties: deze zorgen ervoor dat er minder instructies in het programma voorkomen en er dus ook minder instructies zullen uitgevoerd worden. Anderzijds introduceert factorisatie van bv. coderegio's of basisblokken extra controleverloop in het programma, wat dan weer tot een vertraging zal leiden. Daartegenover staat dan echter weer dat het gedrag van de tussengeheugens beter kan worden door factorisatie.

Voor de SPEC evaluatieprogramma's zijn de versnellingsfactoren voor de Compaq en de Gnu versies weergegeven in figuur 5.10. Net als een compactiefactor wordt een versnellingsfactor gedefinieerd als

$$\text{versnellingsfactor} = \frac{\text{uitvoeringstijd gecompacteerde programma}}{\text{uitvoeringstijd basisversie}}$$

Zoals we kunnen vaststellen worden de programma's gemiddeld sneller door ze te compacteren, met een gemiddelde versnellingsfactor van 0.96. Deze varieert sterk tussen 0.79 en 1.15. Sommige programma's worden dus trager. Gemiddeld zijn de versnellingsfactoren voor Gnu en Compaq versies vergelijkbaar.

Men dient er rekening mee te houden dat deze versnellingsfactoren slechts indicatief zijn, omdat ze sterk door toevalligheden beïnvloed worden. Met name het niet toevoegen van no-ops aan de programma's om ze beter in te roosteren kan de uitvoeringstijden van twee nage-



Figuur 5.10: Versnellingsfactoren bij optimale compactie.

noeg identieke programma's toch sterk doen verschillen. De processor waarop de uitvoeringstijden gemeten zijn is een superscalaire processor, die vier instructies per cyclus kan ophalen en voor uitvoering naar de pijplijnen doorsturen. Voor het ophalen van de instructies moet echter aan verschillende regels voldaan worden. De belangrijkste is ongetwijfeld de alignementsvoorwaarde: de vier in één cyclus opgehaalde instructies moeten op een 16-byte grens gealigneerd zijn. Daarnaast zijn er beperkingen voor elk van de vier instructies wat betreft hun type (IO-operaties, aritmetische bewerkingen op gehele of vlottendekommagetallen, enz.). Het niet toevoegen van no-ops resulteert in een volledig toevallig alignement van de instructies, zodat afhankelijk van het beginadres van bv. de code in een binnenste lus, deze lus zeer goed of zeer slecht kan ingeroosterd zijn.

Men zou deze toevalligheden kunnen trachten te vermijden door voor de heel frequent uitgevoerde code toch no-ops toe te voegen. Op die manier zou men relatief weinig aan codecompactie moeten inboeten. We hebben dit evenwel nog niet numeriek geëvalueerd.

Uitvoeringstijden compactie

Om de bijdragen van de verschillende compactietechnieken aan de totale compactietijd te kunnen evalueren is het nuttig een beeld te hebben van de absolute tijden die SQUEEZE nodig heeft om programma's te compacteren. Ze zijn weergegeven in tabel 5.6. De cijfers in de tabel zijn de optelling van de eerste en tweede uitvoering van SQUEEZE voor elk programma.

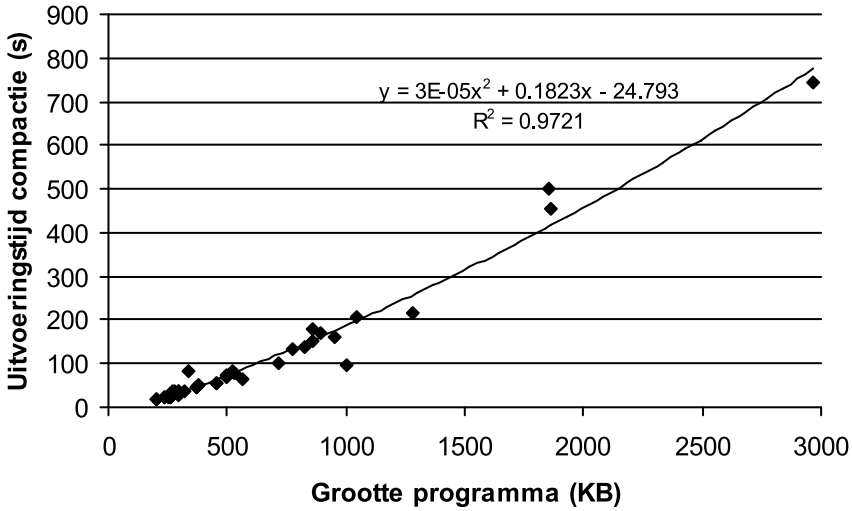
In figuur 5.11 zijn deze tijden in een grafiek geplaatst en hebben we Microsoft Excel een tweede-graads kromme laten passen aan de meetwaarden. Dat de compactietijden bij benadering een orde O(n^2) verloop hebben hoeft niet te verbazen: een aantal algoritmen, zoals constantenpropagatie, hebben orde O($P \times K$) met P het aantal pijlen in de ICVG en K het aantal knopen. Bovendien wordt het gedrag van het tussengeheugen slechter naarmate de programma's groter worden en er minder localiteit door SQUEEZE kan uitgebuit worden.

5.3.3 Bijdrage van levensduuranalyse

In deze sectie zullen we de verschillende varianten van levensduuranalyse evalueren.

Programma	Compaq compactietijd	Gnu compactietijd
MediaBench		
adpcm	19	19
epic	35	36
gsm	26	25
mpeg2decode	29	30
mpeg2encode	44	48
SPECint2000		
164.gzip	24	26
175.vpr	53	55
176.gcc	455	500
181.mcf	36	36
186.crafty	70	74
197.parser	49	52
252.eon	101	
253.perlbmk	162	
254.gap	138	135
255.vortex	150	168
256.bzip2	23	24
300.twolf	79	83
SPECfp2000		
168.wupwise	178	81
178.galgel	205	
C++ programma's		
blackbox	217	
deltablue	35	
fpt	744	
gtl	98	
lcom	66	
richards	24	

Tabel 5.6: Compactietijden in seconden bij optimale compactie. Merk op dat deze tijden twee uitvoeringen van SQUEEZE inhouden.



Figuur 5.11: Compactietijden bij optimale compactie. Aan de meetpunten is een tweede-graads kromme gepast.

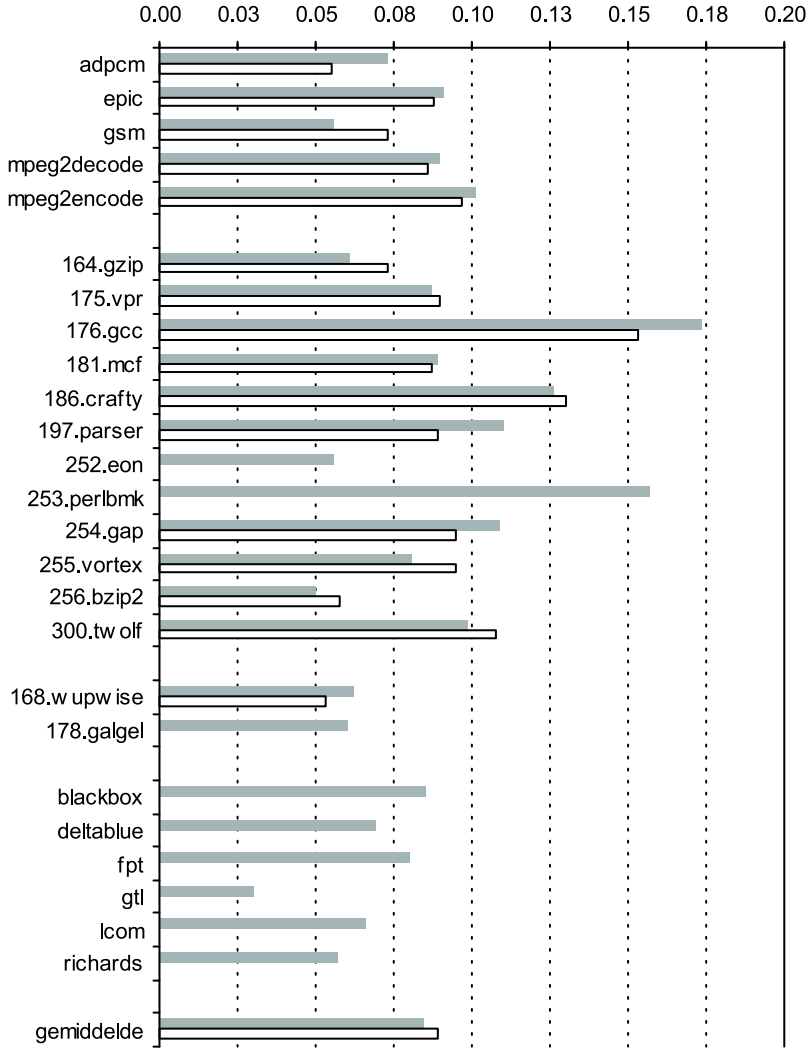
Contextongevoelige levensduuranalyse

Allereerst gaan we na wat de invloed van de contextgevoelige levensduuranalyse op de eindresultaten is door ze te vervangen door de contextongevoelige variant. In figuur 5.12 is de fractie weergegeven van de codecompactie die wegvalt als we enkel een contextongevoelige levensduuranalyse toepassen i.p.v. een contextgevoelige analyse. Deze fractie werd berekend als

$$\text{fractie} = \frac{\text{grootte contextongevoelig} - \text{grootte optimale compactie}}{\text{grootte optimale compactie}}$$

Men kan zien dat men gemiddeld ongeveer 9% minder codecompactie behaalt. Voor de verschillende programma's varieert dit tussen 3 en 18%.

Aan de grootte van de datasecties verandert er courant nauwelijks iets, alhoewel het verlies aan datacompactie toch kan oplopen tot 12%, zoals we kunnen zien in figuur 5.13, die de weggevallen fractie van de datacompactie weergeeft. De reden is dat er tijdens de detectie van



Figuur 5.12: Fractie mindere codecompactie met contextongevoelige levensduuranalyse.

dode data en het propageren van data-adressen nu meer dode data-adressen met elkaar samengevoegd worden waar pijlen samenkomen in de ICVG.

Uiteraard wordt de compactietijd ingekort door het gebruik van de contextongevoelige levensduuranalyse. In figuur 5.14 is de fractie van de compactietijd gegeven die wegvalt bij het gebruik van de minder krachtige variant. Men ziet dat er gemiddeld 21% minder tijd nodig is om de programma's aldus te compacteren.

Triviale levensduuranalyse

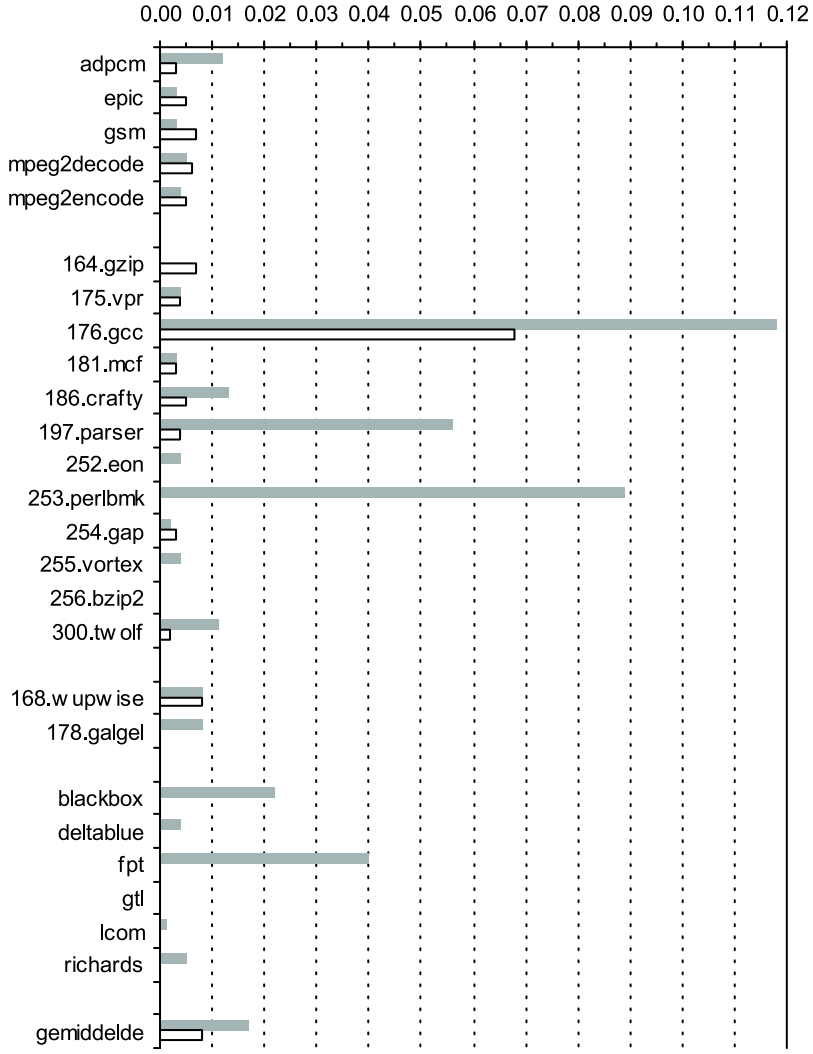
Men kan zich de vraag stellen of een levensduuranalyse sowieso wel nuttig is. Om dit na te gaan hebben we een triviale levensduuranalyse geïmplementeerd in SQUEEZE. Daarbij worden alle registers levend verondersteld aan de uitgang van alle basisblokken.

Uit figuur 5.15 kan men afleiden dat levensduuranalyse wel degelijk een nuttige analyse is. Door de triviale variant te gebruiken i.p.v. de contextgevoelige analyse verliest men gemiddeld 25% van de optimale codecompactie.

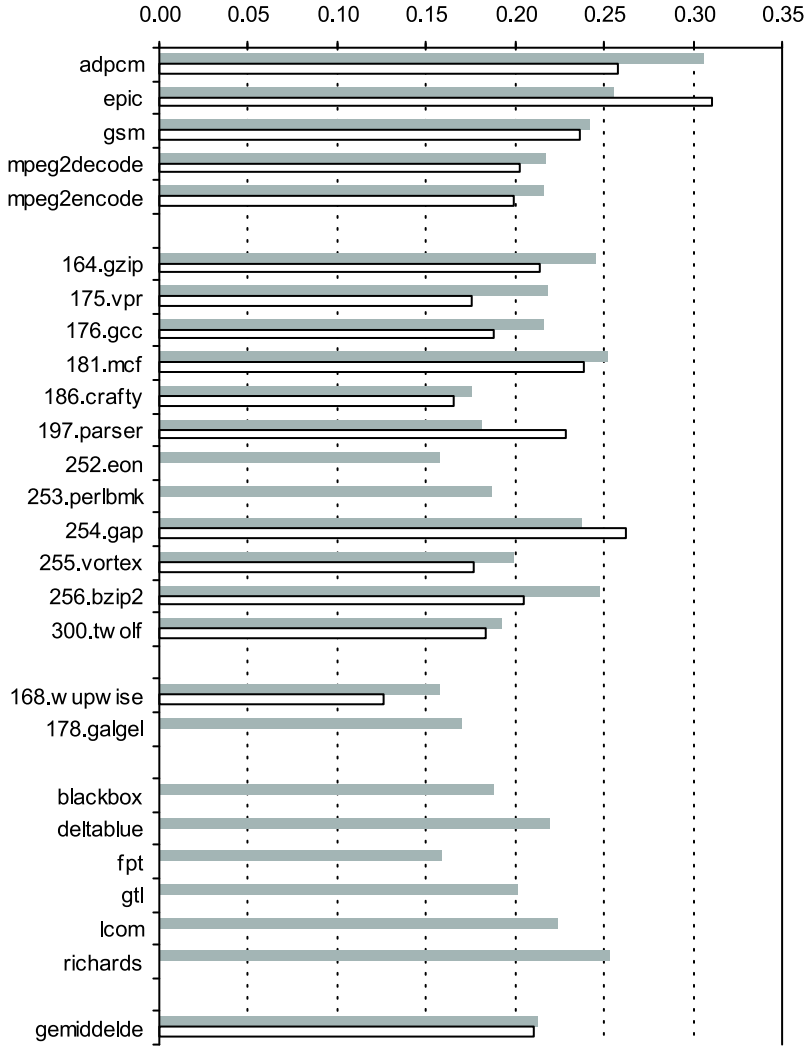
Voor de Gnu versies van de programma's ligt dit vaak een stuk hoger dan voor de Compaq versies. De reden voor dit verschil is eenvoudig: voor die versies is het aandeel van factorisatietechnieken in de totale compactie hoger (zie verder). Het verlies aan compactie door de triviale levensduuranalyse te gebruiken komt vooral voort uit twee transformaties die nu minder presteren: loze-code-eliminatie en factorisatie. Bij gebrek aan vrije registers om terugkeeradressen in te stoppen en doordat er minder basisblokken hernoemd kunnen worden naar elkaar lijdt factorisatie sterk. Omdat net factorisatie een groter aandeel heeft in de compactie van de Gnu versies, is het logisch dat de Gnu versies meer lijden onder het gebrek aan een degelijke levensduuranalyse.

Omdat meer registers levend beschouwd worden, waaronder ook registers die data-adressen bevatten tijdens de analyse van het gebruik van de statisch gealloceerde data, presteert ook de detectie van dode en constante data slechter. Gemiddeld wordt 15-20% minder van de data-secties verwijderd bij het gebrek aan een degelijke levensduuranalyse, zoals te zien is in figuur 5.16. Dit kan evenwel oplopen tot bijna 32%.

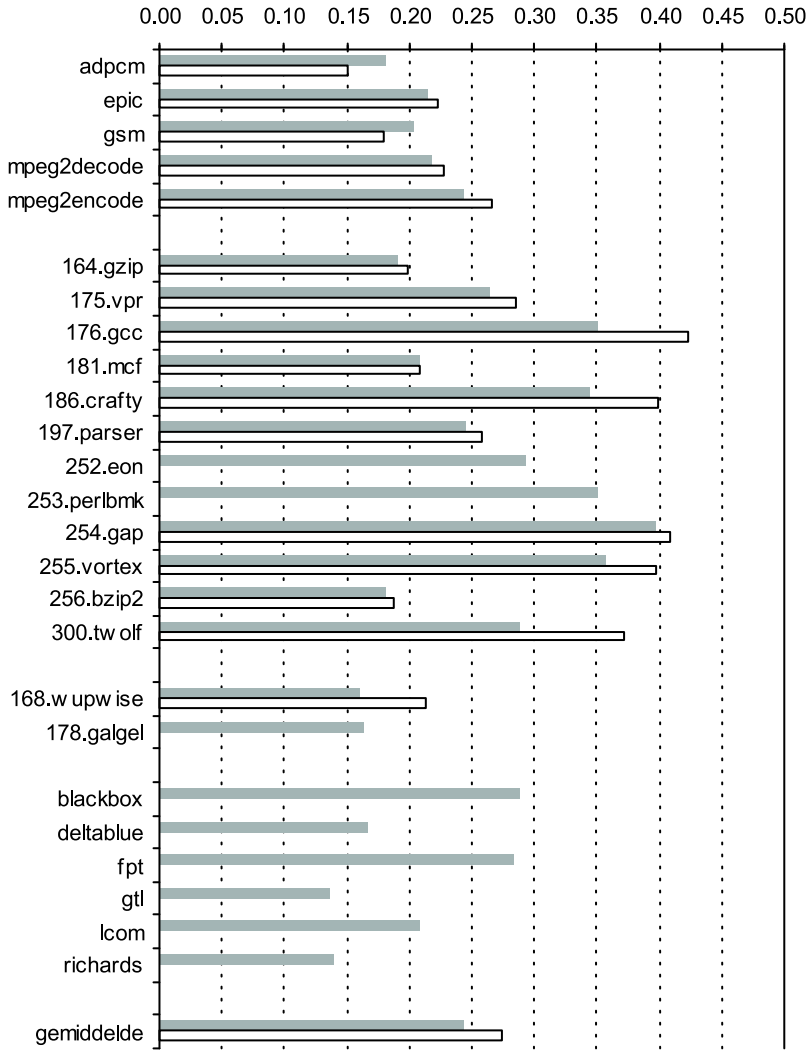
Uit figuur 5.17 blijkt dat levensduuranalyse een flinke brok van de totale compactietijd uitmaakt. Met de triviale levensduuranalyse wordt de compactie gemiddeld immers ongeveer 32% sneller. Dit betekent



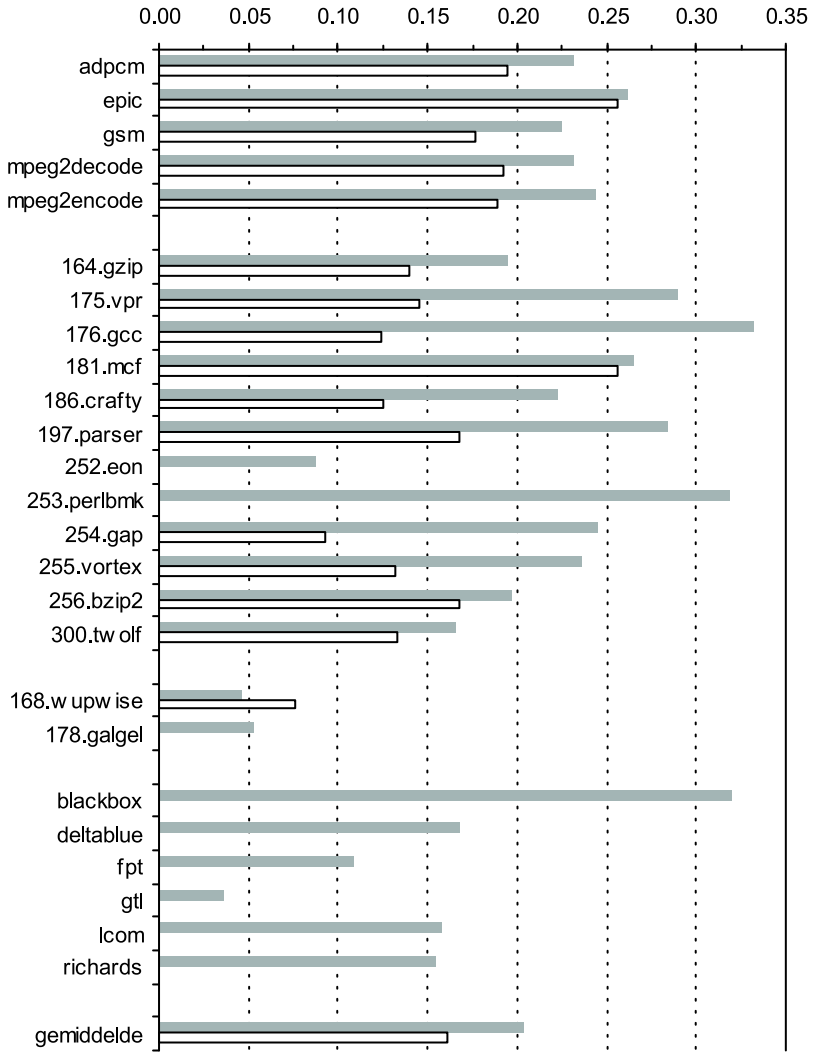
Figuur 5.13: Fractie mindere datacompactie met contextongevoelige levensduuranalyse.



Figuur 5.14: Fractie mindere compactietijd met contextongevoelige levensduuranalyse.



Figuur 5.15: Fractie mindere codecompactie met triviale levensduuranalyse.



Figuur 5.16: Fractie mindere datacompactie met triviale levensduuranalyse.

niet dat er gemiddeld zoveel tijd in de contextgevoelige levensduuranalyse wordt doorgebracht. Omdat de code gevoelig groter blijft, nemen de andere algoritmen immers meer tijd in beslag. Dit verklaart voor een belangrijk deel de verschillen tussen de versnellingen voor de diverse evaluatieprogramma's: men ziet over het algemeen dat daar waar er meer aan codecompactie ingebonden wordt, minder compactietijd gespaard kan worden.

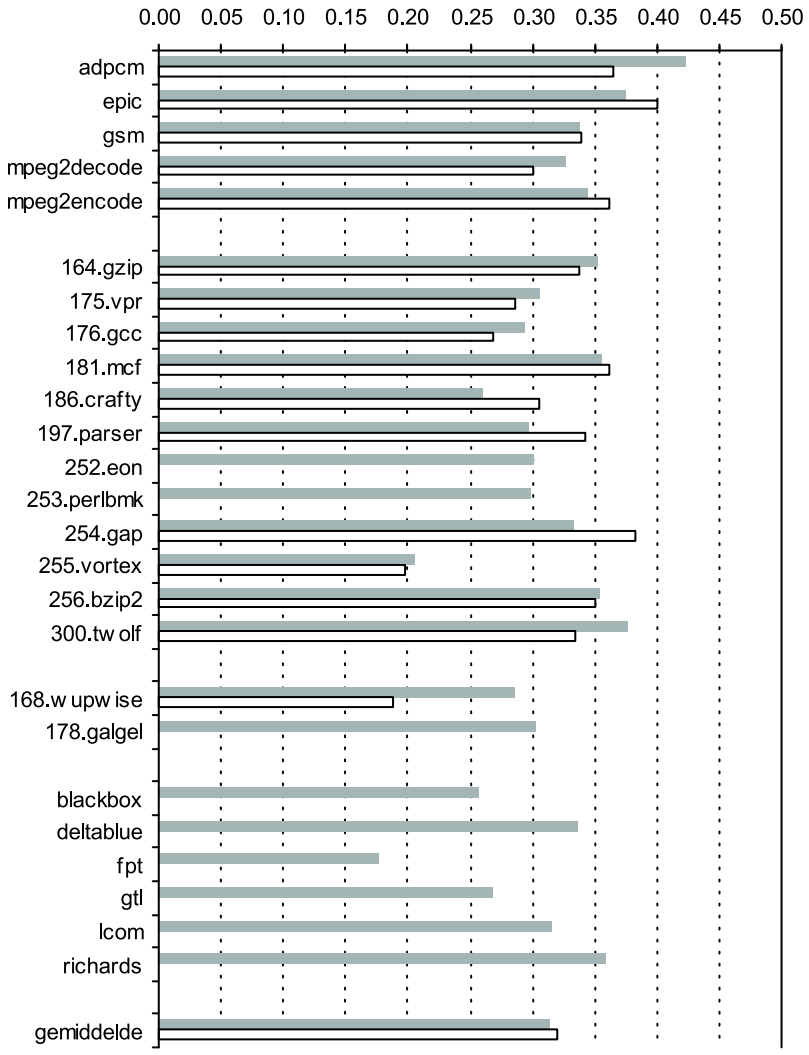
De reden dat de krachtigere levensduuranalyses zoveel tijd in beslag nemen, nog los van de tijd die erdoor in andere algoritmen uitgespaard wordt, is dat ze bijzonder vaak uitgevoerd worden. Telkens wanneer de levensduur van registers kan veranderd zijn en we levensduurinformatie nodig hebben voor een bepaalde analyse of transformatie, wordt die levensduurinformatie helemaal opnieuw berekend. Dit gebeurt overigens steeds met dezelfde variant. Om de compactie sneller te laten verlopen zonder al te veel aan compactie te moeten inboeten zou men een minder krachtige variant kunnen kiezen vóór transformaties die minder nood hebben aan de krachtiger varianten. Daarnaast zou men kunnen trachten de levensduuranalyse op een aantal plaatsen te vermijden door de levensduurinformatie aan te passen tijdens het uitvoeren van de transformaties op de code i.p.v. ze nadien helemaal opnieuw te berekenen. We hebben dit (nog) niet gedaan, SQUEEZE is en blijft immers een prototype met als voornaamste doelstelling een indicator te zijn van de mogelijkheden van compactie na het linken.

Contextgevoelige levensduuranalyse met diepte 1

Van deze additionele verfijning aan de levensduuranalyse is gebleken dat ze nauwelijks iets extra opbrengt. Voor geen enkel evaluatieprogramma bedraagt de extra codecompactie meer dan 1%. We hebben er dan ook geen grafieken voor opgenomen.

5.3.4 Bijdrage van constantenpropagatie

Contextgevoelige constantenpropagatie of constantenpropagatie met laat samenvoegen hebben we niet toegevoegd aan SQUEEZE voor optimale compactie. De reden is dat deze analyses vrij duur zijn qua tijdsgebruik en nauwelijks additionele compactie met zich meebrengen. Enkel voor de Fortran programma's was er een additionele compactie van meer dan een 0.5%, die dan nog beperkt bleef tot minder dan 3%.



Figuur 5.17: Fractie mindere compactietijd met triviale levensduuranalyse.

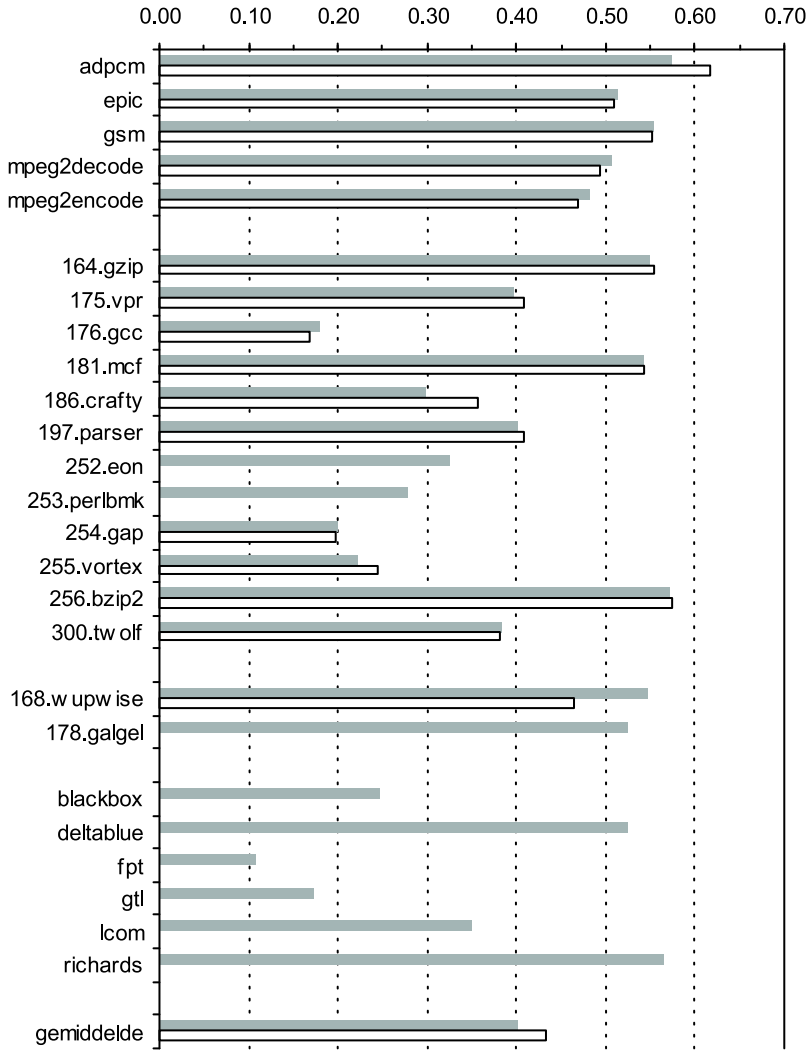
Hier zullen we dus enkel de invloed van contextongevoelige constantenpropagatie beschouwen. Daartoe zetten we de constantenpropagatie af in SQUEEZE en kijken we hoeveel de gecompacteerde programma's groter worden door dit afzetten. Aangezien de analyse van het datagebruik dan geen enkele zin meer heeft, zetten we die eveneens af.

Enkel de waarde van de globale wijzer is overal in het programma bekend (dit is vereist opdat SQUEEZE correct zou werken). Daarnaast zetten we ook nog indirecte procedure-oproepen om naar directe procedure-oproepen wanneer het doeladres uit de globale adrestabel opgeladen wordt in het basisblok van de indirecte oproep. Daarvoor is enkel een eenvoudige propagatie binnen het blok nodig. Merk op dat dit niet volstaat voor de verwijdering van oproepen uit de oproepershelleprocedure op basis van de relocatie-informatie over het gebruik van uit de globale adrestabel opgeladen adressen. Het zou immers kunnen dat het opladen en het gebruik van het adres niet in hetzelfde basisblok gebeuren. In zo'n geval zal bij gebrek aan constantenpropagatie het doel van de indirecte procedure-oproep niet ontdekt worden.

In figuur 5.18 is de fractie van de codecompactie die we niet kunnen verkrijgen zonder constantenpropagatie weergegeven. Deze is bijzonder groot: gemiddeld meer dan 40%, met uitschieters tot meer dan 60%! De belangrijkste reden is dat we geen pijlen vanuit de helleknopen kunnen verwijderen. Daar waar het verlies aan compactie klein is, is dit te wijten aan de codefactorisatie die het gebrek aan constantenpropagatie grotendeels compenseert. Het gaat over die evaluatieprogramma's waar codefactorisatie sowieso al een belangrijk aandeel in de optimale compactie heeft, dat nu nog versterkt wordt.

De tijd die men kan winnen door geen constantenpropagatie uit te voeren is weergegeven in figuur 5.19 en is eveneens bijzonder hoog: de gemiddelde tijdswinst ligt rond de 40%, met uitschieters tot bijna 60%. Een deel van deze tijdswinst moet toegeschreven worden aan het minder uitvoeren van de levensduuranalyse. Omdat het grotendeels door het transformeren van berekeningen op constanten is dat er registers dood worden, worden er veel minder registers dood. Een gevolg hiervan is dat er in SQUEEZE minder iteraties tussen levensduuranalyse en loze-code-eliminatie nodig zijn.

Merk op dat we voor deze tijdsmetingen uitgegaan zijn van twee uitvoeringen van SQUEEZE. Dit maakt een faire vergelijking met de andere varianten mogelijk. Immers, een gebruiker kan ook voor alle



Figuur 5.18: Fractie mindere codecompactie zonder constantenpropagatie.

andere bestudeerde varianten van SQUEEZE opteren om mèt constantenpropagatie en mèt detectie van dode data SQUEEZE toch slechts eenmaal uit te voeren. Hij krijgt dan wel de codecompactie die met deze detectie gepaard gaat, maar niet de datacompactie. Dan moeten alle uitvoeringstijden van de andere analyses door twee gedeeld worden, net zoals dit bij deze variant van SQUEEZE zonder constantenpropagatie het geval is. Op de relatieve tijden of tijdswinsten maakt dit dus geen verschil.

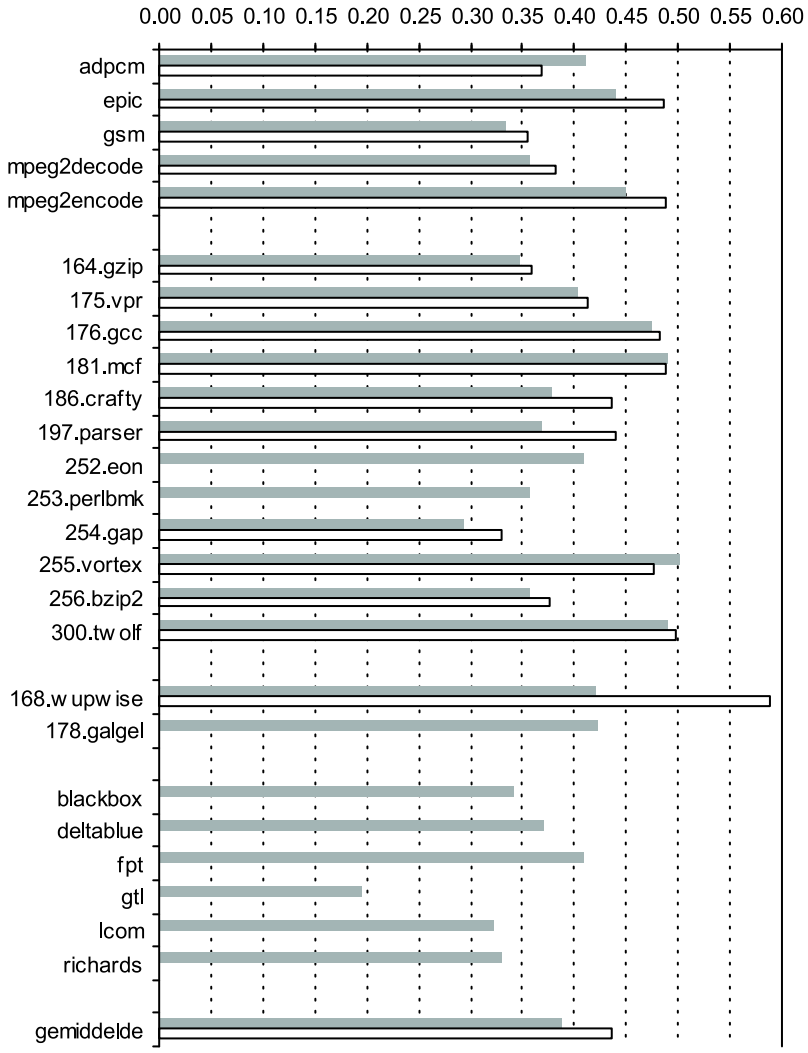
Merk ook op dat de codecompactie zonder constantenpropagatie mede versneld wordt door het niet uitvoeren van de detectie van dode data. Men kan een constantenpropagatie evenwel ook uitvoeren zonder de detectie van dode data. Dan moeten er tijdens de constantenpropagatie geen slechtste-geval-veronderstellingen gemaakt worden en is er geen partiële evaluatie van adresberekeningen volgend op de constantenpropagatie. Het aandeel van de detectie van dode data in de cijfers uit figuur 5.19 wordt duidelijk bij de volgende variant van SQUEEZE die we geëvalueerd hebben.

5.3.5 Bijdrage detectie gebruik data

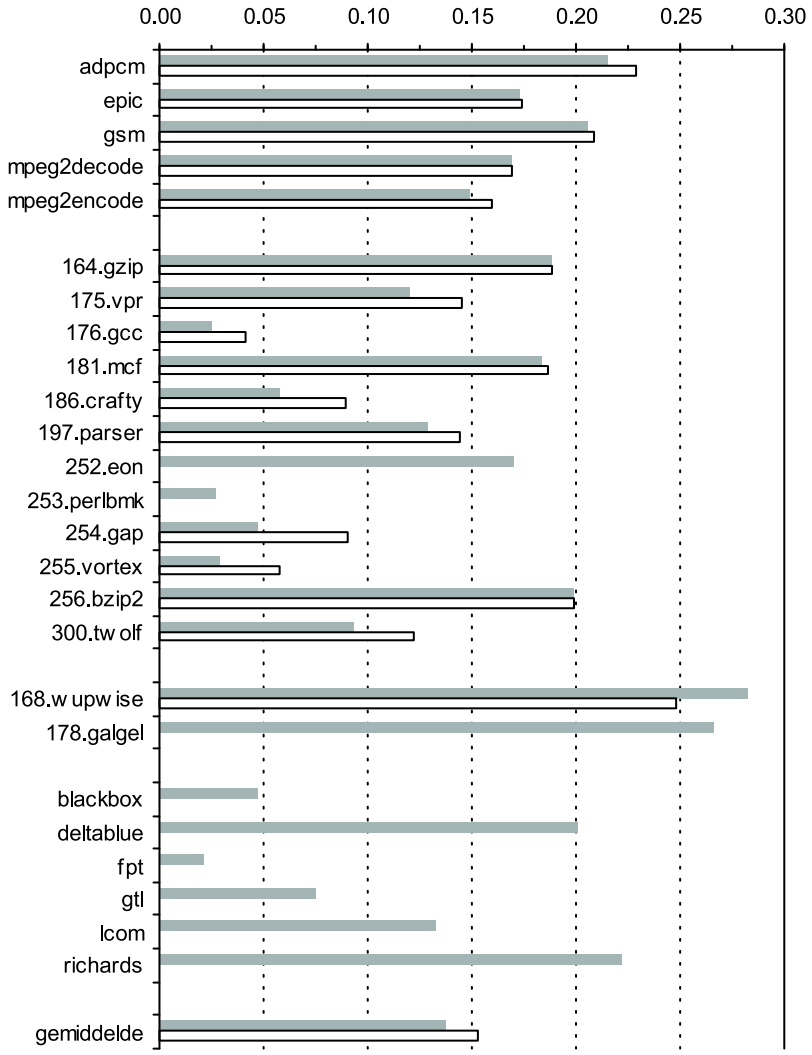
In figuur 5.20 is de fractie van de codecompactie weergegeven die wegvalt wanneer we geen analyse van het datagebruik uitvoeren. Deze fractie is nagenoeg geheel te wijten aan het niet kunnen verwijderen van pijlen vanuit de oproepershelleknoop omdat er geen code-adressen in de data dood worden. Gemiddeld zien we dat we rond de 15% minder codecompactie behalen. Uiteraard is er geen datacompactie zonder deze analyse, van de datacompactie gaat met andere woorden 100% verloren.

Als we deze grafiek vergelijken met die bij het uitblijven van constantenpropagatie (figuur 5.18) kunnen we concluderen dat gemiddeld minder dan 25% van de optimale codecompactie gehaald wordt uit de optimalisaties van de code aan de hand van gevonden constanten.

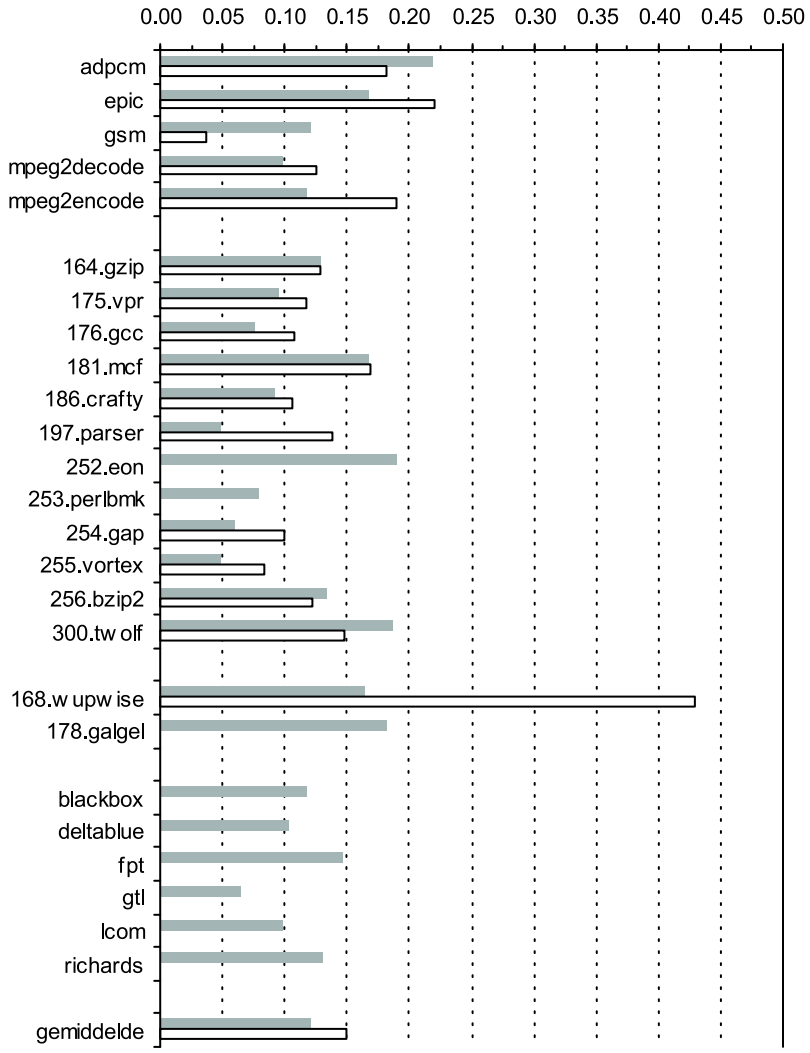
In figuur 5.21 is de fractie van de compactietijd weergegeven die we uitsparen door geen detectie van het datagebruik te doen. Gemiddeld ligt dit onder de 15%. De uitschieter voor de Gnu versie van 168.wupwise kunnen we niet echt verklaren. Samen met de versnelling die er is als we helemaal geen constantenpropagatie uitvoeren kunnen we concluderen dat de constantenpropagatie op zich minder dan 25% van de uitvoeringstijd in beslag neemt.



Figuur 5.19: Fractie mindere compactietijd zonder constantenpropagatie.



Figuur 5.20: Fractie mindere codecompactie zonder detectie van datagebruik.



Figuur 5.21: Fractie mindere compactietijd zonder detectie van datagebruik.

Zoals we in sectie 3.3.7 reeds geschreven hebben is het mogelijk de prestatie van deze analyse te verhogen door op de een of andere manier de granulariteit van de datablokken te verkleinen. We hebben voorlopig nog geen sluitende analyse kunnen ontwikkelen om blokken op te splitsen tijdens de compactie na het linken. Daarom proberen we de blokken op te splitsen vóór het linken, door de broncode te bewerken.

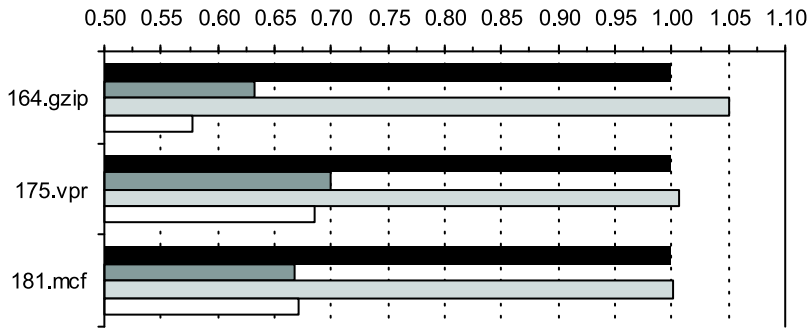
Daartoe hebben we een ontleder geschreven die semi-automatisch bronbestanden in C opsplitst in verschillende bronbestanden, zodanig dat er per bronbestand slechts 1 globaal object (procedure of globale data) gedefinieerd wordt. Als we het programma vertalen en linken vertrekkend van de opgesplitste bronbestanden, krijgen we een programma met meer en dus kleinere datablokken. Als gevolg van het opsplitsen kan de vertaler echter minder code-optimalisaties uitvoeren, o.m. omdat alle globale objecten die "static" gedeclareerd stonden, dat nu niet meer zijn en de interprocedurale optimalisaties niet meer uitgevoerd worden, aangezien alle interprocedurale sprongen nu intermodulaire sprongen geworden zijn. Het aldus gegenereerde programma wordt dus groter. Merk op dat we de opsplitsing van bronbestanden enkel op de applicatiespecifieke code toegepast hebben. Van de bibliotheekcode beschikken we immers enkel over objectbestanden.

Omdat onze ontleder slechts semi-automatisch is hebben we het opsplitsen in verschillende bronbestanden slechts voor drie van de evaluatieprogramma's uitgevoerd. Bovendien hebben we enkel de Compaq versies van de programma's gebruikt. De resultaten zijn dus enkel richtinggevend.

Figuren 5.22 en 5.23 bevatten de relatieve groottes van resp. code en data voor de basisversies en de optimaal gecompacteerde versies, met en zonder opsplitsing van de bronbestanden. Zoals te verwachten valt neemt de grootte van code en data in de basisversies toe als de bronbestanden opgesplitst worden, zij het dat de verschillen niet altijd groot moeten zijn. Belangrijker evenwel is dat we dit voor twee van de drie programma's kunnen compenseren, zowel voor de grootte van de code als die van de data in de optimaal gecompacteerde programma's. Daarvoor zijn twee redenen.

- Omdat de datablokken kleiner zijn na het opsplitsen van de bronbestanden presteert de detectie van het datagebruik beter.

- Omdat alle originele 'static' procedures in de applicatiespecifieke code nu niet meer 'static' zijn, kan de compactie profiteren van

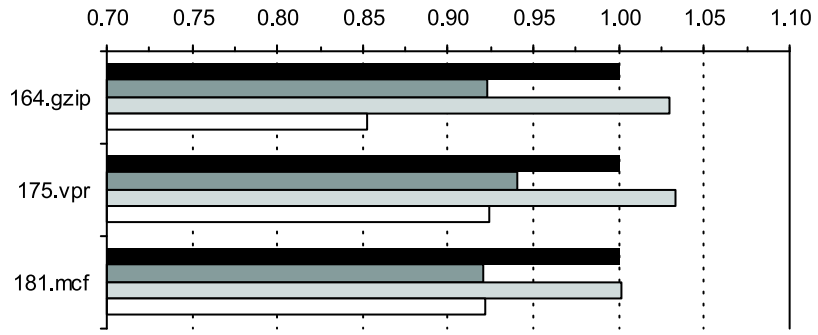


Figuur 5.22: Relatieve groottes van de code bij opsplitsing bronbestanden. Alle balkjes in deze grafiek zijn relatieve waarden. Ze zijn gerelativeerd t.o.v. de grootte van de code in de basisversies van de evaluatieprogramma's zonder dat de bronbestanden opgesplitst werden. Het zwarte balkje is de referentiewaarde. Het donkergrijze balkje geeft de relatieve grootte van de optimaal gecompacteerde programma's zonder opsplitsing van de bronbestanden, d.i. de optimale compactiefactor voor code. Het lichtgrijze balkje geeft de relatieve grootte aan van de basisversies na opsplitsing van de bronbestanden. Het witte balkje tenslotte geeft de relatieve grootte aan van de code van de optimaal gecompacteerde programma's na opsplitsing van de bronbestanden.

de hellepijlen die naar deze procedures wijzen om het stapelgedrag van deze procedures minder conservatief in te schatten. Bij "static" procedures is dit niet het geval, omdat de procedures niet geëxporteerd worden uit het hun bron- of objectbestand en het dus kan dat de vertaler niet alle oproepconventies gevolgd heeft.

Mochten we ook bibliotheekbestanden met onze ontleder kunnen opsplitsen, dan zouden de resultaten ongetwijfeld nog beter zijn: uit de bibliotheken wordt er immers meer code en data nutteloos in het gelinkte programma opgenomen.

Merk op dat vele vertalers voor ingebedde systemen een soortgelijke techniek toepassen: ofwel wordt per object of groep van sterk gerelateerde objecten een apart objectbestand aangemaakt, zodanig dat niet alle objectcode of -data horend bij één bronbestand automatisch moet meegelinkt worden, ofwel kunnen vertaler en linker werken met objectbestanden waarbinnen een soortgelijke opdeling kan gemaakt worden, zodat niet altijd alle code- en/of datablokken uit een objectbestand moeten meegelinkt worden.



Figuur 5.23: Relatieve groottes van de data bij opsplitsing bronbestanden. Alle balkjes in deze grafiek zijn relatieve waarden. Ze zijn gerelativeerd t.o.v. de grootte van de data in de basisversies van de evaluatieprogramma's zonder dat de bronbestanden opgesplitst werden. Het zwarte balkje is de referentiewaarde. Het donkergrijze balkje geeft de relatieve grootte van de optimaal gecompacteerde programma's zonder opsplitsing van de bronbestanden, d.i. de optimale compactiefactor voor data. Het lichtgrijze balkje geeft de relatieve grootte aan van de basisversies na opsplitsing van de bronbestanden. Het witte balkje tenslotte geeft de relatieve grootte aan van de data van de optimaal gecompacteerde programma's na opsplitsing van de bronbestanden.

5.3.6 Bijdrage van factorisatie

In deze sectie wordt de invloed van de verschillende factorisatiealgoritmen besproken. Daartoe zullen we eerst de programma's compacteren zonder globale factorisatie. Vervolgens zullen we, te beginnen bij factorisatie van procedures, steeds meer globale factorisatietechnieken terug toevoegen aan SQUEEZE. De reden dat we hier wel "additief" te werk gaan en de technieken niet afzonderlijk gewoon afzetten om ze te evalueren is natuurlijk dat de factorisatietechnieken op een grotere codegranulariteit de factorisatiemogelijkheden voor de lagere niveaus wegnemen. Als bv. twee identieke procedures gefactoriseerd worden, moeten de basisblokken in die procedures niet apart meer gefactoriseerd worden.

Daarna bekijken we hoe de lokale factorisatietechnieken en het hergebruiken van statisch gealloceerde data de codefactorisatie beïnvloeden. Voor het tweede zullen we alle codefactorisatietechnieken uitvoeren, maar de datafactorisatietechnieken afzetten.

Globale factorisatie

In figuur 5.24 is het verlies aan compactie door het afschakelen van alle globale factorisatietechnieken (procedures, coderegio's, basisblokken en instructiesequenties) weergegeven.

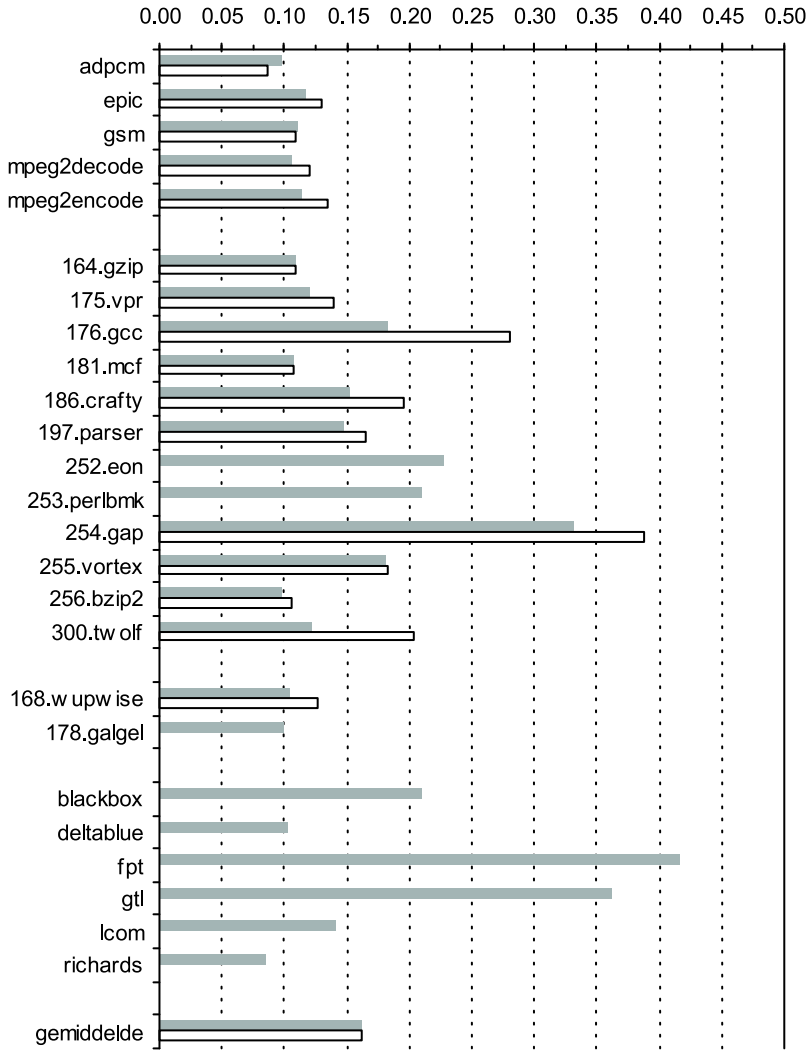
Gemiddeld halen we 15% minder codecompactie bij gebrek aan factorisatie. Opvallend is dat voor zowat alle programma's (de enige uitzondering is adpcm) de factorisatie belangrijker is bij de Gnu versies. Verder merkt men op dat factorisatie minder belangrijk is bij kleine programma's, wat logisch is: de kans dat er identieke of naar elkaar te hernoemen codefragmenten in het programma voorkomen is kleiner.

Op de grootte van de datasecties heeft codefactorisatie nagenoeg geen invloed. We hebben er dan ook geen grafiek voor opgenomen.

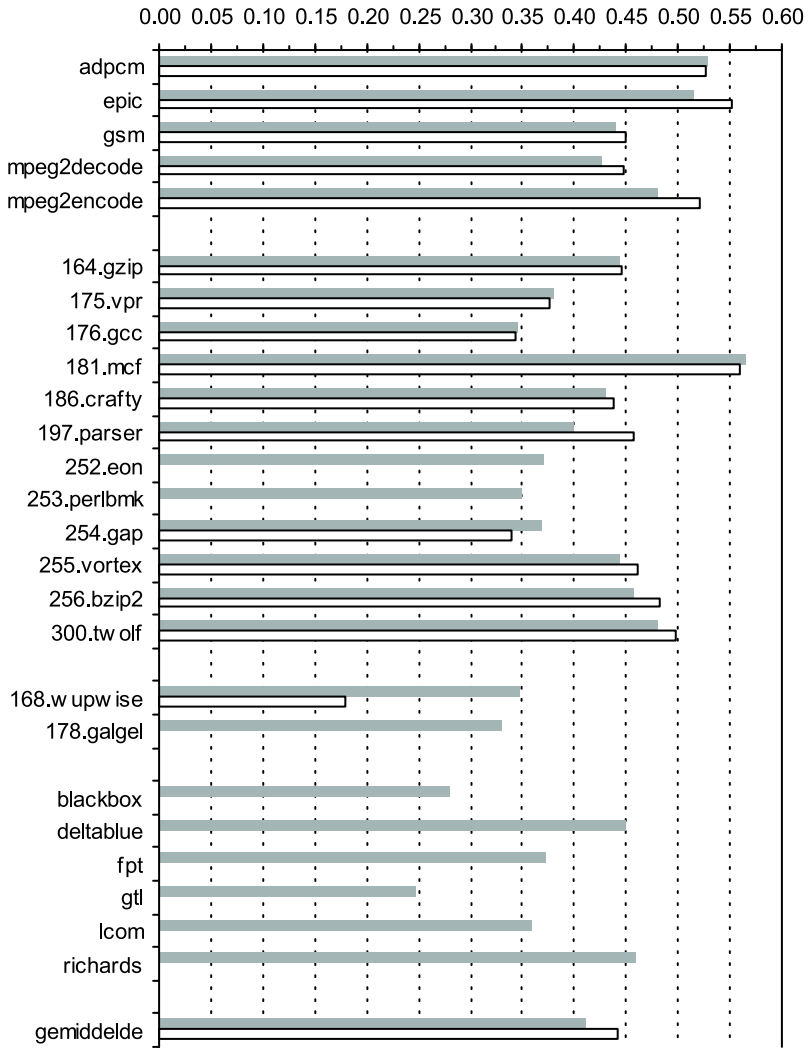
In figuur 5.25 zien we dat factorisatie een flinke brok uit het tijdsbudget voor de compactie inneemt, zeker voor de kleinere programma's. Gemiddeld bedraagt dit meer dan 40%. Men dient er rekening mee te houden dat de tijd nodig om te factoriseren niet enkel naar de factorisatietechnieken zelf gaat. Ondersteunende analyses zoals levensduuranalyse worden ook minder uitgevoerd. Bovendien wordt de laatste uitvoering van alle basistransformaties gevoelig sneller omdat de ICVG van het programma in die fase eenvoudiger is als men geen factorisatie toepast: de verhouding tussen aantal pijlen en aantal knopen in de graaf verandert immers grondig door het factoriseren.

De bijdragen aan codecompactie door globale factorisatietechnieken is in figuren 5.26 en 5.27 opgesplitst over de verschillende technieken volgens granulariteit. Het meest linkse balkje is de fractie van de codecompactie uit globale factorisatietechnieken die op rekening van de procedurale factorisatie kan geschreven worden. Voert men daar bovenop nog factorisatie van coderegio's uit, dan levert dit de extra winst op in het tweede balkje van links. Voegt men er factorisatie van basisblokken aan toe, dan krijgt men er ook de fractie van het derde balkje bij. Het vierde balkje wordt toegevoegd door ook de specifieke instructiesequenties voor het bewaren van achteraf te bewaren registers te factoriseren. Het vijfde balkje tenslotte is het aandeel van een additionele factorisatie van instructiesequenties die geen volledig basisblok vormen.

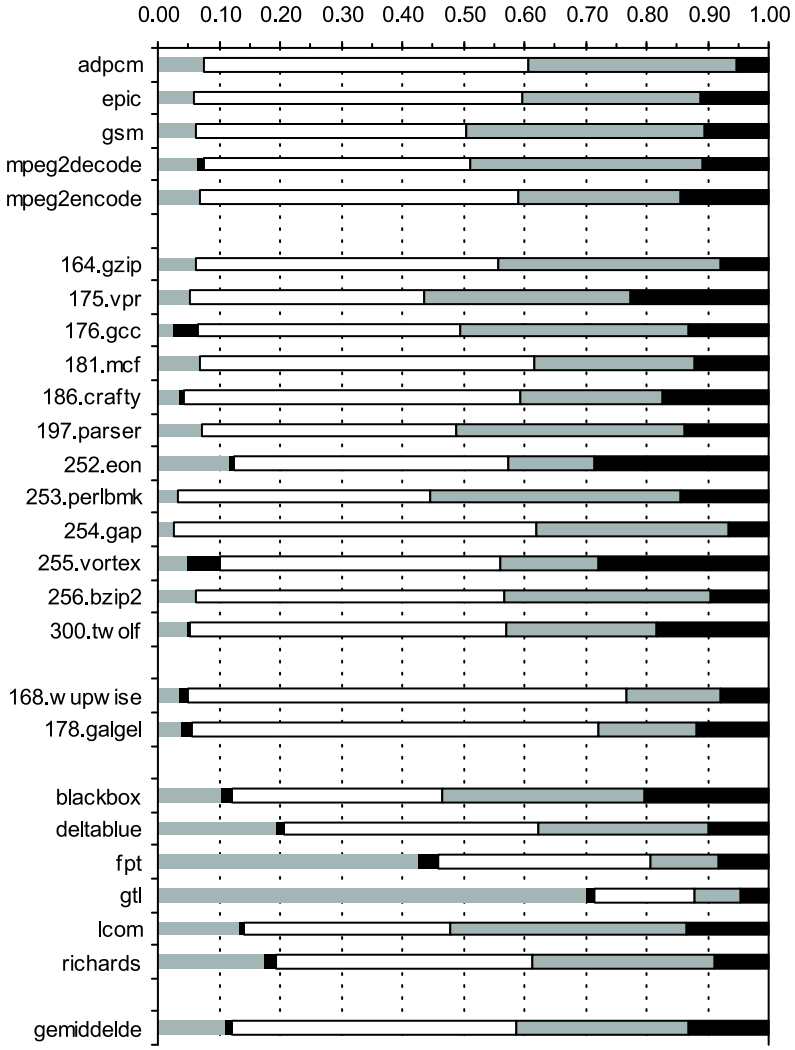
Uit de grafiekjes valt af te leiden dat de bijdragen van de technieken op verschillende granulariteiten gelijklopen voor de Gnu en Compaq fracties, ondanks het gemiddeld beter presteren van de globale factori-



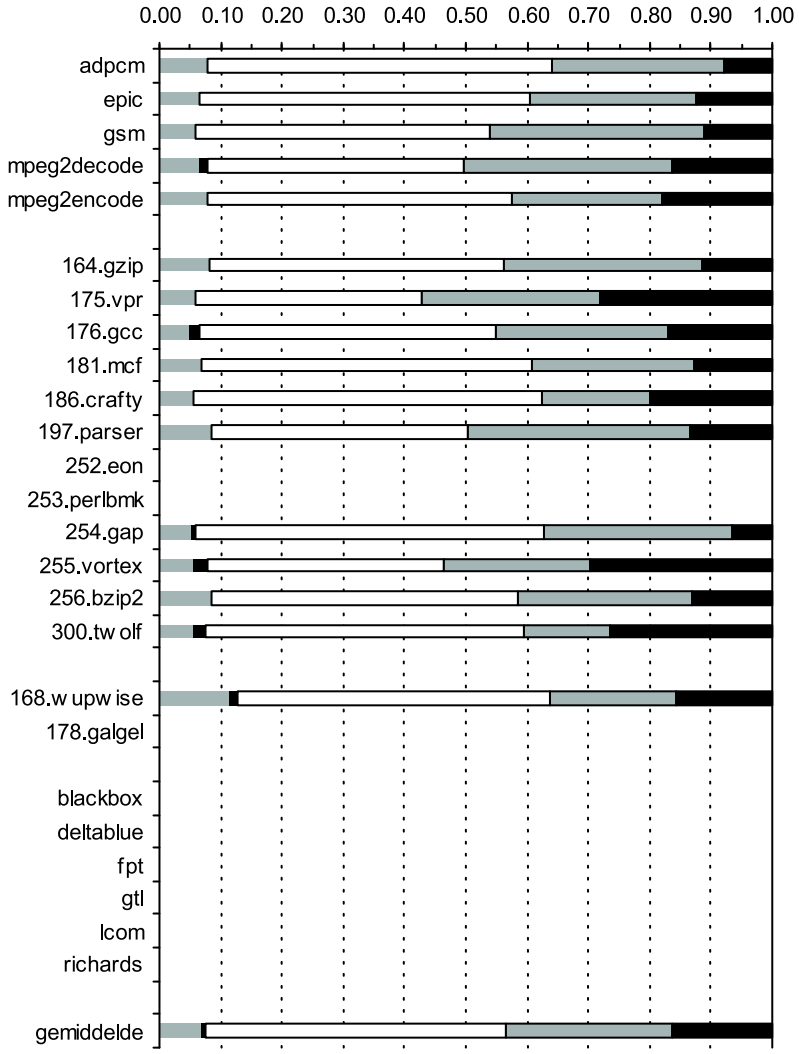
Figuur 5.24: Fractie mindere codecompactie zonder factorisatie.



Figuur 5.25: Fractie mindere compactietijd zonder factorisatie.



Figuur 5.26: Opsplitsing voor verschillende granulariteiten van de bereikte codecompactie door globale factorisatietechnieken voor de Compaq versies van de evaluatieprogramma's. Van links naar rechts zijn de fracties van procedurale factorisatie, factorisatie van regio's, factorisatie van volledige basisblokken, factorisatie van specifieke instructiesequenties en tenslotte factorisatie van algemene instructiesequenties weergegeven. De balkjes zijn additief: het derde balkje geeft bv. de fractie van de codecompactie door globale factorisatietechnieken aan die behaald wordt door bovenop procedurale factorisatie en factorisatie van regio's ook basisblokken te factoriseren.



Figuur 5.27: Opsplitsing voor verschillende granulariteiten van de bereikte codecompactie door globale factorisatietechnieken voor de Gnu versies van de evaluatieprogramma's.

satie in zijn geheel voor Gnu programma's. Voor de C++ programma's valt de grote bijdrage van de factorisatie van procedures op, wat te verwachten viel. Vooral de factorisatie van regio's levert niet veel op, voor de kleinere programma's meestal zelfs niks. Gemiddeld gaan de factorisatie van basisblokken en die van specifieke instructiesequenties met de grootste brok lopen.

In figuren 5.28 en 5.29 is dezelfde opsplitsing gemaakt, maar nu zijn de bijdragen aan de compactietijd van de globale factorisatietechnieken getoond. Hierbij moeten we opmerken dat de tijd die aan factorisatie van regio's toegeschreven wordt niet alleen vereist is voor deze vorm van factorisatie. Het opsplitsen van de basisblokken in een blok met berekeningen en een blok met enkel de controletransfer moet sowieso gebeuren als men factorisatie van basisblokken wil uitvoeren. Het overgrote deel van de factorisatietijd voor regio's zou bij het afzetten van deze vorm van factorisatie bij de factorisatie van basisblokken moeten toegevoegd worden.

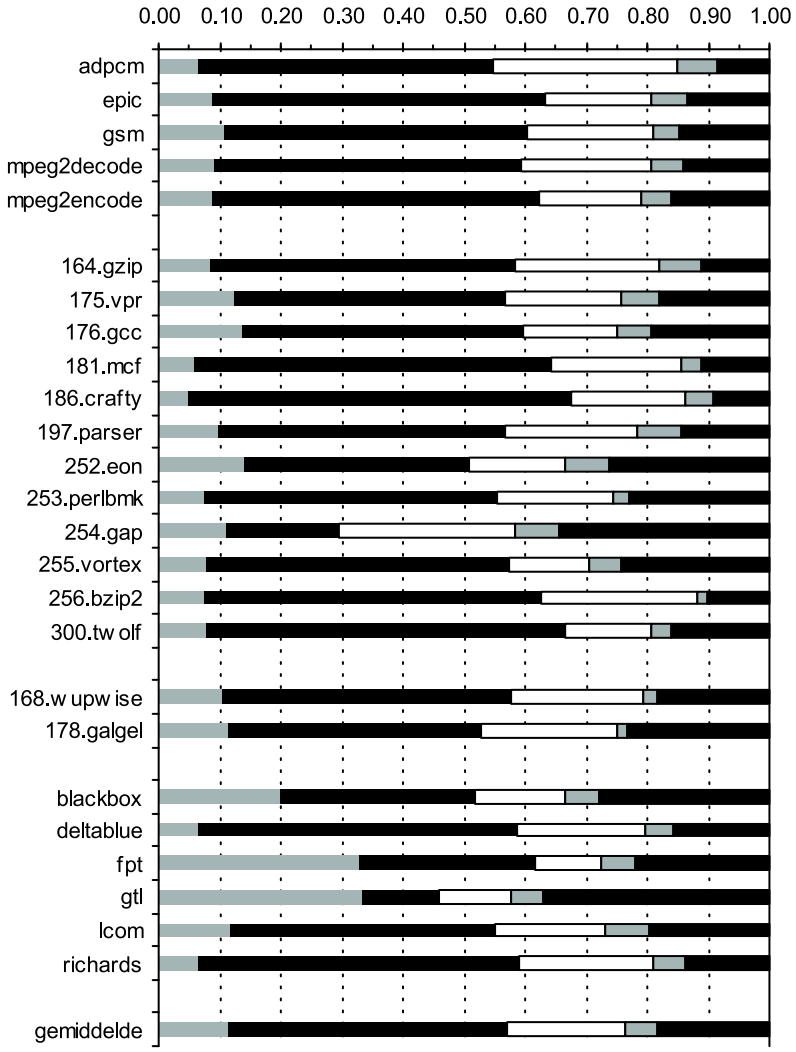
Algemeen kunnen we vaststellen dat de fractie van de totale factorisatietijd die in de verschillende factorisatietechnieken gestopt wordt evenredig is met de compactie die erdoor behaald wordt. De factorisatie van specifieke instructiesequenties gebruikt wel een opvallend kleinere fractie van de factorisatietijd dan haar aandeel in de codecompactie.

Lokale factorisatie

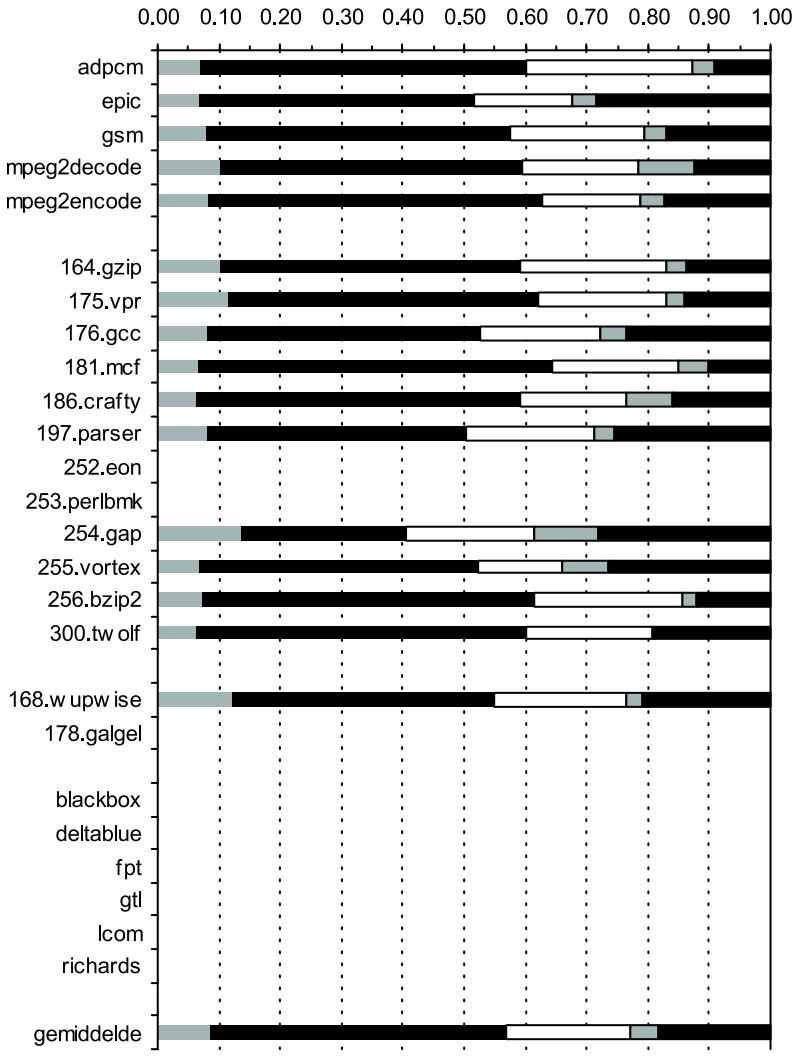
De lokale factorisatie hebben we apart behandeld. Strikt genomen is dit immers geen factorisatietechniek, maar gaat het om het verplaatsen van code (ook wel eliminatie van gemeenschappelijke code genoemd).

Lokale factorisatie is, zoals in figuur 5.30 te zien is, verantwoordelijk voor gemiddeld ongeveer 3% van de totale codecompactie, wat niet veel is. Op de data heeft deze vorm van factorisatie uiteraard geen invloed.

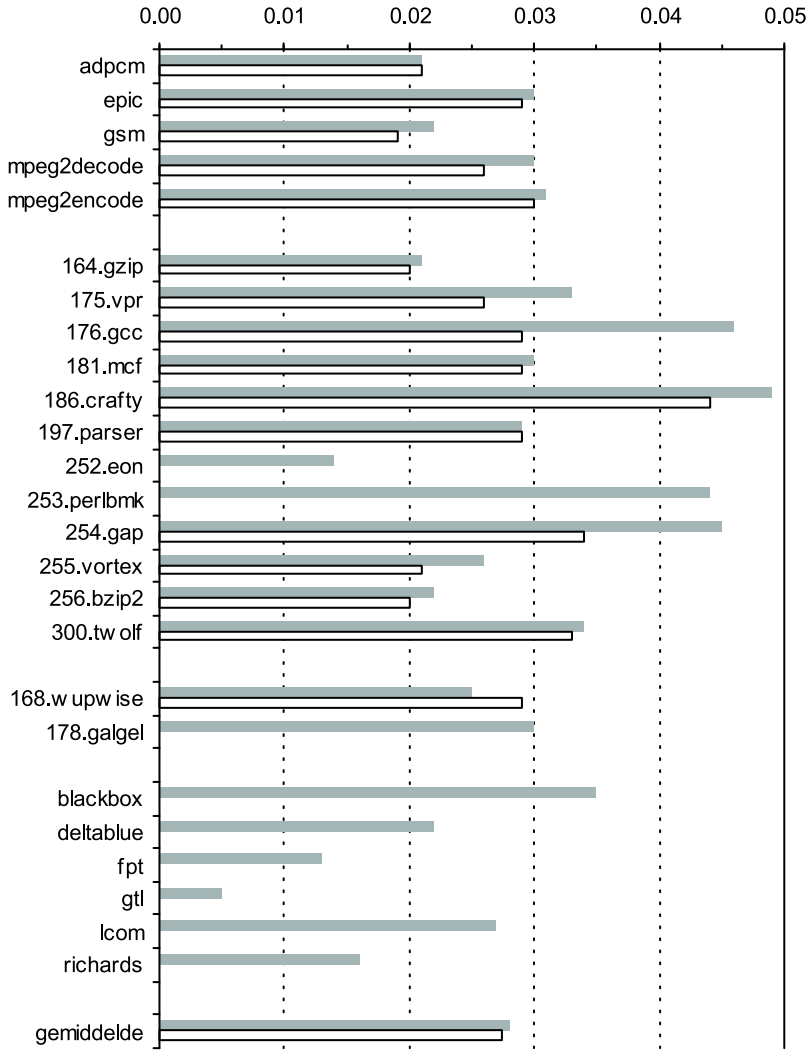
Gemiddeld wordt ongeveer 6% van de totale compactietijd besteed aan lokale factorisatie en aan de analyses waarvan ze afhankelijk is, zoals levensduuranalyse, aliasanalyse en de eliminatie van onbereikbare code. Dit laatste is in ons prototype vereist om aliasanalyse te kunnen uitvoeren. Omdat we bij het uitschakelen van de lokale factorisatie ook die analyses uitschakelen, inclusief de eliminatie van onbereikbare code, zal er soms tijdelijk wat meer code in het programma aanwezig



Figuur 5.28: Opsplitsing voor verschillende granulariteiten van de uitvoeringstijd van globale factorisatietechnieken voor de Compaq versies van de evaluatieprogramma's.



Figuur 5.29: Opsplitsing voor verschillende granulariteiten van de uitvoeringstijd van globale factorisatietechnieken voor de Gnu versies van de evaluatieprogramma's.



Figuur 5.30: Verlies aan optimale codecompactie als niet lokaal gefactoriseerd wordt.

zijn gedurende de transformaties die volgen op lokale factorisatie, tot de volgende eliminatie van onbereikbare code uitgevoerd wordt. Dit is de reden waarom er voor sommige programma's trager gecompacteerd wordt als we de lokale factorisatie afschakelen.

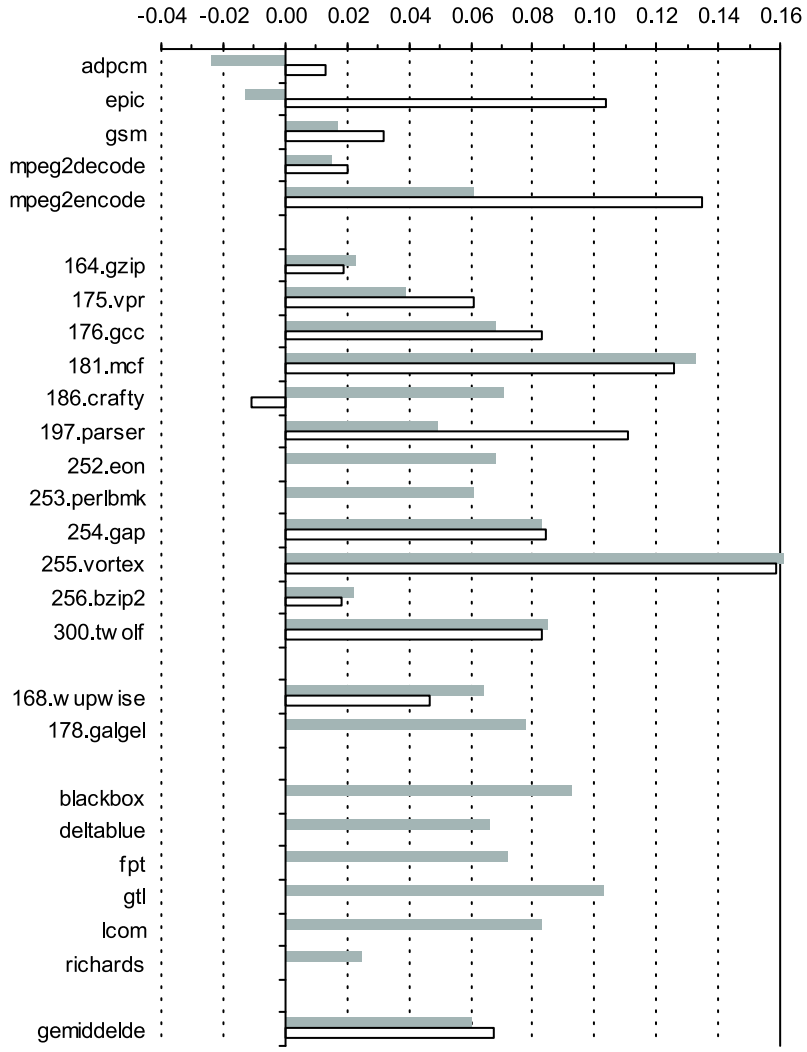
Parametriseren van procedures

Zoals we in sectie 4.2 reeds aangehaald hebben, hebben we parametrisatie van gelijkaardige procedures nog niet succesvol kunnen toepassen omdat we nauwelijks of geen kandidaten vinden waarvan we kunnen aantonen dat de geparametriseerde procedure niet recursief opgeroepen kan worden. Enkel voor procedures waarvoor dat het geval is moeten we de parameter niet op de stapel plaatsen.

Om de (toekomstige) mogelijkheden van procedureparametrisatie toch enigszins te kunnen inschatten hebben we enkele experimenten uitgevoerd, waarbij we ervan uitgaan dat de samengevoegde procedures niet recursief opgeroepen zullen worden (wat uiteraard tot foutieve programma's aanleiding geeft). Is er een vrij register om de parameter in op te slaan, dan doen we dit. Is er geen vrij register, dan slaan we de parameter op in een stukje statisch gealloceerd geheugen dat we zelf alloceren. Daartoe maken we zelf een nieuw datablokje aan dat precies één parameter kan bevatten. Procedures worden slechts per 2 geparametriseerd.

Deze implementatie hebben we getest op twee programma's waarin nogal wat sjablonen en overerving gebruikt worden: 252.eon en gtl. In tabel 5.7 zijn de codegroottes voor de optimaal gecompacteerde programma's met en zonder parametrisatie er bovenop weergegeven. Men ziet dat de programma's nog eens enkele procenten kleiner worden. Dit is niet heel veel, wat natuurlijk voor een deel verklaard wordt door het feit dat andere globale factorisatietechnieken het uitblijven van parametrisatie gedeeltelijk kunnen opvangen.

Om dit effect enigszins te kunnen inschatten hebben we het aantal directe procedure-oproepen in beide versies van de programma's eveneens in de tabel opgenomen. Zulke oproepen worden gebruikt wanneer er procedurale abstractie gebruikt wordt, bv. bij de factorisatie van basisblokken of coderegio's. Men ziet dat het aantal directe procedure-oproepen veel sterker daalt, wat erop wijst dat de andere factorisatietechnieken inderdaad minder toegepast moeten worden. Er zullen dus minder onkosten in het programma geïntroduceerd worden



Figuur 5.31: Fractie van de compactietijd die men wint als niet lokaal gefactoriseerd wordt.

	optimale compactie		optimale compactie + parametriseren	
programma	omvang code	# bsr	omvang code	# bsr
252.eon	302528	8156	295936 (97.8%)	7699 (94.4%)
gtl	254848	8435	246080 (96.6%)	7026 (83.3%)

Tabel 5.7: Resultaten van de parametrisatie: de omvang van de code voor de gecompacteerde programma's zonder en met parametrisatie bovenop de optimale compactie. Tussen haakjes staat de verhouding tussen beide. Daarnaast is ook het aantal bsr (*branch to subroutine*) instructies in het programma opgenomen in de tabel.

door procedurale abstractie, wat de uitvoeringssnelheid ten goede zal komen.

Hiermee hopen we enigszins beargumenteerd te hebben dat het parametriseren van procedures wel degelijk een nuttige extra vorm van factorisatie zou kunnen worden, mochten we de oproepgraaf van het programma nauwkeuriger kennen of het stapelgedrag van procedures kunnen aanpassen voor parametrisatie.

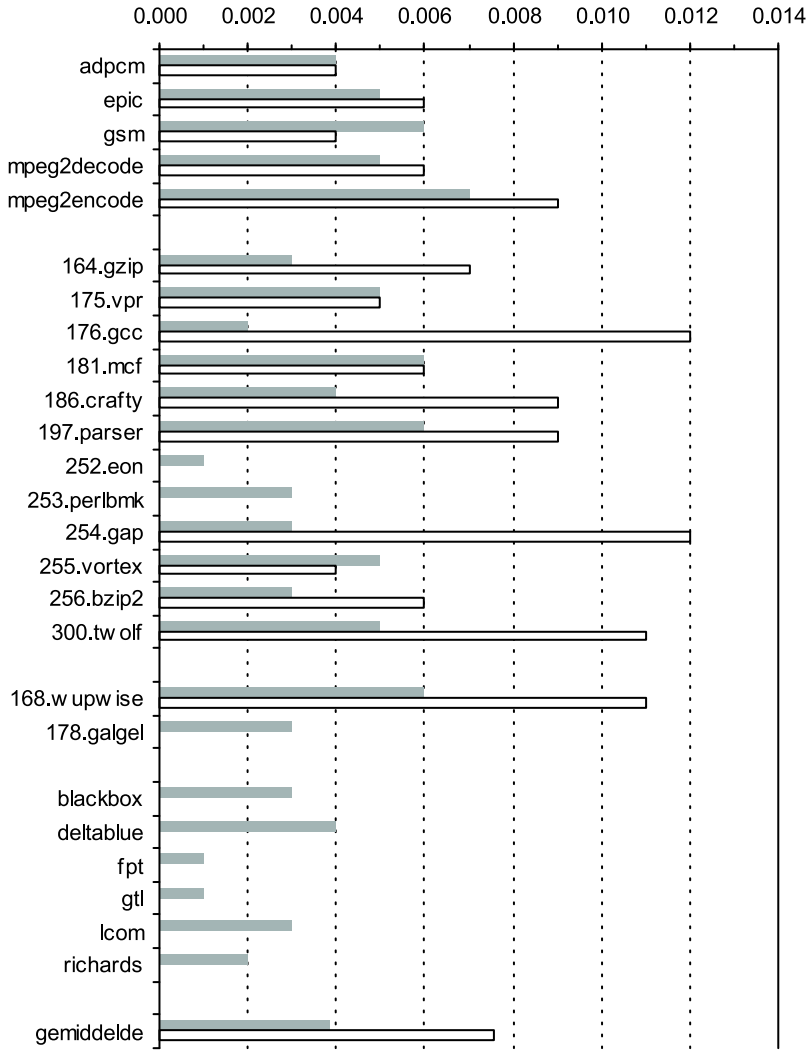
Hergebruik van data

De resultaten voor code- en datacompactie ten gevolge van het afschakelen van het hergebruiken van data zijn in grafieken 5.32 en 5.33 weergegeven. Men ziet dat deze transformaties nauwelijks bijdragen aan de bereikte compactie van de code. Enkele uitzonderingen terzijde gelaten geldt dit ook voor de data. Zonder de voorafgaande compactie van de globale adrestabel zou de invloed van deze transformatie veel groter geweest zijn.

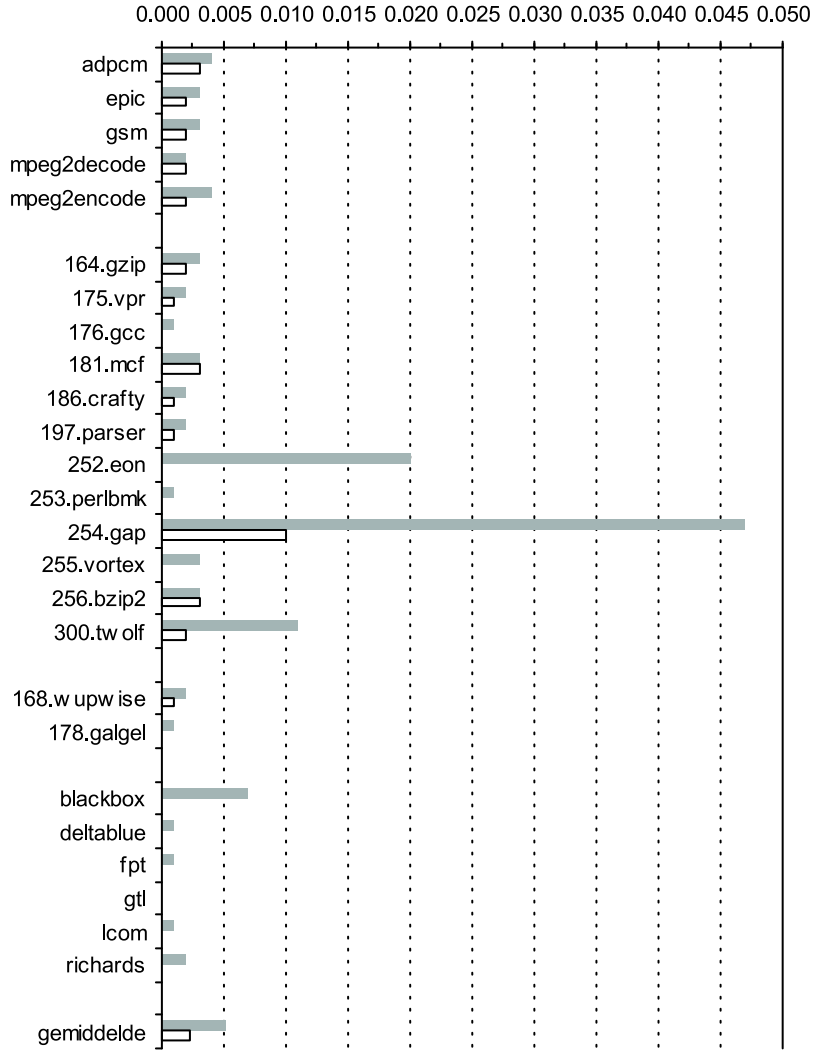
Deze analyse gebruikt evenwel relatief weinig tijd van de volledige compactie, zoals in figuur 5.34 te zien is. Over het algemeen schommelt dit rond de 5% van de totale compactietijd.

Invloed van factorisatie op de uitvoeringssnelheid

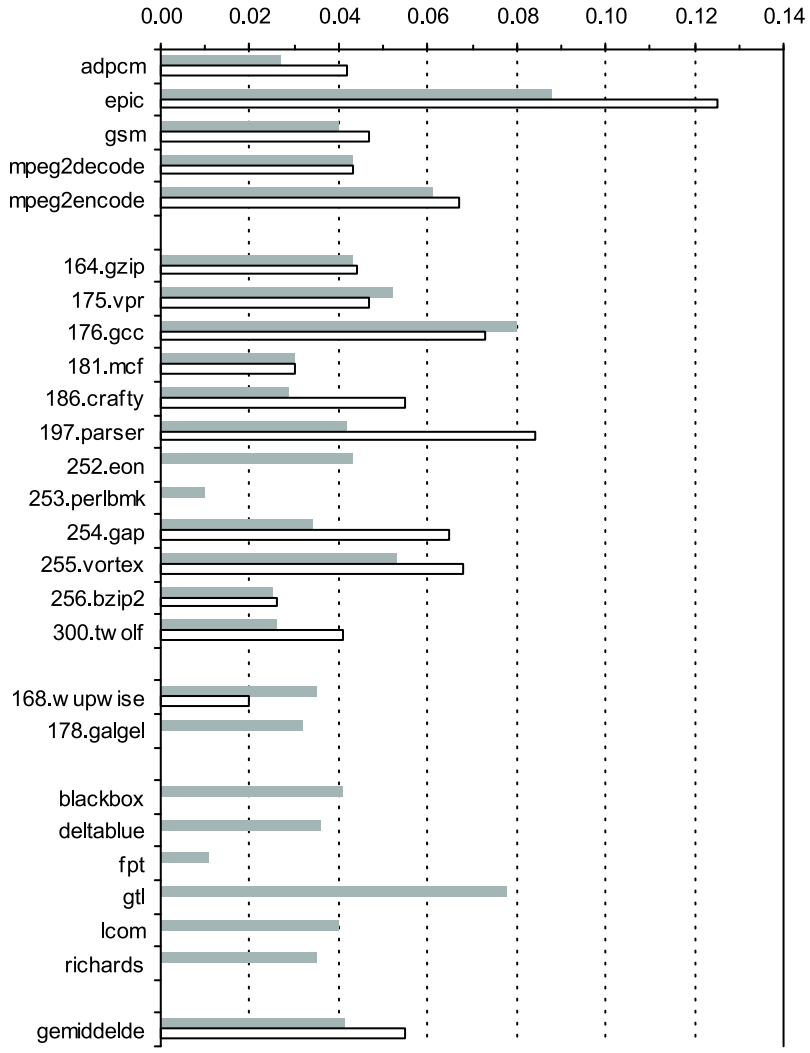
Aangezien factorisatie gepaard gaat met nogal wat nieuw controleverloop en met het kopiëren van registers voor hernoemde blokken, heeft factorisatie een negatieve invloed op de uitvoeringssnelheid van de gecompacteerde programma's. In deze sectie meten we deze negatieve



Figuur 5.32: Verlies aan optimale codecompactie bij het niet hergebruiken van data.



Figuur 5.33: Verlies aan optimale datacompactie bij het niet hergebruiken van data.



Figuur 5.34: Winst aan compactietijd bij het niet hergebruiken van data.

invloed.

Daartoe vergelijken we de uitvoeringssnelheid van vier versies van de SPEC evaluatieprogramma's met elkaar : de basisversie van de programma's, de optimaal gecompacteerde versie, de versie die gecompacteerd is zonder globale factorisatie en tenslotte een versie waarin de globale factorisatie beperkt is tot koude code. In die laatste versie hebben we alle globale factorisatie (behalve op procedures, omdat daar geen extra controleverloop mee gepaard gaat) afgeschakeld voor zogenaamd hete code. Dit is de volgens de profielinformatie meest frequent uitgevoerde code. Om te bepalen welke regio's (d.w.z. de dominator ervan) en basisblokken heet zijn, zijn we als volgt te werk gegaan.

Het totaalgewicht van het programma wordt bepaald als

$$W = \sum_{k \in K} W_k = \sum_{k \in K} I_k \cdot C_k$$

Hierin is I_k het aantal instructies in basisblok k en C_k het aantal keer dat blok k uitgevoerd werd volgens de profielinformatie. W is dus het totaal aantal uitgevoerde instructies tijdens het trainen van het programma. Daarbij gaan we uit van volgens aantal uitvoeringen gesorteerde basisblokken:

$$\forall k, k' : C_k < C'_k \Rightarrow k < k'$$

De hete blokken zijn dan alle blokken $0, ..., i$ waarvoor geldt dat

$$\sum_{k=o}^{k=i} W_k \leq t \cdot W \quad \wedge \quad \sum_{k=o}^{k=i+1} W_k > t \cdot W$$

met t vrij te kiezen. Kiest men $t = 0.1$ dan zijn de hete blokken m.a.w. die blokken die het meest frequent uitgevoerd worden en verantwoordelijk zijn voor 10% van het totale gewicht. Wij hebben $t = 0.6$ gekozen voor de laatste versie van de gecompacteerde programma's. Door t kleiner te maken zal meer (frequent uitgevoerde) code gefactoriseerd worden, door t te vergroten wordt de factorisatie beperkt tot minder (frequent uitgevoerde) code.

In figuren 5.35 en 5.36 hebben we de compactiefactoren voor optimale compactie, compactie zonder globale factorisatie en compactie zonder globale factorisatie op hete code neergezet voor de Compaq resp. Gnu versies van de SPEC evaluatieprogramma's. Men ziet dat

we nauwelijks iets aan compactie moeten inboeten door de globale factorisatie te beperken tot koude code. Zonder globale factorisatie is er, zoals in vorige secties reeds werd besproken, veel minder compactie mogelijk.

In figuren 5.37 en 5.38 hebben we de versnellingsfactoren voor de drie versies van de programma's uitgezet. Men ziet dat het volledige toepassen van globale factorisatie inderdaad de gecompacteerde programma's vertraagt. Als men de globale factorisatie echter beperkt tot koude code, dan kan men een deel van het snelheidsverlies recupereren. Zoals we bij de bespreking van de versnellingsfactoren bij optimale compactie al opgemerkt hebben (zie sectie 5.3.2), moet men de cijfers in deze figuur met een korrel zout nemen omwille van de grote invloed die het gebrek aan ingeroosterde no-ops zowel in positieve als in negatieve zin kan hebben op de versnelling.

We kunnen evenwel besluiten dat de gebruiker van SQUEEZE door t te laten variëren dus meer prioriteit bij compactie dan wel bij programmaversnelling kan leggen. Ook voor het al dan niet inroosteren van no-ops zal dit het geval zijn. Daarvoor hebben we evenwel geen metingen gedaan.

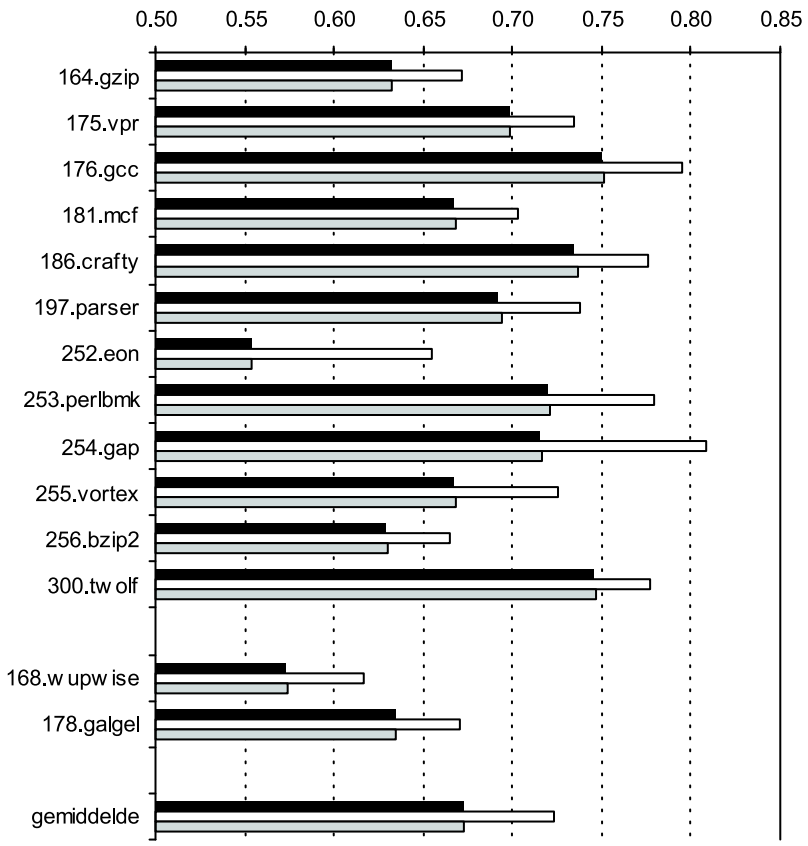
5.3.7 Overzicht bijdragen verschillende technieken

In tabel 5.8 zijn de gemiddelde fracties verlies aan codecompactie door het gebruik van minder geavanceerde varianten van analyses en het afschakelen van compactietechnieken weergegeven. Tevens is aangeduid hoeveel de compactietijd ermee kan ingekort worden. Deze cijfers zijn voor de programmaversies die gegenereerd werden met de Compaq vertalers.

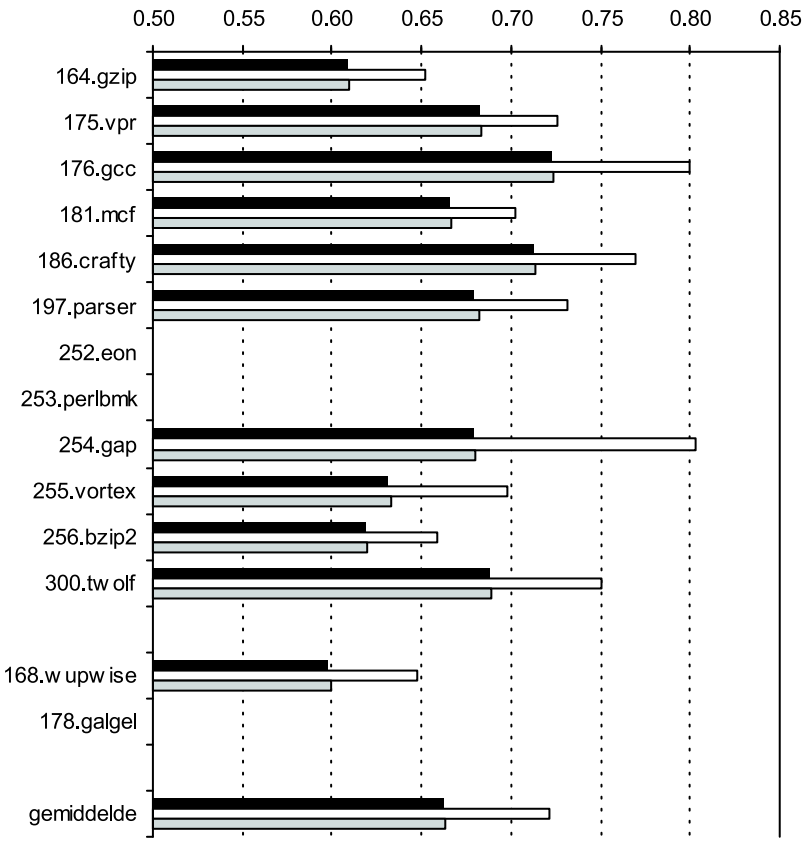
5.3.8 Opsplitsing in applicatie- en bibliotheekcode

Een vraag die zich opdringt is in hoeverre de besproken technieken toegepast kunnen worden op dynamisch gelinkte programma's en wat daar de winst zou kunnen zijn. Daartoe hebben we in SQUEEZE applicatiespecifieke code en bibliotheekcode van elkaar gesplitst als volgt:

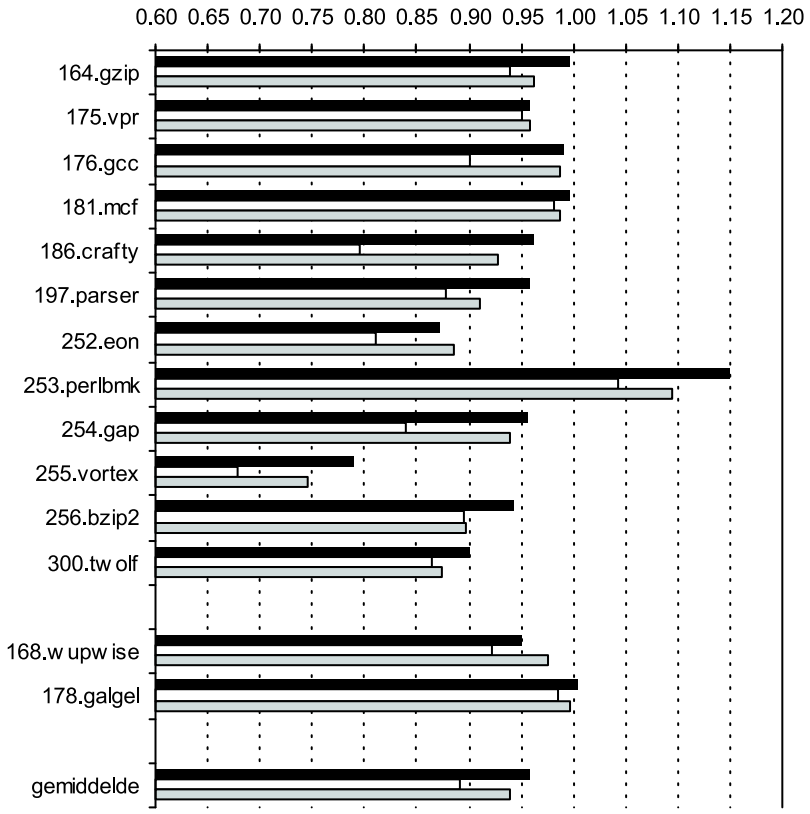
- Aan de hand van de `a.out.map` bestanden worden alle procedures in SQUEEZE opgedeeld in twee groepen: applicatiespecifieke procedures en procedures uit bibliotheken.



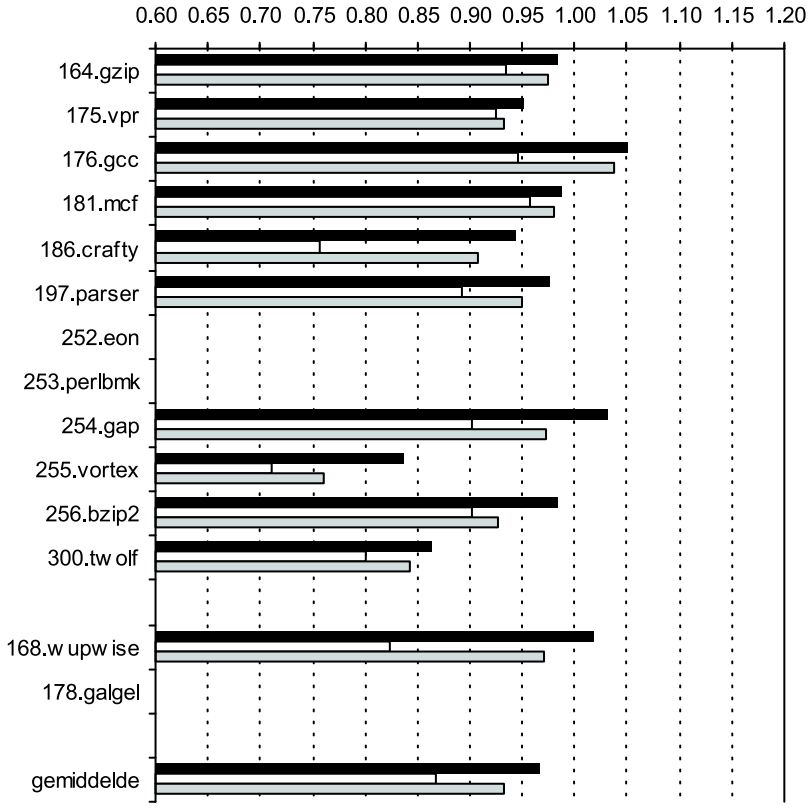
Figuur 5.35: Optimale compactiefactor (zwart), compactiefactor zonder globale factorisatie (wit), compactiefactor zonder globale factorisatie op hete regio's, basisblokken en instructiesequenties (grijs) voor de code in de Compaq versies van de programma's.



Figuur 5.36: Optimale compactiefactor (zwart), compactiefactor zonder globale factorisatie (wit), compactiefactor zonder globale factorisatie op hete regio's, basisblokken en instructiesequenties (grijs) voor de code in de Gnu versies van de programma's.



Figuur 5.37: Versnellingsfactoren bij optimale compactie (zwart), compactie zonder globale factorisatie (wit) en compactie zonder globale factorisatie op hete regio's, basisblokken en instructiesequenties (grijs) voor de Compaq versies van de programma's.



Figuur 5.38: Versnelling bij optimale compactie (zwart), compactie zonder globale factorisatie (wit) en compactie zonder globale factorisatie op hete regio's, basisblokken en instructiesequenties (grijs) voor Gnu versies van de programma's.

variant/techniek	verlies codecompactie	winst compactietijd
Levensduuranalyse		
- Contextongevoelig	9%	21%
- Triviaal	25%	32%
Constantenpropagatie	40%	34%
Detectie van datagebruik	13%	12%
Globale factorisatie	16%	40%
Lokale factorisatie	2.5%	6%

Tabel 5.8: Gemiddelde verlies aan optimale codecompactie en gemiddelde versnelling compactietijd voor diverse varianten van de bespoken analyses en compactietechnieken.

- Het omzetten van indirecte procedure-oproepen in directe oproepen wanneer de opgeroepen procedure bekend is, wordt niet uitgevoerd wanneer de betrokken procedures niet tot dezelfde groep behoren. De oproeppijlen naar de opgeroepenenhelleprocedure worden in dat geval ook niet vervangen door oproeppijlen naar de opgeroepen procedure. Op deze manier worden alle procedure-oproepen in de applicatiespecifieke code conservatief behandeld als oproepen naar onbekende procedures. Er zullen dus ook geen bibliotheekprocedures gesubstitueerd worden in applicatiespecifieke code.

  Er zijn evenwel twee uitzonderingen: een aantal wiskundige bewerkingen zoals de modulobewerking en de deling van gehele getallen worden door de Compaq vertalers geïmplementeerd als oproepen naar bibliotheekcode die de gevraagde berekeningen uitvoert. De vertalers weten welke registers door deze procedures overschreven worden en behandelen in de oproepcontexten alle registers die niet overschreven worden in de opgeroepen procedures als achteraf te bewaren registers. Ze doen dit ondanks het feit dat die procedures nog steeds met indirecte oproepen aangesproken worden. Als we in zulke gevallen geen oproeppijl aanmaken naar de opgeroepen procedure, maar er integendeel een naar de opgeroepenenhelleprocedure laten staan, resulteert dit in een foutieve inschatting van de achteraf te bewaren registers en dus van de registers die dode waarden bevatten voor de oproep. Loze-code-eliminatie gaat dan in de fout. Om dit te vermijden

vervangen we in zulke gevallen dus wel de oproeppijl naar de opgeroepenenhelleprocedure door een oproeppijl naar de betrokken procedure. We maken daarbij gebruik van de symboolinformatie. De betrokken procedures hebben immers allemaal een naam die met "_Ots" begint. Voor de aldus aangemaakt oproeppijlen passen we evenwel geen proceduresubstitutie toe.

De tweede uitzondering is de exit() procedure. Ook hiernaar laten we oproeppijlen aanmaken vanuit de applicatiespecifieke code. Zonder deze pijlen detecteert SQUEEZE niet dat er nooit teruggekeerd wordt uit exit() en wordt de ICVG niet verfijnd met deze kennis. Aangezien deze kennis ook te achterhalen is uit de relocatie- en symboolinformatie corresponderend met de opgeroepen procedure, zou men ook zonder zo'n oproeppijl kunnen achterhalen dat het om een procedure-oproep gaat waarvan nooit teruggekeerd wordt.

- De globale factorisatietechnieken factoriseren enkel code die tot dezelfde groep behoort. Daartoe hebben we bij het op zoek gaan naar identieke of te hernoemen procedures, regio's, basisblokken of instructiesequenties de extra eis gesteld dat ze tot dezelfde groep behoren. De factorisatie van specifieke instructiesequenties wordt twee keer uitgevoerd: eenmaal voor applicatiespecifieke code en eenmaal voor bibliotheekcode. Dit maakt de scheiding compleet.

In figuur 5.39 hebben we de compactiefactor voor applicatiespecifieke code weergegeven voor de verschillende evaluatieprogramma's. Men ziet dat er wel degelijk nog een belangrijke compactie mogelijk is indien men de programma's dynamisch zou linken. Dat de compactiefactoren voor de applicatiespecifieke code van de Gnu versies nu merkelijk kleiner (en dus beter) zijn dan bij de Compaq versies bevestigt nogmaals dat de Gnu vertaler code genereert die beter kan gecompacteerd worden. Bij de gewone versies van SQUEEZE zijn deze verschillen kleiner omdat daar ook de compactie op de bibliotheekcode in rekening wordt gebracht en de bibliotheekcode voor Gnu en Compaq versies van C programma's steeds dezelfde is (op 176.gcc en 168.wupwise na).

Voor het evaluatieprogramma gtl moeten we opmerken dat de GTL bibliotheek in dit geval als applicatiespecifiek wordt beschouwd, aangezien hij zulk een specifieke functionaliteit aanbiedt dat niet te verwachten valt dat hij dynamisch zou meegelinkt worden met programma's. Een heel groot deel van het programma gtl, niet toevallig

dat deel dat sterk gecompacteerd wordt, bestaat uit instantiëringen van sjablonen. Deze worden sowieso aangemaakt van zodra de ".h" bestanden met de interface naar de bibliotheek in de eigenlijk applicatie opgenomen worden, iets wat typisch is voor het gebruik van sjablonen. Het lijkt ons dan ook correct dit deel van het programma als applicatiespecifieke code te beschouwen.

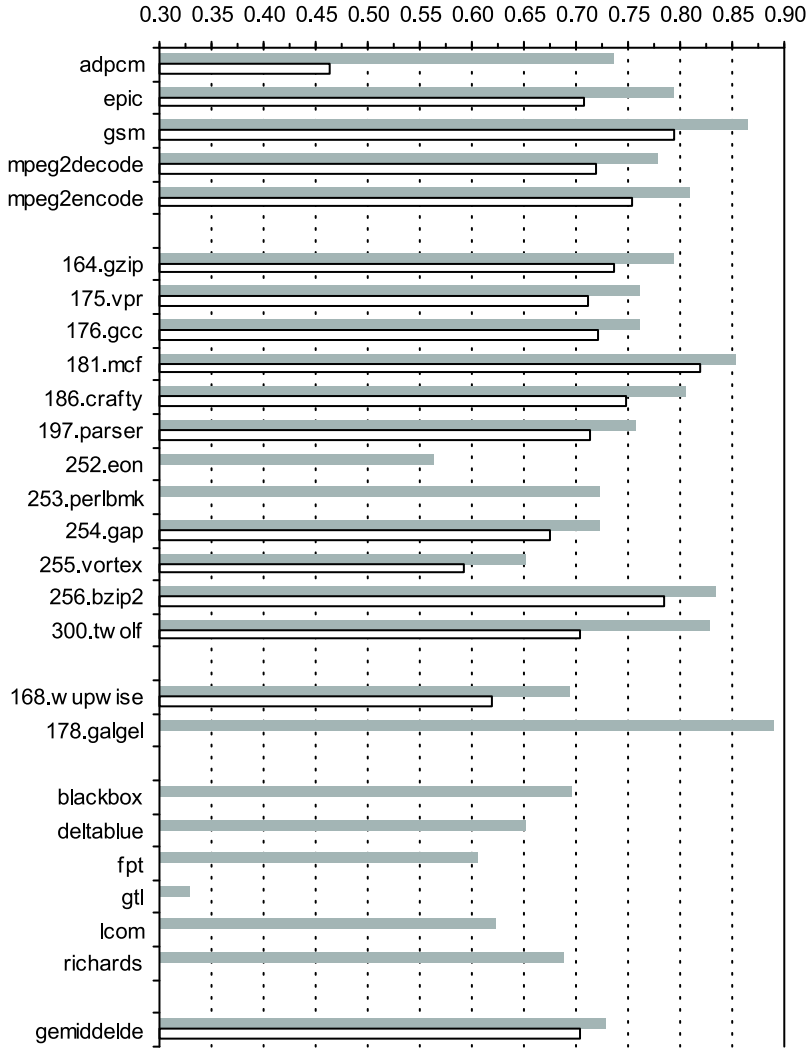
De behaalde codecompactie is ons inziens een goede maat voor de mogelijke codecompactie die op dynamisch gelinkte programma's kan behaald worden. Er zijn drie redenen waarom de getallen in figuur 5.39 toch slechts een schatting zijn:

- De statisch gelinkte code bevat geen stompjes die de oproepen naar dynamisch geladen code implementeren in dynamisch gelinkte programma's. Deze stompjes nemen ofwel echter weinig plaats in in het dynamisch gelinkte programma als er een generieke stompje gebruikt wordt voor alle oproepen (zoals wanneer het eigenlijk stompje in de dynamisch gelinkte bibliotheek zit), of ze zullen uitermate goed factoriseerbaar zijn als er voor elke oproep een apart stompje in de code opgenomen is.

- Het zetten van de globale wijzer na intermodulaire sprongen is ook nu geëlimineerd uit het programma. Bij dynamisch gelinkte programma's zouden we dit enkel kunnen doen bij sprongen die de applicatiespecifieke code niet verlaten. Ons inziens mogen we aannemen dat deze bij doorsnee applicaties in de meerderheid zullen zijn. Zelfs bij IO-gebonden applicaties zal het aantal plaatsen waar bibliotheekcode opgeroepen wordt beperkt zijn, al kunnen deze oproepen natuurlijk heel frequent uitgevoerd worden.

- Voor de wiskundige bewerkingen die door de vertaler geïmplementeerd worden a.d.h.v. oproepen naar door hem bekende procedures en voor de procedure `exit()` hebben we toch een link gelegd tussen applicatiespecifieke en bibliotheekcode. Omdat we er evenwel geen proceduresubstitutie uitvoeren en omdat het per definitie al om zeer sterk geoptimaliseerde code gaat uit de Compaq bibliotheek, kan SQUEEZE uit deze link nauwelijks of geen extra compactiemogelijkheden halen.

We denken dus te mogen besluiten dat de weergegeven cijfers een vrij goede schatting geven van de mogelijkheden van codecompactie na het dynamisch linken van programma's.



Figuur 5.39: Compactiefactoren voor de van bibliotheekcode afgescheiden applicatiecode.

Van de codecompactie voor bibliotheekcode geven we geen cijfers mee. Het zou immers bijzonder moeilijk zijn deze cijfers te interpreteren. Zo blijven er in de speciale versie van SQUEEZE oproeppijlen vanuit de oproepershelleknoop staan naar bibliotheekprocedures die vanuit de applicatiespecifieke code opgeroepen worden. Dit is dan weer niet het geval voor bibliotheekprocedures waarvan het adres in applicatiespecifieke noch in bibliotheekcode gebruikt wordt. Men zou dus kunnen stellen dat de bibliotheekcode gedeeltelijk gespecialiseerd wordt voor de applicatie waaraan ze gelinkt is. Indien men een dynamische bibliotheek volledig zou willen specialiseren voor één applicatie kan men natuurlijk evengoed een statisch gelinkte bibliotheek gebruiken.

5.3.9 Eigen bijdragen van de auteur

Omdat dit proefschrift een idee moet geven van de eigen bijdrage van de auteur, evalueren we tenslotte ons prototype waarin de belangrijkste bijdragen van de auteur afgeschakeld zijn. Dit zijn:

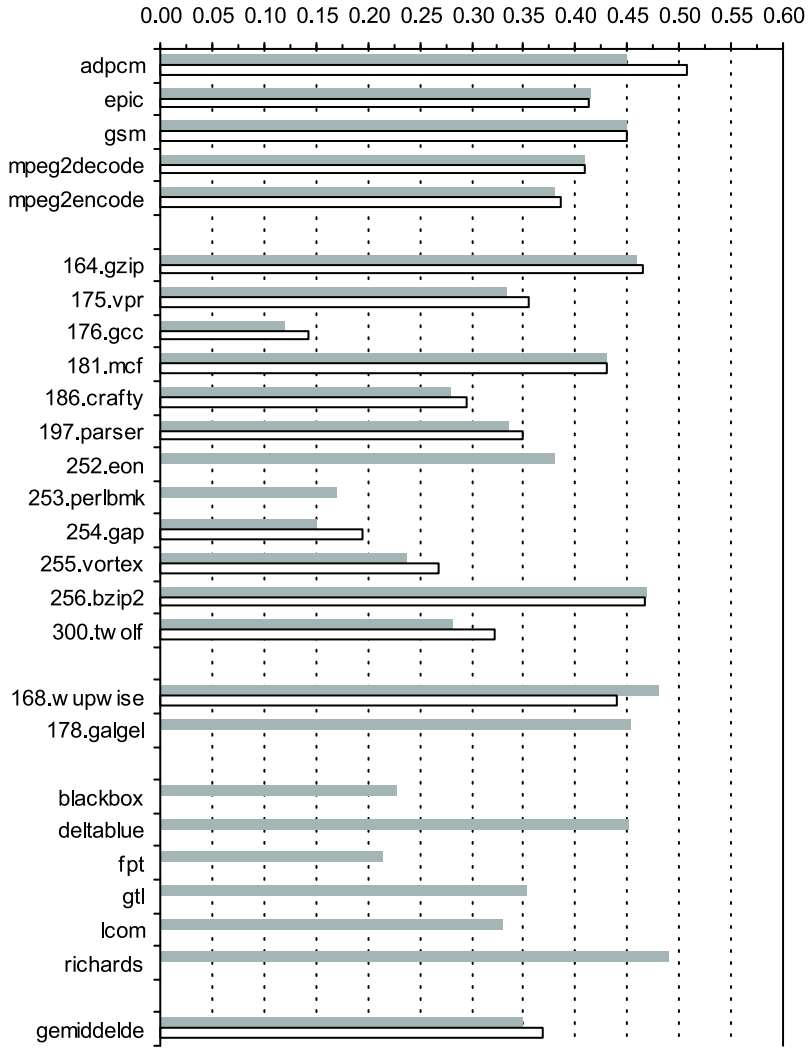
- De analyse van het datagebruik, met de detectie van dode en constante data.
- De verfijningen aan de ICVG die te maken hebben met het weghalen van pijlen vanuit de helleknopen.
- De verfijningen aan de ICVG op basis van het herkennen van exit()-achtige procedures.
- Conditionele constantenpropagatie i.p.v. simpele propagatie.
- Het verfijnen van de analyse van het stapelgedrag aan de hand van pijlen uit de helleprocedures.
- Factorisatie op het niveau van procedures en algemene instructiesequenties.
- Factorisatie door het hergebruiken van data.

In figuur 5.40 is voor alle evaluatieprogramma's de fractie van de optimale codecompactie weergegeven die verloren gaat bij het afzetten van de opgesomde verfijningen en transformaties. Het is met andere woorden de bijdrage van de auteur van dit proefschrift tot de volledige

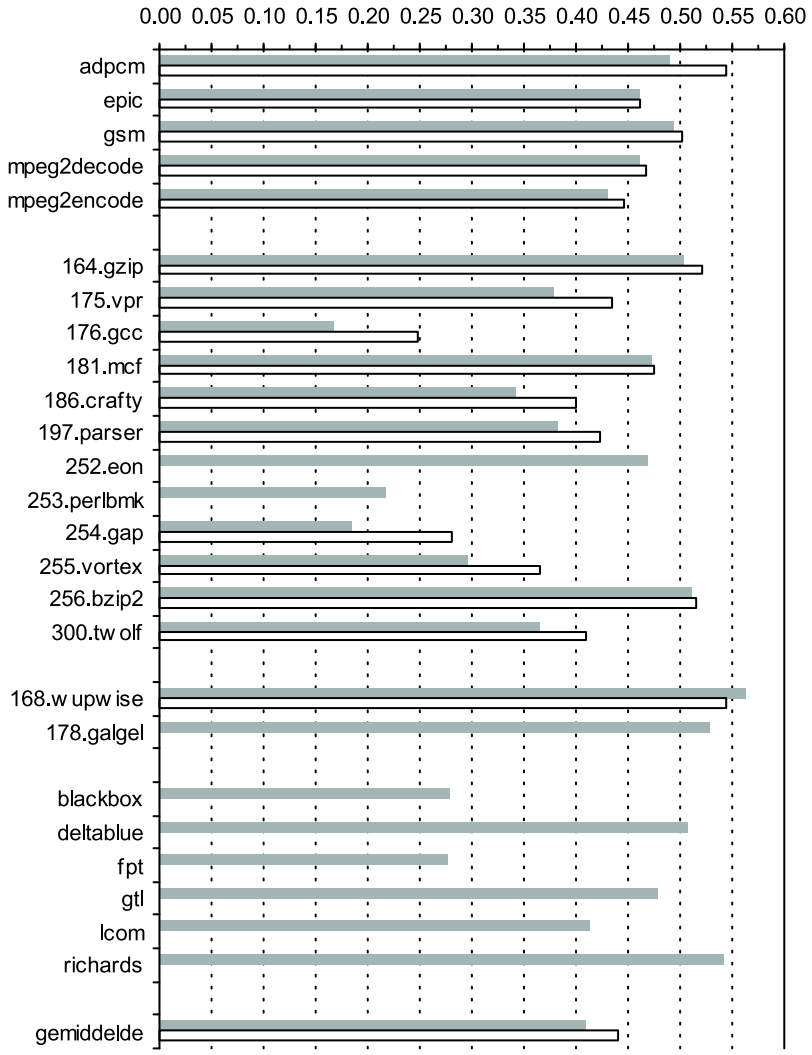
codecompactie. Men ziet dat er gemiddeld ongeveer 1/3 van de totale codecompactie uit bijdragen van dit proefschrift komt. Deze eigen bijdrage weegt het zwaarst door bij de kleinere evaluatieprogramma's, d.i. bij de programma's waarvan slechts een klein deel van de code applicatiespecifiek is. Dit hoeft niet te verwonderen, aangezien de verfijningen van de graaf met het oog op het detecteren van onbereikbare code vooral voor bibliotheekcode nuttig zijn. Van de applicatiespecifieke code valt immers niet te verwachten dat veel code nutteloos in het programma zit. Bij de meegelinkte bibliotheekcode is dit wel het geval.

Zonder de in dit proefschrift beschreven analyse van het gebruik van data is er helemaal geen eliminatie van dode data mogelijk, zodat de bijdrage van in dit proefschrift beschreven analyse aan de totale datacompactie 100% bedraagt.

Kijkt men naar code en data samen, met andere woorden naar de compactie van hele programma's, dan is de eigen bijdrage gemiddeld verantwoordelijk voor meer dan 40% van de bereikte compactie. Deze bijdrage is voor de verschillende evaluatieprogramma's weergegeven in figuur 5.41.



Figuur 5.40: Fractie van codecompactie uit eigen bijdrage.



Figuur 5.41: Fractie van programmacompactie uit eigen bijdrage.

Hoofdstuk 6

Besluit

"Een afgestudeerde denkt alles te weten.
Een meester weet dat hij niets weet.
Een doctor weet dat niemand iets weet."
Vrije vertaling van een Angelsaksische volkswijsheid

"Elk wijsgerig onderzoek eindigt op een verbijsterende onzekerheid."
Arabisch gezegde

"De toekomst wordt bepaald door het herinnerende verleden."
Louis Ferron

6.1 Conclusies van het onderzoek

Met de opkomst van draagbare en ingebedde computersystemen waarin de grootte van het geheugen beperkt is en omwille van het gebruik van moderne ingenieurstechnieken gericht op het hergebruik van code, is er een groeiende vraag naar automatische technieken om kleinere programma's te genereren. Zoals in hoofdstuk 1 geduid werd hebben we ervoor geopteerd om te trachten programma's na het statisch linken te verkleinen.

In hoofdstuk 2 hebben we aangegeven hoe men met de beperkte informatie waarover men na het linken kan beschikken toch een bruikbare voorstelling van een programma kan opbouwen om er compactietechnieken op los te laten. Om deze voorstelling te verfijnen zijn

een aantal belangrijke mogelijkheden besproken. Een belangrijke conclusie die daaruit kan getrokken worden is dat er een aantal complexe analyses zoals levensduuranalyse, constantenpropagatie en de analyse van het gebruik van statisch gealloceerde data nodig zijn op het hele programma om van de initiële, nogal ruwe voorstelling naar een meer verfijnde voorstelling te evolueren.

In hoofdstuk 3 werden deze analyses meer in detail besproken en werden er een aantal al dan niet belangrijk gebleken verfijningen voor gesuggereerd. Bij de bespreking en evaluatie van deze technieken zijn een aantal zaken gebleken:

- De analyses hangen in heel sterke mate van elkaar af om goede resultaten te behalen.
- Men kan voldoende krachtige varianten van de analyses best gebruiken, maar daar hoeft dan ook weer niet in overdreven te worden. Een contextongevoelige constantenpropagatie, waarbij enkel achteraf te bewaren registers contextgevoelig gepropageerd worden, voldoet bv., en een volledig contextgevoelige constantenpropagatie met diepte 1 kan daar nauwelijks iets aan toevoegen.
- Men dient daarbij, net als bij de verfijningen aan de interne voorstelling van het programma, zoveel mogelijk informatie uit de objectbestanden te gebruiken, zoals de wetenschap of een procedure al dan niet geëxporteerd werd uit zijn objectbestand, hoe de finale datasecties samengesteld zijn uit datasecties van objectbestanden, enz.

Factorisatie van code op het assemblerniveau, zoals besproken in hoofdstuk 4, zorgt eveneens voor een aanzienlijke bijdrage. De aangewezen niveaus van granulariteiten om globale factorisatie op toe te passen zijn procedures, basisblokken, specifieke en meer algemene instructiesequenties. Voor de factorisatie van hele basisblokken is het bovendien nuttig registerhernoeming door te voeren met het oog op het creëren van identieke blokken.

Vooral voor programma's waarin veel (nagenoeg) identieke code voorkomt omdat het hergebruik van code gefaciliteerd wordt door de concepten in de (object-georiënteerde) programmeertaal waarin ze geschreven zijn, is codefactorisatie belangrijk. Het parametriseren van nagenoeg identieke procedures als factorisatietechniek voor zulke programma's is beloftevol.

Een uitgebreide evaluatie van de besproken technieken in hoofdstuk 5 bracht aan het licht dat:

- men na het statisch linken wel degelijk programma's significant kan compacteren: gemiddeld kan men 35% code uit de programma's verwijderen en zo'n 12% van de statisch gealloceerde data kan verwijderd worden uit de programma's, wat aanleiding geeft tot een totale gemiddelde compactie van de hele programma's met 30%;

- we over heel sterke indicaties beschikken dat dit, zij het in iets mindere mate, ook het geval zal zijn voor dynamisch gelinkte programma's;

- de compactie van programma's na het linken niet gepaard hoeft te gaan met een tragere uitvoering van de programma's, maar dat men integendeel (en zeker wanneer een kleine fractie van de compactie opgeofferd wordt aan uitvoeringssnelheid) een snelheidswinst kan verwachten;

- aparte vertaling van bronbestanden, gepaard gaand met minder sterke optimalisatie van het programma door de vertalers, niet noodzakelijk tot grotere programma's aanleiding moet geven als er na het linken compactietechnieken worden toegepast.

6.2 Belangrijkste originele bijdragen

De belangrijkste originele bijdragen van de auteur situeren zich op het vlak van de *verfijning van de voorstelling* van te compacteren programma's. Dit gebeurt aan de hand van de *detectie van het gebruik van statisch gealloceerde data* [DS01] en het verregaande gebruik van *alle mogelijke beschikbare informatie* over de programma's [DS00, Debr00].

Met het *elimineren van dode statisch gealloceerde data* is de compactie niet langer beperkt tot codecompactie, maar tot alle onderdelen van een programma. De titel van dit proefschrift lijkt ons dan ook terecht "Compactie van programma's na het linken" te zijn.

Daarnaast zijn tal van verbeteringen aan de bestaande analyses en transformaties toegevoegd, waarbij *globale factorisatie op het niveau van procedures en algemene instructiesequenties* het meest in het oog springen.

Iets meer dan *40% van de totale compactie* die we momenteel bereiken aan de hand van in dit proefschrift en in voorgaande heel verwante publicaties [Muth99, Muth01] besproken technieken komt voort uit de eigen bijdragen.

6.3 Toekomstig werk

Ons inziens komen voortvloeiend uit dit onderzoek vooral volgende richtingen naar voor voor verder onderzoek:

- Men zou nog veel meer moeten trachten alle beschikbare informatie over de te compacteren programma's te benutten. Zo bevatten veel objectformaten informatie over het stapelgedrag van procedures. Door alle analyses en transformaties deze beschikbare informatie te laten benutten en na elke transformatie te laten aanpassen, kan men vermijden zijn toevlucht te moeten nemen tot tijdrovende en in meer conservatieve resultaten resulterende analyses die telkens opnieuw vanaf nul moeten beginnen. Dit geldt overigens niet alleen voor de analyse van het stapelgedrag, maar voor zowat alle analyses en transformaties.

  We hebben hiertoe enigszins een aanzet gegeven waar we oproeppijlen uit de helleknopen laten staan omdat ze ons informatie opleveren over het programma. Eigenlijk is dit een noodoplossing en zou men moeten trachten te komen tot een meer algemene manier van werken, waarin allerlei vormen van informatie bijgehouden kunnen worden.

- Compactie na het linken zou ons inziens sterk gebaat zijn bij extra informatie van de vertalers, bv. wat betreft de oproepgraaf. Analyses op hele programma's op het bronniveau die aangeven welke procedures of methodes kunnen aangesproken worden op verschillende punten in de programma's, kunnen het bv. mogelijk maken om de procedures die nu door één oproepershelleknoop kunnen opgeroepen worden te partitioneren.

- Zoals eerder gesteld werd, is het platform waarvoor we onze technieken geïmplementeerd en geëvalueerd hebben niet het platform waarvoor compacte programma's de belangrijkste doelstelling zijn. Het werk in dit proefschrift moet dan ook herbekeken worden in het kader van een echte ingebedde architectuur.

Vanaf het academiejaar 2001-2002 zijn we, met de hulp van enkele goede laatstejaarsstudenten, begonnen met het implementeren van een compactor voor de ARM (Thumb) architectuur. Onze tot nog toe eerder beperkte ervaringen daarmee geven ons inziens aan dat ook in de ontwikkelomgeving van ARM nog vele mogelijkheden liggen in compactie na het linken, ondanks het feit dat bij het ontwerp van de architectuur, de vertalers en de opbouw van bibliotheken al bijzonder veel aandacht gegaan is naar het genereren van compacte programma's, veel meer dan op het doelplatform van SQUEEZE.

Bijlage A

Formele bespreking van een dekpuntberekening

Bij wijze van voorbeeld gaan we hier iets formeler in op het algoritme voor het iteratief berekenen van het dekpunt van de levensduuranalyse (figuur 3.2). We hebben het algoritme opnieuw in deze sectie opgenomen in figuur A.1.

Voor dit algoritme zullen we nu bewijzen dat het de oplossing berekent van vergelijkingen 3.1 en 3.2. Deze vergelijkingen zijn:

$$\mathrm{Uit_k}[k] = \bigcup_{l \in \mathrm{Opv}[k]} \mathrm{In_k}[l] \tag{A.1}$$

$$\mathrm{In_k}[k] = \mathrm{Cons_k}[k] \cup (\mathrm{Uit_k}[k] \setminus \mathrm{Prod_k}[k]) \tag{A.2}$$

Vooraf passen we het algoritme echter aan, om vlotter de correctheid ervan te kunnen bewijzen. De veranderlijke $\mathrm{In_k}$ is een lokale veranderlijke, die in elke iteratie herbruikt wordt, opdat het algoritme minder geheugen zou verbruiken. Voor het formelere bewijs van het algoritme vervangen we deze veranderlijke door $\mathrm{In_k}[k]$. Dat dit het algoritme zelf niet verandert, is triviaal. Het aangepast algoritme is in figuur A.2 weergegeven.

Merk overigens op dat de keuze van k uit M in een bepaalde iteratie van de while-lus nondeterministisch is, zodat dit een zeer algemeen algoritme is. In hoofdstuk 3 zijn we ingegaan op een selectiestrategie voor het kiezen van k. Met zo'n selectiestrategie wordt enkel de efficiëntie van het algoritme beïnvloed, niet de correctheid ervan.

De initialisatie van dit aangepaste algoritme wordt eveneens aan-

```
1  | M := K+
2  | while (M ≠ ∅)
3  |    M := M \ {k}
4  |    In_k := Cons_k[k] ∪ (Uit_k[k] \ Prod_k[k])
5  |    for all l ∈ Voorg[k]
6  |       OudUit_k := Uit_k[l]
7  |       NieuwUit_k := OudUit_k ∪ In_k
8  |       if (NieuwUit_k ≠ OudUit_k)
9  |          Uit_k[l] := NieuwUit_k
10 |          M := M ∪ {l}
```

Figuur A.1: Iteratief algoritme voor contextongevoelige levensduuranalyse. Hierin is Voorg[k] de verzameling van alle voorgangers van knoop k in de ICVG.

```
1 | M := K+
2 | while (M ≠ ∅)
3 |    M := M \ {k}
4 |    In_k[k] := Cons_k[k] ∪ (Uit_k[k] \ Prod_k[k])
5 |    for all l ∈ Voorg[k]
6 |       if ((Uit_k[l] ∪ In_k[k]) ≠ Uit_k[l])
7 |          Uit_k[l] := Uit_k[l] ∪ In_k[k]
8 |          M := M ∪ {l}
```

Figuur A.2: Aangepast iteratief algoritme voor contextongevoelige levensduuranalyse. Hierin is Voorg[k] de verzameling van alle voorgangers van knoop k in de ICVG.

```
1   do for all k ∈ K+
2     In_k[k] := ∅
3     Uit_k[k] := Cons_k[k] := Prod_k[k] := ∅
4     do for all instructies i in k in omgekeerde volgorde
5       Prod_k[k] := Prod_k[k] ∪ {doeloperandi van i}
6       Cons_k[k] := Cons_k[k] \ {doeloperandi van i}
7       Cons_k[k] := Cons_k[k] ∪ {bronoperandi van i}
8   Cons_k[helleknoop] := R
9   Cons_k[oproepershelleknoop] := R_glob ∪ R_achteraf ∪ R_func
10  Cons_k[opgeroepenenhelleknoop] := R_glob ∪ R_achteraf ∪ R_arg
11  Cons_k[systeemhelleknoop] := R_glob ∪ R_achteraf ∪ R_arg
```

Figuur A.3: Aangepaste initialisatie voor de contextongevoelige levensduuranalyse.

gepast en wordt zoals weergegeven in figuur A.3.

Het bewijs van terminatie van dit algoritme is reeds in sectie 3.1.2 gegeven: het algoritme werkt op een tralie die bestaat uit alle deelverzamelingen van de verzameling van alle registers. Daarbij merken we op:

1. Dit is een tralie met eindige hoogte.
2. De toewijzing op regel 7 komt overeen met een monotone functie, aangezien de verzameling $\text{Uit}_k[l]$ er enkel groeit.
3. Bijgevolg is ook de toewijzing op regel 4 monotoon.
4. Er is een eindig aantal knopen l waarvoor de toewijzing op regel 7 kan uitgevoerd worden. Het aantal knopen in de graaf is immers eindig.
5. Gelet op de conditie in regel 6 en opmerking 2, is de toewijzing op regel 7 strikt monotoon: telkens ze uitgevoerd wordt voor een bepaalde knoop l is de toegewezen waarde anders, en bevat $\text{Uit}_k[l]$ dus meer elementen. Aangezien de tralie een eindige hoogte heeft wordt regel 7 dus slechts een eindig aantal keren toegepast, stel L keren. Er wordt dus maximaal L keren een element aan M toegevoegd.
6. Zijn er K knopen in de graaf, dan zal het algoritme dus zeker eindigen na $K + L$ iteraties, aangezien er per iteratie 1 knoop uit M verwijderd wordt.

Om een correct algoritme te hebben moeten vergelijkingen 3.1 en 3.2 bovendien gelden bij terminatie. Wisselen we in vergelijking 3.1 de vrije veranderlijken k en l door i en j, dan kunnen we stellen dat het algoritme correct is als bij terminatie geldt:

$$\forall i \in K^+ \quad : \quad \mathrm{Uit_k}[i] = \bigcup_{j \in \mathrm{Opv}[i]} \mathrm{In_k}[j] \tag{A.3}$$

$$\forall k \in K^+ \quad : \quad \mathrm{In_k}[k] = \mathrm{Cons_k}[k] \cup (\mathrm{Uit_k}[k] \setminus \mathrm{Prod_k}[k]) \tag{A.4}$$

Aangezien bij (de gegarandeerde) terminatie M leeg is, kunnen we deze predicaten als volgt herschrijven:

$$\forall i \in K^+ : i \notin M \quad \Rightarrow \quad \mathrm{Uit_k}[i] = \bigcup_{j \in \mathrm{Opv}[i]} \mathrm{In_k}[j] \tag{A.5}$$

$$\forall k \in K^+ : k \notin M \quad \Rightarrow \quad \mathrm{In_k}[k] = \mathrm{Cons_k}[k] \cup (\mathrm{Uit_k}[k] \setminus \mathrm{Prod_k}[k]) \tag{A.6}$$

We zullen nu aantonen dat deze predicaten A.5 en A.6 steeds geldig zijn aan het begin en aan het einde van een iteratie van de while-lus. Deze predicaten zijn m.a.w. de *invarianten* van de lus. We bewijzen dit met inductie. Bij het binnenkomen van de lus in de eerste iteratie gelden beide predicaten overigens triviaal, aangezien alle knopen in M zitten.

We bewijzen nu dat als de invarianten waar zijn aan het begin van iteratie, ze ook waar zijn op het einde van die iteratie, d.w.z. aan het begin van de volgende iteratie of bij terminatie.

Stel dat het predikaat A.6 waar is aan het begin van een iteratie. Op regel 4 wordt $\mathrm{In_k}[k]$ gewijzigd. De waarde die eraan toegekend wordt zorgt ervoor dat na regel 4 predikaat A.6 nog steeds waar is. De enige manier waarop dit predikaat vervolgens onwaar kan worden is door de toewijzing op regel 7, als daar geldt dat $l = k$. In dat geval zal omwille van de toewijzing op regel 8 echter gelden dat $k \in M$ op het einde van de iteratie, waardoor predikaat A.6 opnieuw triviaal waar wordt.

Veronderstel nu dat predikaat A.5 waar is bij het begin van een iteratie. Opnieuw zullen alle knopen l waarvoor de toewijzing op regel 7 uitgevoerd wordt tot M behoren na de iteratie. We moeten dus enkel bewijzen dat het predikaat geldig blijft voor knopen l waarvoor $\mathrm{Uit_k}[l]$ onveranderd blijft. Merk vooreerst op dat indien dit het geval is geldt dat

$$(\mathrm{Uit_k}[l] \cup \mathrm{In'_k}[k]) = \mathrm{Uit_k}[l] \tag{A.7}$$

of m.a.w.

$$\mathrm{In}'_{\mathrm{k}}[k] \subset \mathrm{Uit}_{\mathrm{k}}[l] \tag{A.8}$$

omwille van regel 6 en 7, waarbij $\mathrm{In}'_{\mathrm{k}}[k]$ de in regel 4 toegewezen waarde is.

Het is tevens triviaal om in te zien dat voor zulke knopen l waarvan k geen opvolger is, het predikaat eveneens blijft gelden: gedurende de iteratie is er dan immers niks veranderd aan $\mathrm{Uit}_{\mathrm{k}}[l]$ en er is niks veranderd aan $\bigcup_{k \in \mathrm{Opv}[l]} \mathrm{In}_{\mathrm{k}}[k]$. De overblijvende verzameling knopen waarvoor we het predikaat dus moeten bewijzen is

$$B = Voorg[k] \setminus M \tag{A.9}$$

Wat dus te bewijzen blijft is dat na de iteratie geldt dat

$$\forall i \in B : \mathrm{Uit}_{\mathrm{k}}[i] = \bigcup_{j \in \mathrm{Opv}[i]} \mathrm{In}_{\mathrm{k}}[j] \tag{A.10}$$

We kunnen dit herschrijven als

$$\forall i \in B : \mathrm{Uit}_{\mathrm{k}}[i] = \Big(\bigcup_{j \in (\mathrm{Opv}[i] \setminus \{k\})} \mathrm{In}_{\mathrm{k}}[j] \Big) \cup \mathrm{In}'_{\mathrm{k}}[k] \tag{A.11}$$

We weten dat de toewijzing op regel 4 een monotone toewijzing is. Bijgevolg bestaat er een verzameling X waarvoor geldt dat $\mathrm{In}'_{\mathrm{k}}[k] = \mathrm{In}_{\mathrm{k}}[k] \cup X$. Het predikaat wordt dan

$$\forall i \in B : \mathrm{Uit}_{\mathrm{k}}[i] = \Big(\bigcup_{j \in (\mathrm{Opv}[i] \setminus \{k\})} \mathrm{In}_{\mathrm{k}}[j] \Big) \cup \mathrm{In}_{\mathrm{k}}[k] \cup X \tag{A.12}$$

Aangezien de invariant geldig was voor de iteratie, wordt dit

$$\forall i \in B : \mathrm{Uit}_{\mathrm{k}}[i] = \mathrm{Uit}_{\mathrm{k}}[i] \cup X \tag{A.13}$$

Gelet op A.8 moet ook gelden dat $X \subset \mathrm{Uit}_{\mathrm{k}}[i]$ en dus kunnen we het predikaat tenslotte herschrijven als

$$\forall i \in B : \mathrm{Uit}_{\mathrm{k}}[i] = \mathrm{Uit}_{\mathrm{k}}[i] \tag{A.14}$$

wat triviaal waar is.

Bibliografie

[Ages94] O. AGESEN EN D. UNGAR. Sifting Out the Gold: Delivering Compact Applications From an Exploratory Object-Oriented Environment. In *Proceedings of the ACM OOPSLA'94 Conference on Object-Oriented Programming Systems, Languages and Applications*, blzn. 355–370, Oktober 1994.

[Aho86] A. AHO, R. SETHI, EN J. ULLMAN. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

[Alpa] Alpha Migration Tools, Freeport Express. http://www.support.compaq.com/amt/freeport/.

[Alpb] Alpha Migration Tools, DECmigrate. http://www.support.compaq.com/amt/decmigrate/index.html/.

[ARM95] ARM. *An Introduction to Thumb*. Advanced RISC Machines Ltd., Maart 1995.

[AT98] A. ADL-TABATABAI, M. CIERNIAK, C. LUEH, V. PARIKH, EN J. STICHNOTH. Fast, Effective Code Generation in a Just-in-Time Java Compiler. *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, blzn. 280–290, 1998.

[Autr94] T. AUTREY. Demand-Driven Interprocedural Constant Propagation Implementation and Evaluation. Technisch rapport, Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, Mei 1994. Voorlopig rapport.

[Bake93] B. BAKER. A theory of parameterized pattern matching: Algorithms and applications (extended abstract). In *Pro-*

ceedings of the 25th Annual ACM Symposium on the Theory of Computing, blzn. 71–80, Mei 1993.

[Bene97] M. BENES, A. WOLFE, EN S. NOWICK. A High-Speed Asynchronous Decompression Circuit for Embedded Processors. In *Proceedings of the 17th Conference on Advanced Research in VLSI*, September 1997.

[Bene98] M. BENES, S. NOWICK, EN A. WOLFE. A Fast Asynchronous Huffman Decoder for Compressed-Code Embedded Processors. In *Proceedings of the 4th IEEE Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, September 1998.

[Boeh98] H. BOEHM EN M. WEISER. Garbage Collection in an uncooperative environment. *Software – Practice and Experience*, 18(9):807–820, September 1998.

[Brad98] Q. BRADLEY, R. HORSPOOL, EN J. VITEK. JAZZ: An efficient compressed format for Java archive files. In *Proceedings of CASCON'98*, blzn. 294–302, November 1998.

[Bray84] G. BRAY. Sharing Code Among Instances of Ada Generics. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction*, blzn. 276–284, Juni 1984.

[Bund92] J. BUNDA, D. FUSSELL, R. JENEVEIN, EN W. ATHAS. 16-bit Vs. 32-bit Instructions for Pipelined Microprocessors. Technical Report TR-92-39, The University of Texas at Austin, Department of Computer Sciences, 1992.

[Call86] D. CALLAHAN, K. COOPER, K. KENNEDY, EN L. TOREZON. Interprocedural Constant Propagation. In *Proceedings of the ACM SIGPLAN '86 conference on Compiler Construction*, blzn. 152–162, 1986.

[Cari95] P. CARINI EN M. HIND. Flow-Sensitive Interprocedural Constant Propagation. In *Proceedings of the ACM SIGPLAN '95 conference on Programming Language, Design and Implementation*, blzn. 23–31, 1995.

[Cham96] G. CHAMBERS, J. DEAN, EN D. GROVE. Whole-Program Optimization of Object-Oriented Languages. Technisch

Rapport 96-06-02, Department of Computer Science and Engineering, University Of Washington, Juni 1996.

[Cher98] A. CHERNOFF, M. HERDEG, R. HOOKWAY, C. REEVE, N. RUBIN, T. TYE, S. YADAVALLI, EN J. YATES. FX!32 A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, Maart-April 1998.

[Cifu00] C. CIFUENTES EN M. VAN EMMERIK. UQBT: Adaptable Binary Translation at Low Cost. *IEEE Computer*, 33(3):60–66, Maart 2000.

[Clau00] L. CLAUSEN, U. SCHULTZ, C. CONSEL, EN G. MULLER. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, 2000.

[Clic95] C. CLICK EN K. COOPER. Combining Analyses, Combining Optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, Maart 1995.

[Cohn97] R. COHN, D. GOODWIN, P. LOWNEY, EN N. RUBIN. Spike: An Optimizer for Alpha/NT Executables. In *USENIX Windows NT Workshop*, Augustus 1997.

[Com98] Compaq Computer Corporation. *Alpha Architecture Handbook*, Oktober 1998.

[Com00] Compaq Computer Corporation. *Tru64 UNIX Object File and Symbol Table Format Specification*, Augustus 2000.

[Coop99] K. COOPER EN N. MCINTOSH. Enhanced Code Compression for Embedded RISC Processors. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, blzn. 139–149, 1999.

[Cous77] P. COUSOT EN R. COUSOT. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, blzn. 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[Deb] Persoonlijke communicatie met prof. Debray.

[Debr98] S. DEBRAY, R. MUTH, EN M. WEIPPERT. Alias Analysis of Executable Code. In *Symposium on Principles of Programming Languages*, blzn. 12–24, 1998.

[Debr00] S. DEBRAY, W. EVANS, R. MUTH, EN B. SUTTER. Compiler Techniques for Code Compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

[Debr01a] S. DEBRAY EN W. EVANS. Profile-Guided Code Compression. Manuscript, 2001.

[Debr01b] S. DEBRAY, R. MUTH, EN S. WATTERSON. Software Power Optimization via Post-Link-Time Binary Rewriting. http://www.cs.arizona.edu/alto, 2001.

[dEUS94] DEUS. worst case scenario. Island Records Ltd., 1994.

[Dron99] P. DRONGOWSKI, D. HUNTER, M. FAYYAZI, D. KAELI, EN C. J.. Studying the Performance of the FX!32 Binary Translation System. In *proceedings of the Binary Translation Workshop*, 1999.

[DS00] B. DE SUTTER, B. DE BUS, K. DE BOSSCHERE, P. KEYNGNAERT, EN B. DEMOEN. On the Static Analysis of Indirect Control Transfers in Binaries. In *Proceedings of the 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, blzn. 1013–1019, Juni 2000.

[DS01] B. DE SUTTER, B. DE BUS, K. DE BOSSCHERE, EN S. DEBRAY. Combining Global Code and Data Compaction. In *Proceedings ACM SIGPLAN 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems*, blzn. 29–38, Juni 2001.

[Dyn] Transitives, Dynamite. http://www.transitives.com/.

[Erns97] J. ERNST, C. FRASER, W. EVANS, S. LUCCO, EN T. PROEBSTING. Code Compression. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, blzn. 358–365, Juni 1997.

[Fern96] M. FERNÁNDEZ. *A Retargetable, Optimizing Linker*. Doctoraatsproefschrift, Princeton University, January 1996.

[Fern99] M. FERNÁNDEZ EN R. ESPASA. Dixie: A Retargetable Binary Instrumentation Tool. In *Proceedings of the Workshop on Binary Translation (WBT-1999)*, 1999.

[Fran94] M. FRANZ. *Code-Generation On-The-Fly: A Key to Portable Software*. Doctoraatsproefschrift, Swiss Federal Institute of Technology, 1994.

[Fran97] M. FRANZ EN T. KISTLER. Slim Binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[Fras84] C. FRASER, E. MYERS, EN A. WENDT. Analyzing and Compressing Assembly Code. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, volume 19, blzn. 117–121, Juni 1984.

[Fras91] C. FRASER EN D. HANSON. A Retargetable Compiler for ANSI C. Technisch Rapport CS–TR–303–91, Princeton University, Princeton, N.J., 1991.

[Fras95] C. FRASER EN T. PROEBSTING. Custom Instruction Sets For Code Compression. Technisch rapport, Microsoft Research, 1995. http://research.microsoft.com/~toddpro/papers/pldi2.ps.

[Game98] M. GAME EN A. BOOKER. *CodePack: Code Compression for PowerPC Processors*. International Business Machines (IBM) Corporation, 1998.

[GH00] E. GADI HABER EN V. EISENBERG. Reliable Post-link Optimizations based on Partial Information. In *Proceedings of the 3th ACM Feedback Directed Optimizations Workshop*, blzn. 91–100, December 2000.

[Ghiy01] R. GHIYA, D. LAVERY, EN D. SEHR. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, blzn. 47–58, Juni 2001.

[Good97] D. GOODWIN. Interprocedural Dataflow Analysis in an Executable Optimizer. In *Proceedings of the ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, blzn. 122–133, Juni 1997.

[Grov93] D. GROVE EN T. L.. Interprocedural Constant Propagation: A Study of Jump Function Implementations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, blzn. 90–99, Juni 1993.

[Hank97] A. HANKERSSON, G. HARRIS, EN P. JOHNSON. *Introduction to Information Theory and Data Compression*. CRC Press, 1997.

[Henn90] J. HENNESSY EN D. PATTERSON. *Computer Architecture - A Quantitive Approach*. Morgan Kaufmann Publishers Inc., 1990.

[Hoog99] J. HOOGERBRUGGE, L. AUGUSTEIJN, J. TRUM, EN R. VAN DE WIEL. A Code Compression System Based on Pipelined Interpreters. *Software – Practice and Experience*, 29(11):1005–1023, 1999.

[Hook97] R. HOOKWAY EN M. HERDEG. Digital FX!32: Combining Emulation and Binary Translation. *Digital Technical Journal*, 9(1):3–12, 1997.

[Hors98] R. HORSPOOL EN J. CORLESS. Tailored Compression of Java Class Files. *Software – Practice and Experience*, 28(12):1253–1268, 1998.

[Huff52] D. HUFFMAN. A Method for Construction of Minimum Redundancy Codes. In *Proceedings of the Institute of Radio Engineers*, volume 40, blzn. 1098–1101, September 1952.

[IBM98] IBM. *CodePack: PowerPC Code Compression Utility User's Manual. Version 3.0*. International Business Machines (IBM) Corporation, 1998.

[Int99] Intel Corporation. *Intel Architecture Software Developer's Manual, Volumes 1,2,3*, 1999.

[Jone93] N. JONES, C. GOMARD, EN P. SESTOFT. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993.

[Kemp98] T. KEMP, R. MONTOYE, J. HARPER, J. PALMER, EN D. AUERBACH. A Decompression Core for PowerPC. *IBM Journal of Research and Development*, 42(6), Nov 1998.

[Kiro97] D. KIROVSKI, J. KIN, EN W. MANGIONE-SMITH. Procedure Based Program Compression. In *Proceedings of the 30th Annual International Symposium on Microarchitecture (MICRO-30)*, December 1997.

[Kiss97] K. KISSELL. *MIPS16: High-density MIPS for the Embedded Market*. Silicon Graphics MIPS Group, 1997.

[Kozu94] M. KOZUCH EN A. WOLFE. Compression of Embedded System Programs. In *Proceedings of the International Conference on Computer Design*, 1994.

[Land91] W. LANDI EN B. RYDER. Pointer-induced aliasing: A problem taxonomy. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, blzn. 93–103, Orlando, FL, USA, Jan 1991. ACM Press.

[Lari99] S. LARIN EN T. CONTE. Compiler-driven cached code compression schemes for embedded ILP processors. In *Proceedings of the 32th International Symposium on Microarchitecture (MICRO-32)*, November 1999.

[Laru95] J. LARUS EN E. SCHNARR. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, blzn. 291–300, Juni 1995.

[Laru96] J. LARUS. EEL Guts: Using the EEL Executable Editing Library. Technisch rapport, Computer Science Department, University Of Wisconsin - Madison, 1996.

[Lefu00] C. LEFURGY. *Efficient Execution of Compressed Programs*. Doctoraatsproefschrift, University of Michigan, Juni 2000.

[Levi00] J. LEVINE. *Linkers and Loaders*. Morgan Kaufman, 2000.

[Lind99] T. LINDHOLM EN F. YELLIN. *The Java (tm) Virtual Machine Specification*. Addison-Wesley, 1999.

[Marl95] T. MARLOWE, B. RYDER, EN M. BURKE. Defining Flow Sensitivity in Data Flow Problems. Technisch Rapport LCSR-TR-249, Rutgers University, 1995.

[Metz93] R. METZGER EN S. STROUD. Interprocedural Constant Propagation: An Empirical Study. *ACM Letters on Programming Languages and Systems*, 2(1–4):213–232, Maart-December 1993.

[Much97] S. MUCHNICK. *Advanced Compiler Design & Implementation*. Morgan Kaufmann Publishers, 1997.

[Muth99] R. MUTH. *Alto: A Platform for Object Code Modification*. Doctoraatsproefschrift, University Of Arizona, 1999.

[Muth01] R. MUTH, S. DEBRAY, S. WATTERSON, EN K. BOSSCHERE. alto : A Link-Time Optimizer for the Compaq Alpha. *Software – Practice and Experience*, 31:67–101, Januari 2001.

[Pett90] K. PETTIS EN R. HANSEN. Profile-guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, blzn. 16–27, 1990.

[Proe95] T. PROEBSTING. Optimizing a ANSI C Interpreter with Superoperators. In *Proceedings of the ACM POPL '95 Conference on Principles of Programming Languages*, blzn. 322–332, 1995.

[Pugh99] W. PUGH. Compressing Java Class Files. In *Proceedings of the ACM SIGPLAN'99 Conference on Programming Language Design and Implementation*, blzn. 247–258, Mei 1999.

[Rome97] T. ROMER, G. VOELKER, D. LEE, A. WOLMAN, W. WONG, H. LEVY, B. BERSHAD, EN J. CHEN. Instrumentation and Optimization of Win32/Intel Executables Using Etch. In *USENIX Windows NT Workshop*, blzn. 1–8, 1997.

[Rons98] M. RONSSE EN K. DE BOSSCHERE. JiTI: Tracing Memory References for Data Race Detection. *Parallel Computing: Fundamentals, Applications and New Directions*, 12:327–334, Feb 1998.

[Rons99] M. RONSSE EN K. DE BOSSCHERE. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems*, 17(3):133–152, Mei 1999.

[Rons00] M. RONSSE EN K. DE BOSSCHERE. JiTI: A Robust Just in Time Instrumentation Technique. In *Proceedings of Workshop on Binary Translation - 2000*, blzn. 43–54, Maart 2000.

[Rose83] J. ROSENBERG. *Generating Compact Code for Generic Subprograms*. Doctoraatsproefschrift, Carnegie-Mellon University, 1983.

[Sagi95] M. SAGIV, T. REPS, EN S. HORWITZ. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. Technisch rapport, Computer Science Department, University of Wisconsin - Madison, 1995.

[Site92] R. SITES, A. CHERNOFF, M. KIRK, M. MARKS, EN S. ROBINSON. Binary translation. *Digital Technical Journal of Digital Equipment Corporation*, 4(4):137–152, Herfst 1992.

[Sriv93] A. SRIVASTAVA EN D. WALL. A Practical System for Intermodule Code Optimization at Link-Time. *Journal of programming Languages*, blzn. 1–18, Maart 1993. Ook beschikbaar als WRL Research Report 92/06.

[Sriv94a] A. SRIVASTAVA EN A. EUSTACE. ATOM: A System For Building Customized Program Analysis Tools. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, blzn. 196–205, Juni 1994. Also available as WRL Research Report 92/06.

[Sriv94b] A. SRIVASTAVA EN D. WALL. Link-time optimization of address calculation on a 64-bit architecture. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, blzn. 49–60, Juni 1994. Also available as WRL Research Report 94/1.

[Ston75] H. STONE EN D. SIEWOREK. *Introduction to Computer Organization and Data Structures: PDP-11 Edition*. McGraw-Hill, 1975.

[Swee98] P. SWEENEY. EN F. TIP. A Study of Dead Data Members in C++ Applications. In *Proceedings of the ACM SIGPLAN '98 conference on Programming Language Design and Implementation*, blzn. 324–323, Juni 1998.

[Tip99] F. TIP, C. LAFFRA, EN P. SWEENEY. Practical Experience with an Application Extractor for Java. In *Proceedings of the ACM OOPSLA'99 Conference on Object-Oriented Programming Systems, Languages and Applications*, blzn. 292–305, November 1999.

[Tip00] F. TIP EN J. PALSBERG. Scalable Propagation-Based Call Graph Construction Algorithms. In *Proceedings of the ACM OOPSLA'00 Conference on Object-Oriented Programming Systems, Languages and Applications*, blzn. 281–293, Oktober 2000.

[TM] *TriMedia Software Documentation, Book 4: Software Tools, Program Development Tools*.

[Tri00] TriMedia Technologies Inc. *TriMedia32 Architecture*, Oktober 2000.

[Turl95] J. TURLEY. Thumb Squeezes ARM Code Size. *Microprocessor Report*, 9(4):1–5, Maart 1995.

[Ung00] D. UNG EN C. CIFUENTES. Machine-Adaptable Dynamic Binary Translation. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization*, blzn. 37–47, 2000.

[Vasa00] B. VASANTH, E. DUESTERWALD, EN S. BANEJIA. Dynamo: A Transparent Dynamic Optimization System. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, blzn. 1–12, 2000.

[Wall86] D. WALL. Global Register Allocation at Link Time. In *Proceedings of the ACM SIGPLAN '86 conference on Compiler Construction*, blzn. 264–275, 1986.

[Wall92] D. WALL. Systems for Late Code Modification. Technisch Rapport WRL-92/3, Digital Western Research Laboratory, Mei 1992.

[Watt01] S. WATTERSON EN S. DEBRAY. Goal-Directed Value Profiling. In *Proceedings of the 2001 International Conference on Compiler Construction (CC 2001)*, blzn. 319–333, 2001.

[Weav94] D. Weaver en T. Germond, redactie. *The SPARC Architecture Manual, version 9*. PTR Prentice Hall, 113 Sylvan Avenue, Englewood Cliffs, New Jersey 07632, USA, 1994.

[Wegm91] M. WEGMAN EN F. ZADECK. Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

[Weis84] M. WEISER. Program slicing. *IEEE - Transactions on Software Engineering*, 10(4):352–357, 1984.

[Wils95] R. WILSON EN M. LAM. Efficient context-sensitive pointer analysis for C programs. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation.*, 30(6):1–12, 1995.

[Wolf92] A. WOLFE EN A. CHANIN. Executing Compressed Programs on an Embedded RISC Architecture. In *Proceedings of the 25th International Symposium on Microarchitecture (MICRO-25)*, blzn. 81–91, 1992.

[Yang99] B. YANG, S. MOON, S. PARK, J. LEE, S. LEE, J. PARK, Y. CHUNG, S. KIM, K. EBCIOGLU, EN E. ALTMAN. LaTTe: A Java VM Just-In-Time Compiler with Fast and Efficient Register Allocation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT-99)*, blzn. 128–138, 1999.

[Zast95] M. ZASTRE. Compacting Object Code via Parameterized Procedural Abstraction. Thesis, University of Victoria, 1995.