Jämförelse av registerallokeringsstrategier för programmeringsspråket PLEX

Peter Karlsson, peter.karlsson@mds.mdh.se
Stefan Ohlsson, stefan.ohlsson@mds.mdh.se

Januari 2000

Redovisning av examensarbete på 20 poäng inom ämnena datateknik respektive datalogi

Sammanfattning

PLEX är ett realtidsspråk som används för programmering av telefonväxlar vid Ericsson. Kompilatorn för språket skrevs redan på 1980-talet, och även registerallokeringsdelen är från den tiden. Det har nu forskats fram bättre sätt att allokera register, varför det ansågs vara dags att byta ut den gamla algoritmen. Denna rapport tar upp och studerar ett antal olika registerallokeringsalgoritmer som kan passa, beskriver hur bytet utförts, och jämför hur bra den nya algoritmen fungerar gentemot gamla.

Innehåll

Kapitel 1

Förutsättningar

Denna rapport avser ett examensarbete på D-nivå, vilket omfattar 20 poäng inom ämnena datateknik (Peter Karlsson) respektive datalogi (Stefan Ohlsson).

1.1 Exjobbsdefinition

Så här ser definitionen på det exjobb vi genomfört ut:

> PLEX är ett realtidsspråk som används för programmering av telefonisystem vid Ericsson. Språket är baserat på en signalparadigm och och kan väldigt effektivt behandla tusentals pågående "processer." Effektivitet är dock alltid viktigt och fortsatta förbättringar av kompilatorn för att den ska producera bättre kod är högprioriterat. En global registerallokeringsteknik kallad "sekventiell graffärgning med grafreduktion" används för närvarande i PLEX-kompilatorn. En ny algoritm kallad "fat cover" är enligt forskning i medel bättre än den sekventiella graffärgningsalgoritmen.

1.2 Bakgrund

Kompilatorer som de skrivs idag utför sitt arbete i flera steg. Först tolkas den programkod som programmeraren har skrivit, samtidigt som en kontroll sker att rätt syntax har använts. Denna tolkning ger en internrepresentation av programmet som det är lättare att arbeta på i andra delar av kompilatorn. På internformen utförs sedan ett flertal olika analyser för att bestämma vilka optimeringar som skall göras, och vilken maskinkod som ska genereras för målsystemet. Precis före kodgenereringen görs ett registerallokeringssteg där det bestäms vilket maskinregister (om något) som ska användas för varje variabel och konstant [1].

1.3 Mål med exjobbet

Målet är att byta ut den algoritm för registerallokering som idag används i PLEX-kompilatorn mot en modernare variant. Den allokerare som används idag ansågs när den skrevs i början av 1980-talet vara väldigt bra, men forskning som utförts sedan dess har visat på att det går att implementera det på andra, mer effektiva sätt [9].

Ursprungligen var det meningen att vi skulle undersöka om det gick att implementera algoritmen "fat cover" [10], men den visade sig inte vara lämpad för programmeringsspråket som det ser ut idag, så vi har valt att istället utvärdera några andra algoritmer.

Kapitel 2

Vad är registerallokering?

Registerallokeringen är det steg i analysen som väljer hur olika variabler ska lagras under programmets exekvering, om de ska ligga i minnet eller i processorregister, och i så fall i vilka register [2].

2.1 Identifiering av livstider

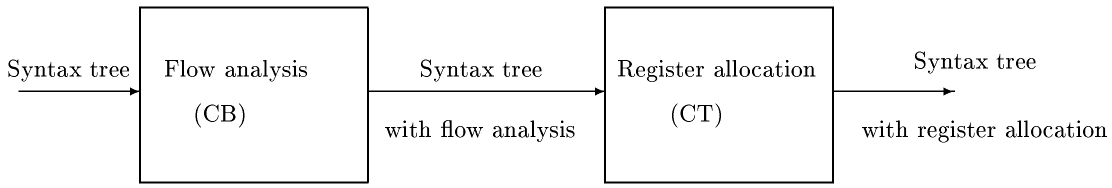
För att kunna identifiera variablernas livstider på ett bra sätt måste kompilatorn ta reda på när de olika variablerna används. Detta görs genom att variablerna delas upp i livstider, *"live ranges,"* vilka består av tiden från då variabeln får ett definierat värde, tills den sista möjligheten för användning av detta värde, eller ett värde beräknat från det, finns. Genom att dela upp variablerna på detta sätt slipper man oroa sig för variabler som används för olika saker på olika ställen i programmet, såsom exempelvis index i slingor [4].

2.1.1 Livstider i PLEX-kompilatorn

Flödesanalysdelen av PLEX-kompilatorn delar in variablerna i instanser, vilka består av respektive variabels definitions- (*def*) och användningspunkter (*use*) och området däremellan. En instans sträcker sig från variabelns första *def* till dess sista *use* [13], med en granularitet på PLEX-instruktionsnivå. Detta motsvarar de livstider som beskrevs ovan. Att granulariteten är såpass låg beror på den höga nivån på vilken internrepresentationen (SPIL, *"Simple PLEX Intermediate Language"*) ligger.

2.2 Graffärgning

En graf bestående av ett antal noder (instanser) som är hopkopplade av bågar (interferenser) ska färgas med ett begränsat antal färger (register) på ett sådant sätt att inga noder som är direkt hopkopplade får samma färg.



Figur 2.1: Flöde i registerallokeringen

Ofta så är denna graf inte planär, vilket gör problemet än mer komplext. En graf är planär om grafen kan göras om så att inga bågar korsar varandra. Det välkända fyrfärgsproblemet är definierat över planära grafer och det har bevisats att det alltid går att färga dem med som mest fyra färger. När grafen inte är planär krävs i värsta fall lika många färger som det finns noder i grafen (alla noder är hopkopplade med alla andra).

2.2.1 Graffärgning i PLEX-kompilatorn

Den globala registerallokeringsprocessen i PLEX-kompilatorn behandlar utdata från den globala flödesanalysatorn. Informationen om de olika instansernas livstider används i registerallokeringsdelen för att avgöra vilka instanser som interfererar med varandra och därför måste läggas i olika register. Två instanser interfererar om en definitionspunkt för den ena finns i det område den andra sträcker sig över [12]. Se även kapitel 7.4 för vidare information om kompilatorns uppbyggnad.

Denna information samlas i en interferensgraf som används i algoritmen för att på ett enkelt sätt avgöra hur registren bäst ska allokeras till instanserna. Om någon instans av någon anledning inte kan få ett register ges ett felmeddelande, mer information om hur detta görs i PLEX-kompilatorn finns i kapitel 5.

2.2.2 Global och interprocedurell graffärgning

Vanligtvis talar man om två olika sorters graffärgning, *lokal* graffärgning som arbetar på *"basic block"*-nivå, respektive *global* graffärgning, vilket är för hela procedurer. Det vanligaste i dagsläget är den globala graffärgningen.

Eftersom PLEX saknar funktioner såsom de definieras i konventionella programmeringsspråk (se kapitel 4 för mer information), så utförs istället graffärgningen på hela programmet, något som kallas för *interprocedurell* graffärgning [3]. Detta innebär att instanser från vitt skilda delar av programexekveringen samsas i interferensgrafen, vilket ökar dess storlek markant. Det har inte förekommit så mycket forskning på hur man förenklar interprocedurell graffärgning, eftersom det verkar vara allmänt accepterat att global graffärgning är "bäst."

Att lösa graffärgningen på något annat sätt än med just interprocedurell graffärgning är svårt i det här fallet, eftersom det just saknas funktioner, och det går att hoppa fritt mellan de olika delarna av programmet.

2.3 NP-fullständiga problem

Problemet med att färga icke-planära grafer är NP-fullständigt, *"non-deterministic polynomial-time complete,"* vilket innebär att antalet kombinationer, och därmed den tid det tar att hitta lösningen, växer exponentiellt med antalet ingående noder, något som leder till att det är mycket svårt att att hitta en optimal lösning till registerallokeringen. Olyckligtvis växer även problemet med att avgöra om man ligger inom ett visst avstånd från den optimala lösningen på samma sätt, och det är därför också ett NP-fullständigt att finna approximationer till den optimala lösningen. På grund av detta är det alltid möjligt att skriva ett program som ger en registerallokering långt ifrån den optimala, utan att man på ett enkelt sätt kan se att så är fallet.

2.4 Registerallokering för PLEX

Som PLEX-språket är definierat idag är det programmerarens skyldighet att välja om en variabel ska ligga i ett register eller i minnet. Detta leder till att registerallokeringsalgoritmen inte behöver kunna flytta ut variabler till minnet. Istället är allt den behöver göra att tala om för programmeraren vilka variabler den tycker bör läggas i minnet om inte alla får plats i register [8].

Detta förenklar för kompilatorn, men försvårar för programmeraren, bland annat eftersom det är hela variabler som måste flyttas ut till minnet av programmeraren, inte bara enstaka livstider (såvida inte programmeraren själv delar upp variabeln innan den flyttas ut).

Något som försvårar för kompilatorn är PLEX stränga krav på vilka register instanserna måste ligga i vid t ex signalsändning.

Kapitel 3

Noteringar om hur vi gjort

Det första vi gjorde var att läsa in oss på dokumentationen om PLEX-kompilatorn och dess registerallokering samt olika artiklar om generell registerallokering. Vi passade på att göra det när det var som mest semester på UAB för då kunde vi i alla fall inte få särskilt mycket hjälp med själva koden. Värt att nämna är också att när vi först hörde talas om PLEX-C förväntade vi oss att det skulle vara mer likt C än vad det egentligen är.

3.1 Simulering

Vi började genast använda CVS för att lagra kod och dokumentation, eftersom vi var två på jobbet blev det mycket enklare att hålla allting uppdaterat utan några inkonsistenser.

När vi läst in oss på några algoritmer implementerade vi fyra på prov, Yorktown [6, 5], Optimistic [4], Msk[1] och sekventiell graffärgning som används av PLEX-kompilatorn för att få en uppfattning om hur mycket bättre eller sämre de nya algoritmerna var.

I dessa test gav algoritmerna för enkelhetens skull upp helt om registerallokeringen misslyckas, vilket gjordes eftersom man i PLEX ändå manuellt i källkoden måste flytta ut de instanser som inte får plats i register till minnet. Det vi letade efter var den rutin som producerade bäst resultat utan spill.

Optimistic och Msk lyckades bäst på de slumpmässiga interferensgrafer vi provade med. De var väldigt ofta mycket bättre än både Yorktown och den sekventiella. Däremot var den sekventiella ofta bättre än Yorktown. Se tabell A.6 och figur A.7.

3.2 Implementation

Efter att testerna granskats implementerades Yorktown, Optimistic och Msk i PLEX-kompilatorn. För att förenkla arbetet delade vi upp den över 100 Kbyte stora filen där alla funktioner låg i mindre delar. En speciell fil för respektive allokerare, en för gemensamma funktioner och en för spill.

Första kompileringen av själva kompilatorn tog en halv dag, men sedan behövde bara själva registerallokeringssteget kompileras om, vilket tog betydligt kortare tid, men ändå borde kunna göras kortare genom att göra det möjligt att bara köra en omlänkning av den. Som det är nu söker makefilen igenom alla kataloger, och det tar ändå ganska lång tid. Vi kompilerade ändå alltid om registerallokeraren för att kolla efter fel innan vi körde make på hela kompilatorn.

Först implementerades algoritmerna utan hänsyn till spill eftersom de testprogram vi fick tillgång till redan var fixade så att de gick att allokera med den gamla algoritmen. Som alltid uppstod det många buggar på vägen som måste rättas till, men det som var svårast var att hitta runt i den gamla koden för att se vilka datastrukturer och värden som gällde.

[1] En egen variant på Matula-Beck [11]

Våra antaganden om att våra nya algoritmer som visat sig vara bättre i vårt föregående test skulle visa sig vara bättre även här visade sig vara riktiga. Optimistic och Msk var de enda som implementerades fullt ut, Yorktown implementerades bara delvis. Eftersom Yorktown visat sig vara sämre p g a dess tidigare spillbeslut gjordes spillberäkningarna inte om så att de även skulle passa den.

När de båda algoritmerna börjat fungera implementerades kod för att ta hand om spill. Mycket av den gamla koden för spill fick göras om eftersom de nya algoritmerna fungerar lite annorlunda än den sekventiella. Man kan inte göra samma antaganden om fria/upptagna register o s v. Även ett fel i den existerande kompilatorn, i beräkningen av vilka instanser som är bäst att flyttas till minnet, rättades. Det var en division som av misstag blev heltalsdivision i stället för en flyttalsdivision, ett fel som troligtvis uppkom i o m att kompilatorn konverterats från Pascal. Många små rester av Pascal syns, i en del tester som testar om ett tal ligger inom ett visst intervall skapas en mängd med alla tal i intervallet och det testas sedan om talet ifråga finns i mängden... Det kan ju kännas lite onödigt.

3.3 Utökningar

Efter att koden för spill börjat fungera (efter en del om och men och andra inte riktigt fullt så fina ord) började experiment med Maries lag (se kapitel 7.2 för mer information). Maries lag är en lag som har till uppgift att hålla vissa nyckelregister fria eftersom det då blir kortare och bättre kod. Det underliga är att när Maries lag inte användes alls så blev den totala koden i testprogrammen tillsammans kortare än med lagen. Vissa program blev längre utan Maries lag (i ett fall mycket), men andra blev lite kortare och uppvägde detta. Bara om vi inte använde Maries lag alls blev totala koden kortare än med gamla kompilatorn.

Registerallokeringsordningen ändrades också till den som den gamla kompilatorn körde, vilket drastiskt minskade antalet använda register. Först så struntade vi i vilken ordning registren togs och det resulterade i den något olyckliga omständigheten att bl a PR0 togs tidigt även när ingen instans krävde det. Men redan innan denna förändring var registeranvändningen lägre än den gamla kompilatorns.

3.4 Testning

Tester gjordes på 2 samlingar av program, den ena bestod av 29 rätt små program (i genomsnitt 400 Kbyte källkod per program) och den andra av 19 stora (i genomsnitt 1,5 Mbyte källkod per program). Att kompilera alla testprogram tog lång tid, cirka 2,5 timmar för de små programmen och 4 timmar för de stora. Många av testerna gjordes med varierade konstanter för Maries lag. Vi utvecklade också många små program och skriptfiler för att läsa av och ta ut information ur debugfilerna som kompilatorn skrev. Dess program tog bl a fram kodstorlek, antal och vilka register som använts och hur mycket tid som gått åt.

Kapitel 4

Introduktion till PLEX

Den kompilator vi har arbetat med är skriven för att kompilera programmeringsspråket PLEX, ett maskinvarunära realtidsprogrammeringsspråk skrivet 1970 av Ericsson (med utökningar från 1983) för användning i sina telefonväxlar. PLEX används i tusentals växlar över hela världen.

4.1 PLEX

PLEX är en förkortning för *"Programming Language for EXchanges."* Programspråket är till stor del specialiserat för sitt användningsområde, och är till viss del knuten mot den underliggande maskinvaran, telefonväxeln *AXE* [8].

PLEX finns i två varianter, PLEX-C och PLEX-M. Den variant som vi arbetat med är PLEX-C som används i *CP*, centralprocessorerna i telefonväxlarna.

PLEX-M är en åttabitarsvariant av PLEX vilken används i *EMRP*, *"extension module regional processor,"* vilket är den del av telefonväxeln som hanterar abonnentsteget. Denna variant har vi inte arbetat med.

4.2 Begrepp

Några begrepp som används i PLEX:

- Ett *"kodsegment"* är kod som utförs från ett anrop till ett uthopp, och kan liknas vid en funktion.
- Ett *"block"* är en samling kodsegment som delar på globala variabler. Flera block samarbetar vanligtvis för att tillsammans hantera en hel telefonväxel.
- För att kommunicera sinsemellan skickar kodsegmenten *signaler* till varandra.

4.3 Maskinvara

PLEX är nära knuten till den maskinvara den kör på, Ericssons egna system APZ som används i telefonväxlar. APZ har inbyggd funktionalitet för meddelandeköer med tillhörande sändning och mottagning av meddelanden, vilket ärvts av PLEX-språket, något som gör att översättningen mellan programkod och maskinkod är väldigt enkel för de instruktionerna. Detta är dock en nackdel om man vill flytta programmeringsspråket till en annan maskinvaruarkitektur, men det ligger utanför denna rapports ämnesområde.

Det är även möjligt att direkt i PLEX-koden flika in snuttar skrivna i processorns assemblerspråk i så kallade *ASA-block* (också kallat ASA-sektorer).

4.4 Signaler

Ett centralt begrepp i PLEX är *signaler*, vilka kan liknas med funktionsanrop i mer konventionella programmeringsspråk. När ett program vill anropa ett annat kodsegment, antingen i samma eller i ett annat block så sänder det en signal.

Det finns två typer av signaler, enkla signaler och kombinerade signaler. De enkla signalerna lagras i en meddelandekö, från vilken operativsystemet kommer ta reda på vilken programkod som skall utföras när det aktuella avsnittet avslutar.

Kombinerade signaler kan liknas vid funktionsanrop i konventionella program, eftersom de innehåller två signalsändningar, en i varje riktning. Dock skiljer sig semantiken något från funktionsanrop i och med att den anropade signalen kan välja att skicka en av flera olika signaler tillbaka, och detta hanteras av det kodsegment som skickade den ursprungliga signalen genom att ange var de olika möjliga resultatsignalerna tas om hand.

För att undvika att det tar för lång tid från det att en signal skickas tills det mottagande kodsegmentet exekveras går det att skicka signaler som inte skall köas. Likaså går det att skicka signaler som skall vänta en viss tid innan de exekveras.

Tillsammans med signalen kan ett antal olika data skickas. I fallet enkla signaler kopieras dessa till signalkön och hämtas tillbaka när signalen exekveras. Kombinerade signaler från sin data sänd som vanliga funktionsargument i andra programmeringsspråk.

Signalerna är i huvudsak icke-avbrytbara, *"non-preemptive"* (se dock även nästa stycke angående exekveringsnivåer), något som medför att bara en del av programkoden utförs åt gången. Detta förenklar gentemot konventionella programspråks funktionsanrop eftersom de lokala variablerna inte behöver skrivas ner mot minne varje gång. Dock innebär detta när det gäller kombinerade signaler att man inte kan garanteras att någon av de temporära variablerna har kvar sina värden när man kommer tillbaka efter anropet [7].

4.5 Exekveringsmodell

Exekveringen av PLEX-programmen är helt centrerat runt signalerna, allting som exekveras är signaler. När en process skickar en enkel signal läggs denna in i en signalkö, vanligtvis sist, men det kan påverkas med hjälp av signalprioriteterna.

Stöd för signalköerna finns i maskinvaran, något som möjliggör effektiva byten mellan de olika signalkörningarna. När en signal har kört färdigt används speciella instruktioner för att läsa ut vilken signal som kommer härnäst ur kön, varpå denna signal tas bort ur kön och tas bort. Om det fanns fördröjda signaler vars tid har kommit flyttas dessa från väntekön till exekveringskön, men de placeras sist, vilket gör att det kan dröja en liten stund extra innan de exekveras.

Det finns en begränsad form av avbrytbarhet, genom fyra olika exekveringsnivåer, där en signal som körs på en högre nivå automatiskt avbryter de som kör på lägre nivåer. De fyra nivåer som finns är (från högst till lägst):

1. TRL *"trace level"* används för felsökning.
2. THL *"traffic handling level"*, vilket är den nivå som används för normal telekommunikation.
3. CL *"C Level"* används för I/O-hantering av olika slag.
4. DL *"D Level"* används för självtest av program- och maskinvara.

Uppdelningen i prioritetsnivåer visar även den hur specialiserad program- och maskinvara är för just detta syfte, den prioriterar telekomtrafik framför andra uppgifter, och den har direkt stöd för att göra så i maskinvaran.

Detta påverkar dock inte registerallokeringen, eftersom det finns en komplett uppsättning register för varje exekveringsnivå.

4.6 PLEX-variabler

Variabler i PLEX finns i två varianter; temporära variabler som bara bibehålls under en körning från det att en signal startas tills den avslutas samt permanenta variabler som alltid finns tillgängliga och som lagras i minnet (DS – *"datastore"*.)

Eftersom ett PLEX-program inte kan avbrytas lagras alla temporära variabler i register utan att ha någon minnesadress att sparas ut till, på samma sätt så används inte register för de permanenta variablerna mer än temporärt när de används i olika beräkningar.

4.6.1 Inverkningar på registerallokeringen

I PLEX är det programmerarens ansvar att se till att de register som används räcker till för de temporära variabler som används, och om så inte är fallet måste man självmant flytta ut en eller fler variabler till det permanenta minnet.

Detta gör att registerallokeringsrutinen inte behöver kunna introducera kod för att spilla ut variabler som inte får plats, något som den måste kunna göra för konventionella programmeringsspråk. Dock bör kompilatorn ge förslag om vilka variabler som är lämpliga att spilla, eller åtminstone vilka som mest skulle förenkla registerallokeringen om de spills, så sådan funktionalitet finns, men så fort en temporärvariabel måste spillas kommer det att få till effekt att kompilatorn stannar i slutet av kompileringsfasen och rapporterar vilken eller vilka variabler som bör förläggas till datastore.

Kapitel 5

Existerande algoritm

När en variabel inte kan allokeras till register av PLEX-kompilatorn måste variabeln helt läggas i minnet, *"datastore,"* till skillnad från konventionella programmeringsspråk där spillkod läggs in för att hämta tillbaka den temporärt till ett register när den behövs och sedan lägga ut den i minnet igen. När kompilatorn upptäcker att en instans måste spillas läggs den ändå in i ett dummyregister så att kodgeneratorn skall kunna köras och eventuellt hitta fler instanser som måste spillas (p g a registerbrist vid mellanlagring av resultat o s v). Användaren får sedan ett meddelande om vilken eller vilka variabler som bör läggas ut i minnet för att kompileringen skall lyckas (om några).

5.1 Beskrivning

Registerallokeringen baseras på graffärgning, men en sekventiell sådan kallad *"Sequential Graph Coloring with Graph Reduction."*

Grafen traverseras sekventiellt nod för nod. Varje nod ges en färg som är olik de färger som de interfererande noderna som redan tilldelats en färg har. Till denna färgning används ett obegränsat antal symboliska register. När färgningen är klar räknas antalet olika färger som omger varje nod, d v s antalet färger hos de noder som interfererar med noden i fråga. Detta görs för varje nod i grafen. Observera att antalet färger som omger en nod kan vara färre än antalet noder som omger den, på grund av att noder som inte direkt interfererar med varandra kan ha samma färg. Sedan kontrolleras att antalet färger runt varje enskild nod inte överstiger det tillåtna antalet färger (antal tillgängliga riktiga maskinregister). Om det på någon nod överskrids eller är lika med antalet register så måste noden i fråga eller en av dess grannar tas bort ur grafen. Den nod som har flest olika färger runt sig väljs för borttagning. Processen fortsätter på samma sätt tills grafen är färgad med ett antal färger mindre eller lika med antalet register.

5.2 Tilldelning av register

Symboliska register allokeras till instanserna så att ingen nod har en granne med samma färg (register). Ett oändligt antal symboliska register finns tillgängliga. Om detta lyckas, d v s antalet använda symboliska register inte överstiger antalet tillgängliga maskinregister, tilldelas instanserna riktiga register direkt, annars stannar kompilatorn och ger ett felmeddelande.

I PLEX kan man även direkt lägga in assemblerkod, i en så kallad *ASA-sektor*. Det måste garanteras att de registervariabler som används i assemblerkoden verkligen tilldelas de register de representerar.

5.3 Affinitet

Vissa assemblerinstruktioner kräver att parametrarna ligger i vissa specifika register. Om en eller flera parametrar ligger i andra register så måste de flyttas till de register som krävs, och sedan tillbaka igen. Om registren som instruktionen krävde redan användes måste det ursprunliga innehållet i dem sparas undan innan instruktionen, och läggas tillbaka igen efteråt. Detta tar tid och det vill man gärna undvika. Därför försöker man allokera de instanser som innehåller dessa parametrar till de rätta registren på en gång om möjligt. Dessa register kallas för instansens affinitetsregister. Se kapitel 7.1 för vidare information om affinitet.

5.4 Nackdelar

Ett problem med att allokera register sekventiellt är att det sällan är den optimala ordningen, något som även leder till att allokeringen onödigt ofta misslyckas. Eftersom graffärgning av icke-planära grafer är ett NP-fullständigt problem är det svårt att hitta den optimala ordningen, men det finns metoder för att hitta bättre lösningar än just att gå igenom noderna i ordning. Det är sådana strategier vi har undersökt närmare.

5.5 Pseudokod

Den sekventiella graffärgningen ser ut ungefär så här:

```
" This algorithm goes through the instances sequentially, and tries to
  find a color for each instance in the order they are numbered"

BEGIN Color the Instances ( startinst, maxnrsmbregs).

   PreColor Pseudo Variables( All Instances ).

   "Loop through each instance and try to assign a color to it"
   FOR inst IN instances DO
      "Try to find a color for this instance"
      FOR col IN available_colors DO
         "Is this color busy?"
         busy=FALSE.

         FOR i IN interferands DO
            IF i INTERFERES inst THEN
               "inst interferes with i"
               IF Assigned a Color(i) AND
                  col = Color of (i) THEN
                  "Color is busy! Try again."
                  busy=TRUE.
               END IF
            END IF
         END FOR

         IF NOT busy THEN
            "The color col is available"
            Assign Color(inst,col).
         END IF
```

```
      END FOR

      IF busy THEN
         "Failure! No color was available."
      END IF
      IF ERROR THEN
         EXIT
      END IF
   END FOR
END.
```

Kapitel 6

Ny algoritm

Vi har implementerat några olika algoritmer, vilka var för sig har sina för- och nackdelar.

6.1 Vilka algoritmer finns?

Det finns flera olika algoritmer för att allokera register i funktioner, här följer en kort genomgång av några av dem:

6.1.1 Graffärgning

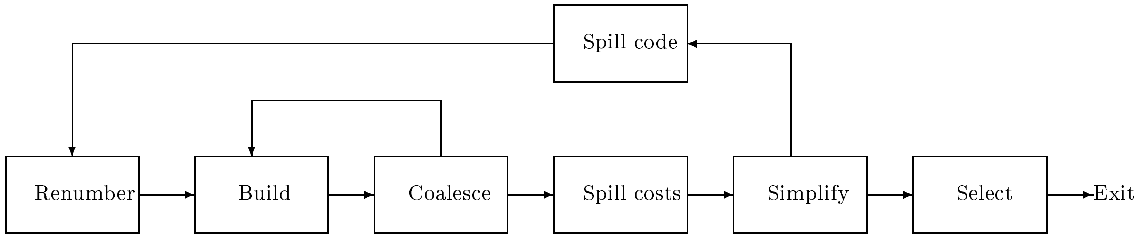
Graffärgning, *"graph coloring,"* ser problemet med att tilldela register till variabler som ett färgningsproblem där variablernas livstider är noder i grafen och de binds samman då motsvarande livstider interfererar.

6.1.2 Fat cover

Fat cover försöker göra bättre val med variabler i loopar eftersom andra algoritmer ofta underskattar hur mycket man tjänar på att lägga dem i register. Genom att gå inifrån och ut och allokera register till variabler i loopar först hoppas man få snabbare kod. För att kunna göra bra val måste programmeringsspråket vara strukturerat, ju mer desto bättre, men PLEX är inte särskilt strukturerat, utan innehåller ofta `goto`, tillåter godtyckliga hopp in och ut ur slingor och mellan signalhanteringsrutiner, något som försvårar implementationen avsevärt, och minskar vinsten. Dessutom skulle en implementering av Fat Cover kräva omfattande modifikation av både flödesanalysatorn och registerallokeraren, något som det helt enkelt inte fanns tid till.

6.1.3 Priority based

Prioritetsbaserad graffärgning introducerades av Chow och Hennessy. Istället för att göra grafen enklare med hjälp av spill delar de upp instanser i mindre delar (*"live range splitting."*) För att hålla nere på kostnaden för uppdelningen måste interferensgrafen göras grövre. Där Chaitins algoritm (och derivat) ser interferenser på instruktionsnivå ser denna algoritm endast ner till "basic block"-nivå, där ett *"basic block"* är en sekvens instruktioner som alltid utförs linjärt [1]. Därmed antar den att alla instanser som förekommer i samma "basic block" interfererar. Prioritetsbaserad färgning kan dock ibland ge bättre resultat än spillande algoritmer (Yorktown, Optimistic, o s v). En annan skillnad är att Chaitin antar att alla instanser ligger i register och spiller ut vid behov, medan den Prioritetsbaserade antar att alla ligger i minnet och plockar in de den kan i register.



Figur 6.1: Chaitins registerallokerare

6.2 Varför valde vi Optimistic?

Optimistic var tillräckligt lik den sekventiella algoritmen för att göra implementationen enkel men samtidigt så mycket bättre att det lönade sig att implementera den.

6.3 Övergripande beskrivning av graffärgning

De nedanstående algoritmerna bygger alla på Chaitins registerallokerare och består av 7 grundläggande delar [4]. Se figur 6.1.

Renumber numrerar alla livstider unikt.

Build bygger sedan upp en interferensgraf som visar var de olika intervallen interfererar med varandra.

Coalesce tar bort onödiga kopior genom att kombinera två eller fler livstider som egentligen är samma värde men under olika namn, genom att ta bort kopieringsinstruktionen och kombinera dem. Eftersom en sådan operation kan förändra interferensgrafen så repeteras *Build* och *Coalesce* tills det inte går att kombinera fler.

Spill costs beräknar kostnaden i form av längre exekveringstid som orsakas om man tvingas spilla en viss livstid. Detta görs för varje livstid i grafen.

Simplify tillsammans med *Select* är de delar som färgar grafen. Simplify plockar isär grafen en nod i taget och lägger dem på en stack enligt vissa specifika kriterier (vilka varierar något från algoritm till algoritm) för senare färgning. Stacken är en vanlig stack av typen sist-in-först-ut och används för mellanlagring av noder medan grafen plockas isär.

Select tar ut noderna en och en från stacken, vilket alltså innebär motsatt ordning mot den de lades in på, och färgar dem (om möjligt).

Spill code sätter in kod för att ladda/spara undan register för de livstider som tidigare markerats för spillning, och om det behövdes sättas in kod så görs allt om fr o m *Renumber* eftersom vi nu fått en ny graf.

På grund av det sätt som registerallokeringen är implementerad i PLEX-kompilatorn har vi inte kunnat följa detta förfarande helt och fullt. De skillnader som förekommer finns beskrivna i kapitel 7.

6.4 Yorktown (Chaitin)

Beskrivning

Yorktownallokeraren var den första implementationen av global registerallokering baserad på graffärgning [6]. För effektivitet är interferensgrafen som Build tillverkar lagrad på två sätt samtidigt, en triangulär bitmatris och vektorer som visar vilka intervall som interfererar med vilka, s k intilliggandevektorer, *"adjacency vectors."* Simplify tar alltid bort noder som har grad mindre än k i godtycklig ordning. Om ingen nod med grad mindre än k finns så spills en instans genast och hela graffärgningen görs om från början.

För-/nackdelar

På grund av Simplify-algoritmens pessimistiska antagande att en graf *inte* går att färga om det bara finns noder med grad $\geq k$ kvar så kommer den att spilla många gånger då det inte behövs, men samtidigt kommer det automatiskt vara möjligt att färga grafen så fort Simplify-fasen lyckas.

Pseudokod

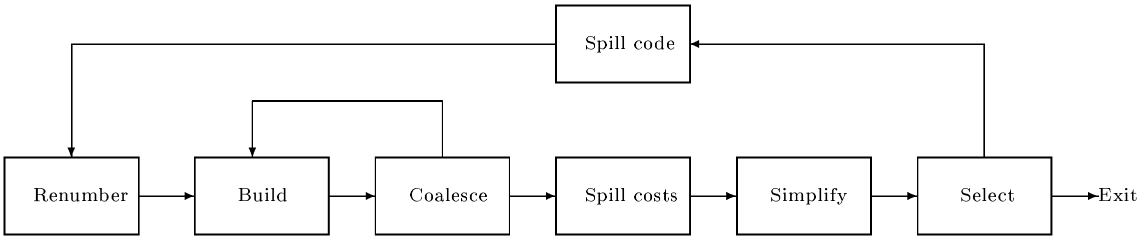
```
BEGIN Allocate Symbolic Registers to the Instances
        ( startinst , maxnrsmbregs )

  LEFTTOCOLOR := { All Instances } - { Spilled Instances } -
                 { Pseudo Registers }.
  REMOVESTACK := ().

  WHILE LEFTTOCOLOR is non-empty DO
    current := Find Instance That Can Be Colored ( maxnrsmbregs , LEFTTOCOLOR ).
    IF no instance was found THEN
      " Ignore it, which leads to spilling "
      current := Find Instance With Highest Degree ( LEFTTOCOLOR ).
    ELSE
      Push Instance ( REMOVESTACK , current ).
    END IF no instance was found.
    LEFTTOCOLOR LESS := current.
  END WHILE LEFTTOCOLOR.

  WHILE REMOVESTACK is non-empty DO
    current := Pop Instance ( REMOVESTACK ).
    Color Instance ( current ).
  END WHILE REMOVESTACK.

END Allocate Symbolic Registers to the Instances
```

Notera: Övrig pseudokod finns i bilaga B.

6.5 Optimistic (Briggs)

Beskrivning

Den största skillnaden mellan Yorktown och Optimistic är sättet den senare algoritmen plockar isär grafen på, se figur 6.2. Istället för att ge upp och spilla om det bara finns noder av grad $\geq k$ hoppas den istället på det bästa och tar bort en godtycklig nod. I fallet med vår implementation tar den



Figur 6.2: Briggs registerallokerare

den första noden ur listan med ofärgade noder. Om det sedan i färgningssteget visar sig att grafen inte går att färga med k färger måste även denna algoritm spilla, men det blir mycket mer sällan.

För-/nackdelar

Kan färga alla grafer som Yorktown-allokeraren kan färga och många fler, men det syns inte förrän i Select-steget huruvida spill måste ske eller inte.

Pseudokod

```
BEGIN Allocate Symbolic Registers to the Instances
        ( startinst , maxnrsmbregs )

  LEFTTOCOLOR := { All Instances } - { Spilled Instances } -
                 { Pseudo Registers }.
  REMOVESTACK := {}.

  WHILE LEFTTOCOLOR is non-empty DO
    current := Find Instance That Can Be Colored ( maxnrsmbregs , LEFTTOCOLOR ).
    If no instance was found THEN
      current := First ( LEFTTCOLOR ).
    END IF non-empty.
    Push Instance ( REMOVESTACK , current ).
    LEFTTCOLOR LESS := current.
  END WHILE LEFTTCOLOR.

  WHILE REMOVESTACK is non-empty DO
    current := Pop Instance ( REMOVESTACK ).
    Color Instance ( current ).
  END WHILE REMOVESTACK.

END Allocate Symbolic Registers to the Instances
```

Notera: Övrig pseudokod finns i bilaga B.

6.6 Msk

Beskrivning

Denna metod försöker färga grafen på två sätt, först genom att ta bort noderna i ordning flest grannar först och om det misslyckas provar den att färga i motsatt ordning, minst antal grannar först. Denna har visat sig i stort lika effektiv som Optimistic.

Denna algoritm är en egenutvecklad variant på den algoritm som beskrivs av Matula-Beck [11].

För/nackdelar

Färgar många fler grafer utan spill än Yorktown, men tar längre tid på sig eftersom den försöker igen om den misslyckats med första försöket.

Pseudokod

```
" Variant of Matula-Beck (dubbed ‘msk’) "
BEGIN Allocate Symbolic Registers to the Instances
        ( startinst , maxnrsmbregs )

  LEFTTOCOLOR := { All Instances } - { Spilled Instances } -
                 { Pseudo Registers }.
  REMOVESTACK := ().

  WHILE LEFTTOCOLOR is non-empty DO
    current := Find Instance With Highest Degree ( LEFTTOCOLOR ).
    Push Instance ( REMOVESTACK , current ).
    LEFTTOCOLOR LESS := current.
  END WHILE LEFTTOCOLOR.

  REVERSESTACK := Reverse ( REMOVESTACK ).

  allocatable := true.
  WHILE REMOVESTACK is non-empty DO
    current := Pop Instance ( REMOVESTACK ).
    Color Instance ( current ).
    IF could not color THEN
      " We could not color, try the other direction "
      allocatebale := false.
      QUIT WHILE REMOVESTACK.
    END IF no color.
  END WHILE REMOVESTACK.

  IF allocatable THEN
    REVERSESTACK := ().
    RETURN.
  END IF allocatable.

  REMOVESTACK := ().

  " This direction is the ‘real’ Matula-Beck "
  WHILE REVERSESTACK is non-empty DO
    current := Pop Instance ( REMOVESTACK ).
    Color Instance ( current ).
  END WHILE REVERSESTACK.

END Allocate Symbolic Registers to the Instances
```

Notera: Övrig pseudokod finns i bilaga B.

Kapitel 7

Tekniska detaljer

När vi implementerat registerallokeringen i PLEX-kompilatorn har vi stött på några implementationstekniska problem vars lösningar inte direkt går att hitta i beskrivningarna av de algoritmer vi implementerat.

7.1 Affinitetsgrader

Det första problemet är att de olika registren i den arkitektur PLEX-kompilatorn arbetar mot används på olika sätt, vilket leder till att vissa beräkningar bara kan ske i vissa register. Denna specialisering av register är ett utmärkande drag för CISC-arkitekturer, såsom den systemet är uppbyggt på.

Detta är bakgrunden till de ”affinitetsregister” som tidigare nämnts, och varje instans av en variabel som är involverad i en beräkning som måste (eller bör) utföras i ett visst register får en viss ”affinitetsgrad” för det registret tilldelad sig. Detta tal används sedan för att beräkna vilket register som mest lämpar sig för att hålla variabeln, för att minska antalet onödiga kopieringar.

De algoritmer vi har utvärderat har i sina grundformer inget stöd för att vissa instanser ska ligga i specifika register, så det blir mer eller mindre ”påklistrat” på algoritmen för att det ska fungera på ett tillfredsställande sätt. Vi har löst det hela genom att dels modifiera det sätt som instanserna plockas ut, dels genom att modifiera själva färgningen till att ta hänsyn till vilka register som kan tilldelas.

Vi har gjort vad vi kunnat för att detta tillägg inte skall påverka de matematiska grunder på vilka algoritmerna grundar sig.

7.2 ”Maries lag”

I den ursprungliga kompilatorn har man dessutom valt att använda något de kallat för ”Maries lag” [1] för att undvika problem med att instanser hamnar i register de inte bör hamna i. Maries lag innebär att kompilatorn undanhåller vissa register från registerallokeringen om det finns flera instanser som någon gång har behov av att hamna i det registret.

Det är främst två register som berörs, dels det pekarregister som används för att peka ut poster i datafiler, PR0, dels indexregistret IR, vilket används som index för fältvaribler.

Eftersom dessa operationer är väldigt ofta förekommande i programkoden låter inte Maries lag i sin ursprungliga form en instans få ett sådant register om den inte har betydligt fler användningar av registret än någon annan av de interfererande instanserna. Undantaget är om instansen är helt ensam om att använda registret under sin livstid, i vilket fall den får registret direkt.

[1] Döpt efter en tidigare chef på utvecklingsavdelningen

Register	Beskrivning	Funktion
IR	Indexregister	Index i fält
PR0	Pekarregister	Peka ut poster i filer
PR1	Pekarregister	Peka ut buffertvariabler
CR	Jämförelseregister	Lagra ena värdet vid jämförelse
SR0, SR1	Signalregister	Lagra signalinformation
DR0–DR23	Dataregister	Lagra signaldata
AR0–AR3	Aritmetisk/logiska	Beräkningar
WR0–WR29	Arbetsregister	Temporära variabler

Tabell 7.1: Registeranvändning

Ett senare tillägg är dessutom att dataregistren DR0–DR23 omfattas av lagen. Dessa register är viktiga vid signalsändning, eftersom de data som skickas ligger i registren, i turordning från DR0 och uppåt.

De olika specialregistren i processorn listas i tabell 7.1 [8].

Eftersom Maries lag, såsom den implementerats, var avpassad för den existerande registerallokeringsalgoritmen har vi genom tester försökt komma fram till vilka parametrar som passar de nya algoritmerna bäst.

7.3 Lokal registerallokering

Ytterligare ett problem vi haft är att registerallokeringen sker på en högre nivå än maskinkoden, en begränsning som tillkom på grund av det IBM-stordatorsystem kompilatorn ursprungligen körde på. Den datorn kunde inte hantera mer än 16 Mbyte minne, och analyser visade att om registerallokeringen då skulle körts direkt på maskinkoden skulle det ta upp mer minne än så, varför man lät göra den på en högre nivå.

Denna abstrahering får till följd att man ibland behöver extra register utöver de register som den globala *(interprocedurella)* registerallokeraren tilldelat, något som tillgodoses i vid en extra *lokal* registerallokeringsfas, vilken anropas från kodgeneratorn. Denna extra registerallokeringsfas gör att det inte alltid är bra att generera en ”perfekt” registerallokering i det globala steget.

I efterhand har man inte ändrat på denna begränsning, eftersom det skulle innebära en rejäl omskrivning av kompilatorn. Vi har därför valt att ”ignorera” problemet och riktat in oss på att göra registerallokeringssteget så bra som möjligt, något som får till följd att det ibland uppstår fel i efterhand, p g a att kodgenereringen behöver använda register temporärt för beräkningar, o s v.

7.4 Interaktion mellan delar av kompilatorn

PLEX-kompilatorn är uppbyggd i en mängd olika utbytbara moduler, något som medfört att det varit enkelt att byta ut just registerallokeringsdelen. Det är två moduler inblandade i den bit vi har tittat på, dels CB, den globala flödesanalysatorn [13], och dels CT, vilket är själva registerallokeringen [12].

Flödesanalysatorn (CB) är den del som delar upp de olika variablerna i instanser, och som driver registerallokeringssteget (CT). Efter CB anropas kodgenereringsmodulen (*”Code Selector”* – CS), vilken i sin tur anropar den tidigare nämnda lokala registerallokeringen (CL). CT använder sig av vissa biblioteksfunktioner för läsning av instansinformation och hantering av värdemängder som är gemensamma med CB-steget, varför vi var tvungna att sätta oss in i funktionen hos även detta.

I CB har vi lagt in några extra biblioteksfunktioner vi behövde och som inte fanns med sedan tidigare, och även lagt in definitioner på de extra värdemängder de algoritmer vi har implementerat använder.

Vi har inte ändrat i några andra delar av kompilatorn än just CB och CT, något som inte heller har ställt till några problem, kompilatorn fungerar som helhet bra även med registerallokeringsalgoritmen utbytt.

Kapitel 8

Resultat av jämförelser

För att avgöra vilken algoritm som fungerar bäst har vi jämfört hur många register som allokerats av de olika algoritmerna, hur många instanser som fått sina affinitetsregister, samt hur stor den resulterade koden har blivit.

8.1 Tillvägagångssätt

Vi har kompilerat en mängd olika program, riktiga program som används/har använts i Ericssons telefonväxlar.

8.1.1 Vad vi gjort

Det vi sedan har tittat närmare på är:

- Antalet unika register som tilldelats de olika instanser, ett mått på hur ”snål” algoritmen är med att allokera register. Ju färre register som använts, desto fler instanser torde den kunna allokera innan man tvingas flytta ut variabler till minnet.

  Nu säger kanske inte detta mått allting om hur bra algoritmen är på att allokera register, eftersom ett enda parti med hög registerbeläggning kommer att göra så att statistiken inte blir helt korrekt. Överlag kan man nog ändå anse det som ett hyfsat mått på hur bra algoritmen är på att snåla på register.

- Antalet instanser som tilldelas sina affinitetsregister, ett mått på hur ”bra” algoritmen är på att placera de olika instanserna i rätt register.

  Den här siffran påverkas till stor del av Maries lag, eftersom den förhindrar vissa instanser från att allokeras till sina affinitetsregister. Det är därför inte så underligt att våra nya algoritmer när de används helt utan Maries lag visar sig betydligt bättre på att allokera instanser till sina affinitetsregister, så de resultaten är bara med som en referens. Resultaten för de nya algoritmerna när de använder Maries lag är de som är intressanta, även om vi haft vissa problem att implementera den korrekt (se nedan).

- Storleken på den genererade koden, vilket sammanvägt med affinitetsregistren ovan är ett mått på hur väl algoritmen placerar de olika instanserna, eftersom kod som genereras för att flytta värden mellan olika register ökar kodstorleken i onödan.

8.1.2 Vad vi inte gjort

Vi har *inte* försökt jämföra exekveringstider, eller hur många instanser vi har kunnat lägga i register innan spill har framtvingats.

Det första beror helt enkelt på att programmen dels inte arbetar linjärt, och dessutom på att de samarbetar ute i telefonväxlarna, något som medför att det är svårt att beräkna exekveringstider. Dessutom är problemet med att mäta exekveringstiderna utanför området för vårt examensarbete, det är ett problem som andra har arbetat på under samma tidsperiod som oss.

Att vi inte har mätt antalet instanser som kan läggas i register utan att spillas beror dels på att de redan existerande programmen vi testat algoritmerna på redan har varit anpassade så att de går att kompilera så att alla instanser får register med den existerande algoritmen, och dels på att det är svårt att generera realistiska testfall utan att ha erfarenhet av hur PLEX-program för telefonväxlar skrivs.

Vi har dock kontrollerat att spillalgoritmen faktiskt fungerar genom att skriva ett enkelt program som har löjligt många samtidiga instanser, för att helt enkelt framtvinga spill. Detta test körde dock in i gränsen på antalet möjliga register, och den gränsen är samma oavsett om vi kör den ursprungliga algoritmen, eller någon av våra modifierade algoritmer, så testet säger ingenting om vilken som är bättre, bara att den faktiskt fungerar.

8.2 Resultat

Rent generellt så har våra nya algoritmer visat sig vara bättre än originalalgoritmen på de flesta områden, även om vi haft vissa problem att anpassa Maries lag (se kapitel 7.2) till de nya algoritmerna. Vi har testat våra registerallokeringsalgoritmer på 48 olika PLEX-program, och resultaten diskuteras här och presenteras i bilaga A.

Vi har mätt två olika uppsättningar program – en uppsättning huvudsakligen större program och en uppsättning huvudsakligen mindre program (se kapitel 3.4), i två olika situationer – dels utan Maries lag (tabell A.2 och A.4) och dels med (tabell A.3 och A.5) (se kapitel 8.1). Notera dock att originalalgoritmen alltid är körd *med* Maries lag, eftersom den alltid används i den ursprungliga kompilatorn.

Diagram över resultaten finns även de i bilaga A.

8.2.1 Jämförelse 1: Stora program

Se tabell A.2 och A.3 samt figur A.1, A.2 och A.3.

Vad gäller de större programmen kan vi se att programmen blir ungefär 5% större när de kompileras med de nya algoritmerna, jämfört med den gamla, något som gäller både när Maries lag används och när den inte används. Minskningen av antalet använda register är markant (runt 20%) för båda de nya algoritmerna, och tämligen konstant mellan de båda testerna. Även ökningen av antalet register som får sina affinitetsregister är markant (runt 30% för Msk och runt 40% för Optimistic). Förbättringen är något mindre med Maries lag implementerad, jämfört med utan den, och på motsvarande sätt är programstorleken något mindre utan Maries lag än med den.

Förbättringen mätt som skillnaden i antalet allokerade register dividerat med skillnaden i storlek (båda mätt gentemot originalalgoritmen) är nästan desamma för de båda algoritmerna, Optimistic är 1,11 gånger bättre utan Maries lag jämförd med då den används, och Msk 1,13 gånger.

8.2.2 Jämförelse 2: Mindre program

Se tabell A.4 och A.5 samt figur A.4, A.5 och A.6.

Vad gäller de mindre programmen kan vi se att programmen blir ungefär lika stora när de kompileras med de nya algoritmerna, jämfört med den gamla, och i ett fall blir de till och med något mindre. Minskningen av antalet använda register är markant (runt 10%), om än inte lika stort som i den andra gruppen av program, och tämligen konstant mellan de båda testerna. Även ökningen av antalet register som får sina affinitetsregister ligger på omkring 10–15%, även det mindre än i den andra jämförelsegruppen, och precis som den minskar antalet affinitetsregister och ökar kodstorleken när Maries lag implementeras.

Jämförelsetalet beräknat som i föregående stycke visar att Msk är 1,02 gånger bättre utan Maries lag än med det för denna programuppsättning, men tvärtemot alla andra ger Optimistic *mindre* kod än vad originalalgoritmen ger, dock är den fortfarande bättre utan Maries lag än med.

8.2.3 Kompileringstider

I tabell A.1 visas totala kompileringstider (exklusive själva kodanalysen) för de två programuppsättningarna för de olika algoritmerna. Notera att kompileringen skett samtidigt som datorerna använts för andra ändamål, så siffrorna är ungefärliga. Kompileringstiderna är insamlade på SUN SparcStation 5-datorer med en SPARC-processor på 110 MHz och 128 Mbyte minne.

Kapitel 9

Möjliga förbättringar

Under arbetets gång har vi funderat på några detaljer som skulle kunna förbättras, men som gått utanför det område arbetet täcker, eller skulle ha tagit alldeles för lång tid att implementera. De flesta av förbättringarna finns beskrivna i andra avhandlingar och litteratur på området.

9.1 Affinitet

Som alternativ och/eller komplement till affinitet kan man låta kompilatorn lägga in nya instanser med förfärgade register precis innan signaler och andra operationer som kräver att parametrar är i speciella register [3].

Låt säga att ett funktionsanrop kräver att parametrarna ligger i ordning med början i register r_0. Då skulle kompilatorn lägga in kod för att flytta parametrar på så sätt:

$$x \leftarrow f(a, b, c, d);$$

blir:

$$\begin{aligned} &r_0 \leftarrow a; r_1 \leftarrow b; r_2 \leftarrow c; r_3 \leftarrow d; \\ &x \leftarrow f(r_0, r_1, r_2, r_3); \end{aligned}$$

Att lägga in kod för att flytta parametrar till rätt register innan registerallokeringen utförs kommer att möjliggöra en effektivare allokering och kodgenerering eftersom kodgeneratorn inte behöver göra förflyttningar i efterhand, och registerallokeraren därför inte behöver ta hänsyn till det och försöka hålla register fria.

Detta kommer inte heller att producera någon extra kod eftersom coalescing-steget kommer att sätta ihop den förfärgade registerinstansen med den instans som först användes i funktionsanropet. Om coalescing inte kunde göras ligger koden kvar, men den skulle ändå ha satts in av kodgeneratorn senare.

Registerallokeraren i PLEX-kompilatorn arbetar på SPIL-nivå, där SPIL är en internrepresentation av PLEX-programmets syntaxträd, något som kan försvåra en eventuell implementation av detta i kompilatorn.

9.2 Spill

9.2.1 Räkna på hela variabler

När spillkostnad beräknas görs det bara på de instanser som krockar, men när man spiller spills hela variabeln och alla dess instanser. Därför vore det kanske effektivare att räkna på en variabels alla instanser när spillkostnad beräknas. Låt säga att en instans av P och en instans av Q krockar och en måste spillas. Nu är det så att om Q:s instans har lägre spillkostnad är P:s instans så spills Q:s alla instanser. Det tål att undersökas om det är bättre att räkna spillkostnad på både P och Q:s alla instanser och spilla den av P och Q som har lägst totala spillkostnad.

9.2.2 Bäst av tre

Som en effekt av att det inte går att avgöra hur långt ifrån den optimala lösningen den funna lösningen befinner sig följer att det oftast inte går att avgöra om en algoritm är bättre än en annan på att allokera registren, vilket gör att resultaten pendlar mellan att vara bättre för en algoritm för vissa program, men för en annan för andra.

Detta gäller även i allra högsta grad spillkostnadsberäkningen, varför det kan vara intressant att beräkna kostnaden med flera olika metoder och sedan ta ett medelvärde, för att jämna ut eventuella stora fel på vissa variabler. Detta föreslås även i [4], men kostar naturligtvis tid, varför vi inte har implementerat det.

9.3 Delning av instanser

Ju längre en instans livstid sträcker sig, desto större är risken att den interfererar med andra. För att undvika detta problem kan man dela upp instansen i flera mindre livstider, alltså precis motsatsen mot "coalesce"-steget, där varje livstid kan tilldelas varsitt register, med registerförflyttningar mellan dem. Denna metod föreslås i [3].

Uppdelningen får dock till följd att registerallokeringen blir mer komplex, eftersom antalet instanser som skall allokeras ökar. Det finns också nackdelar med att göra på detta vis, eftersom man måste ta hänsyn till de nyligen delade instanserna i coalescing-steget så att de inte sätts ihop igen. Detta medför en kostnadsökning i tid för registerallokeringen.

Kapitel 10

Slutsats

Vi insåg rätt tidigt att den sekventiella graffärgningen som ursprungligen var implementerad inte var så väldigt effektiv, något som visade sig vara korrekt från de tester vi genomfört. Något vi inte var lika beredda på var dock att den trots allt skulle visa sig vara bättre än en ”ren” implementation av Yorktownallokatorn, såsom beskriven av Chaitin. Däremot har den optimistiska allokeringsrutinen, såsom beskriven av Briggs, överträffat våra förväntningar, och lyckats allokera variablerna till färre register, men samtidigt gett fler instanser deras affinitetsregister.

Nu har det tyvärr varit svårt att dra några klara slutsatser om huruvida de nya allokeringsrutinerna faktiskt ger någon förbättring hos effektiviteten hos programmen i verkligheten, något som huvudsakligen beror på två orsaker:

- De program som vi testat på har redan varit anpassade så att de gick att kompilera med den gamla kompilatorn.
- Exekveringstider för programmen har varit svåra att beräkna, eftersom programmen måste köras tillsammans i en helhet för att fungera.

Vi har därför bara tittat närmare på hur många (och vilka) register de olika variablerna allokeras till, samt hur stort det kompilerade programmet blev.

10.1 Registeranvändning

Eftersom de program vi har kompilerat redan var anpassade så att alla variabler gick att allokera med den gamla kompilatorn, har det varit svårt att dra någon slutsats om kompilatorn faktiskt underlättar det för programmerarna genom att tillåta fler program att kompileras utan att man behöver flytta variablerna till minnet. I de enkla test vi gjort har det dock verkat som om så var fallet.

Det vi har jämfört är istället mängden olika register som används i de olika programmen, och det visar på att de nya algoritmerna i de allra flesta fallen är bättre än den gamla. Det finns dock vissa fall då den gamla är bättre, men det är bara naturligt eftersom alla algoritmer försöker approximera en ”perfekt” lösning, en lösning som inte går att hitta inom polynomisk tid.

Vad gäller de olika algoritmer vi har testimplementerat har den optimistiska algoritmen varit den som visat sig vara bäst.

10.2 Kodstorlek

I en av testsviterna fick vi mindre kod men i den andra större (I-programmen). Våra experiment med Maries lag var inte så lyckade, i de fall då vi använde den blev kodstorleken större än vad originalkompilatorn gjorde. I en av de två testsviterna blev kodstorleken mindre än originalkompilatorns om vi helt tog bort Maries lag. Kodstorleken blev större ju större konstanter vi hade i Maries lag, men däremot blev registeranvändningen mindre.

Vi drar slutsatsen om Maries lag att konstanterna verkar vara bundna till registerallokeringsrutinen och man måste ta fram nya konstanter för den, kanske även utforma den på ett annat sätt. Konstanterna har blivit empiriskt framtagna för den gamla kompilatorn och man måste gå tillväga på samma sätt här.

Bilaga A

Tabeller och diagram

Denna bilaga innehåller de tabeller och diagram som diskuteras i kapitel 8. Tabell A.1 innehåller kompileringstiderna, och tabellerna A.2, A.3, A.4 och A.5 innehåller följande information:

Register Hur många register som använts till programmet.

Reg-Diff Skillnad jämfört med originalalgoritmen.

Affinitet Hur många av de instanser som vill ha affinitetsregister som fått ett av dem den vill ha.

Program	Opt	Msk	Org
Stora	3.39	3.32	3.53
Små	1.55	2.31	1.58

Tabell A.1: Kompileringstider (timmar och minuter)

Data från tabellerna A.2 och A.3 visas i tre diagram; figur A.1 visar hur många register som använts i de olika programmen, A.2 antalet register som får sina affinitetsregister samt A.3 storleken på den genererade koden.

På motsvarande sätt visas data från tabellerna A.2 och A.3 i de tre figurerna A.4 (antalet register), A.5 (affinitetsregister) samt A.6 (kodstorlek).

Resultat från våra simuleringar, där vi tog en slumpmässig interferensgraf och såg om det gick att allokera register utan att spilla några instanser visas i tabell A.6 och figur A.7. Varje kombination av antal instanser och register har kört i ungefär en timme vardera (på en Pentium Classic 120 MHz-bestyckad dator), men eftersom de tar olika lång tid att utföra har antalet tester varierat från 500 (för kombinationen 1000/30) till 8000 (200/7), men de flesta har kört ungefär 3500 tester.

Program	Register			Reg-Diff		Affinitet				Kodstorlek		
	Opt	Msk	Org	Opt	Msk	Opt	Msk	Org	Max	Opt	Msk	Org
iucach	32	32	49	-3	-17	417	349	264	576	9696	9436	9281
iucbca	29	30	31	-2	-1	656	585	401	947	21167	21003	21132
iuccs	36	43	52	-6	-9	777	738	510	1367	25387	26844	23357
iuchdb	34	33	46	-12	-13	416	428	275	549	7681	7707	7698
iucsca	29	30	30	-1		654	571	387	963	20636	20573	20711
iucspa	32	33	42	-10	-9	487	474	324	606	8949	8875	8974
iucw	30	31	34	-4	-3	647	586	356	873	21385	21307	21534
iudb	32	32	44	-12	-12	1096	924	689	2173	35680	35481	34532
iuinte	30	30	37	-7	-7	644	631	508	694	8786	8807	9088
iuisph	31	32	36	-5	-4	919	904	679	1044	14938	15333	14962
iumcid	32	32	37	-5	-5	719	679	527	918	20661	20461	20378
iusccs	36	43	52	-16	-9	777	738	510	1367	25387	26844	23357
iussc	32	39	46	-14	-7	625	614	340	812	38747	35318	32149
iusscm	32	32	47	-15	-15	385	369	251	481	9704	9676	9659
iusscr	35	39	45	-10	-6	1485	1368	1142	1699	25403	24577	21630
msdh2	38	38	46	-8	-8	155	155	119	183	14775	14865	14018
mtv	30	30	37	-7	-7	860	862	674	906	9104	9103	9287
reu	30	31	34	-4	-3	2184	2161	1833	2252	18508	18245	18572
sufda	27	27	31	-4	-4	386	381	340	494	16295	16294	16175
TOTAL	607	637	776	-169	-139	14289	13457	10129	18904	352889	350753	336494
%	78,2	82,1	100			75,6	71,2	53,6	100	104,9	104,2	100
						141,1	132,9	100				

Tabell A.2: Registerallokering utan Maries lag, stora program

Program	Register			Reg-Diff		Affinitet				Kodstorlek		
	Opt	Msk	Org	Opt	Msk	Opt	Msk	Org	Max	Opt	Msk	Org
iucach	32	32	49	-3	-17	404	336	264	576	9689	9482	9281
iucbca	29	31	31	-2		665	542	401	947	21239	21319	21132
iuccs	35	43	52	-17	-9	753	715	510	1367	26250	26907	23357
iuchdb	33	33	46	-13	-7	403	398	275	549	7711	7782	7698
iucsca	29	31	30	-1	+1	655	526	387	963	20728	20919	20711
iucspa	32	33	42	-10	-9	474	427	324	606	8967	9054	8974
iucw	30	31	34	-4	-3	648	571	356	873	21451	21369	21534
iudb	32	32	44	-12	-12	1238	881	689	2173	35449	35715	34532
iuinte	30	30	37	-7	-7	640	625	508	694	8819	8841	9088
iuisph	31	32	36	-5	-4	905	895	679	1044	14996	15343	14962
iumcid	32	32	37	-5	-5	703	670	527	918	20714	20465	20378
iusccs	35	43	52	-17	-9	753	715	510	1367	26250	26907	23357
iussc	32	39	46	-14	-7	587	584	340	812	38918	35352	32149
iusscm	32	32	47	-15	-5	350	336	251	481	9779	9749	9659
iusscr	35	39	45	-10	-6	1466	1359	1142	1699	25245	24581	21630
msdh2	38	38	46	-8	-8	153	153	119	183	14776	14866	14018
mtv	30	30	37	-7	-7	860	859	674	906	9179	9115	9287
reu	30	31	34	-4	-3	2179	2155	1833	2252	18556	18259	18572
sufda	27	27	31	-4	-4	393	381	340	494	16267	16294	16175
TOTAL	604	639	776	-172	-137	14229	13128	10129	18904	354983	352319	336494
%	77,8	82,3	100			75,3	69,4	53,6	100	105,5	104,7	100
						140,5	129,6	100				

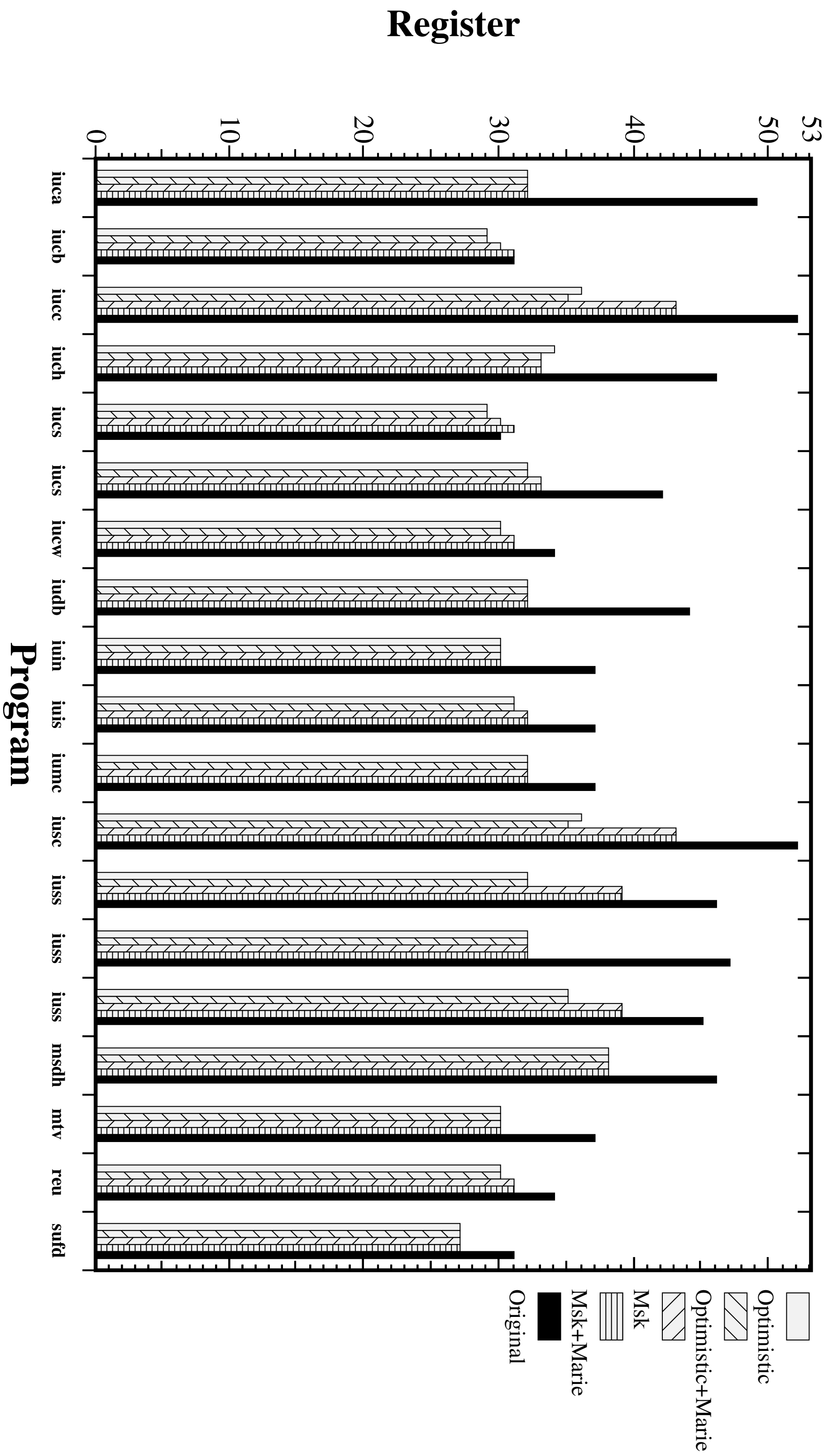
Tabell A.3: Registerallokering med Maries lag, stora program

Program	Register			Reg-Diff		Affinitet				Kodstorlek		
	Opt	Msk	Org	Opt	Msk	Opt	Msk	Org	Max	Opt	Msk	Org
bada	12	13	12		+1	153	108	70	182	7353	7436	7439
baeou	12	12	13	-1	-1	100	100	70	106	1042	1042	1087
bch	16	16	16			159	158	88	229	4924	4930	4996
bt7	23	23	22	+1	+1	375	375	276	388	6194	6194	6443
cawe	22	22	22			319	312	189	336	4971	4970	5299
cawemt	22	22	22			293	276	174	302	4233	4249	4504
cju	24	24	28	-4	-4	272	255	159	292	7918	7935	8165
conman1	38	39	41	-3	-2	73	56	16	179	9373	11490	8734
csu	22	22	23	-1	-1	130	130	119	151	6462	6464	6467
dau	20	20	22	-2	-2	87	86	34	120	4340	4309	4021
dblogh	31	31	45	-14	-14	153	152	101	210	6169	6182	6018
dipst6	21	21	24	-3	-3	150	150	88	175	5351	5351	5503
iucda	30	31	33	-3	-2	277	242	163	335	4845	4784	4831
iuiniw	29	29	32	-3	-3	253	256	206	302	4388	4390	4486
iutcph	30	30	36	-6	-6	205	199	145	306	2796	2800	2898
loas	13	13	13			140	139	101	156	3803	3804	3848
maptc	46	46	42	+4	+4	85	86	56	95	2178	2171	2233
mash	22	23	22		+1	40	50	27	61	2087	2058	2078
milo	18	19	19	-1		25	25	17	32	901	901	890
pestu	12	12	12			91	91	77	105	6271	6271	6271
prmcu	12	14	13	-1	+1	76	48	23	82	2768	2798	2836
rceau	25	25	26	-1	-1	211	185	103	292	7201	7044	6832
re	29	29	37	-8	-8	284	251	165	337	8334	8377	8580
rssa	13	13	13			88	86	52	173	6967	6957	6973
scapr26	23	23	27	-4	-4	258	247	144	347	9447	9423	9422
ssint	23	23	22	+1	+1	125	118	70	151	3483	3488	3466
suscfsb	29	29	39	-10	-10	308	275	163	335	10582	10626	10744
teaman3	31	31	32	-1	-1	88	87	26	107	2465	2459	2512
xpreaau	13	13	13			428	428	156	521	13547	13547	13838
TOTAL	661	668	721	-60	-53	5246	5031	3078	6407	160388	162446	161414
%	91,7	92,6	100			81,9	78,5	48,0	100	99,4	100,6	100
						170,4	163,5	100				

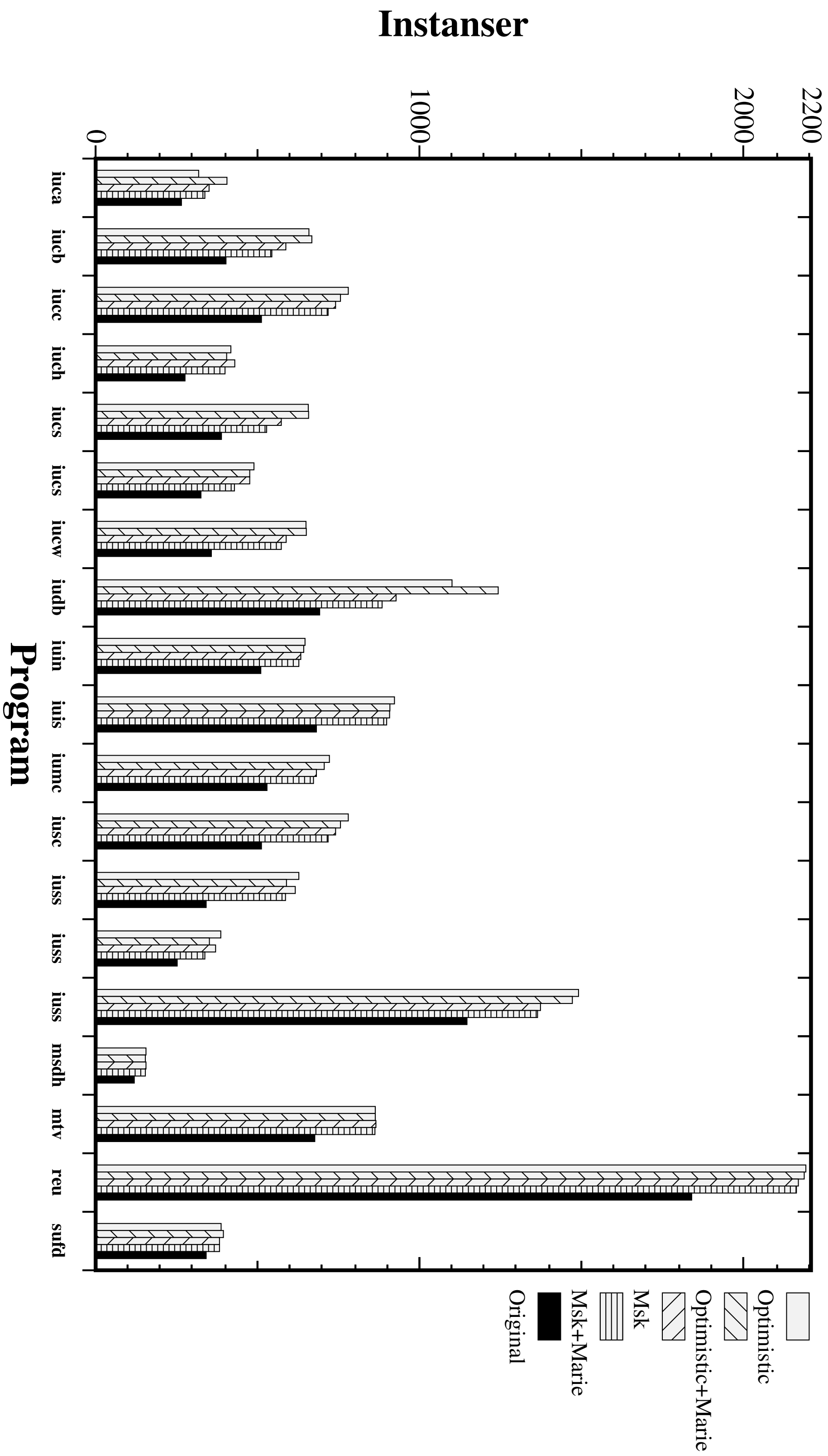
Tabell A.4: Registerallokering utan Maries lag, små program

Program	Register			Reg-Diff		Affinitet				Kodstorlek		
	Opt	Msk	Org	Opt	Msk	Opt	Msk	Org	Max	Opt	Msk	Org
bada	12	13	12		+1	153	108	70	182	7352	7436	7439
baeou	12	12	13	-1	-1	100	100	70	106	1053	1053	1087
bch	16	16	16			155	157	88	229	4934	4930	4996
bt7	23	23	22	+1	+1	378	378	276	388	6203	6201	6443
cawe	21	21	22	-1	-1	318	309	189	336	4974	4978	5299
cawemt	21	21	22	-1	-1	292	275	174	302	4241	4255	4504
cju	24	24	28	-4	-4	267	250	159	292	7950	7963	8165
conman1	38	39	41	-3	-2	67	52	16	179	9451	11547	8734
csu	22	22	23	-1	-1	130	130	119	151	6464	6464	6467
dau	20	20	22	-2	-2	86	83	34	120	4252	4250	4021
dblogh	31	31	45	-14	-14	153	152	101	210	6172	6182	6018
dipst6	21	21	24	-3	-3	152	152	88	175	5380	5351	5503
iucda	30	31	33	-3	-2	272	235	163	335	4872	4813	4831
iuiniw	29	29	32	-3	-3	256	252	206	302	4345	4416	4486
iutcph	30	30	36	-6	-6	200	190	145	306	2957	2845	2898
loas	13	13	13			140	139	101	156	3813	3814	3848
maptc	46	46	42	+4	+4	86	86	56	95	2177	2171	2233
mash	23	23	22	+1	+1	49	49	27	61	2056	2059	2078
milo	18	19	19	-1		25	25	17	32	901	901	890
pestu	12	12	12			91	91	77	105	6271	6271	6271
prmcu	12	14	13	-1	+1	76	48	23	82	2768	2798	2836
rceau	25	25	26	-1	-1	226	183	103	292	7094	7063	6832
re	29	29	37	-8	-8	284	248	165	337	8371	8420	8580
rssa	13	14	13		+1	92	84	52	173	6997	6968	6973
scapr26	23	23	27	-4	-4	262	232	144	347	9589	9601	9422
ssint	23	23	22	+1	+1	126	118	70	151	3493	3530	3466
suscfsb	29	29	39	-10	-10	309	276	163	335	10610	10638	10744
teaman3	31	31	32	-1	-1	88	87	26	107	2472	2461	2512
xpreaau	13	13	13			428	428	156	521	13547	13547	13838
TOTAL	660	667	721	-61	-54	5264	4917	3078	6407	160528	162926	161414
%	91,5	92,5	100			82,2	76,7	48,0	100	99,5	100,9	100
						171,0	159,7	100				

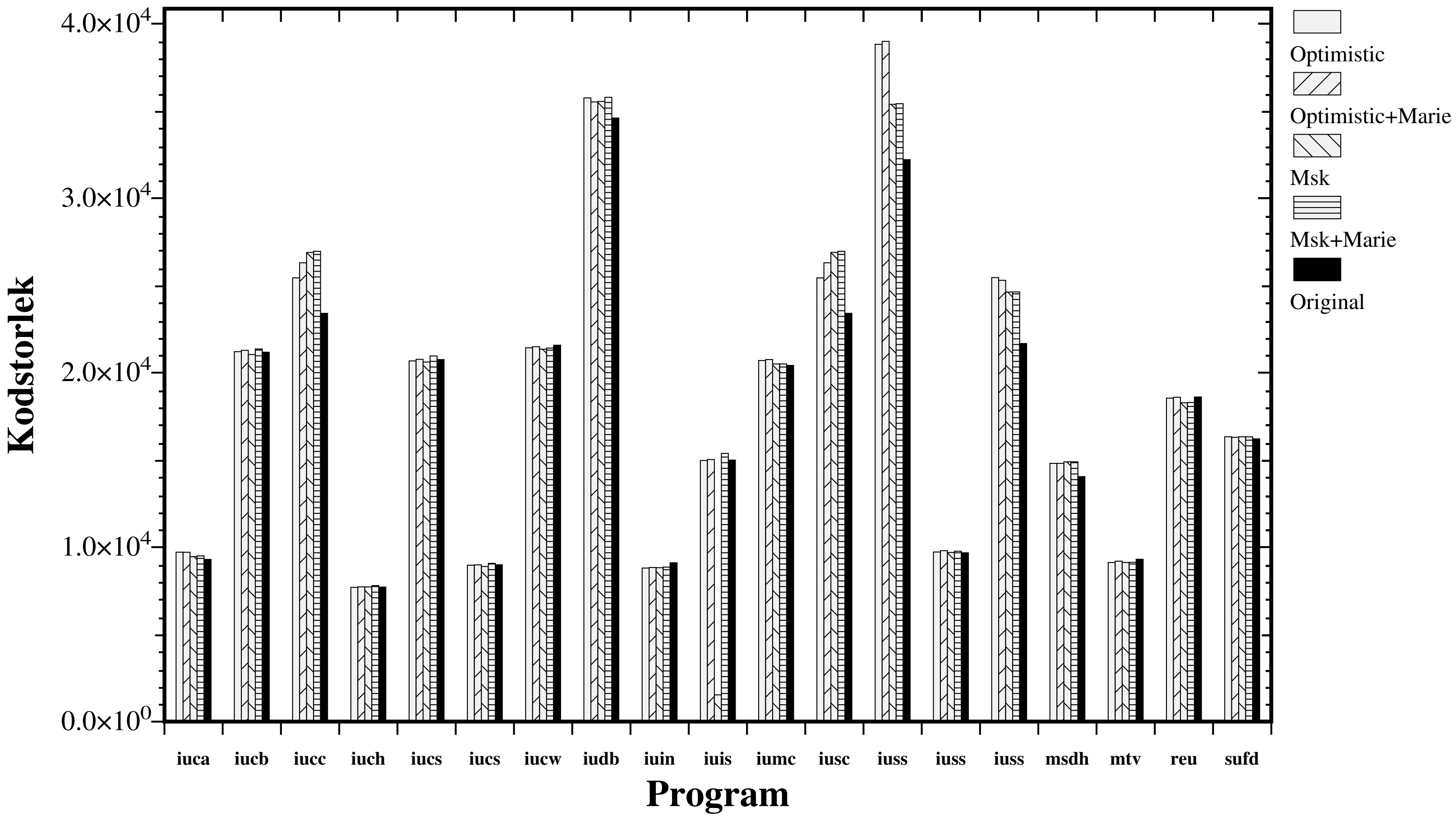
Tabell A.5: Registerallokering med Maries lag, små program



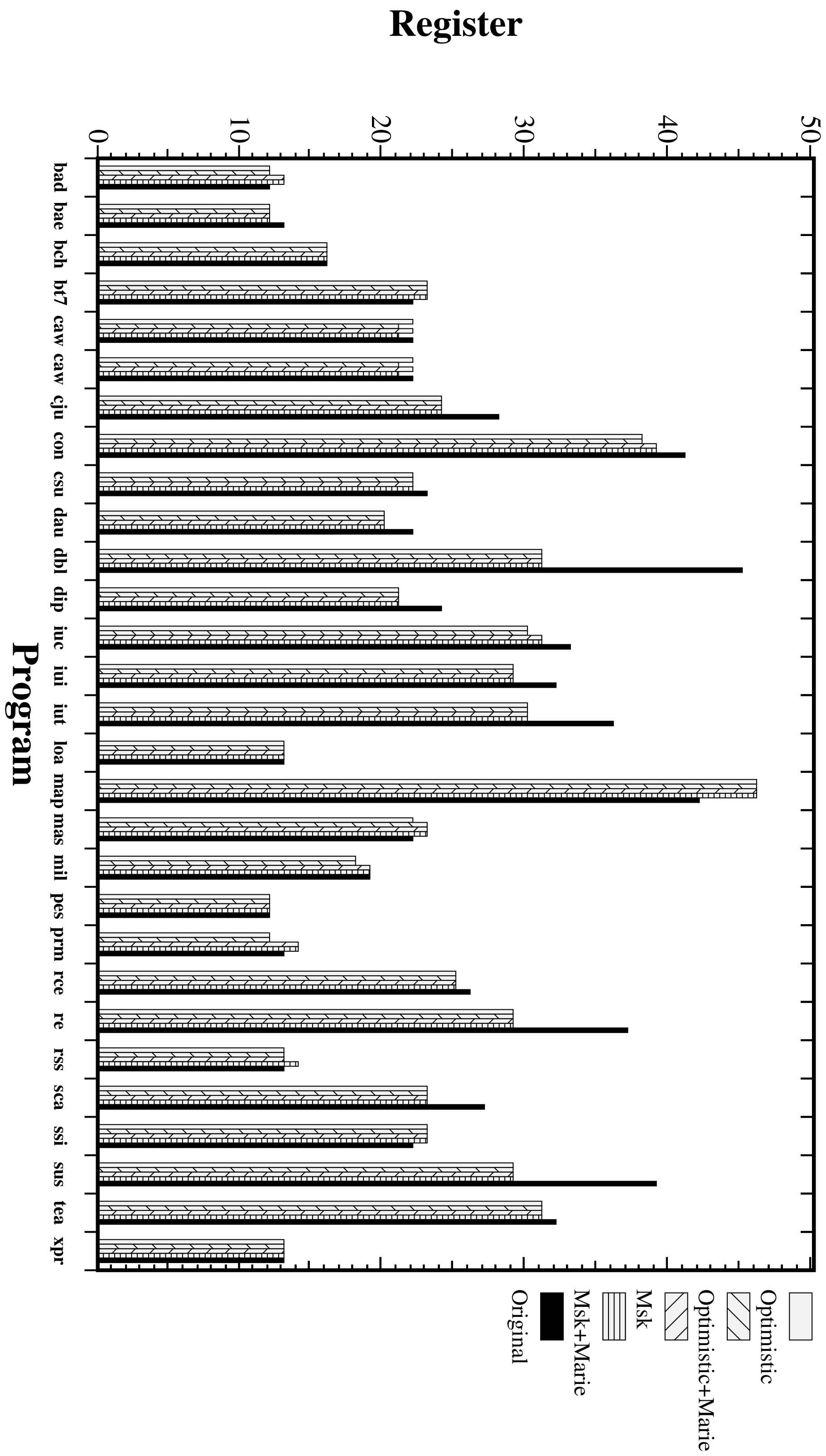
Figur A.1: Registerallokering för programuppsättning 1



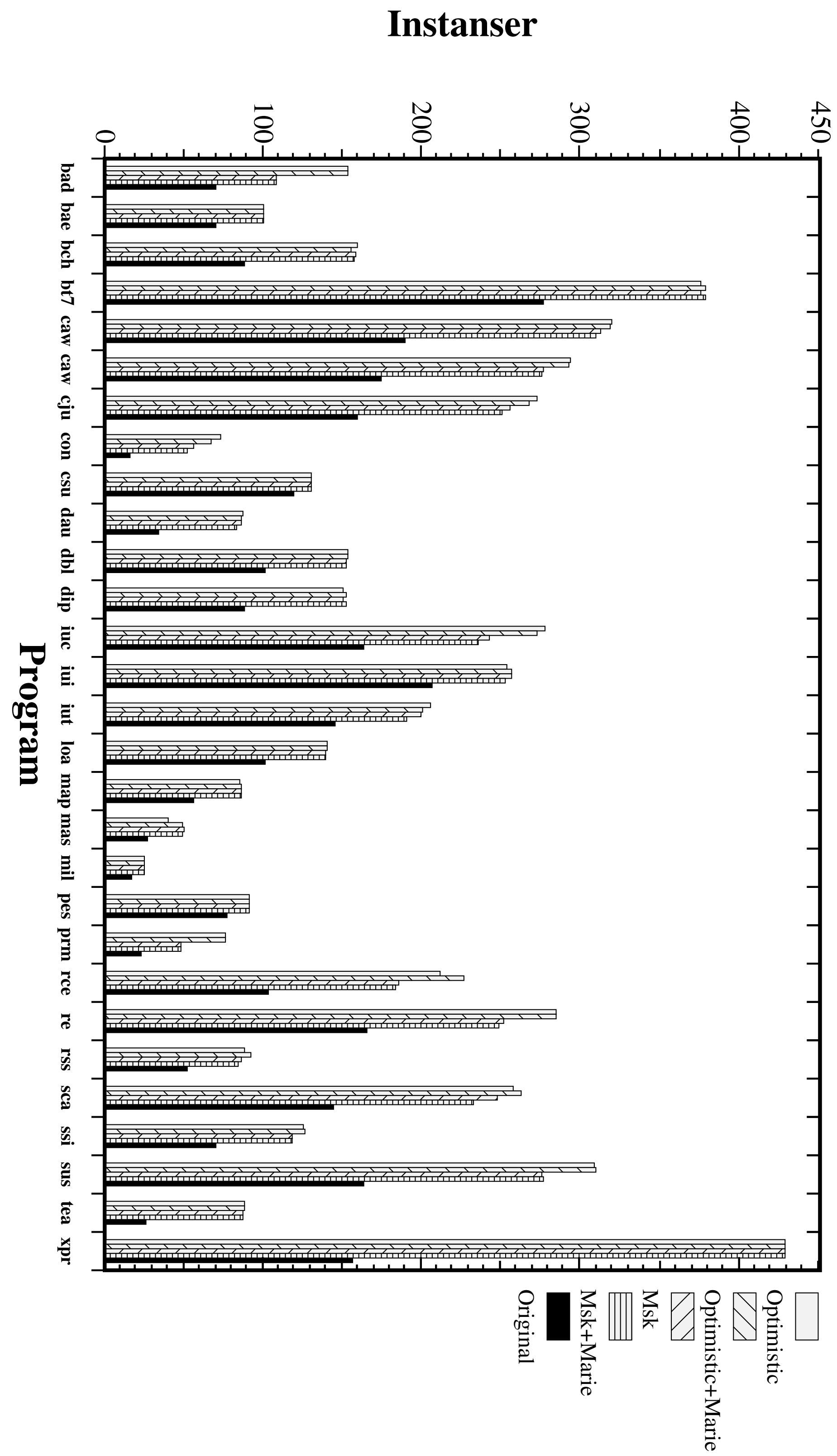
Figur A.2: Tilldelning av affinitetsregister för programuppsättning 1



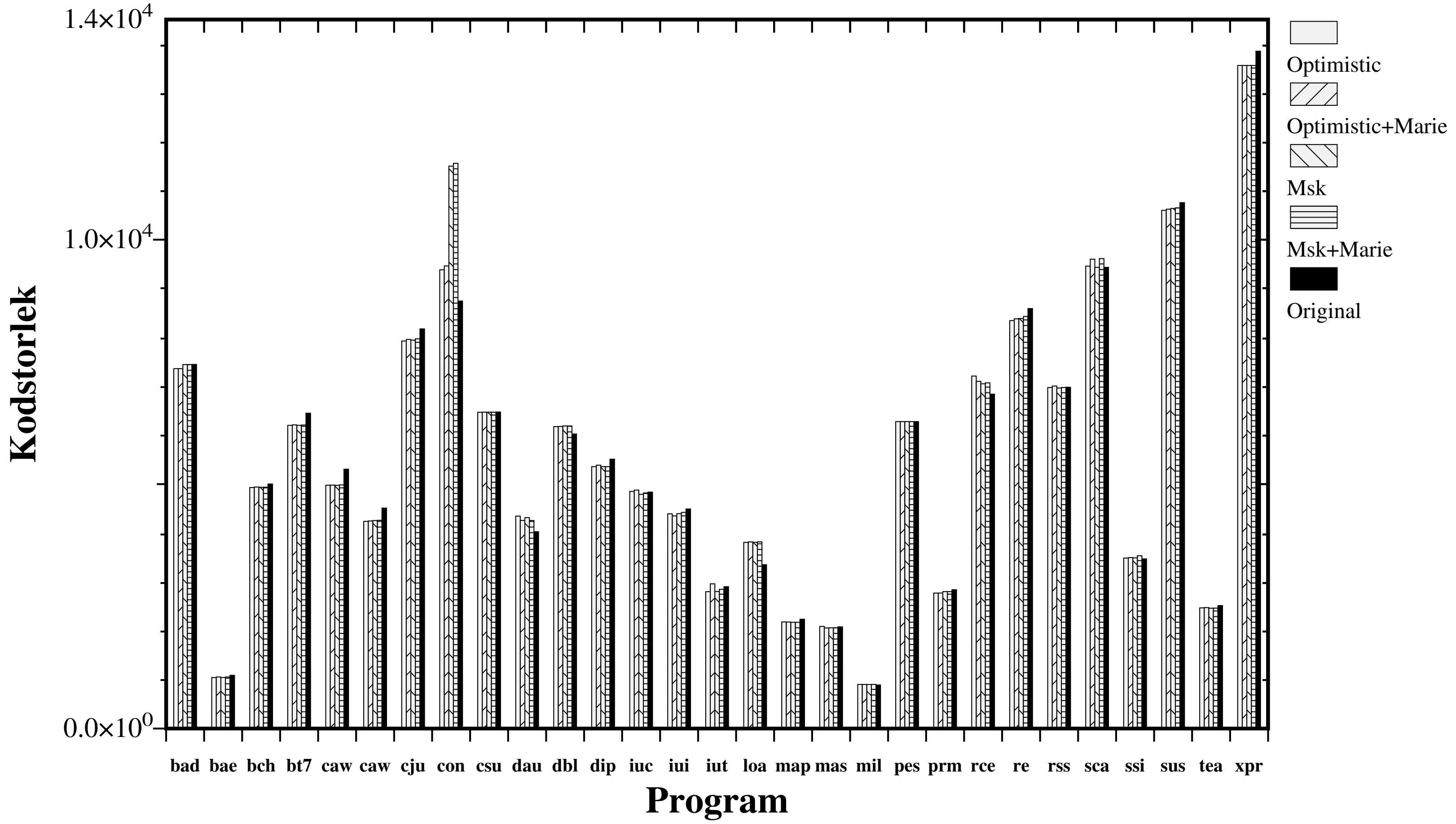
Figur A.3: Kodstorlek för programuppsättning 1



Figur A.4: Registerallokering för programuppsättning 2



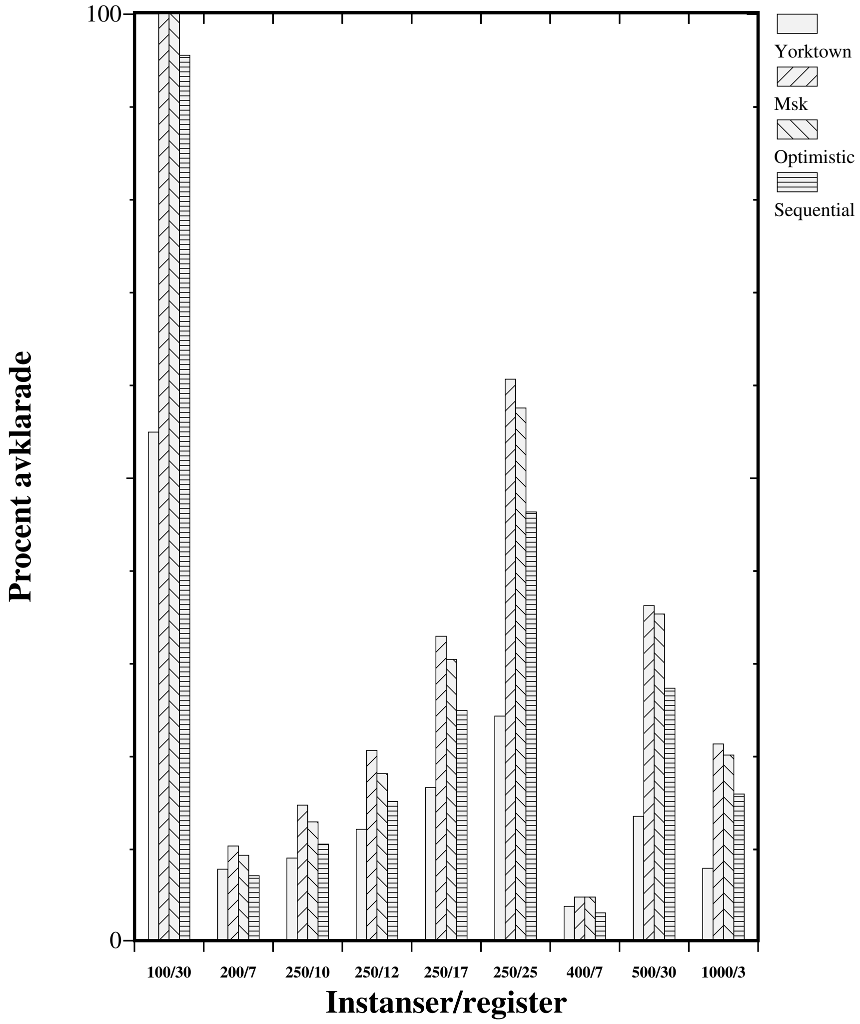
Figur A.5: Tilldelning av affinitetsregister för programuppsättning 2



Figur A.6: Kodstorlek för programuppsättning 2

Instanser/register	Yrk	Msk	Opt	Seq
100/20	54,8%	100%	100%	95,4%
200/7	7,7%	10,2%	9,2%	7,0%
250/10	8,9%	14,6%	12,8%	10,4%
250/12	12,0%	20,5%	18,0%	15,0%
250/17	16,5%	32,8%	30,3%	24,8%
250/25	24,2%	60,5%	57,4%	46,2%
400/7	3,7%	4,7%	4,2%	3,0%
500/30	13,4%	36,1%	35,2%	27,7%
1000/30	7,8%	21,2%	20,0%	15,8%

Tabell A.6: Simuleringar med slumpade interferensgrafer



Figur A.7: Simuleringar med slumpade interferensgrafer

Bilaga B

Övrig pseudokod

Denna bilaga innehåller den pseudokod som är gemensam för de olika registerallokeringsrutinerna vi har utvärderat, dels den som anropas av allokeringsrutinerna, dels den som anropar dem.

B.1 Delad kod

```
" This routine finds instance that can be colored, and gives priority
  to those that have affinity defined[1] "
BEGIN Find Instance That Can Be Colored ( available_colors , nodedomain ).

  " Note: ‘Nodes With[out] Affinity’ are not implemented as functions "
  FOR node IN Nodes Without Affinity ( nodedomain ) DO
    IF Number of Neighbors ( node , nodedomain ) < available_colors THEN
      RETURN node.
    END IF Number of Neighbors.
  END FOR node.

  FOR node IN Nodes With Affinity ( nodedomain ) DO
    IF Number of Neighbors ( node , nodedomain ) < available_colors THEN
      RETURN node.
    END IF Number of Neighbors.
  END FOR node.

  " No valid node was found "
  Exit With ERROR.

END Find Instance That Can Be Colored.

" [1] (so that they will be read back early on, and thus hopefully can be
  allocated to a register they have affinity towards "

" This routine finds the instance with the highest number of neighbors.
  Used by ‘mysko’. "
BEGIN Find Instance With Highest Degree ( nodedomain )
  best = -1.
  register = undef_reg.

  FOR node IN nodedomain DO
```

```
    IF Number of Neighbors ( node , nodedomain ) > best THEN
      best = Number of Neighbors ( node , nodedomain).
      register = node.
    END IF node.
  END FOR node.

  RETURN node.
END Find Instance With Highest Degree

" Color the instance ‘node’ with an available register "
BEGIN Color Instance ( node , smbregs ).

  freesmbregs := { symbolic representations of physical registers }.

  " Traverse all interferands "
  FOR interferand IN inst_interferands DO
    IF Allocated ( interferand ) THEN
      freemsbregs LESS := interferand_smbreg.
    END IF interferand.

    IF freesmbregs = {} THEN
      " All colors taken "
      Exit With ERROR.
    END IF freemsbregs.
  END FOR interferand.

  " Tries to allocate an affinity register first, and if none was found,
    then allocates the first available register. "
  node_smbreg := Determine Affinity Register ( node, freesmbregs ).
  IF no affinity register was found THEN
    node_smbreg :=
      Find Register Without Neighbor Affinity ( node , freesmbregs ).
    IF no affinity register was found THEN
      node_smbreg := FirstInAllocOrder ( freesmbregs ).
    END IF no affinity.
  END IF no affinity.

  RETURN node_smbreg.

END Color Instance.


" Find the number of neighbors to ‘node’ in the interference graph that is
  contained in the set ‘nodedomain’"
BEGIN Number of Neighbors ( node , nodedomain )

  count := 0.

  FOR interferand IN node_interferands DO
    IF interferand IN nodedomain THEN
      count + := 1.
    END IF interferand.
```

```
  END FOR interferand.

  RETURN count.

END Number of Neighbors


" Find the best matching affinity register for the instance ‘node’ in
  the set of free registers ‘regs’ "
" Maybe the old code is better? "
BEGIN Determine Affinity Register ( node, regs )

  " Create a set of all free affinity registers "
  freeaffregs = {}.
  FOR register IN regs DO
    IF inst_affregs(register) > 0 THEN
      freeaffregs WITH := register.
    END IF register.
  END FOR register.

  " Remove registers unallocated interferands have a higher affinity to "
  FOR register IN freeaffregs DO
    total_used := 0.
    removed := FALSE.

    FOR interferand IN inst_interferands DO
      IF NOT Is Allocated ( interferand ) THEN
        IF interferand_affregs[register] > inst_affregs[register] THEN
          freeaffregs LESS := register.
          removed := TRUE.
          QUIT FOR interferand.
        ELSE
          total_used + := interferand_affregs[register].
        END IF interferand_affreg.
      END IF Is Allocated.
    END FOR interferand.

    " Our adaptation of Marie’s Law "
    IF NOT removed THEN
      " PR0 is only given to pointers with _strong_ affinity, or
        undisturbed temporaries "
      IF register = PR0 THEN
        IF NOT ((var = ptr AND inst_affregs[PR0] >= MARIE * total_used)
                OR
                (var = temp AND total_used = 0)) THEN
          freeaffregs LESS := PR0.
        END IF NOT.
      END IF register.

      " IR is only given to temporaries with _strong_ affinity, or
        undisturbed pointers "
      IF register = IR THEN
```

```
        IF NOT ((var = temp AND inst_affregs[IR] >= MARIE_IR * total_used)
                OR
                (var = ptr AND total_used = 0)) THEN
          freeaffregs LESS := IR.
        END IF NOT.
      END IF register.

      IF register IN [DR0...DR23] THEN
        IF inst_affregs[register] >= MARIE_DR*totalUsed THEN
          freeaffregs LESS := register.
        END IF inst_affregs
      END IF register

    END IF NOT removed.
  END FOR register.

  " Select the best register from those which are left "
  IF freeaffregs is empty THEN
    RETURN undef_reg.
  END IF freeaffregs.

  affinity_reg := undef_reg.
  max_used     := 0.
  FOR register IN freeaffregs DO
    IF inst_affregs(. register .) > max_used THEN
      affinity_reg := register.
      max_used := inst_affregs ( register ).
    END IF inst_affregs.
  END FOR register.

  RETURN affinity_reg.

END Determine Affinity Registers

" See if we can find a free register that no yet unallocated neighbor
  has affinity to "
BEGIN Find Register Without Neighbor Affinity ( node , regs )

  FOR register IN regs DO
    affinity := FALSE.

    FOR interferand IN node_interferands DO
      IF NOT Is Allocated ( interferand ) AND
         interferand_affreg[register] > 0 THEN
        affinity := TRUE.
        QUIT FOR interferand.
      END IF interferand_affreg.
    END FOR interferand.

    IF NOT affinity THEN
      RETURN register.
    END IF NOT affinity.
```

```
  END FOR register.

  RETURN undef_reg.

END Find Register Without Neighbor Affinity.
```

B.2 Huvudslinga

```
" This is the routine that calls the register allocation routine itself "
BEGIN Determine the Graph's Allocatability
        ( graphstatus , startinst-input , blockedinst , maxnrsmbregs ).

  IF startinst NOT INITIALIZED THEN
    " static startinst "
    startinst = startinst-input.
  END IF startinst.

  FOR inst IN Global Instance Information DO

    " skip spilled instance "
    IF inst_status = spilled THEN
      CONTINUE FOR inst.
    END IF inst_status.

    inst_interfersmbregs := {}.
    inst_smbreg := undef_reg.   " Remove all allocations "
  END FOR inst.

  Allocate Symbolic Registers to the Instances ( startinst , maxnrsmbregs ).
  Search for Blocked Instance ( graphstatus , blockedinst ).

END Determine the Graph's Allocatability

" This rouine searches the graph for the first instance that could not
  be allocated, and spills it "
BEGIN Search for Blocked Instance ( startinst , graphstatus , blockedinst ).

  FOR inst IN Global Instance Information FROM startinst DO

    " skip spilled instances "
    IF inst_status = spilled THEN
      CONTINE FOR inst.
    END IF inst_status.

    IF NOT Allocated ( inst ) THEN
      blockedinst := inst.
      graphstatus := unallocatable.
      QUIT FOR inst.
    END IF NOT Allocated.
  END FOR inst.
```

```
END Search for Blocked Instance.
```

B.3 Spillkod

```
" Searches for an instance that was not coloured, and that can be spilled "
BEGIN Search for an Instance to Spill ( blockedinst , spillinst , spillvar )

  largest := 0.

  " Collect a set of the unallocated instances and their interferands "
  SPILLCANDIDATES := {}.
  FOR inst IN Global Instance Information FROM blockedinst DO

    " skip spilled instances "
    IF inst_status = spilled THEN
      CONTINUE FOR inst.
    END IF inst_status.

    " skip allocated instances "
    IF Allocated ( inst ) THEN
      CONTINUE FOR inst.
    END IF Allocated.

    SPILLCANDIDATES WITH := inst.

    FOR interferand IN inst_interferands DO
      " skip spilled instances "
      IF inst_status /= spilled AND interferand > nroccregs THEN
          SPILLCANDIDATES WITH := interferand.
      END IF inst_status.
    END FOR inteferand.

  END FOR inst.

  " Traverse the set and see which one is best to spill "
  FOR inst IN SPILLCANDIDATES DO
    spillvar := inst_var.

    " skip unspillable instances "
    IF spillvar_type = pointer THEN
      CONTINUE FOR inst.
    END IF.

    IF spillvar_syref ( 1 ) = '%' THEN
      CONTINUE FOR inst.
    END IF.

    " Find out how many interfering symbolic registers this instance has "
    FOR interferand IN inst_interferands DO
      inst_interfersmbregs WITH := interferand_smbreg.
    END FOR interferand.
```

```
    spillcriterion := inst_interfersmbregs / spillvar_spillcost.

    IF largest <= spillcriterion THEN
      largest   := spillcriterion.
      spillinst := inst.
    END IF largest.

    inst_interfersmbregs := {}.

  END FOR inst.

  spillvar := spillinst_var.

END Search for an Instance to Spill
```

Författarnas tack

Författarna önskar tacka Anton Massoud på Ericsson Utvecklings AB för ovärderligt stöd under tiden vi arbetade på examensarbetet där.

Vi vill även tacka vår handledare på Mälardalens högskola, Kjell Post, som alltid finns tillgänglig för frågor.

Även ett litet tack till våra kamrater Andreas A, Andreas G och Camilla för deras synpunkter och trevligt sällskap under den tid de, samtidigt som vi, gjorde sitt exjobb på Ericsson.

Litteraturförteckning

[1] Alfred V. Aho, Ravi Sethi, och Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*, kapitel 9. Addison-Wesley Publishing Company, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation In Java: Basic Techniques*, kapitel 11. Cambridge University Press, 1997.

[3] Urban Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. Doktorsavhandling, Chalmers University of Technology, Göteborg University, 1999.

[4] Preston Briggs. *Register Allocation via Graph Coloring*. Doktorsavhandling, Rice University, 1992.

[5] G[regory] J. Chaitin. Register allocation & spilling via graph coloring. Ur *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, årgång 17, sid 98–105, 1982.

[6] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, och Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.

[7] Ericsson Telecom AB. *PLEX-C Language Description*, 1994.

[8] Ericsson Telecom AB. *PLEX-C 1*, 1997.

[9] Patric Hedlin. A prestudy survey on register allocation. Ericsson internal memo, juni 1999.

[10] L. J. Hendren, G. R. Gao, E. R. Altman, och C. Mukerji. A register allocation framework based on hierarchical cyclic interval graphs. Technical Memo 33, McGill Universtity, februari 1993.

[11] David W. Matula och Leland L. Beck. Smallest-last ordering and clustering and graph coloring algorithms. *Journal of the ACM*, 30(3):417–427, juli 1983.

[12] SoftLab. *Code Generator Temporary Variable Allocator Function*, juni 1993. Ericsson No. 1/1551 - CNA 377 09.

[13] SoftLab. *Code Generator Basic Block Analyser Function*, juli 1997. Ericsson No. 1/1551 - CNA 377 08.