HÖGSKOLAN
SKÖVDE
Institutionen för kommunikation och information

Optimering av ARM-maskinkod med predikatbaserad exekvering

En undersökning av predikatbaserad exekvering i ARM-arkitekturen

Marcus Barstorp

Examensarbete i datalogi med inriktning mot dataspelsutveckling
30 hp C-nivå, vårterminen 2010

Optimering av ARM-maskinkod med predikatbaserad exekvering

Examensrapport inlämnad av Marcus Barstorp till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för kommunikation och information. Arbetet har handletts av Thomas Fischer.

2010-08-30

Härmed intygas att allt material i denna rapport, vilket inte är mitt eget, har blivit tydligt identifierat och att inget material är inkluderat som tidigare använts för erhållande av annan examen.

Signerat: ____

Optimering av ARM-maskinkod med predikatbaserad exekvering

Marcus Barstorp

Handledare: Thomas Fischer

Student-email: e07marba@student.his.se

Sammanfattning

Arbetet har undersökt hur predikatbaserad exekvering, i form av ARM-arkitekturens stöd för villkorlig exekvering, fungerar som optimering av specifika implementationer av if- och if-else-satser i ARM-maskinkod. Som en del av arbetet implementerades en för ändamålet konstruerad optimeringsalgoritm som optimerar if- och if-else-satser med hjälp av predikatbaserad exekvering, vilken användes för att kunna utföra de mätningar som ligger till grund för resultatet. Fokus låg på den skillnad i kodstorlek och/eller tidseffektivitet som optimeringen gav upphov till. Resultatet tyder på att predikatbaserad exekvering använd som optimering kan leda till vissa vinster i kodstorlek (direkt beroende på antalet if-/if-else-satser) och lite större vinster i tidseffektivitet (beroende på hur stora if-/if-else-satserna är, samt hur ofta de exekveras).

Nyckelord: Optimering, predikatbaserad exekvering, ARM, maskinkod

Innehållsförteckning

1 Introduktion....... 1

2 Bakgrund....... 2

2.1 ARM-arkitekturen....... 2

2.2 Pipelining....... 2

2.2.1 Pipelining inom ARM-arkitekturen....... 3

2.3 Hoppinstruktioner....... 3

2.3.1 Hoppinstruktioner inom ARM-arkitekturen....... 4

2.4 Pipelining och hoppinstruktioner....... 5

2.4.1 Pipelining och hoppinstruktioner inom ARM-arkitekturen....... 5

2.5 Predikatbaserad exekvering....... 6

2.5.1 Predikatbaserad exekvering inom ARM-arkitekturen....... 6

2.6 Predikatbaserad exekvering och optimering....... 7

3 Problemformulering....... 9

3.1 Optimering med optimeringsalgoritmen....... 9

3.1.1 Problemavgränsning....... 10

3.2 Metodbeskrivning....... 10

4 Genomförande....... 12

4.1 Det producerade systemet....... 12

4.1.1 Översiktligt om optimeringsalgoritmen....... 12

4.1.2 Avgränsning av optimeringsalgoritmen....... 12

4.1.3 Hur optimeringsalgoritmen analyserar koden....... 12

4.1.1 Applikationen och optimeringsalgoritmens implementation....... 15

4.2 Genomförda mätningar....... 17

4.2.1 Testplattformar....... 17

4.2.2 Storlekstest av grundblock....... 17

4.2.3 Bubblesort....... 22

4.2.4 Mergesort....... 23

4.2.5 Resultat för optimeringsalgoritmen....... 24

4.3 Analys av mätningar....... 25

4.3.1 Kodstorlek....... 25

4.3.2 Tidseffektivitet....... 26

5 Slutsatser....... 28

5.1 Resultatsammanfattning 28
5.2 Diskussion 28
5.3 Framtida arbete 29
Referenser **31**
Bilaga A **A-1**
A.1 Storlekstest av grundblock A-1
A.2 Bubblesort A-6
A.3 Mergesort A-7

1 Introduktion

En egenskap hos kod är att den kan skrivas på en mängd olika sätt och ändå lösa samma problem. Till detta hör också att kodens effektivitet varierar, vilket innebär att det finns lösningar som är bättre än vissa och sämre än andra, beroende på den plattform programmet ska köras på. Hyde (2006) definierar ett programs effektivitet (och därmed kodens effektivitet) som ett minimerat utnyttjande av en resurs, där resurs främst avser minnesutrymme och hastighet (antal klockcykler/total exekveringstid). För att förbättra, eller optimera, koden kan man alltså utföra olika förändringar av koden vilka syftar till att minska minnesåtgången (minska det utrymme programmet tar upp i minnet) och/eller att öka hastigheten (minska antalet klockcykler programmet använder).

Sloss, Symes och Wright (2004) beskriver en egenskap hos merparten av de instruktioner som implementeras av processorer tillhörande ARM-arkitekturen, kallad *villkorlig exekvering* (eng. *conditional execution*). Villkorlig exekvering är ARM-arkitekturens sätt att stödja *predikatbaserad exekvering* (eng. *predicated execution*), vilket enligt August, Hwu och Mahlke (1997) innebär att exekveringen av enskilda instruktioner avgörs beroende på om ett visst predikat, eller villkor, håller eller inte. Om villkoret håller exekveras instruktionen, och om det inte håller så ignoreras instruktionen. Sloss, et al. (2004) förklarar att villkorlig exekvering av instruktioner till ARM-processorer kan användas för att bl.a. reducera antalet hoppinstruktioner (eng. *branch instructions*). Detta medför att villkorlig exekvering kan användas som en specifik optimering av ARM-maskinkod med avseende på hastighet (färre instruktioner ger snabbare exekvering) och minnesutrymme (färre instruktioner ger mindre kodstorlek).

Enligt Sloss, et al. (2004) är ARM-processorer vanliga i inbäddade system, som t.ex. mobiltelefoner, och har ofta en begränsad mängd minne och processorkraft. Optimeringar av program, som exempelvis dataspel, kan därmed bidra till att de redan begränsade resurserna används effektivare, vilket detta examensarbete ämnar att försöka visa när det gäller predikatbaserad exekvering som optimering.

Detta examensarbete fokuserar på att undersöka huruvida predikatbaserad exekvering, i form av ARM-arkitekturens stöd för villkorlig exekvering, kan användas för att optimera ARM-maskinkod med avseende på minnesutrymme och hastighet. För att kunna utföra optimeringen på ett konsekvent sätt implementeras en för ändamålet framtagen optimeringsalgoritm som en del av detta examensarbete. Optimeringsalgoritmen använder villkorlig exekvering för att optimera program skrivna i ARM-maskinkod, och till en början avgränsas optimeringsalgoritmen så att den kan optimera konstruktioner som nyttjar hoppinstruktioner motsvararande if- och if-else-satser.

Undersökningen baseras på ett antal program skrivna i ARM-maskinkod, vilka optimeras av optimeringsalgoritmen. De ickeoptimerade versionerna av programmen jämförs sedan mot de optimerade versionerna av programmen för att se om och hur de skiljer sig åt med avseende på hastighet och minnesutrymme. För att utvärdera skillnader i hastighet exekveras de ickeoptimerade och optimerade versionerna av programmen, vartefter skillnaden i exekveringstid mäts. Skillnader i minnesutrymme, d.v.s. kodstorlek, mäts direkt genom att jämföra antalet instruktioner i maskinkoden som utgör programmen före respektive efter optimering.

2 Bakgrund

Detta kapitel ägnas åt att ge en bakgrund till och förklara villkorlig exekvering och hur den kan användas, samt lite teoretisk bakgrund av vikt för optimeringsalgoritmen.

2.1 ARM-arkitekturen

Företaget ARM:s processorkärnor är, enligt Sloss, Symes och Wright (2004), en komponent i många framgångsrika 32-bitars inbäddade system och används i en mängd konsumentartiklar (bl.a. mobiltelefoner). De olika processorkärnorna har inte exakt samma design, utan delar en gemensam instruktionsuppsättningsarkitektur (hädanefter ARM ISA, eng. *ARM Instruction Set Architecture*) och liknande designprinciper.

Enligt Sloss, et al. (2004) har ARM ISA:n utökats successivt genom en serie revisioner, som bl.a. lägger till instruktioner och ibland hårdvaruegenskaper, vilka man till stor del låtit vara bakåtkompatibla med varandra. Sloss, et al. (2004) förklarar vidare att ARM-processorerna delas in i processorfamiljer efter vilken processorkärna de använder, vilka skiljer sig åt beroende på vilken revision av ARM ISA:n de implementerar samt eventuella tillägg till hårdvaran. Program gjorda för processorkärnor som implementerar äldre revisioner ska dock fortfarande kunna köras på de som implementerar yngre revisioner.

En grundpelare i ARM ISA:ns design är att den är *RISC-baserad*. RISC står för *Reduced Instruction Set Computer* och betyder, fritt översatt från engelska, reducerad instruktionsuppsättningsdator. Sloss, et al. (2004) understryker att detta bl.a. innebär att instruktionerna är lika långa (i antal bitar) och tar en klockcykel vardera för processorn att exekvera. En annan egenskap hos RISC-baserade processorer (inkl. ARM-processorerna) är att de använder sig av en *pipeline* för att parallellisera exekveringen av sina instruktioner, vilket kapitel 2.2 behandlar närmare. Sloss, et al. (2004) beskriver fortsatt att de har ett stort antal generella register som kan användas både för att lagra och manipulera data och minnesadresser med hjälp av instruktioner som arbetar på datan i registren. Dock finns inga instruktioner för att manipulera data direkt i arbetsminnet, utan bara för att hämta/skicka data mellan register och arbetsminne.

Sloss, et al (2004) förklarar att, för att ARM-arkitekturen bättre ska passa inbäddade system, dess designers bl.a. har lagt stor vikt vid minskad strömförbrukning, hög koddensitet (program upptar mindre minnesutrymme ju högre koddensiteten är) och även processorns fysiska storlek. Dessa egenskaper relaterar till vanliga begränsningar hos inbäddade system som t.ex. batteridrift, lite arbetsminne och lite fysiskt utrymme (för kretsarna) vilket gäller t.ex. mobiltelefoner och digitalkameror. Sloss, et al. (2004) poängterar att ARM-arkitekturens designers valt att göra vissa avsteg från RISC-filosofin vilket bl.a. visar sig i tillägget av en 16-bitars instruktionsuppsättning kallad Thumb (för att åstadkomma högre koddensitet) från och med ARM ISA version 4T, vilket går emot RISC-principen om att alla instruktioner ska ha samma längd (i ARM-arkitekturens fall 32-bitar). Processorkärnor från och med denna version av ARM-arkitekturen har ett exekveringsläge för att exekvera Thumb-instruktioner (dock inte samtidigt som vanliga ARM-instruktioner).

2.2 Pipelining

Enligt Brorsson (1999) insåg man tidigt under datorns utveckling att exekveringen av en instruktion inte behöver ses som ett odelbart steg som upptar hela processorns maskinvara, utan att det kan ses som två delsteg som utförs av två delar av processorns maskinvara. Brorsson poängterar att maskinvaran för att t.ex. hämta en instruktion från minnet, kan utföra

uppgiften självständigt från maskinvaran för att exekvera instruktionen. Av den anledningen kan de separeras och utföra sitt arbete på varsin instruktion parallellt.

Pipelining som teknik är en förlängning av detta koncept och innebär, enligt Abd-El-Barr och El-Rewini (2005) att en uppgift delas upp i flera mindre delsteg, där varje steg är beroende av att föregående steg utförts, varför pipelining kan liknas vid systemet med det löpande bandet i en bilfabrik. Abd-El-Barr och El-Rewini menar att eftersom varje steg utför en del av exekveringen av en instruktion kan vart och ett av stegen arbeta med en instruktion vardera samtidigt, vilket skiljer sig från en processor som exekverar en hel instruktion åt gången (sekventiellt, en och en).

Brorsson (1999) påpekar att pipelining inte gör att exekveringen av enskilda instruktioner går snabbare, utan att antalet exekverade instruktioner (genomströmningen) ökar. Detta beror enligt Sloss, et al. (2004) på att då pipelinen fyllts, d.v.s. när varje steg arbetar med varsin instruktion, kommer en instruktions exekvering slutföras varje klockcykel.

När det gäller hastighetsökningen med hjälp av pipelining förklarar Brorsson (1999) att klockfrekvensen för en processor utrustad med en pipeline styr takten med vilken instruktioner flyttas från ett steg till nästa igenom pipelinens alla steg, varför den av den anledningen inte kan vara så hög att det långsammaste steget inte hinner bli klart. För att kunna höja klockfrekvensen förklarar Sloss, et al. (2004) att ju längre en pipeline blir, d.v.s. ju fler steg den består av, desto mindre arbete måste hårdvaran för varje steg utföra. Detta leder i sin tur till att alla steg blir klara snabbare och vilket innebär att klockfrekvensen kan ökas. Både Brorsson (1999) och Sloss, et al. (2004) menar dock att det inte bara är att lägga till fler och fler steg i pipelinen för att öka klockfrekvensen; för det första för blir det svårare att balansera tidsåtgången jämnt för varje steg, för det andra kan det finnas beroenden mellan instruktioner som exekveras i de olika stegen och för det tredje så tar det längre tid att fylla pipelinen.

I kapitel 2.3 behandlas en typ av instruktioner, *hoppinstruktioner* (eng. *branch/jump instructions*), som orsakar problem för processorer utrustade med en pipeline. Själva problemet behandlas i kapitel 2.4, och handlar om hur hoppinstruktioner kan orsaka en tömning av pipelinen (eng. *pipeline flush*).

2.2.1 Pipelining inom ARM-arkitekturen

Pipeline-designen skiljer sig lite åt mellan ARM-arkitekturens processorfamiljer. Sloss, et al. (2004) nämner att ARM7-familjen t.ex. använder en tredelad pipeline med stegen hämtning (eng. *fetch*) som hämtar en instruktion från minnet, avkodning (eng. *decode*) som identifierar instruktionen som ska exekveras samt exekvering (eng. *execute*) som exekverar instruktionen. De yngre processorfamiljerna ARM9, ARM10 och ARM11 har, enligt Sloss, et al. (2004), istället utökat denna pipeline till fem, sex respektive åtta steg. Program som skapats till en processor i t.ex. ARM7-familjen kan dock fortfarande köras på processorer tillhörande en av de senare processorfamiljerna då deras pipelines delar samma exekveringegenskaper.

2.3 Hoppinstruktioner

Abd-El-Barr och El-Rewini (2005) beskriver hoppinstruktioner som en typ av instruktioner som kan användas för att utföra ett hopp från ett kodavsnitt i ett program till ett annat. Ett hopp kan ske antingen till ett tidigare kodavsnitt i programmet eller till ett kodavsnitt längre fram i programmet, och beroende på vilken typ av hoppinstruktion som avses kan hoppet antingen exekveras när ett visst villkor håller (villkorligt hopp) eller varje gång (ovillkorligt hopp). Abd-El-Barr och El-Rewini (2005) förklarar fortsättningsvis att om villkoret för en

villkorlig hoppinstruktion inte håller utförs inte hoppet, och processorn fortsätter istället att hämta och exekvera instruktioner som ligger direkt efter hoppinstruktionen (snarare än från hoppinstruktionens destination).

Olika typer av hoppinstruktioner delar enligt Abd-El-Barr och El-Rewini (2005) egenskapen att de hoppar relativt ett kodavsnitt i programmet som pekas ut av *programräknaren* (eng. *program counter*), ä.k. instruktionspekaren (eng. *instruction pointer*), som är ett register som pekar ut den minnesadress från vilken processorn ska hämta nästa instruktion (för exekvering) ifrån. Abd-El-Barr och El-Rewini (2005) förklarar att själva hoppet utförs genom ändra den minnesadress programräknaren för närvarade innehåller, vilket innebär att processorn börjar hämta instruktioner för exekvering från och med denna nya minnesadress.

När det gäller själva villkoret, som används av processorn för att avgöra om ett villkorligt hopp ska utföras eller inte, beskriver Abd-El-Barr och El-Rewini (2005) hur ett särskilt register (på engelska kallat *condition code register*) används för att testa om ett villkor håller eller inte. Värdena, av Sloss, et al. (2004) kallat villkorsflaggor (eng. *condition flags*), i registret sätts beroende på resultatet från vissa tidigare exekverade instruktioner, och används för att representera de villkor som de villkorliga hoppinstruktionerna använder.

2.3.1 Hoppinstruktioner inom ARM-arkitekturen

Inom ARM-arkitekturen finns det, vilket Sloss, et al. (2004) beskriver, två hoppinstruktioner bortsett från de två hoppinstruktioner som berör exekvering av Thumb-instruktioner. Den första instruktionen, B (för eng. *branch*), utför ett vanligt PC-relativt hopp, och den andra, BL (för eng. *branch with link*), utför ett PC-relativt hopp som även sparar minnesadressen för den instruktion som ligger direkt efter hoppinstruktionen i programmet. Den sparade minnesadressen kan användas för att hoppa tillbaka till instruktionen efter ett det ursprungliga hoppet, vilket används för att kunna utföra subrutinanrop. Om man vill kan man, enligt Sloss et al. (2004), istället för att använda hoppinstruktionerna direkt ändra minnesadressen i programräknaren med en vanlig instruktion vilket har samma effekt som ett hopp orsakat av en hoppinstruktion.

Sloss, et al. (2004) beskriver hur villkorsflaggorna lagras tillsammans med processorns övriga statusflaggor i ett register kallat *cpsr* (för eng. *Central Processor Status Register*), och just villkorsflaggorna utgör i olika kombinationer specifika villkor. Om t.ex. hoppinstruktionerna B eller BL anges utan villkor, eller med det särskilda villkoret AL (alltid, från eng. *always*) så exekveras de alltid. Om de däremot kombineras med något annat villkor exekveras de förutsatt att villkoret håller (vilket det gör om det matchar de villkorsflaggor som är satta i cpsr). Sloss, et al (2004) beskriver fortsatt att villkorsflaggorna sätts av särskilda instruktioner som jämför ett register med ett annat register (alternativt ett heltalsvärde) och sätter villkorsflaggorna baserat på resultatet av jämförelsen. Vanliga instruktioner kan också sätta villkorsflaggor om suffixet S anges tillsammans med instruktionen.

Nedan följer ett exempel på en tillämpning med två hoppinstruktioner och suffixet S; maskinkodsavsnittet består av en nästlad oändlig loop där ytterloopen initierar registren och innerloopen beräknar summan av de tio första heltalen. Villkorsflaggorna sätts endast av SUB-instruktionen (p.g.a. suffixet S), och hoppinstruktionen BGE kommer därför att utföras så länge r0 är större än eller lika med 1 (villkoret är GE och står för ***g**reater than or **e**qual*). Hoppinstruktionen B (samt instruktionerna ADD och MOV) körs alltid eftersom villkoret AL är implicit.

Exempel 1

```
outer_loop
     MOV r0, #10
     MOV r1, #0
inner_loop
     ADD r1, r1, r0  ; r1 = r1+r0
     SUBS r0, r0, #1 ; r0 = r0 – 1
     BGE inner_loop  ; hoppa till inner_loop om r0 >= 1
     B outer_loop    ; hoppa (alltid) till outer_loop
```

2.4 Pipelining och hoppinstruktioner

Abd-El-Barr och El-Rewini (2005) förklarar att hoppinstruktioner utgör ett särskilt problem för en processor med en pipeline. Problemet är att de modifierar programräknaren, villket påverkar varifrån i programmet nästa instruktion ska hämtas. Detta leder enligt Abd-El-Barr och El-Rewini till att pipelinen blir fördröjd (eng. *pipeline stall*) eftersom den måste ”vänta” på att en hoppinstruktion exekverats så att programräknaren, beroende på om hoppet utförts eller ej, innehåller rätt minnesadress från vilken nästa instruktion ska hämtas. Notera att själva utvärderingen av villkoren gentemot villkorsflaggorna är en del av ett pipelinesteg, och kostar alltså inget (det stör inte pipelinen).

För ovillkorliga hoppinstruktioner, som alltid utförs, finns det enligt Abd-El-Barr och El-Rewini (2005) ett antal tekniker som kan användas för att på olika sätt reducera eller förebygga fördröjning av pipelinen. Ett exempel på en av dessa tekniker går ut på att redan när en ovillkorlig hoppinstruktion hämtats beräkna och hämta instruktioner från den minnesadress (relativt programräknaren) som utgör hoppets destination.

För villkorliga hoppinstruktioner, som antingen ska genomföras eller inte genomföras beroende på det angivna villkoret, blir det dock lite mer problematiskt. Om villkoret håller utförs hoppet vilket gör att nästa instruktion hämtas från den av hoppinstruktionen modifierade programräknarens minnesadress, och om det inte håller så hämtas den instruktion som ligger direkt efter hoppinstruktionen i minnet (eftersom programräknarens minnesadress inte modifierats av hoppinstruktionen). I detta fall finns det enligt Abd-El-Barr och El-Rewini (2005) andra tekniker som kan användas gentemot de tekniker som tillämpas för ovillkorliga hopp. Ett exempel på en av dessa tekniker, kallad hopprediktering (eng. *branch prediction*), innebär att hårdvaran på olika sätt försöker att ”förutspå” om ett hopp ska tas eller ej när en hoppinstruktion hämtats. Instruktioner hämtas sedan från den minnesadress som programräknaren innehåller, beroende på om hårdvaran gissade att hoppet ska tas eller ej, alltmedans hoppinstruktionen fortsätter att gå igenom pipelinens steg. Om förutsägelsen är felaktig förklarar Abd-El-Barr och El-Rewini att de felaktigt hämtade instruktionerna måste göras ogjorda (exekveras i motsatt ordning) och pipelinen fyllas med instruktioner hämtade från rätt minnesadress, vilket leder till fler bortslösade klockcykler än utan hopprediktion.

2.4.1 Pipelining och hoppinstruktioner inom ARM-arkitekturen

Inom de olika pipelines som processorfamiljerna inom ARM-arkitekturen använder har man enligt Sloss, et al. (2004) bestämt att exekvering av hoppinstruktioner, samt hopp genom modifiering av programräknaren med vanliga instruktioner, orsakar en tömning (eng. *pipeline*

flush) av de steg i pipelinen som föregår exekveringssteget. På så vis kan pipelinen återigen fyllas med instruktioner hämtade från rätt plats i programmet. Vissa processorfamiljer, bl.a. ARM10-familjen, har dock stöd för hopprediktion.

2.5 Predikatbaserad exekvering

Utan hoppinstruktioner skulle ett program bestå av instruktioner som exekveras sekventiellt, d.v.s. i turordning från den första till den sista instruktionen – ett sekventiellt flöde av instruktioner med andra ord (programflöde). Hoppinstruktioner används för att modifiera programräknaren och därmed vilka instruktioner som ska exekveras, vilket innebär att exekveringsordningen mellan instruktionerna i programflödet kan styras av hoppinstruktionerna.

Predikatbaserad exekvering (eng. *predicated execution*) innebär enligt August, Hwu och Mahlke (1997) att exekveringen av en instruktion bestäms av ett s.k. predikat eller villkor, och principen påminner om villkorliga hoppinstruktioner; instruktioner vars villkor håller exekveras och om de inte håller ignoreras de av processorn. August, et al. (1997) förklarar att instruktioner som beror på att ett villkorligt hopp utförs beror egentligen på att villkoret för hoppet håller, varför det villkoret kan anges för var och en av dessa instruktioner och därmed kan den villkorliga hoppinstruktionen helt tas bort ur programmets kod. Predikatbaserad exekvering är enligt August, et al. (1997) ett alternativ till hoppinstruktioner för att styra vilka instruktioner som ska exekveras i programflödet, och detta indikeras bl.a. av att fördröjningar i pipelinen orsakade av hoppinstruktioner kan elimineras (eftersom hoppinstruktionerna tas bort), och för processorer som använder hopprediktion innebär det färre hopp att förutsäga vilket sparar potentiellt förlorade klockcykler (p.g.a. felaktiga förutsägelser).

2.5.1 Predikatbaserad exekvering inom ARM-arkitekturen

Enligt Sloss, et al (2004) är ARM-arkitekturen designad på ett sätt som innebär att i stort sett alla instruktioner kan exekveras villkorligt genom att ange ett villkor tillsammans med instruktionen, vilket anges på samma sätt som för hoppinstruktioner i ARM-arkitekturen. Sloss, et al (2004) förklarar fortsättningsvis att detta åstadkoms genom att utnyttja villkorsflaggorna i cpsr och hårdvaran i processorns pipeline som utvärderar villkoren genom att matcha dem mot villkorsflaggorna för instruktioner som inte är villkorliga hoppinstruktioner. En viktig detalj när det gäller villkoren är att de utgör en del av alla instruktioner som stödjer villkorlig exekvering eftersom de avsatts ett bestämt antal bitar av de 32-bitar som utgör en instruktion inom ARM-arkitekturen. Det är därför de instruktioner som inte har ett villkor angivet implicit tolkas som att de använder villkoret AL (*alltid*, för eng. *always*) varför de alltid körs (villkoret håller alltid).

Att instruktioner kan utnyttja villkor kallas i ARM-arkitekturen, enligt Sloss, et al. (2004), för villkorlig exekvering men är egentligen detsamma som predikatbaserad exekvering (August et al., 1997). Villkorlig exekvering kan, tillsammans med det faktum att de allra flesta instruktionerna kan fås att påverka villkorsflaggorna i cpsr på samma sätt som jämförelseinstruktionerna, användas för att implementera enkla if-satser (en högnivåkonstruktion) utan hoppinstruktioner. Detta menar Sloss, et al. (2004) ska förbättra kodens effektivitet (då hoppinstruktioner undviks) och dessutom öka koddensiteten (d.v.s. reducera kodens storlek i antal instruktioner).

Nedan följer ett exempel på en tillämpning av villkorlig exekvering; maskinkodsavsnitten jämför två positiva heltal och räknar ut den absoluta skillnaden mellan dem. Variant a) ger en implementation utan villkorlig exekvering och b) ger en implementation med villkorlig

exekvering. Variant b) är både mindre (två instruktioner mindre) och snabbare (inga hoppinstruktioner) än variant a).

Exempel 2

a)

```
            CMP r0, r1          ; jämför r0 och r1
            BLT else            ; om r0 < r1 hoppa till else
            SUB r3, r0, r1      ; r3 = r0-r1
            B end               ; hoppa (alltid) till end
      else
            SUB r3, r1, r0      ; r3 = r1 - r0
      end
```

b)

```
      CMP r0, r1                ;jämför r0 och r1
      SUBGE r3, r0, r1          ;om r0 >= r1 utför r3 = r0 – r1
      SUBLT r3, r0, r1          ;om r0 < r1 utför r3 = r1-r0
```

2.6 Predikatbaserad exekvering och optimering

Enligt Aho, et al. (2007) använder kompilatorer olika typer av representationer, som träd och grafer, för att representera kod i olika faser av kompileringsprocessen. När en kompilator översätter källkod (kod i ett högnivåspråk) till maskinkod använder den en representation som generellt kallas mellanliggande kod (eng. *intermediate code*). Maskinkod ska genereras från den mellanliggande koden, och enligt Aho, et al. (2007) finns det ett sätt som underlättar processen; att konstruera en *flödesgraf*. För att konstruera flödesgrafen måste den mellanliggande koden först delas in i s.k. *grundblock* (eng. *basic block*), vilka motsvarar noder i flödesgrafen, vilka sedan kopplas med kanter i den ordning koden i grundblocken relaterar till varandra. Flödesgrafen och grundblocken kan sedan, enligt Aho, et al. (2007), användas för att på olika sätt upptäcka optimeringstillfällen.

Det som gör grundblocken intressanta är de tre regler för indelning av mellanliggande kod i grundblock som beskrivs av Aho, et al. (2007): 1) den första instruktionen i koden är en ledare, 2) alla instruktioner som är en destination för ett villkorligt eller ovillkorligt hopp är en ledare och 3) alla instruktioner som följer ett villkorligt eller ovillkorligt hopp är en ledare. Dessa tre regler delar alltså in den mellanliggande koden i grundblock som sinsemellan antingen är kopplade till varandra med ett hopp eller ligger i följd. Reglerna är på en sådan nivå att det går att se hur dessa kan tillämpas på maskinkod istället för mellanliggande kod, för att på samma sätt dela in maskinkoden i grundblock.

Enligt Aho, et. Al (2007) kopplas grundblock samman med kanter för att bilda en flödesgraf, där kanterna mellan noderna skapas av två anledningar; antingen leder ett villkorligt eller ovillkorligt hopp från slutet på det ena grundblocket till början på det andra, eller så ligger grundblocken direkt efter varandra jämfört med koden innan den delades i grundblock (förutsatt att det första av de två grundblocken inte slutar med ett ovillkorligt hopp till ett annat grundblock). Även här ligger reglerna på en sådan nivå att det går att se hur grundblock bestående av maskinkod kan sättas ihop till flödesgrafer.

Park och Schlansker (1991) beskriver en algoritm som med utgångspunkt i en flödesgraf av ett program associerar predikat med grundblock beroende på hur flödet ser ut mellan grundblocken. En grov förenkling av algoritmen är att den tar villkoret från en vilkorlig hoppinstruktion och associerar det med grundblocket som är dess destination, och associerar komplementet (det motsatta villkoret) till det grundblock som kommer direkt efter hoppinstruktionen (d.v.s. om hoppet inte tas). När det är gjort kan hoppinstruktionen tas bort och de tre grundblocken slås samman; predikaten som grundblocken är associerade med tillämpas på instruktionerna i respektive grundblock, och på så vis kan hoppinstruktioner elimineras för att ersättas med predikatbaserad exekvering.

3 Problemformulering

Detta examensarbete syftar till att undersöka vilken eventuell skillnad predikatbaserad (eller villkorlig) exekvering kan göra på olika program bestående av maskinkod inom ARM-arkitekturen. Begränsningarna den mängd inbäddade system som använder ARM-processorer står inför innebär att det kan vara av intresse att programmen är så effektiva (med aveseende på antalet klockcykler) och tar så lite plats i som möjligt (hög koddensitet).

En studie av Bringmann et al. (1995) introducerar en kompilatorteknik som använder s.k. hyperblock för att bl.a. utnyttja predikatbaserad exekvering vid kompilering av kod till VLIW- och superskalära processorer. En annan studie av Bringmann et al. (1994) undersöker vilken påverkan predikatbaserad exekvering kan ha på hoppinstruktioner och hopprediktering för program till en superskalär processorarkitektur, även här med hjälp av hyperblock. I en studie av August et al. (1997) introduceras ett alternativt kompilatorramverk, som bland annat utnyttjar hyperblocktekniken, i syftet att försöka förbättra befintligt kompilatorstöd för predikatbaserad exekvering.

Dessa studier skiljer sig från detta examensarbete då de riktar in sig på olika aspekter av kompilatorstöd för predikatbaserad exekvering och hur väl detta utnyttjas av kompilatorn när olika fall av källkod kompileras, till skillnad från detta examensarbete som riktar in sig på nyttan av predikatbaserad exekvering som optimeringsalternativ för olika fall av ARM-maskinkod. En annan detalj är att de utgår från en annan typ av processorer (VLIW- och superskalära processorer), vilket gör att resultaten kan skilja sig från de resultat som en undersökning av predikatbaserad exekvering på ARM-processorer (som är målarkitekturen för undersökningen i detta examensarbete) kommer att ge.

Det finns dock ett arbete av Chanet, De Bosschere, De Bus, De Sutter och Van Put (2007) som behandlar optimering av kod till ARM-arkitekturen. Optimeringarna utförs med en optimerare som konstruerats för ändamålet och syftar till att reducera energikonsumtion och kodens storlek samt öka hastigheten på koden. Studien skiljer sig dock på ett par punkter i detta examensarbete. För det första utförs optimeringarna i och med länkningssteget, vilket utförs efter att all källkod/maskinkod kompilerats/assemblerats. För det andra utförs ett flertal olika optimeringar vilket gör det svårt att undersöka hur just predikatbaserad exekvering påverkar resultatet.

Eftersom predikatbaserad exekvering kan användas till att reducera antalet hoppinstruktioner och även implementera konstruktioner motsvarande if- och if-else-satser i ARM-maskinkod, som nämndes i kapitel 2.5.1, så kan en eventuell skillnad mätas i maskinkodens storlek/densitet samt i antalet klockcykler före resp. efter tillämpning av predikatbaserad exekvering. Frågan är därför om tillämpning av predikatbaserad exekvering som en optimeringsteknik ger en fördelaktig skillnad med avseende på ett programs kodstorlek och/eller dess tidseffektivitet (hastighet). Till skillnad från de ovan nämnda (och liknande) studierna ligger fokus i detta examensarbete på att genom en optimeringsalgoritm, som tar bort hopp m.h.a. predikatbaserad exekvering, optimera program skrivna i ARM-maskinkod.

3.1 Optimering med optimeringsalgoritmen

Optimeringsalgoritmen ska fungera så att det tar ett program i ARM-maskinkod som indata, och sedan i någon form, direkt eller indirekt, utnyttja grundblock och flödesgrafer för att kunna upptäcka möjliga optimeringar. Algoritmen som beskrivs av Park och Schlansker (1991) är en källa till inspiration för optimeringsalgoritmens implementation.

3.1.1 Problemavgränsning

För att avgränsa problemet kommer antalet instruktioner som kan används i de program som ska optimeras att begränsas, eftersom antalet instruktioner i ARM-arkitekturen är ganska stort och optimeringsalgoritmen måste kunna känna igen och bearbeta enskilda instruktioner. Urvalet av instruktioner kommer att bestå av samtliga hoppinstruktioner samt ett par aritmetisk-logiska instruktioner och dataförflyttningsinstruktioner (d.v.s till/från register och minne), men urvalet kan komma att utökas i mån av tid. Instruktionerna ska räcka till att kunna implementera en uppsättning olika program som optimeringsalgoritmen sedan ska kunna optimera.

En annan viktig avgränsning i det här fallet är att endast hoppinstruktioner är tillåtna att utföra hopp; i ARM-arkitekturen är programräknaren synlig, och vanliga instruktioner kan komma åt och ändra dess värde för att på så viss åstadkomma ett hopp. Att avgöra vad som är ett hopp eller inte blir därmed svårare, varför det är bättre om programmen som ska optimeras endast använder hoppinstruktioner för att ändra exekveringsordningen bland instruktioner.

Sist men inte minst begränsas optimeringsalgoritmen, åtminstonde som grund, till att kunna identifiera och optimera vad som i ARM-maskinkod motsvarar if- och if-else-satser.

3.2 Metodbeskrivning

För att utvärdera optimeringsalgoritmen och den optimerade maskinkoden används en experimentbaserad metodologi i likhet med Chanet et al. (2007). I studien testades optimeringsalgoritmen på ett antal program/programbibliotek, och resultatet jämfördes mot samma program/programbibliotek utan att optimeringsalgoritmen använts med avseende på de egenskaper optimeringsalgoritmens optimeringar riktats in på (kodstorlek, energieffektivitet och kodeffektivitet).

I detta examensarbete kommer optimeringsalgoritmen att köras på en uppsättning program skrivna i ARM-maskinkod. Både den ickeoptimerade och optimerade maskinkoden assembleras med hjälp av *armasm* (ingår i *RealView MDK-ARM* från Keil) med alla optimeringsalternativ avstängda för att producera ett exekverbart program. Maskinkoden för de ickeoptimerade respektive optimerade varianterna av programmet jämförs med varandra i storlek (antal instruktioner), och exekveringstiden mäts genom att köra den ickeoptimerade respektive optimerade varianten av det exekverbara programmet. Skillnaden mellan de ickeoptimerade och de optimerade programmen sätts i relation till hur lång tid det tar för optimeringsalgoritmen att utföra optimeringen av respektive program. Detta gör det möjligt att se hur effektiv optimeringsalgoritmen är i förhållande till optimeringen som utförs.

För att mäta och jämföra exekveringstiden körs programmen på en simulator som ingår i utvecklingsmiljön *µVision* (ingår som en del av RealView MDK-ARM från Keil), vilken kan simulera ett antal olika kretsar som använder ARM-processorer och även mäta exekveringstiden med stor precision. Simulatorn tillåter att man sätter brytpunkter i programmet, vilket gör att exekveringen stannar tills användaren väljer att återuppta exekveringen. Detta kan utnyttjas under mätningen för att nollställa tidräknaren vid olika tillfällen under programmets exekvering, vilket innebär att det går att mäta exekveringstiden för ett helt program eller delar av ett program (subrutiner, enskilda instruktioner). Hur de enskilda programmen mäts beskrivs i kapitel 4.2.

För att mäta hur lång tid optimeringsalgoritmen behöver för att optimera maskinkoden mäts tiden det tar för en miljon anrop till optimeringsalgoritmen att optimera maskinkoden, vilket

sedan delas med en miljon för att få den genomsnittliga tiden. Detta ska göras i programmet som optimeringsalgoritmen utgör en del av.

Eftersom en utvärderingsversion av RealView MDK-ARM används är antalet processorer i µVisions simulator begränsade till ett par processorer från ARM7- respektive ARM9-familjerna, och då pipelinen är olika lång i de två processorfamiljerna kommer testningen av programmen att ske på ett par kretsar som använder en processor från ARM7-familjen (tredelad pipeline) samt en processor från ARM9-familjen (femdelad pipeline) (Sloss, et al., 2004).

4 Genomförande

Detta kapitel behandlar genomförandet av metoden som beskrivs i kapitel 3. Först beskrivs optimeringsalgoritmen och dess implementation, vilken ligger till grund för att kunna utföra mätningarna. Kapitlet fortsätter med en beskrivning av de mätningar som utförts och avslutas med en analys av de resultat mätningarna gett upphov till.

4.1 Det producerade systemet

4.1.1 Översiktligt om optimeringsalgoritmen

Mätningarna som beskrivs i kapitel 4.2 bygger på att, enligt metodbeskrivningen i kapitel 3.2, jämföra ickeoptimerade och optimerade program skrivna i ARM-maskinkod med avseende på kodstorlek och tidseffektivitet. Optimeringsalgoritmen är en förutsättning för att dessa mätningar ska kunna ske, då den producerar program som optimerats med villkorlig exekvering givet ickeoptimerade program som indata. Detta upplägg innebär att optimeringsalgoritmen implementerats i form av en konsolapplikation skriven i C++. Applikationen tar emot en källkodsfil skriven i ARM-maskinkod, läser in innehållet till en minnesbuffer, processar det och skriver resultatet till en ny källkodsfil (namngiven efter originalet fast med prefixet ”optimized_” i filnamnet). Resultatet är att man har en ickeoptimerad och en optimerad variant av programmet i varsin källkodsfil, vilka sedan individuellt kan assembleras och jämföras.

4.1.2 Avgränsning av optimeringsalgoritmen

Två inriktningar för implementationen av optimeringsalgoritmen övervägdes. Den första utgår från en hög abstraktionsnivå, där grundblock och flödesdiagram implementeras direkt i form av klasser vilka den inlästa ARM-maskinkoden stegvis transformeras till för att sedan analyseras, optimeras och sedan transformeras tillbaka till ARM-maskinkod. Den andra utgår från en lägre abstraktionsnivå, där grundblock och flödesdiagram används indirekt, genom att gå igenom den inlästa ARM-maskinkoden rad för rad och med hjälp av definitionen av grundblock indirekt analysera programflödet och hitta möjliga optimeringar.

Båda inriktningarna har sina för och nackdelar, men den senare valdes då den av sin sekventiella natur innebär att den inlästa ARM-maskinkoden kan bearbetas i ett svep utan att behöva transformeras på det sätt som den första inriktningen skulle innebära (förutom den bearbetning som krävs för att identifiera instruktioner och operander i en kodrad). Dock innebär den valda inriktningen att det blir komplicerat att utföra optimeringar av loopar, där det kan finnas hoppdestinationer som ligger tidigare i koden. Av den anledningen avgränsades implementationen av optimeringsalgoritmen till att optimera ARM-maskinkod motsvarande if- och if-else-satser, vilka implementeras med hjälp av hopp till destinationer längre ner i koden.

4.1.3 Hur optimeringsalgoritmen analyserar koden

Exempel 3 visar en implementation av en if- och en if-else-sats i ARM-maskinkod, med en motsvarande representation i pseudokod till höger. Exempel 3 a) visar hur hoppinstruktionen används till att hoppa över den kod som if-satsen består av om innehållet i register r0 är mindre än eller lika med (**L**ess than or **E**qual) innehållet i register r1. Notera att detta innebär att koden i if-blocket körs om innehållet i register r0 är större än (**G**reater **T**han) innehållet i register r1. Samma princip gäller för exempel 3 b), med tillägget att koden i if-satsen avslutas med ett ovillkorligt hopp som hoppar över den koden else-satsen består av. Exempel 3 visar

hur hoppinstruktioner och hoppdestinationer (*skip_if*, *else* och *done*) tillsammans implementerar if- och if-else-satser i ARM-maskinkod.

Exempel 3

a)

```
        ...                     ...
        CMP r0, r1              if( r0 GT r1 )
        BLE skip_if             {
        ADD r0, r0, #1                ADD r0, r0, #1
skip_if                         }
        ADD r2, r2, #1          ADD r2, r2, #2
        ...                     ...
```

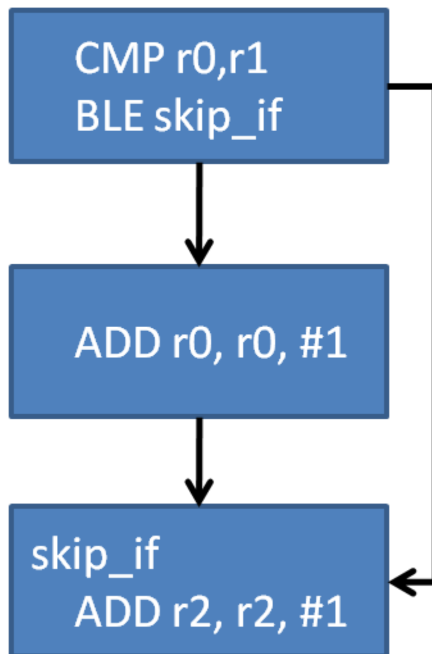
b)

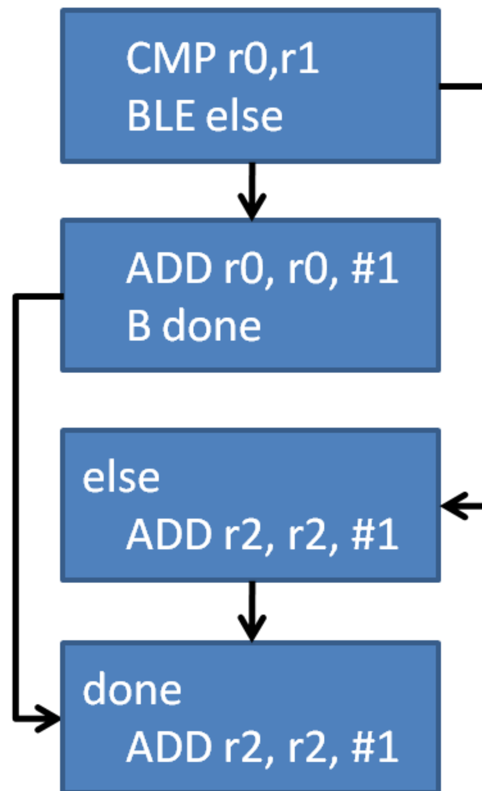
```
        ...                     ...
        CMP   r0, r1            if( r0 GT r1 )
        BLE   else              {
        ADD   r0, r0, #1              ADD r0, r0, #1
        B     done              }
else                            else
                                {
        ADD   r2, r2, #1              ADD r2, r2, #1
done                            }
        ADD r2, r2 #1           ADD r2, r2, #1
        ...                     ...
```

Figur 1 nedan visar ett flödesdiagram bestående av de grundblock exempel 3 a) och 3 b) kan delas upp i enligt definitionen av grundblock. Notera att antalet instruktioner i varje grundblock kan variera, och att det i båda fallen rör sig om att exekveringsflödet delas upp i två möjliga vägar för att sedan mötas i ett grundblock längre ner. Detta innebär att om ett program läser in den första kodraden i t.ex. exempel 3 a) eller 3 b) och sedan fortsätter att läsa in en rad i taget av koden, utan att ta hänsyn vilket grundblock den befinner sig i, så kommer den när den nått den sista raden att ha läst in hela programmet och har då gått igenom samtliga grundblock (och därmed flödesdiagrammet).

Optimeringsalgoritmen läser koden rad för rad på samma sätt som det hypotetiska programmet, med den avgörande skillnaden att den undersöker varje kodrad för att ta reda på om den nått ett nytt grundblock. Detta kan den göra p.g.a. grundblockets definition, som gör gällande att det första grundblocket börjar vid den första kodraden i ett program. En etikett (eng. *label*), som t.ex. *else* eller *done* i exempel 3 b), innebär att ett nytt grundblock tar vid enligt definitionen då de utgör destinationer för hoppinstruktioner (m.a.o. en ingång till grundblocket). Påträffas en hoppinstruktion i en kodrad innebär det att det aktuella

grundblocket är slut och ett nytt tar vid direkt efter hoppinstruktionen. På detta vis kan optimeringsalgoritmen alltså veta vilket grundblock den befinner sig i.





Figur 1. Grundblock konstruerat av exempel 3 a) och b), sammansatta till varsitt flödesdiagram

Att optimeringsalgoritmen håller reda på vilket grundblock den befinner sig i är dock inte hela lösningen. Studerar man lite närmre på figur 1 så går det i a) se att hoppdestinationen *skip_if* som används av det första grundblockets villkorliga hoppinstruktion dyker upp på den första kodraden i det sista grundblocket. Från den villkorliga hoppinstruktionen kan optimeringsalgoritmen få ut tre fakta: vilket villkor som ska gälla för att hoppet ska tas, vilket villkor som ska gälla för att hoppet inte ska tas (komplementet till hoppinstruktionens villkor) samt vad hoppdestinationen heter. Då optimeringsalgoritmen arbetar uppifrån och ner så kommer den efter den villkorliga hoppinstruktionen att nå det mellersta grundblocket för att sedan på nästa kodrad nå etiktetten *skip_if* i det sista grundblocket. Eftersom det första och det mellersta grundblocket leder till samma grundblock utan andra grundblock emellan, så kan optimeringsalgoritmen, utan risk för att förändra semantiken, ta bort hoppinstruktionen i det första grundblocket. Instruktionen i det mellersta grundblocket görs villkorlig genom att lägga till komplementet till hoppinstruktionens villkor i instruktionen, eftersom det endast var då hoppinstruktionens villkor inte höll som instruktionen i det mellersta grundblocket skulle ha exekverats. På detta vis kan optimeringsalgoritmen optimera motsvarande en if-sats i ARM-maskinkod.

Ett liknande tillvägagångssätt kan användas på if-else-satser, t.ex. den i figur 1 b), men i det fallet krävs det att optimeringsalgoritmen håller reda på två etiketter och att båda dessa påträffas. I figur 1 b) skulle optimeringsalgoritmen på samma sätt som i figur 1 a) först stöta

på den villkorliga hoppinstruktionen i det första grundblocket och få ut samma information (villkoret, komplementet till villkoret och hoppdestinationen), för att sedan fortsätta till det andra grundblocket. Detta grundblock avslutas dock till skillnad från figur 1 a) med ett ovillkorligt hopp till hoppdestinationen *done*. På kodraden efter denna påträffas den tidigare kända etiketten *else* och en vanlig instruktion efter det, varpå etiketten *done* påträffas i det fjärde grundblocket. Eftersom cirkeln nu är sluten kan optimeringsalgoritmen därmed ta bort både den villkorliga och den ovillkorliga hoppinstruktionen. Instruktionen i det andra grundblocket görs precis som det andra grundblocket i figur 1 b) villkorligt med komplementet till den villkorliga hoppinstruktionens villkor. Däremot görs instruktionen i det tredje grundblocket villkorligt med den villkorliga hoppinstruktionens villkor, eftersom det var hoppinstruktionens destination vilket innebär att instruktionen i det tredje grundblocket endast skulle ha exekverats när hoppet togs (d.v.s. när hoppinstruktionens villkor höll). Notera dock att optimeringar av motsvarande if-else-satser i ARM-maskinkod, som den i figur 1 b), kan vara osäkra om någon tidigare eller senare hoppinstruktion (hoppinstruktioner före det första grundblocket eller från och med det sista grundblocket) har *else* som hoppdestination. Detta tas dock hänsyn till av optimeringsalgoritmen genom att samtliga hoppdestinationer associeras med en räknare som håller reda på hur många gånger en viss etikett använts som hoppdestination. Om antalet gånger är större än ett hoppar optimeringsalgoritmen över optimeringen av den aktuella if-else-satsen.

Optimeringsalgoritmen kan optimera ARM-maskinkod med en eller flera uppsättningar kodsekvenser vars flödesdiagram motsvarar flödesdiagrammen i figur 1; antalet instruktioner i respektive grundblock kan dock vara godtyckligt och av vilken typ som helst med ett förbehåll. Om en CMP eller någon annan instruktion som uppdaterar statusflaggorna i cpsr-registret (inklusive instruktioner med suffixet *S*) påträffas i något av de mellanliggande grundblocken (det andra grundblocket i figur 1 a) samt det andra och tredje i figur 1 b) så utförs ingen optimering av dessa grundblock. Anledningen är att det skulle leda till att de villkorliga instruktioner som kommer efter en sådan instruktion kan riskera att exekveras när de egentligen inte ska det och tvärtom.

4.1.1 Applikationen och optimeringsalgoritmens implementation

Applikationen utgörs av totalt tre filer, varav två definierar singletonklassen InstructionSet som kapslar in samtliga instruktioner, villkor och funktionalitet för att avgöra bl.a. om en sträng är en instruktion eller en villkorlig/ovillkorlig hoppinstruktion. Den används både av optimeringsalgoritmens logik och av applikationens övriga logik. Den tredje filen implementerar applikationens logik och optimeringsalgoritmens logik, där applikationens logik ansvarar för att läsa in en källkodsfil skriven i ARM-maskinkod och förbereda den för bearbetning av optimeringsalgoritmen.

En viktig del av applikationens logik är att den sorterar in maskinkodsfilen i optimeringsbara respektive ignorerade segment. Detta åstadkoms genom att man använder två speciella kommentarer i de program som ska optimeras: ";{" och ";}". Dessa kommentarer betecknar början och slutet på ett optimeringsbart segment, och används för att ringa in de delar av maskinkodsfilen som optimeringsalgoritmen kan behandla (d.v.s. som inte innehåller assemblersdirektiv och liknande). På detta vis kan applikationen ta emot vanliga maskinkodsfiler (förutsatt att kommentarerna används), vilka, beroende på vilken assemblerare som används, kan innehålla ett eller flera assemberdirektiv. När de optimeringsbara segmenten behandlats av optimeringsalgoritmen kan det de sedan flätas samman med de ignorerade segmenten och skrivas till en ny fil, vilken därmed kan assembleras utan förändring.

Nästa rad

Bxx A påträffad

Nästa rad

1: Gå igenom kod rad för rad

Annan etikett, villkorligt hopp eller cpsr-uppdatering eller om B B påträffad och A är hoppdestination fler än en gång

2: Spara rader [A]

B B påträffad och A är hoppdestination endast en gång

Annan etikett, cpsr-uppdatering, villkorligt hopp

Nästa rad

Etikett A påträffad

3: Spara rader [B]

Optimering av [A] och ev. [B] klar

Etikett B påträffad

4: Optimera sparade rader i [A] och ev. [B]

Figur 2. Tillståndsdiagram över optimeringsalgoritmens logik

Nedan förklaras de olika tillstånden från figur 2 som utgör optimeringsalgoritmens logik:

1. Detta utgör starttillståndet, och har endast till uppgift att hämta ut nästa kodrad tills ett villkorligt hopp påträffas (t.ex. Bxx A, där xx är ett villkor och A en etikett), varpå en övergång till tillstånd 2 sker. Innan övergången sparas en pekare till hoppinstruktionen, dess villkor samt etiketten (Etikett A i tillståndsdiagrammet).

2. Detta tillstånd sparar pekare till alla kodrader som följer det villkorliga hoppet (som orsakade övergången från tillstånd 1) i buffer A. Påträffas ett nytt villkorligt hopp (till etikett A eller annan etikett), en instruktion som påverkar flaggorna i cpsr (t.ex. en CMP-instruktion) eller en annan etikett än etikett A är innehållet i buffer A inte en del av en if- eller if-else-sats, varför en övergång till tillstånd 1 sker. Påträffas etikett A är däremot innehållet i buffer A en if-sats, enligt figur 1 a), varför en övergång till tillstånd 4 sker. Påträffas ett ovillkorligt hopp (t.ex. B B, där det sista B:t är en etikett) rör det sig eventuellt om en if-else-sats, varför en övergång till tillstånd 3 sker. Innan övergången sparas en pekare till hoppinstruktionen samt etiketten (Etikett B i tillståndsdiagrammet).

3. Detta tillstånd fungerar precis som tillstånd 2, men här sparas pekare till de instruktioner som följer det ovillkorliga hoppet (som orsakade övergången från tillstånd 2) i buffer B. Påträffas etikett B rör det sig nu om en if-else-sats, enligt figur 1 b), varför en övergång till tillstånd 4 sker.

4. Detta tillstånd utgör själva optimeringssteget, och konverterar instruktionerna i buffer A till predikatbaserade instruktioner genom att applicera det omvända villkoret (komplementet) från det villkorliga hoppet (som sparades i tillstånd 1), på respektive instruktion. Om buffer B också innehåller instruktioner konverteras även dessa till sin predikatbaserade form, fast med det faktiska villkoret (snarare än det omvända villkoret) från det villkorliga hoppet (som sparades i tillstånd 1). Efter detta tas hoppinstruktionerna bort (via pekarna) och en övergång till tillstånd 1 sker.

4.2 Genomförda mätningar

4.2.1 Testplattformar

Som en del av att utvärderingsversionen av RealView MDK-ARM begränsats i användbarhet, har simulatorn i µVision begränsats till ett antal implementationer av ARM-processorer inom ett par familjer, där det endast är ARM-processorer från ARM7- samt ARM9-familjerna som går att köra testerna på (övriga implementerar endast Thumb ISA, inte ARM ISA). En annan begränsning i RealView MDK-ARM är den medföljande assembleraren endast kompilerar kod vars kodstorlek (kod inklusive statisk data) är mindre än 32 KB. Detta har inte påverkat testerna annat än att eventuell statisk data inte kunnat anta alltför stora dimensioner.

Givet det begränsade utbudet valdes tre kretsar med processorer baserade på ARM7TDMI-S från ARM7-familjen, samt två med processorer baserade på ARM968E-S från ARM9-familjen. Detta gör att eventuella skillnader mellan processorerna inom respektive familj, samt eventuella skillnader mellan ARM7- och ARM9-familjerna kan belysas under analysen av mätningarna i kapitel 4.3. Tabell 1 nedan sammanfattar information från µVision om de valda kretsarna.

Kretsnamn	Tillverkare	Processor	Klockfrekvens	Minne
Generisk krets	ARM	ARM7TDMI-S-baserad	60 MHz	Okänt
STA2051	STElectronics	ARM7TDMI-S-baserad	16 MHz	64 KB
W90N740	Nuvoton	ARM7TDMI-S-baserad	33 MHz	Okänt
SJA510HL/696	NXP	ARM968E-S-baserad	10 MHz	80 KB
LPC2927	NXP	ARM968E-S-baserad	10 MHz	56 KB

Tabell 1. Testplattformar

4.2.2 Storlekstest av grundblock

Detta test utgjordes av ett program delat in i sexton subrutiner. De åtta första subrutinerna består av som vad som motsvarar en if-sats innehållandes ett ökande antal ADD-instruktioner. De åtta sista subrutinerna består av vad som motsvarar en if-else-sats innehållandes ett ökande antal ADD-instruktioner. Programmet anropar varje subrutin i varsin loop 2000 gånger med indata som gör att if-satserna exekveras vartannat anrop, och att if-else satserna exekveras växelvis. Exempel 4 visar en mall för implementationen av de två typerna av subrutiner (Se bilaga A, kapitel A.1 för hela implementationen i icke-optimerat tillstånd).

Exempel 4

If-sats:

```
if_x
        CMP r0, #0
```

```
        BEQ if_x_return
        [sätt in x st ADD-instruktioner]

    if_x_return
        BX lr
```

If-else-sats:

```
    if_else_x
        CMP r0, #0
        BNE if_else _x_else
        [sätt in x st ADD-instruktioner]
        B if_else_x_return
        [sätt in x st ADD-instruktioner]
    if_else _x_return
        BX lr
```

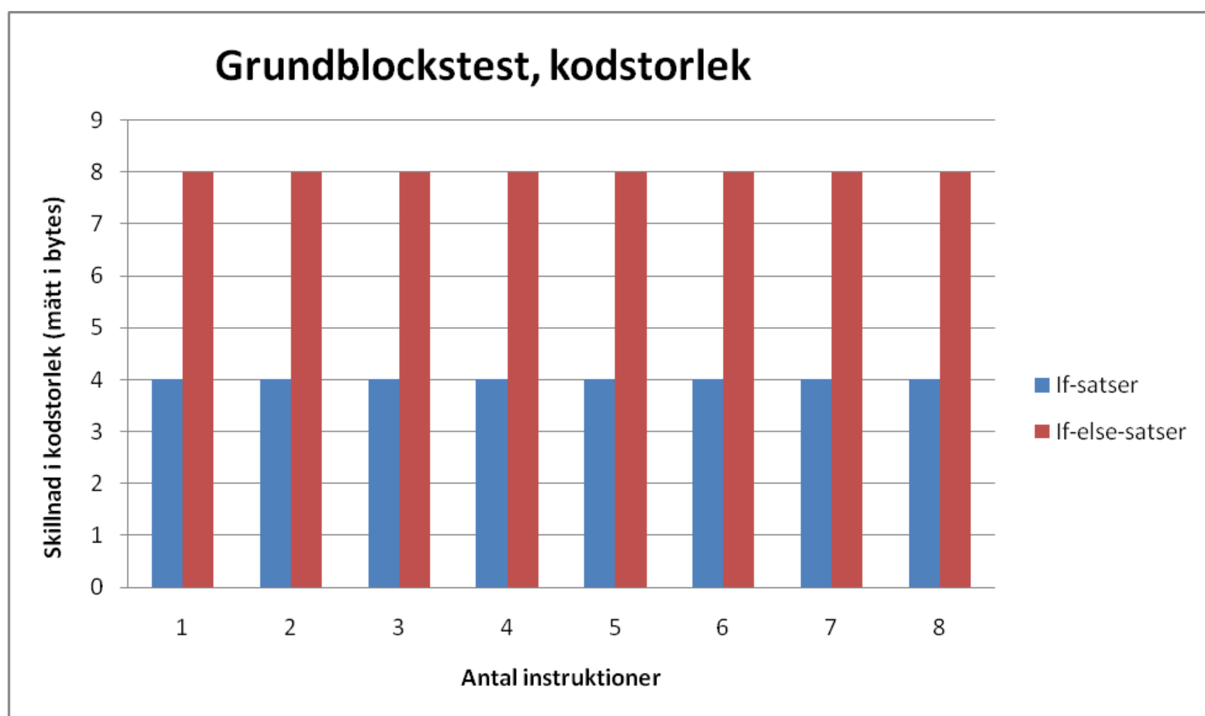
Syftet med testet är att mäta hur stort antal instruktioner ett grundblock, som if- och if-else satserna i subrutinerna motsvarar, kan bestå av och vilket mätresultat det ger före och efter en optimering. ADD-instruktionen valdes godtyckligt, då den precis som majoriteten av instruktionerna i ARM ISA exekveras på en klockcykel enligt Sloss, et al. (2004).

För att mäta kodstorleken jämfördes antalet instruktioner i varje subrutin mellan den ickeoptimerade och den optimerade versionen av koden för varje if- och if-else-sats. Skillnaden i kodstorleken är beräknad i bytes, vilket för att förtydliga innebär att antalet instruktioner som optimerats bort multiplicerades med fyra (en instruktion tar upp fyra bytes minne).

För att mäta exekveringstiden med simulatorn i µVision sattes en brytpunkt mellan varje loop som anropar en subrutin. Varje gång programmet stannade till på en brytpunkt lästes ARM-simulatorns tidsmätare av och nollställdes sedan innan exekveringen återuptogs varpå processen upprepades. Dessa avläsningar användes för att beräkna den procentuella skillnaden i exekveringstid för de optimerade och ickeoptimerade varianterna av varje if- och if-else-sats, vilket för att förtydliga innebär att skillnaden i exekveringstid delades med exekveringstiden före optimering. Positiva och negativa procentsatser motsvarar därmed kortare resp. längre exekveringstid till följd av optimeringen.

Resultat för mätning av kodstorlek

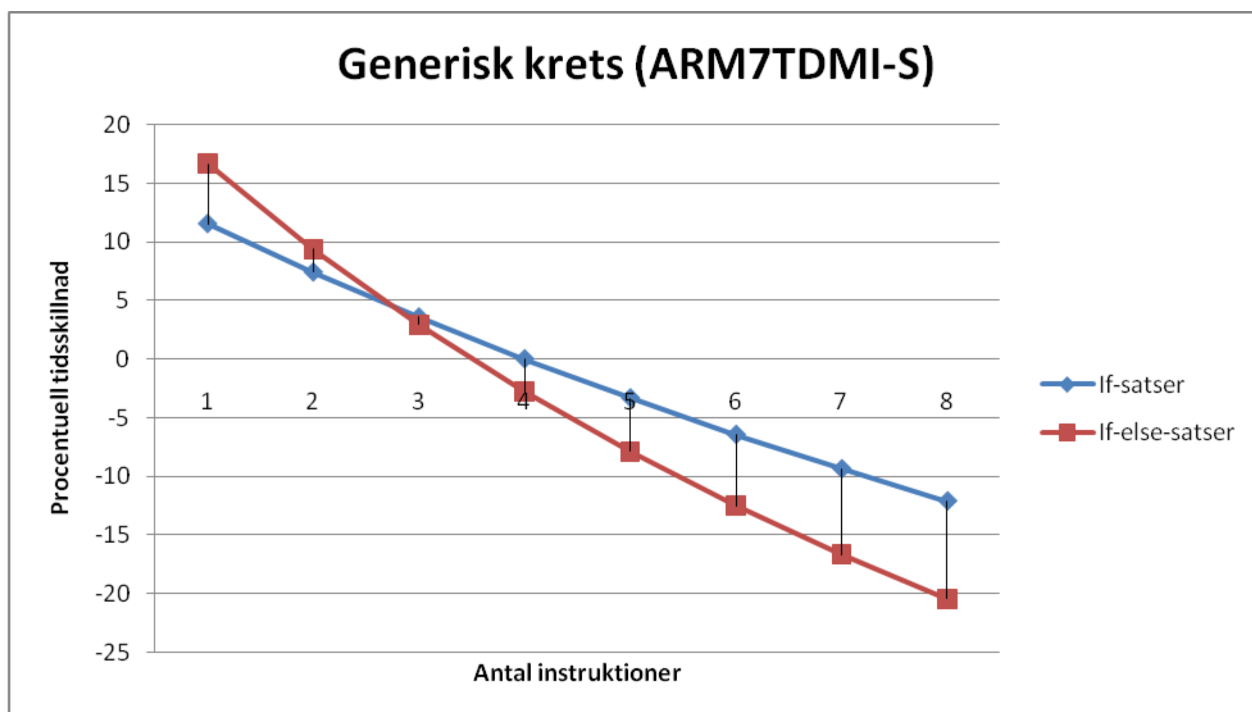
Figur 3 visar den skillnaden i kodstorlek mellan de ickeoptimerade och optimerade if- respektive if-else-satserna.



Figur 3. Mätresultat för kodstorleken

Resultat för mätning av exekveringstid

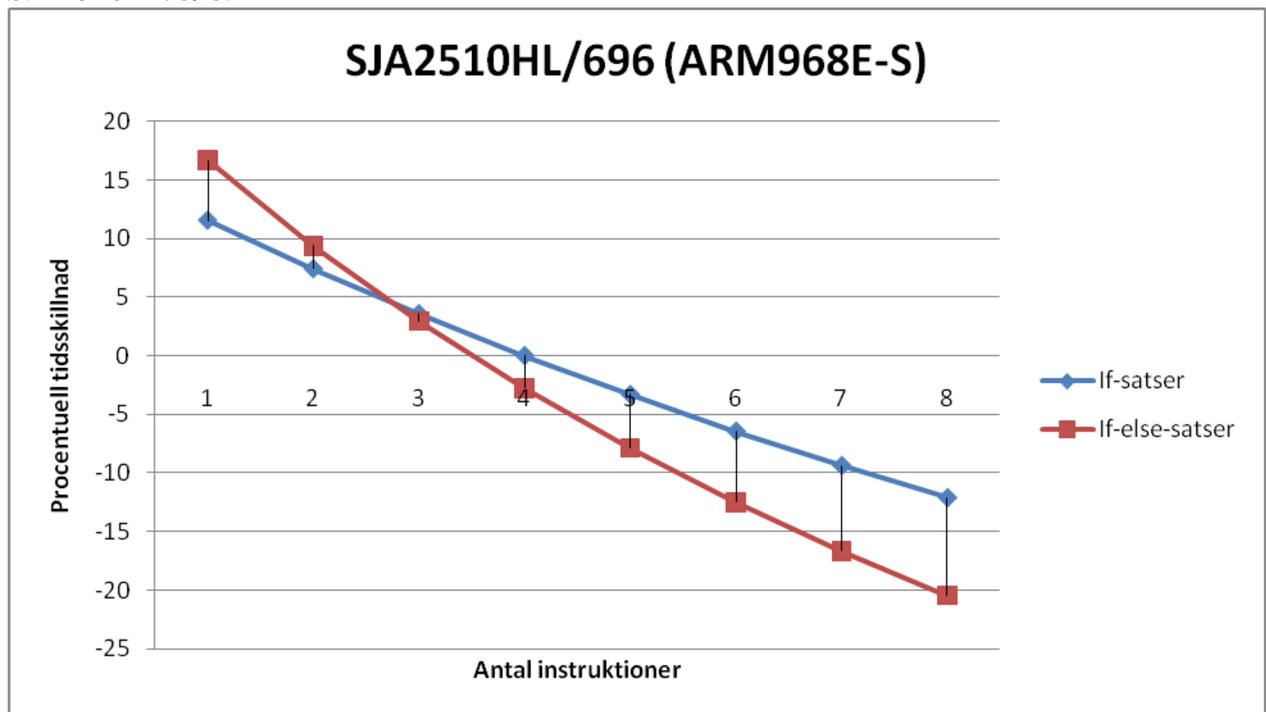
Figur 4 visar den procentuella skillnaden i exekveringstid mellan de ickeoptimerade och optimerade if- respektive if-else-satserna på den generiska testplattformen.



Figur 4. Mätresultat för den generiska testplattformen

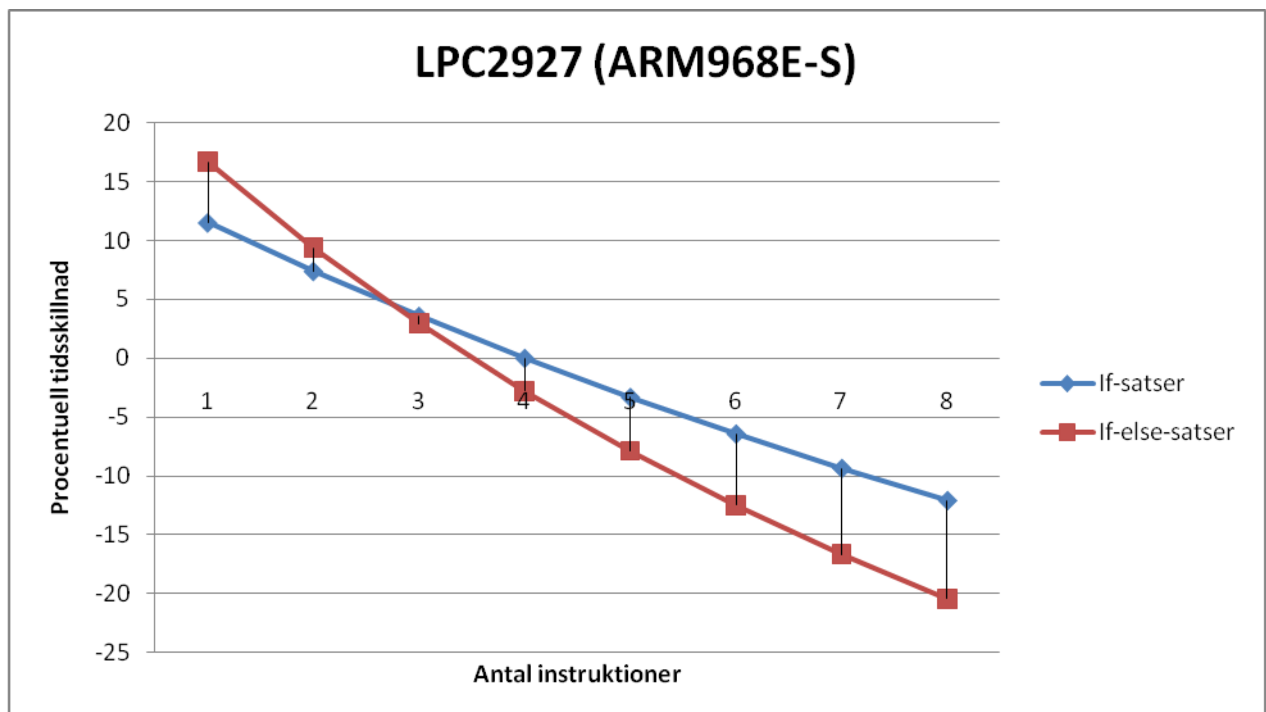
Figur 5 visar den procentuella skillnaden i exekveringstid mellan de ickeoptimerade och optimerade if- respektive if-else-satserna när de exekverades på testplattformen

SJA2510HL/696.



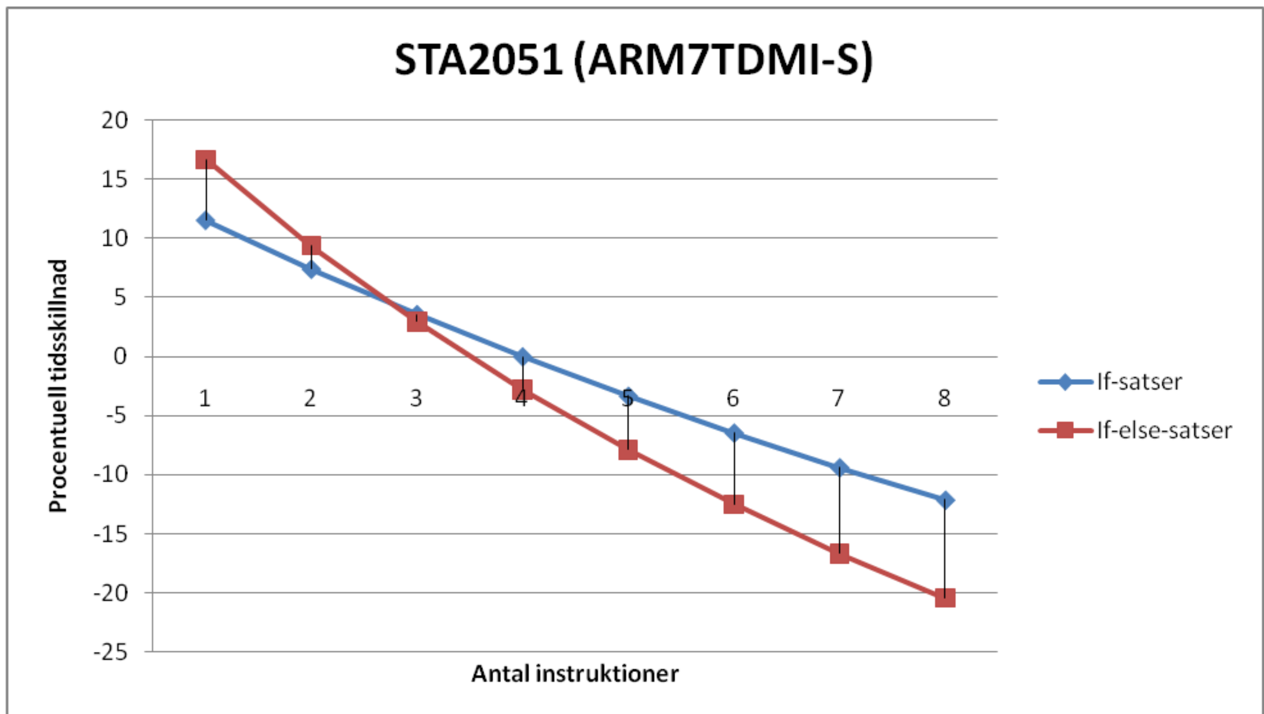
Figur 5. Mätresultat för SJA2510HL/696

Figur 6 visar den procentuella skillnaden i exekveringstid mellan de ickeoptimerade och optimerade if- respektive if-else-satserna när de exekverades på testplattformen LPC2927.



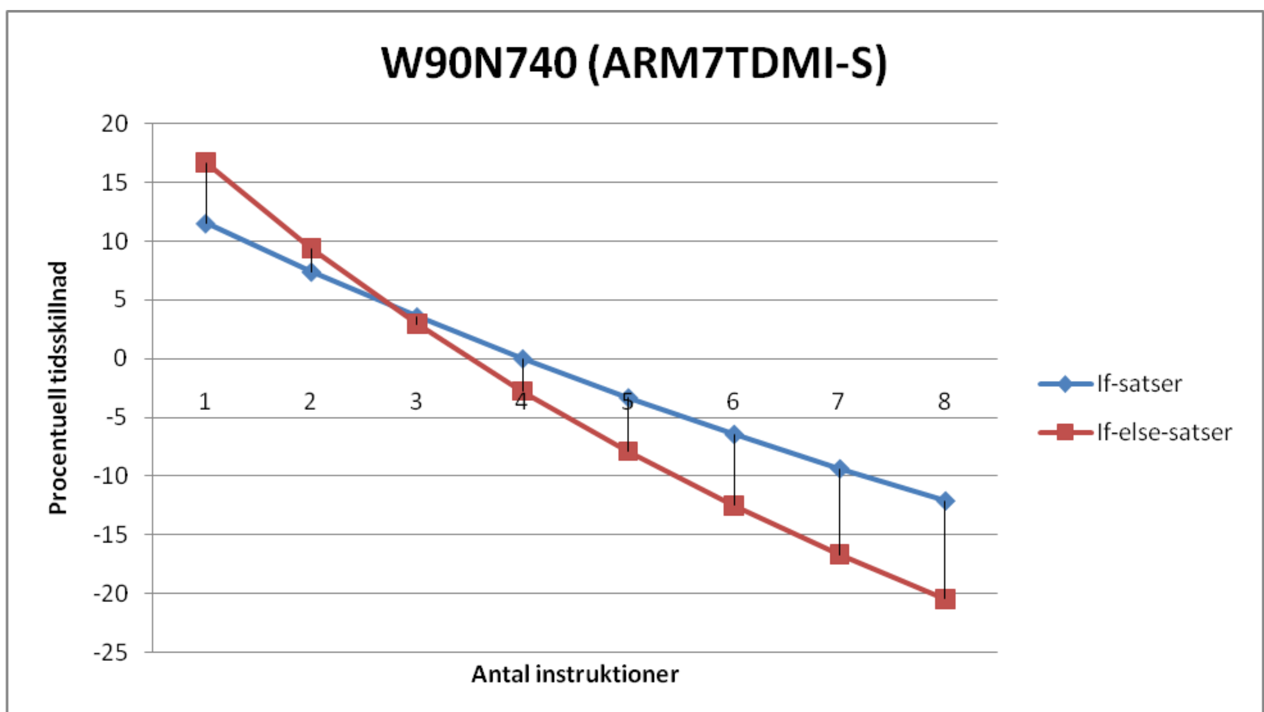
Figur 6. Mätresultat för LPC2927

Figur 7 visar den procentuella skillnaden i exekveringstid mellan de ickeoptimerade och optimerade if- respektive if-else-satserna när de exekverades på testplattformen STA2051.



Figur 7. Mätresultat för STA2051

Figur 8 visar den procentuella skillnaden i exekveringstid mellan de ickeoptimerade och optimerade if- respektive if-else-satserna när de exekverades på testplattformen W90N740.



Figur 8. Mätresultat för W90N740

4.2.3 Bubblesort

Detta test utgjordes av en implementation i ARM-maskinkod av sorteringsalgoritmen bubblesort baserad på en implementation av bubblesort i programmeringsspråket C (Sedgewick, R., 1990), vilken sorterar heltal i stigande storleksordning. Se bilaga A, kapitel A.2 för hela implementationen i icke-optimerat tillstånd. För att kunna testa sorteringsalgoritmen krävdes det en uppsättning heltal som indata till algoritmen, varför 1000 slumpmässiga heltal genererades. Som en utökning av testet gjordes tre olika uppsättningar av de slumpmässiga heltalen; den första bestod av heltalen i den ordning de genererats, den andra bestod av heltalen sorterad i stigande storleksordning (d.v.s. färdigsorterad) och slutligen den tredje uppsättningen som bestod av heltalen sorterad i fallande storleksordning (eftersom talen skulle sorteras i stigande storleksordning var det alltså motsatsen till färdigsorterad). Den ickeoptimerade och den optimerade versionen av sorteringsalgoritmen fick sortera varje uppsättning heltal en gång per testplattform, vilket syftade till att ge en bredare bild av den eventuella skillnaden i exekveringstid mellan den ickeoptimerade och optimerade versionen av sorteringsalgoritmen.

För att mäta kodstorleken jämfördes antalet instruktioner i den ickeoptimerade och den optimerade versionen av sorteringsalgoritmen. Skillnaden i kodstorleken är beräknad i bytes, vilket för att förtydliga innebär att antalet instruktioner som optimerats bort multiplicerades med fyra (en instruktion tar upp fyra bytes minne).

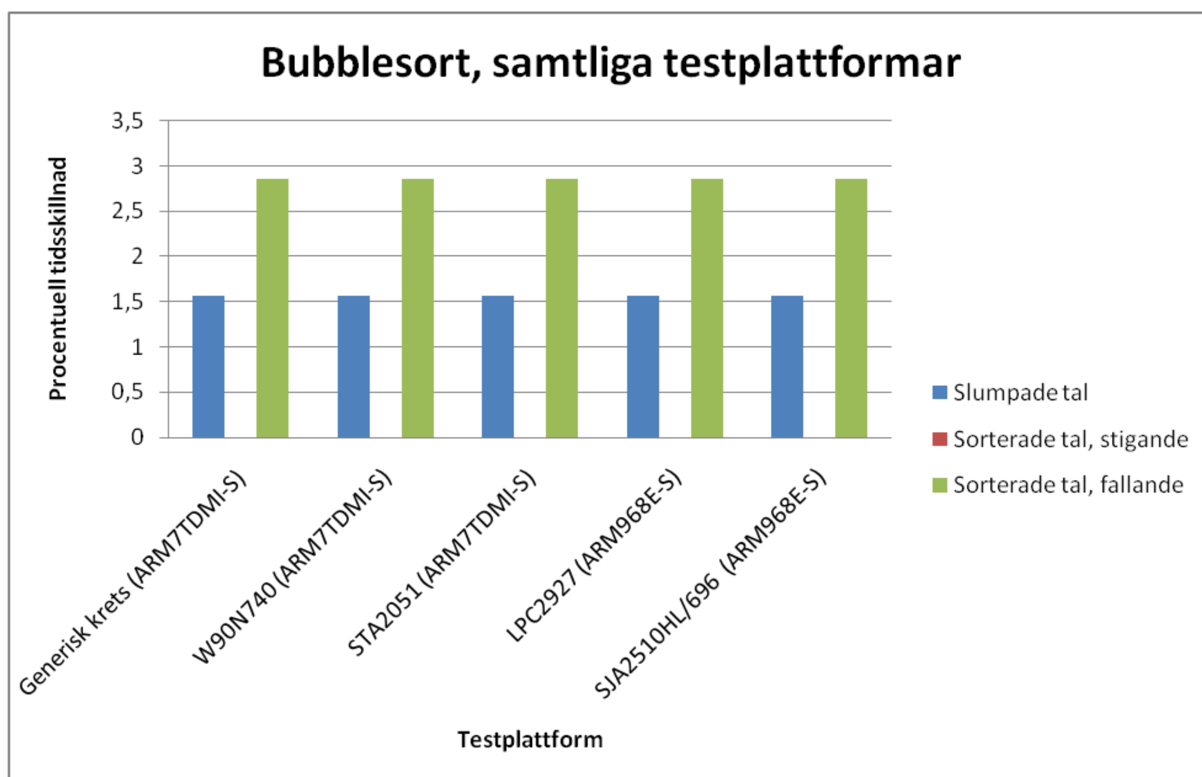
För att mäta exekveringstiden med simulatorn i µVision sattes en brytpunkt i slutet av programmet, vartefter exekveringstiden för sorteringsalgoritmen lästes av och programmet avslutades. Dessa avläsningar gjordes för varje körning av den ickeoptimerade respektive optimerade versionen av sorteringsalgoritmen, där den eventuella skillnaden mellan avläsningarna användes för att beräkna den procentuella skillnaden i exekveringstid för de två versionerna. För att förtydliga innebär det att skillnaden i exekveringstid delades med exekveringstiden före optimering. Positiva och negativa procentsatser motsvarar därmed kortare resp. längre exekveringstid till följd av optimeringen.

Resultat för mätning av kodstorlek

Skillnaden i kodstorlek uppmättes till en borttagen instruktion motsvarande fyra bytes minne.

Resultat för mätning av exekveringstid

Figur 9 visar den procentuella skillnaden i exekveringstid mellan den ickeoptimerade och optimerade versionen av sorteringsalgoritmen för samtliga testplattformar. De tre uppsättningarna heltal (slumpade, sorterade i stigande storleksordning och sorterade i fallande storleksordning) sorterades en gång vardera av den ickeoptimerade respektive den optimerade versionen av sorteringsalgoritmen, vilket gav upphov till tre olika procentuella skillnader i exekveringstid för varje testplattform (den mellersta syns inte då den blev noll).



Figur 9. Mätresultat för bubblesort, samtliga testplattformar

4.2.4 Mergesort

Detta test utgjordes av en implementation i ARM-maskinkod av sorteringsalgoritmen mergesort baserad på en implementation av mergesort i programmeringsspråket C (Sedgewick, R., 1990), vilken sorterar heltal i stigande storleksordning. Se bilaga A, kapitel A.3 för hela implementationen i icke-optimerat tillstånd. För att kunna testa sorteringsalgoritmen krävdes det en uppsättning heltal som indata till algoritmen, varför 1000 slumpmässiga heltal genererades. Som en utökning av testet gjordes tre olika uppsättningar av de slumpmässiga heltalen; den första bestod av heltalen i den ordning de genererats, den andra bestod av heltalen sorterad i stigande storleksordning (d.v.s. färdigsorterad) och slutligen den tredje uppsättningen som bestod av heltalen sorterad i fallande storleksordning (eftersom talen skulle sorteras i stigande storleksordning var det alltså motsatsen till färdigsorterad). Den ickeoptimerade och den optimerade versionen av sorteringsalgoritmen fick sortera varje uppsättning heltal en gång per testplattform, vilket syftade till att ge en bredare bild av den eventuella skillnaden i exekveringstid mellan den ickeoptimerade och optimerade versionen av sorteringsalgoritmen.

För att mäta kodstorleken jämfördes antalet instruktioner i den ickeoptimerade och den optimerade versionen av sorteringsalgoritmen. Skillnaden i kodstorleken är beräknad i bytes, vilket för att förtydliga innebär att antalet instruktioner som optimerats bort multiplicerades med fyra (en instruktion tar upp fyra bytes minne).

För att mäta exekveringstiden med simulatorn i µVision sattes en brytpunkt i slutet av programmet, vartefter exekveringstiden för sorteringsalgoritmen lästes av och programmet avslutades. Dessa avläsningar gjordes för varje körning av den ickeoptimerade respektive optimerade versionen av sorteringsalgoritmen, där den eventuella skillnaden mellan avläsningarna användes för att beräkna den procentuella skillnaden i exekveringstid för de två versionerna. För att förtydliga innebär det att skillnaden i exekveringstid delades med

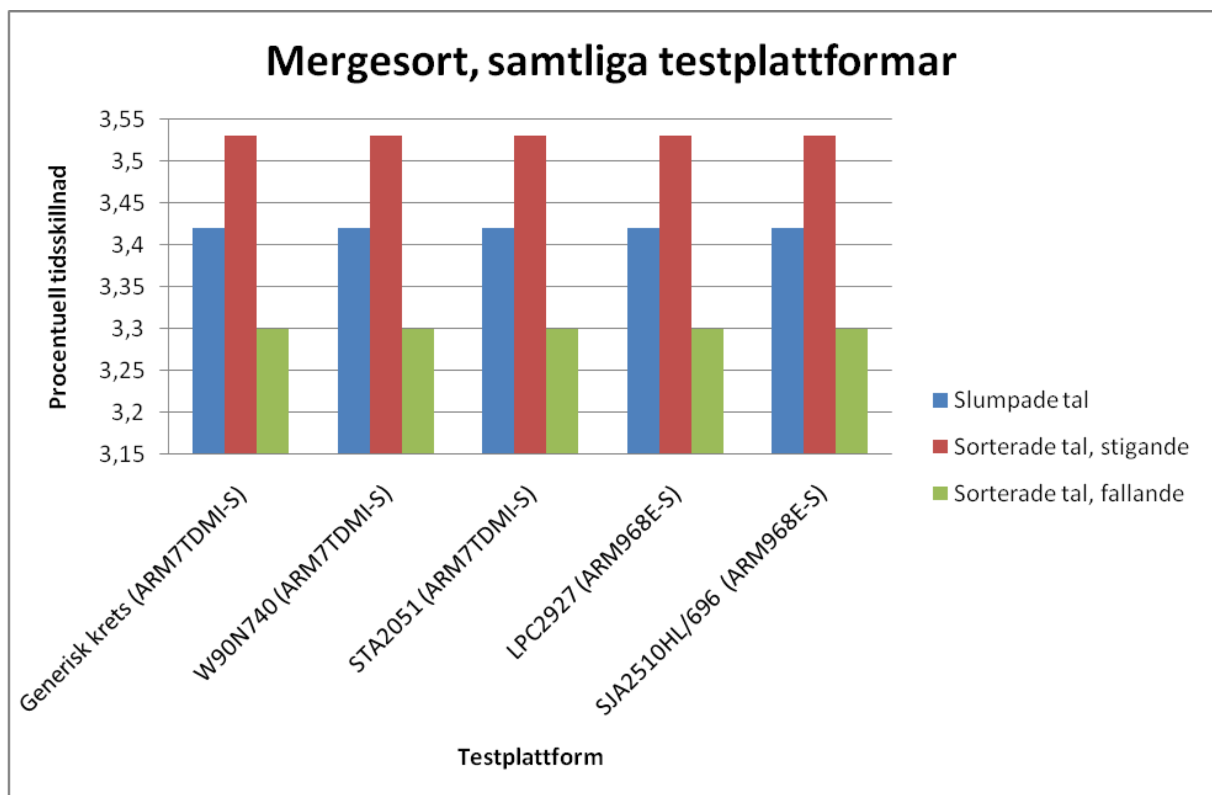
exekveringstiden före optimering. Positiva och negativa procentsatser motsvarar därmed kortare resp. längre exekveringstid till följd av optimeringen.

Resultat för mätning av kodstorlek

Skillnaden i kodstorlek uppmättes till två borttagna instruktioner motsvarande åtta bytes minne.

Resultat för mätning av exekveringstid

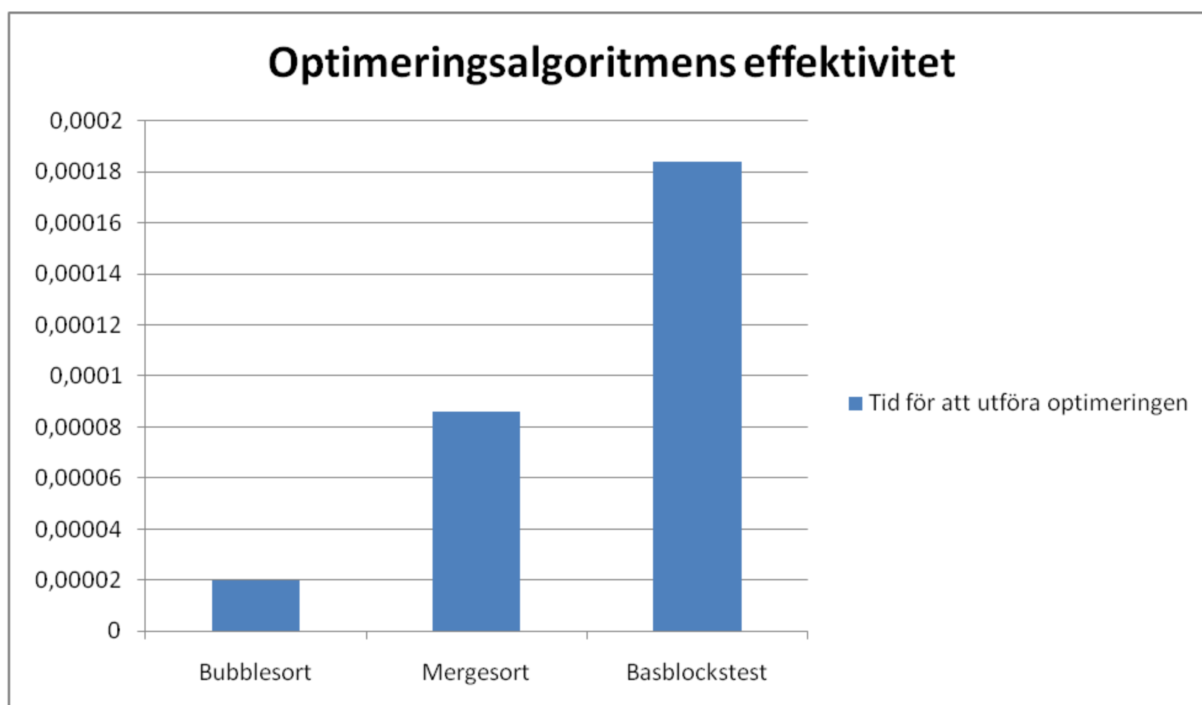
Figur 10 visar den procentuella skillnaden i exekveringstid mellan den ickeoptimerade och optimerade versionen av sorteringsalgoritmen för samtliga testplattformar. De tre uppsättningarna heltal (slumpade, sorterade i stigande storleksordning och sorterade i fallande storleksordning) sorterades en gång vardera av den ickeoptimerade respektive den optimerade versionen av sorteringsalgoritmen, vilket gav upphov till tre olika procentuella skillnader i exekveringstid för varje testplattform.



Figur 10. Mätresultat för mergesort, samtliga testplattformar

4.2.5 Resultat för optimeringsalgoritmen

Figur 11 visar hur lång tid det tog för optimeringsalgoritmen att optimera koden för de tre programmen, i stigande ordning.



Figur 11. Mätresultat för optimeringsalgoritmen

4.3 Analys av mätningar

I detta delkapitel analyseras mätningarna från delkapitel 4.2.

4.3.1 Kodstorlek

Figur 3 visar tydligt skillnaden i kodstorlek för de individuella if- och if-else satserna i storlekstestet för grundblock före och efter optimering. Skillnaden i kodstorlek är konstant och motsvaras av en borttagen hoppinstruktion (motsvarande fyra bytes) i respektive if-sats och två borttagna hoppinstruktioner (motsvarande åtta bytes) i respektive if-else-sats. Sett till hela programmet motsvarar detta en besparing på 24 instruktioner (motsvarande 96 bytes). Det innebär att kodstorleken minskar i förhållande till antalet if- och if-else-satser i ett program som ska optimeras. Detta visar även mätningen av kodstorlek för bubblesort vars implementation innehöll en if-sats och alltså minskade i kodstorlek med en hoppinstruktion (fyra bytes), samt för mergesort vars implementation innehöll en if-else sats och följaktligen minskade i kodstorlek med två borttagna hoppinstruktioner (åtta bytes).

Hur stor optimeringen av kodstorleken blir beror med andra ord på hur många if- och/eller if-else-satser den ickeoptimerade koden innehåller samt hur stor den ickeoptimerade koden var från början. Om den ickeoptimerade koden består av flera tusen instruktioner och endast tiotalet if- och/eller if-else-satser blir skillnaden i kodstorlek mycket mindre vid en optimering, jämfört med om den ickeoptimerade koden bestod av hundratalet instruktioner och tiotalet if- och/eller if-else-satser.

Figur 11 visar hur lång tid det tog för optimeringsalgoritmen att utföra optimeringen av storlekstestet för grundblock, bubblesort och mergesort. De ickeoptimerade versionerna av de tre programmen är alla olika stora med avseende på kodstorlek, och det innebär också att optimeringsalgoritmen hade olika mycket kod att gå igenom vid optimeringen av respektive program. Av detta följer att det tar längre tid för optimeringsalgoritmen att gå igenom och eventuellt optimera koden, men det ser ut som att det kommer att gå väldigt snabbt att optimera kod för större program. Dock tyder testerna på att det främst är program vars kod har

väldigt mycket if- och/eller if-else-satser i förhållande till sin storlek som har något att hämta från optimeringen, med avseende på kodstorlek, och då särskilt program i stort behov av att tillvarata mängden minne som används.

4.3.2 Tidseffektivitet

Figur 4-8 tyder på att den relativa skillnaden i exekveringstid för de individuella if- och if-else-satserna i storlekstestet för grundblock är lika stora oavsett om kretsarna har en processor från ARM7- eller från ARM9-familjen. Eftersom de olika kretsarna har processorer med olika frekvenser så tog det olika lång tid att köra de ickeoptimerade respektive optimerade if- och if-else-satserna, men som figurerna visar var den relativa skillnaden i exekveringstid densamma. Detta tyder också på att längden på pipelinen inte varit en faktor när det gäller tidsvinsten i just det här fallet, men det utesluter inte att det skulle kunna vara det om exempelvis instruktioner för läsning/skrivning till/från minnet hade använts istället för ADD-instruktioner i testet.

Testet visar dock att tidsvinsten minskar ju fler instruktioner if- och if-else-satserna innehåller, och att vinsten är som störst för både if- och if-else-satser när de endast innehåller en instruktion (i fallet if-else-satser en instruktion vardera för if- respektive elsedelen). En intressant egenskap är att det får plats upp till fyra instruktioner i en if-sats innan den optimerade versionen blir långsammare än den ickeoptimerade versionen av if-satsen, vilket tyder på att det går att minska antalet hoppinstruktioner utan att förlora tidseffektivitet för if-satser bestående av som mest fyra instruktioner. För if-else-satser antyds dock att det går att minska antalet hoppinstruktioner utan att förlora tidseffektivitet när if- och else-delen består av som mest tre instruktioner vardera.

Som helhet antyder storlekstestet för grundblock att det främst är relativt små if- och if-else-satser som tjänar på att optimeras med optimeringsalgoritmen, oavsett om det är en krets med en processor från ARM7- respektive ARM9-familjen eller hur hög klockfrekvens processorn har.

Figur 9 och 10 tyder på att, i likhet med figur 4-8, den relativa skillnaden i exekveringstid är lika stor oavsett om kretsarna har en processor från ARM7- eller från ARM9-familjen för bubblesort och mergesort. Till skillnad från storlekstestet för grundblock så innehåller den ickeoptimerade och optimerade versionen av bubblesort en if-sats med två instruktioner som skriver ett värde till minnet utöver en MOV-instruktion som nollställer ett värde i ett register. Detsamma gäller den ickeoptimerade versionen av mergesort, fast den innehåller en if-else-sats där if-delen innehåller en instruktion som skriver till minnet, samt en ADD-instruktion. Else-delen innehåller även den en instruktion som skriver till minnet, samt en SUB-instruktion. Dessa if- respektive if-else-satser optimerades av optimeringsalgoritmen, och i likhet med storlekstestet för grundblock antyder mätningarna i figur 9 och 10 att längden på pipelinen inte spelat in som en faktor när det gäller tidsvinsten för de optimerade versionerna av bubblesort respektive mergesort.

Figur 9 visar även att tidsvinsten är väldigt liten när bubblesort ska sortera taluppsättningen som redan är sorterad i stigande storleksordning, jämfört med när den sorterar taluppsättningen som är sorterad i fallande storleksordning (tvärtemot vad slutresultatet ska bli). Detta beror på att if- satsen i implementationen av bubblesort jämför två tal, och om de är i fel ordning exekveras instruktionerna i if-satsen vilka byter plats på instruktionen. Följden blir att när taluppsättningen som är sorterad i fallande storleksordning ska sorteras så exekveras aldrig instruktionerna i if-satsen, medans de exekveras hela tiden när taluppsättningen som är sorterad i fallande storleksordning sorteras (varje heltal ligger ju på fel position). Därför antyder detta att tidsvinsten blir större vid en optimering om

instruktionerna i if-satserna som optimeras exekveras oftare än de inte exekveras. Att tidsvinsten för taluppsättningen med en slumpmässig ordning av heltalen hamnar någonstans mellan de två andra taluppsättningarna tyder på att instruktionerna i if-satsen exekveras ungefär varannan gång.

Figur 10 visar en annan sida av optimeringen i förhållande till figur 9. I mergesort jämförs två heltal och beroende på vilket av dem som är störst exekveras antingen if-delen eller else-delen av if-else-satsen, vilka skriver heltalen till ett fält i minnet stigande storleksordning. Av den anledningen visar figur 10 på en tidsvinst för var och en av de tre taluppsättningarna. Instruktionerna i if- och else-delen skriver båda de jämförda heltalen till minnet (i olika ordning), så finns det ingen uppenbar förklaring till varför tidsvinsten är störst när taluppsättningen som redan är sorterad i stigande ordning sorterats, om man inte tittar närmare på vad optimeringen gjort med koden. If-else-satsen implementeras som en CMP-instruktion (jämförelsen av de två heltalen), följt av en *villkorlig* hoppinstruktion (hoppar till else-delen om villkoret håller). Instruktionerna som utgör if-delen följer direkt efter den villkorliga hoppinstruktionen och avslutas med ett *ovillkorligt* hopp förbi instruktionerna som utgör else-delen. Optimeringen gör att båda hoppinstruktionerna tas bort, och instruktionerna i if- respektive else-delen blir istället predikatbaserade. När taluppsättningen som är sorterad i stigande ordning sorteras exekveras instruktionerna tillhörande if-delen oftare än else-delen, varför det ovillkorliga hoppet exekveras oftare än det villkorliga hoppet (som ignoreras när villkoret inte håller, och därmed förbrukar en klockcykel). Det omvända gäller för taluppsättningen som är sorterad i fallande ordning, och i likhet med bubblesort hamnar taluppsättningen med en slumpmässig ordning av heltalen någonstans emellan. Figur 10 tyder med andra ord på att tidsvinsten blir större när if-else-satser vars if-del körs oftare än else-delen optimerats.

Tillsammans tyder figurerna 4-10 på att det framförallt är små if- och if-else-satser (d.v.s få instruktioner) vars instruktioner exekveras ofta som ger störst tidsvinst vid en optimering, t.ex. sådana som är ligger i loopar. Det innebär omvänt att små if- och if-else-satser som exekveras sällan ger marginella tidsvinster. Större if- och if-else-sater orsakar istället en tidsförlust, och om de exekveras ofta (jämför med små och ofta exekverade if- och if-else satser) riskerar detta istället att ge en allt större tidsförlust relaterat till storleken.

Satt i relation till optimeringsalgoritmens effektivitet som figur 11 illustrerar sker optimeringen väldigt snabbt även för lite större program, och det krävs troligen väldigt stora program innan optimeringen inte är värd att göra sett till tiden optimeringsalgoritmen behöver för att gå igenom ett programs kod. Det är dock av större vikt att koden har vissa förutsättningar för att optimeringen ska ge utdelning. Som mätningarna antyder är det skillnad på kod och kod, och för att optimeringen ska ge utdelning i förhållande till tiden det tar för optimeringsalgoritmen att gå igenom koden bör koden innehålla små if- och/eller if-else-satser som exekveras ofta. I annat fall riskerar ”optimeringen” att ge ingen eller till och med negativ inverkan på programmet tidseffektivitet.

5 Slutsatser

Detta avslutande kapitel sammanfattar resultaten och slutsater som kan dras kring resultaten, och avslutas med en kort diskussion om detta examensarbete och framtida arbeten detta examensarbete skulle kunna mynna ut i.

5.1 Resultatsammanfattning

Resultatet från optimeringen, med avseende på kodstorlek och tidseffektivitet, visade sig bero väldigt mycket på koden och dess nyttjande av if- och if-else-satser. Mätningarna har i kapitel 4.2 dock pekat ut ett antal faktorer som påverkar hur stor skillnad optimeringen gör med avseende på kodstorlek och tidseffektvitet. De visar på att predikatbaserad exekvering kan användas till att reducera ett programs kodstorlek i ett linjärt förhållande till antalet if- och if-else-satser i koden. Beroende på hur många instruktioner if- och if-else-satserna innehåller och hur ofta instruktionerna exekveras kan predikatbaserad exekvering även ge större tidsvinster. Antalet instruktioner i if- och if-else-satserna påverkar dock tidsvinsten, och om de innehåller för många instruktioner leder predikatbaserad exekvering istället till en tidsförlust.

Optimeringsalgoritmen som konstruerades för att utföra optimeringen av koden till testerna visade sig kunna gå igenom och optimera kod snabbt. Även om kodens storlek påverkar hur lång tid det tar för optimeringsalgoritmen att gå igenom och optimera eventuella if- och if-else-satser i koden, så är det av större betydelse hur koden relaterar till en eller flera av de faktorer mätningarna pekade ut för att ge ett positivt resultat med avseende på kodstorlek och tidseffektivitet.

5.2 Diskussion

Med resultatet för detta examensarbete i åtanke kan man ställa sig frågan huruvida predikatbaserad exekvering, så som den använts för optimering av ARM-maskinkod, ger tillräckligt bra resultat för att användas som en fristående optimering. Resultaten tyder på att optimeringen kan reducera kodstorleken och öka tidseffektiviteten, men frågan är om eventuella vinster är tillräckligt stora för att motivera att kod optimeras enbart med predikatbaserad exekvering.

Kompilatorer kan utföra ett flertal olika optimeringar av både generell och mer specifik natur, där de generella optimeringarna kan tillämpas och ge resultat i större utsträckning än de mer specifika (Aho et al., 2007). Optimeringen som optimeringsalgoritmen i detta examensarbete utför är av en väldigt specifik natur, då resultatet beror på i hur stor grad koden utnyttjar kontrollflödeskonstruktioner implementerade med hoppinstruktioner motsvarande if- och if-else-satser. Av den anledningen kan det tänkas att optimeringen kanske lämpar sig att användas tillsammans med flera andra optimeringar i en mer omfattande optimeringsalgoritm, men det behöver inte nödvändigtvis innebära att optimeringsalgoritmen måste användas som en del av kompileringsprocessen från högnivåkod till exekverbart program. Optimeringsalgoritmen skulle lika gärna kunna användas som ett försteg (eller preprocessor) till en assemblerare för att försöka förbättra maskinkod, innan den assembleras till ett exekverbart program, och därmed hjälpa utvecklare av program till ARM-baserade system att ta tillvara på så mycket resurser som möjligt.

Totalt sett är optimeringsalgoritmen en lyckad konstruktion, och dess preprocessorliknande funktion har förmodligen många fler tillämpningsområden. Om dagens teknik fortsätter att utvecklas i samma takt kan optimeringsalgortimen, givet vissa förbättringar i algoritmens

hantering av programkod, t.ex. kunna användas som en realtidspreprocessor för optimering av olika sorters kod (inte nödvändigtvis optimering av if-/if-else-satser med predikatbaserad exekvering). Den skulle därmed kunna integreras i en tolk (eng. *interpreter*) för programmeringsspråk som t.ex. Java, C# och Python eller skriptspråk som PHP och Lua i syfte att snabba upp exekvering och minska minnesanvändning på en viss plattform/arkitektur (t.ex. ARM-arkitekturen). På liknande vis skulle optimeringsalgoritmen också kunna integreras i specifika implementationer av virtuella maskiner som simulerar en viss plattform/arkitektur, men körs på en annan plattform/arkitektur. Detta skulle på så vis förhoppningsvis kunna leda till en effektivare simulering och en bättre användarupplevelse, beroende på vilka optimeringar som görs. En annan, möjligen lite mer avlägsen, variant är att utnyttja det preprocessorliknande upplägget från optimeringsalgoritmen för att optimera kommunikationen mellan datorer över lokala nätverk (och även Internet) genom att behandla och eventuellt optimera de meddelanden som skickas mellan dem.

Eftersom detta arbete tidigt avgränsades till att gälla optimering av if- och if-else-satser med hjälp av predikatbaserad exekvering finns det inget som tyder på huruvida predikatbaserad exekvering kan användas som en mer omfattande och kanske mer generell optimering. Så länge ett program använder någon form av kontrollflödeskonstruktioner, och inte bara består av en linjär sekvens av instruktioner utan förgreningar, kan det finnas tillfällen där predikatbaserad exekvering kan utnyttjas för att minska kodstorleken eller öka tidseffektiviteten. Som mätningarna i kapitel 4.2 antyder så är det också en optimering som ger samma resultat på fler än en ARM-processorfamilj, och givet bakåtkompatibiliteten mellan processorfamiljerna med avseende på ARM ISA:n, så kan det tänkas att optimeringen kan ge snarlika eller samma resultat på ARM-processorer från framtida processorfamiljer.

Ett frågetecken kring predikatbaserad exekvering på ARM-processorer är hur de nyare processorfamiljernas stöd för hopprediktion (Sloss, et al., 2004) påverkar eventuella vinster när predikatbaserad exekvering används jämfört med att använda hoppinstruktioner. Det kan tänkas att mätningarna som gjorts i detta examensarbete skulle få en annan utgång om de även utförts på testplatformar med ARM-processorer utrustade med hopprediktering. På grund av resursbrist kunde mätningarna inte utföras på hårdvara utrustad med nyare ARM-processorer (eller äldre) för att sättas i relation till resultaten av mätningarna från de testplatformar som använts i detta examensarbete. Inte heller simulatorn i µVision kunde möjliggöra att mätningarna utfördes på nyare ARM-processorer, vilket gör att det inte går att dra några slutsatser kring hopprediktering kontra nyttjandet av predikatbaserad exekvering som optimering. Att mätningarna utfördes på simulerad hårdvara hade dock sina fördelar, bl.a. möjligheten att direkt kunna avläsa hur lång tid programmet exekverats och att kunna sätta brytpunkter på utvalda ställen i programmet för att stoppa exekveringen. Det är inte säkert att mätningar på verklig hårdvara har samma möjligheter, vilket skulle kunna göra att mätningarna får utföras på ett helt annat sätt.

5.3 Framtida arbete

I ett kort perspektiv skulle fortsättning på det här examensarbetet kunna handla om optimering av vad som i ARM-maskinkod skulle motsvara nästlade if- och if-else-satser och även if-elseif-else-satser. Något som detta examensarbete inte gick in på närmare när det gäller if-else-satser är mer komplexa testvillkor där exekveringen av en if- eller if-else-sats beror på ett flertal jämförelseinstruktioner snarare än en enkel jämförelseinstruktion. Sloss, et al. (2004) ger ett exempel på hur flera villkor i ett uttryck av typen villkor1 AND vilkor2 AND … AND villkorN kan implementeras genom att utnyttja predikatbaserad exekvering, och detta skulle

kunna kombineras med if- och if-else-satser i en lite mer omfattande undersökning av optimering av if- och if-else-satser.

En annan tänkbar fortsättning på det här examensarbetet skulle kunna undersöka vilka resultat ytterligare mätningar skulle ge på verklig hårdvara med en eller flera olika ARM-processorer snararare än simulerad hårdvara, vilket skulle kunna bekräfta huruvida mätningarna som gjorts i detta examensarbete överrensstämmer tillräckligt med verkligheten. En jämförelse med mätningar gjorda på en ARM-processor med respektive utan hopprediktering skulle också vara intressant för att se hur kod som nyttjar hoppinstruktioner skulle påverkas jämfört med motsvarande predikatbaserad kod. Det är inte omöjligt att mätningarna skulle behöva utföras på ett annat, mer eller mindre omfattande sätt på verklig hårdvara.

En lite mer långsiktig fortsättning skulle framförallt kunna utvärdera en bredare tillämpning av predikatbaserad exekvering som optimering av ARM-maskinkod, särskilt genom att inte bara fokusera på vad som i ARM-maskinkod motsvarar if- och if-else-satser. Eftersom predikatbaserad exekvering är ett alternativ till att styra ett programs kontrollflöde med hoppinstruktioner skulle en undersökning av optimering av olika loopkonstruktioner vara intressant, särskilt i kombination med optimering av motsvarande if- och if-else satser och andra konstruktioner som påverkar ett programs kontrollflöde. Mätningarna skulle också kunna utökas i antal och omfattning, och t.ex. undersöka hur olika typer av program som dataspel eller olika typer av realtidsapplikationer skulle påverkas vid en optimering, särskilt om optimeringsalgoritmen skulle utökas med flera typer av optimeringar som nyttjar predikatbaserad exekvering.

Längre bort på horisonten finns arbeten som bygger på att utvärdera optimeringsalgoritmen som en del av t.ex. en eller flera kompilatorer/assemblerare. I fallet kompilator/assemblerare skulle optimeringsalgoritmen kunna ersätta befintliga optimeringar (inte enbart optimeringar som använder predikatbaserad exekvering) i kompilatorn/assembleraren för att jämföra befintliga optimeringar med egenhändigt konstruerade optimeringar, eller förändrade varianter av befintliga optimeringar. På samma sätt skulle man även kunna integrera optimeringsalgoritmens preprocessorliknande funktionalitet i tolkar för programmeringsspråk/skriptspråk, virtuella maskiner och olika nätverksbaserade applikationer för att utvärdera hur optimeringsalgoritmen, i nuvarande eller förändrad form, presterar i realtid. Fokus behöver inte nödvändigtvis ligga på optimering, eller åtminstonde inte optimering med hjälp av predikatbaserad exekvering, utan skulle lika gärna kunna ligga på hur optimeringsalgoritmen, betraktad som preprocessor (eller databehandlare), skulle kunna användas för att minimera olika flaskhalsar (vare sig det är i realtid eller ej).

Referenser

Abd-El-Barr, M. & El-Rewini, H. (2005), *Fundamentals of computer organization and architecture*. John Wiley & Sons Incorporated.

Aho, A. V. Lam, M. S. Sethi, R. & Ullman, J. D. (2007), *Compilers: principles, techniques, & tools*. Pearson Addison-Wesley.

August, D. I. Hwu, W. W. & Mahlke, S. A. (1997), A framework for balancing control flow and predication. *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture* (sid. 92-103). IEEE Computer Society.

Bringmann, R. A. Chen, W. Y. Hank, R. E. Lin, D. C. & Mahlke, S. A. (1995), Effective compiler support for predicated execution using the hyperblock. *Instruction-level Parallel Processors* (sid 161-170). IEEE Computer Society Press.

Bringmann, R. A. Gallagher, D. M. Gyllenhaal, J. C. Hank, R. E. Hwu, W. W. & Mahlke, S. A. (1994), Characterizing the impact of predicated execution on branch prediction. *Proceedings of the 27th Annual International Symposium on Microarchitecture* (sid. 217-227). ACM.

Chanet, D. De Bosschere, K. De Bus, B. De Sutter, B. & Van Put, L. (2007), Link-time compaction and optimization of ARM executables. *ACM Transactions on Embedded Computing Systems*. Vol. 6, nr. 1. ACM.

Hyde, R. (2006), *Write great code volume 2: thinking low-level, writing high-level*. No Starch Press.

RealView MDK-ARM (Version 4.1). [Utvecklingsverktyg] Keil Software, Incorporated. Tillgänglig på Internet: http://www.keil.com/arm/mdk.asp

Sedgewick, R. (1990), *Algorithms In C*. Addison-Wesley

Sloss, A. N. Symes, D. & Wright, C. (2004), *ARM system developer's guide: designing and optimizing system software*. Morgan Kaufmann Publishers.

Park, J. C. H. & Schlansker, M. (1991), *On Predicated Execution*. Hewlett Packard Software Systems Laboratory. Tillgänglig på Internet: http://www.hpl.hp.com/techreports/91/HPL-91-58.pdf [Hämtad 2010-02-01]

Bilaga A

Denna bilaga innehåller de icke-optimerade versionerna av de tre olika programmen som behandlas i kapitel 4.2. I samtliga implementationer har hoppdestinationerna (etiketterna, eng. *labels*) markerats i fetstil.

A.1 Storlekstest av grundblock

Med anledning av implementationens storlek och kodens repetitiva mönster har implementationen som den återges i denna bilaga kortats ned. Områden där kod uteslutits markeras med "...".

```
      AREA  code, CODE
      ENTRY

value          RN 0
dummy_variable  RN 1
loop_variable   RN 2
number_of_loops RN 3

main
      LDR number_of_loops, numloops
      MOV loop_variable, #0
if_one_loop
      MOV value, #0
      BL  if_one
      MOV value, #1
      BL if_one
      ADD loop_variable, loop_variable, #1
      CMP loop_variable, number_of_loops
      BLE if_one_loop
      MOV loop_variable, #0
if_two_loop
      MOV value, #0
      BL if_two
      MOV value, #1
      BL if_two
      ADD loop_variable, loop_variable, #1
```

```
    CMP loop_variable, number_of_loops
    BLE if_two_loop
    MOV loop_variable, #0
...
if_eight_loop
    MOV value, #0
    BL if_eight
    MOV value, #1
    BL if_eight
    ADD loop_variable, loop_variable, #1
    CMP loop_variable, number_of_loops
    BLE if_eight_loop
    MOV loop_variable, #0
if_else_one_loop
    MOV value, #0
    BL if_else_one
    MOV value, #1
    BL if_else_one
    ADD loop_variable, loop_variable, #1
    CMP loop_variable, number_of_loops
    BLE if_else_one_loop
    MOV loop_variable, #0
if_else_two_loop
    MOV value, #0
    BL if_else_two
    MOV value, #1
    BL if_else_two
    ADD loop_variable, loop_variable, #1
    CMP loop_variable, number_of_loops
    BLE if_else_two_loop
    MOV loop_variable, #0
...
if_else_eight_loop
    MOV value, #0
```

```
      BL  if_else_eight
      MOV value, #1
      BL  if_else_eight
      ADD loop_variable, loop_variable, #1
      CMP loop_variable, number_of_loops
      BLE if_else_eight_loop
end_main
      B    end_main

if_one
;{
      CMP value, #0
      BEQ if_one_return
      ADD dummy_variable, dummy_variable, #1
if_one_return
;}
      BX lr    ; register lr innehåller återhoppsadressen

if_two
;{
      CMP value, #0
      BEQ if_two_return
      ADD dummy_variable, dummy_variable, #1
      ADD dummy_variable, dummy_variable, #1
if_two_return
;}
      BX lr    ; register lr innehåller återhoppsadressen

...

if_eight
;{
      CMP value, #0
      BEQ if_eight_return
```

```
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
if_eight_return
;}
	BX lr    ; register lr innehåller återhoppsadressen

if_else_one
;{
	CMP value, #0
	BNE if_else_one_else
	ADD dummy_variable, dummy_variable, #1
	B   if_else_one_return
if_else_one_else
	ADD dummy_variable, dummy_variable, #2
if_else_one_return
;}
	BX lr    ; register lr innehåller återhoppsadressen

if_else_two
;{
	CMP value, #0
	BNE if_else_two_else
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	B if_else_two_return
if_else_two_else
	ADD dummy_variable, dummy_variable, #2
	ADD dummy_variable, dummy_variable, #2
```

```
if_else_two_return
;}
	BX lr	; register lr innehåller återhoppsadressen

...

if_else_eight
;{
	CMP value, #0
	BNE if_else_eight_else
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	ADD dummy_variable, dummy_variable, #1
	B   if_else_eight_return
if_else_eight_else
	ADD dummy_variable, dummy_variable, #2
	ADD dummy_variable, dummy_variable, #2
	ADD dummy_variable, dummy_variable, #2
	ADD dummy_variable, dummy_variable, #2
	ADD dummy_variable, dummy_variable, #2
	ADD dummy_variable, dummy_variable, #2
	ADD dummy_variable, dummy_variable, #2
	ADD dummy_variable, dummy_variable, #2
if_else_eight_return
;}
	BX lr

	AREA  data, DATA
numloops
```

```
        DCD 10000
        END
```

A.2 Bubblesort

```
        AREA  code, CODE
        ENTRY

        EXTERN stack_ptr
        EXTERN array_a_ptr
        EXTERN array_a_length
main
        LDR sp, =stack_ptr
        LDR r0, =array_a_ptr
        LDR r1, array_a_length
        BL bubblesort
end_main
        B end_main

base RN 0
length RN 1
i RN 2
j RN 3
swapped RN 4
i_ RN 5
j_ RN 6
temp RN 7

bubblesort
;{
outer
        MOV swapped, #0
        MOV i, #0
        ADD j, i, #4
inner
        LDR i_, [base, i]
```

```
        LDR j_, [base, j]
        CMP i_, j_
        BLE skip_if
        STR i_, [base, j]
        STR j_, [base, i]
        MOV swapped, #1
skip_if
        ADD i, i, #4
        ADD j, i, #4
        CMP j, length
        BLT inner
        CMP swapped, #0
        BNE outer

;}
        BX lr     ; register lr innehåller återhoppsadressen
        END
```

A.3 Mergesort

```
        AREA  code, CODE
        ENTRY

        EXTERN stack_ptr
        EXTERN array_a_ptr
        EXTERN array_a_length
        EXTERN array_b_ptr
main
        LDR sp, =stack_ptr
        LDR r0, =array_a_ptr
        MOV r1, #0
        LDR r2, array_a_length
        LDR r3, =array_b_ptr
        BL mergesort
end_main
        B end_main
```

```
a_base RN 0
b_base RN 3
left RN 1
right  RN 2
middle RN 4
i RN 5
j RN 6
array1 RN 7
array2 RN 8
tmp_index RN 9
k RN 10

mergesort
;{
    STMFD sp!,{r1, r2, r4-r10, lr}
    MOV i, #0
    MOV j, #0
    MOV k, #0
    MOV middle, #0
    CMP right, left
    BLE return
    ADD middle, right, left
    MOV middle, middle, ASR #1
    STMFD sp!,{right}
    MOV right, middle
    BL mergesort
    LDMFD sp!,{right}
    STMFD sp!,{left}
    ADD left, middle, #1
    BL mergesort
    LDMFD sp!, {left}
    ADD i, middle, #1
for1
```

```
	CMP i, left
	BEQ for1_end
	SUB tmp_index, i, #1
	LDR array1, [a_base, tmp_index, LSL #2]
	STR array1, [b_base, tmp_index, LSL #2]
	SUB i, i, #1
	B for1
for1_end
	MOV j, middle
for2
	CMP j, right
	BEQ for2_end
	ADD tmp_index, j, #1
	LDR array1, [a_base, tmp_index, LSL #2]
	ADD tmp_index, right, middle
	SUB tmp_index, tmp_index, j
	STR array1, [b_base, tmp_index, LSL #2]
	ADD j, j, #1
	B for2
for2_end
	MOV k, left
for3
	CMP k, right
	BGT for3_end
	LDR array1,[b_base, i, LSL #2]
	LDR array2,[b_base, j, LSL #2]
	CMP array1, array2
	BGE else
	STR array1,[a_base, k, LSL #2]
	ADD i, i, #1
	B end_if
else
	STR array2,[a_base, k, LSL #2]
	SUB j, j, #1
```

```
end_if
      ADD k, k, #1
      B for3
for3_end
return
;}
      LDMFD sp!,{r1, r2, r4-r10, lr}
      BX lr    ; register lr innehåller återhoppsadressen
      END
```