HÖGSKOLAN
SKÖVDE
Institutionen för kommunikation och information

Utvärdering av en tidsoptimeringsteknik

Optimering av utvalda if-satser i språket C

Andreas Sundin

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

Utvärdering av en tidsoptimeringsteknik

Examensrapport inlämnad av Andreas Sundin till Högskolan i Skövde, för Kandidatexamen (B.Sc.) vid Institutionen för kommunikation och information. Arbetet har handletts av Henrik Svensson.

2010-06-04

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: _____

Utvärdering av en tidsoptimeringsteknik

Andreas Sundin

Sammanfattning

Det här examensarbetet beskriver en specifik optimeringsteknik och undersöker dess förmåga att öka exekveringshastigheten. Optimeringstekniken ersätter villkorliga hopp i form av if-satser med ovillkorligt hopp. Ett program har utvecklats som mäter exekveringstiden för flera små programkoder, som både använder respektive inte använder optimeringstekniken, så att de kan jämföras med varandra. Med hjälp av mätresultaten från programmet har optimeringstekniken utvärderats. Resultaten visar att optimeringstekniken inte kan öka exekveringshastigheten i de flesta av testerna, men i några specifika fall kan optimeringstekniken öka exekveringshastigheten.

Nyckelord: optimering, exekveringstid, programmering, villkorliga hopp

Innehållsförteckning

1 Introduktion....1

2 Bakgrund....2

2.1 Villkorliga hopp....2

2.2 Ovillkorliga hopp....2

2.2.1 Tilldelade goto....2

2.3 Processorns instruktionspipeline....3

2.3.1 Pipelinekonflikter....4

2.4 Tidigare arbeten....4

3 Problemformulering....6

3.1 Optimeringsteknik som ska utvärderas....6

3.2 Problemprecisering....7

3.3 Metodbeskrivning: Utveckling....8

3.4 Metodbeskrivning: Utvärdering....10

4 Genomförande....13

4.1 Det producerade systemet....13

4.1.1 Testprogrammet....13

4.1.2 Datorspelet....14

4.2 Genomförda mätningar....17

4.3 Analys av mätningar....19

5 Slutsatser....21

5.1 Resultatsammanfattning....21

5.2 Diskussion....21

5.3 Framtida arbete....22

Referenser....24

Bilaga 1: Testernas exekveringstider....25

Bilaga 2: Testprogrammets källkod....27

1 Introduktion

Program som datorspel använder många villkorliga hopp i sin programkod, i form av if-satser och loopar. Dessa villkorliga hopp utvärderar en eller flera variabler med logiska operatorer varje gång de körs. Vissa villkorliga hopp kan utvärderas ofta, exempelvis ett villkorligt hopp i en game-loop i ett spel, medan variablerna som den utvärderar i vissa fall sällan blir tilldelade nya värden, till exempel en variabel som enbart ändras i options-menyn i ett spel. I dessa fall skulle det villkorliga hoppet kunna ersättas med ett dynamiskt ovillkorligt hopp. Det vill säga en funktionspekare eller en *tilldelad goto* (se avsnitt 2.2.1).

Genom att endast en gång kontrollera om utvärderingen av variablerna förefaller att ta det villkorliga hoppet (if-satsen) eller inte, kan man byta ut hoppet till ett ovillkorligt hopp, så länge som variablerna inte tilldelas nya värden. Variablerna i frågan behöver sedan övervakas så att varje gång en av dessa variabler tilldelas ett nytt värde, utvärderas hoppet igen om det skulle tas eller inte och det ovillkorliga hoppet uppdateras.

Det här examensarbetets område är optimering av dataprogram, med inriktning på hopp. Examensarbetet undersöker vilka tidsprestandaförbättringar denna ovan beskrivna optimeringsteknik kan skapa. Detta undersöks genom att mäta exekveringstiden av flera implementerade testprogramkoder, som både använder respektive inte använder optimeringstekniken. Tester kommer att utföras för både if-satser och tilldelning av de variabler som behöver övervakas efter optimeringen. Totalt är det tio programkoder som ska testas. Optimeringstekniken ska även tillämpas på en if-sats i ett datorspel och antalet uppdateringar per sekund ska mätas, både före och efter optimeringen.

Rapportens olika kapitel innehåller i korthet följande: Bakgrundskapitlet beskriver bland annat processorns instruktionspipeline och hur prestanda kan förloras på grund av hopp samt tidigare arbeten inom optimering av dataprogram. Problemkapitlet beskriver den optimeringsteknik som det här examensarbetet behandlar, arbetets preciserade problem samt metodbeskrivning för arbetets två faser, utveckling och utvärdering. Genomförandekapitlet beskriver hur ett testprogram har utvecklats, hur optimeringstekniken tillämpas på ett datorspel och resultaten från utvärderingen. Det sista kapitlet Slutsatser sammanfattar resultaten av min undersökning, diskuterar det i ett större sammanhang och ger förslag på framtida arbeten.

2 Bakgrund

Det här kapitlet beskriver kunskap som kan behövas för att förstå problemställningen. Villkorliga och ovillkorliga hopp beskrivs för att utmärka deras skillnader. Sedan beskrivs processorns instruktionspipeline samt styrkonflikter som illustrerar motivet för det här examensarbetet. Till sist beskrivs några tidigare arbeten inom samma område.

2.1 Villkorliga hopp

Villkorliga hopp är ett vanligt inslag i programmeringskod. De används för att styra programflödet, det vill säga vad programmet utför. Hur det styr programflödet beror på utvärderingen av data, det vill säga vilket värde en viss variabel (eller flera) har. Selektioner och iterationer är vanliga inslag av villkorliga hopp. Exempel på selektioner i programspråket C är if-satser eller switch-case-satser och exempel på iterationer är for-loopar och while-loopar.

```
if(a < 0)
{
      a = a * -1;
}
...
```

Figur 1. Ett villkorligt hopp i form av en if-sats i programmeringsspråket C.

2.2 Ovillkorliga hopp

Ett ovillkorligt hopp är ett hopp som alltid tas och det sker inga beräkningar som utvärderar om hoppet ska tas. Exempel på ovillkorliga hopp är funktionsanrop och den inte (längre) så vanligt använda goto-satsen.

```
goto label2;
label1:
a = a * -1;
label2:
...
```

Figur 2. Ett ovillkorligt hopp i form av en goto-sats.

2.2.1 Tilldelade goto

Tilldelade goto (eng: assigned goto) skiljer sig från de vanliga goto på så sätt att tilldelade goto kan (under exekvering) tilldelas nya adresser som den ska hoppa till. Programmeringsspråket C har inget stöd för tilldelade goto, men tidigare språk som Fortran har stöd för detta. Kompilatorn GCC (GNU Compiler Collection) stödjer dock tilldelade goto i språket C med hjälp av extensionen ”labels as values” (Goto, 2010). En tilldelad goto realiseras då i C-koden med hjälp av void pekare. En pekare är en variabel som lagrar en minnesadress. En void pekare kan tilldelas ett adressvärde (eng: label) med operatorn ”&&” (Wall och Von Hagen, 2004).

```
void* labelPtr = &&label2;
if(a < 0){
      labelPtr = &&label1;
}
else{
      labelPtr = &&label2;
}
...
goto *labelPtr;
label1:
      a = a * -1;
label2:
...
```

Figur 3. Exempel på en tilldelad goto i C som kan kompileras med GCC.

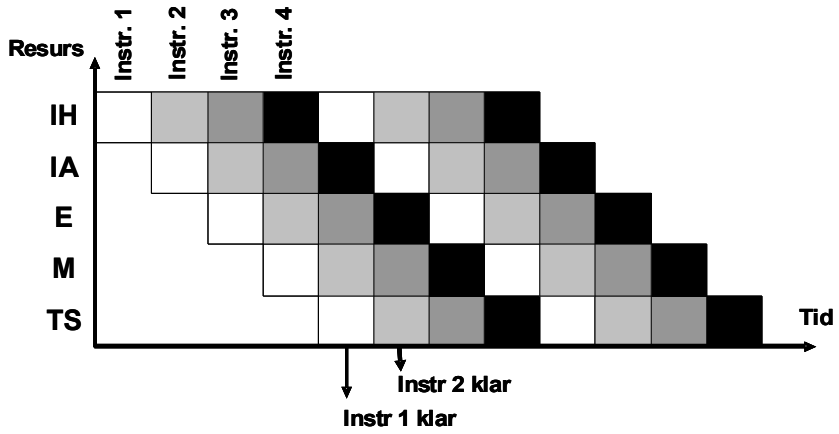
2.3 Processorns instruktionspipeline

Moderna processorer använder flera tekniker för att snabbare exekvera instruktioner. En teknik är *pipelining*. Processorn består av flera delar som en instruktion ska passera igenom. Utan pipelining så passerar instruktionen genom en del i taget, medan de andra delarna bara står och väntar. Inte förrän instruktionen har passerat genom alla delarna, börjar nästa instruktion behandlas av första delen (Eklund, 1999).

Pipelining kan liknas vid Ford's löpande band. Med pipelining så kan varje del i processorn, under ideala förhållanden, arbeta på varsin instruktion samtidigt. En typisk uppdelning av en instruktionspipeline är delarna (Eklund, 1999):

- Instruktionshämtning (IH),
- Instruktionsavkodning (IA),
- Exekvering (E),
- Minnesläsning (M) och
- Tillbakaskrivning (TS).

Först hämtas en ny instruktion av instruktionshämtningen. När den sedan går vidare till instruktionsavkodningen, hämtar instruktionshämtningen en ny instruktion. På så sätt kan alla delarna arbeta på varsin instruktion samtidigt under ideala förhållanden och kan därmed exekvera program snabbare.

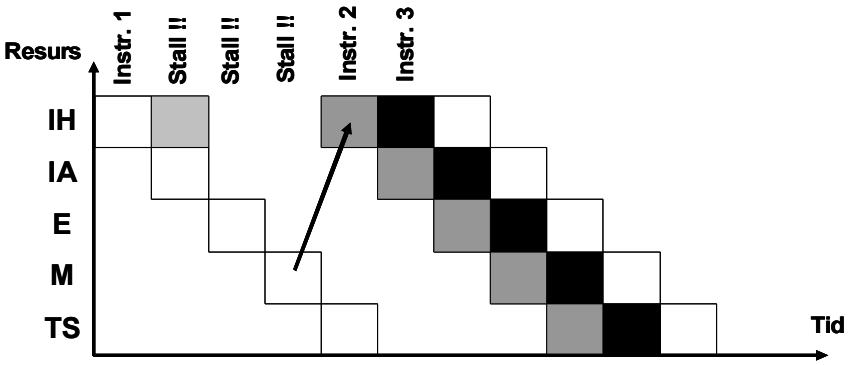


Figur 4. En visuell representation av hur en pipeline arbetar under ideala förhållanden. Rutorna med olika gråtonerna är olika instruktioner som på ett tidssteg passerar ett steg nedåt i instruktionspipelinen.

2.3.1 Pipelinekonflikter

Förhållandena är dock inte alltid ideala och det finns konflikter som kan uppstå när flera instruktioner hanteras samtidigt som i en instruktionspipeline. Pipelinekonflikterna delas in i tre grupper: strukturella konflikter, datakonflikter och styrkonflikter (Eklund, 1999). En konflikt måste hanteras av processorn, annars hotar konflikten att förändra vad programmets instruktioner utför. Det enklaste sättet att hantera konflikter är med en pipeline *stall*. Det innebär att en instruktion i pipelinen (samt alla instruktioner som kom efter den i pipelinen) får vänta en eller flera klockcykler, vilket försämrar exekveringshastigheten något. Det finns ”hårdvarutekniker” i moderna processorer, som hanterar olika slags konflikter på uppfinningsrika sätt och minskar antalet pipeline stall (Eklund, 1999).

Styrkonflikter är orsakade av hoppinstruktioner, vilket omfattar if-satser. Processorn vet inte om if-satsen kommer att utvärderas som sann och om hoppet kommer att tas innan den har hamnat i minnesläsningssteget, vilket innebär tre stallcykler.



Figur 5. En visuell representation av en styrkonflikt i en instruktionspipeline.

Vissa processorer kan utvärdera och utföra hopp i instruktionsavkodningssteget, och därmed minska till bara en stallcykel. En sådan stallcykel brukar kallas för en hopplucka (Eklund, 1999).

2.4 Tidigare arbeten

Massalin (1987) beskriver en superoptimerare (eng: superoptimizer), som genererar optimal programkod för en funktion. Detta genom en uttömmande sökning av kombinationer av processorinstruktioner. Han konstaterade bland annat att villkorliga

hopp i många fall ersattes med andra programmeringssatser, vilket resulterade i snabbare exekvering för vissa datorer utrustade med pipeline. Massalins superoptimerare hade dock stora begränsningar vad gällde programstorleken. Författarens version var begränsad till att generera högst 12 instruktioner, vilket krävde flera timmars exekvering på en 16 MHz dator.

Granlund och Kenner (1992) bygger vidare på Massalins idé rörande superoptimerare, men nyttjar en alternativ teknik för att konstruera en ”GNU Superoptimizer” förkortad GSO. GSO används bland annat för att tillföra programmönster till GCC, vilket gör att den kan eliminera hoppinstruktioner när programkod genereras. Genom att låta GSO söka efter den kortaste programsekvensen för vissa funktioner, kunde man konvertera if-satser till andra effektivare programsatser. Granlund och Kenner (1992) pekar dock på superoptimerarens begränsningar, så snart antalet hopp är stort och hoppalternativen många.

Mueller och Whalley (1995) beskriver ett tillvägagångssätt för att undvika villkorliga hopp, genom att använda automatisk kodreplikering (skapandet av ny programkod som har samma funktionalitet). Trots att replikeringen ofta resulterade i ett större program, (i genomsnitt +16%) minskade exekveringstiden (i genomsnitt -5%). Författarna konstaterade att effektiviteten i att undvika villkorliga hopp, var störst när hoppsatserna utvärderades ofta. Metoden bör därför utnyttjas selektivt, för att uppnå störst effekt.

Pan och Eigenmann (2008) skriver om deras program som automatiskt och effektivt, kan optimera program till den hårdvara som datorn har. De kallar deras program för "PEAK". Det fungerar genom att först hitta de delar av ett program som körs allra oftast, med hjälp av ett profileringsprogram. För dessa delar av programmet, hittas några snabbare alternativa programkoder, från alla möjliga alternativa programkoder, med en snabb uteslutningsalgoritm. Dessa snabba programkoder testas, genom att exekvera dem på den nuvarande hårdvaran och mäta dem med tre pricksäkra mättekniker. Den programkod som presterade bäst i mät-testet, används i det optimerade programmet. De skriver att i deras tester har denna optimeringsprocess tagit 5,85 minuter att utföra i genomsnitt och att optimeringen blivit 12% snabbare, än om det kompileras med GCC's högsta optimeringsval (”-O3”).

Yang (2009) har skrivit ett examensarbete, där han optimerar skript-kod till spelet Neverwinter Nights, som är skriven i skriptspråket NWScript. Han använder fyra optimeringstekniker i sitt arbete, nämligen ”constant folding”, ”dead-branch elimination”, ”function inlining” och en enklare version av ”redundant code elimination”. Han har gjort ett program som automatiskt tillämpar dessa optimeringar på ett skript. Optimeringen har utvärderats på fem skript av varierande storlekar. Resultatet av optimeringen blev i genomsnitt 25% färre antal rader kod i skripten och 14% mindre storlek i byte på skripten i kompilerad byte-kod. Yang (2009) undersöker inte exekveringshastigheten, men beskriver varför en mindre storlek på skript-koden bör exekvera snabbare. Yangs och mitt arbete skiljer sig i att han använder välbeprövade optimeringstekniker, medan jag undersöker en helt ny optimeringsteknik.

3 Problemformulering

Datorspel och program har i allmänhet många villkorliga hopp i form av if-satser. Hoppen minskar effektiviteten i exekveringen, bland annat i form av pipeline-stallar med ökad exekveringstid som följd. Genom att ersätta villkorliga hopp med ovillkorliga hopp som alltid tas, skulle exekveringshastigheten möjligtvis kunna ökas. Eftersom dagens datorspel ofta utnyttjar datorns hela kapacitet så skulle en ökad effektivitet vara betydelsefull.

Det här kapitlet beskriver först den optimeringsteknik som det här examensarbetet ska utvärdera. Efter detta beskrivs examensarbetets två delmål, utveckling och utvärdering. I första delmålet ska testprogram utvecklas, som sedan ska användas i andra delmålet, för att utvärdera optimeringstekniken.

3.1 Optimeringsteknik som ska utvärderas

Vissa delar av koden körs ofta, till exempel en game-loop, som vanligtvis körs vid varje bilduppdatering av ett datorspel. Game-loopen kan innehålla exempelvis AI och fysik med mera. Villkorliga hopp i dessa delar av koden utvärderas mycket frekvent, huruvida hoppet ska tas eller inte. Många av dessa villkorliga hopp kan utvärdera variabler, som sällan blir tilldelade nya värden. Vid sådana fall kan sannolikt exekveringshastigheten kunna ökas, om dessa villkorliga hopp ersätts med ovillkorliga hopp. Det här examensarbetet avgränsas till att endast optimera villkorliga hopp i form av if-satser.

Optimeringstekniken som kommer att undersökas i det här examensarbetet fungerar på följande sätt. Första steget är att identifiera de if-satser som körs ofta, när programmet exekveras. Därefter gäller det att identifiera vilka av dessa if-satser, där de utvärderade variablerna sällan blir tilldelade nya värden under exekveringen, i jämförelse med hur ofta if-satsen körs. Dessa if-satser som ofta körs och som utvärderar variabler som sällan tilldelas nya värden, är goda kandidater till den här optimeringen.

Optimeringen går ut på att ersätta dessa if-satser med dynamiska ovillkorliga hopp. Dynamisk innebär att de måste kunna förändras under exekveringen. Då denna rapport endast behandlar programmeringsspråket C, kan dynamiska ovillkorliga hopp vara funktionspekare eller med GCC's extension ”Labels as values”, kan det även vara tilldelade goto (se avsnitt 2.2.1). Vid användning av funktionspekare måste det skapas en ny funktion, för varje if- eller else-block som ska optimeras. Variabler som används i if- eller else-blocket ska skickas vidare till funktionen som argument. Eftersom både funktionen för if- och else-blocket använder samma funktionspekare, måste deras parametrar se likadana ut. Vid användning av tilldelade goto, finns inget behov av att skapa nya funktioner och tilldelade goto genererar även mindre antal tecken i koden. Därför ska detta examensarbete använda tilldelad goto. Nedan (se figur 6a & 6b) visas ett exempel på hur detta kan se ut, före och efter optimeringen.

```
if(onlineGame &&
 nearbyPlayers > 1)
{
   PacketHandle *h;
   h = netSend(x,y,dX,dY);
   packets->insert(h);
}
...
```

```
void* labelPtr = &&labelFalse;
...
/*if(onlineGame &&
   nearbyPlayers > 1) */
goto *labelPtr;
labelTrue:
{
   PacketHandle *h;
   h = netSend(x,y,dX,dY);
   packets->insert(h);
}
labelFalse:
...
```

Figur 6a (till vänster). Kod i C med ett villkorligt hopp i form av en if-sats. Figur 6b (till höger). Koden från figur 6a efter att optimeringstekniken har blivit applicerad. Den optimerade koden använder ett dynamiskt ovillkorligt hopp i form av en tilldelad goto.

If-satsen (som man ersatte) utvärderade variabler, för att avgöra om koden i if-blocket skulle köras eller ej. Dessa variabler blir tilldelade värden i koden. Vid varje sådan tilldelning ska variablerna utvärderas och det dynamiska ovillkorliga hoppet ska uppdateras, för att det ska hoppa till rätt plats i koden. En metod att säkerställa detta, är att lägga till kod direkt efter varje tilldelning, som utvärderar variablerna och uppdaterar det dynamiska hoppet. Nedan är ett exempel på hur det kan se ut, före och efter optimeringen.

```
...
onlineGame = true;
...
nearbyPlayers = input();
...
```

```
onlineGame = true;
if(nearbyPlayers > 1){
   labelPtr = &&labelTrue;
}
...
nearbyPlayers = input();
if(onlineGame && nearbyPlayers > 1)
{
   labelPtr = &&labelTrue;
}
else{
   labelPtr = &&labelFalse;
}
```

Figur 7a (till vänster). Kod där variabler från if-satsen i figur 6a blir tilldelade värden. Figur 7b (till höger). Koden från figur 7a efter att optimeringstekniken har blivit applicerad. Varje gång variablerna blir tilldelade nya värden så uppdateras det dynamiska ovillkorliga hoppet från figur 6b.

3.2 Problemprecisering

Detta examensarbete, syftar till att undersöka, huruvida den föreslagna optimeringstekniken i avsnitt 3.1, baserad på tilldelade goto, kan minska exekveringstiden för program skrivna i C, med särskilt fokus på datorspelsutveckling.

Undersökningen är indelad i två olika faser, utveckling och utvärdering.

3.3 Metodbeskrivning: Utveckling

Som första steg, ska ett testprogram utvecklas, i syfte att kunna testa om optimeringstekniken minskar exekveringstiden.

Testprogrammet ska implementera och mäta exekveringstiden på tio små programkoder. Dessa programkoder är exempel på if-satser, där optimeringstekniken kan användas samt exempel på tilldelningen av variabler, som också förändras av optimeringstekniken. Programkoderna finns i vanliga och optimerade versioner, för att sedan kunna jämföra exekveringstiden för en vanlig programkod, med en optimerad programkod. De tio programkoderna som ska testas är de följande:

```
if(integerValue > 5)
{
   j++;
}
```

Figur 8a. Test 1 – En vanlig if-sats.

```
goto *labelPtr;
labelTrue:
{
   j++;
}
labelFalse:
```

Figur 8b. Test 2 – En optimerad if-sats.

```
integerValue = value;
```

Figur 9a. Test 3 – Tilldelning av en variabel.

```
integerValue = value;
if(integerValue > 5)
{
   labelPtr = &&labelTrue;
}
else
{
   labelPtr = &&labelFalse;
}
```

Figur 9b. Test 4 – Tilldelning av en variabel i optimerad version.

```
if(integerValue > 5 &&
   integerValue < 10)
{
   j++;
}
```

Figur 10a. Test 5 – En if-sats med två utvärderingar. Den optimerade versionen av den här if-satsen är samma som test 2.

```
integerValue = value;
if(integerValue > 5 &&
   integerValue < 10)
{
   labelPtr = &&labelTrue;
}
else
{
   labelPtr = &&labelFalse;
}
```

Figur 10b. Test 6 – Optimerad version av tilldelning av variablerna från test 5. Den ej optimerade versionen är samma som test 3.

```
if(bool1)
{
   j++;
}
else if(bool2)
{
   j++;
}
else if(bool3)
{
   j++;
}
```

Figur 11a. Test 7 – Tre if-satser varav endast en körs.

```
goto *labelPtr;
label1:
{
   j++;
}
goto labelFalse;
label2:
{
   j++;
}
goto labelFalse;
label3:
{
   j++;
}
labelFalse:
```

Figur 11b. Test 8 – De tre if-satserna i optimerad version.

```
bool1 = value1;
bool2 = value2;
bool3 = value3;
```

Figur 12a. Test 9 – Tilldelning av en variabel från de tre if-satserna från test 7.

```
bool1 = value1;
bool2 = value2;
bool3 = value3;
if(bool1)
{
   labelPtr = &&label1;
}
else if(bool2)
{
   labelPtr = &&label2;
}
else if(bool3)
{
   labelPtr = &&label3;
}
else
{
   labelPtr = &&labelFalse;
}
```

Figur 12b. Test 10 – Tilldelning av variabel i optimerad version.

Dessa programkoder är avsiktligt enkla, för att förhindra att andra faktorer än optimeringskoden påverkar resultatet. Testprogrammet ska kompileras med GCC, utan kompilatorns egna optimeringar.

Testningen kommer utgå från två hypoteser:

1. If-satserna (test 1, 5 och 7) förväntas exekvera snabbare i deras optimerade version (test 2 och 8), eftersom programmet då hoppar direkt till rätt plats och inte behöver utvärdera ett eller flera if-villkor.

2. Tilldelningen av variablerna som används i if-villkoren (test 3 och 9) förväntas exekvera långsammare med optimeringen (test 4, 6 och 10), eftersom flera rader kod har lagts till.

Genom att undersöka hypotes 1 och 2, går det att bestämma när det är värt att optimera en if-sats, med tanke på hur ofta ett if-villkor körs, i förhållande till hur ofta variablerna tilldelas nya värden. En if-sats är värd att optimeras när den totala exekveringstiden minskar, som resultat av optimeringen. Ett exempel skulle kunna vara att optimeringstekniken minskar exekveringstiden för en if-sats med en sekund, men ökar exekveringstiden för tilldelningen med två sekunder. Då behöver if-satsen köras mer än dubbelt så ofta som tilldelningen, för att den totala exekveringstiden ska minska.

Utöver testprogrammet ska en if-sats i ett enkelt datorspel, optimeras med optimeringstekniken, för att lägga fokus på datorspelsutveckling. Datorspelet ska optimeras i dess ”game-loop”, för att visa hur optimeringstekniken kan användas i ett färdigt spel. Datorspelet är ett 2D spel där spelaren styr ett rymdskepp på skärmen och ska skjuta andra rymdskepp som kommer in från höger sida av spelytan. Hur och var optimeringen har gjorts, kommer att presenteras i den här rapporten. Datorspelet ska spelas i dess vanliga version samt i dess optimerade version. Antalet game-loopar per sekund ska mätas i båda versionerna, för att se den ungefärliga förbättringen som optimeringen har gett.

3.4 Metodbeskrivning: Utvärdering

Optimeringstekniken beskriven i avsnitt 3.1 ska utvärderas huruvida den kan minska exekveringstiden. Utvärderingen kan göras med hjälp av experiment eller med beräkning. Beräkning är att med ingående kunskap om processorn och övriga datordelar, kunna räkna ut optimeringsteknikens effektivitet för en datorarkitektur. Denna metod är komplicerad, eftersom man måste veta i detalj varje aktivitet som sker i datorn. Beräkning är därför en för avancerad metod att använda i detta examensarbete.

Optimeringstekniken ska i detta examensarbete utvärderas med hjälp av experiment. I experimenten ska exekveringstider mätas för de tio programkoderna från avsnitt 3.3. Detta med hjälp av testprogrammet från avsnitt 3.3. Det framtagna testprogrammet kommer mäta en programkods exekveringstid i sekunder, med hjälp av funktionen ”clock()” (kommer beskrivas i nästa kapitel), som finns i programmeringsspråket C.

För att med större säkerhet få korrekta resultat, mäts den totala tiden för en programkod, då den exekveras många gånger i en loop och sedan divideras tiden med antal gånger som programkoden har exekverats. Mätresultaten från de vanliga programkoderna kommer att jämföras med de optimerade programkoderna, för att se när optimeringstekniken kan minska exekveringstiden.

Denna mätning kommer att utföras på två PC-datorer, för att se att resultaten inte avviker stort, beroende på den specifika datorarkitekturen och för att få fram ett mer säkert resultat. Om resultatet från den ena datorn kraftigt skiljer sig mot resultatet från den andra datorn, kommer optimeringstekniken att testas på ännu fler PC-datorer för att få ett mer säkert resultat. De två datorerna som testningen ska ske på i det inledande skedet har följande specifikationer:

- Bärbar dator
  OS: Windows XP Professional Service Pack 3
  CPU: Intel Pentium 4 – 2,4 GHz. Socket 478.

L1-Cache: 8 + 12 KB (Data + Instructions)
L2-Cache: 512 KB, fullspeed
Moderkort: Intel Brookdale i845E chipset.
RAM: 512 MB DDR PC2100.

- Stationär dator
  OS: Windows XP Professional Service Pack 3
  CPU: AMD Athlon 64 – 3000+ (1,8 GHz). Socket 939.
  L1-Cache: 64 + 64 KB (Data + Instructions)
  L2-Cache: 512 KB, fullspeed
  Moderkort: Nvidia nForce 630a chipset.
  RAM: 2048 MB DDR PC3200.

Flera av testerna kan få olika långa exekveringstider, beroende på variablernas värden. Om en if-sats utförs eller inte beror på en variabels värde och då en if-sats utförs kan exekveringstiden vara annorlunda än då den inte utförs. Därför kommer flera av testerna utföras mer än en gång, men med olika värden på variablerna. Mellantiden räknas ut för ett test med flera variationer på variablerna. Nedan är en tabell på alla tester från avsnitt 3.3 och vilka startvariabler som kommer att testas:

Tester ↓	Deltest ↓	Variabler-variation ↓
Test 1	Test 1a	`integerValue = 6;`
	Test 1b	`integerValue = 4;`
Test 2	Test 2a	`labelPtr = &&labelTrue;`
	Test 2b	`labelPtr = &&labelFalse;`
Test 3	Test 3	`value = 6;`
Test 4	Test 4a	`value = 6;`
	Test 4b	`value = 4;`
Test 5	Test 5a	`integerValue = 4;`
	Test 5b	`integerValue = 6;`
	Test 5c	`integerValue = 11;`
Test 6	Test 6a	`value = 4;`
	Test 6b	`value = 6;`
	Test 6c	`value = 11;`
Test 7	Test 7a	`bool1 = 1;`
	Test 7b	`bool1 = 0; bool2 = 1;`

	Test 7c	`bool1 = 0; bool2 = 0; bool3 = 1;`
	Test 7d	`bool1 = 0; bool2 = 0; bool3 = 0;`
Test 8	Test 8a	`labelPtr = &&label1;`
	Test 8b	`labelPtr = &&label2;`
	Test 8c	`labelPtr = &&label3;`
	Test 8d	`labelPtr = &&labelFalse;`
Test 9	Test 9	`value1 = 0; value2 = 0; value3 = 0;`
Test 10	Test 10a	`value1 = 1;`
	Test 10b	`value1 = 0; value2 = 1;`
	Test 10c	`value1 = 0; value2 = 0; value3 = 1;`
	Test 10d	`value1 = 0; value2 = 0; value3 = 0;`

Figur 13. Tabell över alla testers variationer på variabler som kommer att testas.

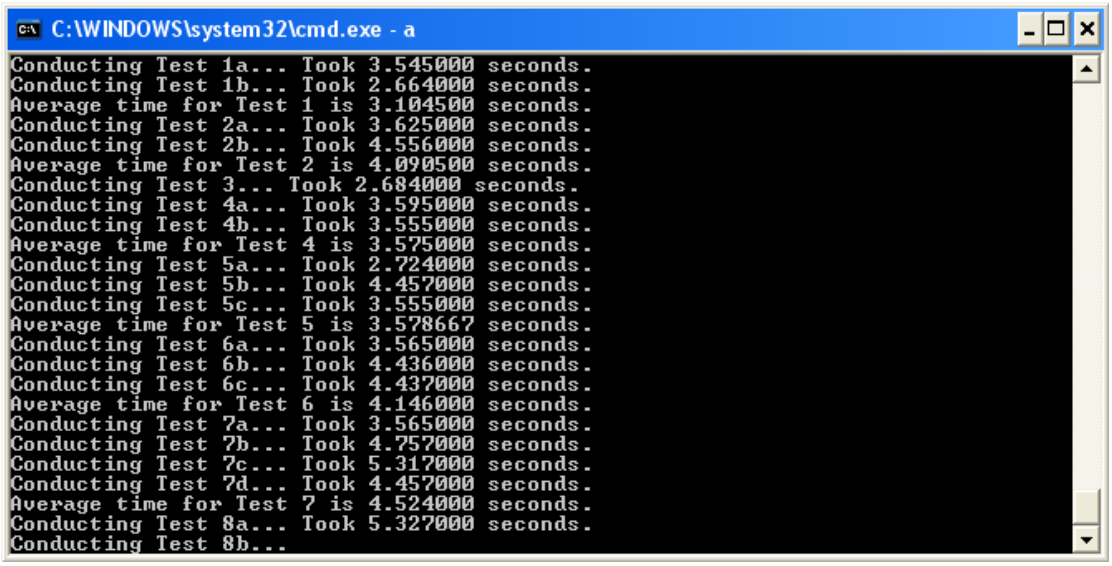
4 Genomförande

Det här kapitlet beskriver hur examensarbetet har genomförts. Avsnitt 4.1 beskriver hur testprogrammet som ska utvärdera optimeringstekniken har implementerats och hur ett datorspel har optimerats med optimeringstekniken. Nästa avsnitt innehåller mätresultat från testprogrammet och datorspelet. Det sista avsnittet i det här kapitlet är en analys av de genomförda mätningarna.

4.1 Det producerade systemet

4.1.1 Testprogrammet

Testprogrammet är skrivet i programmeringsspråket C och mäter exekveringstiden för alla programkoder från avsnitt 3.3 i problemkapitlet. Syftet med att ta fram exekveringstiderna, är att kunna jämföra exekveringstider för programkoder, i dess normala form med dess optimerade form. Testprogrammet mäter tiden för en programkod som körs en miljard gånger. Detta för att mätningsfunktionen inte kan mäta så små tidsenheter, som när man exekverar en av dessa små programkoder endast en gång.



Figur 14. Skärmdump från en körning av testprogrammet.

Tiden mäts med hjälp av funktionen ”clock()” från standardbiblioteket ”time.h” i språket C. Funktionen hämtar processens CPU-tid från operativsystemet. Genom att hämta en start- och sluttid för varje test, kan programmet räkna ut exekveringstiden. Testprogrammet har kompilerats med GCC (version 3.4.5-mingw) utan optimeringsparametrar, det vill säga att GCC inte automatiskt optimerar programkoden.

Många av programkoderna körs i flera tester, där varje deltest har olika värden på variabler, eftersom dessa variabler påverkar programflödet och därmed möjligtvis exekveringstiden. En genomsnittstid räknas ut för varje programkod, som är indelad i flera tester. Vilka värden variablerna har i de olika testerna är specificerat i tabellen figur 13 i avsnitt 3.4. Varje test har programmerats i testprogrammets på följande vis:

```
--- Tilldelning av variabler som används i testet ---
i = 0;
start = clock();
while(i < 1000000000){
   --- Koden från en av programkoderna från avsnitt 3.3 ---
   i++;
}
end = clock();
```

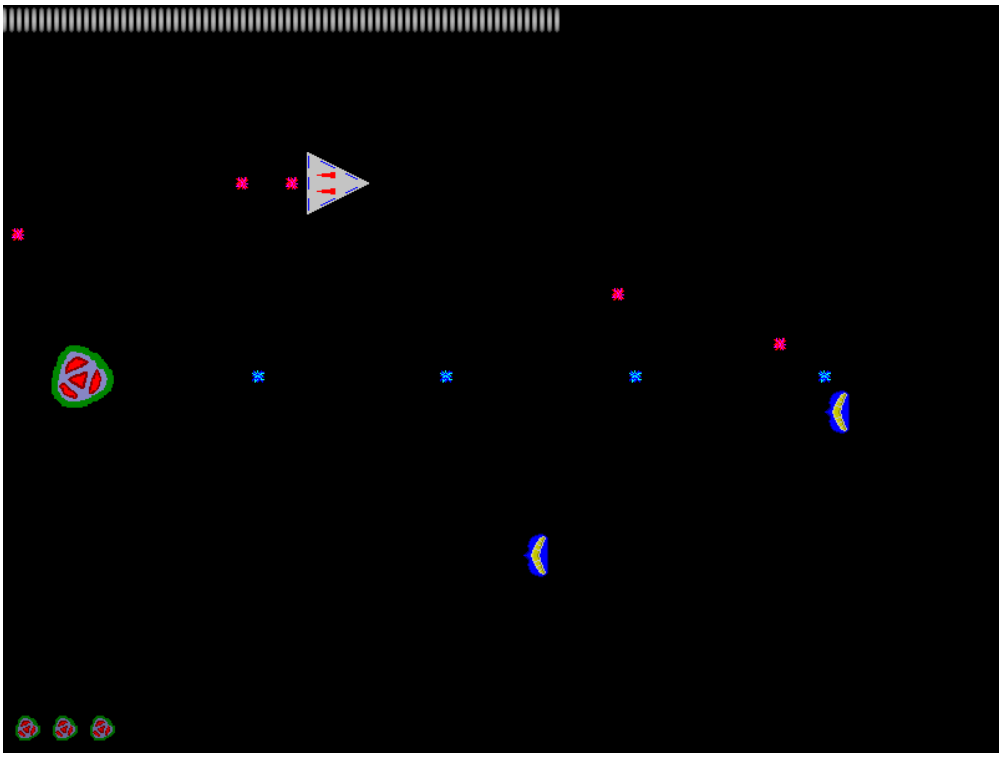
Figur 15. Varje programkods-test är implementerad i den här formen.

För den fullständiga källkoden, se bilaga 2. Testprogrammet ska användas för att ta fram mätresultat, som i sin tur ska användas för att utvärdera optimeringstekniken.

4.1.2 Datorspelet

Datorspelet ska enligt problembeskrivningen optimeras med optimeringstekniken som använder tilldelade goto. Datorspelet har dock inte optimerats med tilldelade goto, utan med funktionspekare, som också är en form av ett ovillkorligt hopp (se avsnitt 3.1). Detta på grund av stora problem, då spelet kompilerades med GCC. Spelets inladdning av bilder fungerade inte, vilket resulterade i programfel och spelet kunde inte köras. Eftersom inladdningen av bilder sköts av ett programbibliotek från en tredje part, togs valet att programmera om optimeringen med funktionspekare, för att kunna kompilera spelet i Microsoft Visual Studio.

I spelet styr spelaren ett rymdskepp och ska skjuta fiendeskepp som kommer in från sidan av skärmen. Spelet är programmerat i C++ och använder i grunden programbiblioteket Simple DirectMedia Layer (2009) för att rita ut grafiken.



Figur 16. Skärmdump från rymdspelet.

En if-sats i spelets gameloop ska optimeras, för att visa hur optimeringstekniken kan användas i ett spel. If-satsen som valts har ett if-block, men inget else-block. Den

existerar i en funktion med namnet `update` som körs en gång varje gameloop. If-satsen ser ut på följande vis:

```
if(shipDead)
{
	m_fireVector.clear();
	m_enemyVector.clear();
	m_enemyFireVector.clear();
	m_powerupVector.clear();
	m_shipVector[shipIndex]->setPosition(50, 215);
	m_shipVector[shipIndex]->subtractExtraLife(1);
	m_shipVector[shipIndex]->setHealth(500);
	m_shipVector[shipIndex]->resetCannon();
	m_level = 0;
	if(m_shipVector[shipIndex]->getExtraLife() <= 0)
	{
		close();
		ofstream result("RESULTAT Normal.txt");
		result << (float)m_updates / m_seconds;
		result.close();
	}
}
```

Figur 17. Innehållet i if-satsen som valts för optimering.

Innehållet i blocket har flyttats till en ny funktion, som funktionspekaren ska kunna peka på. En tom funktion har även skapats, som funktionspekaren kan peka på, om if-satsen skulle utvärderas till att inte gå in i if-blocket.

```
void SGLApp::shipDead_True(unsigned int shipIndex)
{
	m_fireVector.clear();
	m_enemyVector.clear();
	m_enemyFireVector.clear();
	m_powerupVector.clear();
	m_shipVector[shipIndex]->setPosition(50, 215);
	m_shipVector[shipIndex]->subtractExtraLife(1);
	m_shipVector[shipIndex]->setHealth(500);
	m_shipVector[shipIndex]->resetCannon();
	m_level = 0;
	if(m_shipVector[shipIndex]->getExtraLife() <= 0)
	{
		ofstream result("RESULTAT Function Pointer.txt");
		result << (float)m_updates / m_seconds;
		result.close();
		close();
	}
}

void SGLApp::shipDead_False(unsigned int shipIndex)
{
}
```

Figur 18. If-blocket i en funktion som ska anropas med en funktionspekare.

If-satsen som optimeras finns i funktionen `update`, som körs en gång i varje iteration av gameloopen. I följande figur finns de väsentliga delarna i funktionen med tanke på optimeringen:

```
void SGLApp::update(float time)
{
      ++m_updates;
      m_seconds += time;
      ...

      bool shipDead = false;
      void (SGLApp::*funcPtr)(unsigned int) =
            &SGLApp::shipDead_False;
      ...

      shipDead = ...;
      if(shipDead){
            funcPtr = &SGLApp::shipDead_True;
      }
      ...

      shipDead = ...;
      if(shipDead){
            funcPtr = &SGLApp::shipDead_True;
      }
      ...

      //if(shipDead)
      (*this.*funcPtr)(shipIndex);
      ...
}
```

Figur 19. `update`-funktionen där `shipDead` och funktionspekaren `funcPtr` deklareras, initieras och senare tilldelas värden. På slutet anropas `funcPtr`.

Funktionen `update` utför egentligen mycket mer än vad som syns i figuren ovan. De första två kodraderna inuti funktionen från figuren ovan, visar hur spelet mäter antalet uppdateringar och sekunder. Där variabeln `shipDead` deklareras, initieras den till falskt. På samma plats efter optimeringen, deklareras och initieras funktionspekaren `funcPtr` till att peka på funktionen `shipDead_False`.

På två ställen i koden tilldelas `shipDead` ett värde. Där är kod tillagd som kontrollerar om `shipDead` har ändrats till sant och om detta är fallet, ändras funktionspekaren till att peka på funktionen `shipDead_True` istället. I slutet av koden ovan anropas funktionspekaren på den position, där if-satsen tidigare låg. Denna optimerade version och den vanliga versionen ska jämföras i antalet uppdateringar per sekund.

4.2 Genomförda mätningar

Mätningar med testprogrammet (testprogrammet beskrevs i avsnitt 4.1.1) har utförts på två datorer, en bärbar dator och en stationär dator, vars specifikationer finns i avsnitt 3.4. På båda datorerna har testprogrammet körts tre gånger. Det genomsnittliga resultatet från båda datorernas alla körningar har används.

Utöver de tio testerna från problemkapitlet har ytterligare fem tester programmerats in i testprogrammet, vilka ska utvärdera en version av optimeringstekniken, som istället använder funktionspekare. De fem nya testerna liknar de fem tester, som redan använder optimeringstekniken, med skillnaden att de nya använder funktionspekare istället för tilldelad goto. Deras programkoder finns också med i testprogrammets källkod i bilaga 2.

En sammanfattning av mätningarna presenteras i tabellerna här nedan. Mättiderna är angivna i nanosekunder. De avslöjar att programkoderna, som använder optimeringstekniken, tar längre tid att exekvera, än de vanliga programkoderna. En tabell med samtliga deltesters mätdata kan hittas i bilaga 1.

Test 1: Vanlig if-sats med en utvärdering.	Test 5: Vanlig if-sats med två utvärderingar.	Test 2: Optimerad version av dessa två if-satser.	FP Test 2: Optimerad med funktionspekare.
3,28 ns	3,71 ns	4,22 ns	7,43 ns

Figur 20. Exekveringstider för en enkel if-sats i vanlig och optimerad form.

Test 3: Vanlig tilldelning.	Test 4: Optimerad version av tilldelning, för if-satsen med en utvärdering.	Test 6: Optimerad version av tilldelning, för if-satsen med två utvärderingar.
3,44 ns	3,57 ns	3,96 ns

Figur 21. Exekveringstider för tilldelning av en variabel, i vanlig och optimerad form. (De optimerade versionerna är optimerade med tilldelade goto, samma tester fast optimerade med funktionspekare ger ungefär lika tid som de optimerade med tilldelade goto.)

Test 7: En vanlig ”flervägs”-if-sats som utvärderar upp till tre variabler.	Test 8: Optimerad version av ”flervägs”-if-satsen.	FP Test 8: Optimerad version med funktions-pekare.
4,56 ns	4,67 ns	7,27 ns

Figur 22. Exekveringstider för en if-sats med utseendet ”if... else if... else if...” (kommer att hänvisas till en ”flervägs”-if-sats i resten av rapporten), i vanlig och optimerad form.

Test 9: Vanlig tilldelning av variablerna från ”flervägs”-if-satsen.	Test 10: Optimerad version av tilldelning av variablerna från ”flervägs”-if-satsen.
3,49 ns	5,55 ns

Figur 23. Exekveringstider för tilldelning av de tre variablerna, som utvärderas i ”flervägs”-if-satsen i vanlig och optimerad form. (Den optimerade versionen är optimerad med tilldelad goto, samma test fast optimerad med funktionspekare ger ungefär lika tid.)

Datorspelets antal uppdateringar per sekund, är ungefär lika många både med och utan optimeringen. Den normala versionen har i tre mätningar gett 76,2, 76,5 och 77,2 uppdateringar per sekund. Den optimerade versionen har gett 77,8, 76,1 och 76,9 uppdateringar per sekund.

4.3 Analys av mätningar

Som kan ses i figur 20 till 23, ökar exekveringstiden med optimeringstekniken i samtliga tester. Hypotesen från problemkapitlet, att if-satser optimerade med optimeringstekniken bör exekvera snabbare, är därmed felaktig. Som förväntat, ökar exekveringstiden vid tilldelning av variabler, efter att optimeringstekniken har blivit tillämpad. Från mätningarna kan det utläsas att optimeringstekniken ökar exekveringstiden med följande procent:

Användnings-område ↓	If-satser optimerade med tilldelade goto jämfört med ej optimerad version	If-satser optimerade med funktionspekare jämfört med ej optimerad version	Variabel-tilldelning i optimerad version jämfört med ej optimerad version
Enkel if-sats med en utvärdering	+ 29 %	+ 127 %	+ 4 %
Enkel if-sats med två utvärderingar	+ 14 %	+ 100 %	+ 15 %
En flervägs-if-sats med upp till tre utvärderingar	+ 2 %	+ 59 %	+ 59 %

Figur 24. Ökning av exekveringstiden i procent i de olika testerna efter att optimeringstekniken har tillämpats.

Även om den procentuella ökningen är stor, rör det sig bara om en ökning på några nanosekunder. Det finns dock ingen anledning att använda en optimeringsteknik som inte ger ett bättre resultat.

Optimeringstekniken ger ett bättre resultat i den optimerade ”flervägs”-if-satsen (deltest 8d), än i dess icke optimerade motsvarighet (deltest 7d). Detta speciellt på den stationära datorn, där exekveringstiden minskade med 36%. På den bärbara datorn är minskningen 1% (se bilaga 1). I deltest 7d är variablernas värden sådana att det sker tre if-sats-utvärderingar och tre hopp i ”flervägs”-if-satsen. Varför deltest 8d exekverar snabbare är förmodligen därför, att den gör enbart ett hopp och att den inte utvärderar något värde.

Det är troligt att optimeringstekniken kan minska exekveringstiden för ännu större ”flervägs”-if-satser, om de i genomsnitt genomför tre eller fler utvärderingar och hopp. I dessa fall måste dock tilldelningen av variablerna, som utvärderas i if-satserna, fortfarande ske ganska sällan i förhållande till hur ofta if-satserna körs, för att kunna minska programs totala exekveringstid. Stationära datorns mätresultat (se bilaga 1) från den optimerade ”flervägs”-if-satsen har minskat med 2,25 ns (deltest 8d jämfört med deltest 7d). Exekveringstiden för tilldelning har ökat med 3,65 ns (deltest 10d jämfört med test 9). Därför kommer den totala exekveringstiden endast att minska om if-satsen körs åtminstone 63% oftare än tilldelningar av de berörda variablerna ($3{,}65 / 2{,}25 \approx 1{,}62$).

Resultaten från mätningarna av datorspelet är ungefär det samma med och utan optimering. Spelet består av väldigt många olika delar, bland annat en grafikmotor. På

grund av detta är det svårt att påvisa optimeringens mindre effekt. De båda mätningarnas resultat varierar upp och ner då spelet testas flera gånger. Det beror troligen på hur spelandet utformat sig under en testkörning och att spelet skapar nya fiender slumpmässigt. Från testprogrammets mätningar kan det dock härledas att exekveringstiden kommer att öka för datorspelet efter optimeringen, även om det bara rör sig om några nanosekunder mer per uppdatering.

5 Slutsatser

5.1 Resultatsammanfattning

Ett testprogram har utvecklats, som mäter exekveringstider för korta programkoder. Testerna visar att de programkoder, som använder optimeringstekniken tar längre tid att exekvera än deras motsvarande vanliga programkoder. Det betyder att optimeringstekniken inte förbättrar exekveringstiden för alla if-satser.

Optimeringstekniken har dock gett en snabbare exekvering i ett deltest, där if-satsen har behövt utvärderat många variabler innan den kommit fram till rätt exekveringsväg. Exekveringstiden ökar alltid för tilldelningen av de variabler, som behöver övervakas av optimeringstekniken. En analys av programkoden, som ska optimeras, krävs för att se om programmets totala exekveringstid kommer att minska. Detta görs genom att undersöka hur ofta if-satsen, som har optimerats, körs i förhållande till tilldelningen av de variabler, som övervakas. Detta multipliceras sedan med den minskade exekveringstiden för if-satsen. Om denna multiplicerade minskning av exekveringstid, är större än ökningen av exekveringstid vid tilldelning av de övervakade variablerna, kommer programmets totala exekveringstiden att minska. Alternativt kan programmets totala exekveringstiden mätas före och efter optimeringstekniken har applicerats.

Optimeringstekniken har även tillämpats på ett datorspel, dock baserad på funktionspekare istället för tilldelade goto. Antalet uppdateringar per sekund har mäts, men inga slutsatser har dragits från dessa mätningar, då skillnaden mellan de olika versionernas mätresultat har varit tvetydiga och små.

Syftet med det här examensarbetet, är att undersöka huruvida den föreslagna optimeringstekniken kan minska exekveringstiden för dataprogram. Enligt deltestet, som gav minskad exekveringstid, kan optimeringstekniken minska exekveringstiden för dataprogram i några få och specifika fall.

5.2 Diskussion

Jag anser att testerna, som används är en lämplig metod för att avgöra optimeringsteknikens effektivitet, eftersom det är en faktisk exekvering och inte bara en teoretisk beräkning. Att använda deltester anser jag vara viktigt, eftersom de testar programkodens alla exekveringsvägar. Alla deltesterna i ett test har olika startvärden på variabler, i övrigt är deras programkod identiskt. Detta för att täcka in alla exekveringsvägar eller hopp, som den korta programkoden kan ta. Dock antar jag i mina mätningar att varje exekveringsväg körs lika ofta, men i verkliga fallet kan det vara väldigt varierande, beroende på vilket program det är och även hur användaren använder programmet.

Jag anser att optimeringstekniken inte tillhör de småskaliga optimeringar, som många av GCCs automatiska optimeringar tillhör. I min mening tillhör min optimeringsteknik den typ av större och övergripande optimeringar, som en programmerare själv implementerar när ett fungerande program ska optimeras. Optimeringstekniken har ett litet användningsområde och det krävs en omfattande analys av programkoden för att hitta platser, som kan dra nytta av den.

Ingen av GCCs automatiska optimeringar liknar optimeringstekniken i denna rapport (GCC online documentation, 2010). Vid användandet av kompilatorn GCC, bör programmeraren stänga av GCCs automatiska optimering ”global common

subexpression elimination pass”, när programmeraren använder optimeringstekniken. Denna automatiska optimering kan ge försämrad prestanda i program med tilldelade goto. Den är aktiverad på optimeringsnivåerna -O2, -O3 och -Os. ”Global common subexpression elimination pass” kan stängas av med parametern ”-fno-gcse”.

En anledning till att en tilldelad goto inte presterat bättre än en enkel if-sats i mina tester kan vara att båda datorerna, som mätningarna utfördes på, har processorer utrustade med en teknik kallad ”dynamic branch prediction” (Fog, 2010). Denna teknik undviker ofta styrkonflikter och förlorade instruktionscyklar med hjälp av information om hoppet tidigare har tagits eller inte. I mina tester kan denna teknik förutsäga om de villkorliga hoppen ska tas eller inte med nära 100% säkerhet.

Optimeringstekniken kan därför inte snabba upp if-satser, vars utvärderade variabler sällan blir tilldelade nya värden, när exekveringen sker på dagens vanliga processorer, eftersom de är utrustade med ”dynamic branch prediction”.

De switch-case-satser, som i genomsnitt utvärderar en variabel tre eller fler gånger innan den kommit fram till rätt ”case”, kan troligen exekvera snabbare med optimeringstekniken. Switch-case-satser används ofta i program och datorspel, därför skulle detta utöka optimeringsteknikens användningsområde avsevärt. Dock finns det redan en känd optimering att konvertera en switch-case-sats till en *hopptabell* (eng: branch table). Detta kan vara en automatisk optimering i kompilatorer (Branch table, 2010).

Hopptabeller konstrueras vanligen med assemblerspråk. Kort förklarat fungerar en hopptabell så att flera ovillkorliga hopp lagras i en array-datastruktur. Med hjälp av en variabels värde hoppar programmet till en plats i arrayen där ett ovillkorligt hopp finns och tar det hoppet till rätt plats i koden. Det är lämpligt att göra om en switch-case-sats till en hopptabell, då satsens olika ”case” kan ordnas i en enkel ordning (exempelvis 1, 2, 3, 4 o.s.v., eller 36, 38, 40, 42 o.s.v.), utan att det blir stora luckor med saknade tal i den ordningen (Branch table, 2010).

Då det är lämpligt att använda en hopptabell, är den att föredra framför optimeringstekniken i denna rapport, eftersom en hopptabell inte behöver övervaka när en variabel tilldelas nya värden.

Då det är olämpligt att konvertera en switch-case-sats till en hopptabell, kan det fortfarande vara lämpligt att använda optimeringstekniken för att öka exekveringshastigheten, eftersom den inte har något krav att de olika ”case” ska kunna ordnas i någon specifik ordning. En switch-case-sats utvärderar vanligtvis endast en variabel och därför behöver optimeringstekniken, efter den har tillämpats på switch-case-satsen, bara övervaka när den variabeln tilldelas nya värden. Programmeraren behöver givetvis analysera hur ofta switch-case-satsen körs i förhållande till hur ofta variabeln tilldelas nya värden.

5.3 Framtida arbete

Ett framtida arbete kan vara en noggrannare undersökning av de situationer där optimeringstekniken kan minska exekveringstiden. Det vore mest önskvärt att ett sådant framtida arbete leder till ett färdigt program, som automatiskt kan analysera och optimera källkod med optimeringstekniken, med målet att programmet ska exekverar snabbare.

Ett annat framtida arbete kan vara att utvärdera en modifierad version av optimeringstekniken från det här examensarbetet. Den modifierade tekniken ska hitta

funktioner innehållande if-satser, vars utvärderade variabler sällan blir tilldelade nya värden. Sedan skapa flera versioner av den här funktionen, dock med dessa if-satser ersatta med enbart koden från en av if-satsens exekveringsvägar och därmed spara in exekveringstid. De många nya funktionerna innehåller olika exekveringsvägar från if-satserna. Den funktion med den korrekta exekveringsvägen, med tanke på variablernas nuvarande värden, anropas med en funktionspekare. Variablerna i frågan övervakas på samma sätt, som i det här examensarbetet och på så sätt är funktionspekaren alltid inställd på rätt funktion.

Om denna nya teknik finner flera funktioner, som kan optimeras och de alla anropas från en gemensam funktion, då kan flera versioner av den gemensamma funktionen skapas. Genom att använda funktionspekare enbart till den gemensamma funktionen, kan mer exekveringstid sparas. De tidigare funktionerna kan då anropas normalt, men det blir många versioner av den gemensamma funktionen. Två nackdelar med optimeringen är förmodligen att det kompilerade programmets storlek blir mycket större och att funktioner, som anropas av en funktionspekare, inte kan dra nytta av optimeringen ”function inlining”. Fördelen med optimeringen är att den kan minska exekveringstiden.

Referenser

Assigned Goto. (2010) Wikipedia. Hämtad 2010-02-10 från ”http://en.wikipedia.org/wiki/Goto#Assigned_GOTO”

Branch table. (2010) Wikipedia. Hämtad 2010-05-20 från ”http://en.wikipedia.org/wiki/Branch_table”

Eklund, Sven (1999), *Modern mikroprocessordesign: med introduktion till parallella system.* (sid. 123-138). Lund: Studentlitteratur

Fog, Agner (2010), *The microarchitecture of Intel, AMD and VIA CPUs.* Hämtad 2010-06-04 från ”http://www.agner.org/optimize/microarchitecture.pdf”

GCC online documentation – Optimize options. (2010) Hämtad 2010-05-20 från ”http://gcc.gnu.org/onlinedocs/gcc-4.4.4/gcc/Optimize-Options.html”

Granlund, T. & Kenner, R. (1992), *Eliminating branches using a superoptimizer and the GNU C compiler.* (sid. 341-352). ACM SIGPLAN Conference on Programming Language Design and Implementation, San Francisco, California.

Massalin, H. (1987), *Superoptimizer: a look at the smallest program.* (sid. 122-126). In Proceedings of the Second international Conference on Architectual Support For Programming Languages and Operating Systems, Palo Alto, California.

Mueller, F. & Whalley, D. B. (1995), *Avoiding conditional branches by code replication.* (sid. 56-66). ACM SIGPLAN Conference on Programming Language Design and Implementation, La Jolla, California.

Pen, Z. & Eigenmann, R. (2008), *PEAK—a fast and effective performance tuning system via compiler optimization orchestration.* (artikel 17). ACM Transactions on Programming Languages and Systems, Vol. 30, No. 3.

Simple DirectMedia Layer. (2009) [Programbibliotek] Tillgänglig på Internet ”www.libsdl.org”

Wall, Kurt & Von Hagen, William (2004), *The definitive guide to GCC.* (sid. 111-112). Berkeley, California: Apress

Yang, Y. (2009), *Code optimization and detection of script conflicts in video games*. Alberta: University of Alberta. Unpublished Master's Thesis. Tillgänglig på Internet ”http://hdl.handle.net/10048/586”

Bilaga 1: Testernas exekveringstider

Resultaten från testprogrammets alla mätningar kan ses i tabellen nedanför. Tabellen visar exekveringstider från varje deltest av varje test. Deltesterna i ett test har olika värden på variabler innan programkoden och mätningen påbörjas. Dessa variabel-variationer finns angivna i avsnitt 3.4. Testets resultat är genomsnittet av dess deltester och finns under ”*Genomsnitt*” i tabellen. För varje test finns det också ett genomsnitt av båda datorernas mätresultat, vars tider även presenteras i avsnittet ”4.2 Genomförda mätningar”.

Testerna kallade ”FP Test” använder en version av optimeringstekniken med funktionspekare istället för tilldelade goto. De individuella testernas programkoder, förutom funktionspekare-testerna, är beskrivna i avsnitt 3.3 av rapporten. Hela testprogrammets källkod, inklusive funktionspekare-testerna, finns i bilaga 2.

	Bärbar dator ↓		**Stationär dator ↓**		**Båda ↓**
Deltest ↓	*Tid ↓*	*Genomsnitt*	*Tid ↓*	*Genomsnitt*	*Genomsnitt*
Test 1a	3,39	2,96	4,34	3,6	3,28
Test 1b	2,53		2,86		
Test 2a	4,16	4,15	4,08	4,29	4,22
Test 2b	4,13		4,5		
FP Test 2a	9,05	8,31	6,32	6,54	7,43
FP Test 2b	7,57		6,77		
Test 3	2,52	2,52	4,36	4,36	3,44
Test 4a	3,36	3,37	3,94	3,78	3,57
Test 4b	3,38		3,61		
FP Test 4a	3,38	3,47	3,59	3,47	3,47
FP Test 4b	3,57		3,34		
Test 5a	2,52	3,36	3,82	4,05	3,71
Test 5b	4,2		3,85		
Test 5c	3,36		4,48		
Test 6a	3,37	3,93	3,44	3,99	3,96
Test 6b	4,21		4,44		
Test 6c	4,21		4,1		

FP Test 6a	3,37	3,97	3,42	3,81	3,89
FP Test 6b	4,34		4,07		
FP Test 6c	4,2		3,94		
Test 7a	3,38	4,22	4,46	4,91	4,56
Test 7b	4,2		4,5		
Test 7c	5,06		4,5		
Test 7d	4,23		6,19		
Test 8a	4,03	4,7	5,64	4,64	4,67
Test 8b	4,96		4,51		
Test 8c	5,61		4,5		
Test 8d	4,2		3,94		
FP Test 8a	8,41	7,99	6,38	6,56	7,27
FP Test 8b	7,56		6,75		
Test 9	3,31	3,31	3,66	3,66	3,49
Test 10a	4,42	5,33	4,51	5,77	5,55
Test 10b	5,06		5,63		
Test 10c	5,89		5,65		
Test 10d	5,93		7,31		
FP Test 10a	4,2	5,32	4,53	5,73	5,52
FP Test 10b	5,12		4,87		
FP Test 10c	5,96		7,32		
FP Test 10d	6,01		6,19		

(Exekveringstiderna ovan är angivna i sekunder, för ett test upprepat i en loop en miljard gånger. Tiden för enbart en körning av en programkod, blir ungefär samma tid i nanosekunder, eftersom en nanosekund är en miljarddel av en sekund.)

Bilaga 2: Testprogrammets källkod

```
#include <stdio.h>
#include <time.h>

#define ITERATIONS 1000000000


double diffclock(clock_t clock1,clock_t clock2)
{
      double diffticks=clock1-clock2;
      double diffInSec=(diffticks)/CLOCKS_PER_SEC;
      return diffInSec;
}


void testFuncPointerTrue(int *foo){ (*foo)++; }
void testFuncPointerFalse(int *foo){ }


int main()
{
clock_t start, end;
int foo = 0;
int i = 0;
srand(time(0));
int integerValue = 0;
void* labelPtr;
int value = 0;
int bool1, bool2, bool3;
int value1, value2, value3;
void (*funcPointer)(int *foo);



printf("Conducting Test 1a... ");

integerValue = 6;
i = 0;
start = clock();
while(i < ITERATIONS){
      if(integerValue > 5)
      {
            foo++;
      }
      i++;
}
end = clock();
double test1a = diffclock(end, start);
printf("Took %f seconds.\n", test1a); //row 37

printf("Conducting Test 1b... ");

integerValue = 4;
i = 0;
start = clock();
while(i < ITERATIONS){
      if(integerValue > 5)
      {
            foo++;
      }
```

```
        i++;
}
end = clock();
double test1b = diffclock(end, start);
printf("Took %f seconds.\n", test1b);
double test1 = (test1a + test1b) / 2;
printf("Average time for Test 1 is %f seconds.\n", test1);



printf("Conducting Test 2a... ");
labelPtr = &&test2aLabelTrue;
i = 0;
start = clock();
while(i < ITERATIONS){
        goto *labelPtr;
test2aLabelTrue:
        {
                foo++;
        }
test2aLabelFalse:
        i++;
}
end = clock();
double test2a = diffclock(end, start);
printf("Took %f seconds.\n", test2a);

printf("Conducting Test 2b... ");
labelPtr = &&test2bLabelFalse;
i = 0;
start = clock();
while(i < ITERATIONS){
        goto *labelPtr;
test2bLabelTrue:
        {
                foo++;
        }
test2bLabelFalse:
        i++;
}
end = clock();
double test2b = diffclock(end, start);
printf("Took %f seconds.\n", test2b);
double test2 = (test2a + test2b) / 2;
printf("Average time for Test 2 is %f seconds.\n", test2);



printf("Conducting Test Function Pointer 2a... ");
funcPointer = &testFuncPointerTrue;
i = 0;
start = clock();
while(i < ITERATIONS){
        funcPointer(&foo);
        i++;
}
end = clock();
double testFP2a = diffclock(end, start);
printf("Took %f seconds.\n", testFP2a);

printf("Conducting Test Function Pointer 2b... ");
funcPointer = &testFuncPointerFalse;
```

```
i = 0;
start = clock();
while(i < ITERATIONS){
      funcPointer(&foo);
      i++;
}
end = clock();
double testFP2b = diffclock(end, start);
printf("Took %f seconds.\n", testFP2b);
double testFP2 = (testFP2a + testFP2b) / 2;
printf("Average time for Test Function Pointer 2 is %f seconds.\n",
testFP2);



printf("Conducting Test 3... ");
value = 6;
i = 0;
start = clock();
while(i < ITERATIONS){
      integerValue = value;
      i++;
}
end = clock();
double test3 = diffclock(end, start);
printf("Took %f seconds.\n", test3);



printf("Conducting Test 4a... ");
value = 6;
i = 0;
start = clock();
while(i < ITERATIONS){
      integerValue = value;
      if(integerValue > 5)
      {
            labelPtr = &&test2aLabelTrue;
      }
      else
      {
            labelPtr = &&test2aLabelFalse;
      }
      i++;
}
end = clock();
double test4a = diffclock(end, start);
printf("Took %f seconds.\n", test4a);

printf("Conducting Test 4b... ");
value = 4;
i = 0;
start = clock();
while(i < ITERATIONS){
      integerValue = value;
      if(integerValue > 5)
      {
            labelPtr = &&test2aLabelTrue;
      }
      else
      {
            labelPtr = &&test2aLabelFalse;
```

```
        }
        i++;
}
end = clock();
double test4b = diffclock(end, start);
printf("Took %f seconds.\n", test4b);
double test4 = (test4a + test4b) / 2;
printf("Average time for Test 4 is %f seconds.\n", test4);



printf("Conducting Test Function Pointer 4a... ");
value = 6;
i = 0;
start = clock();
while(i < ITERATIONS){
        integerValue = value;
        if(integerValue > 5)
        {
                funcPointer = &testFuncPointerTrue;
        }
        else
        {
                funcPointer = &testFuncPointerFalse;
        }
        i++;
}
end = clock();
double testFP4a = diffclock(end, start);
printf("Took %f seconds.\n", testFP4a);

printf("Conducting Test Function Pointer 4b... ");
value = 4;
i = 0;
start = clock();
while(i < ITERATIONS){
        integerValue = value;
        if(integerValue > 5)
        {
                funcPointer = &testFuncPointerTrue;
        }
        else
        {
                funcPointer = &testFuncPointerFalse;
        }
        i++;
}
end = clock();
double testFP4b = diffclock(end, start);
printf("Took %f seconds.\n", testFP4b);
double testFP4 = (testFP4a + testFP4b) / 2;
printf("Average time for Test Function Pointer 4 is %f seconds.\n",
testFP4);



printf("Conducting Test 5a... ");
integerValue = 4;
i = 0;
start = clock();
while(i < ITERATIONS){
        if(integerValue > 5 && integerValue < 10)
```

```
        {
                foo++;
        }
        i++;
}
end = clock();
double test5a = diffclock(end, start);
printf("Took %f seconds.\n", test5a); //row 37

printf("Conducting Test 5b... ");
integerValue = 6;
i = 0;
start = clock();
while(i < ITERATIONS){
        if(integerValue > 5 && integerValue < 10)
        {
                foo++;
        }
        i++;
}
end = clock();
double test5b = diffclock(end, start);
printf("Took %f seconds.\n", test5b);

printf("Conducting Test 5c... ");
integerValue = 11;
i = 0;
start = clock();
while(i < ITERATIONS){
        if(integerValue > 5 && integerValue < 10)
        {
                foo++;
        }
        i++;
}
end = clock();
double test5c = diffclock(end, start);
printf("Took %f seconds.\n", test5c);
double test5 = (test5a + test5b + test5c) / 3;
printf("Average time for Test 5 is %f seconds.\n", test5);



printf("Conducting Test 6a... ");
value = 4;
i = 0;
start = clock();
while(i < ITERATIONS){
        integerValue = value;
        if(integerValue > 5 && integerValue < 10)
        {
                labelPtr = &&test2bLabelTrue;
        }
        else
        {
                labelPtr = &&test2bLabelFalse;
        }
        i++;
}
end = clock();
double test6a = diffclock(end, start);
printf("Took %f seconds.\n", test6a);
```

```
printf("Conducting Test 6b... ");
value = 6;
i = 0;
start = clock();
while(i < ITERATIONS){
	integerValue = value;
	if(integerValue > 5 && integerValue < 10)
	{
		labelPtr = &&test2bLabelTrue;
	}
	else
	{
		labelPtr = &&test2bLabelFalse;
	}
	i++;
}
end = clock();
double test6b = diffclock(end, start);
printf("Took %f seconds.\n", test6b);

printf("Conducting Test 6c... ");
value = 11;
i = 0;
start = clock();
while(i < ITERATIONS){
	integerValue = value;
	if(integerValue > 5 && integerValue < 10)
	{
		labelPtr = &&test2bLabelTrue;
	}
	else
	{
		labelPtr = &&test2bLabelFalse;
	}
	i++;
}
end = clock();
double test6c = diffclock(end, start);
printf("Took %f seconds.\n", test6c);
double test6 = (test6a + test6b + test6c) / 3;
printf("Average time for Test 6 is %f seconds.\n", test6);



printf("Conducting Test Function Pointer 6a... ");
value = 4;
i = 0;
start = clock();
while(i < ITERATIONS){
	integerValue = value;
	if(integerValue > 5 && integerValue < 10)
	{
		funcPointer = &testFuncPointerTrue;
	}
	else
	{
		funcPointer = &testFuncPointerFalse;
	}
	i++;
}
end = clock();
```

```
double testFP6a = diffclock(end, start);
printf("Took %f seconds.\n", testFP6a);

printf("Conducting Test Function Pointer 6b... ");
value = 6;
i = 0;
start = clock();
while(i < ITERATIONS){
      integerValue = value;
      if(integerValue > 5 && integerValue < 10)
      {
            funcPointer = &testFuncPointerTrue;
      }
      else
      {
            funcPointer = &testFuncPointerFalse;
      }
      i++;
}
end = clock();
double testFP6b = diffclock(end, start);
printf("Took %f seconds.\n", testFP6b);

printf("Conducting Test Function Pointer 6c... ");
value = 11;
i = 0;
start = clock();
while(i < ITERATIONS){
      integerValue = value;
      if(integerValue > 5 && integerValue < 10)
      {
            funcPointer = &testFuncPointerTrue;
      }
      else
      {
            funcPointer = &testFuncPointerFalse;
      }
      i++;
}
end = clock();
double testFP6c = diffclock(end, start);
printf("Took %f seconds.\n", testFP6c);
double testFP6 = (testFP6a + testFP6b + testFP6c) / 3;
printf("Average time for Test Function Pointer 6 is %f seconds.\n",
testFP6);



printf("Conducting Test 7a... ");
bool1 = 1; bool2 = 0; bool3 = 0;
i = 0;
start = clock();
while(i < ITERATIONS){
      if(bool1)
      {
            foo++;
      }
      else if(bool2)
      {
            foo++;
      }
      else if(bool3)
```

```
        {
                foo++;
        }
        i++;
}
end = clock();
double test7a = diffclock(end, start);
printf("Took %f seconds.\n", test7a);

printf("Conducting Test 7b... ");
bool1 = 0; bool2 = 1; bool3 = 0;
i = 0;
start = clock();
while(i < ITERATIONS){
        if(bool1)
        {
                foo++;
        }
        else if(bool2)
        {
                foo++;
        }
        else if(bool3)
        {
                foo++;
        }
        i++;
}
end = clock();
double test7b = diffclock(end, start);
printf("Took %f seconds.\n", test7b);

printf("Conducting Test 7c... ");
bool1 = 0; bool2 = 0; bool3 = 1;
i = 0;
start = clock();
while(i < ITERATIONS){
        if(bool1)
        {
                foo++;
        }
        else if(bool2)
        {
                foo++;
        }
        else if(bool3)
        {
                foo++;
        }
        i++;
}
end = clock();
double test7c = diffclock(end, start);
printf("Took %f seconds.\n", test7c);

printf("Conducting Test 7d... ");
bool1 = 0; bool2 = 0; bool3 = 0;
i = 0;
start = clock();
while(i < ITERATIONS){
        if(bool1)
        {
```

```
            foo++;
      }
      else if(bool2)
      {
            foo++;
      }
      else if(bool3)
      {
            foo++;
      }
      i++;
}
end = clock();
double test7d = diffclock(end, start);
printf("Took %f seconds.\n", test7d);
double test7 = (test7a + test7b + test7c + test7d) / 4;
printf("Average time for Test 7 is %f seconds.\n", test7);



printf("Conducting Test 8a... ");
labelPtr = &&test8aLabel1;
i = 0;
start = clock();
while(i < ITERATIONS){
      goto *labelPtr;
test8aLabel1:
      {
            foo++;
      }
      goto test8aLabelFalse;
test8aLabel2:
      {
            foo++;
      }
      goto test8aLabelFalse;
test8aLabel3:
      {
            foo++;
      }
test8aLabelFalse:
      i++;
}
end = clock();
double test8a = diffclock(end, start);
printf("Took %f seconds.\n", test8a);

printf("Conducting Test 8b... ");
labelPtr = &&test8bLabel2;
i = 0;
start = clock();
while(i < ITERATIONS){
      goto *labelPtr;
test8bLabel1:
      {
            foo++;
      }
      goto test8bLabelFalse;
test8bLabel2:
      {
            foo++;
      }
```

```
	goto test8bLabelFalse;
test8bLabel3:
	{
		foo++;
	}
test8bLabelFalse:
	i++;
}
end = clock();
double test8b = diffclock(end, start);
printf("Took %f seconds.\n", test8b);

printf("Conducting Test 8c... ");
labelPtr = &&test8cLabel3;
i = 0;
start = clock();
while(i < ITERATIONS){
	goto *labelPtr;
test8cLabel1:
	{
		foo++;
	}
	goto test8cLabelFalse;
test8cLabel2:
	{
		foo++;
	}
	goto test8cLabelFalse;
test8cLabel3:
	{
		foo++;
	}
test8cLabelFalse:
	i++;
}
end = clock();
double test8c = diffclock(end, start);
printf("Took %f seconds.\n", test8c);

printf("Conducting Test 8d... ");
labelPtr = &&test8dLabelFalse;
i = 0;
start = clock();
while(i < ITERATIONS){
	goto *labelPtr;
test8dLabel1:
	{
		foo++;
	}
	goto test8dLabelFalse;
test8dLabel2:
	{
		foo++;
	}
	goto test8dLabelFalse;
test8dLabel3:
	{
		foo++;
	}
test8dLabelFalse:
	i++;
}
```

```
end = clock();
double test8d = diffclock(end, start);
printf("Took %f seconds.\n", test8d);
double test8 = (test8a + test8b + test8c + test8d) / 4;
printf("Average time for Test 8 is %f seconds.\n", test8);



printf("Conducting Test Function Pointer 8a... ");
funcPointer = &testFuncPointerTrue;
i = 0;
start = clock();
while(i < ITERATIONS){
      funcPointer(&foo);
      i++;
}
end = clock();
double testFP8a = diffclock(end, start);
printf("Took %f seconds.\n", testFP8a);

printf("Conducting Test Function Pointer 8b... ");
funcPointer = &testFuncPointerFalse;
i = 0;
start = clock();
while(i < ITERATIONS){
      funcPointer(&foo);
      i++;
}
end = clock();
double testFP8b = diffclock(end, start);
printf("Took %f seconds.\n", testFP8b);
double testFP8 = (testFP8a + testFP8b) / 2;
printf("Average time for Test Function Pointer 8 is %f seconds.\n",
testFP8);



printf("Conducting Test 9... ");
value1 = 0; value2 = 0; value3 = 0;
i = 0;
start = clock();
while(i < ITERATIONS){
      bool1 = value1;
      bool2 = value2;
      bool3 = value3;
      i++;
}
end = clock();
double test9 = diffclock(end, start);
printf("Took %f seconds.\n", test9);



printf("Conducting Test 10a... ");
value1 = 1; value2 = 0; value3 = 0;
i = 0;
start = clock();
while(i < ITERATIONS){
      bool1 = value1;
      bool2 = value2;
      bool3 = value3;
      if(bool1)
```

```
        {
                labelPtr = &&test8aLabel1;
        }
        else if(bool2)
        {
                labelPtr = &&test8aLabel2;
        }
        else if(bool3)
        {
                labelPtr = &&test8aLabel3;
        }
        else
        {
                labelPtr = &&test8aLabelFalse;
        }
        i++;
}
end = clock();
double test10a = diffclock(end, start);
printf("Took %f seconds.\n", test10a);

printf("Conducting Test 10b... ");
value1 = 0; value2 = 1; value3 = 0;
i = 0;
start = clock();
while(i < ITERATIONS){
        bool1 = value1;
        bool2 = value2;
        bool3 = value3;
        if(bool1)
        {
                labelPtr = &&test8aLabel1;
        }
        else if(bool2)
        {
                labelPtr = &&test8aLabel2;
        }
        else if(bool3)
        {
                labelPtr = &&test8aLabel3;
        }
        else
        {
                labelPtr = &&test8aLabelFalse;
        }
        i++;
}
end = clock();
double test10b = diffclock(end, start);
printf("Took %f seconds.\n", test10b);

printf("Conducting Test 10c... ");
value1 = 0; value2 = 0; value3 = 1;
i = 0;
start = clock();
while(i < ITERATIONS){
        bool1 = value1;
        bool2 = value2;
        bool3 = value3;
        if(bool1)
        {
                labelPtr = &&test8aLabel1;
```

```
        }
        else if(bool2)
        {
                labelPtr = &&test8aLabel2;
        }
        else if(bool3)
        {
                labelPtr = &&test8aLabel3;
        }
        else
        {
                labelPtr = &&test8aLabelFalse;
        }
        i++;
}
end = clock();
double test10c = diffclock(end, start);
printf("Took %f seconds.\n", test10c);

printf("Conducting Test 10d... ");
value1 = 0; value2 = 0; value3 = 0;
i = 0;
start = clock();
while(i < ITERATIONS){
        bool1 = value1;
        bool2 = value2;
        bool3 = value3;
        if(bool1)
        {
                labelPtr = &&test8aLabel1;
        }
        else if(bool2)
        {
                labelPtr = &&test8aLabel2;
        }
        else if(bool3)
        {
                labelPtr = &&test8aLabel3;
        }
        else
        {
                labelPtr = &&test8aLabelFalse;
        }
        i++;
}
end = clock();
double test10d = diffclock(end, start);
printf("Took %f seconds.\n", test10d);
double test10 = (test10a + test10b + test10c + test10d) / 4;
printf("Average time for Test 10 is %f seconds.\n", test10);



printf("Conducting Test Function Pointer 10a... ");
value1 = 1; value2 = 0; value3 = 0;
i = 0;
start = clock();
while(i < ITERATIONS){
        bool1 = value1;
        bool2 = value2;
        bool3 = value3;
        if(bool1)
```

```
        {
                funcPointer = &testFuncPointerTrue;
        }
        else if(bool2)
        {
                funcPointer = &testFuncPointerTrue;
        }
        else if(bool3)
        {
                funcPointer = &testFuncPointerTrue;
        }
        else
        {
                funcPointer = &testFuncPointerFalse;
        }
        i++;
}
end = clock();
double testFP10a = diffclock(end, start);
printf("Took %f seconds.\n", testFP10a);

printf("Conducting Test Function Pointer 10b... ");
value1 = 0; value2 = 1; value3 = 0;
i = 0;
start = clock();
while(i < ITERATIONS){
        bool1 = value1;
        bool2 = value2;
        bool3 = value3;
        if(bool1)
        {
                funcPointer = &testFuncPointerTrue;
        }
        else if(bool2)
        {
                funcPointer = &testFuncPointerTrue;
        }
        else if(bool3)
        {
                funcPointer = &testFuncPointerTrue;
        }
        else
        {
                funcPointer = &testFuncPointerFalse;
        }
        i++;
}
end = clock();
double testFP10b = diffclock(end, start);
printf("Took %f seconds.\n", testFP10b);

printf("Conducting Test Function Pointer 10c... ");
value1 = 0; value2 = 0; value3 = 1;
i = 0;
start = clock();
while(i < ITERATIONS){
        bool1 = value1;
        bool2 = value2;
        bool3 = value3;
        if(bool1)
        {
                funcPointer = &testFuncPointerTrue;
```

```
        }
        else if(bool2)
        {
                funcPointer = &testFuncPointerTrue;
        }
        else if(bool3)
        {
                funcPointer = &testFuncPointerTrue;
        }
        else
        {
                funcPointer = &testFuncPointerFalse;
        }
        i++;
}
end = clock();
double testFP10c = diffclock(end, start);
printf("Took %f seconds.\n", testFP10c);

printf("Conducting Test Function Pointer 10d... ");
value1 = 0; value2 = 0; value3 = 0;
i = 0;
start = clock();
while(i < ITERATIONS){
        bool1 = value1;
        bool2 = value2;
        bool3 = value3;
        if(bool1)
        {
                funcPointer = &testFuncPointerTrue;
        }
        else if(bool2)
        {
                funcPointer = &testFuncPointerTrue;
        }
        else if(bool3)
        {
                funcPointer = &testFuncPointerTrue;
        }
        else
        {
                funcPointer = &testFuncPointerFalse;
        }
        i++;
}
end = clock();
double testFP10d = diffclock(end, start);
printf("Took %f seconds.\n", testFP10d);
double testFP10 = (testFP10a + testFP10b + testFP10c + testFP10d) /
4;
printf("Average time for Test Function Pointer 10 is %f seconds.\n",
testFP10);



return 0;
}
```