A3: Karakterisering og optimering af køretid for sorteringsalgoritmer

Computer Systems 2018
Department of Computer Science
University of Copenhagen

Finn Schiermer Andersen

Due: Søndag, 28. Oktober, 10:00
Version 1.1 (Teoretiske spørgsmål)

Dette er den fjerde afleveringsopgave i Computersystemer og den anden (og sidste) som dækker Maskinarkitektur. Som tidligere, opfordrer vi til parprogrammering og anbefaler derfor at lave grupper at 2 til 3 studerende. Grupper må ikke være større end 3 studerende og vi advarer mod at arbejde alene.

Afleveringer vil blive bedømt med op til 4 point. Du skal opnå mindst halvdelen af de mulige point over kurset, samt mindst 3 point i hvert emne, for at blive indstillet til eksamen. Denne aflevering hører under Maskinarkitektur. Du kan finde yderligere detaljer under siden "Course description" på Absalon. Det er *ikke* muligt at genaflevere afleveringer.

Introduction

Constant time factors do matter
— Neil D. Jones, *in ACM Symposium on Theory of Computing* (1993)

Som dataloger er vi vant til at kigge på asymptotisk tidskompleksitet. Vi ved f.eks. at en række sorteringsalgoritmer er $O(n \log(n))$. Konstant faktorer er noget vi normalt ser bort fra.

A3 handler primært om at karakterisere opførsel af nogle programmer på 3 forskellige klasser af maskiner samt at udvikle et program med samme asymptotiske kompleksitet men hurtigere køretid.

Derudover skal I lave en teoretisk simulation af programmers opførsel på en givet cache, samt skrivebords-simulation af et lille program på en simpel pipeline.

1 Karakterisering og optimering af køretid

I denne del skal I først karakterisere opførselen af nogle programmer på 3 forskellige klasser af maskiner. Dernæst skal I udvikle et program med samme asymptotiske kompleksitet, men kortere køretid.

De programmer vi skal karakterisere er:

- Heapsort,

- Quicksort, og
- en selv-udviklet implementation af en sorterings-algoritme.

Mere detaljeret bliver opgaverne at:

- I skal karakterisere de 3 programmer - både de to udleverede programmer og det selv-udviklede. Programmerne skal køres på tre forskellige maskiner, to forskellige lager-hierakier og for forskellige størrelser af inddata. Det beskrives i detaljer nedenfor.
- I skal selv skrive en alternativ sorteringsrutine og sammenligne dens køretid med de udleverede programmer. Denne rutine skal kunne afvikles hurtigere end de udleverede (omend den kan have samme asymptotiske kompleksitet).

I forhold til A2 er vor maskine udvidet med:

- venstre- og højreskift (left- and right-shift) instruktioner; I kan bruge << og >> i jeres C programmer,
- instruktioner til at indlæse og udlæse tal; via `read_long\0` og `write_long\1` functionerne i C, samt
- instruktion til at generere tilfældige tal; via `gen_random\0`.

Dette er ikke noget I skal implementere, men er stillet til rådighed når I implementerer programmer i C eller assembler programmer i x86prime.

1.1 Udleveret materiale

Vi udleverer C-kildetekst til en implementation af Heapsort og Quicksort med en rutine der på køretid indlæser hvorvidt der skal printes og hvor mange elementer man vil have sorteret. Dertil er der også en fil der skal udvides med den selv-udviklede sorterings-rutine.

Programmet indlæser (mindst) to heltal fra tastaturet. Tallene fastlægger:

- valg af sorterings-rutine og/eller test-mode versus performance-mode
- størrelse af datasæt. I performance-mode genereres en tabel af pseudo-tilfældige tal som så sorteres
- I test-mode indlæses endvidere en række tal som sorteres

Test-mode kan bruges til at verificere at ens sortering virker korrekt. Test-mode skal være slået fra, når der laves målinger.

Alle programmer findes i vedlagte fil `src.zip`. Denne indeholder også en `README` med nærmere beskrivelser af indholdet, læs denne. Den indeholder også hjælpe script til performance og funktionel tests.

Simulering og modellering af maskiner gøres med x86prime som kan hentes fra GitHub repositoriet:

`https://github.com/finnschiermer/x86prime`.

Som tidligere kan den virtuelle maskine som er stillet til rådighed benyttes, hvor både x86prime og gcc er forud installeret; kør `update_x86prime.sh` for at få seneste version. Da enkelte maskiner har haft problemer med VirtualBox, vil vi igen stille en remote server til rådighed som kan køre gcc og x86prime. Dette annonceres separat.

1.2 Karakterisering af sorteringsprogrammer

Karakterisering af et programs opførsel gøres ved hjælp af en ny version af x86prime. Denne udgave kan modellere forsinkelser i afviklingen af et program som kan skyldes:

- forsinkelser i lager-hierarkiet (f.eks. cache miss),
- forsinkelser på grund af hop, kald, retur (branch mispredicts, return mispredicts), og
- forsinkelser på grund af dataafhængigheder mellem instruktioner.

Karakterisering skal bestå i følgende dele:

- etablering af en baseline,
- bestem forsinkelser i lagerhierakiet, og
- bestem køretid for 3 forskellige maskiner af varierende kompleksitet.

Til karakterisering skal bruges følgende 3 forskellige maskiner:

- en 9-trins standard pipeline som præsenteret til forelæsningen 3. oktober,
- en 3-vejs superscalar, og
- en 3-vejs out-of-order som præsenteret til forelæsningen 10. oktober.

Dertil skal to forskellige lagerhierakier undersøges:

- et "magisk" lagerhieraki hvor al adgang til lageret tager samme tid (3 cykler) og
- et "realistisk" lagerhieraki med to niveauer af cache.

Til alle målingerne beskrevet nedenfor skal bruges inddatasæt med valgte størrelser. Hvor store data størrelser vil vi lade være op til jer. I skal sikre jer at de valgte størrelser kommer til at vise ønskede effekter i cachen samt at antallet er målepunkter er stort nok til at jeres plots beskriver opførslen.

Vær opmærksom på at kørsler med de største inddatasæt kan tage længere tid. Når I udføre målingerne bør I derfor automatiserer jeres kørsler, så hver af kørslerne ikke udføres enkeltvis; brug evt. vedlagte script

1.2.1 Etablering af baseline

Formålet med opgaven er at få en basis for vurdering af hvordan programmerne interagerer med et realistisk lager-hierarki, men før vi har den skal vi fastlægge en baseline for programmets asymptotiske køretid.

Programmerne skal simuleres på den simple 9-trins pipeline, men L2-cache og hovedlageret skal konfigureres til at have 0 cyklers overhead, dvs man får en maskine der kan tilgå hele lageret med samme hastighed som hvis alting kunne være i L1-cachen. (Options: "`-pipe simple -mem magic`".)

Hvert program skal simuleres med inddatasæt af de ovenfor anførte størrelser. Køretiden skal præsenteres i to grafer: den ene skal angive absolut køretid, den anden køretid per element.

Graferne skulle gerne underbygge at alle programmerne har $O(n \log n)$ asymptotisk kompleksitet.

1.2.2 Forsinkelser i lagerhierarkiet

Formålet med denne del er at vurdere hvor meget lagerhierarkiet betyder for de 3 programmer. Programmerne skal simuleres på den simple 9-trins pipeline, men med et mere realistisk lagerhierarki:

- 16Kb L1 cache, 3 cyklers adgang
- 128Kb L2 cache, 15 cyklers adgang
- Hovedlager, 115 cyklers adgang

Dette vælges med option "`-pipe simple -mem real`"

Cache miss i hhv. primær cache og sekundær cache for de 4 programmer skal afbildes i en graf. Derpå skal AMAT, "average memory access time", beregnes og præsenteres.

Overvej (og rapporter) hvorvidt der ud fra målingerne er muligt at sige noget om hvilke af programmerne som har bedst rummelig lokalitet og dermed bedst kan udnytte caching?

1.2.3 Realistisk ydeevne

Formålet med sidste del er her ikke så meget at vurdere programmerne, som at illustrere hvordan forskellige mikroarkitekturer kan "skjule" effekten af lange latens-tider, f.eks. cache-miss, ved at finde andre instruktioner at udføre i stedet for at stalle.

Programmerne skal simuleres på alle tre mikroarkitekturer:

- simpel 9-trins pipeline "`-pipe simple -mem real`",
- 3-vejs superscalar "`-pipe super -mem real`" og
- 3-vejs out-of-order "`-pipe ooo -mem real`".

Resultaterne køretiderne skal præsenteres i en graf.

Overvej (og rapporter) hvad køretiderne siger om de tre forskellige mikroarkitekturer.

1.3 Udvikling af en hurtigere sortering

I sidste del af denne opgave skal I forsøge at skrive et sorteringsprogram som kan sortere de samme tilfældige tal som de udleverede programmer, men bare hurtigere. Målet er at få et program som er i størrelsesordenen 10 procent hurtigere end de udleverede programmer, når det køres på det største inddatasæt.

I kan frit vælge mellem at tage udgangspunkt i en af de udleverede algoritmer og optimere denne, eller at implementere en anden algoritme. I *skal* dog implementere en algoritme med samme asymptotiske kompleksitet som de udleverede.

I kan også frit vælge mellem at optimere i C koden og at optimere direkte i x86prime assembler, eller eventuelt begge dele.

I rapporten skal i forklarer hvilke stridt eller metoder I har brugt til at komme frem til jeres optimerede sorteringsalgoritme og forklarer hvordan disse tiltag har indflydelse på cache opførslen.

Jeres udviklede algoritme skal performance testes på samme måde som de udleverede programmer og i skal i rapporten sammenligne resultaterne med disse. Husk også at lave en funktionel test af jeres algoritme.

Til hjælp med at lave en funktionel test udleverer vi et script der kører simulatoren og tjekker at uddata fra kørslen matcher en reference kørsel med quicksort algoritmen.

2 Teoretiske opgaver

2.1 Simulering af cache

En maskine er byte-addresseret med 32-bit lange addresser. Maskinen er udstyret med en 2-vejs associativ datacache på 32 kilobyte. Cachen har en blokstørrelse på 16 bytes.

Spørgsmål 1 Angiv opsplitningen af addressen ved tilgang til cachen. Hvilke bits bruges til at indexere indenfor en blok, hvilke angiver sæt-index og hvilke udgør tag?

Spørgsmål 2 Betragt følgende strøm af lager-referencer på hexadecimal form:

> `0x0`, `0xF00000`, `0x0`, `0xF0000C`, `0xC00004`, `0xE00008`, `0xF00004`, `0xC00000`, `0x8`, `0x10`, `0x4`.

Antag at cachen i begyndelsen er "kold" og angiv for hver af referencerne om den kan findes i cachen (hit) eller ej (miss).

Spørgsmål 3 Maskinen har også en større sekundær cache. Den er på 256 Kilobyte, 4-vejs associativ og har en blok-størrelse på 32 bytes.

Betragt følgende strøm af lager-referencer på hexadecimal form (det er ikke en fejl at strømmen gentager sig selv):

> `0x0`, `0xF00000`, `0x0`, `0xF0000C`, `0xC00004`, `0xE00008`, `0xF00004`, `0xC00000`, `0x8`, `0x10`, `0x4`, `0x`, `0x0`, `0xF00000`, `0x0`, `0xF0000C`, `0xC00004`, `0xE00008`, `0xF00004`, `0xC00000`, `0x8`, `0x10`, `0x4`.

Antag at den sekundære cache initielt er "kold" og angiv for hver af referencerne om den kan findes i den sekundære cache (hit) eller ej (miss).

Spørgsmål 4 Antag at miss-rate for den primære cache er 20% og miss-penalty er 10 clock. Antag at miss-rate for den sekundære cache er 30% og miss-penalty er 50 clock.

Hvad er AMAT (average memory access time)?

2.2 Program simulation på pipeline maskine

Nedenfor ses afviklingen af en stump x86prime kode på en simple pipeline. Den viste kode er den indre løkke i en funktion som multiplicerer alle elementer i en tabel med et argument. Ved indgang til løkken indeholder `%r10` pointeren til tabellen, `%r12` det tal alle elementer skal multipliceres med og `%r8` antallet af elementer i tabellen.

```
1  Code                    Timing
2
3  Loop:
4    cbl  $0,%r8,Done     FDXMYW
5    movq (%r10),%r11      FDXMYW
6    multq %r12,%r11        FDDXMYW
7    movq %r11,(%r10)        FFDDDXMYW
8    addq $8,%r10              FFFDXMYW
9    subq $1,%r8                   FDXMYW
10   jmp Loop                       FDXMYW
11 Done:
```

Bogstaverne til højre viser hver instruktions passage gennem pipelinen. Betydningen af bogstaverne er:

F: Fetch, instruktionhentning

D: Decode, afkodning, læsning af registre, evt venten på operander

X: eXecute, udførelse af ALU op, af adresseberegning eller af første del af multiplikation

M: Memory, læsning/skrivning af data fra data-cache, midterste del af multiplikation

Y: sidste del af multiplikation

W: Writeback, opdatering af registre

Alle instruktioner passerer gennem de samme 6 trin. Multiplikation udføres over 3 pipeline-trin, E, M og Y. Et ubetinget hop udføres i D-trinnet, dvs den instruktion der hoppes til kan blive hentet i cyklussen efter. Et betinget hop udføres derimod først i X-trinnet.

Der er fuld forwarding af operander fra en instruktion til en afhængig instruktion. Instruktioner venter i D-trinnet indtil operander er tilgængelige.

Spørgsmål 1: Vi indfører en særlig undtagelse for movq instruktioner som skriver til lageret. De skal stadig vente i D på operander til adresseberegning, men skal først vente i X på selve den værdi der skal skrives til lageret.

Gentegn figuren ovenfor under hensyntagen til denne ændring.

Spørgsmål 2: Hvor mange clock-cyklusser tager hvert gennemløb af løkken. Udvid eventuelt figuren med instruktioner fra efterfølgende gennemløb, indtil du er sikker på dit svar.

Spørgsmål 3: Kan du optimere koden ved at flytte rundt på instruktionerne således at der er færre instruktioner der skal vente på vej gennem pipelinen?

Vis den optimerede kode og tegn en tilsvarende figur der viser hvordan den udføres i pipelinen. Hvor meget hurtigere er din optimerede kode?

3 Bedømmelse og rapportering

De forskellige dele af afleveringen bliver vurderet omtrent efter følgende:

- 20% performance testing af eksisterende algoritmer,
- 20% udvikling og test af optimerede algoritme,
- 20% løsning af teoretiske opgaver,
- 40% rapport som dokumenterer jeres implementation.

Rapporten skal berøre problemstillinger som er rejst under opgaven, inklusiv de overvejelser som opgaveteksten lægger op til. Beskriv alle ikke-trivielle dele af jeres løsning, samt evt. tvetydige formuleringer, som I kan have fundet i opgaveteksten. Det er forventeligt at afrapportering til karakterisering og optimering af køretid er omkring 5 sider (eksklusiv grafer) og den må ikke overskrive 7 sider. Dertil kommer de teoretiske opgaver.

For at få point skal man kunne dokumentere at opgaven er løst korrekt. Det gøres ved at udarbejde og køre testprogrammer og/eller scripts, som kan bekræfte at I kan udføre alle simulationer.

Sammen med jeres rapport, `report.pdf`, skal I aflevere en `src.zip` som indeholder alle relevante udviklede programmer, scripts og test programmer. Denne skal også indeholde de data som er genereret fra kørslerne og afbildet i graferne i rapporten. Følg den struktur som er i den udleverede zip-fil.

Dertil skal der afleveres `group.txt` som indeholder en ASCII/UTF8 formateret liste KU-id'er fra alle medlemmer i gruppen; et id pr. line, ved brug af følgende tegnsæt:

$$\{\texttt{0x0A}\} \cup \{\texttt{0x30}, \texttt{0x31}, \ldots, \texttt{0x39}\} \cup \{\texttt{0x61}, \texttt{0x62}, \ldots, \texttt{0x7A}\}$$