A6: Maskinnær Optimering

Jeg har implementeret en prioritets-kø i C. Opgaven er at få den til at køre hurtigere!

En prioritetskø understøtter to operationer:

```
Add(value); // tilføj 'value' til prioritetskøen
Take();     // udskriv og fjern mindste element i prioritetskøen
Take(N);    // som N * take()
```

Vi udleverer:

- En Riscv simulator, forbedret
- Implementation af prioritetskøen i C, inklusiv optimeret I/O så du ikke skal tænke på det
- En inddata generator - som også kan generere tilsvarende uddata til tjek

Besvarelse

Besvarelsen skal indeholde:

- En kort rapport. Rapporten skal indeholde:
  - En kort præsentation af hvordan man har løst opgaven og hvorfor
  - En analyse af om løsningen også vil være hurtigere for større datasæt end de, der praktisk kan afvikles indenfor rammerne af opgaven.
- En hurtigere implementation end den udleverede
- Dokumentation for at denne implementation virker
- Dokumentation for at denne implementation er hurtigere (se nedenfor).

Den udleverede implementation

Er baseret på et radix-træ. Kodet uden særlig opmærksomhed på ydeevne.

Indsættelse og udtagning i rækkefølge har log(B) asymptotisk kompleksitet, hvor B er antallet af elementer i prioritetskøen. Det skulle gerne være (teoretisk) optimalt.

Men konstant faktorer har også betydning.

En forbedret simulator

Der er to store forbedringer i vores riscv simulator:

- En timing model. Den beregner hvor mange clock-cycles en programkørsel tager
- Timing modellen er simpel.
    - In-order pipeline med realistisk cache tilgang.
    - 2 niveauer af Cache.
    - Yderligere detaljer i opgaveformuleringen.
- Muligheden for at få en eksekveringsprofil
    - Antal gange en instruktion er udført
    - Antal pipeline stalls.
    - Latenstid
    - Lagertilgang, hit-fordeling på L1, L2 og lager.

```
00010074 <fib>:
10074 :        177    0   1       0   0   0    addi     sp,sp,-16
10078 :        177    0   1      98   0   1    sw       ra,12(sp)
1007c :        177    0   1     100   0   0    sw       s0,8(sp)
10080 :        177    0   1     100   0   0    sw       s1,4(sp)
10084 :        177    0   1       0   0   0    mv       s0,a0
10088 :        177    0   1       0   0   0    li       a5,1
1008c :        177    0   1       0   0   0    bgeu     a5,a0,100b0      <fib+0x3c>
10090 :         88    0   1       0   0   0    addi     a0,a0,-1
10094 :         88    0   1       0   0   0    auipc    ra,0x0
10098 :         88    0   1       0   0   0    jalr     -32(ra)          # 10074 <fib>
1009c :         88    0   1       0   0   0    mv       s1,a0
100a0 :         88    0   1       0   0   0    addi     a0,s0,-2
100a4 :         88    0   1       0   0   0    auipc    ra,0x0
100a8 :         88    0   1       0   0   0    jalr     -48(ra)          # 10074 <fib>
100ac :         88    0   1       0   0   0    add      a0,s1,a0
100b0 :        177    0   4     100   0   0    lw       ra,12(sp)
100b4 :        177    0   4     100   0   0    lw       s0,8(sp)
100b8 :        177    0   4     100   0   0    lw       s1,4(sp)
100bc :        177    0   1       0   0   0    addi     sp,sp,16
100c0 :        177    0   1       0   0   0    ret
 |               |    |   |       |   |   |
 |               |    |   |       |   |   pct tilgang til lageret
 |               |    |   |       |   pct hit i L2 cache
 |               |    |   |       pct hit i L1 cache
 |               |    |   latenstid (tid fra Ex til resultat tilgængeligt)
 |               |    ventetid (clocks hvor pipelinen stall'er indtil input data er tilgænge
 |               antal gange den pågældende instruktion er udført
 Addresse på instruktionen
```

Anbefalet fremgangsmåde (I)

Forstå det udleverede program. Det kan være svært. Forslag:

- Opstil et helt lille eksempel på inddata og forventet uddata.
- Gennemløb C programmet på papir med dit minimale inddata.
- Tegn data strukturen undervejs
- Indsæt nogle udskrivninger som kan vise, hvilke valg programmet træffer
- Kør programmet med dit minimale inddata. Hold programmets valg op i mod dine forventninger
- Iterer indtil det giver mening :-)

Når du har dannet et overblik over hvordan programmet virker, kan du forsøge at optimere det. Formodentlig har selve arbejdet med at forstå programmet affødt spørgsmål du kan bruge. Endvidere kan du bruge simulatoren til at lave en eksekveringsprofil. Den kan give en ide om, hvor det kan betale sig at optimere.

Anbefalet fremgangsmåde (II)

Det er også en mulighed at lave sin egen løsning fra scratch.

Der er inspiration til hvordan prioritetskøer kan implementeres her: https://en.wikipedia.org/wiki/Priority_queue

Men bemærk at vi alene bedømmer løsningen ud fra, hvordan den interagerer med datasættene fra ./generate med de angivne parametre. Det er kun "put" og "take" der er vigtige operationer her. I en virkelig prioritetskø kan behovet for andre operationer være bestemmende for valget af datastruktur.

Det kan være tillokkende bare at starte forfra, hvis det synes svært at forstå den udleverede kode. Men det er også svært at nå i mål med. Fejlfinding er vanskelig i vores simulerede miljø.

Gode råd:

Det her er en opgave man bør begynde på tidligt.

Overvej nøje hvordan den valgte datastruktur interagerer med caching.

Det vil sandsynligvis være nødvendigt at ændre datastrukturen for at få en markant forbedring af køretiden.

Spørgsmål og Svar