Practicum 2: Optimaliseren

Computerarchitectuur

Groep 11
Mitch De Wilde & Bart Middag
2de bachelor informatica
Academiejaar 2012-2013

1. VERPLICHTE OPGAVEN

1.1 Bepalen van klokcycli en overheadcycli

Volgens het outputbestand dat door het script `timing.sh` met parameter 9 gegenereerd wordt, is het aantal cycli van de functie zelf ongeveer gelijk aan 603954. Dit is immers het verschil van het aantal uitgevoerde instructies (1973315) – het aantal geheugentoegangen (1369361).

Geheugenoperaties hebben een latentie van ongeveer 153 klokcycli, dus de overhead is het aantal geheugentoegangen maal 153, m.a.w. 1369361*153 = 209512233.

1.2 Optimaliseren van de stapel

Per call naar de functie `lucas` hebben we minstens 20 bytes nodig:

- 4 bytes om de returnwaarde van de eerste call naar `lucas` in `.L2` bij te houden
- 4 bytes om de returnwaarde van de tweede call naar `lucas` in `.L2` bij te houden
- 4 bytes voor de originele %ebp op de stack te zetten
- 4 bytes voor de returnwaarde van de functie
- 4 bytes voor het returnadres van de call zelf

Per stap in de recursie moeten we hiervan handmatig 16 vrijmaken.

Door de stack optimaal te gebruiken, wordt de gebruikte geheugenruimte wel kleiner, maar het aantal klokcycli verandert niet. Dit is logisch: hetzelfde aantal instructies wordt nog steeds uitgevoerd.

ORIGINELE CODE

```
lucas:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $40, %esp
    movl    $0, -12(%ebp)
    cmpl    $1, 8(%ebp)
    jg  .L2
    movl    $2, -12(%ebp)
    jmp .L3
.L2:
    movl    8(%ebp), %eax
    subl    $1, %eax
    movl    %eax, (%esp)
    call    lucas
    movl    %eax, -16(%ebp)
    movl    8(%ebp), %eax
    subl    $2, %eax
    movl    %eax, (%esp)
    call    lucas
    movl    %eax, -20(%ebp)
    movl    -20(%ebp), %eax
    addl    %eax, %eax
    addl    -16(%ebp), %eax
    movl    %eax, -12(%ebp)
.L3:
```

OPLOSSING VRAAG 2

```
lucas:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    movl    $0, -4(%ebp)
    cmpl    $1, 8(%ebp)
    jg  .L2
    movl    $2, -4(%ebp)
    jmp .L3
.L2:
    movl    8(%ebp), %eax
    subl    $1, %eax
    movl    %eax, (%esp)
    call    lucas
    movl    %eax, -8(%ebp)
    movl    8(%ebp), %eax
    subl    $2, %eax
    movl    %eax, (%esp)
    call    lucas
    movl    %eax, -12(%ebp)
    movl    -12(%ebp), %eax
    addl    %eax, %eax
    addl    -8(%ebp), %eax
    movl    %eax, -4(%ebp)
.L3:
```

```
26      movl    -12(%ebp), %eax
27      leave
28      ret
```

```
26      movl    -4(%ebp), %eax
27      leave
28      ret
```

1.3 Optimalisatie van de heetste code

De heetste code in de lucas-functie is `lucas` en `.L3`.
Bij de instructie `cmpl $1, 8(%ebp)` in `lucas` zal 8(%ebp) dus meer dan de helft van de keren kleiner dan of gelijk aan 1 zijn. Dit is ook logisch, want als we naar de recursieboom van de functie kijken, zien we dat het een binaire boom is – en in een binaire boom zijn er altijd meer bladeren dan interne knopen.

Na het optimaliseren van de heetste code is de uitvoeringstijd gegenereerd door `./timing.sh 9` van rond de 211 miljoen cycli in totaal gezakt naar 135 miljoen cycli. Dit is dus al bijna een halvering.

De grootste optimalisaties komen van het verplaatsen van de push/pop-instructies naar `.L2` (hierbij moest natuurlijk rekening gehouden worden met de offsets), het weglaten van de leave-instructie die dan overbodig wordt, en het gebruik van %eax in plaats van %ebp en %esp. Hierbij hebben we nog niets veranderd, enkel toegevoegd, aan `.L2`.

```
        OPLOSSING VRAAG 2
1   lucas:
2       pushl   %ebp
3       movl    %esp, %ebp
4       subl    $16, %esp
5       movl    $0, -4(%ebp)
6       cmpl    $1, 8(%ebp)
7       jg  .L2
8       movl    $2, -4(%ebp)
9       jmp .L3
10  .L2:
11      movl    8(%ebp), %eax
12      subl    $1, %eax
13      movl    %eax, (%esp)
14      call    lucas
15      movl    %eax, -8(%ebp)
16      movl    8(%ebp), %eax
17      subl    $2, %eax
18      movl    %eax, (%esp)
19      call    lucas
20      movl    %eax, -12(%ebp)
21      movl    -12(%ebp), %eax
22      addl    %eax, %eax
23      addl    -8(%ebp), %eax
24      movl    %eax, -4(%ebp)
25  .L3:
26      movl    -4(%ebp), %eax
27      leave
28      ret
```

```
        OPLOSSING VRAAG 3
lucas:
    cmpl    $1, %eax
    jg  .L2
    movl    $2, %eax
    ret
.L2:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    #originele code van L2
    movl    8(%ebp), %eax
    subl    $1, %eax
    movl    %eax, (%esp)
    call    lucas
    movl    %eax, -8(%ebp)
    movl    8(%ebp), %eax
    subl    $2, %eax
    movl    %eax, (%esp)
    call    lucas
    movl    %eax, -12(%ebp)
    movl    -12(%ebp), %eax
    addl    %eax, %eax
    addl    -8(%ebp), %eax
    movl    %eax, -4(%ebp)
    #het volgende = leave
    movl    %ebp, %esp
    popl    %ebp
    ret
```

1.4 Algemene optimalisatie

De hele functie kan echter nog veel meer geoptimaliseerd worden. Zo hebben we een uitvoeringstijd van maar liefst 34 miljoen cycli bereikt in plaats van de 135 miljoen cycli die we bekomen waren met de vorige vraag.

Bij de eerste optimalisatiestap hebben we de push/pop-bewerkingen in `.L2` weggewerkt, door het gebruik van %esp in plaats van %ebp. Waarom zouden we immers %ebp gebruiken als we gewoon %esp kunnen gebruiken? Zonder push en pop werkt de functie veel sneller: dit zorgt voor een versnelling van 135 miljoen cycli naar 94 miljoen cycli.

OPLOSSING VRAAG 3

```
lucas:
    cmpl    $1, %eax
    jg  .L2
    movl    $2, %eax
    ret
.L2:
    pushl   %ebp
    movl    %esp, %ebp
    subl    $16, %esp
    #originele code van L2
    movl    8(%ebp), %eax
    subl    $1, %eax
    movl    %eax, (%esp)
    call    lucas
    movl    %eax, -8(%ebp)
    movl    8(%ebp), %eax
    subl    $2, %eax
    movl    %eax, (%esp)
    call    lucas
    movl    %eax, -12(%ebp)
    movl    -12(%ebp), %eax
    addl    %eax, %eax
    addl    -8(%ebp), %eax
    movl    %eax, -4(%ebp)
    #het volgende = leave
    movl    %ebp, %esp
    popl    %ebp
    ret
```

EERSTE OPTIMALISATIESTAP

```
lucas:
    cmpl    $1, %eax
    jg  .L2
    movl    $2, %eax
    ret
.L2:
    #push overbodig
    #na verwijderen van ebp
    subl    $20, %esp

    movl    24(%esp), %eax
    subl    $1, %eax
    movl    %eax, (%esp)
    call    lucas
    movl    %eax, 8(%esp)
    movl    24(%esp), %eax
    subl    $2, %eax
    movl    %eax, (%esp)
    call    lucas
    #dit wordt nergens gebruikt
    #dit is al gebeurd
    addl    %eax, %eax
    addl    8(%esp), %eax
    #instructie overbodig

    #pop overbodig
    addl    $20, %esp
    ret
```

Bij de tweede optimalisatiestap hebben we de heetste code van het algoritme nog geoptimaliseerd door telkens de instructie `movl %eax, (%esp)` te verplaatsen van voor de call naar `lucas`, naar het begin van `.L2` – dus een niveau lager in de recursie. In de functie `lucas` zelf wordt (%esp) immers niet gebruikt. Ook hebben we een addl vervangen door shl omdat dit sneller lijkt. Deze optimalisatie zorgt voor een versnelling van 94 miljoen cycli naar 77 miljoen cycli.

EERSTE OPTIMALISATIESTAP

```
lucas:
    cmpl    $1, %eax
    jg      .L2
    movl    $2, %eax
    ret
.L2:

    subl    $20, %esp
    movl    24(%esp), %eax
    subl    $1, %eax
    movl    %eax, (%esp)
    call    lucas
    movl    %eax, 8(%esp)
    movl    24(%esp), %eax
    subl    $2, %eax
    movl    %eax, (%esp)
    call    lucas
    addl    %eax, %eax
    addl    8(%esp), %eax
    addl    $20, %esp
    ret
```

TWEEDE OPTIMALISATIESTAP

```
lucas:
    cmpl    $1, %eax
    jg      .L2
    movl    $2, %eax
    ret
.L2:
    movl    %eax, 4(%esp)
    subl    $20, %esp
    #staat al in %eax
    subl    $1, %eax
    #verplaatst naar begin .L2
    call    lucas
    movl    %eax, 8(%esp)
    movl    24(%esp), %eax
    subl    $2, %eax
    #verplaatst naar begin .L2
    call    lucas
    shl     %eax
    addl    8(%esp), %eax
    addl    $20, %esp
    ret
```

Bij de derde optimalisatiestap merkten we dat de stapel niet meer optimaal gebruikt werd en hebben we deze weer geoptimaliseerd. Ook gebruikten we een klein trucje om in `.L2` de 2de call naar `lucas` te kunnen overslaan. Als de eerste call naar `lucas` 2 teruggeeft, weten we immers zeker dat de 2de call ook 2 zal teruggeven en kunnen we direct het resultaat berekenen door het te vermenigvuldigen met 3 (2+2*2) aan de hand van de instructie leal (load effective address). Dit is een trucje om te vermenigvuldigen en op te tellen in 1 instructie. We zouden natuurlijk ook `movl $6, %eax` kunnen schrijven (we weten immers dat %eax hier altijd 2 zal zijn en de oplossing dus altijd 6), en dat zou sneller zijn, maar met het oog op toekomstige optimalisaties laten we leal toch even staan…
Deze optimalisatie zorgt voor een versnelling van 77 miljoen naar 61 miljoen cycli.

TWEEDE OPTIMALISATIESTAP

```
lucas:
    cmpl    $1, %eax
    jg      .L2
    movl    $2, %eax
    ret
.L2:
    movl    %eax, 4(%esp)
    subl    $20, %esp
    subl    $1, %eax
    call    lucas


    movl    %eax, 8(%esp)
    movl    24(%esp), %eax
    subl    $2, %eax
    call    lucas
    shl     %eax
    addl    8(%esp), %eax
    addl    $20, %esp
```

DERDE OPTIMALISATIESTAP

```
lucas:
    cmpl    $1, %eax
    jg      .L2
    movl    $2, %eax
    ret
.L2:
    movl    %eax, 4(%esp)
    subl    $8, %esp
    subl    $1, %eax
    call    lucas
    cmpl    $2, %eax
    jle     .L3
    movl    %eax, 4(%esp)
    movl    12(%esp), %eax
    subl    $2, %eax
    call    lucas
    shl     %eax
    addl    4(%esp), %eax
    addl    $8, %esp
```

```
20      ret                                     ret
21                                      .L3:
22                                          leal    (%eax,%eax,2), %eax
23                                          addl    $8, %esp
24                                          ret
```

We kunnen deze optimalisatie nog verder uitwerken en de controle in plaats van na de eerste call naar `lucas` ervoor plaatsen en zorgen dat er meer kans is dat er naar het alternatieve pad gesprongen wordt. We voegen dan nog een extra pad toe, `.L4`, waarnaar gesprongen kan worden voor de tweede call naar `lucas`. Als we met meer mogelijkheden rekening houden, wordt het natuurlijk moeilijk om de instructies in de extra paden `.L3` en `.L4` kort te houden. Hier komen een aantal berekeningen aan te pas.

Op het moment van de eerste call naar `lucas` in `.L2` is %eax al verminderd met 1. Als we de vergelijking hierachter zetten, verliezen we tijd (de subl-instructie wordt altijd uitgevoerd), maar is de berekening eenvoudiger.
We willen dit pad kiezen als %eax na vermindering 1 of 2 is. Het resultaat van de call zal dan 2 of 6 zijn. Wetende dat de %eax bij de tweede call naar `lucas` 1 minder zal zijn (dus 0 of 1) en dan altijd 2 zal teruggeven, zal het eindresultaat van `.L2` dus 2+2*2=6 of 6+2*2=10 zijn. Wat is dan het verband tussen 1 en 6 en het verband tussen 2 en 10? Dit is een simpele vermenigvuldiging met 4 en optelling met 2. We kunnen dit in 1 instructie doen, namelijk `leal 2(,%eax,4), %eax`.
Als we nu de vergelijking met %eax voor de subl-instructie zetten om tijd te sparen, wordt de berekening iets complexer. We vergelijken dus in plaats van met 2 met 3, en de berekening van het eindresultaat wordt (%eax-1)*4+2, ofwel %eax*4-2. We veranderen de 2 in de leal-instructie naar een -2 en we zijn klaar.

Als we %eax vergelijken met 2 voor de tweede call naar `lucas` (of dus 4 voor de vermindering met 2), weten we door de eerste vergelijking dat %eax al zeker niet kleiner kan zijn dan 2. We hoeven dus maar rekening te houden met één mogelijkheid: het resultaat van de call zal 6 zijn en het eindresultaat zal 2*6=12 + de uitkomst van de vorige call zijn.

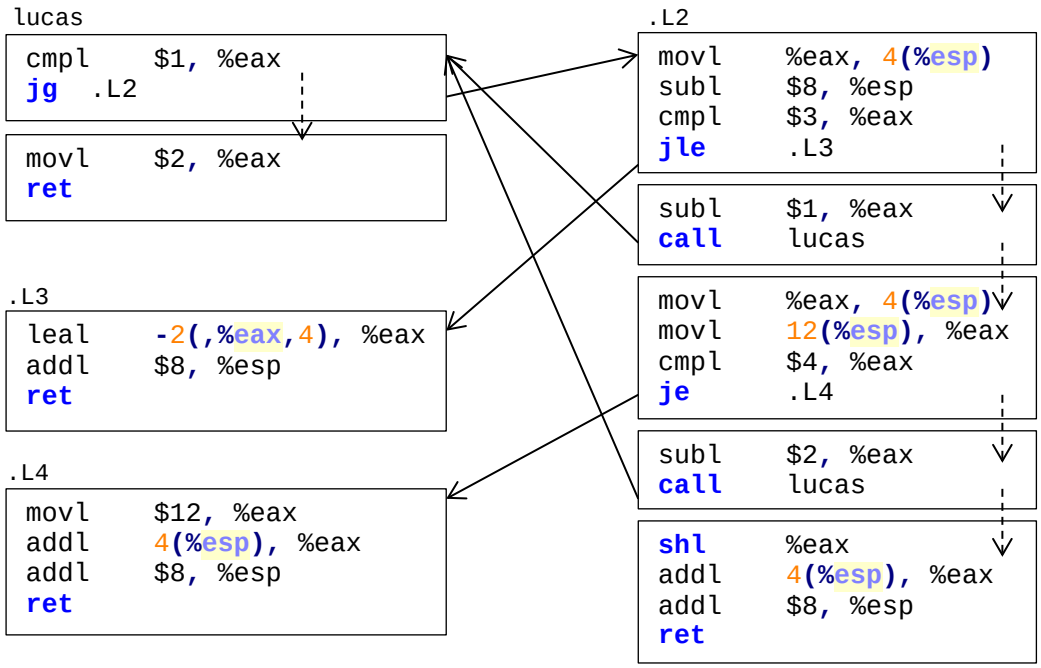
Dankzij deze optimalisatie stranden we bij 34 miljoen, ofwel 34823172 cycli! Tegenover de 211 miljoen cycli van de originele code is dit een uitzonderlijke verbetering.

```
     DERDE OPTIMALISATIESTAP                LAATSTE OPTIMALISATIESTAP
1   lucas:                                  lucas:
2       cmpl    $1, %eax                        cmpl    $1, %eax
3       jg      .L2                             jg      .L2
4       movl    $2, %eax                        movl    $2, %eax
5       ret                                     ret
6   .L2:                                    .L2:
7       movl    %eax, 4(%esp)                   movl    %eax, 4(%esp)
8       subl    $8, %esp                        subl    $8, %esp
9                                               cmpl    $3, %eax
10                                              jle     .L3
11      subl    $1, %eax                        subl    $1, %eax
12      call    lucas                           call    lucas
```

```
13      cmpl    $2, %eax
14      jle     .L3
15      movl    %eax, 4(%esp)
16      movl    12(%esp), %eax
17
18
19      subl    $2, %eax
20      call    lucas
21      shl     %eax
22      addl    4(%esp), %eax
23      addl    $8, %esp
24      ret
25  .L3:
26      leal    (%eax,%eax,2), %eax
27      addl    $8, %esp
28      ret
29
30
31
32
33
```

```
    #voor de call gezet
    #en voor de subl
    movl    %eax, 4(%esp)
    movl    12(%esp), %eax
    cmpl    $4, %eax
    je      .L4
    subl    $2, %eax
    call    lucas
    shl     %eax
    addl    4(%esp), %eax
    addl    $8, %esp
    ret
.L3:
    leal    -2(,%eax,4), %eax
    addl    $8, %esp
    ret
.L4:
    movl    $12, %eax
    addl    4(%esp), %eax
    addl    $8, %esp
    ret
```

1.5 Controleverloopgraaf



1.6 Keuze van het algoritme

Een eerste mogelijkheid om het algoritme zelf te optimaliseren is werken vanaf lucas(1) in een for-lus en zo de Lucasgetallen berekenen en in een array stoppen tot aan lucas(n). Dit elimineert de recursie en zal veel sneller resultaat geven. Er is echter nog een betere manier om de Lucasgetallen te berekenen: een wiskundige formule die we uit het recursieve algoritme kunnen afleiden dankzij onze kennis van het vak Discrete Wiskunde.

De recurrente betrekking van deze functie is $lucas_{n-1} + 2 * lucas_{n-2}$. De karakteristieke vergelijking hiervan is $x^2 - x - 2 = 0$. We krijgen hieruit twee oplossingen:

$$x = -1 \text{ en } x = 2$$

De algemene oplossing van deze recurrente betrekking is dus:

$$lucas_n = \alpha_1 * 2^n + \alpha_2 * (-1)^n$$

Aangezien $lucas_0$ en $lucas_1$ beide gelijk zijn aan 2, kunnen we stellen dat:

$$\begin{cases} \alpha_1 + \alpha_2 = 2 \\ \alpha_1 * 2 - \alpha_2 = 2 \end{cases}$$

Hieruit volgt dat:

$$\begin{cases} \alpha_1 = \frac{4}{3} \\ \alpha_2 = \frac{2}{3} \end{cases}$$

De algemene term uit de Lucasrij is dus:

$$lucas_n = \frac{4}{3} * 2^n + \frac{2}{3} * (-1)^n$$

Of iets vereenvoudigd, zodat we het later gemakkelijk kunnen implementeren in assembler (zie vraag 8):

$$lucas_n = \frac{2}{3} * (2^{n+1} + (-1)^n)$$

Als we dus gewoon een functie schrijven die deze formule berekent, zoals onderstaande implementatie in Java, hebben we meteen het gewenste Lucasgetal.

JAVA-IMPLEMENTATIE

```java
public int lucas(int n) {
    return 2*((int)Math.pow(2,n+1)+(int)Math.pow(-1,n))/3;
}
```

2. EXTRA OPGAVEN

2.7 Optimaliserende compilatie

De uitvoeringstijd van de door de compiler geoptimaliseerde assembler-code is nu nog maar 9185119 cycli. De uitleg hiervoor is dat er nergens wordt teruggesprongen naar de `main`-functie, waar `lucas` wordt opgeroepen. De functie wordt dus niet 1000 keer opgeroepen, maar 1 keer. De niet-geoptimaliseerde `lucas`-functie met 1 iteratie heeft ongeveer dezelfde uitvoeringstijd. De uitleg hiervoor is dat de opties in de makefile ervoor zorgen dat de compiler “slimmer” wordt, ziet dat die iteratie niet nodig is en de hele lus gewoon weglaat.

2.8 Implementatie in assembler

Na implementeren van onze formule in assembler (met gebruik van de `main`-functie van de niet-geoptimaliseerde code) is de uitvoeringstijd van `lucas(n)` steeds 10 miljoen cycli, voor elke `n`. Het verschil tussen de uitvoeringstijden van `lucas(0)` en `lucas(22)` is maar 6112 cycli. De uitvoeringstijd van `main`-functie is ook ongeveer 10 miljoen cycli, dus de uitvoeringstijd van de `lucas`-functie zelf ligt wel erg laag.

Een woordje uitleg bij de code: om iets gemakkelijk te vermenigvuldigen met machten van 2, shiften we dat natuurlijk naar links. Maar in assembler moet deze macht in een register steken: %cl. Net zoals %al, wat de laatste byte van %eax is, is %cl de laatste byte van %ecx. De instructie testb doet een logische AND-operatie, in dit geval tussen 00000001 en %cl, dat de `n` bevat. Als dus de laatste bit van %cl niet op 1 staat maar op 0 en `n` dus even is, zal de zero flag aangezet worden. Dan zal jz dus uitgevoerd worden en zal er ook 2 opgeteld worden (dus wordt er i.p.v. 1 afgetrokken 1 opgeteld) voordat de berekening verder gaat.

```
                    IMPLEMENTATIE IN ASSEMBLER
lucas:
    movb    %al, %cl
    movl    $2, %eax
    shl     %cl, %eax
    movl    $3, %ebx
    subl    $1, %eax
    testb   $1, %cl
    jz      .L3
.L2:
    movb    $1, %cl
    shl     %eax
    divl    %ebx
    ret
.L3:
    addl    $2, %eax
    jmp     .L2
```

2.9 Besluit

Op onderstaande grafiek is het duidelijk dat er een groot verschil is tussen de uitvoeringstijd van de originele Lucasfunctie en de uitvoeringstijd van de geoptimaliseerde. Dit komt echter grotendeels doordat er bij de geoptimaliseerde telkens maar 1 iteratie gebeurt, terwijl er 1000 worden uitgevoerd bij de originele. Hierdoor is ook de grafiek van onze oplossing van vraag 4 onvergelijkbaar met die van de geoptimaliseerde Lucasfunctie. Ook ons resultaat van vraag 8 gebruikt de `main`-functie die telkens 1000 iteraties uitvoert. Toch zien we dat ons resultaat van vraag 8 bij hogere getallen beter presteert dan de geoptimaliseerde Lucasfunctie.

Op een eerlijke grafiek zou de `main`-functie dus overal evenveel geoptimaliseerd zijn, maar zelfs zonder deze optimalisatie is onze implementatie uit vraag 8 duidelijk al veel efficiënter dan de geoptimaliseerde Lucasfunctie.

Over het algemeen kunnen we besluiten dat een optimaliserende compiler programmeurs van een deel van het werk kan besparen. Toch loont het altijd de moeite om zelf de code te bekijken en deze te verbeteren/zelf te maken. Dit is bewezen door onze snelle `lucas`-funcites met een trage `main`-functie uit te voeren in vraag 4 en 8.