Författare: Lasse Karagiannis
Kontakt:lasse.l.karagiannis@gmail.com
Uppgift: C_task_10
Projekt: “C code optimization”
Datum: 2016-11-13

Innehållsförteckning

1 Sammanfattning C-kods optimering

Avsnittet handlade om olika tekniker att skriva optimal C-kod avseende på storlek och snabbhet. Med kännedom om hur C-kompilatorn behandlar olika konstruktioner, så kan man få olika resultat på den genererade assemblern. Följande tips gavs och demonstrerades i boken:

- Reducera en variabels scope, vilket kräver en C-kompilator som implementerar den senaste standarden (C11).
- Använd variabler som är lika långa som databuss-bredden för att cykler för sign-extension undviks.
- Om man vet att en variabel bara kommer att anta positiva värden, så deklarera den explicit som unsigned.
- Använd volatile endast om du ska läsa från en hårdvaruport.

- Undvik att kompilatorn genererar kod för typkastning för enkla saker som att jämföra mellan två integer och inte en integer och en short.
- Undvik temporära variabler, vilket dock kan försämra läsbarheten.
- Använd makrodefinitioner för konstanter. Preprocessorn byter ut dessa emot siffervärden, och skulle addition av konstanterna göras i koden så kommer pre-processorn att göra det vid kompileringstiden.
- Använd C-primitiven **register** för att tvinga kompilatorn att allokera ett dedikerat register inne i CPU:n för variabler som det krävs snabbare åtkomst till.
- Placera den största medlemsvariablen i en struct först. Detta kan snabba upp åtkonmsten, oklart dock hur.

- Gruppera medlemsvaribaler i en struct som anropas konsekutivt. Chansen finns att dessa placeras på samma page och man undviker page-fault med tillhörande vänte tid för att läsa in en ny sida i instruktionscashen.
- Använd ”Bit fields” för att lagra värden i enskilda byter eller bitar i en unsigned int.

- Använd nyckelordet static för hjälp-funktioner som endast anropas av funktioner tillhörande en och samma header-fil. Oklarheter kring hur detta fungerar.
- Använd const för inparametrar för att indikera för kompilatorn att inparametern inte kommer att skrivas till.
- Undvik att skicka pekare till funktioner (pass by reference).
- Undvik rekursion om möjligt.
- Använd inlining. Här tar boken ett exempel med macro-expansion istället för att beskriva nyckelordet **inline**, som hoppats över.
- Flytta ut gemensam kod ur if-else-satser

- Dekrementera forloopar för att kompilatorn ska kunna generera en instruktion som hoppar när Zero-flaggan är satt, för att undvika en subtraktion.
- Rulla upp loopar.
- Använd switch istället för if, när det är möjligt
- Använd break för att gå ur en loop om möjligt.

- Deferera en nästlad pekare en gång för alla och använd resultatet för att deferera inne i en loop.
- Skicka pekare till arrayer strukter och andra stora datastrukturer till funktioner istället för att skicka datastrukturen själv som inparameter

2 Sammanfattning undersökning av anvisningar för C-kods optimering

2.1 Exempelkoden

Exempelkoden för mätning av tiden och kod-storleken kördes. Se nedan.

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



int main()
{
	int offset;
	int value_a, execution_time, time;
	TIMER_RESET;
	TIMER_START;
	offset = TIMER_READ;
	while(1){
		TIMER_RESET;
		TIMER_START;
		value_a = 5;
		time = TIMER_READ;
		execution_time = time - offset;
		printf("offset = %d\ntime = %d\nexecution_time = %d\n\n",
				offset, time, execution_time);
	}
  printf("Hello from Nios II!\n");

  return 0;
}
```

Kodens storlek är 7884 bytes. Detta resultat får man om man gör en ”clean”, därefter bygger och sedan markerar elf-filen. Se nedan

```
Info: (Task10.elf) 7884 Bytes program size (code + initialized data).
Info:              8191 KBytes free for stack + heap.
```

Summary.html ger inte kodstorleken, utan man får använda ovan föreskrivna metod eller använda kommandoshell. Det är dock oklart för mig hur jag hittar till arbetskatalogen, därför att shellet verkar förutsätta att man har sina projekt i anslutning till altera_lite, medan jag har mina projket under ”Mina dokument”.

Slave Descriptor	Address Range	Size	Attributes
jtag_uart	0x01009028 - 0x0100902F	8	printable
sysid	0x01009020 - 0x01009027	8	
sdram_pll	0x01009010 - 0x0100901F	16	
TIMER_HW_IP_0	0x01009000 - 0x0100900F	16	
onchip_ram	0x01004000 - 0x01007FFF	16384	memory
sdram	0x00800000 - 0x00FFFFFF	8388608	memory

Bild1 Summary.html

Exekveringstiden uppmättes till 21 klockcykler med 50MHz klockan. Se nedan inklippta körresultat.

```
offset = 31
time = 52
execution_time = 21

offset = 31
time = 52
execution_time = 21

offset = 31
time = 52
execution_time = 21
```

Exempelkoden laddar endast en immediate till ett register, med teskoderna som undersöks i följande avsnitt testas genom ett funktionsanrop. För att endast undersöka funktionen och inte anropet och skriviningen till variablen value_a i main, så har exekveringstiden och utrymmeskravet för exempelkoden räknats bort i analysen som görs på exempelkoderna.

2.2 Testkoder

Nedan följer tre olika exempel på funktioner med tillika opimeringar. Koden, kodstorleken, exekveringstiden samt assemblern presenteras för var och en. Slutligen görs en analys av resultatet då en förklaring ges till.

2.2.1 Undvik temporära variabler

Testkoden utan optimering har storleken 7984 Bytes.

```
Info:(CASE_4.elf) 7984 Bytes program size (code + initialized data).
Info:              8183 KBytes free for stack + heap.
```

Subtraheras mätkoden så att endast funktion och dess anrop kvarstår, så får vi 7984-7884 = 100 bytes kod.

Exekveringstiden är 866 cykler

```
offset = 32
time = 898
execution_time = 866
```

Vi vet att resultatet av ett funktionsanrop skrivs till ett register, som tilldelas value_a. Räknar vi bort tiden för skrivning till value_a, vilket kanske är onödigt noggrannt, så fås 866-21 = 845 cykler

Optimering avseende att den temporära variabeln eliminerases gav följande resultat:

```
Optimering
Info: (CASE_4.elf) 7964 Bytes program size (code + initialized data).
Info:              8183 KBytes free for stack + heap.

offset = 31
time = 804
execution_time = 773
```

Tabell 1 Resultat eliminiering temporär variabel

	Före optimering	Efter optimering

Kodexempel	`int test(int I) {`
	`int sum = 0;`
	`sum = sum + I + 8 + 10 + 45`
	`+ (300 *`
	`I);`
	`return sum;`
	`}`
		`int test(int I) {`
		
		`return I + 8 + 10 + 45 +`
		`(300 *`
		`I);`
		`}`

Storlek på kod	7984-7884=100bytes	7964-7884 = 80
Tid för exekvering	866-21=845cykler vid 50MHz	773 -21 = 752 cykler vid 50MHz

Assembler Nios II Economy	test:
	int test(int I) {
	addi sp,sp,-20
		test:
		int test(int I) {
		addi sp,sp,-16

	stw ra,16(sp)
	stw fp,12(sp)
	stw r16,8(sp)
	addi fp,sp,12
	stw r4,-8(fp)
	int sum = 0;
	stw zero,-12(fp)
	sum = sum + I + 8 + 10 +
	45 + (300 *
	ldw r3,-12(fp)
	ldw r2,-8(fp)
	add r2,r3,r2
	addi r16,r2,63
	movi r5,300
	ldw r4,-8(fp)
	call 0x800160 <__mulsi3>
	add r2,r16,r2
	stw r2,-12(fp)
	return sum;
	ldw r2,-12(fp)
	}
		stw ra,12(sp)
		stw fp,8(sp)
		stw r16,4(sp)
		addi fp,sp,8
		stw r4,-8(fp)
		return I + 8 + 10 + 45 +
		(300 *
		ldw r2,-8(fp)
		addi r16,r2,63
		movi r5,300
		ldw r4,-8(fp)
		call 0x80014c <__mulsi3>
		add r2,r16,r2
		}

Analys Resultatet visade att kod-storleken minskade med 20 bytes. Det beror på att variablen sum eliminerades . Prestanda ökade således, sådant att loop-exekveringstid minskade med 845-752 = 93 cykler.

2.2.2 Inlining med macro-expandering

Nedan beskrivs resultatet från test om den tidigare optimerade koden kan optimeras ytterliggare medelst användning av makroexpansion.

Tabell 2 Resultat makro-expansion

	Före optimering	Efter optimering

Kodexempel	`int test(int I) {`
	
	`return I + 8 + 10 + 45 +`
	`(300 *`
	`I);`
	`}`
		`#define test(I)`
		`(I+8+10+45+(300*I))`

Storlek på kod	7964-7884 = 80	7888-7884 = 4
Tid för exekvering	773 -21 = 752 cykler vid 50MHz	30-21 = 9 vid 50MHz

Assembler Nios II Economy	test:
	int test(int I) {
	addi sp,sp,-16
	stw ra,12(sp)
	stw fp,8(sp)
	stw r16,4(sp)
	addi fp,sp,8
	stw r4,-8(fp)
	return I + 8 + 10 + 45 +
	(300 *
	ldw r2,-8(fp)
	addi r16,r2,63
	movi r5,300
	ldw r4,-8(fp)
	call 0x80014c <__mulsi3>
	add r2,r16,r2
	}
		value_a = test(100);
		movi r2,30163
		stw r2,-12(fp)

Resultatet visade att kod-storleken minskade med 76 bytes. Det beror på att preprocessorn beräknade uttrycket test(100) vid kompileringstillfället. Koden snabbades upp, så att ”anropet” endast tar 9 cykler istället för 752.

2.2.3 Undvik dynamisk allokering av minne

Nedan redovisas undersökningen om huruvida dynamisk allokering av en integer vektor på heapen än motsvarande fix storlek på stacken.

Storleken för koden som innehåller dynamisk minnesallokering på heapen gavs vid kommandot ”clean”:

```
Info: (CASE_4.elf) 8860 Bytes program size (code + initialized data).
Info:              8182 KBytes free for stack + heap.
```

Exekveringstiden varierade från varv till varv vid detta fall:

```
offset = 31
time = 1402
execution_time = 1371

offset = 31
time = 1407
execution_time = 1376
```

Efter optimering som innebar användning av vektor med fix storlek så fås följande storlek

```
Info: (CASE_4.elf) 7928 Bytes program size (code + initialized data).
Info:              8183 KBytes free for stack + heap.
```

Exekveringstiden förblev blev konstant från ett varv till nästa varv

```
offset = 31
time = 216
execution_time = 185
```

Tabell 3 Resultat dynamisk och statisk allokering

	Före optimering	Efter optimering

Kodexempel	`int test(int I) {`
	`int *intPtr;`
	`int trash;`
	`intPtr = (int *)`
	`malloc(I*sizeof(int));`
	`trash = *intPtr;`
	`free(intPtr);`
	`return trash;`
	
	`}`
		`int test(int I) {`
		
		`int arr[100];`
		`return arr[0];`
		
		`}`

Storlek på kod	8860-7884 = 976	7928-7884 = 44
Tid för exekvering	1376 -21 = 1355 cykler vid 50MHz	185-21 = 164 vid 50MHz

Assembler Nios II Economy	test:
	addi sp,sp,-20
	stw ra,16(sp)
	stw fp,12(sp)
		test:
		addi sp,sp,-408
		stw fp,404(sp)
		addi fp,sp,404

	addi fp,sp,12
	stw r4,-4(fp)
	intPtr = (int *)
	malloc(I*sizeof(int));
	ldw r2,-4(fp)
	add r2,r2,r2
	add r2,r2,r2
	mov r4,r2
	call 0x80015c <malloc>
	stw r2,-12(fp)
	trash = *intPtr;
	dw r2,-12(fp)
	ldw r2,0(r2)
	stw r2,-8(fp)
	free(intPtr);
	ldw r4,-12(fp)
	00800098: call 0x800170
	<free>
	return trash;
	ldw r2,-8(fp)
	}
		stw r4,-4(fp)
		return arr[0];
		ldw r2,-404(fp)
		}

Vi ser att dynamisk allokering på heapen tar mycket längre tid än allokering på stacken. Detta beror på drivrutinen malloc. Om man inte har problem med utrymme kan det vara en idé att allokera vektorer med fast storlek.