Compiler-Optimierungen

Seminar Effiziente Programmierung

Marcel Papenfuss

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

2019-11-12

Gliederung (Agenda)

Grundlagen Assembler

Grundlagen Assembler in Kürze

- Register: Speicherort für Daten mit denen gearbeitet wird
  - 8 Allzweckregister: %eax, %ebx, %ecx, ...
  - Liegen im Prozessor für schnellen Zugriff
- Register reichen nicht aus, dafür Hauptspeicher verwendet
- Immediatewerte: konstante Werte
  - `$5`
- Instruktion: addiere 5 auf den Inhalt des Registers eax
  - `add $5, %eax`

Grundlagen Assembler in Kürze...

- Mov-Befehl: Kopiert Wert
  - `mov Quelle Ziel`
- Compare-Befehl: Subtrahiert Quelle von Ziel
  - ohne Ergebnis Speicherung ändert nur Flags
  - `cmp Quelle, Ziel`
- Jump-Befehl: Bedingte Sprünge anhand von gesetzten Flags
  - `jne Sprungziel`: ZF=0, Quelle != Ziel
  - `je Sprungziel`: ZF=1, Quelle == Ziel
- Label: sind Sprungmarken

Grundlagen Assembler in Kürze...

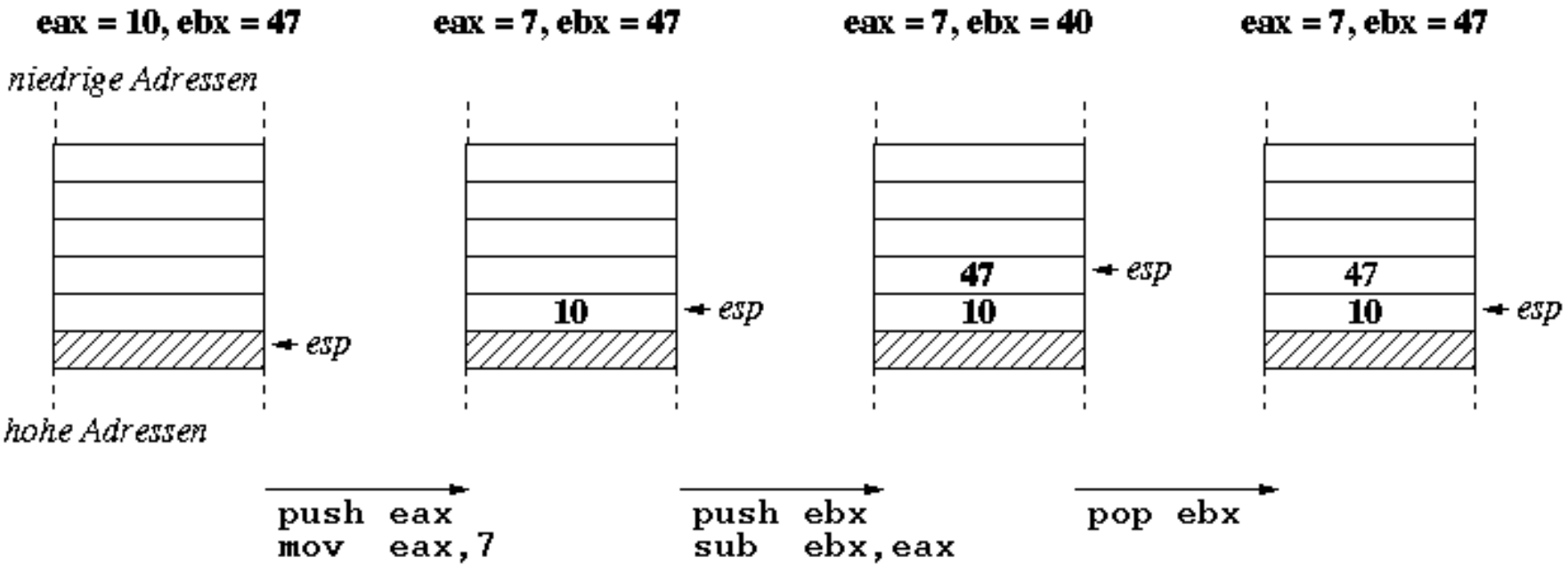

Abbildung: Stack Beispiel [Sta19a]

Wozu optimieren wir?

Optimieren?

"The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it yet."
Michael A. Jackson

Was ist besser?

```
1  for (i = 0; i < 9999; i++) {
2      for (j = 0; j < 999; j++) {
3          m[i][j] = 5;
4      }
5  }
```

Listing 1: Row Major Order

```
6  for (j = 0; j < 999; j++) {
7      for (i = 0; i < 9999; i++) {
8          m[i][j] = 5;
9      }
10 }
```

Listing 2: Column Major Order

Column Row Major Order Vergleich

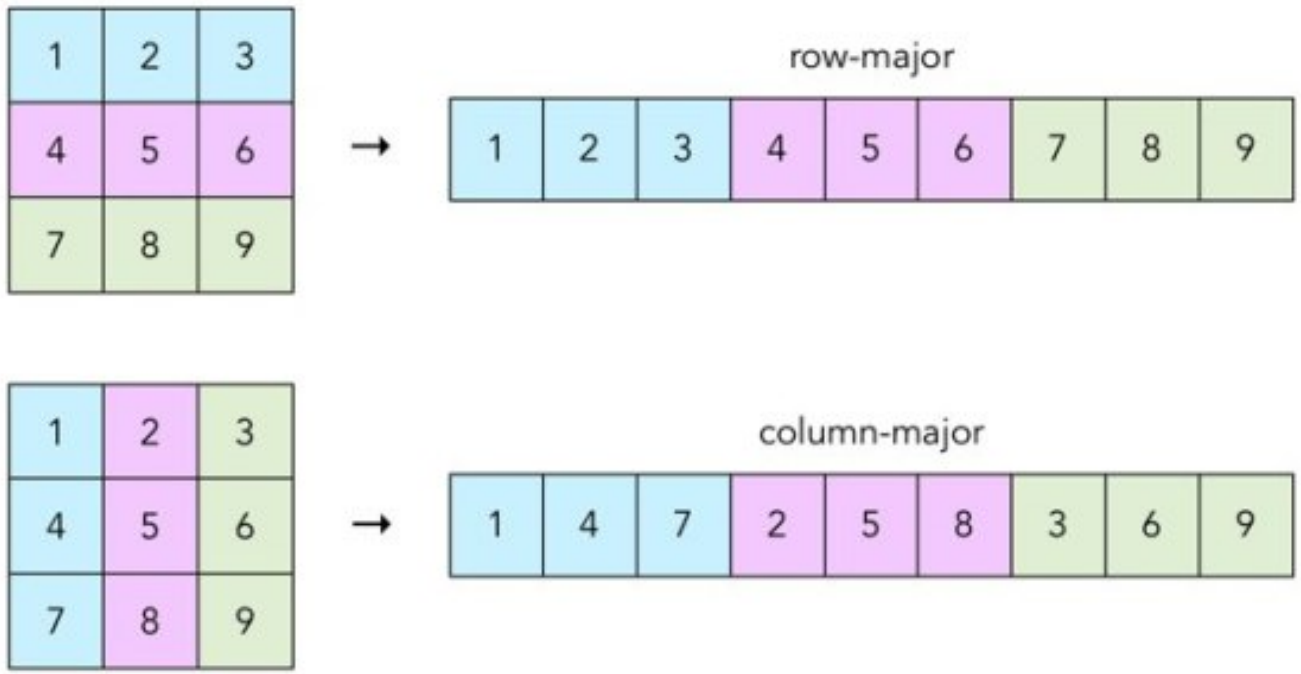

Abbildung: Vergleich Row-Column-Major [Row19]

Space-Time-Tradeoff

- Informatik dreht sich oft um Effizienz (Algorithmen)
- Laufzeit ist nicht immer wichtigste Einschränkung
- Schnellerer Code benötigt in der Regel mehr Speicher
- Kompromiss zwischen Laufzeit und Speicherplatz

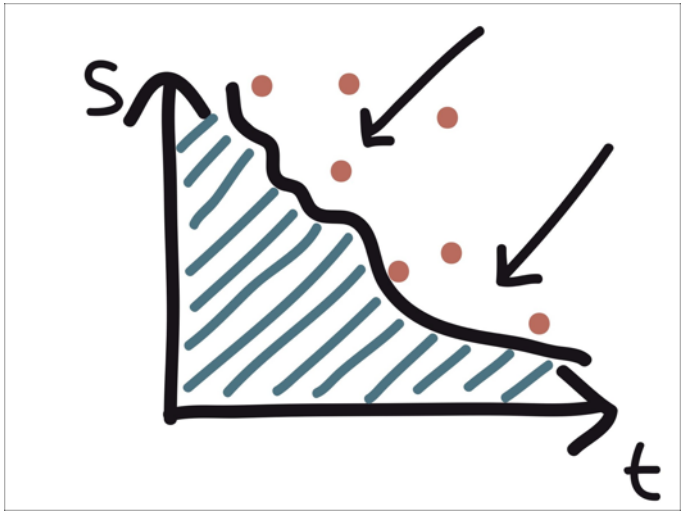

Abbildung: Space-Time-Tradeoff [Spa19]

Optimierungstechniken

- Dead Code Elimination, Constant Folding/Propagation
- Function Inlining
- Loop Unrolling

Dead Code Elimination

- Entfernen von nicht genutzten Codefragmenten
- Codegroße wird verringert
- Nicht benötigte Operationen verbrauchen keine Laufzeit
- Programmstrukturen können vereinfacht werden

Dead Code aufdecken

```
 1 int foo() {
 2     int a = 1;
 3     int b = 500 * 200 / 123;
 4     int c;
 5     if(true) {
 6         c = 5 * a + 5;
 7     } else {
 8         c = 50;
 9     }
10     return (c * 20) / 5;
11     c = 0;
12     return c;
13 }
```

Listing 3: Ausgangsproblem DCE

Dead Code aufdecken...

```
1 int foo() {
2     int a = 1;
3     int c = 5 * a + 5;
4     return (c * 20) / 5;
5 }
```

Listing 4: Dead Code entfernt

Constant Folding und Propagation

- Folding: Ausdrücke zu Konstanten umformen
- Dadurch nicht zur Laufzeit auswerten
- Propagation: Weitergeben der neuen Konstanten

Constant Folding und Propagation...

```
1 int foo() {
2     int a = 1;
3     int c = 5 * a + 5;
4     return (c * 20) / 5;
5 }
```

Listing 5: Neues Ausgangsproblem

```
1 int foo() {
2     return 40;
3 }
```

Listing 6: Nach Constant Folding

Assembler

```
1 gcc -c -O1 deadcode.c -S
```

Listing 7: Kompilieren des Quellcodes

```
1 foo:
2     movl   $40, %eax ; eax = 40
3     ret              ; return
```

Listing 8: Erzeugte Assembler Datei

Function Inlining

- Funktionsaufrufe sind teuer
- Funktionsrumpf wird an die Stelle des Aufrufs eingesetzt
- Aufrufoverhead verringern
  - Kann sich bei kleinen Funktionen bemerkbar machen
- Anwenden weiterer Techniken wie Constant Folding oder Dead Code Elimination

Vorgänge beim Funktionsaufruf

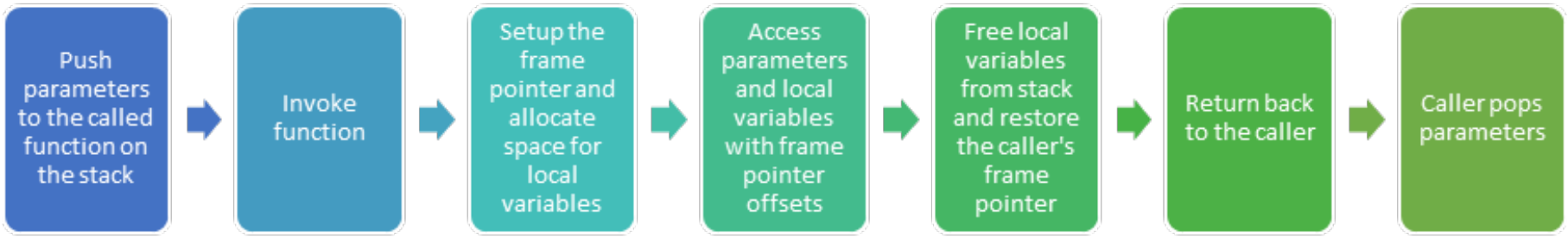

Abbildung: Funktionsaufruf im Stack [Sta19b]

Function Inlining Nachteile

- Steigende Registeranzahl, dadurch Code aufspalten
- Codegröße kann ansteigen
  - Cacheverhalten kann sich verschlechtern (Cache Miss)
  - Code muss aus Hauptspeicher geladen werden
  - Performance sinkt wieder

Function Inlining Code Beispiel

```
11 void max(int a, int b) {
12     if(a < b) {
13         maxValue = b;
14     } else {
15         maxValue = a;
16     }
17 }
18 int main() {
19     int a = 1;
20     int b = 3;
21     max(a, b);
22 }
```

Listing 9: Kein Inlining

```
23 int main() {
24     int a = 1;
25     int b = 3;
26     if(a < b) {
27         maxValue = b;
28     } else {
29         maxValue = a;
30     }
31 }
```

Listing 10: Inlining

Function Inlining Code Assembler

```
1  max:
2    cmpl  %esi, %edi   ; Vgl die Werte von esi und edi
3    cmovl %esi, %edi   ; Wenn Vgl wahr, dann edi = esi
4    movl  %edi, maxValue(%rip) ; maxValue = edi
5    ret                ; return
6  main:
7    movl  $3, %esi     ; esi = 3
8    movl  $1, %edi     ; edi = 1
9    call  max          ; Sprung zur max Methode
10   movl  $0, %eax     ; eax = 0
11   ret                ; return
```

Listing 11: Assembler Code ohne Inlining

Loop Unrolling

- Schleifendurchläufe reduzieren
  - Mehrere Kopien des Rumpfes
  - Dafür Schleifenbedingung anpassen
- Schleife komplett auflösen
  - Rumpf der Schleife so oft ausgeben wie Durchläufe
- Schleifenoverhead wird verringert
- Pipelining kann möglich werden
- Codegröße steigt an, dadurch wieder Cache Probleme

Loop Unrolling Code Beispiel

```
1 for(int i = 0; i < 4; i++) {
2     printf("Loop \n");
3 }
```

Listing 12: Kein Unrolling

```
1 for(int i = 0; i < 4; i+=2) {
2     printf("Loop \n");
3     printf("Loop \n");
4 }
```

Listing 13: Unrolling mit Abrollfaktor 2

Loop Unrolling Assembler

```
1 .L2:
2   call   printf      ; Aufruf von Printf
3   addl   $1, %ebx    ; ebx += 1
4   cmpl   $4, %ebx    ; Vgl. ebx und 4
5   jne    .L2         ; Wenn Vgl. nicht gleich gehe zu L2
6   movl   $0, %eax    ; eax = 0
7   ret                ; return eax
```

Listing 14: Assembler Code für normale Schleife

Loop Unrolling Dynamisch

```
1 for (i = 0; i < (n % 2); i++) {
2   y[i] = i;
3 }
4 for ( ; i + 1 < n; i += 2) {
5   y[i] = i;
6   y[i+1] = i+1;
7 }
```

Listing 15: normale Schleife mit dynamischer Grenze abgerollt

GCC Flags

GCC

- Betrachten GNU Compiler Collection (GCC)
- Vielzahl an Optimierungstechniken (ca. 100)
- Bietet verschiedene Optimierungslevel
- Level wurden durch Heuristiken zusammengestellt
- Verbesserung Performance, Codegröße, Energieeffizienz abhänging von gewählten Optimierungen

GCC Flags

- -O0
  - Keine Optimierungen und die Standardeinstellung
  - Erzeugtes Programm ist wahrscheinlich größer und langsamer, als mit Optimierungen
- -O1
  - Bei diesem Level werden gängige Optimierungstechniken aktiviert
    - z.B. DCE
  - In kurzer Zeit ein optimiertes Programm zu erzeugen
  - Erfordert in der Regel keine größe Kompilierzeit
  - Manchmal gegensätzliche Ziele

GCC Flags..

- -O2
  - Diese Option aktiviert weitere Techniken, die in keinem Konflikt zwischen Zeit und Platz stehen
  - Hat im Vergleich zu -O1 eine langsamere Kompilierzeit
- -O3
  - Verwendet Techniken mit noch aufwendigeren Optimierungen
  - Techniken teurer in Kompilierzeit und Speicherplatz
  - Kann schnellen Code erzeugen
  - Größe kann zu Cache Misses führen, Performanceverlust
  - Meistens besser -O2 zu nutzen, um größere Chancen zu haben die Cachegröße nicht zu überschreiten.

GCC Flags..

- -Os
  - Optimierungen, die die Größe des Programms so gering wie möglich halten soll
  - Für Embedded Systems gut geeignet
- -Og
  - Ist für das Debugging eine bessere Wahl als -O0
  - Wahl zum Bearbeiten, Kompilieren und Debuggen
  - Schnelle Kompilierung und sinnvolle Optimierungen
- -Ofast
  - Erweitert -O3 um mathematische Optimierungen (kann ungenau werden)

Beispiel: Partielle Differentialgleichungen (HLR)

	-O0	-O1	-O2	-O3	-Os	-Og
Kompilierzeit	.668s	.483s	.544s	.550s	.507s	**.451s**
Laufzeit	1m17.634s	0m27.988s	0m14.423s	**0m13.856s**	0m21.312s	0m25.172s
Größe	7.4K	6.3K	6.7K	6.9K	**6.0K**	6.9K

Tabelle: Verschiedene Optimierungsstufen für partdiff.c

Zusammenfassung

- Zeit-Platz-Kompromiss schon bei Planung mit einfließen lassen
- Compiler haben schon viele Möglichkeiten Code zu optimieren
- Nicht immer ersichtlich was optimiert wird
- Der Compiler kann nicht alles optimieren
  - Der Entwicktler spielt immer noch eine wichtige Rolle
- GCC bietet vorgefertigte Optimierungslevel für verschiedene Anforderung

Hoisting

```
1 for (i = 0; i < 100; i++)
2 {
3   a[i] = x + y;
4 }
```

Listing 16: Kein Hoisting

```
1 t = x + y;
2 for (i = 0; i < 100; i++)
3 {
4   a[i] = t;
5 }
```

Listing 17: Hoisting

Literatur

[Ass19] 11 2019. Avaible at https://runtimebasic.net/Assembler:Funktionen:Beschreibung-Kurz-CPU.

[BG04] M. Stallman Brian Gough. An introduction to gcc for the gnu compilers gcc and g++. 2004.

[HS12] Sebastian Hack Helmut Seidl, Reinhard Wilhelm. Compiler design: Analysis and transformation. *Springer Science und Business Media*, 2012.

[KH08] L. Eeckhout K. Hoste. Cole: compiler optimization level exploration. *In Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, pages 165–174, 4 2008.

[Mog17] T. Mogensen. *Introduction to compiler design.* Springer, 2017.

[PL09a] P. Marwedel P. Lokuciejewski. Combining worst-case timing models, loop unrolling, and static loop analysis for wcet minimization. *In 2009 21st Euromicro Conference on Real-Time Systems*, pages 35–44, 7 2009.

[PL09b] P. Marwedel und K. Morik P. Lokuciejewski, F. Gedikli. Automatic wcet reduction by machine learning based heuristics for function inlining. *In 3rd workshop on statistical and machine learning approaches to architectures and compilation*, pages 1–15, 2009.

[PPC89] W-W. Hwu Pohua P. Chang. Inline function expansion for compiling c programs. *ACM SIGPLAN Notices*, (7):246–257, 6 1989.

[Row19] 11 2019. Avaible at https://craftofcoding.wordpress.com/2017/02/03/column-major-vs-row-major-arrays-does-it-matter/.

[Spa19] 11 2019. Avaible at http://people.cs.vt.edu/darendt/about/teaching/Entries/2012/1/30_Space-Time_Tradeoff.html.

[Sri99] Amitabh Srivastava. Link time optimization via dead code elimination, code motion, code partitioning, code grouping, loop analysis with code motion, loop invariant analysis and active variable to register analysis. *U.S. Patent No. 5,999,737*, 12 1999.

[Sta19a] 11 2019. Avaible at https://www4.cs.fau.de/Lehre/WS09/V_BS/Uebungen/oostubs/assembler.shtml#procedure.

[Sta19b] 11 2019. Avaible at https://www.eventhelix.com/RealtimeMantra/Basics/CToAssemblyTranslation.htm.