Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Seminar Effiziente Programmierung

Compiler-Optimierungen

vorgelegt von

Marcel Papenfuss

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

Studiengang:	Informatik (B.Sc.)
Matrikelnummer:	7065349

Betreuer:	Dr. Michael Kuhn

Hamburg, 2020-04-02

Inhaltsverzeichnis

1 Einleitung

Aufgaben, Prozesse oder ganze Programme, die von einem Computersystem ausgeführt werden sollen, werden von einem Entwickler üblicherweise in einer höheren Programmiersprache geschrieben, statt direkt in Maschinensprache. Die Befehle werden anschließend durch einen Compiler oder Interpreter in Maschinensprache übersetzt. Die Ressourcen eines Computers, wie Speicher oder Prozessorleistung, sollten mit Bedacht genutzt werden, weil nicht in jeder Umgebung unendliche viele Kapazitäten zur Verfügung stehen. Der Compiler steht zwischen dem Maschinencode und dem höheren Quellcode und ist deswegen gut geeignet, um während des Kompiliervorganges Optimierungen vorzunehmen, um Ressourcen bestmöglich zu nutzen.
Bei der Programmierung werden größere Aufgabenteile oft in kleinere Teilaufgaben zerlegt und in extra Methoden, Klassen oder Dateien ausgelagert. Das steigert für den Entwickler die Übersichtlichkeit, Verständlichkeit und Wiederverwendbarkeit. Das Vorgehen bringt für den Menschen Vorteile mit sich, die sich aber auf die Leistung auswirken können.
Der Compiler kompiliert den Quellcode typischerweise in einzelnen Modulen. Dieses Vorgehen begrenzt die Optimierungen auf die einzelnen Module. Der Compiler kann dadurch nicht das ganze Verhalten eines Programms verstehen und analysieren. Codeduplikate, die in zwei Modulen vorkommen, kann ein Compiler nicht optimieren, oder auch Variablen, die in einem weiteren Modul verwendet werden, kann ein Compiler so nicht auf Verwendbarkeit prüfen. [Sri99]
Welche anderen Möglichkeiten ein Compiler stattdessen hat, um den Quellcode zu optimieren und welche Auswirkungen das auf das Programm haben kann, wird in dem nächsten Kapitel näher erläutert. Außerdem wird darauf eingegangen welche verschiedenen Optimierungsstufen der GCC hat und welche Vor- und Nachteile daraus resultieren können. Zudem werden die verschiedenen Optimierungslevel auf ein Beispiel angewendet.

2 Space-Time Tradeoff

Der Zeit-Speicher-Kompromiss (Space-Time Tradeoff) ist nicht nur eine Problemstellung beim Kompilieren eines Programms, sondern im gesamten Bereich der Informatik. Nicht immer ist die Laufzeit die wichtigste Entscheidung bei der Entwicklung eines Algorithmus. Auch der schnellste Quellcode muss nicht zwangsläufig der Beste sein. Beschrieben wird mit dem Kompromiss ein Fall bei dem ein Algorithmus oder Programm mehr Speicherplatz (RAM, HDD) benötigt, dafür die Laufzeit (Berechnungszeit oder Antwortzeiten) gesenkt wird, oder der umgekehrte Fall. Bei beiden Fällen steht jeweils die effiziente Ausführung des Programms unter den technischen Möglichkeiten im Vordergrund. [Spa]
Im Themengebiet der Compileroptimierungen gibt es Techniken bei deinen sowohl der Speicherplatz verringert wird, als auch die Laufzeit verringert wird. Ein Beispiel dafür ist das „common subexpression elimination“-Verfahren, das Teilausdrücke im Quellcode sucht, die zuvor bereits berechnet wurden. Es ist nicht bei allen Optimierungen möglich sowohl das Platz- als auch Zeitverhalten zu optimieren, viele müssen den Zeit-Speicher-Kompromiss eingehen.
Ein gutes Beispiel für eine Optimierungstechnik mit einem Zeit-Speicher-Kompromiss ist das Loop Unrolling, das in Abschnitt 3.3 noch genauer erläutert wird. Es verringert die Ausführungszeit wodurch im Gegenzug die Codegröße steigt. Ein Großteil der Ausführungszeit bei einer Schleifenausführung wird bei kleineren Schleifen, die nicht viel Logik beinhalten, beim Testen der Bedingung benötigt. Wenn eine Schleife vollständig abrollt wird, entfällt der Schleifenkopf und es müssen keine Bedingungen mehr geprüft werden. Dafür kann die Codegröße danach stark zugenommen haben, weil zusätzlicher Quellcode in das Programm eingefügt wird. Nicht nur die Ausführungszeit wird verringert, sondern auch die Möglichkeit die neu gewonnenen einzelnen Ausdrücke parallel auf dem Prozessor auszuführen.
Ob es besser ist Speicherplatz oder Laufzeit einzusparen muss individuell für jedes Programm oder jede Funktion entschieden werden. Einige haben die Anforderung schnell auf ein Ereignis zu reagieren. Andere hingegen müssen auf einem kleinen System mit wenig Speicher laufen.

3 Optimierungstechniken

Der Compiler hat eine Vielzahl an Möglichkeiten den Quellcode während der Kompilierung zu optimieren. In diesem Abschnitt werden drei Beispiele für Compileroptimierungen erklärt und anhand von Beispielen verdeutlicht. Alle Varianten dienen letztendlich dazu, den Quellcode und die Ausführungszeit eines Programms zu verbessern.

3.1 Dead Code Elimination, Constant Folding und Propagation

In diesem Abschnitt werden verschiedene intraprozedurale Optimierungsmethoden vorgestellt. Diese sind teilweise eng miteinander verbunden und bauen aufeinander auf.
Während der Entwicklung von großen Systemen kann es vorkommen, dass Quellcode geschrieben wird, der später doch nicht mehr verwendet wird und keinen Einfluss auf das Programm hat. Der Compiler hat das Potential das zu erkennen und den überflüssigen Programmcode zu entfernen. Dieser Vorgang wird Dead Code Elimination genannt. Der Vorteil davon ist simpel, dennoch nicht zu vernachlässigen: Die Codegröße wird verringert und eventuell nicht benötigte Operationen werden dadurch nicht ausgeführt. Außerdem können vorhandene Programmstrukturen vereinfacht werden, wenn durch das Entfernen Terme verkürzt werden können.
Dead Code Eliminiation wird eng mit Constant Folding genutzt. Beim Constant Folding wird während der Kompilierzeit nach Ausdrücken gesucht, die zu Konstanten umgeformt werden können. Dadurch müssen Ausdrücke nicht erst zur Laufzeit ausgewertet werden. Werden Konstanten eingesetzt oder wurden durch Constant Folding welche erzeugt, kann durch Constant Propagation diese Variable weitergegeben werden und bei Verwendung die Variable ersetzen. [HS12]

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

Listing 3.1: Code mit keinen Optimierungen

Anhand des Beispiels Listing 3.1 lassen sich die beschriebenen Techniken veranschaulichen. Zuerst wird nur die Dead Code Elimination ausgeführt, danach die Constant-Folding-Technik.
In Zeile 3 wird eine lokale Variable `b` deklariert. Nach der ersten Initialisierung wird die Variable in `foo` nicht mehr aufgerufen. Da es sich auch um eine lokale Variable handelt, wird der Wert auch global, außerhalb von `foo`, nicht weiter verwendet. Somit kann der Compiler den Speicherplatz freigeben und die Initialisierung nichtig machen.
Im weiteren Verlauf wird ab Zeile 5 eine if-else-Bedingung genutzt. Die Bedingung, dass in den if-Anweisungsblock gesprungen wird, ist mit `true` angegeben. Das ist immer erfüllt und das Programm würde immer die Zeile 6 ausführen, deswegen wird der else-Block nie betreten und die aufwendige Berechnung in Zeile 8 muss nicht ausgeführt werden. Auch die if-Bedingung kann vom Compiler entfernt werden. Der komplette Block lässt sich damit auf `c = 5 * a + 5` reduzieren.
In Zeile 10 wird das erste return-Statement der Methode angegeben. Das Statement wird ohne Bedingung ausgeführt, was darauf schließen lässt, dass nach der Zeile keine weiteren Operationen innerhalb der Methode ausgeführt werden und die Ausführung der Methode beendet wird. Aufgrund dessen ist jeglicher Programmcode, der nach Zeile 8 geschrieben wurde, nicht mehr relevant. Sowohl die neue Zuweisung für `c` als auch das return-Statement von `c` wird nie erreicht und ausgeführt. Der Compiler wird den Quellcode nach dem ersten return-Statement nicht auswerten und keinen Speicher und keine Laufzeit dafür aufwenden.
Das Zusammenspiel zwischen Constant Folding und Constant Propagation lässt sich anhand von Zeile 6 und 10 erklären. Der Ausdruck in Zeile 6 lässt sich vereinfachen. `a` ist eine lokale Variable, die zur Laufzeit festgelegt ist und kann somit in den Ausdruck eingesetzt werden. Die Zeile lässt sich zu

$$c = 5 * 1 + 5$$

umschreiben. Der neue Ausdruck steht wiederum zur Zeit des Kompilierens fest und

kann zu

$$c = 10$$

vereinfacht werden. Die durch Constant Folding umgeformte Term kann jetzt durch Constant Propagation auf Zeile 10 angewendet werden. Da `c` einen statischen Wert hat, kann er einfach gesetzt werden, daraufhin wird noch einmal Constant Folding angewendet. Der Ausdruck in Zeile 10 wird umgeschrieben und das return-Statement besteht nur noch aus einem Wert:

$$(10 * 20)/5 = 40$$

Durch die Anwendung von Dead Code Elimination, Constant Folding und Propagation kann `foo` zu einer einzelnen Codezeile komprimiert werden. Dass der Compiler auf das gleiche Ergebnis kommt, kann man feststellen, wenn man sich den Assemblercode ausgeben lässt. Mit `gcc -c deadcode.c -S` lässt sich ein Assemblerfile erzeugen. Die Ausgabe Listing 3.2 zeigt, dass ausschließlich der Wert 40 in das Register `eax` geschrieben wird. Das Register wird daraufhin mit return zurückgegeben (Zeile 3) und der Aufruf der Funktion wird danach beendet.

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

Listing 3.2: Assemblercode nach den Optimierungen

3.2 Function Inlining

Funktionsaufrufe wirken sich negativ auf die Optimierung während der Kompilierungszeit aus, zum Beispiel auf die Registerallokation, Common Subexpression Elimination oder Constant Propagation. Beim Funktion Inlining versucht der Compiler dem entgegenzuwirken und den Quellcode für die ausführbare Technik zu optimieren. [PPC89]

Function-Inline-Expansion ist eine Optimierungstechnik, bei der der Rumpf der aufzurufenden Funktion direkt an die Stelle eingesetzt wird, an der sie aufgerufen wird. Typischerweise wird er eingesetzt, um die durchnittliche Ausführungszeit zu verringern. Durch die Technik wird der Aufrufoverhead des Programms verringert, indem unter anderem der Funktionsaufruf wegfällt oder auch Variablen nicht auf den Stack gepusht und gepoppt werden müssen.
Durch das Inlining ist es für den Compiler möglich weitere Optimierungsmöglichkeiten anzuwenden. Vorher war es durch die separaten Funktionen nicht möglich Techniken wie Constant Propagation und Dead Code Elimination anzuwenden. Das zeigt die enge Verbundenheit zwischen den einzelnen Optimierungstechniken.

Der Hauptnachteil besteht darin, dass durch die zusätzlichen Variablen, die vom eingefügten Funktionsrumpf in die Caller-Function (Aufruffunktion) kommen, die benötigten Register sich erhöhen können. Wurden bisher bereits viele Register benutzt, zwingt es die Registerverwaltung dazu Programmcode aufzuspalten, was die Performance wieder sinken lässt. Ein weiterer Nachteil ist, wie schon beim Loop Unrolling, die wachsende Codegröße, wodurch sich das Cacheverhalten verschlechtern kann. Wichtige Bereiche vom Programm könnten so aus dem Speicherbereich des Caches in den Hauptspeicher verschoben werden. Das Laden von Elementen aus dem Hauptspeicher ist nicht so effizient, wie das Laden aus dem Cache.
Die Entscheidung, ob ein Compiler ein Inlining durchführt oder nicht beruht auf Heuristiken, die abschätzen, ob die Anwendung einen Performancevorteil bringt oder nicht. In Betracht gezogen wird zum Beispiel die Größe der aufzurufenden Funktion und die Anzahl der Befehle oder Ausdrücke, die aufgewendet werden. Wird dabei ein gewisser Schwellwert überschritten, wird kein Inlining angewendet. [PL09b]

Der Codeausschnitt Listing 3.3 zeigt einen Programmaufbau, in dem kein Inlining verwendet wird. In der main-Methode werden zwei lokale Variablen `a` und `b` initialisiert und in Zeile 11 in die Methode `max` übergeben. In der max-Methode wird von `a` und `b` der größere Wert ermittelt und dann in die globale Variable `maxValue` geschrieben.
In Listing 3.4 wurde die Methode `max` komplett in die main-Methode eingebettet. Der daraufhin nicht mehr benötigte Funktionskopf wurde komplett aus dem Programm entfernt.

```
1  void max(int a, int b) {
2      if(a < b) {
3          maxValue = b;
4      } else {
5          maxValue = a;
6      }
7  }
8  int main() {
9      int a = 1;
10     int b = 3;
11     max(a, b);
12 }
```

Listing 3.3: Code ohne Inlining

```
1  int main() {
2      int a = 1;
3      int b = 3;
4      if(a < b) {
5          maxValue = b;
6      } else {
7          maxValue = a;
8      }
9  }
```

Listing 3.4: Code mit Inlining

```
1  max:
2      cmpl    %esi, %edi   ; Vgl die Werte von esi und edi
3      cmovl   %esi, %edi   ; Wenn Flag gesetzt, 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     ret                  ; return
```

Listing 3.5: Assemblercode ohne Function Inlining

Listing 3.5 zeigt, wie der Assemblercode für Listing 3.3 aussieht. Der Assemblercode wurde für die Übersichtlichkeit teilweise gekürzt und auf die wesentlichen Codeelemente beschränkt.
Die Ausgabe des Assemblerfiles zeigt wie zuerst die Werte `3` und `1` in zwei Allzweckregister geschrieben werden. Anschließend wird in Zeile 9 zum max-Label innerhalb des Assemblercodes gesprungen, in der die beiden Register zuerst verglichen werden und aufgrund dieses Vergleichs wird ein Flag gesetzt. Der cmovl-Befehl prüft, ob vorher das entsprechende Flag gesetzt wurde. Falls ja, dann wird der Wert von `esi` in `edi` geschrieben, falls nicht wird der Befehl nicht ausgeführt. Abschließend wird der Wert aus dem `edi` Register in die globale Variable geschrieben, der Aufruf der Methode beendet und zur main-Methode zurückgesprungen.

3.3 Loop Unrolling

Beim Loop Unrolling („Schleifen abrollen“) wird versucht die Schleifen zu reduzieren oder komplett aufzulösen. Beim Reduzieren wird der Schleifenrumpf so angepasst, dass dieser mehrere Kopien des Rumpfes enthält und dafür die Bedingung, wie oft die Schleifen durchlaufen wird, angepasst wird. Eine Auslösung der Schleife bedeutet, dass der Schleifenrumpf so oft ausgegeben wird, wie die ursprüngliche Schleife Durchläufe gehabt hätte.
Die Anzahl, wie oft eine Schleife abgerollt wird, nennt man den Abrollfaktor (unrolling factor). Ein Abrollen mit Faktor n erzeugt n Kopien vom Schleifenrumpf. Der beste Abrollfaktor lässt sich nur durch Messungen bestimmen [Loo].
Ein Vorteil des Loop Unrollings ist es, dass der Schleifenoverhead verringert wird. Das heißt, dass zum einen das Testen der Abbruchbedingung der Schleife minimiert wird und zum anderen werden die Sprünge reduziert, die benötigt werden, um zum Schleifenanfang zurückzuspringen. Außerdem wird durch das Abrollen die Möglichkeit für weitere Optimierungen, z.B. für das Pipelining, verbessert.
Dem gegenüber steht als Nachteil vor allem eine Vergrößerung der Codegröße. Dadurch kann eventuell das Problem auftreten, dass Teile des Programms nicht mehr aus dem Cache geladen werden können, wie schon beim Function Inlining. Ein Nachladen aus dem Hauptspeicher kann auch notwendig werden, wenn die Anzahl der Register, die durch das Abrollen der Schleife steigen, nicht mehr ausreicht. [PL09a]

Das Beispiel Listing 3.6 zeigt eine einfache for-Schleife ohne Loop Unrolling, die nach der vierten Iteration abbricht. Nach jedem Durchlauf muss die Schleifenbedingung geprüft werden, ob `i` noch kleiner 4 ist.

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

Listing 3.6: For-Schleife ohne Unrolling

Um die Sprünge und Operationen der Abbruchbedingung zu verdeutlichen, zeigt Listing 3.7 den Assemblercode von Listing 3.6. Der Assemblercode wurde mit `gcc -c -O1 loopUnrolling.c -S` erzeugt. Dabei gibt `-O1` an, dass minimale Optimierungen verwendet werden sollen. Die Ausgabe zeigt, dass so lange zu `.L2` gesprungen wird, bis das Flag durch den Vergleich in Zeile 4 nicht mehr gesetzt wird und dadurch `jne` nicht mehr ausgeführt wird.

```
.L2:
    call    printf      ; Aufruf von Printf
    addl    $1, %ebx    ; ebx += 1
    cmpl    $4, %ebx    ; Vgl. ebx und 4
    jne .L2             ; Wenn Vgl. nicht gleich springe zu L2
    ret                 ; return
```

Listing 3.7: Assemblercode für Listing 3.6

Listing 3.8 zeigt wie die aufgelöste Schleife aussieht. Alle print-Statements aus dem Schleifenrumpf werden einzeln ausgegeben. Der Schleifenkopf wurde komplett entfernt.

```
printf("Loop \n");
printf("Loop \n");
printf("Loop \n");
printf("Loop \n");
```

Listing 3.8: For-Schleife komplett abgerollt

Eine weitere Möglichkeit wäre, dass die Aufrufe reduziert werden. Listing 3.9 zeigt die Schleife mit Abrollfaktor 2. Der Schleifenrumpf wird dabei durch ein weiteres print-Statement erhöht und die Abbruchbedingung wird auf `i=+2` gesetzt.

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

Listing 3.9: For-Schleife mit Abrollfaktor 2

Der Compiler ist nicht nur in der Lage bei statischen Obergrenzen eine Schleife abzurollen, sondern auch bei dynamischen Grenzen. Das Beispiel Listing 3.6 kann dafür angepasst werden, um eine dynamische Grenze einzuführen. Im Schleifenkopf kann mit `n` ein beliebiger Wert angegeben werden.

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

Listing 3.10: For-Schleife mit dynamischer Grenze

Die Schleife kann abgerollt werden, wie das Beispiel Listing 3.11 zeigt. Dabei wird die Schleife nicht komplett abgerollt, oder innerhalb einer behandelt, sondern in zwei Schleifen aufgeteilt. Die erste Schleife initialisiert `i = 0`. Wenn `n` ungerade ist, wird der Schleifenrumpf einmal aufgerufen und die Logik innerhalb des Rumpfs ausgeführt. Wenn `n` gerade ist, wird der Rumpf nicht aufgerufen und direkt zur zweiten Schleife gesprungen.

Die zweite Schleife initialisiert `i` nicht erneut, weil die Schleife dort fortsetzt, wo die erste beendet hat. Durch den Aufbau des Rumpfes, mit zwei Zuweisungen während eines Durchlaufes, ist es möglich die Schritte zu parallelisieren. Außerdem werden die Prüfungen der Abbruchbedingung ungefähr um einen Faktor 2 reduziert.

```
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 3.11: Abgerollte Schleife für Listing 3.10

4 GCC

Moderne Compiler wie GNU Compiler Collection (GCC) bieten eine Vielzahl an verschiedene Optimierungstechniken. Das Optimieren ist eine schwierige und komplexe Angelegenheit und für den Benutzer nicht immer einfach nachzuvollziehen. Zudem hängt die Verbesserung der Performance, Codegröße und Energieeffizienz nicht nur stark vom Quellcode ab, sondern auch von den gewählten Compileroptimierungen.
Die Compiler sind standardmäßig mit verschiedenen Optimierungsleveln ausgestattet. Diese sind typischerweise: `-O1, -O2, -O3` und `-Os`. Jedes Level beinhaltet verschiedene Techniken der Optimierung, die teilweise im Konflikt zwischen Codegröße, Codequalität und Kompilierungszeit stehen. Einige Optimierungen reduzieren die Größe des Quellcodes während andere versuchen das Programm auf Schnelligkeit zu optimieren dafür aber Einbußen in Bezug auf die Größe hinnehmen. [KH08]
GCC bietet insgesamt rund 100 verschiedene Optimierungsmöglichkeiten an. Deswegen wurden durch Erfahrungen und Heuristiken Kombinationen von verschiednen Optimierungen zu Level zusammengefasst.

- `-O0`: Keine Optimierungen und die Standardeinstellung. Resultiert in der schnellsten Kompilierungszeit, bietet dafür aber keinerlei Optimierungen. Das erzeugte Programm ist dadurch wahrscheinlich größer und langsamer, als wenn Optimierungen verwendet werden. Andere Compiler optimieren auch bei der Standardeinstellung, was bei GCC nicht der fall ist.

- `-O1`: Bei diesem Level werden gängige Optimierungstechniken aktiviert. Der Zweck der ersten Optimierungsstufe liegt darin, in kurzer Zeit ein optimiertes Programm zu erzeugen. Die verwendeten Optimierungstechniken erfordern in der Regel keine größe Kompilierzeit. In dieser Stufe stehen sich manchmal gegensätzliche Ziele gegenüber. Zum einen soll die Größe des Programms verringert werden, zum anderen die Performance erhöht werden.

- `-O2`: Diese Option aktiviert weitere Techniken, die in einem Konflikt zwischen Zeit und Speicherplatz sparen stehen. Hat im Vergleich zu `-O1` eine langsamere Kompilierzeit.

- `-O3`: Verwendet zu den in `-O2` verwendeten Techniken noch aufwendigere Optimierungen. Das Flag kann schnellen Programmcode erzeugen, allerdings kann durch anwachsende Größe des Programms es dazu führen, dass die Geschwindigkeit sinkt, in dem die Größe des Caches überschritten wird. Daher ist es meistens besser `-O2` zu nutzen, um so größere Chancen zu haben die Cachegröße nicht zu überschreiten.

- `-Og`: Dieses Level sollte während des Debuggens verwendet werden. Es bietet dabei angemessene Optimierungen, die dabei trotzdem schnell kompilieren. Es ist die bessere Wahl als `-O0`, weil bei diesem Level Debug Informationen gesammelt werden, die bei `-O0` deaktiviert sind.
- `-Os`: Dieses Flag benutzt Optimierungen, die die Größe des kompilierten Programms so gering wie möglich halten soll. Das ist vor allem für System interessant, die beschränkt durch Speicher sind.
- `-Ofast`: Aktiviert alle -O3-Optimierungen. Es ermöglicht auch Optimierungen, die nicht für alle standardkonformen Programme gültig sind. [BG04]

4.1 Vergleich von verschiedenen Optimierungsleveln

Im vorherigen Abschnitt wurden einige mögliche Optimierungslevel vorgestellt. Die verschiedenen Level werden auf das Programms `partdiff.c` aus der Vorlesung Hochleistungsrechnen[1] angewendet und die Ergebnisse tabellarisch dargestellt. Dabei wird die Kompilierzeit und Laufzeit dreimal gemessen und daraus wird ein Durchschnitt ermittelt.

	-O0	-O1	-O2	-O3	-Os	-Og	-Ofast
Kompilierzeit	0m0.459s	0m0.470s	0m0.522s	0m0.551s	0m0.505s	**0m0.434s**	0m0.582s
Laufzeit	1m15.779s	1m5.978s	1m2.263s	**0m59.019s**	1m13.252s	1m2.282s	0m59.733s
Größe	7.3K	6.3K	6.7K	8.0K	**6.0K**	6.9K	8.3K

Tabelle 4.1: Verschiedene Optimierungsstufen für partdiff.c

Als Ausgangspunkt zum Vergleich dienen die Werte von `-O0`, da mit dem Level keine Optimierungen benutzt werden. Bei Verwendung von `-O1` lässt sich bei der Laufzeit schon eine kleine Verbesserung von etwa 10s erzielen, zudem wird die Größe des Programms um ungefähr 15% verkleinert. Bei `-O2` und `-O3` werden die Laufzeiten im Vergleich zu `-O1` nochmal um 2s und 6s verbessert. Dahingegen nimmt die gemessene Größe wieder zu, das sich zum Beispiel darauf zurückführen lässt, dass bei dem höheren Optimierungslevel Techniken wie Loop Unrolling aktiviert werden, die zu einer Erhöhung der Codegröße führt. Wie vorher beschrieben wird bei der Stufe `-Os` hauptsächlich auf die Optimierung der Codegröße Wert gelegt, was sich in den Ergebnissen mit dem kleinsten Wert von 6.0K widerspiegelt. Bei der Kompilierzeit wird bei `-Og` die geringste Zeit gemessen. Dies lässt sich darauf zurückführen, dass beim Debuggen-Zyklus ein schnelles Ergebnisprogramm geliefert werden soll. Für das Beispiel wird bei Verwendung von `-Ofast` für die Kompilierzeit und Laufzeit ein ähnliches Ergebnis erzielt wie für `-O3`.

[1] `https://wr.informatik.uni-hamburg.de/teaching/wintersemester_2019_2020/hochleistungsrechnen`

5 Zusammenfassung

Zusammenfassend lässt sich sagen, dass der Compiler eine Reihe von Optimierungen mit sich bringt, die sich deutlich auf die Kompilierzeit, Laufzeit und auf die Größe des kompilierten Programms auswirken können. Neben der hier vorgestellten Optimierungen gibt es noch eine große Vielzahl an weiteren Techniken. Dabei ist, auch bei den von mir erstellten Beispielen, nicht immer sofort erkenntlich, was optimiert wird. Der GCC bietet einige vorgefertigte Level, die durch Heuristiken zusammengestellt und so je nach Level einen bestimmten Bereich optimieren. Dadurch ist es für den Anwender einfach je nach Anwendungsfall einen Optimierungsgrad zu wählen.
Bei der Planungsphase eines Programms sollten Faktoren wie der Space-Time Tradeoff eingeplant werden, um so zum Beispiel einen geeigneten Algorithmus zu wählen. Je nach Anwendungsfall kann der gewählte Algorithmus einen Komplexitätsunterschied machen, auf den der Compiler keinen Einfluss nehmen kann. Aufgrund der Tatsache, dass der Compiler aktuell mit Optimierungen noch nicht auf alle Gebiete Einfluss nehmen kann, bleibt der Entwickler ein wichtiger Bestandteil beim Erstellen eines Programms.

Literaturverzeichnis

[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.* 4 2008.

[Loo] Loop Unrolling. https://www.sciencedirect.com/topics/computer-science/loop-unrolling, letzter Zugriff 26.03.2020.

[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.

[Spa] Space-Time Tradeoff. `http://people.cs.vt.edu/darendt/about/teaching/Entries/2012/1/30_Space-Time_Tradeoff.html`, letzter Zugriff 09.02.2020.

[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.