Compiler Optimization

— Seminararbeit —

Seminar
'Effiziente Programmierung in C'

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

Vorgelegt von:	Mirko Köster
E-Mail-Adresse:	0mkoeste@informatik.uni-hamburg.de
Matrikelnummer:	5616739
Studiengang:	Bsc. Informatik

Betreuer: Julian Kunkel

Hamburg, den 31.03.2013

Inhaltsverzeichnis

1 Einleitung

Dieser Seminarbericht behandelt das Thema 'Compiler Optimization'. Es wird beschrieben, was der Compiler leisten kann und wo seine Grenzen sind.

Der Compiler nimmt normalerweise keine Optimierungen vor. Das Verringert die Kompilierungszeit und hilft beim Debuggen. Beides ist hilfreich während der Entwicklungszeit. Sobald das Programm aber produktiv eingesetzt werden soll, ist es wichtiger, dass das Programm schnell läuft oder dass die ausführbare Datei möglichst klein ist oder dass das ausgeführte Programm möglichst energieeffizient ist. Diese Dinge lassen sich durch Optimierungen, die der Compiler vornimmt, erreichen.

Leider führen einige Bedingungen dazu, dass sich meist nicht alle drei Eigenschaften gleichzeitig erreichen lassen. So führen zum Beispiel einige Optimierungen, die die Geschwindigkeit erhöhen, zu einer größeren Datei. In der Regel ist diese Einschränkung aber vernachlässigbar. Auf Server-, Workstation- und Desktop-Systemen spielen die Dateigröße und Energieeffizienz oft nur eine untergeordnete Rolle. Hier kommt es auf Performance an. Im Gegensatz zum Mobile- und Embedded-Bereich. Hier geht es um Energieeffizienz und gerade bei Embedded-Systemen oft auch um möglichst kleine Dateien.

Dieser Bericht bezieht sich überwiegend auf den GNU C Compiler. Es soll hier aber überwiegend um die Konzepte gehen. Und diese treffen auch auf die anderen Compiler zu, von denen es noch weitere sehr gute gibt. Bemühen Sie das Manual zu Ihrem Compiler, wenn Sie sich für die spezifischen Optimierungsmöglichkeiten Ihres Compilers interessieren. Weiterhin konzentriert sich dieser Bericht auf die x86 Architektur. Aber auch hier gilt, dass die meisten Konzepte auch auf andere Architekturen anwendbar sind.

Bei Fachbegriffen werden, wenn sinnvoll, die englischen Bezeichnungen gewählt.

2 Automatic Optimization

2.1 Erläuterung

Wie in der Einleitung erwähnt, nimmt der Compiler standardmäßig keine Optimierungen vor. Das ist wichtig, damit sich das Programm vom Entwickler im Fehlerfall vernünftig debuggen lässt. Mit Optimierungen wäre dieser Vorgang schwieriger, da einige Optimierungen Befehle für eine effizientere Ausführung umsortieren. Dann entspräche die Reihenfolge der Anweisungen im Quellcode nicht denen im Maschinencode.

Zur Optimierung des Programms für den Produnktiveinsatz werden die verschiedensten Optimierungvarianten eingesetzt. Dabei bietet der GNU C Compiler verschiedene Optimierungs-Level und -Flags an, bei denen der Compiler die Optimierungen automatisch anwendet. Diese automatischen Optimierungen verändern nicht das Ergebnis der Berechnungen (Ausnahmen siehe Kapitel 5) und können die Bereiche Ausführungsgeschwindigkeit, Dateigröße oder Energieeffizienz beeinflussen.

Zur Durchführung der Optimierungen analysiert der Compiler den Source Code, wobei dieser striktere Regeln annimmt als von der C-Sprache vorgeschrieben. Bei der Analyse trifft der Compiler Annahmen, die bewiesen werden müssen. Anschließend kann der Compiler die Optimierungen ausführen, sofern die notwendigen Bedingungen erfüllt sind.

Die verschiedenen Level fassen verscheidene Optimierungen zusammen. Das Standard-Level ist 0; wie schon beschrieben, werden hier keine Optimierungen vorgenommen. Je höher das Level, desto stärker optimiert der Compiler. Die meisten Optimierungen bietet das Level 3. Eine Übersicht bieten die Tabellen 2.1 bis 2.5. Erläuterungen unter `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`.

Dabei sind manche Optimierungen sehr zeitaufwendig, da einige Optimierungsprobleme NP-schwer oder einige Entscheidungsprobleme sogar unentscheidbar sind. In diesen Fällen geht der Compiler einen Kompromiss ein: Dieser liefert dann zwar nicht zwingend das optimale aber immerhin ein gutes Resultat.

-fauto-inc-dec	-fipa-pure-const	-ftree-dominator-opts
-fcompare-elim	-fipa-profile	-ftree-dse
-fcprop-registers	-fipa-reference	-ftree-forwprop
-fdce	-fmerge-constants	-ftree-fre
-fdce	-fsplit-wide-types	-ftree-phiprop
-fdefer-pop	-ftree-bit-ccp	-ftree-slsr
-fdelayed-branch	-ftree-builtin-call-dce	-ftree-sra
-fdse	-ftree-ccp	-ftree-pta
-fguess-branch-probability	-ftree-ch	-ftree-ter
-fif-conversion2	-ftree-copyrename	-funit-at-a-time
-fif-conversion	-ftree-dce	

Tabelle 2.1: Optimierungs-Flags vom Optimierungslevel -O1

-fthread-jumps	-fipa-sra
-falign-functions	-foptimize-sibling-calls
-falign-jumps	-fpartial-inlining
-falign-loops	-fpeephole2
-falign-labels	-fregmove
-fcaller-saves	-freorder-blocks
-fcrossjumping	-freorder-functions
-fcse-follow-jumps	-frerun-cse-after-loop
-fcse-skip-blocks	-fsched-interblock
-fdelete-null-pointer-checks	-fsched-spec
-fdevirtualize	-fschedule-insns
-fexpensive-optimizations	-fschedule-insns2
-fgcse	-fstrict-aliasing
-fgcse-lm	-fstrict-overflow
-fhoist-adjacent-loads	-ftree-switch-conversion
-finline-small-functions	-ftree-tail-merge
-findirect-inlining	-ftree-pre
	-ftree-vrp

Tabelle 2.2: Optimierungs-Flags vom Optimierungslevel -O2 (enthält alle Optimierungen von -O1)

-finline-functions	-ftree-vectorize
-funswitch-loops	-fvect-cost-model
-fpredictive-commoning	-ftree-partial-pre
-fgcse-after-reload	-fipa-cp-clone

Tabelle 2.3: Optimierungs-Flags vom Optimierungslevel -O3 (enthält alle Optimierungen von -O2)

-falign-functions	-freorder-blocks
-falign-jumps	-freorder-blocks-and-partition
-falign-loops	-fprefetch-loop-arrays
-falign-labels	-ftree-vect-loop-version

Tabelle 2.4: Optimierungs-Flags vom Optimierungslevel -Os (Dateigrößen-Optimierungen)

Dieses Level reduziert die Compile-Zeit und sorgt dafür, dass sich das Programm beim Debugging wie erwartet verhält

Tabelle 2.5: Optimierungslevel -O0 (default)

2.2 Architecture Independent

Optimierungen, die nicht von der Ziel-Architektur abhängen, nennt man Architektur-unabhängig. Diese können immer dann angewendet werden, wenn die Vorbedingungen der jeweiligen Optimierung erfüllt sind.

Im folgenden werden einige Optimierungen vorgestellt

2.2.1 Loop Invariant Code Motion

Diese Optimierung bewegt Code aus Schleifen heraus, wenn dieser unabhängig (invariant) von der Schleife ist. In diesem Beispiel erfolgt ein schreibender Zugriff auf die Variable x in jedem Schleifendurchlauf. Und zwar jedes Mal mit dem selben Wert. Gelesen wird die Variable innerhalb der Schleife jedoch nicht. Der Compiler erkennt dies und bewegt die Zuweisung vor die Schleife, so dass der schreibende Zugriff nur noch einmal geschieht.

```
int sum=0, x;
for(int i = 0; i < n; i++) {
  sum += i;
  x = 5;
}
```

Listing 2.1: Loop Invariant Code Motion - unoptimiert

```
int sum=0, x = 5;
for(int i = 0; i < n; i++) {
  sum += i;
}
```

Listing 2.2: Loop Invariant Code Motion - optimiert

2.2.2 Const Propagation (with Loop Optimization)

Der Compiler kann gewisse Optimierungen vornehmen, wenn Variablen konstant sind. In diesem Beispiel erkennt der Compiler, dass die Variable N den konstanten Wert 10 hat und nur an einer Stelle verwendet wird. Daher kann das N im Schleifenkopf ersetzt werden. Weiterhin steht das Ergebnis der Berechnung von sum schon zur Compilezeit fest. Dadurch kann die komplette Schleife gelöscht werden und für sum der Wert 45 eingesetzt werden:

```
int N = 10, sum = 0;
for(int i = 0; i < N; i++)
  sum += i;

printf("sum = %d\n", sum);
```

Listing 2.3: Const Propagation - unoptimiert

```
int sum = 0;
for(int i = 0; i < 10; i++)
  sum += i;

printf("sum = %d\n", sum);
```

Listing 2.4: Const Propagation - optimiert1

```
printf("sum = %d\n", 45);
```

Listing 2.5: Const Propagation - optimiert2

2.2.3 Dead Code Elimination

Der Compiler kann zeigen, dass bestimmte Codeteile nicht erreichbar sind. Diese werden durch diese Optimierung weggelassen. In den folgenden Listings erkennt der Compiler, das der else-Zweig nicht erreichbar ist, da die Variable x unsigned ist und somit nicht negativ sein kann. Weiterhin ist die Bedingung im else-Zweig immer wahr, weshalb die Prüfung der Bedingung entfallen kann. Übrig bleibt nach der Optimierung nur der Aufruf von printf mit dem Parameter foobar() anstatt x:

```
unsigned int x = foobar();
if(x < 0) {
  printf("never executed\n");
} else {
  printf("x: %u\n", x);
}
```

Listing 2.6: Dead Code Elimination - unoptimiert

```
unsigned int x = foobar();
if(x >= 0) {
  printf("x: %u\n", x);
}
```

Listing 2.7: Dead Code Elimination - optimiert1

```
printf("x: %u\n", foobar());
```

Listing 2.8: Dead Code Elimination - optimiert2

2.2.4 Common Subexpression Elimination

Wenn der Compiler feststellt, dass manche Berechnungen oder Zugriffe mehrfach erfolgen, können diese eleminiert werden. Dazu fügt der Compiler in diesem Bespiel eine weitere Variable ein, in der das gemeinsam benutzte Ergebnis gespeichert wird. Somit erfolgt der Zugriff hier nur noch einmal anstatt zweimal:

```
void foo(int *a, int n) {
  for(int i = 0; i < n; i++)
    a[i] += a[i]/n + a[i]*n;
}
```

Listing 2.9: Common Subexpression Elimination - unoptimiert

```
void foo(int *a, int n) {
  int temp;
  for(int i = 0; i < n; i++)
    temp = a[i]
    a[i] += temp/n + temp*n;
}
```

Listing 2.10: Common Subexpression Elimination - optimiert

2.2.5 Interprocedural Optimization

Der Compiler kann analysieren, wie verschiedene Funktionen zusammenarbeiten. Anschließend kann er die Funktionsaufrufe gegebenfalls so optimieren, dass Parameter nicht wie üblich über den Stack übergeben werden, sondern in den Registern. Dazu muss er sowohl die aufrufende als auch die aufgerufene Funktion (caller und callee) modifizieren. Das reduziert den call/return overhead.

2.2.6 Inlining

Diese Optimierung reduziert Funktionsaufrufe, indem an deren Stelle der Funktions-Body geschrieben wird. Dazu gab es einen eigenen Vortrag in diesem Seminar. Daher an dieser Stelle nur ein kurzes Beispiel:

```
int foo(int a) {
  return a * (a+1);
}
...
int a[5];
for(int i = 0; i < 5; i++)
  a[i] = foo(i);
```

Listing 2.11: Inlining - unoptimiert

```
int a[5];

a[0] = 0 * 1;
a[1] = 1 * 2;
a[2] = 2 * 3;
a[3] = 3 * 4;
a[4] = 4 * 5;
```

Listing 2.12: Inlining - optimiert

2.2.7 Interprocedural Constant Propagation

Diese Optimierung schaut sich Funktionsaufrufe an. Wenn das Ergebnis eines Aufrufs schon zur Compilezeit feststeht, kann der Aufruf durch das Ergebnis ersetzt werden:

```
static int square(int x) {
  return x*x;
}

printf("5^2=%d\n",square(5));
```

Listing 2.13: Interprocedural Constant Propagation - unoptimiert

```
static int square(int x) {
  return x*x;
}

printf("5^2 = %d\n", 25);
```

Listing 2.14: Interprocedural Constant Propagation - optimiert

2.3 Architecture Dependent

Architektur-abhängige Optimierungen sind Optimierungen, die sich auf die Zielplattform beziehen. Dazu muss dem Compiler die Zielarchitektur mitgeteilt werden. Dann kann dieser die spezifischen Instruktionen der Zielarchitektur ausnutzen, zum Beispiel Befehlssatzerweiterungen von neueren CPU-Generationen. Aber auch die Anzahl an (special purpose) Registern sowie die Typen der Cache-Level und deren Größe wird durch die Architektur festgelegt.

2.3.1 Instruction Set

Befehlssatzerweiterungen sorgen dafür, dass moderne Prozessoren bestimmte Operationen effizienter ausführen können als mit dem Standard-x86-Befehlssatz. Damit Programme auf allen Computern laufen, müssen diese allerdings den kleinsten gemeinsamen Nenner unterstützen und dürfen keine Befehlssatzerweiterungen wie SSE2 nutzen. Das erhöht zwar die Kompatibilität, geht auf modernen Computern aber zu Lasten der Performance.

Es folgt eine Übersicht über die Historie des x86-Befehlssatz.

Überblick über die Geschichte von (x86-) Befehlssätzen und deren Erweiterungen

- 1985 x86 32bit
- 1989 x87 FPU (Co-Processor)
- 1993 MMX

- 1997 SSE, 3DNow!
- 2000 SSE2
- 2003 x86-64 64bit
- 2004 SSE3
- 2007 SSE4a
- 2011 SSE5/AVX
- 2013 AVX2, FMA3

-mtune & -march

Die Option mtune zusammen mit der Angabe einer Zielarchitektur sorgt dafür, dass der Compiler den Code zwar für diese Architektur optimiert, das Programm aber auch auf anderen Architekturen noch ausführbar bleibt. Auf diesen läuft das Programm aber unter Umständen etwas langsamer, als wenn auf diese Optimierung verzichtet würde. Diese Optimierung ist immer dann sinnvoll, wenn das Programm überwiegend auf moderner Hardware ausgeführt wird, auf Abwärtskompatibilität aber nicht vollständig verzichtet werden kann. In diesem Fall müssen Vor- und Nachteile gegeneinander abgewogen werden.

Die Option march optimiert den Code hingegen so stark wie möglich für die Zielarchitektur. Das sorgt oft für mehr Performance; das Programm läuft jedoch unter Umständen nicht auf anderen Architekturen. Diese Option ist sinnvoll, wenn man genau weiß, auf welcher Hardware das Programm eingesetzt wird.

Beispiele für Zielarchitekturen

- i386
- pentium
- corei7
- amdfam10

2.3.2 32 vs 64 Bit

Wenn man ein Programm für die x86-64 Architektur (AMD64 bzw. EM64T) kompiliert, kann der Compiler auf Befehlssatzerweiterungen wie MMX, SSE und SSE2 zurückgreifen, da die Prozessoren dieser Architektur diese unterstützen (siehe 2.3.1). Weiterhin stehen hier doppelt so viele general purpose Register zur Verfügung (16 Register bei 64 Bit im Vergleich zu 8 Register bei 32 Bit). Auch ist der virtuelle Adressraum weitaus größer (wenigstens 48 Bit, was 256 TiB enspricht).

2.3.3 Automatic Vectorization

Die meisten Befehlssatzerweiterungen bringen Unterstützung von SIMD (Single Instruction Multiple Data) mit. Diese Befehle ermöglichen es, einen Befehl gleichzeitig auf mehrere Daten anzuwenden. Das erhöht die Ausführgeschwindigkeit.

Das folgende Beispiel zeigt, wann der Compiler Gebrauch machen kann von den AVX-Erweiterungen. Die SIMD-Register sind hier 256 Bit breit. Somit passen 8 Float-Werte gleichzeitig in ein Register. Nun kann der Prozessor 8 Berechnungen parallel ausführen. In diesem Fall reduziert sich der Aufwand von 128 sequenziellen auf 16 * 8 parallele Berechnungen.

```
1 float a[128];
  ...
3 for(int i=0; i < 128; i++)
    a[i] *= 2.5f;
```

Listing 2.15: Automatic Vectorization - source (sequenziell)

3 Profile Guided Optimization

3.1 Erläuterung

Es ist möglich, den Compiler das typische Verhalten der Anwendung 'lernen' zu lassen. Dazu wird das Programm entsprechend vorbereitet und dann bei einer typischen Benutzung profiliert. Während der Profilierung werden die anfallenden Profilierungsdaten auf ein Speichermedium geschrieben. Diese Profilierungsdaten kann der Compiler anschließend nutzen, um das Programm zu optimieren.

3.2 Optimierungen

3.2.1 Function Ordering

Wenn der Compiler feststellt, dass bestimmte Funktionen häufig kurz nacheinander aufgerufen werden, kann er diese im Maschinencode umsortieren. Das führt dazu, dass diese Funktionen häufiger zusammen im Instruction Cache liegen und nicht wieder erst aus dem Hauptspeicher geladen werden müssen.

Listing 3.1 und 3.2 zeigen dies. Angenommen, die Funktionen foo() und bar() werden häfig zusammen aufgerufen. Dann kann der Compiler dies den Profilierungsdaten entnehmen und diese effektiv umsortieren.

```
int foo() {
  ... //several lines of code
}
float someFunction() {
  ... //several lines of code
}
... //more functions
int bar() {
  ... //several lines of code
}
```

Listing 3.1: Function Ordering - unoptimiert

```
int foo() {
  ... //several lines of code
}
int bar() {
  ... //several lines of code
}
float someFunction() {
  ... //several lines of code
}
... //more functions
```

Listing 3.2: Function Ordering - optimiert

3.2.2 Basic Block Ordering

Dieses Verfahren geht ähnlich vor wie das Function Ordering. Das Ziel ist das gleiche: Werden Blöcke häufig hintereinander ausgeführt, kann eine Umsortierung zur Erhöhung der Instruction Cache Hit Rate führen.

3.2.3 Switch Statement Optimization

Diese Optimierung sortiert die case-Zeilen in switch-case-Blöcken entsprechend der Häufigkeit der Ausführung, die die Profilierung ergeben hat. Das führt dazu, dass weniger cases geprüft werden müssen. Die nachfolgenden Listings zeigen, wie die optimierte Reihenfolge aussähe, wenn das Profiling ergeben hat, dass 'constant3' häufiger auftritt als 'constant1', welches wiederum häufiger auftritt als 'constant2':

```
switch(expression)
{
   case constant1:
       statements; break;
   case constant2:
       statements; break;
   case constant3:
       statements; break;
   default:
       statements;
}
```

Listing 3.3: Switch Statement Optimization - unoptimiert

```
switch(expression)
{
   case constant3:
       statements; break;
   case constant1:
       statements; break;
   case constant2:
       statements; break;
   default:
       statements;
}
```

Listing 3.4: Switch Statement Optimization - optimiert

3.2.4 Improved Register Allocation

Die Belegung der Register mit Variablen zu optimieren, ist ein NP-schweres Problem. Wenn man Profiling einsetzt, wird diese Optimierung stark vereinfacht, da der Compiler aus den Profilierungsdaten auslesen kann, wann und in welcher Reihenfolge die Zugriffe auf die Variablen erfolgte.

4 Aiding Optimizations

4.1 Erläuterung

Der Compiler setzt die Regeln des C-Standards um. Daher muss dieser in seinen Annahmen oft sehr konservativ sein. Das führt dazu, dass einige Optimierungen nicht durchgeführt werden, obwohl diese möglich wären. Das passiert immer dann, wenn der Compiler nicht beweisen kann, dass die Vorbedingungen tatsächlich erfüllt sind. Hätte der Compiler mehr wissen über den Code, könnte dieser mehr und bessere Optimierungen durchführen.

Daher ist es sinnvoll, wenn der Programmierer an Stellen, an denen das möglich ist, sein Wissen über den Quellcode nutzt und dem Compiler hilft.

4.2 Optimierungen

4.2.1 Data Layout

Oft kann Speicher von Datenstrukturen besser ausgenutzt werden (was den Speicherplatz und das Caching betrifft), wenn diese Strukturen an den Speicher angepasst werden. Dazu ein Beispiel: In Listing 4.1 benötigt jeder Char auf einem 32-Bit System 32 Bit. In Listing 4.2 kann der Compiler die Chars zusammenfassen und diese belegen somit auf dem gleichen System zusammen nur 32 Bit anstatt 96 Bit wie im unoptimierten Beispiel.

```
struct foo {
  char a;
  float x[8];
  char b;
  float y[8];
  char c;
  float z[8];
};
```

Listing 4.1: Data Layout - unoptimiert

```
struct foo {
  float x[8];
  float y[8];
  float z[8];
  char a;
  char b;
  char c;
```

```
8  };
```

Listing 4.2: Data Layout - optimiert

4.2.2 Pragma Vector Aligned

Einige Architekturen führen manche Operationen schneller aus, wenn garantiert ist, dass die Daten an den spezifischen Speichergrenzen ausgerichtet sind. Sofern der Programmierer das garantieren kann, kann er dies dem Compiler mit der Compileranweisung '#pragma vector aligned' mitteilen:

```
float a[128];
2 ...
#pragma vector aligned
4 for(int i=0; i < 128; i++)
  a[i] *= 2.5f;
```

Listing 4.3: Pragma Vector Aligned - optimiert

Es gibt weiterhin die Anweisungen 'unalligned' und 'always'.

5 'Safe' / 'Unsafe' Optimizations

5.1 Erläuterung

Der Compiler geht standardmäßig sehr konservativ vor. Die meisten Optimierungen (insbesondere die aus den -O[level] Optionen) ändern nicht das Ergebnis von Berechnungen.

An manchen Stellen kommt es aber nicht auf die Genauigkeit an, die eine Berechnung liefert. Für diese Fälle, in denen die Geschwindigkeit im Vordergrund steht, bietet der Compiler weitere Optimierungen. Diese können das Resultat von Berechnungen leicht verfälschen (z.B. Genauigkeit bei Gleitkommaberechnungen), sind dafür aber potenziell schneller. Ein Beispiel für diese Sorte von Optimierungen ist das Compiler Flag '-ffast-math'.

Unsichere Optimierungen sollten nur mit großer Vorsicht und bei echtem Bedarf eingesetzt werden.

6 OpenMP

6.1 Erläuterung

OpenMP ist ein Standard, der es ermöglicht, Multi-Core-Systeme (Shared-Memory) besser auszunutzen. Unterstützt werden offiziell die Sprachen C/C++ und Fortran. OpenMP stellt ein API (Shared-Memory Multithreading Programming Interface) und Bibliotheken bereit.

Ein (stark vereinfachter) Workflow mit OpenMP sieht folgendermaßen aus: Der Benutzer gibt beim Kompilieren an, dass er OpenMP nutzen will und fügt an den entsprechenden Stellen im Quellcode einige spezielle Kommentare ein. Diese teilen dem Compiler mit, wie diese Stellen parallelisiert werden sollen.

6.2 Optimierungen

6.2.1 Anwendung

Die Listings 6.1 und 6.2 zeigen Beispielhaft die Anwendung von OpenMP. Die Funktion foobar() füllt ein Array der Länge n jeweils mit dem zweifachen der Indexposition. Durch Einfügen des Kommentars '#pragma omp parallel for' direkt vor der for-Schleife weiß der OpenMP-fähige Compiler, dass diese Schleife parallelisiert werden soll. Die Parallelisierung nimmt der Compiler nun selbständig vor.

```
1 void foobar(int *a, int n) {
    for (int i = 0; i < n; i++)
3     a[i] = 2 * i;
}
```

Listing 6.1: OpenMP - unoptimiert

```
  void foobar(int *a, int n) {
2   #pragma omp parallel for
    for (int i = 0; i < n; i++)
4     a[i] = 2 * i;
}
```

Listing 6.2: OpenMP - optimiert

7 Fazit

Dieser Bericht hat eine Reihe von Optimierungen aufgezeigt, die der Compiler vornehmen kann. Die Liste der Beispiele ist dabei natürlich nicht erschöpfend. Es gibt noch zahllose weitere. Die Intention ist es auch vielmehr, den Leser dafür zu sensibilisieren, an welchen Stellen sich eine manuelle Optimierung lohnt und welche Optimierungen man lieber dem Compiler überlässt.

Oder wie Donald E. Knuth es ausdrückte:
“Premature Optimization is the root of all evil”

Es ist sinnvoller, gut lesbaren Quellcode zu schreiben. Das minimiert die Fehlerwahrscheinlichkeit. Von Hand optimieren sollte man nur an Stellen, die nachweislich einen Performance-Engpass darstellen. An allen anderen Stellen ist es besser, dem Compiler die Arbeit zu überlassen und diesen darin zu unterstützen, indem man z.B. die Zielarchitektur angibt oder Schlüsselworte wie ’restrict’, ’static’ und ’volatile’ (siehe entsprechende Vorträge in diesem Seminar) oder die Optimierungen aus dem Kapitel 4 nutzt. Das hilft dem Compiler, da er dann an diesen Stellen keine Annahmen treffen und diese beweisen muss.

Zusammengefasst lautet das Fazit:
“readability over manual optimization”

Quellen

- http://en.wikipedia.org/wiki/Optimizing_compiler
- http://en.wikipedia.org/wiki/Program_optimization
- http://smackerelofopinion.blogspot.de/2012/09/striving-for-better-code-quality
  html
- http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Optimize-Options.html
- http://gcc.gnu.org/onlinedocs/gcc/i386-and-x86_002d64-Options.html
- http://en.wikipedia.org/wiki/X86-64
- http://www.embedded.com/design/mcus-processors-and-socs/4008892/Tuning-C-C--c
- http://codingfreak.blogspot.com/2008/02/compilation-process-in-gcc.html
- http://en.wikipedia.org/wiki/Basic_block
- http://openmp.org/wp/

Tabellenverzeichnis

Listingverzeichnis