Wie man kleine und schnelle Software schreibt

Felix von Leitner
felix-dietlibc@fefe.de

März 2002

Zusammenfassung

Software wird schneller größer und langsamer als der Speicher billiger und die Prozessoren schneller werden. Gleichzeitig sind gute Programmierer rar und teuer, so daß wir im Moment Müll-Software mit Hardware kompensieren. Im Moment sind die Rechner schnell genug, um Exchange, SAP oder den MS SQL Server laufen zu lassen. Das hat die IT-Branche in die Krise gestürzt.

Einleitung

An der Uni lehrt man: Performance kann man mit mehr Code kaufen (Inline-Expansion, Makros, Loop Unrolling, unglaublich komplexe Monster-Algorithmen, ...)

In der Praxis ist mehr Code fast immer schlecht: er verstopft den CPU-Cache und den Speicherbus, verschwendet CPU-Zyklen, macht eine Sicherheitsüberprüfung des Quellcodes schwieriger, man kann ihm nicht so leicht folgen, die Wartbarkeit leidet, und - last but not least - braucht mehr Code auch mehr Dokumentation.

Schlipskompatible Kurzzusammenfassung

Indirektion erzeugt Bloat und tötet die Performance!

Entferne überflüssige Indirektionen!

Kenne den Feind!

Die Hauptursachen für großen Code sind:

1. Nachlässigkeit
2. Falscher Ehrgeiz („*das Programm ist schon fast perfekt; es fehlt nur noch eine Lateinschnittstelle, eine Tabellenkalkulation und eine Umrechnungssoftware für den Maya-Kalender*“)
3. Versagen im Management (an vier Stellen werden Konfigurationsdateien manuell geparsed)
4. Klopapier-Programmierung (4 weiche Wrapper-Lagen verdecken den Gestank darunter)

Feind: Nachlässigkeit

Es ist oft mehr Aufwand, alten Code aufzuräumen als ihn „mal eben“ neu zu schreiben. Häufige Probleme sind:

1. Beim Code-Merge werden Überlappungen nicht entfernt.
2. Debug-Code in Produktions-Programmen
3. Generische statt spezifische APIs (z.B. `fprintf` statt `write`)
4. Man muß einen Socket öffnen, fragt google, findet eine allumfassende Godzilla-Bibliothek wie ACE, und linkt dann halt die 5MB dazu. „*Mußte schnell fertig werden...*“

Feind: Falscher Ehrgeiz

Besonders junge und unerfahrene Programmierer wollen gerne alles richtig machen und streben nach dem endgültigen, perfekten, allumfassenden API.

Objektorientierte Programme tendieren zu einem Haufen unbenutzter Methoden und Routinen „just in case“. Beispiel: C++ iostream und libg++.

Fakt: Die Erfahrung sagt, daß dein API nicht perfekt sein wird. Gewöhn dich daran.

Fakt: Aufgebohrte APIs sind gewöhnlich außerordentlich schmerzhaft zu benutzen.

Feind: Versagen im Management

Beim Code-Merge muß man Zeit mit der Suche nach Code zu verbringen, der das gleiche Problem löst. Die Anzahl der Instanzen muß dann auf eins reduziert werden (häufige Beispiele: verkettete Listen, Stringroutinen).

Wenn mehr als eine Person an einem Projekt arbeitet, dann gilt jeder CVS Checkin als Code-Merge in diesem Sinne!

Natürlich sollte man Redundanzen am besten gar nicht erst entstehen lassen und sie nicht nur nachträglich wegoperieren.

Es ist kontraproduktiv, Programmierer nach Codezeilen zu bezahlen (duh!).

Feind: Klopapier-Programmierung

Dieses Problem tritt besonders häufig in C++-Code auf.

Klassische Ausrede: Portabilität.

Klassisches Beispiel: Thread Libraries wrappen libpthread, die clone wrappen, welches ein libc-Wrapper um den Syscall ist.

Noch mehr klassische Beispiele: KDE wrappt qt, wrappt xlib. gtk– wrappt gtk++, wrappt xlib. xlib wrappt die libc, die syscalls wrappt. Wrappen macht nur bei Java Sinn, weil native Java-Implementationen noch mal zwei Größenordnungen langsamer sind als zwanzig Wrapper. :-(

Gerüchten zufolge soll es sogar ein Qt-Emulationslayer um gtk+ geben!

Feind: XML

XML ist ein weiteres großes Layer um alles herum. Mit XML kann man zwar einfach Daten ausgeben (im krassen Gegensatz zu DER, einer ASN.1 Kodierung), aber einen validierenden Parser zu schreiben ist sehr untrivial.

Besonders breitenwirkend katastrophal ist XML in Kombination mit Java, wo ein Apache auf www.ccc.de mit der Referenz-Implementation von XSLT pro Seite **eine Minute** gerendert hat. Die C++-Variante braucht „nur“ noch 5 Sekunden.

Dem Trend, Benutzeroberflächen dynamisch per XML zu speichern, muß unbedingt Widerstand geleistet werden!

Feind: Dynamisches Symbolauflösung zur Laufzeit

Late Binding kann man sehr effizient machen, aber normalerweise wird es sehr schlecht gelöst.

Beispiele: Signals in Qt, Hashtabellen für symbolischen Attribut-Lookup in gtk+, late binding in DYLAN.

Was kann ich tun?

1. Nur generalisieren, wenn es nichts kostet
2. Erst spezifische Interfaces schreiben
3. Spezifische Routinen vor generischen bevorzugen
4. Sei dir bewußt, welchen Bloat du erzeugst!
5. *your idea here*

Nur generalisieren, wenn es nichts kostet

Eine interne Funktion der diet libc:

```
int __ltostr(char *s, int size, unsigned long i,
             int base, char UpCase);
unsigned int fmt_ulong(char *dest,unsigned long src);
```

Ein guter Test, ob das Interface zu generisch ist: wenn man die Argumentnamen wegläßt, ist dann immer noch klar, was da übergeben wird?

```
int __ltostr(char *, int, unsigned long, int, char);
int fmt_ulong(char *,unsigned long);
```

Erst spezifische Interfaces schreiben

- Um wiederverwendbaren Code zu schreiben, definiert man ein sehr breites und generisches API
- Leider hat man anfangs das Problem noch nicht gut verstanden
- Nicht gebrauchte Funktionalität wird implementiert, Zeit geht verloren
- Am Ende kann man das API nicht mehr ändern und wird den Bloat nicht los
- Außerdem könnte sich jemand irgendwo auf das breite API verlassen

Wenn du mal ein API erweitern mußt

Mache nicht bestehende Funktionen mächtiger!

Füge lieber eine neue Funktion hinzu!

Schmeißt denn nicht der Compiler unbenutzten Code weg?

```
$ cat > t.c
int foo(int bar) { return bar+2; }
int bar(int baz) { return baz*23; }
int main() { printf("%d\n",bar(17)); }
$ gcc -c t.c
$ nm t.o
0000000000000014 T bar
0000000000000000 T foo
0000000000000000 t gcc2_compiled.
0000000000000034 T main
                 U printf
$
```

Schmeißt denn nicht der Linker unbenutzten Code weg?

```
$ cat a.c
char magic[]="bloat";
$ cat b.c
int main() { puts("hello, world"); }
$ gcc -o b a.c b.c
$ grep bloat b
Binary file b matches
$ gcc -o b b.c
$ grep bloat b
$
```

Was ist, wenn ich Shared Libraries verwende?

Shared Libraries sind die eine Technologie, die am stärksten für den Bloat in der Software-Industrie verantwortlich ist. Denn **der Programmierer sieht nicht mehr die tatsächliche Code-Größe.**

Der ganze Bloat ist immer noch da, und er verschwendet Plattenplatz!

Moderne Betriebssysteme unterstützten Paging und On-Demand Loading, d.h. nur die wirklich benutzten Seiten werden in den Speicher geladen. Das beruhigt bei den Bloat-Tätern das Gewissen, aber in der Realität wird immer noch echt viel RAM verschwendet.

Die Page-Granularität ist 4k. Kleinere Speicherteile können nicht weggemappt werden, wenn sie nicht gebraucht werden. Das Datensegment ist zusammenhängend!

Und wenn ich statisch linke?

ld ignoriert unreferenzierte Objektdateien aus statischen Bibliotheken beim Linken.

Das ist auch der Grund, wieso die diet libc anfangs nur als statische Bibliothek vorliegt.

Spezifische Routinen vor generischen bevorzugen

Dieser Rat bezieht sich nur auf Situationen, wo man nur einen Teil der Funktionalität benutzt!

Für temporären Speicherbedarf benutze `alloca` und nicht `malloc`.

Benutze `strchr` statt `strstr`.

Benutze `qsort` und `bsearch` und keine AVL-Bäume für alles bis auf sehr dynamische oder große Datenbestände.

Optimiere für den Normalfall, nicht den Worst Case

Arrays sind großartig! Der Code ist leicht verständlich und weniger fehleranfällig als bei Listen und Bäumen.

Arrays kann man prima mit `realloc` vergrößern, man kann ein Array traversieren, ohne den Cache zu trashen, man kann alle Elemente auf einmal freigeben (außer sie enthalten Zeiger, natürlich).

Sei dir bewußt, welchen Bloat du erzeugst!

```
$ cat > t.cc
#include <string>
#include <iostream>
main() {
  string s="foo";
  cout << s << endl;
}
$ g++ -static -s -o t t.cc
$ du -H t
387k   t
$
```

Behandle malloc wie eine teure Operation

Malloc und co sind potentiell sehr teuer.

```
$ cat > mymalloc.c
#include <stdio.h>
static int mallocs=0;
void sayit() { fprintf(stderr,"%d mallocs\n",mallocs); }
int malloc(unsigned long size) {
  if (!mallocs++) atexit(sayit);
  return __libc_malloc(size);
}
$ gcc -shared -o mymalloc.so mymalloc.c
$ LD_PRELOAD=$PWD/mymalloc.so grip
20794 mallocs
```

Gute API vs. Schlechte API

Schlecht:

```
int fd=open("foo",O_WRONLY|O_CREAT,0600);
```

Gut:

```
int fd=open_write("foo");
```

Gute API vs. Schlechte API

Schlecht:

```
int i;
char buf[100];
sprintf(buf,"the number is %d",i);
```

Gut:

```
stralloc s={0};
stralloc_copys(&s,"the number is ");
stralloc_catuint(&s,i);
```

Messen, nicht spekulieren!

Speicherplatz für Performance zu opfern ist nicht grundsätzlich schlecht!

Finde mit `gprof` die für die Performance kritischen Code-Stellen. Da (und nur da) benutze Tradeoffs, um Speicher für Performance zu opfern.

Wenig bekannte `gcc`-Option: `-Os` ist wie `-O2`, aber ohne die Trade-Offs, die Speicherplatz für Performance opfern.

Andere gcc-Optionen: `-fomit-frame-pointer -mcpu=i386`

Benutze `nm` und `objdump`, um die Symboltabelle anzuschauen.

Inline-Funktionen

```
$ cat > t.c
inline int foo() { return 3; }
int bar() { return foo(); }
$ gcc -O2 -c t.c
$ nm t.o
00000000 T bar
0000000c T foo
00000000 t gcc2_compiled.
$
```

Static Inline

```
$ cat > t.c
static inline int foo() { return 3; }
int bar() { return foo(); }
$ gcc -O2 -c t.c
$ nm t.o
00000000 T bar
00000000 t gcc2_compiled.
$
```

Lerne den Umgang mit `static` zum Platz sparen (und praktischer Umweltschutz beim Namespace)!

Alloca

`alloca` alloziert Platz **auf dem Stack.** Der Platz wird automatisch freigegeben, wenn das Programm die Funktion verläßt (d.h. man muß es nicht manuell freigeben).

`malloc` muß umfangreiche Datenstrukturen verwalten und Locks pflegen (für pthread-Kompatibilität). `alloca` zieht eine Konstante von einem Register ab.

In C++ versteckt sich malloc() häufig hinter dem `new`-Operator. C++ ist sehr gefährlich, weil es Komplexität und Bloat vor dem Programmierer verbirgt.

Introducing libowfat

Warum der Name? Weil man es mit „`-lowfat`“ dazu linkt!

libowfat ist eine GPL-Reimplementation von Dan Bernsteins internen Hilfsroutinen.

Warum will man das haben? Weil seine APIs exemplarisch für gutes API-Design sind.

libowfat: byte.a

Liefert den index zurück, keinen Pointer. Erstes Argument ist immer der Speicherbereich, auf dem man arbeitet. Der Rest sind die Parameter.

```
int byte_chr(void*,int,char);
int byte_rchr(void*,int,char);
void byte_copy(void* out, int, void* in);
void byte_copyr(void* out, int, void* in);
int byte_diff(void*, int, void*);
void byte_zero(void*, int);
#define byte_equal(s,t) (!byte_diff((s),(t)))
```

libowfat: str.a

Was haben die Leute bloß geraucht, die `strcpy` und `strcat` Zeiger zurückliefern lassen?!

```
int str_copy(char* out,char* in);
int str_diff(char*,char*);
int str_diffn(char*,char*,int);
int str_len(char*);
int str_chr(char*,char);
int str_rchr(char*,char);
int str_start(char*,char*);
#define str_equal(s,t) (!str_diff((s),(t)))
```

libowfat: fmt.a

```
int fmt_long(char*,signed long);
int fmt_ulong(char*,unsigned long);
int fmt_xlong(char*,unsigned long);
int fmt_8long(char*,unsigned long);
int fmt_ulong0(char*,unsigned long,unsigned int);
int fmt_plusminus(char*,int );
int fmt_minus(char*,int);
int fmt_str(char*,char*);
int fmt_strn(char*,char*,int);
```

libowfat: scan.a

Übergib NULL und du kriegst die benötigte Länge zurück.

```
int scan_long(char*,signed long*);
int scan_ulong(char*,unsigned long*);
int scan_xlong(char*,unsigned long*);
int scan_8long(char*,unsigned long*);
int scan_plusminus(char*,int*);
int scan_whitenskip(char*,int);
int scan_nonwhitenskip(char*,int);
int scan_charsetnskip(char*,int);
int scan_noncharsetnskip(char*,int);
```

libowfat: open.a

```
int open_read(const char *filename);
int open_excl(const char *filename);
int open_append(const char *filename);
int open_trunc(const char *filename);
int open_write(const char *filename);
```

libowfat: stralloc.a

```
int stralloc_ready(stralloc*,int);
int stralloc_readyplus(stralloc*,int);
int stralloc_copyb(stralloc*,char*,int);
int stralloc_copys(stralloc*,char*);
int stralloc_copy(stralloc*,stralloc*);
int stralloc_catb(stralloc*,char*,int);
int stralloc_cats(stralloc*,char*);
int stralloc_cat(stralloc*,stralloc*);
int stralloc_append(stralloc*,char*);
int stralloc_starts(stralloc*,char*);
int stralloc_catulong0(stralloc*,unsigned,int);
int stralloc_catlong0(stralloc*,signed,int);
void stralloc_free(stralloc* sa);
```

Duh!

Sei dir bewußt, welchen Code der Compiler generiert!

Benutze gprof und objdump (und gcov).