# Effektive C Programmierung

**43rd Law of Computing:**
**Anything that can go wr**
**fortune: Segmentation violation – Core dumped**

**– fortune**

Hagen Paul Pfeifer

8. November 2005

GCC

# Themenübersicht

1. Tools
2. Optimierung
3. ld
4. Performance Analyse Tools

# Basics - Tools

- Statistiktool's:
  - Sektion und Gesamtgröße: `size a.out`
  - Disassemblieren: `objdump -S a.out`
  - ELF Info drucken: `readelf -a a.out`
  - Symbole auflisten: `nm a.out`
- Performancemessprogramme:
  - gprof/gcov
  - oprofile
  - cachegrind
- Codeverwaltung:
  - Versionsverwaltungstools:
    - cvs
    - subversion
    - git

  - Patchmanagment:
    * patch und diff (`-Nuar`)
    * quilt
- `$EDITOR` (vorzugsweise vim)
- make
- autotools
- gdb/ddd
- pod/groff(lesenswert: roff(7))/whatever
- `indent -kr -i8`
- Softwaretests: expect/dejagnu

# GCC Flags - rudimentäre Flags

- `-c`
  - übersetzen, nicht linken
- `-S`
  - Stoppt nach dem Kompilieren, kein Assemblieren
- `-E`
  - Stoppt nach dem Preprozessor
  - Nützlich um `#DEFINE` zu debuggen
- `-march=arch;-mcpu=arch`
- `-O{s,0,1,2,3}`
  - Optimierungsflags
- `-g`
  - Debugginformationen hinzufügen (Symboltabele)
- `-pg`

  - Erweiterung für `gprof` Profiling
- `-pipe`
  - Pipes für den Datenaustausch verwenden (vs. `/tmp/...`)
  - Vorteile bei SMP, HD wird nicht benutzt
- `-I`
  - Pfad für zusätzliche Headerdateien
- `-L`
  - Pfad für Programmbibliotheken
- `-l`
  - zusätzliche Bibliothek „dazubinden“

# GCC Flags - Warnungen

**-Wall** fasst einige notwendigen Warnungen zusammen, sollte immer benutzt werden!

**-W** zusätzliche Warnungen

**-Wshadow** lokale Variable vs. {lokale,global,parameter} Variable

**-Wsign-compare** sollte klar sein

**-Winline** warnt wenn eine inline Funktion nicht geinlined werden kann

**-pedantic** schaltet empfohlene C Standard Meldungen an

**-Wformat-security** warnt vor eventuellen Sicherheitsproblemen bei Format Funktionen (printf(maliciousstring)).

**...** eure Ideen hier! ( `-Wpointer-arith -Wcast-qual -Wcast-align`)

# GCC Erweiterungen

- C++ Kommentare: `//`, ...häßlich!
- Dollar Zeichen in lvalues
- `__alignof__`

```
printf("Align of char: %d\n", __alignof__(char)); /* 1 */
printf("Align of int: %d\n",  __alignof__(int)); /* 4 */
printf("Align of long int: %d\n", __alignof__(long int)); /* 4 */
```

# GCC-Optimierungen - Optimierungs Flags

- `-O{s,0,1,2,3}`
- Default: `-O0`
- `-f` und `-m` für feingranularere Optimierungen
- `-On` schaltet nicht alle `-f` und `-m` Flags an

# GCC-Optimierungen - Optimierungs Flags (-O0)

-O0

- Keine (naja fast) Code Transformationen
- Perfekt für Debugging

# GCC-Optimierungen - Optimierungs Flags (-O1)

-O1

- Wenige Code Transformationen
- Ausführungs Ordnung bleibt erhalten
- Variablen bleiben erhalten
- Kein Funktionsinlining
- Debugginformationen fast perfekt (Zeilennummern eventuell falsch)

# GCC-Optimierungen - Optimierungs Flags (-O2)

-O2

- Aggressivere Code Transformationen
- Debuggbarkeit eventuell in Mitleidenschaft gezogen
  - Variablen sind eventuell wegoptimiert
  - eventuell auch Funktionsblöcke

# GCC-Optimierungen - Optimierungs Flags (-O3)

-O3

- Aggressivere Optimierungen
- Eventuell besser, längere Compilezeit
- Programm Order wird verändert
- Floating point Arithmetik wird verändert
- Debuggbarkeit sehr schlecht möglich

# GCC-Optimierungen - Optimierungs Flags (-Os)

-Os

- Identisch mit `-O2`
- Optimiert auf Codesize
- Aber: eventuell schnellerer Code (weniger Code zu ausfüren)
  - Cacheline
  - PageFaults
  - ...
- Debuggbarkeit eventuell in Mitleidenschaft gezogen

# GCC-Optimierungen - Optimierungs Flags

gcc-3.3.5 (gcc/toplev.c)

```
if (!optimize) {
  flag_merge_constants = 0;
}
if (optimize >= 1) {
  flag_defer_pop = 1;
  flag_thread_jumps = 1;
  #ifdef DELAY_SLOTS
  flag_delayed_branch = 1;
  #endif
  #ifdef CAN_DEBUG_WITHOUT_FP
  flag_omit_frame_pointer = 1;
  #endif
  flag_guess_branch_prob = 1;
```

```
  flag_cprop_registers = 1;
  flag_loop_optimize = 1;
  flag_crossjumping = 1;
  flag_if_conversion = 1;
  flag_if_conversion2 = 1;
  }
if (optimize >= 2) {
  flag_optimize_sibling_calls = 1;
  flag_cse_follow_jumps = 1;
  flag_cse_skip_blocks = 1;
  flag_gcse = 1;
  flag_expensive_optimizations = 1;
  flag_strength_reduce = 1;
  flag_rerun_cse_after_loop = 1;
  flag_rerun_loop_opt = 1;
  flag_caller_saves = 1;
  flag_force_mem = 1;
  flag_peephole2 = 1;
```

```
  #ifdef
  INSN_SCHEDULING
  flag_schedule_insns = 1;
  flag_schedule_insns_after_reload = 1;
  #endif
  flag_regmove = 1;
  flag_strict_aliasing = 1;
  flag_delete_null_pointer_checks = 1;
  flag_reorder_blocks = 1;
  flag_reorder_functions = 1;
}
if (optimize >= 3) {
  flag_inline_functions = 1;
  flag_rename_registers = 1;
}
if (optimize < 2 || optimize_size) {
  align_loops = 1;
  align_jumps = 1;
```

```
  align_labels = 1;
  align_functions = 1;
  flag_reorder_blocks = 0;
}
```

# GCC-Optimierungen - Prozessor Flags

```
-march=pentium;-mcpu=pentium
```

- `-march` impliziert `-mcpu`
- erzeugt Code für spezielle Architektur
- `-msse` erzeugt Code für builtin Funktionen (`gcc/config/i386/i386.c`)
- `cat /proc/cpuinfo`

# GCC-Optimierungen: Funktions Inlining

- Synopsis:

```
inline int
max(int a, int b)
{
  return ((a > b) ? a : b);
}
```

- Benefit: kein Funktionsaufruf Overhead (vgl. Makro (Debugging))
- inlining nur bei Optimierung (`-Ox`) oder bei `__attribute__((always_inline));`
- selbstständiges Inlining: `-finline-functions`
- Grenzen: alloca(3)
- Warnungen bei nichtgenuzten Inlining: `-Winline`
- keine Änderung für Funktionsexport (`static`, `-fkeep-inline-functions`)

# GCC-Optimierungen: builtin_expect

- ```
  #define likely(x)   __builtin_expect(!!(x), 1)
  #define unlikely(x) __builtin_expect(!!(x), 0)
  ```
- gcc/builtins.c
  ```

# GCC-Optimierungen: Aliasing

- Wenn eine Speicherstelle über mehrere Namen angesprochen wird
- Aliasing Analyse schaut nach diesen Stellen und merkt sie sich (kompliziert!)
- Hilft den Compiler `dead code` **nicht** zu entfernen (u.a.)
- Tunningmöglichkeiten:
  - `-fstrict-aliasing`
  - `-fargument-alias`
  - `-fargument-noalias`
  - `-fargument-noalias-global`
- ```
  for(i = 0; i < 99; i++)
      do_it(*arg);

  versus

  int i = *arg;
  ```
  ```

```
for(i = 0; i < 99; i++)
    do_it(i);
```

- C99 Keyword: `restricted` (GCC_MAJOR >= 3)

# GCC-Optimierungen: misc

- Array Indizies (O(1), flexibel denken!)

```
static char *tmp = "SPD";
[..]
char ltr = tmp[indiz];
```

- Integer (unsigned, float, garantierte Breite)
- Loop unrolling (`-funroll-loops`) (`gcc/unroll.c`)

```
for(i = 5; i--; )
    do_it(i);

versus

do_it(4);
do_it(3);
```

`[...]`

- Pass by Reference, for allen Dingen grosse structs! ;-)
- `const` Modifier
- `puts()` gegen `printf()`
- Rekursion überdenken
- Global und Static

# GCC-Optimierungen: alloca(3) vs. Arrays von variabler Länge (ISO C99)

- `alloca(3)`
  - allokiert Speicher im Stack
  - wird beim Verlassen der **Funktion** freigegeben
  - ACHTUNG: `alloca(3)` wird in der Regel `geinlined`: kein Funktionspointer, kein `NULL` return Fehler bei Stacküberlauf
  - im Grunde ist `alloca(3)` ein inkremment auf `%esp`
- Arrays von variabler Länge (`Arrays of variable length`)
  - allokiert Speicher im Stack
  - wird beim verlassen des **Blockes** freigegeben (`brace level`)
  - eleganter als `alloca(3)`

# GCC-Ausblick: Version 4

Profiled Optimization

- Es können detaillierte Messdaten verwendet werden!
- Nachteil: zweimalige Kompilation
- `gcc -fprofile-generate test.c`
- `./a.out`
- `gcc -fprofile-use test.c`

# GLIBC - Glibc Checks

```
*** glibc detected *** nmap: malloc(): memory corruption: 0x08718a5
```

- glibc beendet Programm wenn Fehler in einer Speicherfunktion entdeckt wird
- fast keine False Positivs - also nicht ignorieren
- TIPP: `valgrind --tool=memcheck a.out` findet diese Fehler!

# GLIBC - Umgebungsvariablen

- `LD_BIND_NOW` Programmsymbole beim Start auflösen
- `LD_PRELOAD` Bibliotheken bevorzugen . . .
- `MALLOC_CHECK_`

# GLIBC - DSO's

- just relocatable executables
- Synopsis: `cc -rdynamic -shared -o libtest.so test.o`
- Wenn nicht notwendig dann nicht verwenden! (ganz einfach eigentlich)
  - Höherer Ressourcenverbrauch
  - Mehr Indirektion notwendig
  -
- PIE
  - Position Indepented Executable
    - Ausführbar
    - ladbar als eine `DT_NEEDED` Abhängigkeit
    - oder via dlopen(3)

# LD - Der Link Editor

- Kombiniert Objekt (*.o) und Archiv (*.a) Dateien, Segmente werden angeordnet und Symbole aufgelöst
- Letzter Schritt bei dem Übersetzen (`gcc hello_world.c`)

# LD - Beispiel

- ```
  ld -o test --eh-frame-hdr -m elf_i386 -dynamic-linker
  /lib/ld-linux.so.2 crt1.o crti.o crtbegin.o
  -L/usr/lib/gcc-lib/i68... test.o -lgcc -lgcc_eh
  -lc -lgcc -lgcc_eh crtend.o crtn.o
  ```

- Sieht komplex aus, ist es auch (Pfadangaben wurden zudem gekürzt)
- `-dynamic-linker` kann hier weggelassen werden, Defaultwert ist korrekt (specs File) angeordnet und Symbole aufgelöst
- Linkmap anschaien `gcc -Wl,-M test.c`
  ```

# LD - Nützliche Optionen

- `-soname=libXXX.so.0` setzt `DT_SONAME` mit `libXXX.so.0` (`readelf -d <file>` Dynamic Section)
- `-nostdlib (-nostdinc)`
  - Nicht gegen System Bibliotheken linken (keine Standardsuchpfade)
- `-static`
  - linkt nicht gegen Shared Libraries (An Reihenfolge denken: erst `.so` danach `.a`) und `static ET_EXEC`
- Linux Linkerscript:
  - `arch/i386/kernel/vmlinux.lds`
  - `objdump -d --start-address=0x100000  /usr/src/linux/vmlinux`

# Linuxthreads vs. NPTL

- LinuxThreads
  - gblic > 2
  - beschränkt auf bestimmte Anzahl von Threads/Process
  - Process Managment Thread
  - Probleme: Signal handling, getpid(), core, Portierbarkeit
- NGPT
  - IBM
  - bessere Performance als LinuxThreads
  - nicht mehr weiterentwickelt -> NTPL
- NPTL
  - ab 2.6 (2.5.x) (RedHat 2.4 Backport)
  - größere POSIX kompatibilität
  - skaliert um einiges besser: SMP, Syncronisation
  - Signalhandling

- Was nutzt mein System? `getconf GNU_LIBPTHREAD_VERSION`
- M:1, M:N (NetBSD)

# Autotools

- Verwendung bei Projekten mit verschiedenen Umgebungen (Hardware, Software)
- autoconf
  - configure.in -> configure
- autoscan
  - *.c,h -> configure.in
- autoheader
  - configure.in config.h.in
- configure
  - config.h.in -> config.h und Makefile.in -> Makefile
- Automake
  - Makefile.am -> Makefile
- Beispiel?

# Performance Analyse Tools - Basics

- Wichtig: zu 90% liegt ein algorithmisches Schwachstelle zu Grunde
- Blindlings `-f` Flags ist der Anfang von allem Übel
- Hot Spots finden!
- Randbedingungen vergleichen: Host RAM usage, Host IO, . . .

# Performance Analyse Tools - time

- time(1)
  - `time ./calculate`
    `./calculate  4.01s user 0.00s system 99% cpu 4.051 total`
  - shell builtin (hier zsh)
  - times, getrusage
- `gettimeofday(2)` und Konsorten
  - relativ ungenau
  - Kontextwechsel beachten! (evtl. Mittelwert bilden)
- `rdtsc`
  - ReaDTimeStapCounter
  - sehr genau
  - Auch hier: Kontextwechsel bedenken!

# Performance Analyse Tools - GProf/GCov

- gprof
  - gehört zu jeder Standard Distribution
  - gcc fügt Zeitproben in den generierten Code
  - usage: `gcc -pg; ./a.out; ls gmon.out; gprof a.out`
  - Implementierung: `setitimer(3)` vs. `profil(3)`
- gcov
  - für Abdeckungsanalyse (coverage)
  - zeigt wieviel mal eine Anweisung angesprungen wurde
  - `gcc -fprofile-arcs -ftest-coverage test.c`
  - `gcov test.c` erzeugt `test.c.gcov`

# Performance Analyse Tools - OProfile

- Nutzt Hardware Counter für Profiling Informationen (Fallback!)
- Sammelt Systemweite Zeit Informationen
- Sourcecode muss nicht modifiziert werden
- Moderne CPU habe viele „spannende“ Register ;-)
- geringer Overhead da Hardware
- `oprofile: using timer interrupt.` -> nicht schön ;-(
- Rootrechte benötigt

# Performance Analyse Tools - OProfile II

- ```
  sudo /usr/bin/opcontrol --shutdown 1>/dev/null 2>&1
  sudo /usr/bin/opcontrol --reset
  sudo /usr/bin/opcontrol --setup --vmlinux=/lib/modules/‘uname -r‘/bu
                                  --event=CPU_CLK_UNHALTED:600000:0:1
                                  --separate=library
  sudo /usr/bin/opcontrol --start
  $@
  sudo /usr/bin/opcontrol --shutdown
  ```
- `opwarp ./calculate`
- `opreport -t5 --long-filenames`
- `opreport -l ./calculate`
- Vgl. Intel’s VTune
  ```

# Performance Analyse Tools - Valgrind

- Valgrind emuliert x86
- Core Modul macht die emulation
- weitere Module für Profiling
  - `memcheck` findet Speichermanagmentprobleme
  - `addrcheck` identisch bis auf überprüfund auf unititialisierten Speicher
  - `Cachegrind` Testen von L1/L2/D2 Cachemisses (cpuid)
  - `Callgrind`
  - `Coregrind` rudimentäre Fehlerprüfung
  - `Massif` heap profiler (wieviel) (`--alloc-fn=xmalloc`)
- verlangsamt extrem (Faktor 5-100, je Modul)
- nicht nur für Performance: z.B. memcheck (Empfehlung!)
- `valgrind --tool=cachegrind ./calculate`
- Gui: KCachegrind

# Performance Analyse Tools - Auswege

- Keine generelle Antwort möglich! (Mist ;-)
- passen die Algorithmen - ist qsort wirklich optimal?
- `-march=ARCH`?
- inline Assembler?
- ...

# FIN


1697 - BEI EINEM TELEFONAT MIT SEINER MUTTER...
JA, JA, NEIN
JA, NEIN, JA
NEIN, NEIN,
JA, JA, JA,
NEIN, JA...
(c) fussel 1999
... ENTDECKT LEIBNIZ DEN BINÄRCODE!

# Literatur, Links, . . .

- Oprofile: `http://people.redhat.com/wcohen/Oprofile.pdf`
- `(info)|(man) gcc`
- diff info pages
- NPTL http://people.redhat.com/drepper/nptl-design.pdf