# Compiler-Optimierungen

| | |
|---|---|
| `-O0` | Keine Optimierungen (Default) |
| `-O1` | Optimierungen mit wenig Compile-Zeit |
| `-Og` | `-O1` mit Fokus auf debugbaren Code |
| `-O2` | Alle Optimierungen ohne Space–Speed Trade-off |
| `-Os` | `-O2` mit Fokus auf minimaler Code-Größe |
| `-O3` | Alle Optimierungen |
| `-Ofast` | `-O3` mit Floating Point Optimierungen ("Disregard strict standards compliance") |

- Compiler-Optimierungen nur valide, wenn Verhalten unverändert
  - Viele Optimierungen nicht vollautomatisch durchführbar!

# Optimierung von Berechnungen

- Constant Folding
  - Berechnung von Konstanten zur Compilezeit
- Constant Propagation
  - Variablen werden mit ihren Werten ersetzt
  - Bei Funktionen: Ergebnis des Funktionsaufrufs wird bereits berechnet
- **C**ommon **S**ubexpression **E**limination

```
1 int x = a * b * 24;
2 int y = a * b * c;
```

$\rightarrow$

```
1 int tmp = a * b;
2 int x = tmp * 24;
3 int y = tmp * c;
```

# Optimierung von Schleifen: Loop Unrolling

## Unoptimiert

```
1 for(int i = 0; i < 6; i++) {
2     arr[i] = 2*arr[i];
3 }
```

## Optimiert[1]

```
1 for(int i = 0; i < 6; i += 2) {
2     arr[i] = 2*arr[i];
3     arr[i+1] = 2*arr[i+1];
4 }
```

+ Erhöhte Geschwindigkeit
  - Loop Conditions werden weniger/gar nicht getestet
- Executable Größe wächst
  - Mehr Instruktionen nötig

[1] `-floop-unroll-and-jam`, ab `-O3`

# Optimierung von Schleifen: Jamming/Loop Fusion

## Unoptimiert

```
1 for(int i = 0; i < 6; i++) {
2     arr1[i] = 2*arr1[i];
3 }
4 for(int i = 0; i < 6; i++) {
5     arr2[i] = arr2[i] + 24;
6 }
```

## Optimiert[1]

```
1 for(int i = 0; i < 6; i++) {
2     arr1[i] = 2*arr1[i];
3     arr2[i] = arr2[i] + 24;
4 }
```

+ Vermeidung von doppeltem Schleifen-Overhead
+ Evtl. mehr Optimierungen in Schleifenkörper möglich

[1] `-floop-unroll-and-jam`, ab `-O3`

# Quiz: Loop Unrolling

Kann Schleife (1) problemlos zu Schleife (2) optimiert werden?

(1)

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

(2)

```
1 for(int i = 0; i < n; i += 2) {
2     arr[i] = 2*arr[i];
3     arr[i+1] = 2*arr[i+1];
4 }
```

☐ Ja, immer

☐ Nein, im Allgemeinen nicht

☐ Nein, nie

# Optimierung von Schleifen: Loop-Invariant Code Motion

Unoptimiert

```
1 int n;
2 for(int i = 0; i < x; i++) {
3     n = sizeof(arr)/
4         sizeof(int);
5     arr[i] = 2*arr[i];
6 }
```

Optimiert[1]

```
1 int n = sizeof(arr)/
2         sizeof(int);
3
4 for(int i = 0; i < x; i++) {
5     arr[i] = 2*arr[i];
6 }
```

- Im Beispiel: falls `x==0`: `n` kann jeden Wert haben, daher Opt. korrekt
  - Beispiel für Ausnutzung von *Undefined Behavior* für Optimierungen!

\+ Vermeidung redundanter Berechnungen

[1] `-fmove-loop-invariants`, ab `-O1`

# Optimierung von Schleifen: Vertauschung

Unoptimiert

```
1 int sum = 0;
2 for(int i = 0; i < 6; i++) {
3   for(int j = 0; j < 9; j++) {
4     sum += arr[j][i];
5   }
6 }
```

Optimiert[1]

```
1 int sum = 0;
2 for(int j = 0; j < 9; j++) {
3   for(int i = 0; i < 6; i++) {
4     sum += arr[j][i];
5   }
6 }
```

+ Verbessert das Cacheverhalten
  - Weniger Cache-Misses aufgrund der Vertauschung
+ Ermöglicht evtl. Vektorisierung

[1] `-floop-interchange`, ab `-O3`

# Optimierung von Funktionsaufrufen: Inlining

Unoptimiert

```
1 static int square(int x) {
2     return x*x; }
3 ...
4 for (int i = 0; i < n; i++)
5     arr[i] = square(i);
```

Optimiert[1]

```
1 for (int i = 0; i < n; i++)
2     arr[i] = i*i;
```

+ Kein Overhead durch Funktionsaufruf
– Kann Code massiv vergrößern und damit verlangsamen

- *Achtung:* Der Specifier `inline` hat hiermit nur bedingt etwas zu tun

[1] `-finline-functions-called-once`, ab `-O1` bzw. `-finline-functions`, ab `-O2`

# Optimierung von Funktionsaufrufen: Tail Call Optimization

Unoptimiert

```
1 int fac(int k, unsigned n) {
2   if (n <= 0) return k;
3   return fac(k * n, n - 1);
4 }
```

Optimiert[1]

```
1 int fac(int k, unsigned n) {
2 fac:
3   if(n <= 0) return k;
4   k *= n;
5   n--;
6   goto fac;
7 }
```

- Bedingung: Funktionsaufruf ist *letzte* Operation vor `return`
- Ersetzte Funktionsaufruf (`call+ret`) durch `jmp` zur Funktion

[1] `-foptimize-sibling-calls`, ab `-O2`

# Quiz: Tail-Call Optimization (1)

Kann folgende Funktion in dieser
Form direkt tail-call optimiert werden?

```
1 unsigned long pow2n(unsigned long n) {
2     if (n == 0)
3         return 1;
4     return 2 * pow2n(n-1);
5 }
```

☐ Ja

☐ Nein

# Quiz: Tail-Call Optimization (2)

Kann folgende Funktion in dieser
Form direkt tail-call optimiert werden?

```
1 unsigned long facq(unsigned long n, unsigned long q) {
2     if (n <= 1)
3         return q;
4     return facq(n - 1, q*n);
5 }
```

☐ Ja

☐ Nein

# Interprozedurale Optimierungen

- Entfernen unnötiger Funktionsparameter
  - Nur bei `static` Funktionen möglich
- Funktions-spezifische Calling Convention
  - Z.B. mehr callee-saved Register, andere Argumentregister
  - Nur bei `static` Funktionen ohne externe Nutzung möglich
- Spezialisierung von Funktionen bei mehreren verschiedenen Aufrufen
  - Duplikation und Optimierung für verschiedene Parameterwerte

# Low-Level Optimierungen

- Instruction Selection: Statement $\rightarrow$ Instruktionen
  - Z.B. Ersetzen von Multiplikation mit `lea`
- Instruction Scheduling: Reihenfolge der Instruktionen
  - Verringern von Abhängigkeiten zwischen Instruktionen
  - Bessere Ausnutzung von Instruction-Level Parallelism im Prozessor
- Register Allocation: Variablen $\rightarrow$ Register/Stack
  - Verringern/vermeiden von Stack-Zugriffen

# Optimierte Funktionen

- `libc` stellt häufig benutzte Funktionen hochoptimiert bereit
- Beste Funktion wird zur Laufzeit ausgewählt → `libc` vor eigener Implementierung bevorzugen!

```
1  static inline void* IFUNC_SELECTOR (void) {
2    const struct cpu_features* cpu_features =
3      __get_cpu_features ();
4    //...
5    if (CPU_FEATURES_ARCH_P (cpu_features, Fast_Unaligned_Load))
6      return OPTIMIZE (sse2_unaligned);
7
8    if (CPU_FEATURE_USABLE_P (cpu_features, SSSE3))
9      return OPTIMIZE (ssse3);
10
11   return OPTIMIZE (sse2);
12 }
```

# Builtins

- Funktionen für bestimmte Anwendungen
  - Von GCC bereitgestellt (nicht Teil der Standardbibliothek!)
  - Häufig direkt mithilfe hardwareabhängiger Instruktionen implementiert

- `__builtin_clz(unsigned int x)`
  - **C**ount **L**eading **Z**eros
  - Auf `x86-64` z.B. mithilfe `bsr` implementierbar

- `__builtin_expect(long exp, long c)`
  - Hinweis, dass vermutlich `exp == c` gilt
  - Generiere für diesen Fall optimierten Code
    - Branch Prediction
  - Z.B. `if (__builtin_expect(ptr != NULL, 1)) { ... }`

# Funktionsattribute

- Komplette Analyse des Programms für Compiler teils nicht möglich
  - Definitionen nicht sichtbar
  - Häufig auftretende Eingabewerte/Muster in den Eingabewerten unbekannt
  - etc.
- Programmierer weiß hierüber evtl. mehr
- Funktionsattribute: Hinweise für den Compiler

# Funktionsattribute - Inlining

- `always_inline`: Inlining wird erzwungen

```
__attribute__((always_inline))
void addTwo(uint8_t* element) {
    *element += 2;
}
```

- `noinline`: Inlining wird verhindert

```
__attribute__((noinline))
void addTwo(uint8_t* element) {
    *element += 2;
}
```

# Funktionsattribute - `const`

- Ausgabe *nur* durch Eingabe bestimmt
  - Ergebnis ist unabhängig vom Zustand des Programms
    - Nicht-read-only Speicher darf den Rückgabewert nicht beeinflussen
  - `const`-Funktion darf nur andere `const`-Funktionen aufrufen
  - Funktion verändert Programmzustand nicht
    - `void`-Rückgabewert sinnlos
- Nur nötig bei Funktionen, deren Definition nicht verfügbar ist
- Zweck: Compiler kann Ergebnisse ggf. einfach wiederverwenden

```
1 __attribute__((const))
2 extern uint32_t mulPi(uint32_t n); // n * pi
```

# Funktionsattribute - pure

- Ähnlich zu, aber weniger restriktiv als `const`
  - Rückgabewert darf von Dereferenzierung übergebener Pointer abhängen
  - `pure`-Fkt. dürfen `pure`-Fkt. und `const`-Fkt. aufrufen

```
1 __attribute__((pure))
2 int my_memcmp(const void *ptr1, const void *ptr2, size_t n) {
3     while (!n--)
4         if (*ptr1++ != *ptr2++)
5             return *ptr2 - *ptr1;
6     return 0;
7 }
```

# Funktionsattribute - `hot/cold`

- `hot` für besonders oft aufgerufene Funktionen
  - höhere Optimierung auf Geschwindigkeit
  - größerer Code
  - eigener Speicherbereich für bessere Cachelokalität
- `cold` für besonders selten aufgerufene Funktionen
  - kleinerer Code
  - langsamer
  - eigener Speicherbereich $\rightarrow$ besseres Cacheverhalten des restlichen Programms

# Quiz: Funktionsattribute

Welche Attribute/welche Änderungen sind für folgende Funktion sinnvoll?

```
1 int contains(char* str, char c) {
2     while (*str)
3         if (*str++ == c)
4             return 1;
5     return 0;
6 }
```

- ☐ `__attribute__((const))`
- ☐ `__attribute__((pure))`
- ☐ `__attribute__((noinline))`
- ☐ `str` könnte `const char*` sein
- ☐ `str` könnte `char* const` sein
- ☐ Keine der Antworten ist sinnvoll

# Layout von Datenstrukturen

- Größe der verwendeten Datentpyen
  - So groß wie nötig
  - So klein wie möglich
- Beispielsweise für Zahlen in
  - $\{0, 1, \ldots, 12800\}$ `unsigned short` besser als `unsigned int`
  - $\{0.00, 0.25, 0.50, \ldots, 100.00\}$ `float` besser als `double`
- Genaue größe von `int`, `short`, etc. implementation defined $\rightarrow$ Verwendung von fixed-width Integern sinnvoll
  - Definiert in `stdint.h`
  - Z.B. `uint32_t` statt `unsigned int`, `int16_t` statt `short`, etc.

# Layout von Datenstrukturen: Structs

```
1  struct PenguinBad {     // Alignment: 8 (char*)
2      char type;          // Offset: 0
3      char* name;         // Offset: 8
4      uint8_t age;        // Offset: 16
5  };                      // Size (mult. of alignment): 24
6
7  struct PenguinGood {    // Alignment: 8 (char*)
8      char type;          // Offset: 0
9      uint8_t age;        // Offset: 1
10     char* name;         // Offset: 8
11 };                      // Size (mult. of alignment): 16
```

- Manuelles umordnen der Member ggf. sinnvoll
  - Kann *nicht* automatisch vom Compiler gemacht werden!
- Häufiges kopieren/umwandeln von Daten möglichst vermeiden

# Quiz: Layout von Datenstrukturen (1)

Welche der folgenden Repräsentationsmöglichkeiten ist geeignet und nutzt den Speicherplatz optimal, um Euro-Geldbeträge im Bereich $[0; 100]$ mit einer Genauigkeit von einem Cent darzustellen?

- ☐ `unsigned char money`
- ☐ `unsigned short money`
- ☐ `struct money { uint8_t euro; uint8_t cent; }`
- ☐ `uint16_t money`
- ☐ `struct money { int8_t euro; int8_t cent; }`
- ☐ `int16_t money`

# Quiz: Layout von Datenstrukturen (2)

Welche der folgenden Datentypen ist am besten für die Speicherung von Geldbeträgen mit Cent-Genauigkeit geeignet?

- ☐ `float` (Euro)
- ☐ `double` (Euro)
- ☐ `long` (Cent)

# Quiz: Layout von Datenstrukturen (3)

Was ist der Unterschied zwischen den folgenden beiden Datenstrukturen?

```
1  struct Penguin1 {
2      uint8_t age;
3      struct {
4          uint8_t id;
5          char *name;
6      };
7  };
8  struct Penguin2 {
9      uint8_t age;
10     uint8_t id;
11     char *name;
12 };
```

- ☐ Unterschiedliche Initialisierung möglich.
- ☐ Eine der beiden führt zu einem Compilerfehler.
- ☐ `struct Penguin1` nimmt mehr Speicherplatz ein.
- ☐ `struct Penguin2` nimmt mehr Speicherplatz ein.

# Pointer Aliasing

- Pointer zeigen auf Speicherobjekte nur *eines* bestimmten Typs
  - Pointer-Casts zwar möglich
  - Aber: Dereferenzierung allgemein *undefined behavior*
- `U* ptr2` zeigt auf gleichen Speicherbereich wie `T* ptr1` $\rightarrow$ `ptr2` ist Alias von `ptr1`
  - Nicht jeder Pointer kann Alias für jeden anderen sein
  - Nur gültige Aliase sollten auf gleichen Speichebereich zeigen

# Pointer Aliasing: `restrict`

```
1 void foo(unsigned* ptr_a, int* ptr_b) {
2     // ... do something with the pointers ...
3 }
4 void foo2(unsigned* restrict ptr_a, int* restrict ptr_b) {
5     // ... do something with the pointers ...
6 }
```

# restrict: Beispiel (1)

```
1 void count_a(const char *arr, int* sum) {
2     while (*arr) {
3         *sum += *arr++ == 'a';
4     }
5 }
```

- arr und sum zeigen nicht auf gleichen Speicher
  - Aber: Compiler kann das nicht wissen (char* kann alles aliasen)

```
1 void count_a(const char* restrict arr, int* sum) {
2     while (*arr) {
3         *sum += *arr++ == 'a';
4     }
5 }
```

# restrict: Beispiel (2)

```
1 void count_a_short(const short arr[4], int* sum) {
2     for (size_t i = 0; i < 4; i++) {
3         *sum += arr[i] == 'a';
4     }
5 }
```

- `arr` und `sum` können keine (gültigen) Aliase sein → Zeigen nicht auf gleiche Speicherbereiche
  - Aus historischen Gründen: GCC optimiert per default nicht basierend darauf
  - Optimierungen erst ab `-O2` oder mit Flag `-fstrict-aliasing`
- Optimierung: `sum` muss nicht bei jeder Iteration in den Speicher geschrieben werden