Effiziente Programme

Gruppe 107

Programm: Forth-Dictionary

Es gibt 256 Wörterlisten

Algorithmus - Datei iterieren:

- ‘\n’ ➔
  - Wort mit serialNumber++ bis <= ‘ ’ in Wörterliste[erstes Zeichen] eintragen
  - Iterator: um Wort erhöhen (=weiter nach dem Abstand)
- ‘\t’ ➔
  - Es kommt bis <= ‘ ’ eine Zeichenfolge mit WörterlisteIDs (pro Zeichen) ➔ Order setzen
  - Iterator: um Zeichenfolge erhöhen
- ‘ ‘ ➔
  - Das Wort in bis <= ‘ ’ in den Wörterlisten der Order nach suchen
  - Daraus einen Hash-Wert updaten
  - Iterator darum erhöhen
- ‘\0’ ➔ Ende

Perf

Name	Cycles	instructions	branch-misses	time elapsed (s)
Original	164803699	80381725	520701	0.056704570

1) Wörterlisten optimieren: Hash table

- Implementierung: Linked List[256]
- Änderung: Linked List ➔ Hash table (Size=65536) pro Wörter-Tabelle[256]
- VT: Iteration der ganzen Liste nicht mehr nötig ➔ Geschwindigkeit
- NT: Speichergröße: 65536 Einträge pro Wörterbuch
- NT: Speicher für die Hash tables zu reservieren benötigt Zeit

1) Kollisionen?

2 Strategien

- Nicht kollisionssicher
  - Hash zeigt auf Speicherbereich in dem der Eintrag steckt
  - Schnell
  - Selber Hash für zwei Keys ➔ Pech
  - Wenn wir die hash size groß genug machen, wäre es für unseren Input kein Problem
    - Aber nicht mehr sicher ➔ wir verzichten darauf
- Kollisionssicher
  - Hash zeigt auf Liste in dem der Eintrag mit Key steckt
  - Eintrag muss auch noch mit Key verglichen werden ➔ langsam

1) Hash-Funktion?

Name	Lowercase(ns)/Kollisionen	Random nr(ns)/Kollisionen	Zahlen(ns)/Kollisionen
Murmur	**145 / 6**	259 / 5	92 / 0
FNV-1a	152 / 4	504 / 4	86 / 0
FNV-1	184 / 1	730 / 5	92 / 0
DBJ2a	158 / 5	443 / 6	91 / 0
DJB2	156 / 7	437 / 6	93 / 0
SDBM	148 / 4	484 / 6	90 / 0
CRC32	250 / 2	946 / 0	130 / 0

http://programmers.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed

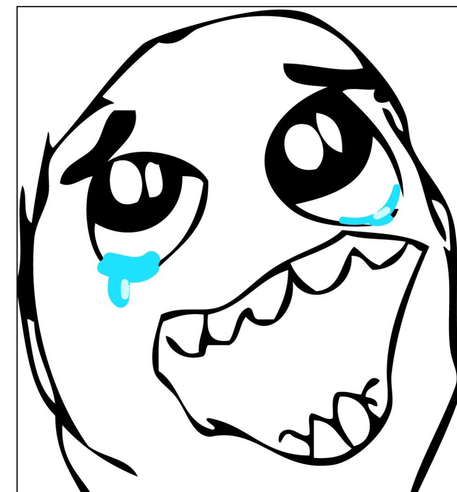
2) Wie Tabellen nur anlegen, wenn benötigt?

Table mit 0 initialisieren und:

```
currentTable = createTable(index):
tables[index] != NULL -> return tables[index]
tables[index] == NULL ->
        tables[index] = malloc(hashSize * sizeof(entry_ht*));
        memset(tables[index], 0, hashSize * sizeof(entry_ht*));
        return tables[index];
```

Perf

Name	Cycles	instructions	branch-misses	time elapsed (s)
Original	164803699	80381725	520701	0.056704570
Hash table (Size = 65536)	6808659	10259268	62915	0.007099838



3) Tabelle optimieren: Hash table size

- Mehr Einträge (65536) pro HT ➔
  - malloc teurer
  - mehr Speicher
  - weniger Kollisionen (0 bei dieser Größe)
- Weniger Einträge (8192) pro HT ➔
  - **malloc billiger**
  - weniger Speicher
  - mehr Kollisionen(20!)
    - Glücklicherweise sind wir geschützt
- Wichtig: wir haben hier nur für den vorgegebenen Input optimiert

Perf

Name	Cycles	instructions	branch-misses	time elapsed (s)
Original	164803699	80381725	520701	0.056704570
Hash table (Size=65536)	6808659	10259268	62915	0.007099838
Hash table (Size=8192)	6083808	10036241	69914	0.004777776

4) Inline-Keyword vor Funktionen

- `inline`-Keyword kopiert den Code
  - die Funktionen müssen nicht mehr aufgerufen werden
- VT: Schneller
- NT: Programmgröße

➔ `inline`

- getHashed()
- setHashed()
- createTable()
- hash()
- find()
- murmur()
- set_order()
- create()

Perf

Name	Cycles	instructions	branch-misses	time elapsed (s)
Original	164803699	80381725	520701	0.056704570
Hash table (Size=65536)	6808659	10259268	62915	0.007099838
Hash table (Size=8192)	6083808	10036241	69914	0.004777776
inline	5837387	9907937	69101	0.004658551

5) Kollision vergleichen

Aktuell: `if(strcmp(newKey, next->key) == 0)`

Verbesserung: `if(len==next->len && strcmp(newKey, next->key)==0)`

- VT: Geschwindigkeit
- NT: Einträge haben ein zusätzliches Feld ➔ Speicherplatz
- Ergebnis: Geschwindigkeitsvorteil wird wohl durch mehr malloc wieder ausgeglichen ➔ bleibt gleich ➔ wir wenden es (vorerst) nicht an

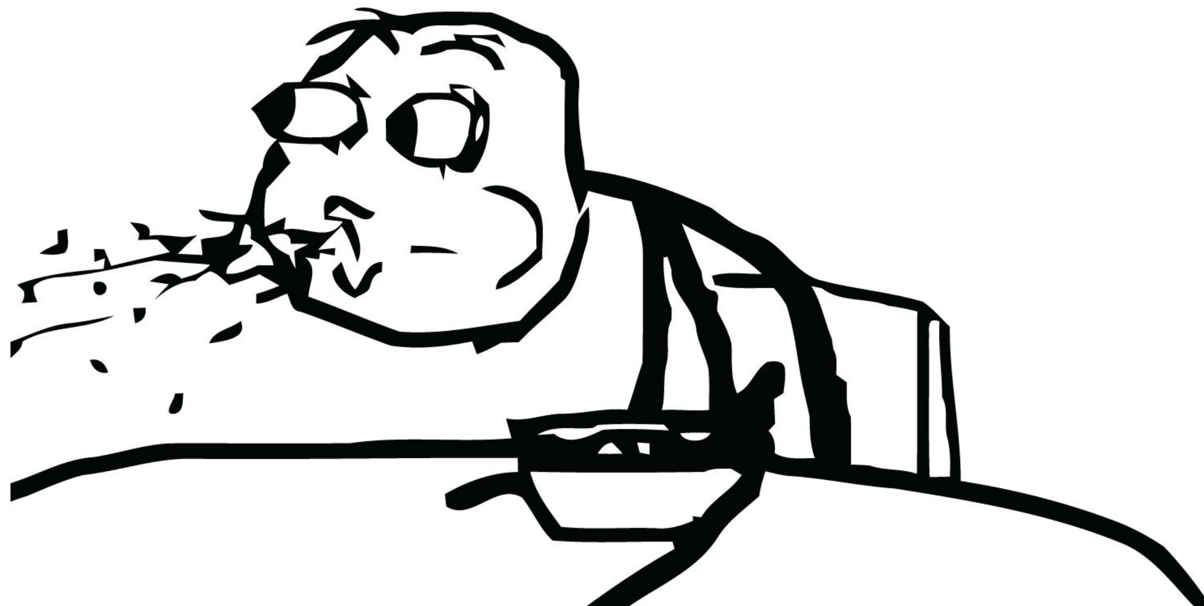
Perf

Name	Cycles	instructions	branch-misses	time elapsed (s)
Original	164803699	80381725	520701	0.056704570
Hash table (Size=65536)	6808659	10259268	62915	0.007099838
Hash table (Size=8192)	6083808	10036241	69914	0.004777776
inline	5837387	9907937	69101	0.004658551
Kollisionen vergleichen - FAILED	5857387	9927517	69328	0.004662549

6) Order optimieren

- find() durchsucht den char* order Zeichen für Zeichen
  - sucht in wordlist[order[j]] bis sie das Suchwort findet
- Wie schaut eine order aus?
  - printf("%s; ", strndup(order, order_len));

44; ; 4; **44**; 47; 4**77**; 4**777;** 4**77**4; 4746; 474**66**; 4746:; 4746; 474**66**; 47464; 4746; 4787; 4746; 4787; 4746; 4787; 4746; 4746;; 4746; 4787; 4746; 4787; 4746; 4787; 4746; 4787; 4746; 4787; 4746; [...]



6) Order optimieren

- find() durchsucht den String “order” Zeichen für Zeichen
  - sucht in wordlist[order[j]] bis sie das Suchwort findet
- Wie schaut so eine order aus?
  - printf("%s; ", strndup(order, order_len));

➔ Lauter doppelte Zugriffe auf Tabelle

```
for (j=order_len-1; j>=0; j--) {
        if(j != (order_len -1) && order[j] == order[j+1])
                continue;
```

➔ Werte etwas besser

Perf

Name	Cycles	instructions	branch-misses	time elapsed (s)
Original	164803699	80381725	520701	0.056704570
Hash table (Size=65536)	6808659	10259268	62915	0.007099838
Hash table (Size=8192)	6083808	10036241	69914	0.004777776
inline	5837387	9907937	69101	0.004658551
Kollisionen vergleichen - FAILED	5857387	9927517	69328	0.004662549
Order optimiert	5852603	9988506	69675	0.004689033

5) Kollision vergleichen

Aktuell: `if(strcmp(newKey, next->key) == 0)`

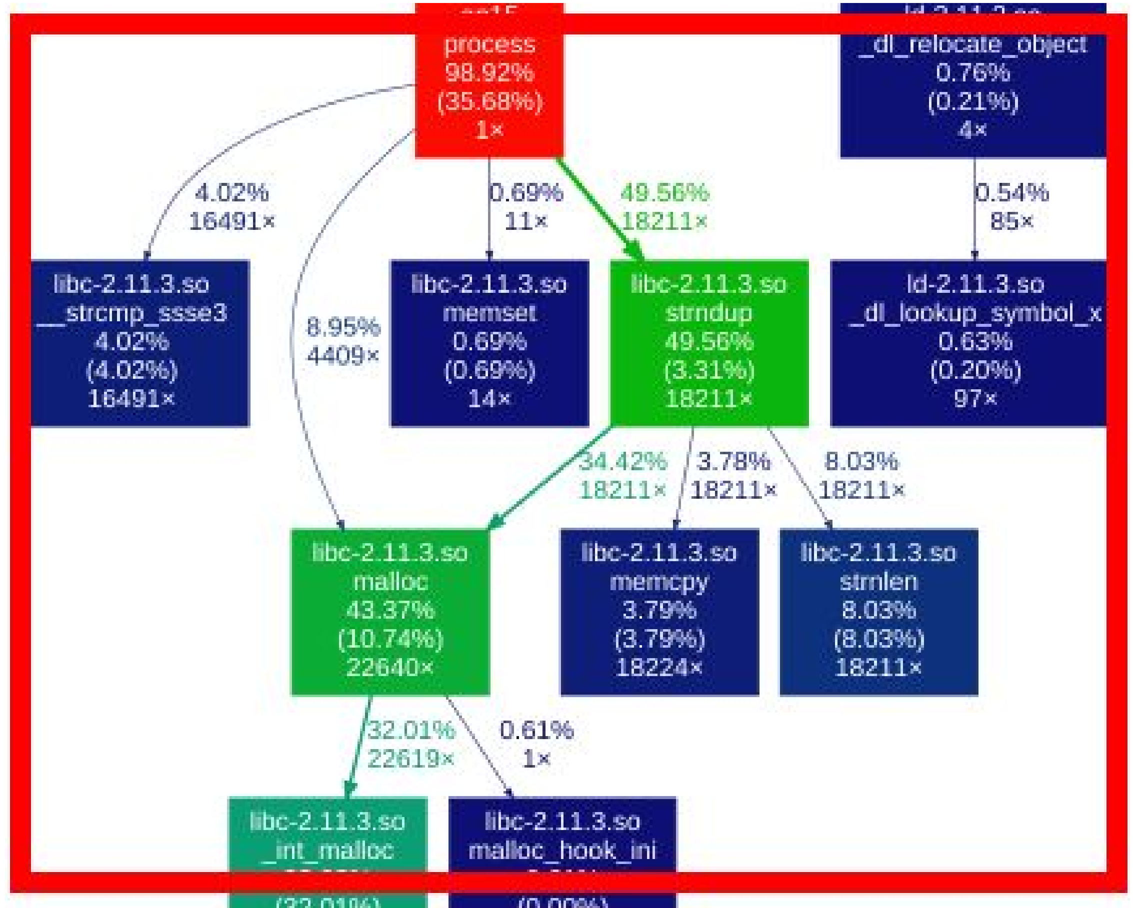
Verbesserung: `if(len==next->len && strcmp(newKey, next->key)==0)`

- VT: Geschwindigkeit
- NT: Einträge haben ein zusätzliches Feld ➔ Speicherplatz
- Ergebnis: Geschwindigkeitsvorteil wird wohl durch mehr malloc wieder ausgeglichen ➔ bleibt gleich ➔ wir wenden es nicht an

Zeit zu profilen

valgrind --tool=callgrind ep15 cross.input

gprof2dot -f callgrind callgrind.out.6507 | dot -Tsvg -o output.svg



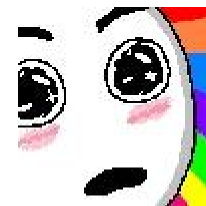
7) Zurück zum Kollisionen-Vergleichen

- Beim Hash-Vergleich haben wir den Key immer dupliziert um ihn zu prüfen
- Weil: Hash Key in Tabelle könnte ja länger sein

➔ schlechte Idee
➔ Key-Length wieder anlegen und vergleichen (Schritt 5)

```
if(len==next->len && strncmp(key, next->key, len)==0)
```

Perf



Name	Cycles	instructions	branch-misses	time elapsed (s)
Original	164803699	80381725	520701	0.056704570
Hash table (Size=65536)	6808659	10259268	62915	0.007099838
Hash table (Size=8192)	6083808	10036241	69914	0.004777776
inline	5837387	9907937	69101	0.004658551
Kollisionen vergleichen - FAILED	5857387	9927517	69328	0.004662549
Order optimiert	5852603	9988506	69675	0.004689033
KeyLength mit strndup()-Elimination	3475239	5049100	45973	0.002896824

8) strncmp ersetzen

- strncmp: Einfache Schleife
- http://mgronhol.github.io/fast-strcmp/ block(size_t) XOR(same XOR same = 0):

```
while( current_block < fast ){ +inline
    if( (lptr0[current_block] ^ lptr1[current_block])){
        int pos;
        for(pos = current_block*sizeof(size_t); pos < len ; ++pos ){
            if( (ptr0[pos] ^ ptr1[pos]) || (ptr0[pos] == 0) || (ptr1[pos] == 0) ){
                return 1 // Erweiterung von uns: nur 0(match) oder 1(none match)
...
```

Perf

Name	Cycles	instructions	branch-misses	time elapsed (s)
Original	164803699	80381725	520701	0.056704570
Hash table (Size=65536)	6808659	10259268	62915	0.007099838
Hash table (Size=8192)	6083808	10036241	69914	0.004777776
inline	5837387	9907937	69101	0.004658551
Kollisionen vergleichen - FAILED	5857387	9927517	69328	0.004662549
Order optimiert	5852603	9988506	69675	0.004689033
KeyLength mit strndup()-Elimination	3475239	5049100	45973	0.002896824
strncmp() ersetzen	3120222	5310864	34906	**0.002749979**

Ende