185.190
Effiziente Programme

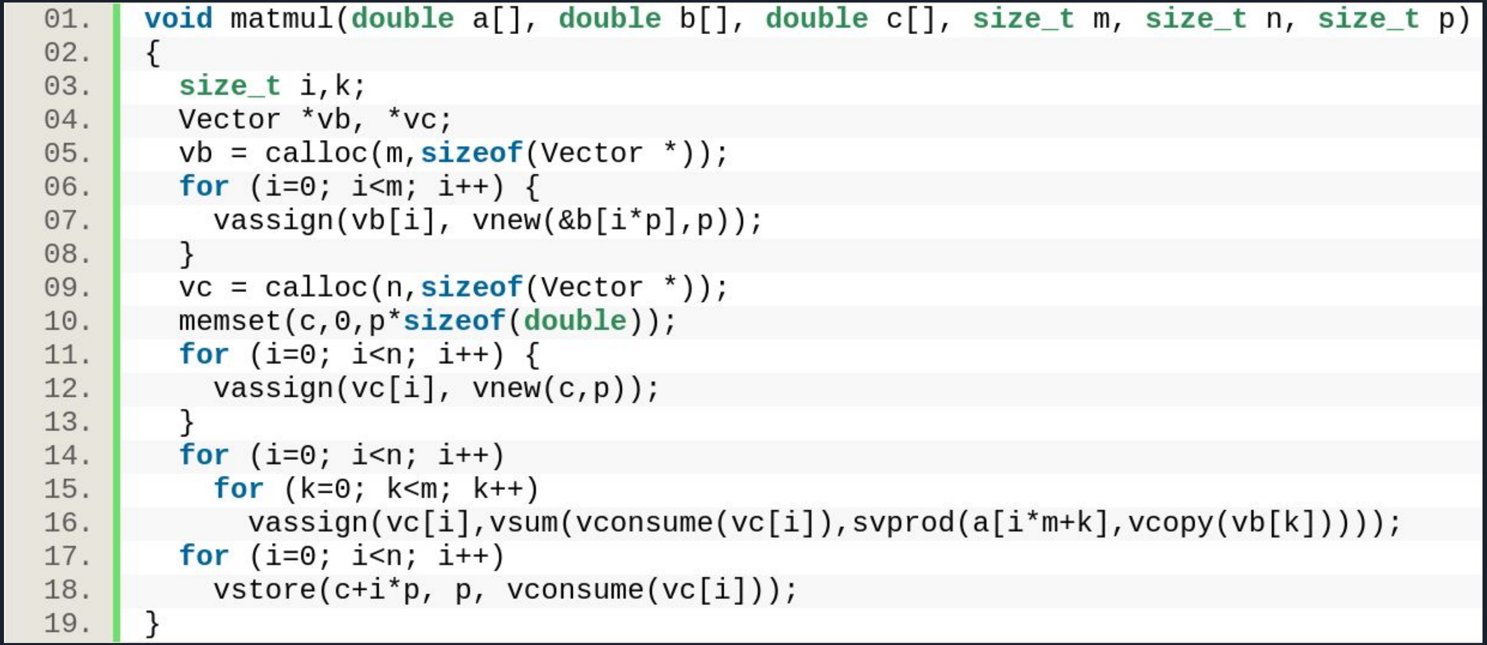
Theodor Mittermair, 1426389
Stephan Felber, 1426418
Torsten Fux, 1425720

Aufgabenstellung

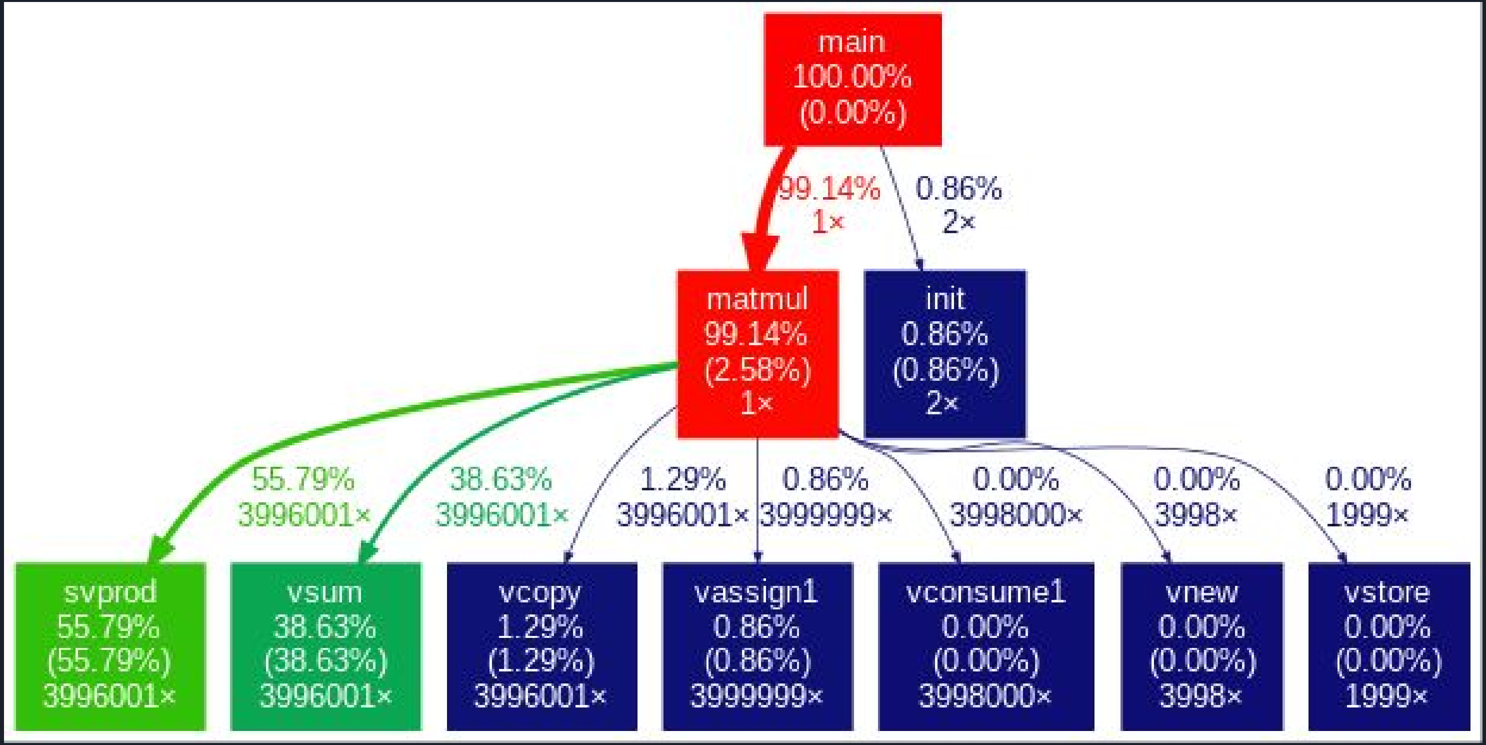
- Vektor Library
  - Vektor-(Punkt-)Summe   U .+ V
  - Vektor-(Punkt-)Produkt   U .* V
  - Vektor-Skalar-Produk   V * s
- Werte Semantik
- Interface und Verhalten vorgegeben
- Optimierungen nur innerhalb der Library

Baseline

Metrik	Absolut	Relative
Cycles	1.790.767.358	100%
Instructions	4.571.583.802	100%



```
void matmul(double a[], double b[], double c[], size_t m, size_t n, size_t p)
{
  size_t i,k;
  Vector *vb, *vc;
  vb = calloc(m,sizeof(Vector *));
  for (i=0; i<m; i++) {
    vassign(vb[i], vnew(&b[i*p],p));
  }
  vc = calloc(n,sizeof(Vector *));
  memset(c,0,p*sizeof(double));
  for (i=0; i<n; i++) {
    vassign(vc[i], vnew(c,p));
  }
  for (i=0; i<n; i++)
    for (k=0; k<m; k++)
      vassign(vc[i],vsum(vconsume(vc[i]),svprod(a[i*m+k],vcopy(vb[k]))));
  for (i=0; i<n; i++)
    vstore(c+i*p, p, vconsume(vc[i]));
}
```


main
100.00%
(0.00%)
99.14%
1×
0.86%
2×
matmul
99.14%
(2.58%)
1×
init
0.86%
(0.86%)
2×
55.79%
3996001×
38.63%
3996001×
1.29%
3996001×
0.86%
3999999×
0.00%
3998000×
0.00%
3998×
0.00%
1999×
svprod
55.79%
(55.79%)
3996001×
vsum
38.63%
(38.63%)
3996001×
vcopy
1.29%
(1.29%)
3996001×
vassign1
0.86%
(0.86%)
3999999×
vconsume1
0.00%
(0.00%)
3998000×
vnew
0.00%
(0.00%)
3998×
vstore
0.00%
(0.00%)
1999×

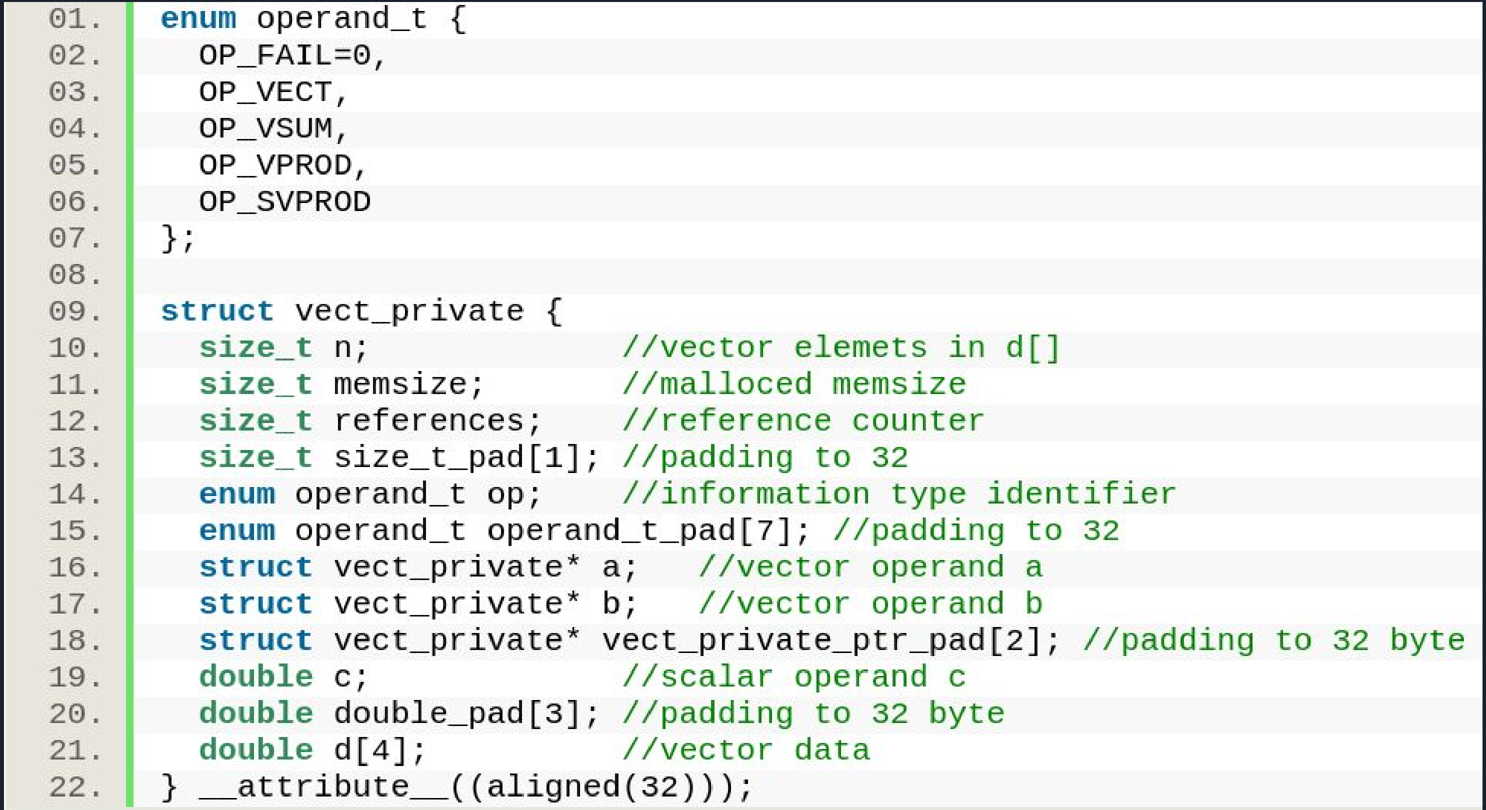
Optimization?

- Nur Library modifizieren erlaubt
    - -> keine Datenumordnung
- Kompiliert bereits mit -03 -mavx
    - -> kaum Assembly Optimierungen möglich
    - -> Gewinn “minimal” (Compiler 1 : Mensch 0)

Optimization ...?

- Veränderung des Programmablaufs
    - Lazy Evaluation
        - Ermöglicht Datenumordnung
        - Ermöglicht Wiederverwendung
        - Ermöglicht Situationsspezifische Optimierung
- Optimierungen des Speichermanagements

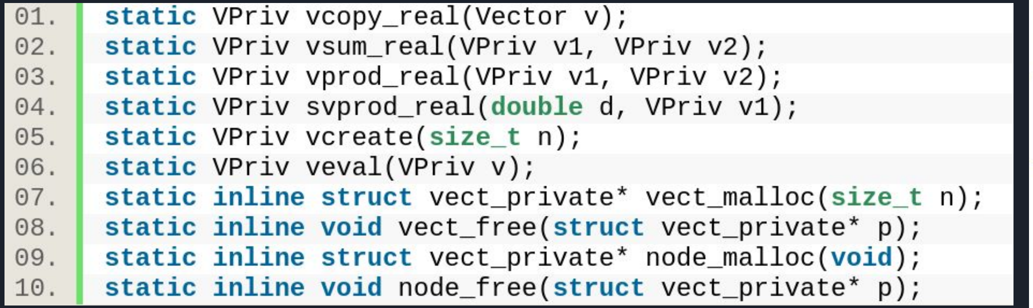
Lazy Evaluation



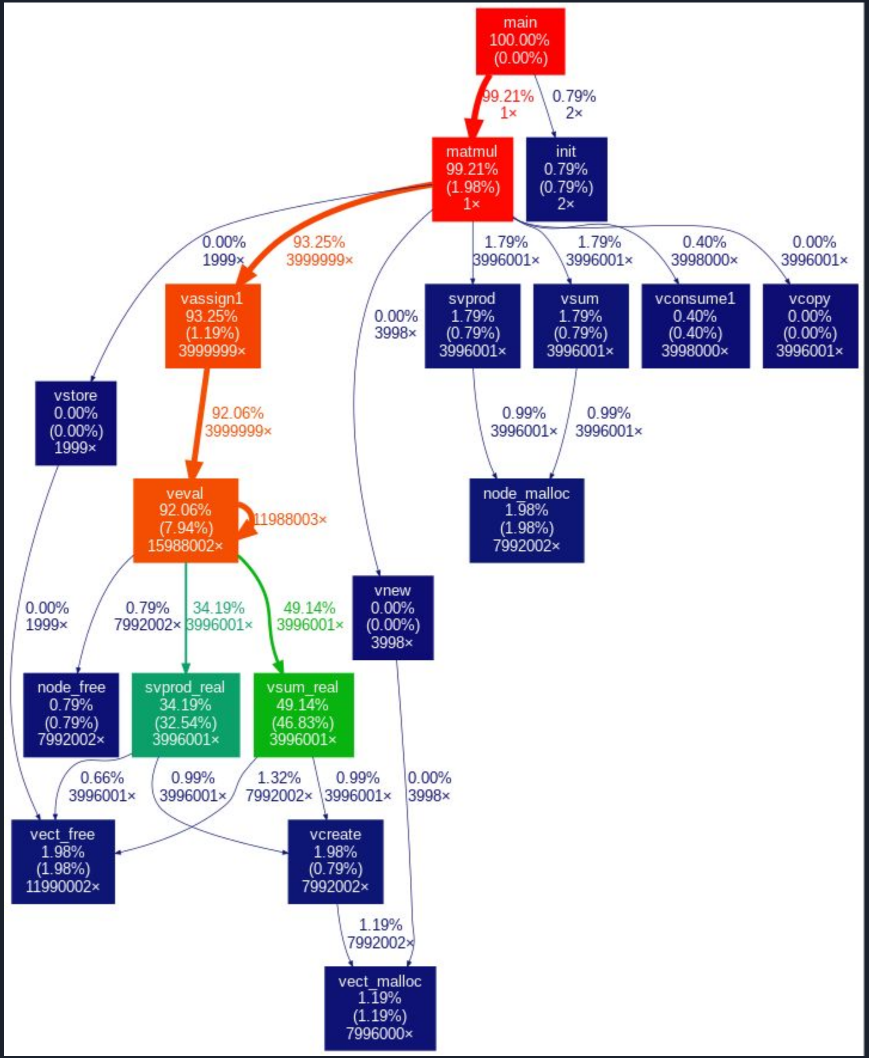
```c
enum operand_t {
  OP_FAIL=0,
  OP_VECT,
  OP_VSUM,
  OP_VPROD,
  OP_SVPROD
};

struct vect_private {
  size_t n;             //vector elemets in d[]
  size_t memsize;       //malloced memsize
  size_t references;    //reference counter
  size_t size_t_pad[1]; //padding to 32
  enum operand_t op;    //information type identifier
  enum operand_t operand_t_pad[7]; //padding to 32
  struct vect_private* a;   //vector operand a
  struct vect_private* b;   //vector operand b
  struct vect_private* vect_private_ptr_pad[2]; //padding to 32 byte
  double c;             //scalar operand c
  double double_pad[3]; //padding to 32 byte
  double d[4];          //vector data
} __attribute__((aligned(32)));
```

Lazy Evaluation

Metrik	Absolut	Relative
Cycles	6.918.121.173	386,3%
Instructions	14.822.838.134	324,2%



```c
static VPriv vcopy_real(Vector v);
static VPriv vsum_real(VPriv v1, VPriv v2);
static VPriv vprod_real(VPriv v1, VPriv v2);
static VPriv svprod_real(double d, VPriv v1);
static VPriv vcreate(size_t n);
static VPriv veval(VPriv v);
static inline struct vect_private* vect_malloc(size_t n);
static inline void vect_free(struct vect_private* p);
static inline struct vect_private* node_malloc(void);
static inline void node_free(struct vect_private* p);
```



main 100.00% (0.00%)
99.21% 1×
0.79% 2×
matmul 99.21% (1.98%) 1×
init 0.79% (0.79%) 2×
0.00% 1999×
93.25% 3999999×
1.79% 3996001×
1.79% 3996001×
0.40% 3998000×
0.00% 3996001×
vassign1 93.25% (1.19%) 3999999×
0.00% 3998×
svprod 1.79% (0.79%) 3996001×
vsum 1.79% (0.79%) 3996001×
vconsume1 0.40% (0.40%) 3998000×
vcopy 0.00% (0.00%) 3996001×
vstore 0.00% (0.00%) 1999×
92.06% 3999999×
0.99% 3996001×
0.99% 3996001×
veval 92.06% (7.94%) 15988002×
11988003×
node_malloc 1.98% (1.98%) 7992002×
vnew 0.00% (0.00%) 3998×
0.00% 1999×
0.79% 7992002×
34.19% 3996001×
49.14% 3996001×
node_free 0.79% (0.79%) 7992002×
svprod_real 34.19% (32.54%) 3996001×
vsum_real 49.14% (46.83%) 3996001×
0.66% 3996001×
0.99% 3996001×
1.32% 7992002×
0.99% 3996001×
0.00% 3998×
vect_free 1.98% (1.98%) 11990002×
vcreate 1.98% (0.79%) 7992002×
1.19% 7992002×
vect_malloc 1.19% (1.19%) 7996000×

But why the <insert swearword> did we do this?

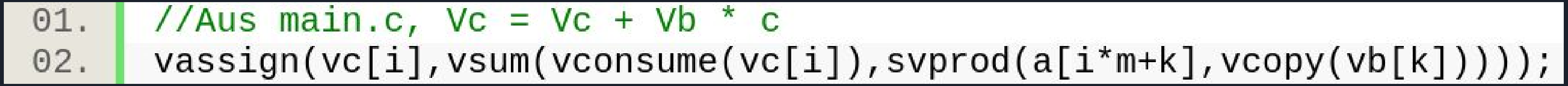
Math.

Vc = Vb * d;   (load d, load Vb, *, store Vc)

Va = Vc + Va;   (load Va, load Vc, +, store Va)

Vc = Vc + Vb * c (load Vc, load Vb, load c, *, + store Vc)

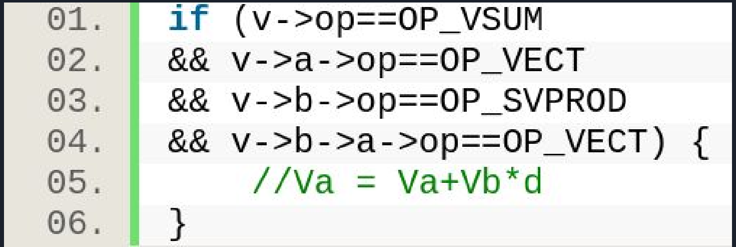
-> weniger Instruktionen, weniger Speicherzugriff



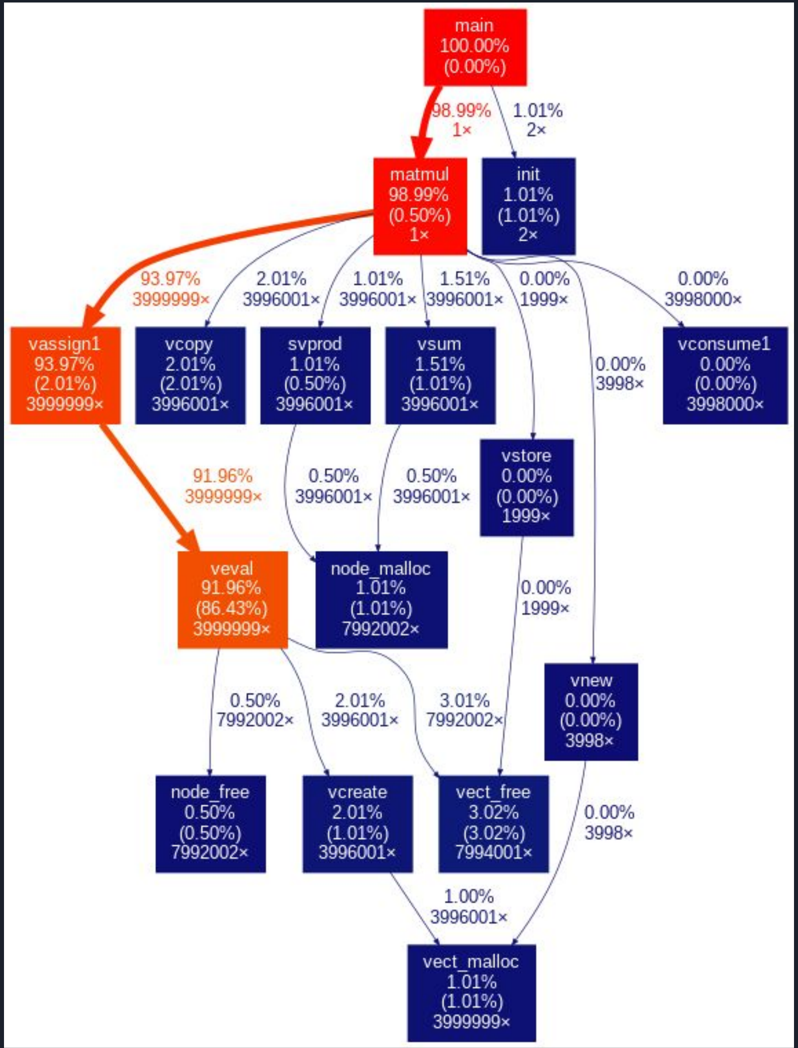
```
//Aus main.c, Vc = Vc + Vb * c
vassign(vc[i],vsum(vconsume(vc[i]),svprod(a[i*m+k],vcopy(vb[k]))));
```

Situational Optimization

Metrik	Absolut	Relative
Cycles	3.501.391.479	195,5%
Instructions	7.975.040.428	174,4%

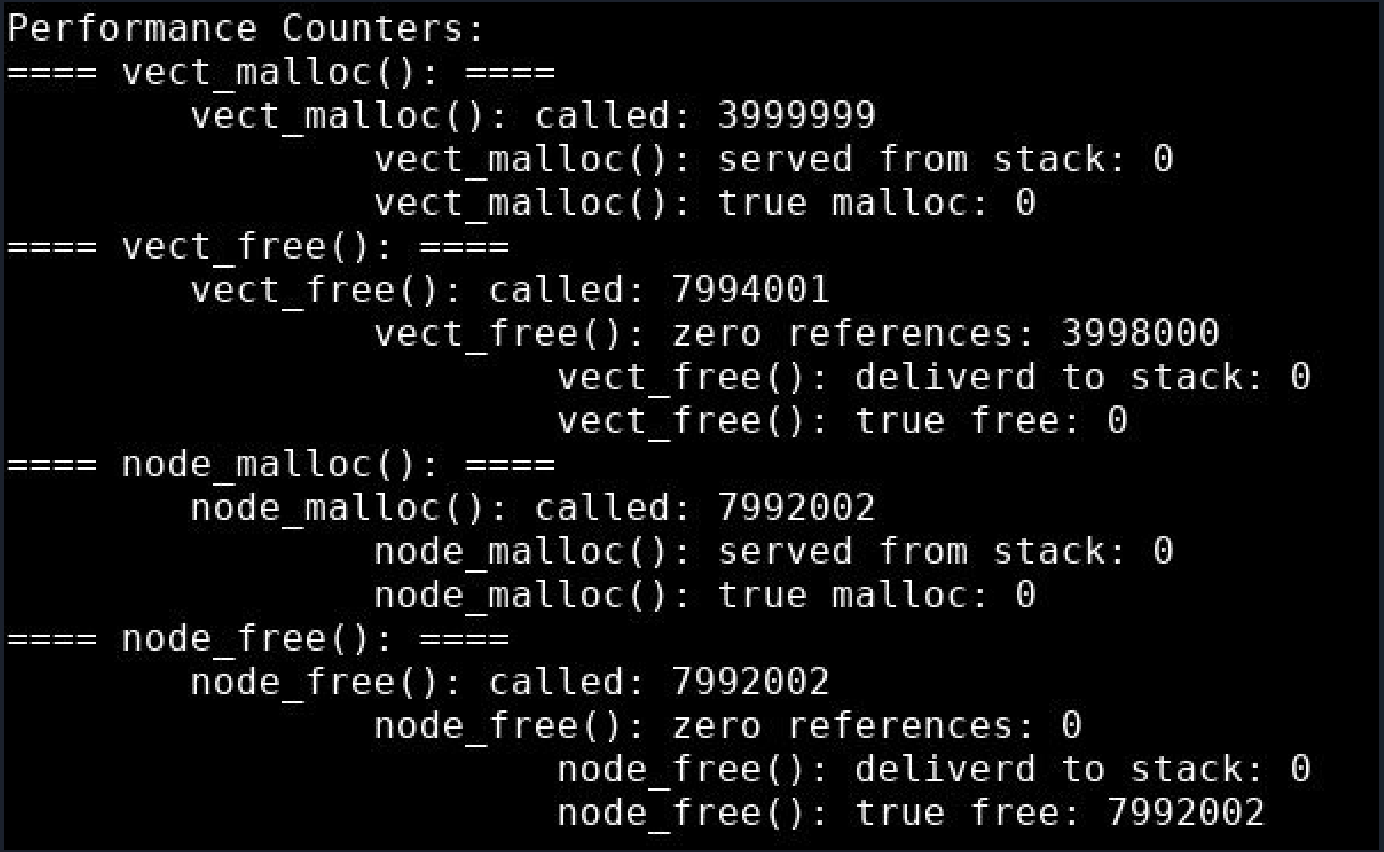


```
if (v->op==OP_VSUM
&& v->a->op==OP_VECT
&& v->b->op==OP_SVPROD
&& v->b->a->op==OP_VECT) {
     //Va = Va+Vb*d
}
```



main 100.00% (0.00%)
98.99% 1×
1.01% 2×
matmul 98.99% (0.50%) 1×
init 1.01% (1.01%) 2×
93.97% 3999999×
2.01% 3996001×
1.01% 3996001×
1.51% 3996001×
0.00% 1999×
0.00% 3998000×
vassign1 93.97% (2.01%) 3999999×
vcopy 2.01% (2.01%) 3996001×
svprod 1.01% (0.50%) 3996001×
vsum 1.51% (1.01%) 3996001×
0.00% 3998×
vconsume1 0.00% (0.00%) 3998000×
vstore 0.00% (0.00%) 1999×
91.96% 3999999×
0.50% 3996001×
0.50% 3996001×
veval 91.96% (86.43%) 3999999×
node_malloc 1.01% (1.01%) 7992002×
0.00% 1999×
vnew 0.00% (0.00%) 3998×
0.50% 7992002×
2.01% 3996001×
3.01% 7992002×
node_free 0.50% (0.50%) 7992002×
vcreate 2.01% (1.01%) 3996001×
vect_free 3.02% (3.02%) 7994001×
0.00% 3998×
1.00% 3996001×
vect_malloc 1.01% (1.01%) 3999999×

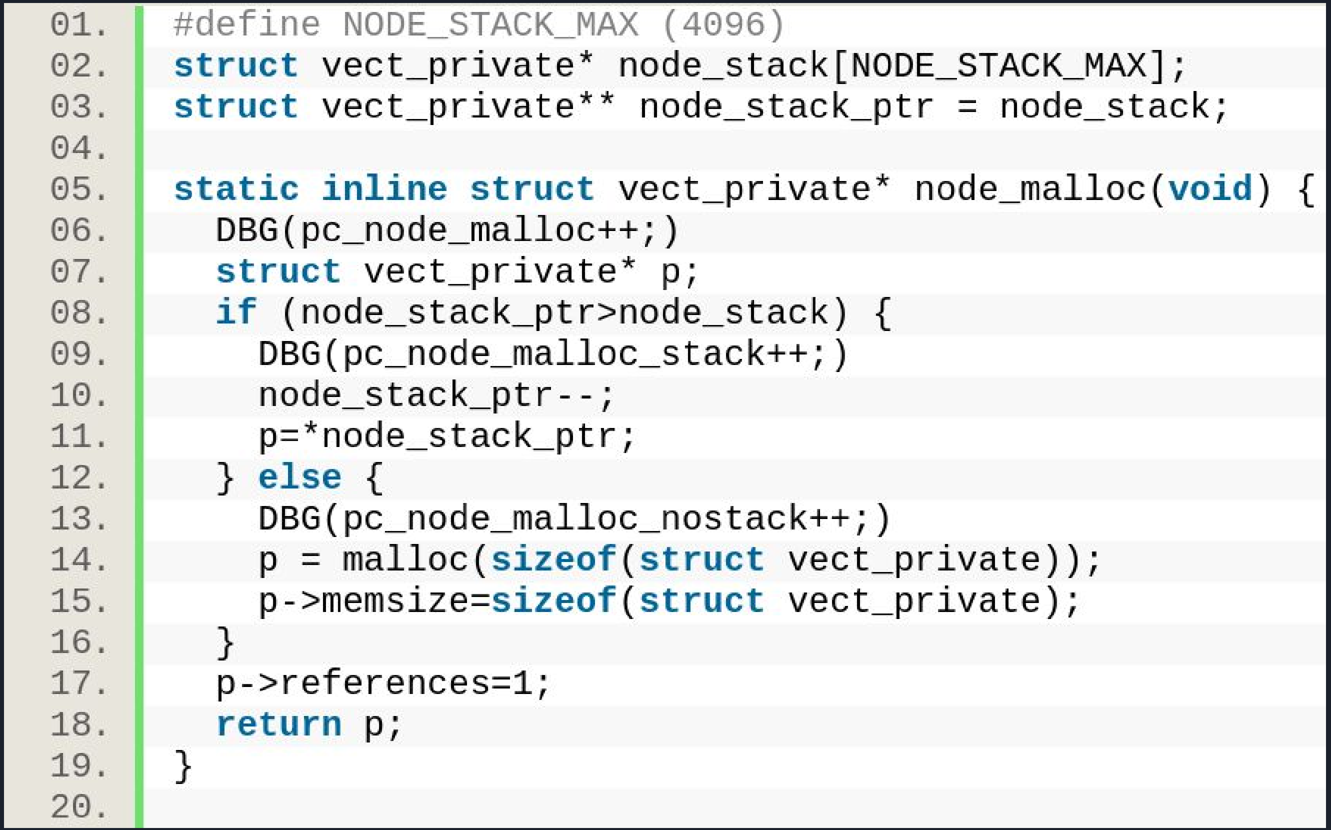
OK, richtige Richtung, Lets see ...



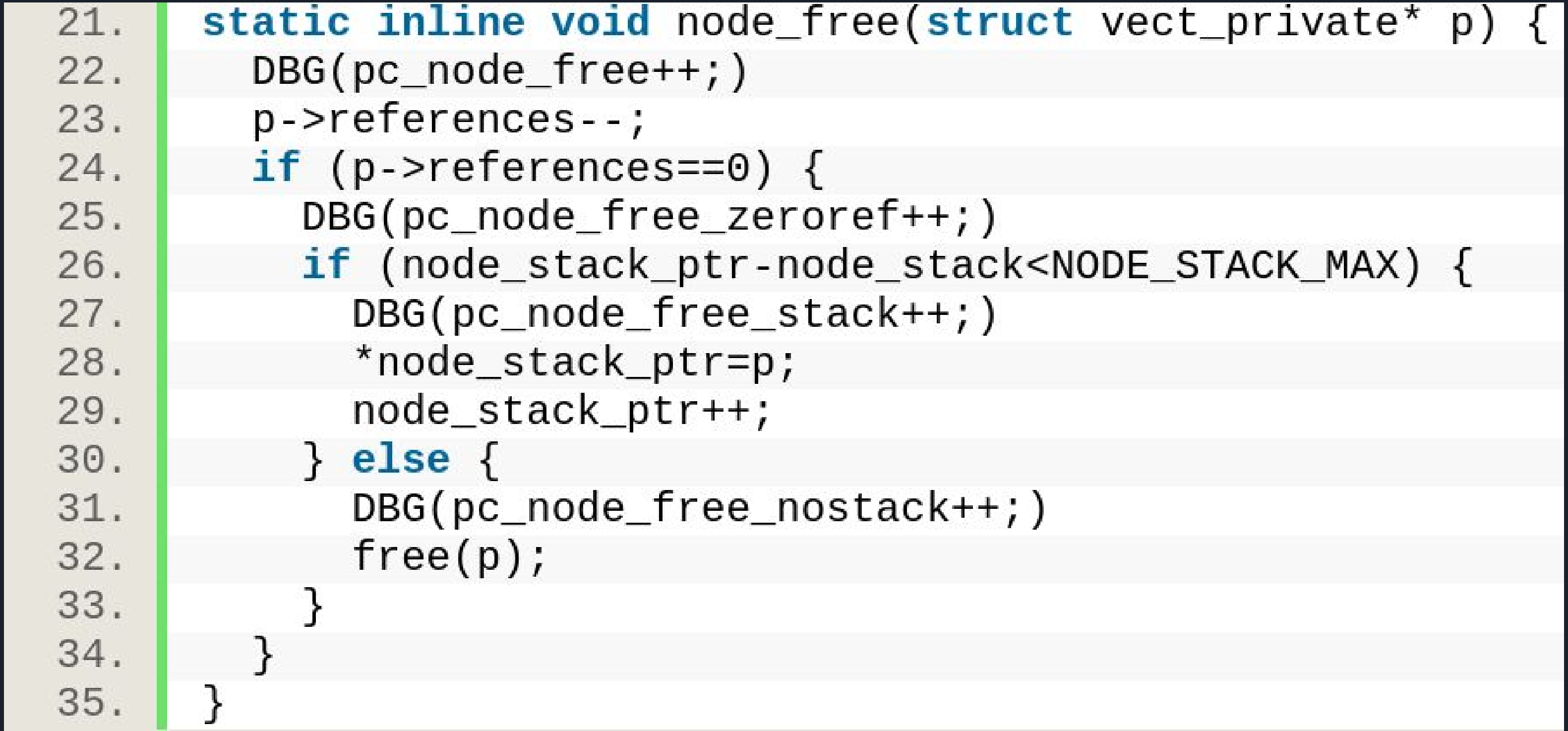
```
Performance Counters:
==== vect_malloc(): ====
        vect_malloc(): called: 3999999
                vect_malloc(): served from stack: 0
                vect_malloc(): true malloc: 0
==== vect_free(): ====
        vect_free(): called: 7994001
                vect_free(): zero references: 3998000
                        vect_free(): deliverd to stack: 0
                        vect_free(): true free: 0
==== node_malloc(): ====
        node_malloc(): called: 7992002
                node_malloc(): served from stack: 0
                node_malloc(): true malloc: 0
==== node_free(): ====
        node_free(): called: 7992002
                node_free(): zero references: 0
                        node_free(): deliverd to stack: 0
                        node_free(): true free: 7992002
```

malloc free malloc free malloc free malloc free malloc free malloc free malloc free …

- Analyse:
    - vcopy allokiert Speicher …
    - … für einzelnen Funktionsaufruf
    - Viele syscalls
- Erkenntnis:
    - kurzlebiger Speicher
    - ähnlicher oder identer Größe
    - 2 Arten (Nodes, Vectors)
- Memory Reuse!

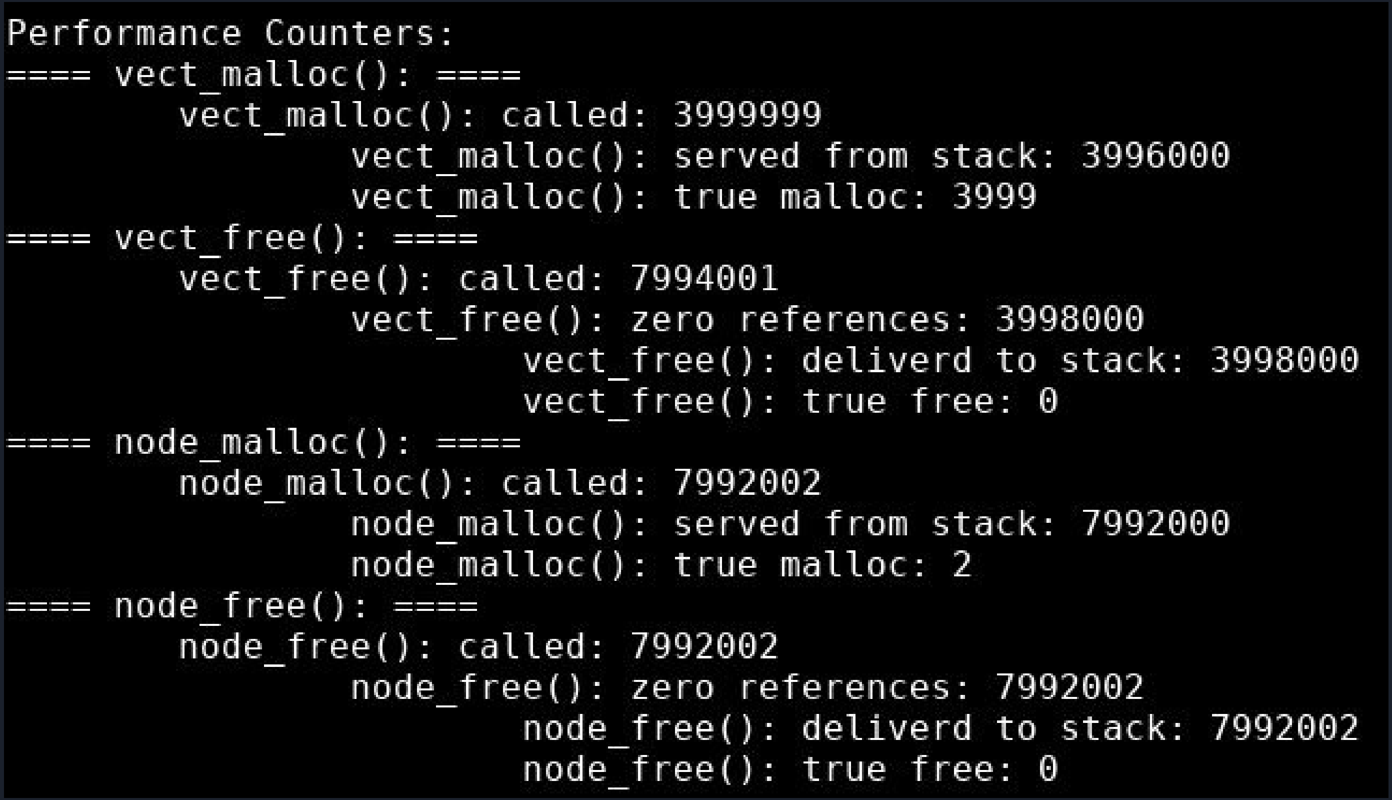


```c
#define NODE_STACK_MAX (4096)
struct vect_private* node_stack[NODE_STACK_MAX];
struct vect_private** node_stack_ptr = node_stack;

static inline struct vect_private* node_malloc(void) {
  DBG(pc_node_malloc++;)
  struct vect_private* p;
  if (node_stack_ptr>node_stack) {
    DBG(pc_node_malloc_stack++;)
    node_stack_ptr--;
    p=*node_stack_ptr;
  } else {
    DBG(pc_node_malloc_nostack++;)
    p = malloc(sizeof(struct vect_private));
    p->memsize=sizeof(struct vect_private);
  }
  p->references=1;
  return p;
}

```



```c
static inline void node_free(struct vect_private* p) {
  DBG(pc_node_free++;)
  p->references--;
  if (p->references==0) {
    DBG(pc_node_free_zeroref++;)
    if (node_stack_ptr-node_stack<NODE_STACK_MAX) {
      DBG(pc_node_free_stack++;)
      *node_stack_ptr=p;
      node_stack_ptr++;
    } else {
      DBG(pc_node_free_nostack++;)
      free(p);
    }
  }
}
```

Stacked.



```
Performance Counters:
==== vect_malloc(): ====
        vect_malloc(): called: 3999999
                vect_malloc(): served from stack: 3996000
                vect_malloc(): true malloc: 3999
==== vect_free(): ====
        vect_free(): called: 7994001
                vect_free(): zero references: 3998000
                        vect_free(): deliverd to stack: 3998000
                        vect_free(): true free: 0
==== node_malloc(): ====
        node_malloc(): called: 7992002
                node_malloc(): served from stack: 7992000
                node_malloc(): true malloc: 2
==== node_free(): ====
        node_free(): called: 7992002
                node_free(): zero references: 7992002
                        node_free(): deliverd to stack: 7992002
                        node_free(): true free: 0
```

Memory Stacks

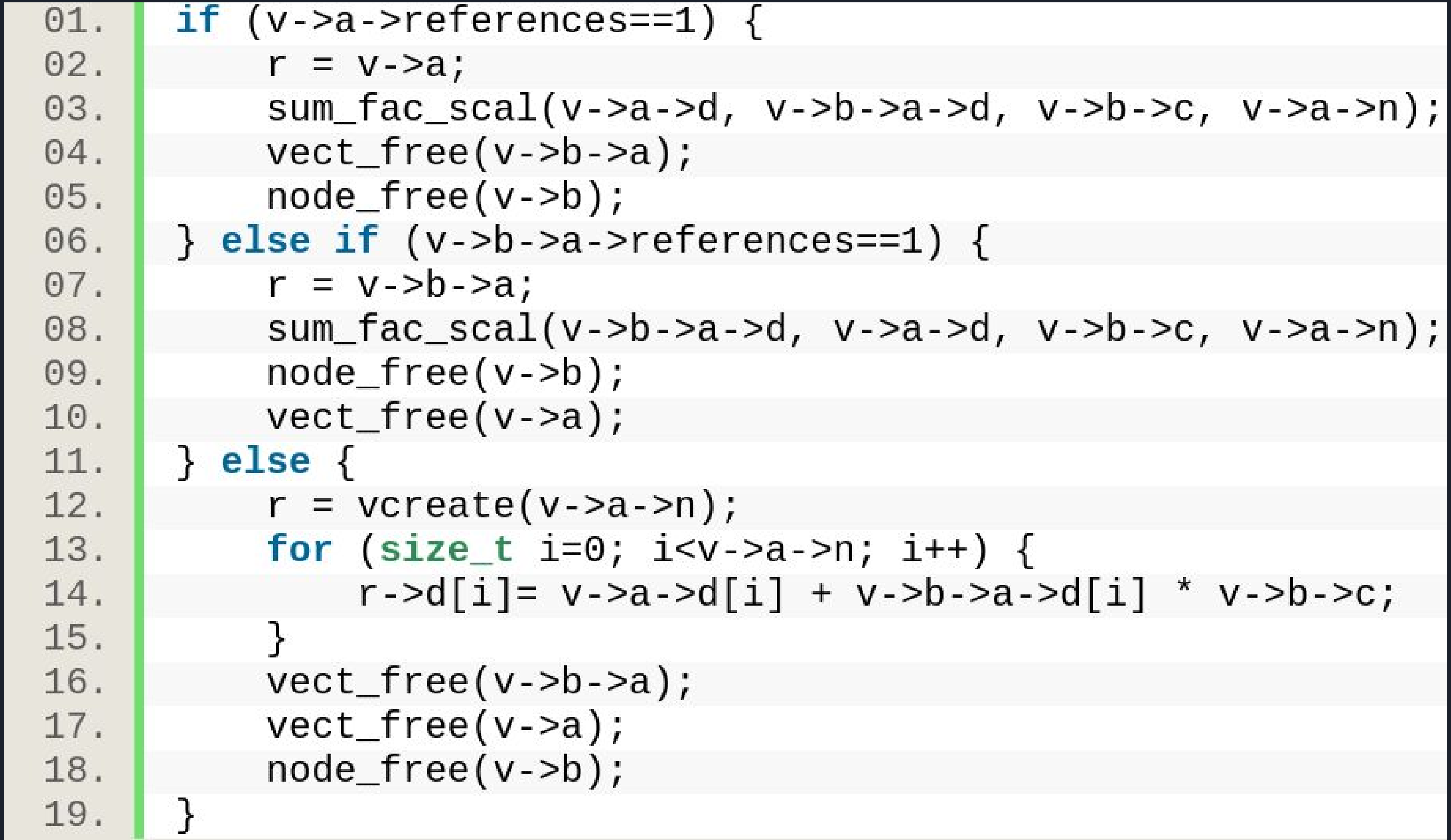
Vector Stack		
Metrik	Absolut	Relative
Cycles	1.930.073.190	107,8%
Instructions	4.330.685.118	94,7%

Node Stack		
Metrik	Absolut	Relative
Cycles	2.122.996.398	118,6%
Instructions	4.645.897.212	101,6%

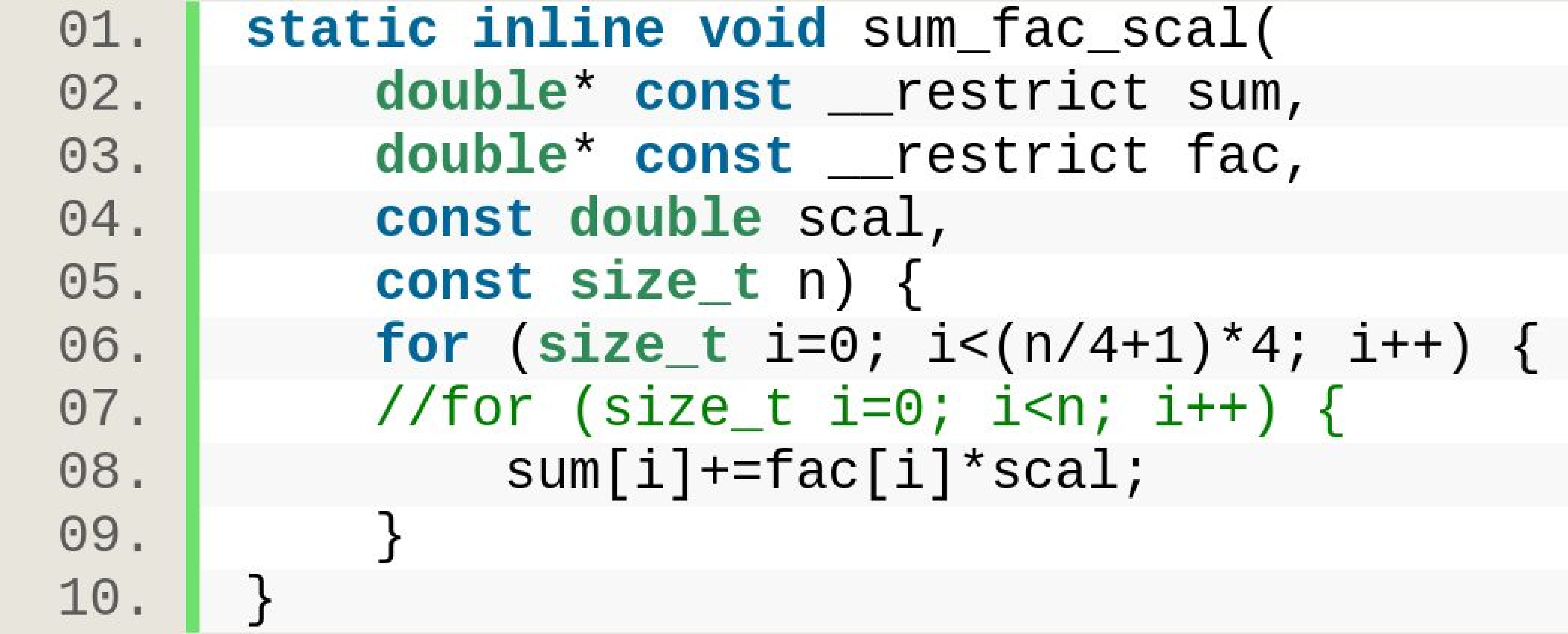
Vector & Node Stacks		
Metrik	Absolut	Relative
Cycles	868.780.341	48,5%
Instructions	1.613.760.109	35,3%

What now?

- Minor Optimizations
  - Vector Wiederverwendung
  - Multiple-Of-4



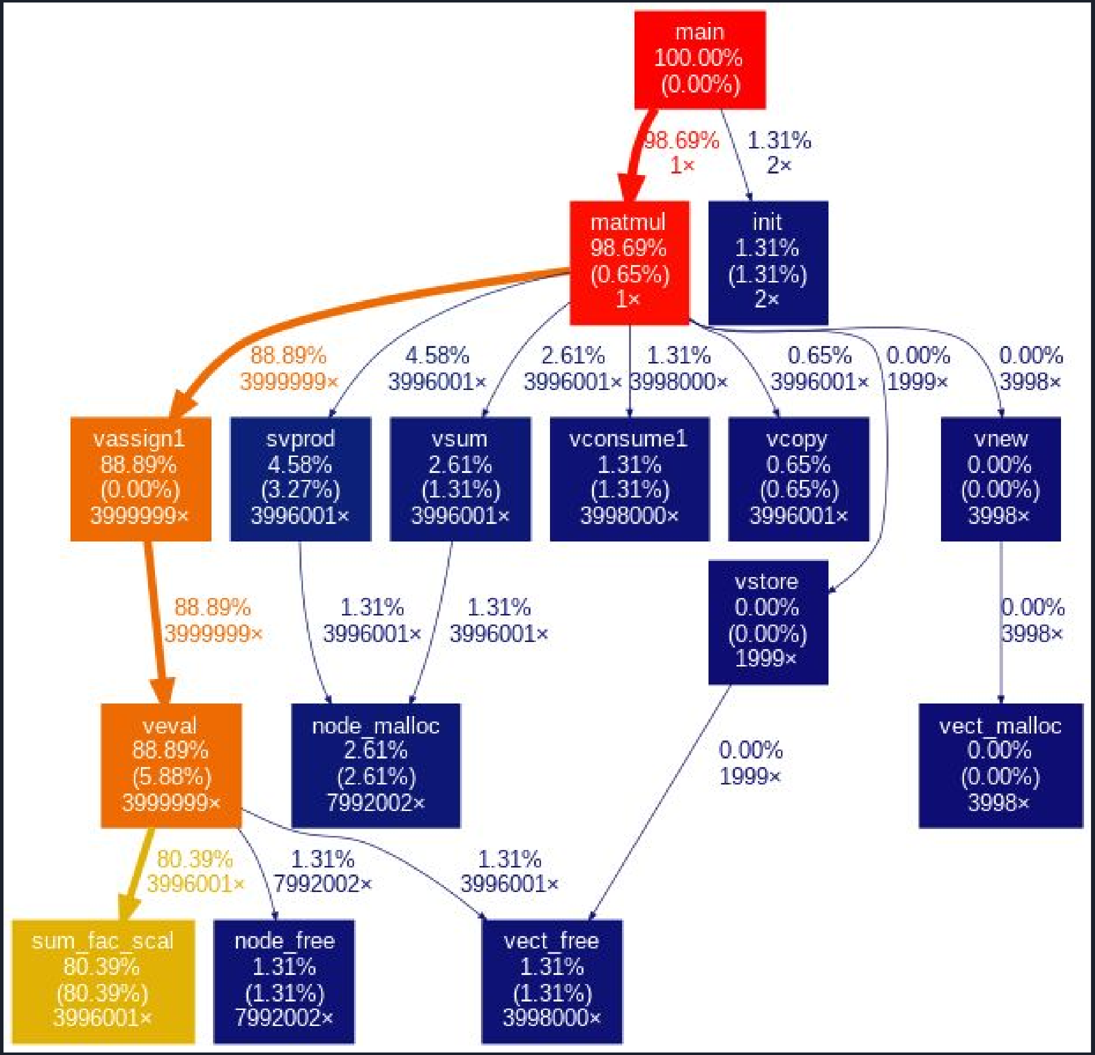
```c
if (v->a->references==1) {
    r = v->a;
    sum_fac_scal(v->a->d, v->b->a->d, v->b->c, v->a->n);
    vect_free(v->b->a);
    node_free(v->b);
} else if (v->b->a->references==1) {
    r = v->b->a;
    sum_fac_scal(v->b->a->d, v->a->d, v->b->c, v->a->n);
    node_free(v->b);
    vect_free(v->a);
} else {
    r = vcreate(v->a->n);
    for (size_t i=0; i<v->a->n; i++) {
        r->d[i]= v->a->d[i] + v->b->a->d[i] * v->b->c;
    }
    vect_free(v->b->a);
    vect_free(v->a);
    node_free(v->b);
}
```



```
static inline void sum_fac_scal(
    double* const __restrict sum,
    double* const __restrict fac,
    const double scal,
    const size_t n) {
    for (size_t i=0; i<(n/4+1)*4; i++) {
    //for (size_t i=0; i<n; i++) {
        sum[i]+=fac[i]*scal;
    }
}
```

Result

Minor Optimizations		
Metrik	Absolut	Relative
Cycles	816.896.085	45,6%
Instructions	1.765.960.904	38,6%



Weitere.... Möglichkeiten?

- Insignificant Optimizations
  - Entfernung von Sicherheitschecks
- Verworfene Ansätze
  - Keine Nodes mehr -> Vektoren + 1 Referenz
  - Verzögerte Evaluierung (Problematisch!)

Fertig? Fast.
Now Unroll (All) the loops.

- -funroll-all-loops
  - Führt loop-unrolling auch bei Schleifen ohne hinreichende Indizien aus.
- Ähnlich mit profiling
  - -fprofile-generate
  - -fprofile-use

Compiler Optionen

Loop Unrolling		
Metrik	Absolut	Relative
Cycles	767.622.685	42,9%
Instructions	1.413.462.136	30,9%

Profiling		
Metrik	Absolut	Relative
Cycles	743.195.155	41,5%
Instructions	1.384.962.747	30,3%

Endresultat / Vergleich

Metrik	Basis Wert	Absolut Wert	relativ
Cycles	1.790.767.358	767.865.713	42,9%
Instructions	4.571.583.802	1.415.842.439	30,9%
Branch Misses	29.314	24.542	83,7%
Cache Misses	1.127.025	1.060.918	94,1%
Exec. Time	0,545311722 sec	0,242798895 sec	44,5%

Speedup: 2,25

Fragen?