https://lquenti.de

Lars Quentin

(Technische) Lügen des Informatikstudiums

Lies-to-children des Studiums

Table of contents

Grundannahmen

- Durch meine Interessen hat es einen Performance bias
- Um mehr zu verstehen was passiert nehmen wir C (Sorry)
- Bei Verständnisfragen: Einfach reinrufen oder melden

Variablendefinition

Was passiert hier?

Schweres Codebeispiel

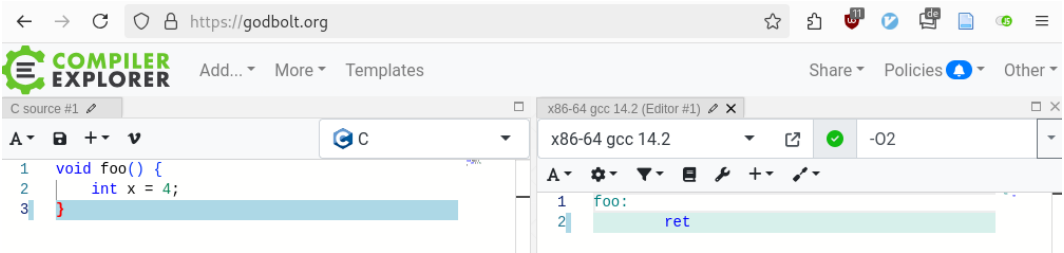
```
1  void foo() { int x = 4; }
```

Variablendefinition

Was passiert hier?

Schweres Codebeispiel

```
1  void foo() { int x = 4; }
```



Variablendefinition: Versuch 2

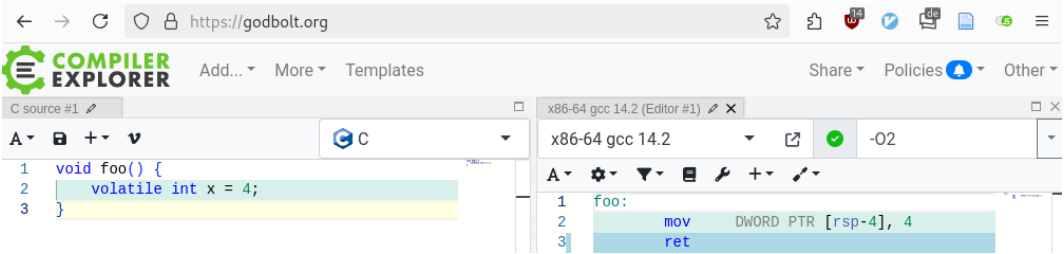
Schweres Codebeispiel

```
1  void foo() { volatile int x = 4; }
```

Variablendefinition: Versuch 2

Schweres Codebeispiel

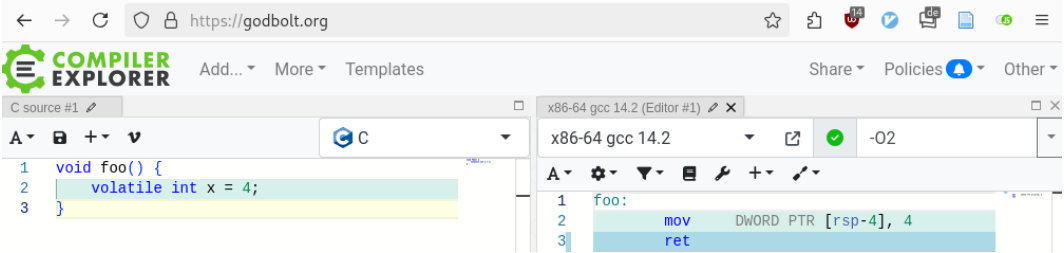
```
1  void foo() { volatile int x = 4; }
```



Variablendefinition: Versuch 2 (cont.)

Was uns beigebracht wird

- Der Stack pointer ist ein Pointer auf eine Stelle **im RAM**
- 4 bytes unter dem Pointer wird implizit "reserviert"
- Der Prozessor schreibt diese Zahl **an diese Addresse im RAM**



Virtual Memory

- Jeder Prozess denkt, er wäre im Memory alleine

Virtual Memory

- Jeder Prozess denkt, er wäre im Memory alleine
- Der Virtuelle Addressbereich wird in echten Addressbereich umgewandelt
  - Vom *Memory Management Unit (MMU)*, Teil des CPUs
  - Wichtige Übersetzungen werden gecached im *Translation Lookaside Buffer (TLB)*

Virtual Memory

- Jeder Prozess denkt, er wäre im Memory alleine
- Der Virtuelle Addressbereich wird in echten Addressbereich umgewandelt
  - Vom *Memory Management Unit (MMU)*, Teil des CPUs
  - Wichtige Übersetzungen werden gecached im *Translation Lookaside Buffer (TLB)*
- Menge Vorteile: Security, Swapping, Easier Programming, Overbooking

Variablendefinition: Versuch 3

Schweres Codebeispiel

```
1  volatile int x = 4;
```

Was passiert

- 4 bytes unter dem Pointer wird implizit "reserviert"
- Der Prozessor **schreibt** die 4 bytes in den RAM

Variablendefinition: Versuch 3

Schweres Codebeispiel

```
1  volatile int x = 4;
```

Was passiert

- 4 bytes unter dem Pointer wird implizit "reserviert"
- Der Prozessor **schreibt** die 4 bytes in den RAM

Schreibt er immer wirklich **alle** Bytes?

Quiz: Wie viel Memory braucht dieses Programm?

```c
int main() {
    void *ptr;
    // Allocate 8KB until we have 8GB
    for (int i=0; i<1024*1024; ++i) {
      ptr = malloc((size_t)(8*1024));
    }
    printf("allocation done\n");
    while (true) sleep(1);
}
```

Quiz: Wie viel Memory braucht dieses Programm?

```c
int main() {
    void *ptr;
    // Allocate 8KB until we have 8GB
    for (int i=0; i<1024*1024; ++i) {
      ptr = malloc((size_t)(8*1024));
    }
    printf("allocation done\n");
    while (true) sleep(1);
}
```

- Mehr als 0 byte

Quiz: Wie viel Memory braucht dieses Programm?

```c
int main() {
    void *ptr;
    // Allocate 8KB until we have 8GB
    for (int i=0; i<1024*1024; ++i) {
      ptr = malloc((size_t)(8*1024));
    }
    printf("allocation done\n");
    while (true) sleep(1);
}
```

- Mehr als 0 byte
- Nicht viel mehr als 8GB

Quiz: Wie viel Memory braucht dieses Programm?

```c
int main() {
    void *ptr;
    // Allocate 8KB until we have 8GB
    for (int i=0; i<1024*1024; ++i) {
      ptr = malloc((size_t)(8*1024));
    }
    printf("allocation done\n");
    while (true) sleep(1);
}
```

- Mehr als 0 byte
- Nicht viel mehr als 8GB
- Mehr können wir nicht sagen
  - In unserem Fall ca 4GB

Und wie viel Platz braucht DIESES Programm?

```c
int main() {
    void *ptr = malloc((size_t)(8ll*1024*1024*1024));
    if (!ptr) return 1;
    printf("allocation done\n");

    while (1) {
        printf("Running...\n");
        sleep(1); // Pause for 1 second
    }
}
```

Und wie viel Platz braucht DIESES Programm?

```c
int main() {
    void *ptr = malloc((size_t)(8ll*1024*1024*1024));
    if (!ptr) return 1;
    printf("allocation done\n");

    while (1) {
        printf("Running...\n");
        sleep(1); // Pause for 1 second
    }
}
```

- braucht 800 byte

Und wie viel Platz braucht DIESES Programm?

```c
int main() {
    void *ptr = malloc((size_t)(8ll*1024*1024*1024));
    if (!ptr) return 1;
    printf("allocation done\n");

    while (1) {
        printf("Running...\n");
        sleep(1); // Pause for 1 second
    }
}
```

- braucht 800 byte
- Nennt sich *Overcommitment*
- Siehe `www.kernel.org/doc/Documentation/vm/overcommit-accounting`
- Behebbar durch `memset`

Aside: Wie groß ist mein Struct?

```
1  struct MyStruct {
2    int A;
3    long B;
4    char C;
5  };
```

Möglichkeiten:

- 5 byte
- 13 byte
- 24 byte
- Spezifiziert der C Standard nicht...

Aside: Wie groß ist mein Struct? (2)

```
1  struct MyStruct {
2    int32_t A;
3    int64_t B;
4    int8_t C;
5  };
```

Möglichkeiten:

- 5 byte
- 13 byte
- 24 byte
- Spezifiziert der C Standard nicht...

Aside: Wie groß ist mein Struct **auf meinem Rechner mit gcc**?

```
1  struct MyStruct {
2    int32_t A;
3    int64_t B;
4    int8_t C;
5  };
```

Möglichkeiten:

- 5 byte
- 13 byte
- 24 byte
- ~~Spezifiziert der C Standard nicht...~~

Aside: Wie groß ist mein Struct **auf meinem Rechner mit gcc**?

```
1  struct MyStruct { int32_t A; int64_t B; int8_t C; };
```

- 24 byte!

Aside: Wie groß ist mein Struct **auf meinem Rechner mit gcc**?

```
1  struct MyStruct { int32_t A; int64_t B; int8_t C; };
```

- 24 byte!

Alignment

- Prozessoren können n-byte Zahlen am schnellsten n-byte aligned lesen
- Structs sind aligned mit dem größten Element-Alignment (8)
- Das Kerneltool `Pahole` hilft euch beim Finden

Aside: Wie groß ist mein Struct **auf meinem Rechner mit gcc**?

```
1  struct MyStruct { int32_t A; int64_t B; int8_t C; };
```

- 24 byte!

Alignment

- Prozessoren können n-byte Zahlen am schnellsten n-byte aligned lesen
- Structs sind aligned mit dem größten Element-Alignment (8)
- Das Kerneltool `Pahole` hilft euch beim Finden

Back to topic

Noch Schwereres Codebeispiel

```
1  volatile int x = 4;
2  // hier kann irgendwas anderes auch passieren, oder auch nicht, schon okay...
3  printf("%d\n", x);
```

Was passiert

- 4 bytes werden im virtual memory des Prozesses reserviert
  - Hier ist er auch wirklich da!
- Das virtual memory wird dann auf dem RAM gemapped
- Dann wird im `printf` x wieder aus dem RAM gelesen

Back to topic

Noch Schwereres Codebeispiel

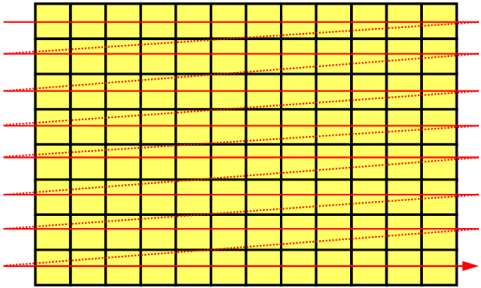
```c
volatile int x = 4;
// hier kann irgendwas anderes auch passieren, oder auch nicht, schon okay...
printf("%d\n", x);
```

Was passiert

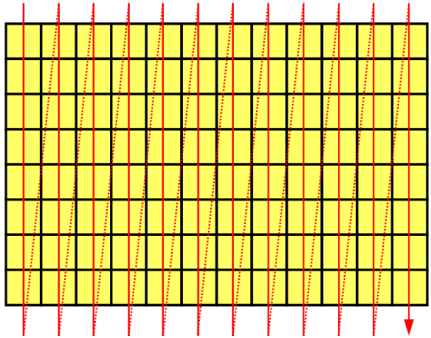
- 4 bytes werden im virtual memory des Prozesses reserviert
  - Hier ist er auch wirklich da!
- Das virtual memory wird dann auf dem RAM gemapped
- Dann wird im `printf` x wieder aus dem RAM gelesen

LIEST ER WIRKLICH AUS DEM RAM?

Column vs Row major (Stolen from Scott Myers)



Row Major



Column Major

Experiment: Liest er wie wir denken aus dem RAM?

Row order

```
#define M 10000
#define N 10000
void add_row(double *arr,double c){
  for (int i=0; i<M; ++i)
    for (int j=0; j<N; ++j)
      arr[i*N+j]+=c;
}
```

Column Order

```
#define M 10000
#define N 10000
void add_col(double *arr,double c){
  for (int j=0; j<N; ++j)
    for (int i=0; i<M; ++i)
      arr[i*N+j]+=c;
}
```

Experiment: Liest er wie wir denken aus dem RAM?

Row order

```
#define M 10000
#define N 10000
void add_row(double *arr,double c){
  for (int i=0; i<M; ++i)
    for (int j=0; j<N; ++j)
      arr[i*N+j]+=c;
}
```

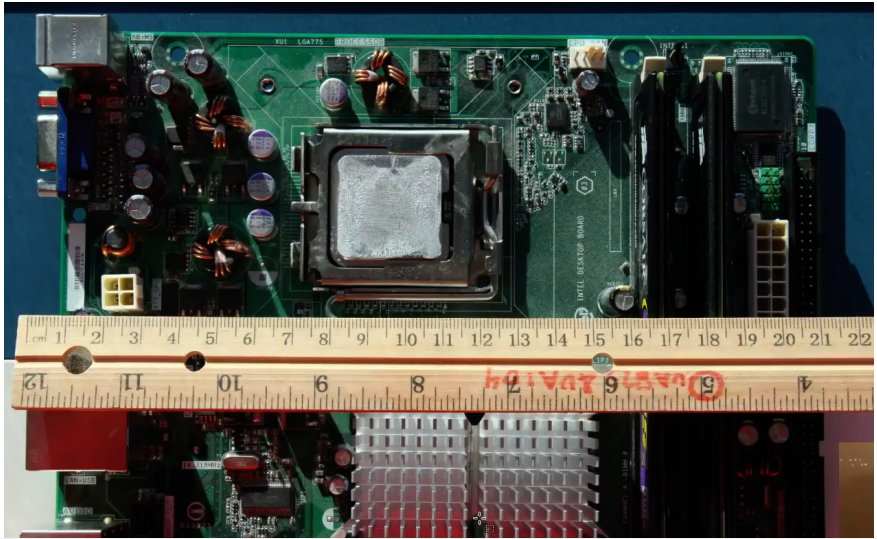
Column Order

```
#define M 10000
#define N 10000
void add_col(double *arr,double c){
  for (int j=0; j<N; ++j)
    for (int i=0; i<M; ++i)
      arr[i*N+j]+=c;
}
```

	Row-Major	**Column-Major**
Unoptimiert	214.6 ms	541.6 ms
Optimiert (O2)	114.9 ms	513.5 ms

Es liegt an RAM. Was ist das RAM-Problem?

Problem mit RAM (Idee von Casey Muratori)



Problem mit RAM (Idee von Casey Muratori)

- Formel für Datentransfer: Latenz + Size/Throughput + Processing

Problem mit RAM (Idee von Casey Muratori)

- Formel für Datentransfer: Latenz + Size/Throughput + Processing
- Um eine lower bound zu berechnen:

Problem mit RAM (Idee von Casey Muratori)

- Formel für Datentransfer: Latenz + Size/Throughput + Processing
- Um eine lower bound zu berechnen:
  - $\lim_{\text{Throughput} \to \infty}$

Problem mit RAM (Idee von Casey Muratori)

- Formel für Datentransfer: Latenz + Size/Throughput + Processing
- Um eine lower bound zu berechnen:
  - $\lim_{\text{Throughput} \to \infty}$
  - $\lim_{\text{Processing} \to 0}$

Problem mit RAM (Idee von Casey Muratori)

- Formel für Datentransfer: Latenz + Size/Throughput + Processing
- Um eine lower bound zu berechnen:
  - $\lim_{\text{Throughput} \to \infty}$
  - $\lim_{\text{Processing} \to 0}$
  - Wir berechnen die Luftlinie

Problem mit RAM (Idee von Casey Muratori)

- Formel für Datentransfer: Latenz + Size/Throughput + Processing
- Um eine lower bound zu berechnen:
  - $\lim_{\text{Throughput} \to \infty}$
  - $\lim_{\text{Processing} \to 0}$
  - Wir berechnen die Luftlinie
  - Wir übertragen so schnell wie möglich (Lichtgeschwindigkeit)

Problem mit RAM (Idee von Casey Muratori)

- Formel für Datentransfer: Latenz + Size/Throughput + Processing
- Um eine lower bound zu berechnen:
  - $\lim_{\text{Throughput} \to \infty}$
  - $\lim_{\text{Processing} \to 0}$
  - Wir berechnen die Luftlinie
  - Wir übertragen so schnell wie möglich (Lichtgeschwindigkeit)
- Beispiel vom Screenshot: $6cm$ Distanz (one-way), $3.2GHz$ Taktrate

Problem mit RAM (Idee von Casey Muratori)

- Formel für Datentransfer: Latenz + Size/Throughput + Processing
- Um eine lower bound zu berechnen:
  - $\lim_{\text{Throughput} \to \infty}$
  - $\lim_{\text{Processing} \to 0}$
  - Wir berechnen die Luftlinie
  - Wir übertragen so schnell wie möglich (Lichtgeschwindigkeit)
- Beispiel vom Screenshot: $6cm$ Distanz (one-way), $3.2GHz$ Taktrate
- Wie lange 1 bit nur an Distanz braucht:

$$\frac{299,792,458m/s}{3.2GHz} \approx 9.36cm < 2 \cdot 6cm$$

Problem mit RAM (Idee von Casey Muratori)

- Formel für Datentransfer: Latenz + Size/Throughput + Processing
- Um eine lower bound zu berechnen:
  - $\lim_{\text{Throughput} \to \infty}$
  - $\lim_{\text{Processing} \to 0}$
  - Wir berechnen die Luftlinie
  - Wir übertragen so schnell wie möglich (Lichtgeschwindigkeit)
- Beispiel vom Screenshot: $6cm$ Distanz (one-way), $3.2GHz$ Taktrate
- Wie lange 1 bit nur an Distanz braucht:

$$\frac{299,792,458m/s}{3.2GHz} \approx 9.36cm < 2 \cdot 6cm$$

SOMIT ZU LANGSAM

Lösung: Cache Hierarchien und Zugriffe

Holly Cummins
@holly_cummins

L1 cache is a beer in hand, L3 is fridge, main memory is walking to the store, disk access is flying to another country for beer. @net0pyr

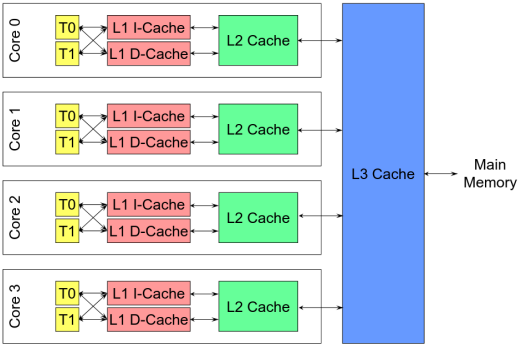
5:52 PM · Nov 6, 2014 · Twitter for iPhone

1.5K Retweets **962** Likes

Lösung: Cache Hierarchien und Zugriffe

Caches

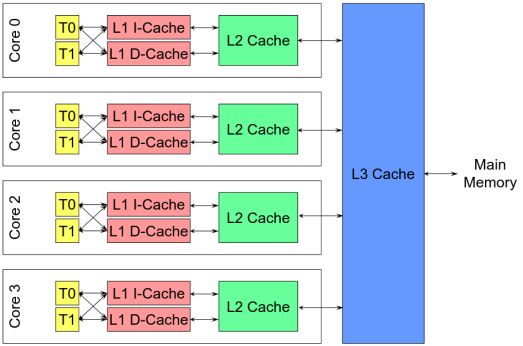
- Nah an den Prozessor, ultra schnell, ultra klein
- Mehrere Level
- Mein Laptop (i5-10210U)
- Level 1: 64 KiB pro Core (4)
- Level 2: 1 MiB pro Core (4)
- Level 3: 6 MiB (1)
- Selbst herausfinden: `lscpu`



Lösung: Cache Hierarchien und Zugriffe

Caches

- Nah an den Prozessor, ultra schnell, ultra klein
- Mehrere Level
- Mein Laptop (i5-10210U)
- Level 1: 64 KiB pro Core (4)
- Level 2: 1 MiB pro Core (4)
- Level 3: 6 MiB (1)
- Selbst herausfinden: `lscpu`



Memory-Zugriff: Erst L1, dann L2, dann L3, dann Main Memory...

Extralüge: Code wird Zeile für Zeile ausgefürt

Ist eine Lüge, da:

- Pipelining
- Superskalare Prozessoren
- Spekulative Ausführung
- Memory Ordering

Extralüge: Code wird Zeile für Zeile ausgefürt

Ist eine Lüge, da:

- Pipelining
- Superskalare Prozessoren
- Spekulative Ausführung
- Memory Ordering

Halb gelogen, siehe “Parallel Computing” für eine verbose Einführung...

Zusammenfassend: Bisher aufgedeckt

- Memory wird nicht unbedingt angelegt wenn du fragst
- Variablen werden nicht unbedingt komplett im Memory gespeichert
- Structs/Klassen sind nicht so groß wie die Summe ihrer Elemente
- Variablen werden im Optimalfall nicht aus dem RAM gelesen
- Code wird nicht Zeile für Zeile ausgeführt

Algos und Perf: Lügen die wir nun aufdecken werden

Algos und Perf: Lügen die wir nun aufdecken werden

- Asymptotic Complexity ist ein guter Referenzwert für Performance
  - Schreib doch dein high performance code in Python

Algos und Perf: Lügen die wir nun aufdecken werden

- Asymptotic Complexity ist ein guter Referenzwert für Performance
  - Schreib doch dein high performance code in Python
- Linked Lists sind ernstzunehmend
  - Gleicher Grund: Binary Trees sind ernstzunehmend

Algos und Perf: Lügen die wir nun aufdecken werden

- Asymptotic Complexity ist ein guter Referenzwert für Performance
  - Schreib doch dein high performance code in Python
- Linked Lists sind ernstzunehmend
  - Gleicher Grund: Binary Trees sind ernstzunehmend
- Instruktionen zählen ist ein guter Referenzwert für Performance

Algos und Perf: Lügen die wir nun aufdecken werden

- Asymptotic Complexity ist ein guter Referenzwert für Performance
  - Schreib doch dein high performance code in Python
- Linked Lists sind ernstzunehmend
  - Gleicher Grund: Binary Trees sind ernstzunehmend
- Instruktionen zählen ist ein guter Referenzwert für Performance
  - Wenn noch Zeit: Weniger Arbeit ist *immer* besser

Array vs Linked List

Array

- Ein großer Block
- Fixe Größe
  - Größe verändern: Reallocation
- $\mathcal{O}(1)$ random access:
  `xs[i] == *(xs+i)`
- $\mathcal{O}(1)$ insert/delete (am Ende)
  - Amortisiert tho
- $\mathcal{O}(n)$ insert/delete (nicht am Ende)
- Insert front: Immer worst case

Array vs Linked List

Array

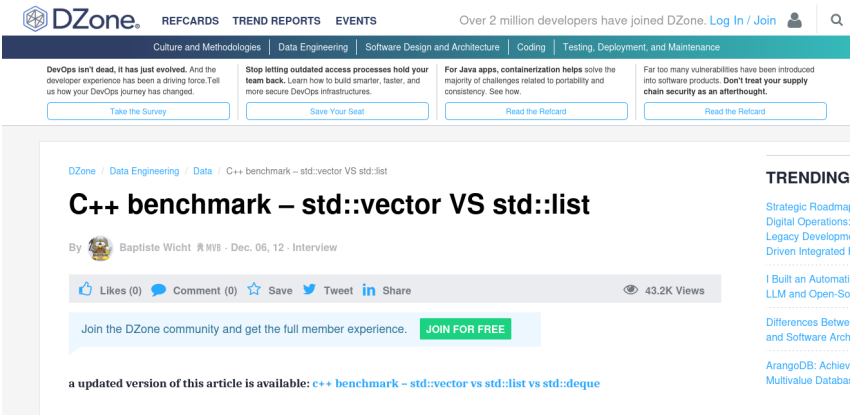
- Ein großer Block
- Fixe Größe
  - Größe verändern: Reallocation
- $\mathcal{O}(1)$ random access:
  `xs[i] == *(xs+i)`
- $\mathcal{O}(1)$ insert/delete (am Ende)
  - Amortisiert tho
- $\mathcal{O}(n)$ insert/delete (nicht am Ende)
- Insert front: Immer worst case

Linked List

```
1  template <typename T>
2  struct LinkedList {
3    T element;
4    LinkedList<T> *next;
5  }
```

- Größe on Demand
- Einfach weiteren Node allocaten
- $\mathcal{O}(n)$ random access
- $\mathcal{O}(1)$ insert delete (immer)
  - ((wenn du bereits da bist))

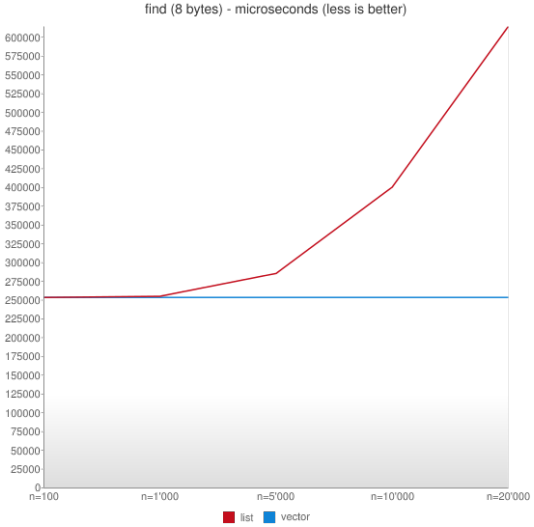
Array vs Linked List: Benchmarks (Alles geklaut)



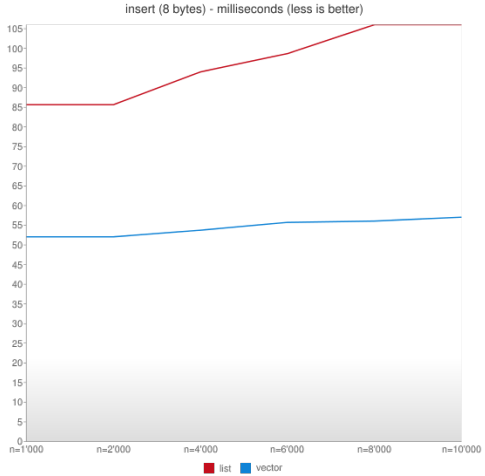
Quelle:

https://dzone.com/articles/c-benchmark-%E2%80%93-stdvector-vs

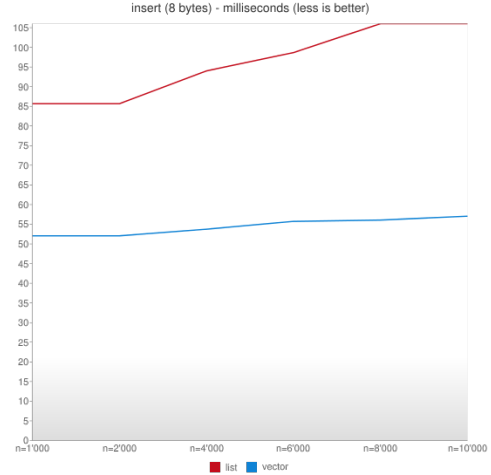
Array vs Linked List: Random Linear Find

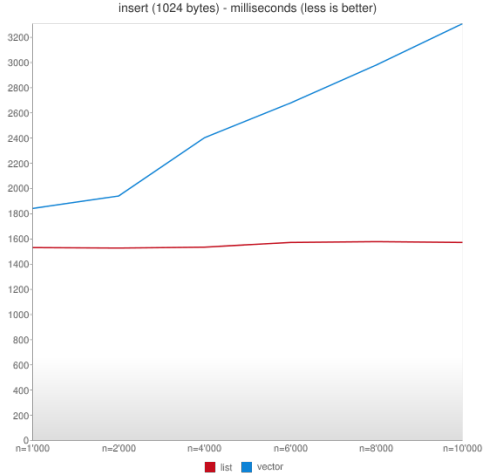


Array vs Linked List: Random Insert!!

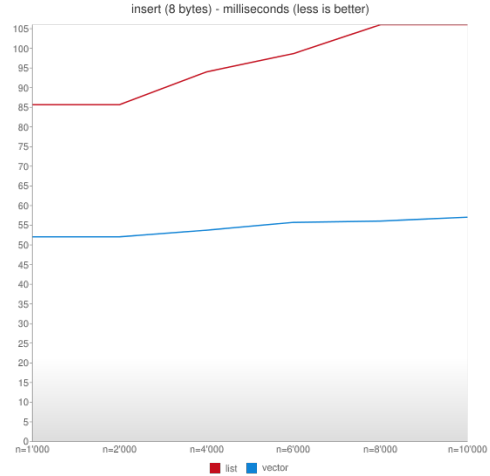


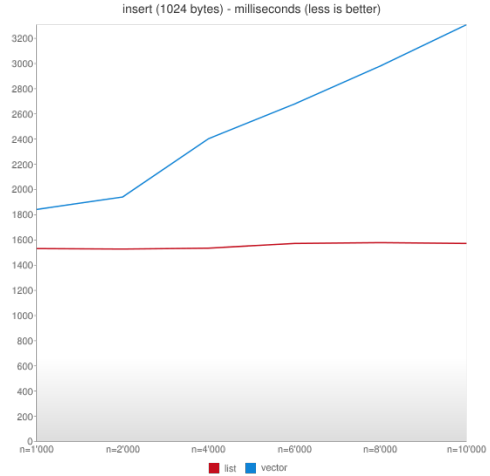
Array vs Linked List: Random Insert!!





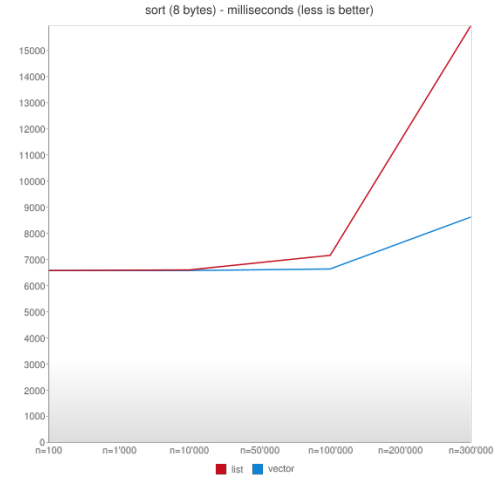
Array vs Linked List: Random Insert!!



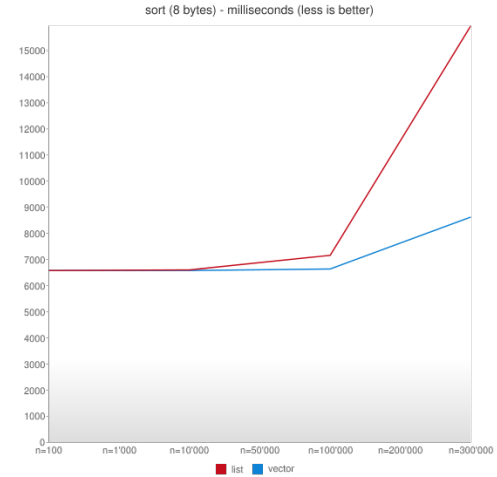


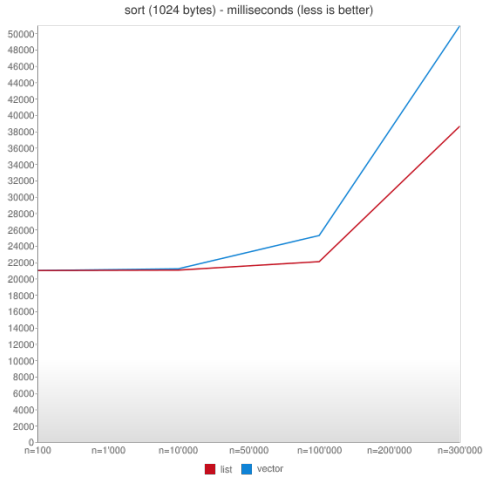
Note: große Structs in Arrays sind selten gut (s. später!)

Array vs Linked List: Sort

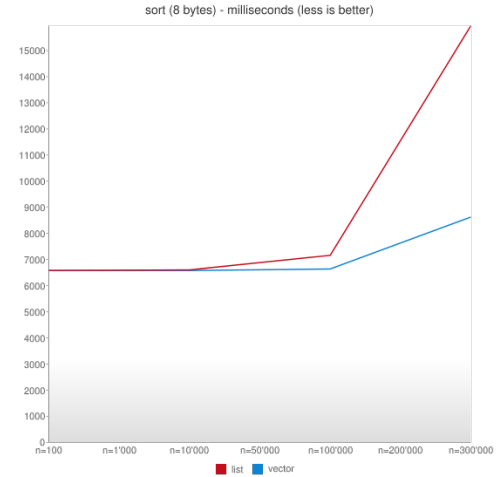


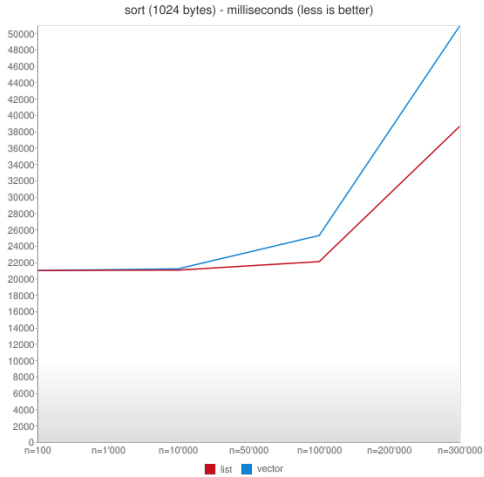
Array vs Linked List: Sort





Array vs Linked List: Sort





Note: bei LL werden keine Elemente kopiert, sondern nur pointer swapped.

Array vs Linked List: Begründung

- Pointer chasing
  - Memory Locality
  - Schlechtes Pipelining
  - Fetching Dependency
- Vectorization (später)

Binary Trees

bla bla genau gleiche idee keine zeit lol

- benchmark left as an exercise to the reader
- denkt mal darüber nach warum B-Bäume so toll sind
  - https://www.cs.cornell.edu/courses/cs312/2004sp/lectures/lec24.html
- Bei Arrays: Selbst dort gibt es locality improvements
- Perf improvement: Sort by hotness frequency (Eytzinger)
  - https://algorithmica.org/en/eytzinger
  - https://en.algorithmica.org/hpc/data-structures/binary-search/
  - http://arxiv.org/pdf/1509.05053

Asmyptotic Complexity ist zu grob...

Asmyptotic Complexity ist zu grob...

Also einfach Instruktionen Zählen...

Asmyptotic Complexity ist zu grob...

Also einfach Instruktionen Zählen...

Lüge: Man kann annehmen was der Compiler macht!

Array of Structs vs Struct of Arrays

$N = 100.000.000$. Compiled with `gcc -O3 -march=skylake` (native)

Array of Struct

```
1  typedef struct {
2    int32_t x;
3    int32_t y;
4    int32_t z;
5  } Position;
6  void update_AoS(Position *aos,
7        size_t count) {
8    for (size_t i=0; i<count; i++)
9      aos[i].x -= 5;
10 }
```

Array of Structs vs Struct of Arrays

$N = 100.000.000$. Compiled with `gcc -O3 -march=skylake` (native)

Array of Struct

```
1  typedef struct {
2    int32_t x;
3    int32_t y;
4    int32_t z;
5  } Position;
6  void update_AoS(Position *aos,
7        size_t count) {
8    for (size_t i=0; i<count; i++)
9      aos[i].x -= 5;
10 }
```

Struct of Arrays

```
1  typedef struct {
2    int32_t *x;
3    int32_t *y;
4    int32_t *z;
5  } PositionsSoA;
6  void update_SoA(PositionsSoA *soa,
7        size_t count) {
8    for (size_t i=0; i<count; i++)
9      soa->x[i] -= 5;
10 }
```

Array of Structs vs Struct of Arrays

$N = 100.000.000$. Compiled with `gcc -O3 -march=skylake` (native)

Array of Struct

```
1  typedef struct {
2    int32_t x;
3    int32_t y;
4    int32_t z;
5  } Position;
6  void update_AoS(Position *aos,
7        size_t count) {
8    for (size_t i=0; i<count; i++)
9      aos[i].x -= 5;
10 }
```

Struct of Arrays

```
1  typedef struct {
2    int32_t *x;
3    int32_t *y;
4    int32_t *z;
5  } PositionsSoA;
6  void update_SoA(PositionsSoA *soa,
7        size_t count) {
8    for (size_t i=0; i<count; i++)
9      soa->x[i] -= 5;
10 }
```

Took: 0.171748s

Array of Structs vs Struct of Arrays

$N = 100.000.000$. Compiled with `gcc -O3 -march=skylake` (native)

Array of Struct

```c
typedef struct {
  int32_t x;
  int32_t y;
  int32_t z;
} Position;
void update_AoS(Position *aos,
      size_t count) {
  for (size_t i=0; i<count; i++)
    aos[i].x -= 5;
}
```

Took: 0.171748s

Struct of Arrays

```c
typedef struct {
  int32_t *x;
  int32_t *y;
  int32_t *z;
} PositionsSoA;
void update_SoA(PositionsSoA *soa,
      size_t count) {
  for (size_t i=0; i<count; i++)
    soa->x[i] -= 5;
}
```

Took: 0.055744s

AoS vs SoA: Begründung

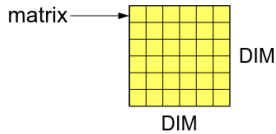
- Vektorisierung! Lange Register
- Kann bis 512 bit (64 byte) auf einmal processen
- Und natürlich cache locality wieder
- Keyword: Data Oriented Design (DOD)
  - Coolste Quelle: Entity Component Systems
  - Anti $\text{OOP}_{\text{gang gang}}$

Als nächstes: Weniger Arbeit ist weniger

A Scalability Story

Herb Sutter's scalability issue in counting odd matrix elements.

- Square matrix of side DIM with memory in array matrix.
- Sequential pseudocode:

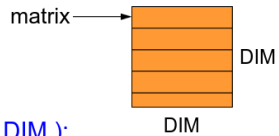



```
int odds = 0;
for( int i = 0; i < DIM; ++i )
   for( int j = 0; j < DIM; ++j )
      if( matrix[i*DIM + j] % 2 != 0 )
         ++odds;
```

A Scalability Story

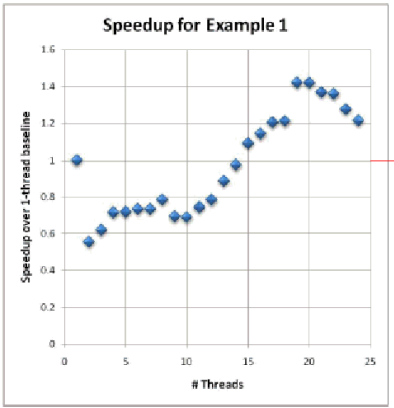
- Parallel pseudocode, take 1:

```
int result[P];

// Each of P parallel workers processes 1/P-th of the data;
// the p-th worker records its partial count in result[p]
for (int p = 0; p < P; ++p )
  pool.run( [&,p] {
    result[p] = 0;
    int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++result[p]; } );

pool.join();                          // Wait for all tasks to complete

odds = 0;                             // combine the results
for( int p = 0; p < P; ++p )
  odds += result[p];
```



A Scalability Story

Scalability unimpressive:



Faster than 1 core

Slower than 1 core

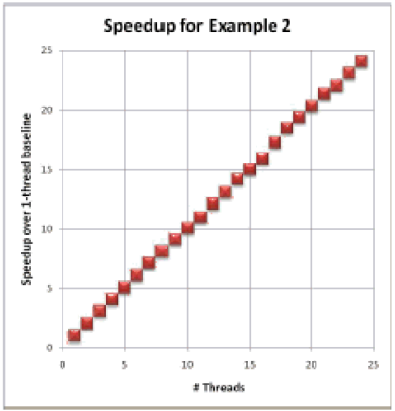
A Scalability Story

- Parallel pseudocode, take 2:

```
int result[P];

for (int p = 0; p < P; ++p )
  pool.run( [&,p] {
    int count = 0;                          // instead of result[p]
    int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++count;                          // instead of result[p]
    result[p] = count; } );                 // new statement

...                                         // nothing else changes
```

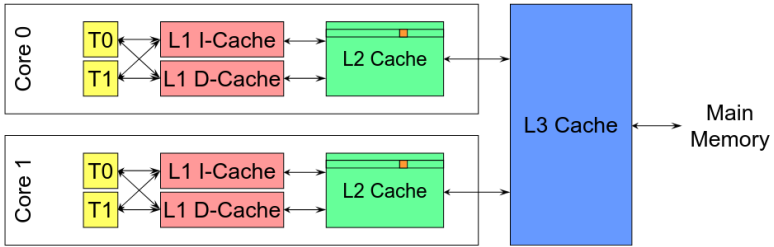
A Scalability Story

Scalability now perfect!



Cache Coherency

From core i7's architecture:



Assume both cores have cached the value at (virtual) address *A*.

- Whether in L1 or L2 makes no difference.

Consider:

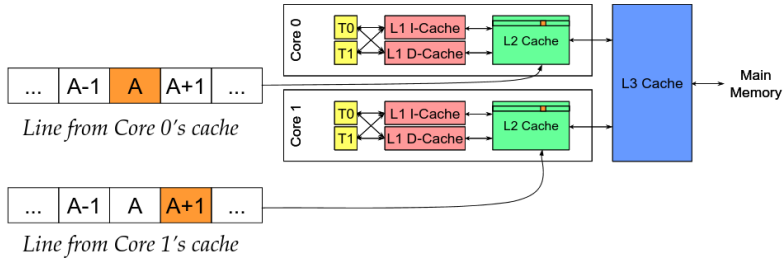
- Core 0 writes to *A*.
- Core 1 reads *A*.

What value does Core 1 read?

False Sharing

Suppose Core 0 accesses *A* and Core 1 accesses *A+1*.

- *Independent* pieces of memory; concurrent access is safe.
- But *A* and *A+1* probably map to the same cache line.
  - ➡ If so, Core 0's writes to *A* invalidates *A+1*'s cache line in Core 1.
    - And vice versa.
    - This is *false sharing*.



False Sharing

It explains Herb Sutter's issue:

```
int result[P];                          // many elements on 1 cache line

for (int p = 0; p < P; ++p )
  pool.run( [&,p] {                     // run P threads concurrently
    result[p] = 0;
    int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++result[p]; } );             // each repeatedly accesses the
                                        // same array (albeit different
                                        // elements)
```

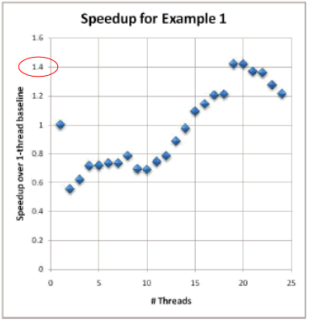
False Sharing

And his solution:

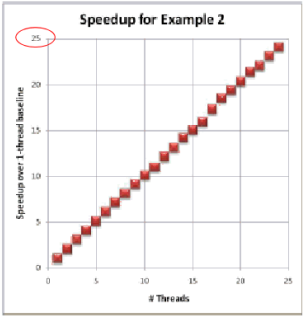
```
int result[P];                          // still multiple elements per
                                        // cache line

for (int p = 0; p < P; ++p )
  pool.run( [&,p] {
    int count = 0;                      // use local var for counting
    int chunkSize = DIM/P + 1;
    int myStart = p * chunkSize;
    int myEnd = min( myStart+chunkSize, DIM );
    for( int i = myStart; i < myEnd; ++i )
      for( int j = 0; j < DIM; ++j )
        if( matrix[i*DIM + j] % 2 != 0 )
          ++count;                      // update local var
    result[p] = count; } );             // access shared cache line
                                        // only once
```

False Sharing

His scalability results are worth repeating:



With False Sharing



Without False Sharing

Part 2? Lügen welche aus Zeitgründen ausgelassen wurden

Part 2? Lügen welche aus Zeitgründen ausgelassen wurden

- Busy Waiting ist immer schlecht

Part 2? Lügen welche aus Zeitgründen ausgelassen wurden

- Busy Waiting ist immer schlecht
- Zu gutem OOP zählt Inheritance

Part 2? Lügen welche aus Zeitgründen ausgelassen wurden

- Busy Waiting ist immer schlecht
- Zu gutem OOP zählt Inheritance
- TCP ist heutzutage sinnvoll (HTTP3, QUIC)

Part 2? Lügen welche aus Zeitgründen ausgelassen wurden

- Busy Waiting ist immer schlecht
- Zu gutem OOP zählt Inheritance
- TCP ist heutzutage sinnvoll (HTTP3, QUIC)
- Ne Menge “real world assumptions”:
  - Alle Sprachen haben Konzepte von Buchstaben
  - Zeitzonen sind komplette Stunden
    - Zeitzonen sind halbe Stunden
  - Leute haben Vor- und Nachnamen (s/o Mauladi)
  - Straßen haben Namen
  - Leute haben eine Adresse
  - ...

Conclusions

- Memory wird nicht unbedingt angelegt wenn du fragst

Conclusions

- Memory wird nicht unbedingt angelegt wenn du fragst
- Variablen werden nicht unbedingt komplett im Memory gespeichert

Conclusions

- Memory wird nicht unbedingt angelegt wenn du fragst
- Variablen werden nicht unbedingt komplett im Memory gespeichert
- Structs sind nicht so groß wie die Summe ihrer Elemente

Conclusions

- Memory wird nicht unbedingt angelegt wenn du fragst
- Variablen werden nicht unbedingt komplett im Memory gespeichert
- Structs sind nicht so groß wie die Summe ihrer Elemente
- Variablen werden im Optimalfall nicht aus dem RAM gelesen

Conclusions

- Memory wird nicht unbedingt angelegt wenn du fragst
- Variablen werden nicht unbedingt komplett im Memory gespeichert
- Structs sind nicht so groß wie die Summe ihrer Elemente
- Variablen werden im Optimalfall nicht aus dem RAM gelesen
- Code wird nicht Zeile für Zeile ausgeführt

Conclusions

- Memory wird nicht unbedingt angelegt wenn du fragst
- Variablen werden nicht unbedingt komplett im Memory gespeichert
- Structs sind nicht so groß wie die Summe ihrer Elemente
- Variablen werden im Optimalfall nicht aus dem RAM gelesen
- Code wird nicht Zeile für Zeile ausgeführt
- Big $\mathcal{O}$ reicht nicht für Performanceanalyse

Conclusions

- Memory wird nicht unbedingt angelegt wenn du fragst
- Variablen werden nicht unbedingt komplett im Memory gespeichert
- Structs sind nicht so groß wie die Summe ihrer Elemente
- Variablen werden im Optimalfall nicht aus dem RAM gelesen
- Code wird nicht Zeile für Zeile ausgeführt
- Big $\mathcal{O}$ reicht nicht für Performanceanalyse
- Linked Lists und Binary Trees sind selten performant

Conclusions

- Memory wird nicht unbedingt angelegt wenn du fragst
- Variablen werden nicht unbedingt komplett im Memory gespeichert
- Structs sind nicht so groß wie die Summe ihrer Elemente
- Variablen werden im Optimalfall nicht aus dem RAM gelesen
- Code wird nicht Zeile für Zeile ausgeführt
- Big $\mathcal{O}$ reicht nicht für Performanceanalyse
- Linked Lists und Binary Trees sind selten performant
- Operationen zählen reicht nicht für Performanceanalyse