

System Software for Energy-efficient Computing

Hands-On: Energy Efficient Sorting with QuickSort

Timo Hönig

Operating Systems and System Software

Ruhr-Universität Bochum (RUB)

25. September 2026

Agenda

- 1 Motivation
- 2 Funktionsprinzip und Implementierung
- 3 Nicht-funktionale Eigenschaften
- 4 Vergleich, Bewertung und praktische Anwendung

Motivation

Warum ist Sortierung wichtig?

- Beschleunigung von Datenzugriffen
↔ z.B. Latenzreduktion
- Verwaltung großer Datenmengen
↔ z.B. Datenspeicherpartitionierung



Welche Art von Sortierung?



Motivation

Warum ist Sortierung wichtig?

- Beschleunigung von Datenzugriffen
↪ z.B. Latenzreduktion
- Verwaltung großer Datenmengen
↪ z.B. Datenspeicherpartitionierung



Welche Art von Sortierung?

- Sortierung *nach* Typ
↪ z.B. heterogene Daten
- Sortierung *innerhalb eines* Typs
↪ z.B. Rangordnung und Hierarchie



Motivation

Relevanz in der Praxis

- Sortierung von Schlüsselwertspeicher (engl. key-value store) → DynamoDB
- Teile-und-Herrsche-Algorithmen (eng. divide and conquer) → Hadoop



Sortieren mit Quicksort

Motivation

Relevanz in der Praxis

- Sortierung von Schlüsselwertspeicher (engl. key-value store) → DynamoDB
- Teile-und-Herrsche-Algorithmen (eng. divide and conquer) → Hadoop



Sortieren mit Quicksort

- Sortieralgorithmus, dessen Funktionsprinzip auf Teile und Herrsche beruht
- Entwurf: Charles Antony Richard Hoare



CAR Hoare: Quicksort.

The Computer Journal, 5(1), 1962, pp. 10-16.



Agenda

- 1 Motivation ✓
- 2 Funktionsprinzip und Implementierung
- 3 Nicht-funktionale Eigenschaften
- 4 Vergleich, Bewertung und praktische Anwendung

Quicksort: Funktionsprinzip und Implementierung

```
1 def function quicksort(array a):
2   if (length(a)) <= 1
3     return a
4
5   int bound = a[length(a)-1]
6   array a_less, a_greater = []
7
8   for each x in a:
9     if x <= bound then
10      append x to a_less
11    else
12      append x to a_greater
13
14   return concatenate (quicksort(a_less), array(bound), quicksort(a_greater))
```

- Funktionsprinzip und Pseudocode-Implementierung
- rekursive ex-situ Variante

Quicksort: Funktionsprinzip und Implementierung

```
1 def function quicksort(array a):
2     if (length(a)) <= 1
3         return a
4
5     int bound = a[length(a)-1]
6     array a_less, a_greater = []
7
8     for each x in a:
9         if x <= bound then
10             append x to a_less
11         else
12             append x to a_greater
13
14     return concatenate (quicksort(a_less), array(bound), quicksort(a_greater))
```

- Abbruchbedingung der Rekursion
- Arrays der Länge ≤ 1 gelten als sortiert

Quicksort: Funktionsprinzip und Implementierung

```
1 def function quicksort(array a):
2     if (length(a)) <= 1
3         return a
4
5     int bound = a[length(a)-1]
6     array a_less, a_greater = []
7
8     for each x in a:
9         if x <= bound then
10             append x to a_less
11         else
12             append x to a_greater
13
14     return concatenate (quicksort(a_less), array(bound), quicksort(a_greater))
```

- Bestimme Pivotelement für Arrays der Länge > 1 : bound
- Neue Listen für kleinere und größere Elemente: $a_{\{less, greater\}}$

Quicksort: Funktionsprinzip und Implementierung

```
1 def function quicksort(array a):
2     if (length(a)) <= 1
3         return a
4
5     int bound = a[length(a)-1]
6     array a_less, a_greater = []
7
8     for each x in a:
9         if x <= bound then
10             append x to a_less
11         else
12             append x to a_greater
13
14     return concatenate (quicksort(a_less), array(bound), quicksort(a_greater))
```

- Iteriere: Speichere Elemente mit Wert $\leq$ bound in a_less
- Speichere Elemente mit Wert $>$ bound in a_greater

Quicksort: Funktionsprinzip und Implementierung

```
1 def function quicksort(array a):
2     if (length(a)) <= 1
3         return a
4
5     int bound = a[length(a)-1]
6     array a_less, a_greater = []
7
8     for each x in a:
9         if x <= bound then
10             append x to a_less
11         else
12             append x to a_greater
13
14     return concatenate (quicksort(a_less), array(bound), quicksort(a_greater))
```

- Rekursiv: Sortiere analog die Listen a_less und a_greater
- Abbau der Rekursion liefert gesamtsortiertes Array

Quicksort: Funktionsprinzip und Implementierung

```
1 def function quicksort(array a):  
2     if (length(a)) <= 1  
3         return a  
4  
5     int bound = a[length(a)-1]  
6     array a_less, a_greater = []  
7  
8     for each x in a:  
9         if x <= bound then  
10             append x to a_less  
11         else  
12             append x to a_greater  
13  
14     return concatenate (quicksort(a_less), array(bound), quicksort(a_greater))
```

Probleme und Verbesserungsmöglichkeit

- 1 Pivotisierung
→ Dreh- und Angelpunkt
- 2 dynamische
Speicherooperationen
- 3 entbehrliche Operationen

Quicksort: Funktionsprinzip und Implementierung

```
1 def function quicksort(array a):  
2     if (length(a)) <= 1  
3         return a  
4  
5     int bound = a[length(a)-1] ←  
6     array a_less, a_greater = []  
7  
8     for each x in a:  
9         if x <= bound then  
10             append x to a_less  
11         else  
12             append x to a_greater  
13  
14     return concatenate (quicksort(a_less), array(bound), quicksort(a_greater))
```

Probleme und Verbesserungsmöglichkeit

- 1 Pivotisierung
→ Dreh- und Angelpunkt
- 2 dynamische
Speicherooperationen
- 3 entbehrliche Operationen

- Wahl des Pivotelements (Pivotisierung)
- Verbesserung: Verwenden des arithmetischen Mittels...
 - ... aller Arraywerte
 - ... der Arraywerte an der ersten, mittleren und letzten Position

Quicksort: Funktionsprinzip und Implementierung

```
1 def function quicksort(array a):  
2   if (length(a)) <= 1  
3     return a  
4  
5   int bound = a[length(a)-1]  
6   array a_less, a_greater = []  
7  
8   for each x in a:  
9     if x <= bound then  
10      append x to a_less  
11    else  
12      append x to a_greater  
13  
14  return concatenate (quicksort(a_less), array(bound), quicksort(a_greater))
```

Probleme und Verbesserungsmöglichkeit

- 1 Pivotisierung
→ Dreh- und Angelpunkt
- 2 dynamische
Speicherooperationen
- 3 entbehrliche Operationen

- nicht-deterministisches Zeitverhalten
- Verbesserung:
 - Analyse der Eingabedaten → statische Speicherallokation
 - Operationen auf Datenstrukturen der Eingabedaten

Quicksort: Funktionsprinzip und Implementierung

```
1 def function quicksort(array a):  
2     if (length(a)) <= 1  
3         return a  
4  
5     int bound = a[length(a)-1]  
6     array a_less, a_greater = []  
7  
8     for each x in a:  
9         if x <= bound then  
10             append x to a_less  
11         else  
12             append x to a_greater  
13  
14     return concatenate (quicksort(a_less), array(bound), quicksort(a_greater))
```



Probleme und Verbesserungsmöglichkeit

- 1 Pivotisierung
→ Dreh- und Angelpunkt
- 2 dynamische
Speicherooperationen
- 3 entbehrliche Operationen

- Speicherdruck wegen vieler Funktionsaufrufe der Rekursion
- Verbesserung:
 - alternative Algorithmen (z.B. Insertionsort) für kleine Arrays
 - iterative Implementierung → Vermeidung von Stapelüberläufen

Quicksort: Programmcode, verbesserte Variante

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4
5     bound = arr[high] # Pivotisierung
6     idx_l = low, idx_r = high - 1
7
8     while idx_l <= idx_r
9         # Überspringe Elemente im Array < Pivotelement
10        while idx_l <= idx_r and a[idx_l] < bound
11            idx_l += 1
12        # Überspringe Elemente im Array >= Pivotelement
13        while idx_l <= idx_r and a[idx_r] >= bound
14            idx_r -= 1
15        # Tausche Elemente im Array
16        if idx_l < idx_r
17            swap(a, idx_l, idx_r)
18
19    # Platziere Pivotelement
20    swap(a, idx_l, high)
21
22    quicksort(a, low, idx_l - 1) # Rekursion: linker Teilbereich
23    quicksort(a, idx_l + 1, high) # Rekursion: rechter Teilbereich
```

Funktionsweise und Verbesserungen

- Laufindex idx_l : $\rightarrow$
- Laufindex idx_r : $\leftarrow$
- Tausche falls Element $a[idx_l] \geq \text{Pivot}$ und $a[idx_r] < \text{Pivot}$

Quicksort: Programmcode, verbesserte Variante

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4
5     bound = arr[high] # Pivotisierung
6     idx_l = low, idx_r = high - 1
7
8     while idx_l <= idx_r
9         # Überspringe Elemente im Array < Pivotelement
10        while idx_l <= idx_r and a[idx_l] < bound
11            idx_l += 1
12        # Überspringe Elemente im Array >= Pivotelement
13        while idx_l <= idx_r and a[idx_r] >= bound
14            idx_r -= 1
15        # Tausche Elemente im Array
16        if idx_l < idx_r
17            swap(a, idx_l, idx_r)
18
19        # Platziere Pivotelement
20        swap(a, idx_l, high)
21
22        quicksort(a, low, idx_l - 1) # Rekursion: linker Teilbereich
23        quicksort(a, idx_l + 1, high) # Rekursion: rechter Teilbereich
```

Funktionsweise und Verbesserungen

- Speicheroperationen auf Eingabedaten
- Vermeidung dynamischer Speicherallokation

Quicksort: Programmcode, verbesserte Variante

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4
5     bound = arr[high] # Pivotisierung
6     idx_l = low, idx_r = high - 1
7
8     while idx_l <= idx_r
9         # Überspringe Elemente im Array < Pivotelement
10        while idx_l <= idx_r and a[idx_l] < bound
11            idx_l += 1
12        # Überspringe Elemente im Array >= Pivotelement
13        while idx_l <= idx_r and a[idx_r] >= bound
14            idx_r -= 1
15        # Tausche Elemente im Array
16        if idx_l < idx_r
17            swap(a, idx_l, idx_r)
18
19    # Platziere Pivotelement
20    swap(a, idx_l, high)
21
22    quicksort(a, low, idx_l - 1) # Rekursion: linker Teilbereich
23    quicksort(a, idx_l + 1, high) # Rekursion: rechter Teilbereich
```

Funktionsweise und Verbesserungen

- Speicheroperationen auf Eingabedaten
- Rekursion arbeitet ebenfalls auf Array der Eingabedaten

Quicksort: Exemplarisches Beispiel

Index	0	1	2	3	4	5
Array a	3	15	512	47	11	42
	↑					↑
	low					high

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4     bound = arr[high]
5     idx_l = low, idx_r = high - 1
6     while idx_l <= idx_r
7         while idx_l <= idx_r and a[idx_l] < bound
8             idx_l += 1
9         while idx_l <= idx_r and a[idx_r] >= bound
10            idx_r -= 1
11        if idx_l < idx_r
12            swap(a, idx_l, idx_r)
13    swap(a, idx_l, high)
14    quicksort(a, low, idx_l - 1)
15    quicksort(a, idx_l + 1, high)
```

Quicksort: Exemplarisches Beispiel

Index	0	1	2	3	4	5
Array a	3	15	512	47	11	42
	↑				↑	↔
	idx_l=0				idx_r=4	Pivot: 42

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4     bound = arr[high]
5     idx_l = low, idx_r = high - 1
6     while idx_l <= idx_r
7         while idx_l <= idx_r and a[idx_l] < bound
8             idx_l += 1
9         while idx_l <= idx_r and a[idx_r] >= bound
10            idx_r -= 1
11        if idx_l < idx_r
12            swap(a, idx_l, idx_r)
13    swap(a, idx_l, high)
14    quicksort(a, low, idx_l - 1)
15    quicksort(a, idx_l + 1, high)
```

1. Abbruchbedingung

2. Intialisierung

Quicksort: Exemplarisches Beispiel

Index	0	1	2	3	4	5
Array a	3	15	512	47	11	42
	↑ idx_l=0		↑ idx_l=2		↑ idx_r=4	↔ Pivot: 42

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4     bound = arr[high]
5     idx_l = low, idx_r = high - 1
6     while idx_l <= idx_r
7         while idx_l <= idx_r and a[idx_l] < bound
8             idx_l += 1
9         while idx_l <= idx_r and a[idx_r] >= bound
10            idx_r -= 1
11         if idx_l < idx_r
12             swap(a, idx_l, idx_r)
13     swap(a, idx_l, high)
14     quicksort(a, low, idx_l - 1)
15     quicksort(a, idx_l + 1, high)
```

1. Abbruchbedingung

2. Intialisierung

3a. Durchlaufe Array
3b. Vergleiche Elemente
mit Pivot
3c. Tausche Elemente

Quicksort: Exemplarisches Beispiel

Index	0	1	2	3	4	5
Array a	3	15	512	47	11	42
	↑ idx_l=0		↑ idx_l=2		↑ idx_r=4	↔ Pivot: 42

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4     bound = arr[high]
5     idx_l = low, idx_r = high - 1
6     while idx_l <= idx_r
7         while idx_l <= idx_r and a[idx_l] < bound
8             idx_l += 1
9         while idx_l <= idx_r and a[idx_r] >= bound
10            idx_r -= 1
11         if idx_l < idx_r
12             swap(a, idx_l, idx_r)
13     swap(a, idx_l, high)
14     quicksort(a, low, idx_l - 1)
15     quicksort(a, idx_l + 1, high)
```

1. Abbruchbedingung

2. Intialisierung

3a. Durchlaufe Array
3b. Vergleiche Elemente
mit Pivot
3c. Tausche Elemente

Quicksort: Exemplarisches Beispiel

Index	0	1	2	3	4	5
Array a	3	15	512	47	11	42
	↑ idx_l=0		↑ idx_l=2		↑ idx_r=4	↔ Pivot: 42

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4     bound = arr[high]
5     idx_l = low, idx_r = high - 1
6     while idx_l <= idx_r
7         while idx_l <= idx_r and a[idx_l] < bound
8             idx_l += 1
9         while idx_l <= idx_r and a[idx_r] >= bound
10            idx_r -= 1
11         if idx_l < idx_r
12             swap(a, idx_l, idx_r)
13     swap(a, idx_l, high)
14     quicksort(a, low, idx_l - 1)
15     quicksort(a, idx_l + 1, high)
```

1. Abbruchbedingung

2. Intialisierung

3a. Durchlaufe Array
3b. Vergleiche Elemente
mit Pivot
3c. Tausche Elemente

Quicksort: Exemplarisches Beispiel

Index	0	1	2	3	4	5
Array a	3	15	11	47	512	42
	↑ idx_l=0		↑ idx_l=2		↑ idx_r=4	↔ Pivot: 42

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4     bound = arr[high]
5     idx_l = low, idx_r = high - 1
6     while idx_l <= idx_r
7         while idx_l <= idx_r and a[idx_l] < bound
8             idx_l += 1
9         while idx_l <= idx_r and a[idx_r] >= bound
10            idx_r -= 1
11         if idx_l < idx_r
12             swap(a, idx_l, idx_r)
13     swap(a, idx_l, high)
14     quicksort(a, low, idx_l - 1)
15     quicksort(a, idx_l + 1, high)
```

1. Abbruchbedingung

2. Intialisierung

3a. Durchlaufe Array
3b. Vergleiche Elemente
mit Pivot
3c. Tausche Elemente

Quicksort: Exemplarisches Beispiel

Index	0	1	2	3	4	5
Array a	3	15	11	47	512	42

↑ ↑ ↔ Pivot: 42
idx_l=3 idx_r=4

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4     bound = arr[high]
5     idx_l = low, idx_r = high - 1
6     while idx_l <= idx_r
7         while idx_l <= idx_r and a[idx_l] < bound
8             idx_l += 1
9         while idx_l <= idx_r and a[idx_r] >= bound
10            idx_r -= 1
11         if idx_l < idx_r
12             swap(a, idx_l, idx_r)
13     swap(a, idx_l, high)
14     quicksort(a, low, idx_l - 1)
15     quicksort(a, idx_l + 1, high)
```

1. Abbruchbedingung

2. Intialisierung

3a. Durchlaufe Array
3b. Vergleiche Elemente
mit Pivot
3c. Tausche Elemente

Quicksort: Exemplarisches Beispiel

Index	0	1	2	3	4	5
Array a	3	15	11	47	512	42
			↑	↑		↔ Pivot: 42
			idx_r=2	idx_l=3		

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4     bound = arr[high]
5     idx_l = low, idx_r = high - 1
6     while idx_l <= idx_r
7         while idx_l <= idx_r and a[idx_l] < bound
8             idx_l += 1
9         while idx_l <= idx_r and a[idx_r] >= bound
10            idx_r -= 1
11         if idx_l < idx_r
12             swap(a, idx_l, idx_r)
13     swap(a, idx_l, high)
14     quicksort(a, low, idx_l - 1)
15     quicksort(a, idx_l + 1, high)
```

1. Abbruchbedingung

2. Intialisierung

3a. Durchlaufe Array
3b. Vergleiche Elemente
mit Pivot
3c. Tausche Elemente

Quicksort: Exemplarisches Beispiel

Index	0	1	2	3	4	5
Array a	3	15	11	47	512	42
			↑	↑		↔ Pivot: 42
			idx_r=2	idx_l=3		

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4     bound = arr[high]
5     idx_l = low, idx_r = high - 1
6     while idx_l <= idx_r
7         while idx_l <= idx_r and a[idx_l] < bound
8             idx_l += 1
9         while idx_l <= idx_r and a[idx_r] >= bound
10            idx_r -= 1
11         if idx_l < idx_r
12             swap(a, idx_l, idx_r)
13     swap(a, idx_l, high)
14     quicksort(a, low, idx_l - 1)
15     quicksort(a, idx_l + 1, high)
```

1. Abbruchbedingung

2. Intialisierung

3a. Durchlaufe Array
3b. Vergleiche Elemente
mit Pivot
3c. Tausche Elemente

Quicksort: Exemplarisches Beispiel

Index	0	1	2	3	4	5
Array a	3	15	11	47	512	42
	↑			↑		↑
	low			idx_l		high

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4     bound = arr[high]
5     idx_l = low, idx_r = high - 1
6     while idx_l <= idx_r
7         while idx_l <= idx_r and a[idx_l] < bound
8             idx_l += 1
9         while idx_l <= idx_r and a[idx_r] >= bound
10            idx_r -= 1
11         if idx_l < idx_r
12             swap(a, idx_l, idx_r)
13     swap(a, idx_l, high)
14     quicksort(a, low, idx_l - 1)
15     quicksort(a, idx_l + 1, high)
```

1. Abbruchbedingung

2. Intialisierung

3a. Durchlaufe Array
3b. Vergleiche Elemente
mit Pivot
3c. Tausche Elemente

4. Positioniere Pivot

Quicksort: Exemplarisches Beispiel

Index	0	1	2	3	4	5
Array a	3	15	11	42	512	47
	↑			↑		↑
	low			idx_l		high

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4     bound = arr[high]
5     idx_l = low, idx_r = high - 1
6     while idx_l <= idx_r
7         while idx_l <= idx_r and a[idx_l] < bound
8             idx_l += 1
9         while idx_l <= idx_r and a[idx_r] >= bound
10            idx_r -= 1
11         if idx_l < idx_r
12             swap(a, idx_l, idx_r)
13     swap(a, idx_l, high)
14     quicksort(a, low, idx_l - 1)
15     quicksort(a, idx_l + 1, high)
```

1. Abbruchbedingung

2. Intialisierung

3a. Durchlaufe Array
3b. Vergleiche Elemente
mit Pivot
3c. Tausche Elemente

4. Positioniere Pivot

Quicksort: Exemplarisches Beispiel

Index	0	1	2	3	4	5
Array a	3	15	11	42	512	47
	↑		↑	↑	↑	↑
	low		idx_l - 1	idx_l	idx_l + 1	high

```
1 def function quicksort(array a, int low, int high):
2     if low >= high
3         return
4     bound = arr[high]
5     idx_l = low, idx_r = high - 1
6     while idx_l <= idx_r
7         while idx_l <= idx_r and a[idx_l] < bound
8             idx_l += 1
9         while idx_l <= idx_r and a[idx_r] >= bound
10            idx_r -= 1
11         if idx_l < idx_r
12             swap(a, idx_l, idx_r)
13     swap(a, idx_l, high)
14     quicksort(a, low, idx_l - 1)
15     quicksort(a, idx_l + 1, high)
```

1. Abbruchbedingung

2. Intialisierung

3a. Durchlaufe Array
3b. Vergleiche Elemente
mit Pivot
3c. Tausche Elemente

4. Positioniere Pivot

5. Rekursion

Quicksort: Exemplarisches Beispiel, Rekursion

Index	0	1	2	3	4	5	
Array a	3	15	11	42	512	47	✓ ₄₂

Index	0	1	2	3	4	5	
Array a	3	15	11	42	512	47	

Index
Array a

Index
Array a

Index
Array a

Index
Array a

Quicksort: Exemplarisches Beispiel, Rekursion

Index	0	1	2	3	4	5	
Array a	3	15	11	42	512	47	✓ ₄₂

Index	0	1	2	3	4	5	
Array a	3	11	15	42	512	47	

Index
Array a

Index
Array a

Index
Array a

Index
Array a

Quicksort: Exemplarisches Beispiel, Rekursion

Index	0	1	2	3	4	5	
Array a	3	15	11	42	512	47	✓ ₄₂

Index	0	1	2	3	4	5	
Array a	3	11	15	42	512	47	

Index	0	1	2	3	4	5	
Array a	3	11	15	42	512	47	✓ ₁₁

Index
Array a

Index
Array a

Index
Array a

Quicksort: Exemplarisches Beispiel, Rekursion

Index	0	1	2	3	4	5	
Array a	3	15	11	42	512	47	✓ ₄₂

Index	0	1	2	3	4	5	
Array a	3	11	15	42	512	47	

Index	0	1	2	3	4	5	
Array a	3	11	15	42	512	47	✓ ₁₁

Index	0	1	2	3	4	5	
Array a	3	11	15	42	512	47	✓ _{3, 15}

Index
Array a

Index
Array a

Quicksort: Exemplarisches Beispiel, Rekursion

Index	0	1	2	3	4	5	
Array a	3	15	11	42	512	47	✓ ₄₂

Index	0	1	2	3	4	5	
Array a	3	11	15	42	512	47	

Index	0	1	2	3	4	5	
Array a	3	11	15	42	512	47	✓ ₁₁

Index	0	1	2	3	4	5	
Array a	3	11	15	42	47	512	✓ _{3, 15}

Index
Array a

Index
Array a

Quicksort: Exemplarisches Beispiel, Rekursion

Index	0	1	2	3	4	5	
Array a	3	15	11	42	512	47	✓ ₄₂
Index	0	1	2	3	4	5	
Array a	3	11	15	42	512	47	
Index	0	1	2	3	4	5	
Array a	3	11	15	42	512	47	✓ ₁₁
Index	0	1	2	3	4	5	
Array a	3	11	15	42	47	512	✓ _{3, 15}
Index	0	1	2	3	4	5	
Array a	3	11	15	42	47	512	✓ ₄₇
Index							
Array a							

Quicksort: Exemplarisches Beispiel, Rekursion

Index	0	1	2	3	4	5	
Array a	3	15	11	42	512	47	✓ ₄₂
Index	0	1	2	3	4	5	
Array a	3	11	15	42	512	47	
Index	0	1	2	3	4	5	
Array a	3	11	15	42	512	47	✓ ₁₁
Index	0	1	2	3	4	5	
Array a	3	11	15	42	47	512	✓ _{3, 15}
Index	0	1	2	3	4	5	
Array a	3	11	15	42	47	512	✓ ₄₇
Index	0	1	2	3	4	5	
Array a	3	11	15	42	47	512	✓ ₅₁₂

Agenda

- 1 Motivation ✓
- 2 Funktionsprinzip und Implementierung ✓
- 3 Nicht-funktionale Eigenschaften
- 4 Vergleich, Bewertung und praktische Anwendung

Nicht-funktionale Eigenschaften von Quicksort

Laufzeit

- durchschnittlich $\mathcal{O}(n \log(n))$ Vergleiche
↪ Rekursionstiefe: $\log(n)$, je n Elemente
- im schlechtesten Fall $\mathcal{O}(n^2)$
↪ Laufzeit steigt quadratisch zur Eingabe



Speicherplatz



Nicht-funktionale Eigenschaften von Quicksort

Laufzeit

- durchschnittlich $\mathcal{O}(n \log(n))$ Vergleiche
↪ Rekursionstiefe: $\log(n)$, je n Elemente
- im schlechtesten Fall $\mathcal{O}(n^2)$
↪ Laufzeit steigt quadratisch zur Eingabe



Speicherplatz

- Speicherbedarf (Stapelspeicher) hängt von der jeweiligen Implementierung ab
↪ rekursiv: hoch
↪ iterativ: niedrig



Nicht-funktionale Eigenschaften von Quicksort

Energiebedarf

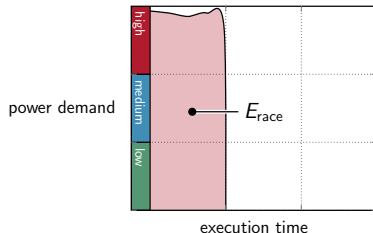
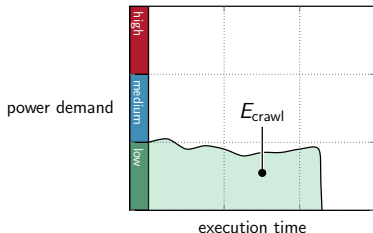
- externe Einflussfaktoren
 - Eingabegröße → Laufzeit
 - System → Speicher- vs. CPU-Zeit
- implementierungsspezifische Faktoren
 - hardwareabhängige Spezialbefehle
 - systemspezifische Energiesparmethoden



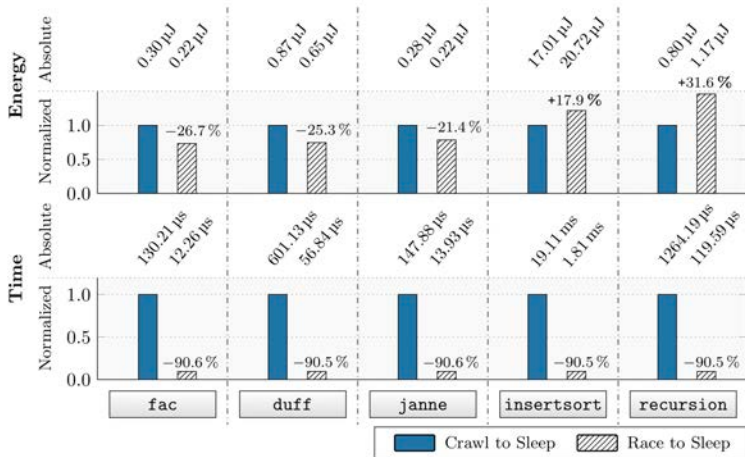
Nicht-funktionale Eigenschaften von Quicksort

Energiebedarf

- externe Einflussfaktoren
 - Eingabegröße → Laufzeit
 - System → Speicher- vs. CPU-Zeit
- implementierungsspezifische Faktoren
 - hardwareabhängige Spezialbefehle
 - systemspezifische Energiesparmethoden

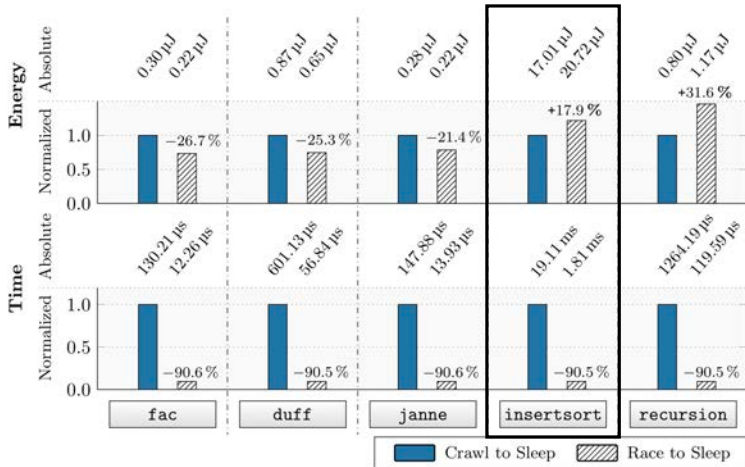


Nicht-funktionale Eigenschaften von Quicksort



System: ARM Cortex-M0+, Speicher: 4 KB RAM, CPU: max. 48 MHz

Nicht-funktionale Eigenschaften von Quicksort



System: ARM Cortex-M0+, Speicher: 4 KB RAM, CPU: max. 48 MHz

Agenda

- 1 Motivation ✓
- 2 Funktionsprinzip und Implementierung ✓
- 3 Nicht-funktionale Eigenschaften ✓
- 4 Vergleich, Bewertung und praktische Anwendung

Vergleich, Bewertung und praktische Anwendung

Quicksort: Vergleich mit anderen Sortieralgorithmen

Algorithmus	Best-Case	Average-Case	Worst-Case	Speicherbedarf
Bubblesort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Heapsort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(1)$
Insertionsort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Mergesort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n)$
Quicksort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)/\mathcal{O}(\log n)$
Selectionsort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$

Tabelle: Vergleich unterschiedlicher Sortieralgorithmen

Vergleich, Bewertung und praktische Anwendung

Vergleich und Bewertung

- Quicksort: Average-Case vs. Worst-Case
↪ auf effiziente Implementierung achten
- Erweiterungen, spezialisierte Algorithmen
↪ Parallelisierung, Spezialbefehle



Praktische Anwendung



Vergleich, Bewertung und praktische Anwendung

Vergleich und Bewertung

- Quicksort: Average-Case vs. Worst-Case
↪ auf effiziente Implementierung achten
- Erweiterungen, spezialisierte Algorithmen
↪ Parallelisierung, Spezialbefehle



Praktische Anwendung

- Bibliotheken und Laufzeitsysteme bieten Referenzimplementierungen
↪ Maßschneiderung, Anwendungsfall
↪ Linux: Heapsort, Worst-Case-Verhalten



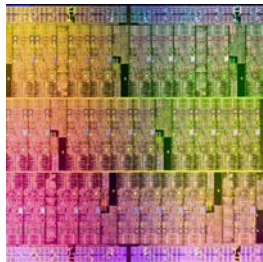
Zusammenfassung und Ausblick

Zusammenfassung

- Quicksort Sortieralgorithmus
- effizienter Algorithmus bei...
 - ↪ unsortierten Daten
 - ↪ guter Pivotisierung



Ausblick



Zusammenfassung und Ausblick

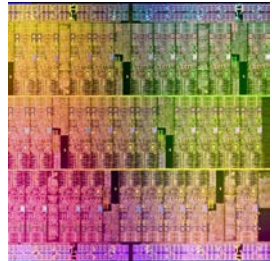
Zusammenfassung

- Quicksort Sortieralgorithmus
- effizienter Algorithmus bei...
 - ↔ unsortierten Daten
 - ↔ guter Pivotisierung



Ausblick

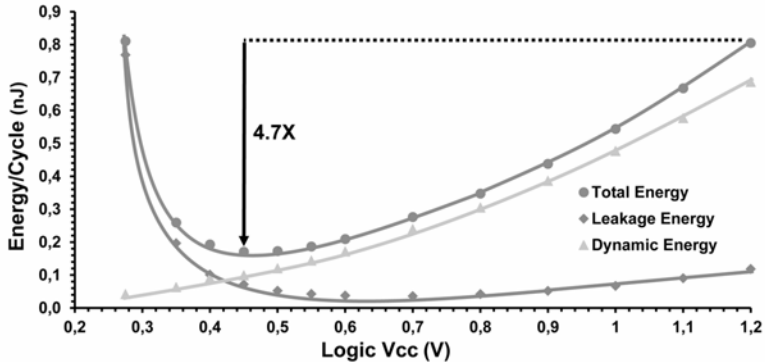
- Parallelisieren
 - ↔ mit mehreren Prozessoren sortieren
- alternative Verarbeitungseinheiten
 - ↔ GPU, FPGA



Diskussion

Vertiefungsfolien

Leckstrom in MOS-Schaltungen



- Shailendra Jain, Surhud Khare, Satish Yada et al.
A 280mV-to-1.2V Wide-Operating-Range IA-32 Processor in 32 nm CMOS
IEEE International Solid-State Circuits Conference (ISSCC), 2012.

Symbolische Ausführung mit Annotationen

```
1 #include <fibonacci.h>
2
3 /* @mera: #mode=strict
4 *      #func=foo
5 *      #para=i
6 */
7
8 uint32 bar (uint32 i)
9 {
10     if (i > 0)
11         return fibonacci(i);
12     return -1;
13 }
14
15 uint32 foo (uint32 i)
16 {
17     uint32 res;
18     /* @mera: #i=42 */
19     res = bar(i);
20     /* @arem */
21     return res;
22 }
23 /* @arem */
```

(a) Strict Mode

```
1 #include <fibonacci.h>
2
3 /* @mera: #mode=immersive
4 *      #func=*
5 *      #para=*
6 */
7
8 uint32 bar (uint32 i)
9 {
10     if (i > 0)
11         return fibonacci(i);
12     return -1;
13 }
14
15 uint32 foo (uint32 i)
16 {
17     uint32 res;
18
19     res = bar(i);
20
21     return res;
22 }
23 /* @arem */
```

(b) Immersive Mode

```
1 #include <fibonacci.h>
2
3 /* @mera: #mode=managed
4 *      #func=foo
5 *      #para=i
6 */
7
8 uint32 bar (uint32 i)
9 {
10     if (i > 0)
11         return fibonacci(i);
12     return -1;
13 }
14
15 uint32 foo (uint32 i)
16 {
17     uint32 res;
18     /* @mera: #i=[1,42] */
19     res = bar(i);
20     /* @arem */
21     return res;
22 }
23 /* @arem */
```

(c) Managed Mode

- Koushik Sen, Haruto Tanno et al.
GuideSE: Annotations for Guiding Concolic Testing
AST'15 / 37th International Conference on Software Engineering (ICSE 2015), 2012.

Basisblock, Übersetzeroptimierungen

```
1 main.entry:  
2   push r7, lr  
3   mov r7, sp  
4   sub sp, #16  
5   movs r0, #0  
6   str r0, [sp, #12]  
7   str r0, [sp, #4]  
8   movs r1, #5  
9   str r1, [sp, #0]  
10  str r0, [sp, #8]  
11  b.n label
```

(a) Level 00

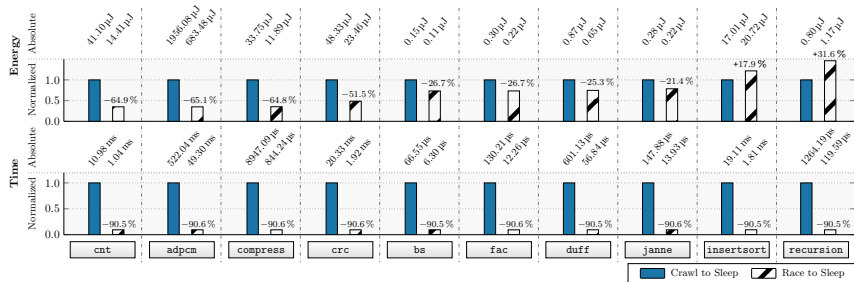
```
1 main.entry:  
2   sub sp, #4  
3   movs r0, #5  
4   str r0, [sp, #0]  
5   ldr r0, [sp, #0]  
6   cmp r0, #0  
7   ittt lt  
8   movlt r0, #0  
9   addlt sp, #4  
10  bxlt lr  
11
```

(b) Level 03

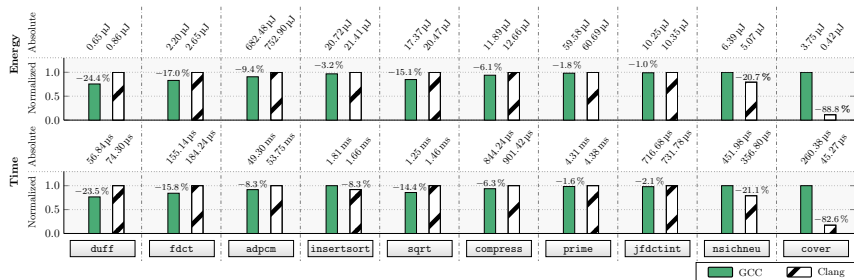
```
1 main.entry:  
2   sub sp, #4  
3   movs r0, #5  
4   str r0, [sp, #0]  
5   ldr r0, [sp, #0]  
6   blt.n label  
7  
8  
9  
10  
11
```

(c) Level 0s

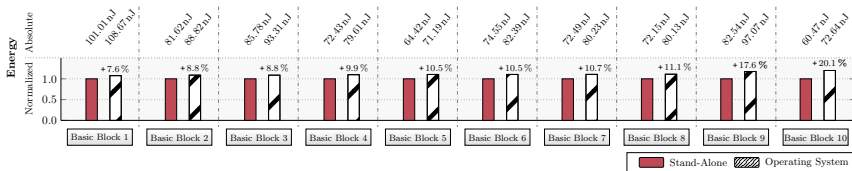
Laufzeit vs. Energiebedarf (DVFS)



Laufzeit vs. Energiebedarf (Übersetzer)

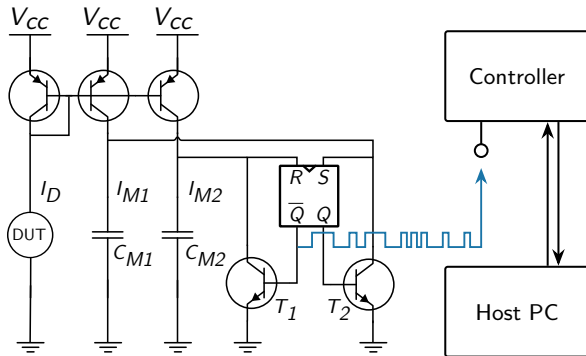


Betriebssystem: Hintergrundrauschen

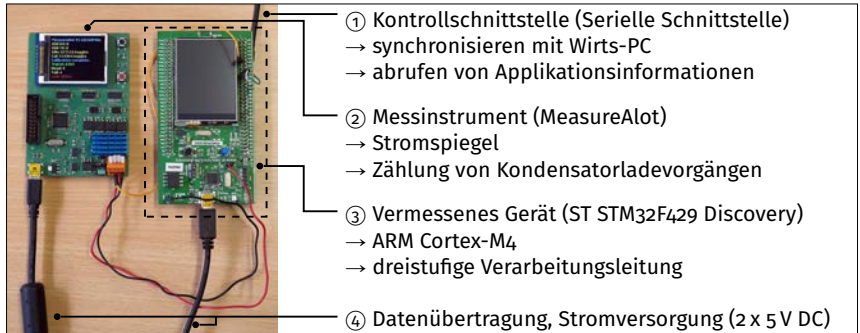


Stromspiegel-Messschaltung

Messschaltung: Funktionsschaltbild

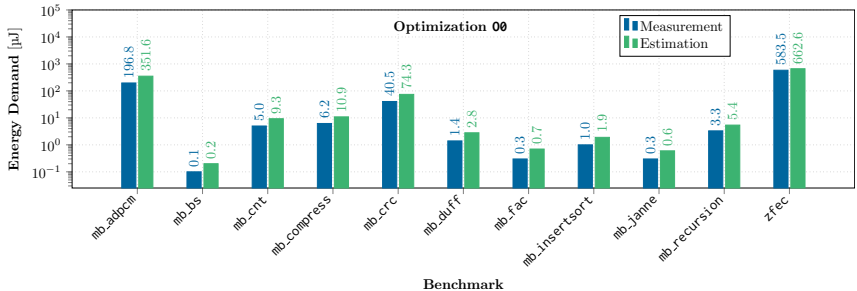


Messschaltung: MCU-Platine

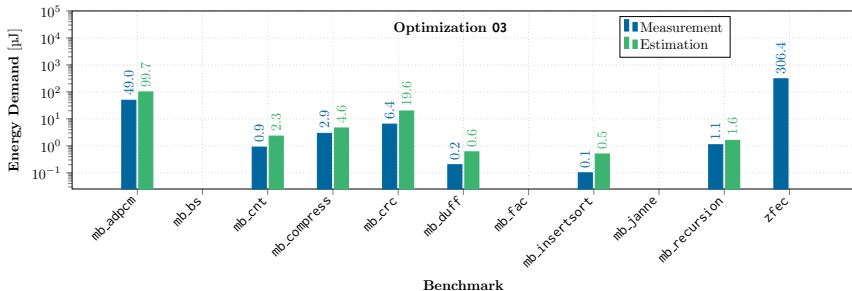


Artificial Neural Network

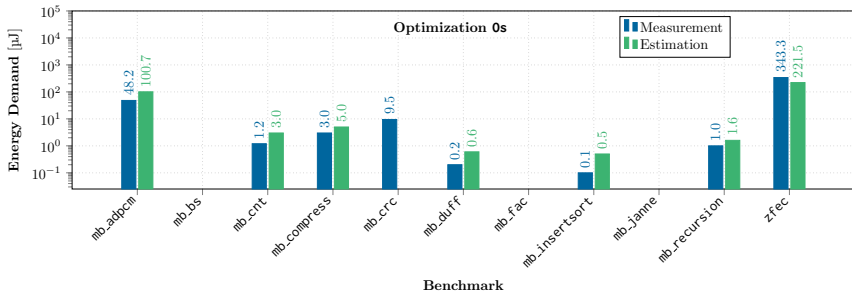
Energiebedarfsvorhersagen mit neuronalem Netzwerk (-00)



Energiebedarfsvorhersagen mit neuronalem Netzwerk (-03)



Energiebedarfsvorhersagen mit neuronalem Netzwerk (-0s)



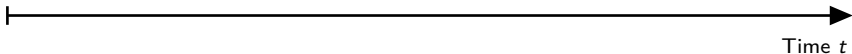
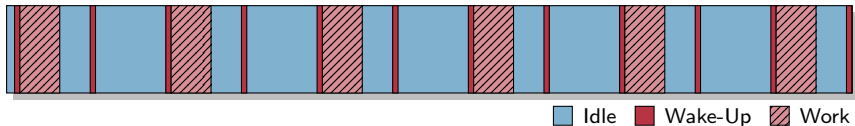
Energy-Aware Systems: Low-Energy Optimisation

- Statistics at process level (e.g., PowerTOP), unit of measurement is *wake-ups per second*
- Wake-ups cause the CPU to return from C-state, subsequent activities (e.g., I/O) are likely to follow
- Less wake-ups → lower energy consumption

Energy-Aware Systems: Low-Energy Optimisation

- Statistics at process level (e.g., PowerTOP), unit of measurement is *wake-ups per second*
- Wake-ups cause the CPU to return from C-state, subsequent activities (e.g., I/O) are likely to follow
- Less wake-ups → lower energy consumption

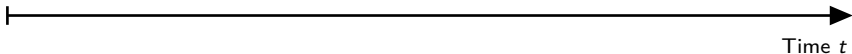
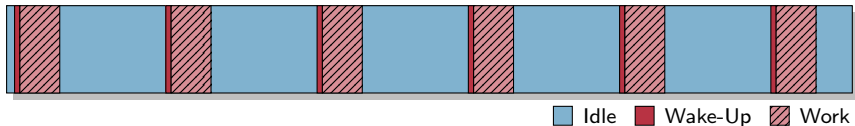
Process Activity



Energy-Aware Systems: Low-Energy Optimisation

- Statistics at process level (e.g., PowerTOP), unit of measurement is *wake-ups per second*
- Wake-ups cause the CPU to return from C-state, subsequent activities (e.g., I/O) are likely to follow
- Less wake-ups → lower energy consumption

Process Activity



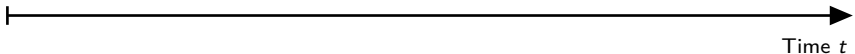
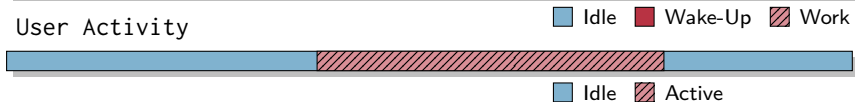
Energy-Aware Systems: Low-Energy Optimisation

- Statistics at process level (e.g., PowerTOP), unit of measurement is *wake-ups per second*
- Wake-ups cause the CPU to return from C-state, subsequent activities (e.g., I/O) are likely to follow
- Less wake-ups → lower energy consumption

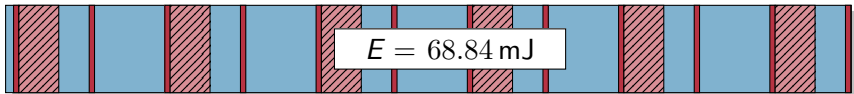
Process Activity



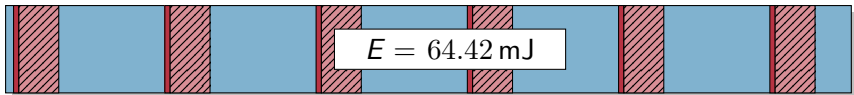
User Activity



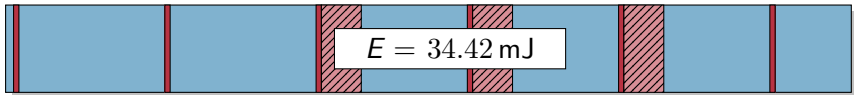
Process Activity 1



Process Activity 2

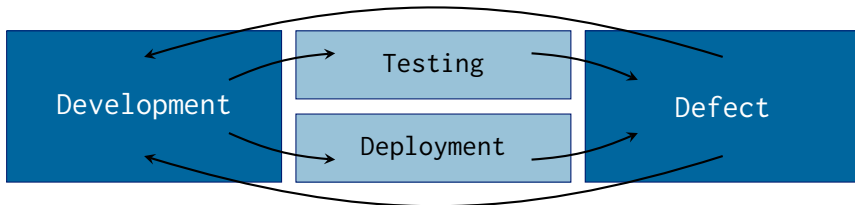
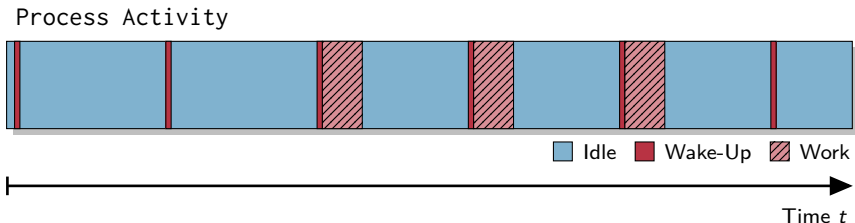


Process Activity 3



 Idle
  Wake-Up
  Work

Low-Energy Optimisation: Backward-Looking Approach



Low-Energy Optimisation: Forward-Looking Approach

