1

2

SIEMENS

Agenda

Teil I: Virtual Threads

1 Ziel
2 Konzept und Architektur
3 Anwendung
4 Verhalten
5 Performance und Skalierbarkeit

TEIL II: Sprachfeatures zur Unterstützung von Virtual Threads

6 Structured Concurrency
7 Scoped Values

Teil III: Architekturen auf Basis von Virtual Threads

8 Geeignete Architekturen
9 Ungeeignete Architekturen
10 Entscheidungskriterien für Migration

Page 3

3

SIEMENS

Agenda

Teil I: Virtual Threads

1 Ziel: Thread-per-Request Programmiermodell skalierbar machen
2 Konzept und Architektur: Implizite Freigabe des Carrier-Threads bei Blocking-I/O
3 Anwendung: API und Programmiermodell
4 Verhalten: Was passiert bei einem blockierenden Aufruf?
5 Performance und Skalierbarkeit: Thread-per-Request Server

TEIL II: Sprachfeatures zur Unterstützung von Virtual Threads

6 Structured Concurrency: Lösung der Struktur-Probleme von Threads
7 Scoped Values: Datenzuweisung an Threads

Teil III: Architekturen auf Basis von Virtual Threads

8 Geeignete Architekturen: Systeme, die von Virtual Threads profitieren
9 Ungeeignete Architekturen: Systeme, die nicht von Virtual Threads profitieren
10 Entscheidungskriterien für Migration zu Virtual Threads

Page 5

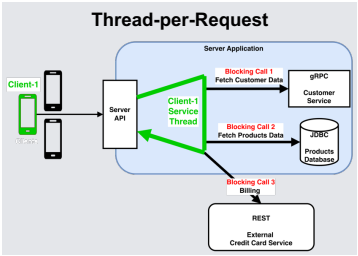
5

SIEMENS

Ziel von Virtual Threads: Thread-per-Request skalierbar

Skalierbarkeit: Oft limitiert durch Speicherbedarf von Threads, nicht durch CPU

- **Typische Server-Applikation: Blocking-Calls**
  - Datenbank, REST-Calls, Kreditkartenzahlung
  - Performancegewinn durch Nebenläufigkeit möglich
- **Thread-per-Request Programmiermodell**
  - Jeder Client sequentiell von einem Thread bedient
- **Java Platform-Threads skalieren nicht gut**
  - Speicherbedarf: Stack pro Thread
- **Virtual Threads**
  - Blockierender Aufruf: Platform-Thread Freigabe
  - Transparente Asynchronität
  - Keine Speicherplatz-Kosten bei Blocking



Extreme hohe Skalierbarkeit

Page 6

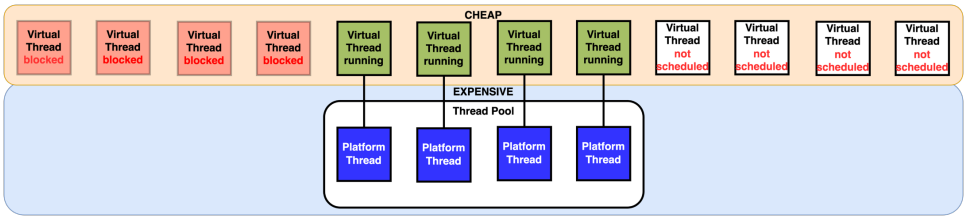
© Marwan Abu-Khalil

6

SIEMENS

Virtual Threads: Lösung für Ressourcen-Probleme klassischer Java Platform-Threads

- **Speicher + CPU**
  - Virtual Threads belegen kaum Speicherplatz (Platform-Thread hingegen: z.B. 2 MB Call-Stack)
  - Werden ohne Zutun des OS geschedult (User-Mode Scheduling): spart CPU-Zeit
- **Skalierbarkeit**
  - Viele Hunderttausend Virtual Threads gleichzeitig in JVM möglich
  - Platform-Threads in dieser Größenordnung nicht sinnvoll (typisch: einige Hundert in Server)
- **Performance**
  - Einsatzgebiet: Vor allem bei signifikantem Anteil blockierender Aufrufe (z.B. in Server-Systemen)
  - Theorie Little's Law: Request konstante Zeit **+** mehr Nebenläufigkeit **=>** höherer Durchsatz



Page 8

© Marwan Abu-Khalil

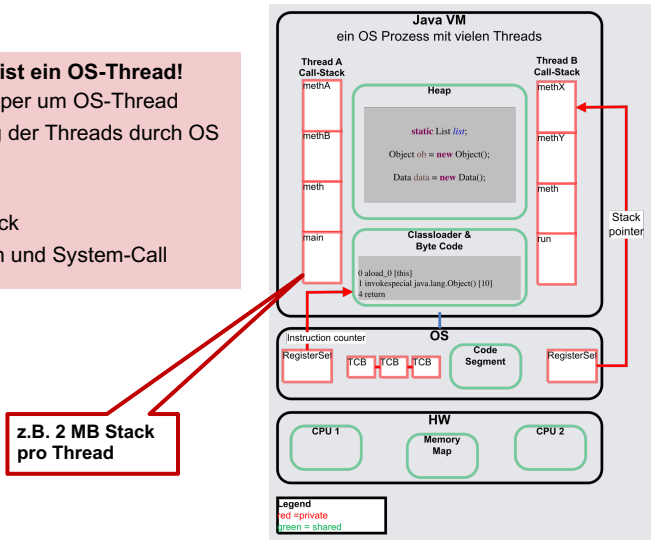
8

Abgrenzung:
Warum ist ein Platform-Thread in Java teuer?

SIEMENS

Jeder Java Platform-Thread ist ein OS-Thread!
- Klasse `Thread` dünner Wrapper um OS-Thread
- Scheduling und Realisierung der Threads durch OS

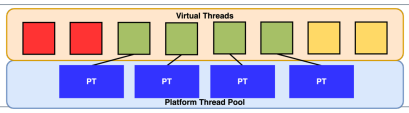
=> Teuer
- Thread hat eigenen Call-Stack
- CPU-Kosten: Context-Switch und System-Call



z.B. 2 MB Stack pro Thread

9

Virtual Threads (Java 21, 2023)
Steckbrief



Ziel
- Klassische Platform-Threads ablösen (in Server-Applikationen)

Innovationsschritt
- Extrem hohe Skalierbarkeit: Problemlos 100.000+ Virtual Threads
- Performancevorteil (falls Architektur und Use-Case geeignet)

Technologie
- API wie klassische Threads, Innenleben ganz anders
- Virtual Threads von Pool mit wenigen Platform-Threads ausgeführt
- Blockierender Aufruf: Automatische Freigabe des Platform-Threads
- Dynamischer Call-Stack (Speicherplatz-Ersparnis)
- Scheduling im User-Mode, ohne OS (CPU-Time Ersparnis)

API wie klassischer Thread

```
// Starten
Thread virtualThread =
  Thread.startVirtualThread(
    () -> {
    // Thread Code
});

// Warten
virtualThread.join();
```

Bewertung
- **Kaum Nachteile**: Quantensprung, Java schließt auf zu modernen Sprachen (Kotlin, Go)
- **Aber**: Strukturierung einer System-Architektur: Klassische Threads besser geeignet

10

SIEMENS

Agenda

Teil I: Virtual Threads

1 Ziel: Thread-per-Request Programmiermodell skalierbar machen

2 Konzept und Architektur: Implizite Freigabe des Carrier-Threads bei Blocking-I/O

3 Anwendung: API und Programmiermodell

4 Verhalten: Was passiert bei einem blockierenden Aufruf?

5 Performance und Skalierbarkeit: Thread-per-Request Server

TEIL II: Sprachfeatures zur Unterstützung von Virtual Threads

6 Structured Concurrency: Lösung der Struktur-Probleme von Threads

7 Scoped Values: Datenzuweisung an Threads

Teil III: Architekturen auf Basis von Virtual Threads

8 Geeignete Architekturen: Systeme, die von Virtual Threads profitieren

9 Ungeeignete Architekturen: Systeme, die nicht von Virtual Threads profitieren

10 Entscheidungskriterien für Migration zu Virtual Threads

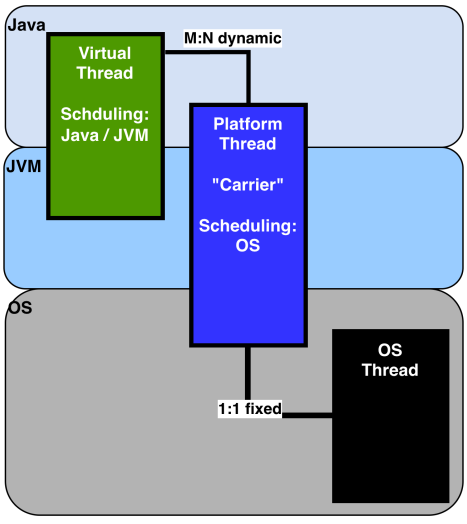
Page 15

15

SIEMENS

Ebenen: Virtual Thread, Platform-Thread, OS-Thread

- **Virtual Thread (effizient)**
  - In Java und JVM realisiert
  - Dynamischer Call-Stack
  - Scheduling in Java / JVM (schnell)
- **Platform-Thread 1:1 OS-Thread (teuer)**
  - Wrapper um OS-Thread
  - Call-Stack statischer Größe
  - Scheduling im Betriebssystem
- **OS-Thread (teuer)**
  - Im Betriebssystem realisiert
  - Im Betriebssystem geschedult
  - System-Calls



Page 16

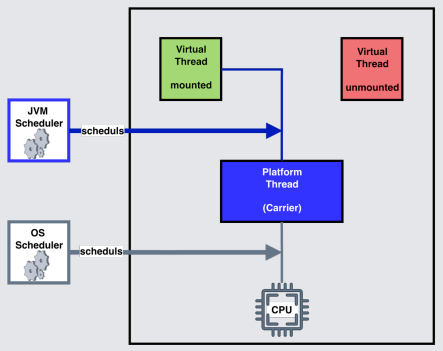
16

Scheduling Zustände: Carrier-Thread und Virtual Thread

SIEMENS

Virtual Thread "mounted"
- Mit einem Platform-Thread verbunden
- Platform-Thread: "**Carrier**" des Virtual Threads

Virtual Thread "unmounted"
- Mit keinem Platform-Thread verbunden
- "Hat keinen Carrier": Kann nicht laufen



Nur wenn beide Scheduler "ja sagen" läuft der Virtual-Thread

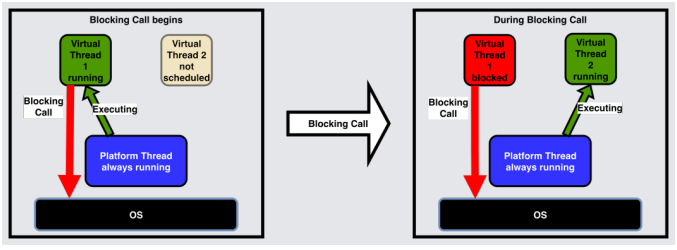
Page 17

17

Blocking Calls: Platform-Thread wird freigegeben

SIEMENS

- Virtual Thread 1 blockiert
- Der Carrier Platform-Thread wird freigegeben
- Der freigegenbe Platform-Thread kann Virtual Thread 2 ausführen
- Nach Ende des blockierenden Aufrufs wird Thread 1 fortgeführt



- Typische blockiernede Aufrufe in Servern
  - Datenbank, REST-Service, …

Page 19

19

SIEMENS

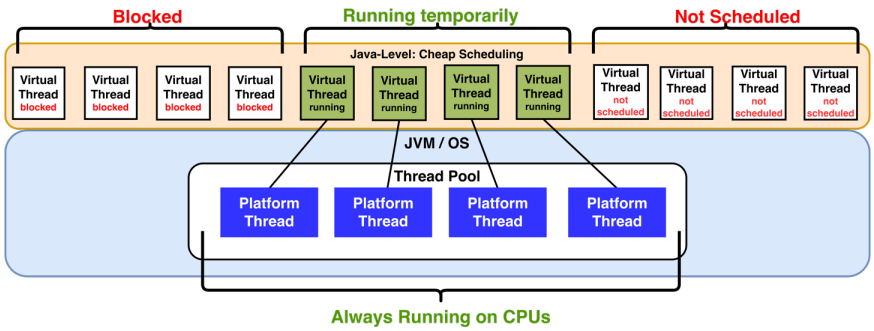
Architektur-Konzept der Virtual Threads

1. Virtual Threads dynamisch auf Platform-Threads abgebildet

- Platform-Thread führt Virtual Thread aus
- Platform-Threads Pool klein (≈CPU-Cores)
- Lebenszyklus: Unterschiedliche Carrier

2. Blockierender Betriebssystem-Aufruf: Carrier wird freigegeben

- Carrier führt anderen Virtual Thread aus
- Keine Speicher-Kosten während Blocking
- Nach Blocking: Virtual Thread fortgeführt



Page 20

20

SIEMENS

Technisches Lösungskonzept für Ressourcen-Problematik

1. Speicher (Stack)

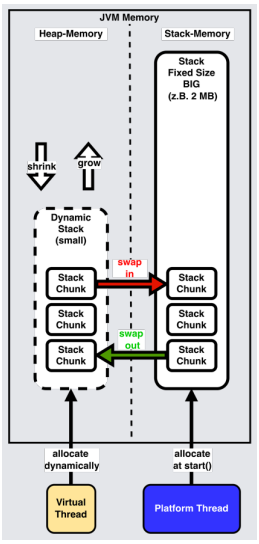
Platform-Thread: Groß (MB)	Virtual Thread: Klein (KB)
• Stack fester Größe • Bei `start()` allokiert	• Dynamischer Stack • Stack-Chunks • Nach Bedarf allokiert • Zur Laufzeit • Swap-In / Swap-Out

Risiko: Dynamischer Stack groß => Vorteil schwindet!
(=> Nur für Kurzläufer geeignet!)

2. CPU (Scheduling)

Platform-Thread: Langsam	Virtual Thread: Schnell
• OS-Scheduling • System-Call • CPU Context-Switch	• Scheduling in der JVM • Ein Methodenaufruf • Ohne OS-System-Call

Konsequenz: Nicht-preemptives Scheduling



Page 21

21

SIEMENS

Agenda

Teil I: Virtual Threads

1 Ziel: Thread-per-Request Programmiermodell skalierbar machen

2 Konzept und Architektur: Implizite Freigabe des Carrier-Threads bei Blocking-I/O

3 Anwendung: API und Programmiermodell

4 Verhalten: Was passiert bei einem blockierenden Aufruf?

5 Performance und Skalierbarkeit: Thread-per-Request Server

TEIL II: Sprachfeatures zur Unterstützung von Virtual Threads

6 Structured Concurrency: Lösung der Struktur-Probleme von Threads

7 Scoped Values: Datenzuweisung an Threads

Teil III: Architekturen auf Basis von Virtual Threads

8 Geeignete Architekturen: Systeme, die von Virtual Threads profitieren

9 Ungeeignete Architekturen: Systeme, die nicht von Virtual Threads profitieren

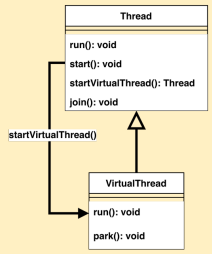
10 Entscheidungskriterien für Migration zu Virtual Threads

24

SIEMENS

Sequentielles Programmiermodell wie klassische Platform-Threads: Kein Umdenken nötig

Virtual Thread extends Thread
=> Schlüssel für einfache Migration



Programmiermodell
sequentiell

```
// Starten
Thread virtualThread = Thread.startVirtualThread(() -> {
  try {
    // Blockieren: Platform-Thread freigeben
    Thread.sleep(1000);
  }
  catch(Exception e){}
});
/ Warten
virtualThread.join();
```


25

Virtual Threads API: Start ähnlich zum klassischen Thread

SIEMENS

- **Start durch Klasse Thread**

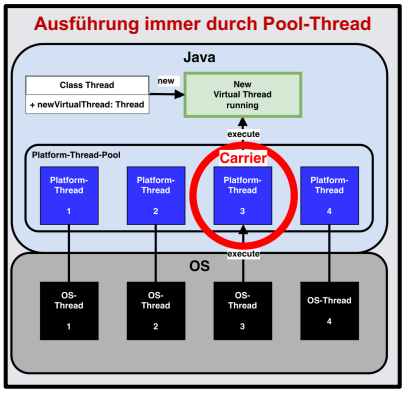
```
// Starten: Klasse Thread
Thread virtualThread =
  Thread.startVirtualThread(() -> { doWorkInThread(); });
```

- **Start durch ExecutorService**

```
// Starten: ExecutorService
Executors.newVirtualThreadPerTaskExecutor()
  .submit(() -> doWorkInThread(); });
```

- **Start durch ThreadBuilder**

```
// Starten: ThreadBuilder
Thread.ofVirtual().name("MyVirtualThread")
  .unstarted(() -> { doWorkInThread(); }).start();
```



Page 26

26

Synchrone vs. asynchrone Programmiermodelle

SIEMENS

Sequentiell / synchron: Thread-per-Request

```
for(int i = 1 ; i <= 8; ++i) {
  final int inputValue = i;
  Thread thread = new Thread(() ->{
    // DB Access: 1 Sec blocking
    var databaseList = readFromDatabase(inputValue);
    // CPU: 1 Sec computing
    var cpuList = computeCPUData(databaseList);
    // UI
    showInUI(cpuList);
  });
  thread.start();
}
```

Einfach aber ineffizient

Asynchrone Ausführung: Reactive-Streams

```
Flux.range(1, 8)
  // DB Access: 1 Sec blocking
  .map( i -> readFromDatabase(i))
  // Stage decoupling
  .publishOn(Schedulers.parallel())
  .map(dbList -> computeCPUData(dbList))
  .publishOn(Schedulers.parallel())
  .subscribe(cpuList -> showInUI(cpuList));
```

Effizient aber proprietäres Prog. Modell

Nicht-sequentiell und asynchron: Completable-Futures

```
for(int i = 1 ; i <= 8; ++i) {
  final int arg = i;
  // Stage 1: DB Access
  CompletableFuture.supplyAsync(()->readFromDatabase(arg))
    // Callback Stage 2: CPU Computation
    .thenApplyAsync((List<Integer> dbResultSet) -> {
      return computeCPUData(dbResultSet);
    })
      // Callback Stage 3: UI Rendering
      .thenAcceptAsync((cpuData) ->{
        showInUI(cpuData);
      });
}
```

Effizient aber komplexe Programmierung

Page 30

30

Virtual Threads: Das Beste aus beiden Welten
Sequentielle Programmierung + asynchrone Ausführung

SIEMENS

Ziel

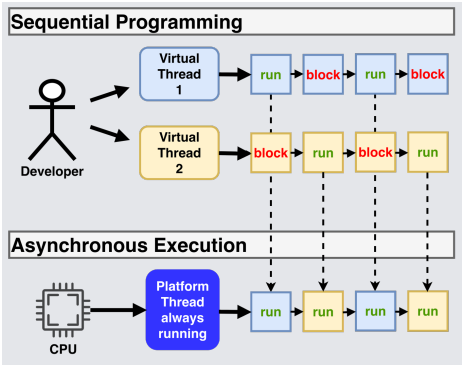
- **Einfache sequentielle Programmierung**
  - Wie klassische Threads (ohne deren Ressourcen-Probleme)
- **Effiziente asynchrone Ausführung**
  - Wie Reactive-Streams (ohne deren Komplexität)

Mechanismus

- OS-Thread: Freigabe bei Blocking
- Asynchronität: Automatisch durch Framework

Technologisches Konzept

- Abbildung Virtual -> Platform-Thread
- Asynchrones I/O des Betriebssystems
- Non-preemtive Scheduling



Page 31

31

SIEMENS

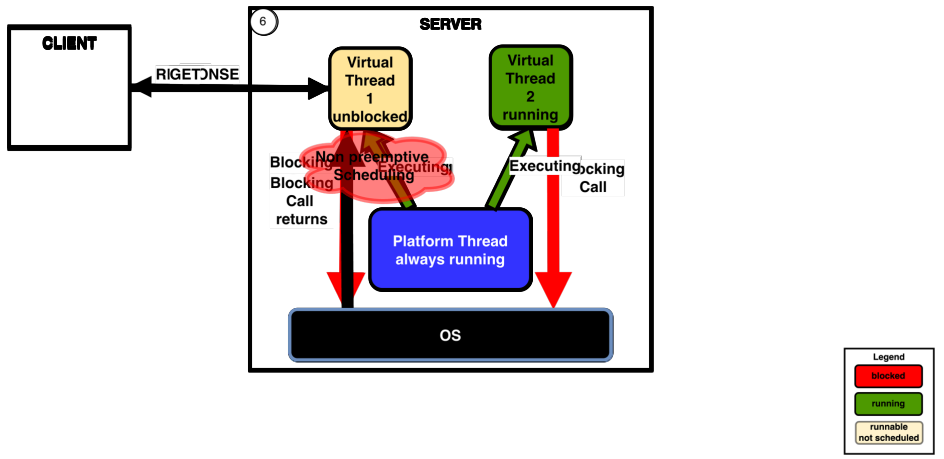
Agenda

Page 34

34

Verhalten bei Blocking-Calls: Freigabe, Scheduling, Wiederaufnahme

SIEMENS



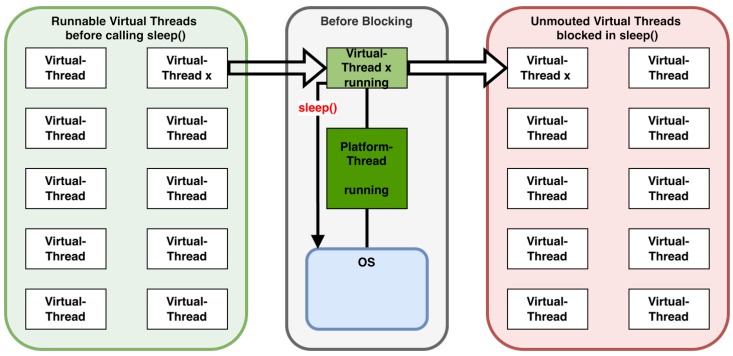
Page 35

35

Bsp.: Effizientes Scheduling Viele Virtual Threads rufen gleichzeitig Blocking-Call

SIEMENS

- **Viele Virtual Threads blockieren, um auf Daten zu warten (z.B. sleep())**
- **Ein einziger Platform-Thread genügt**: Führt sie alle nacheinander aus
- Virtual Threads **blockieren in sleep(): unmouted** Zustand
- **sleep() Aufrufe kehren zurück: Nahezu gleichzeitig!**



Page 38

38

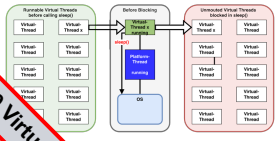
Output zeigt: Viele Virtual Threads simultan von wenigen Platform-Threads ausgeführt

SIEMENS

- **Viele (1000) Virtual Threads starten**
  - jeder ruft sleep() auf
- **Wenige (14) Platform-Threads**
  - realisieren Ausführung
  - innerhalb weniger Millisekunden

```
for(int i = 0; i < 1000; ++i) {
  final int cnt = i;
  Thread.startVirtualThread(() ->{
    System.out.println("Virtual Thread Nr. " + cnt
      + " " + Thread.currentThread());

    // Blocking call releases Carrier-Thread
    Thread.sleep(1000);
  }); // … Exceptionhandling
}
```



1000 Virtual Threads

14 Platform-Threads

15 Millisekunden

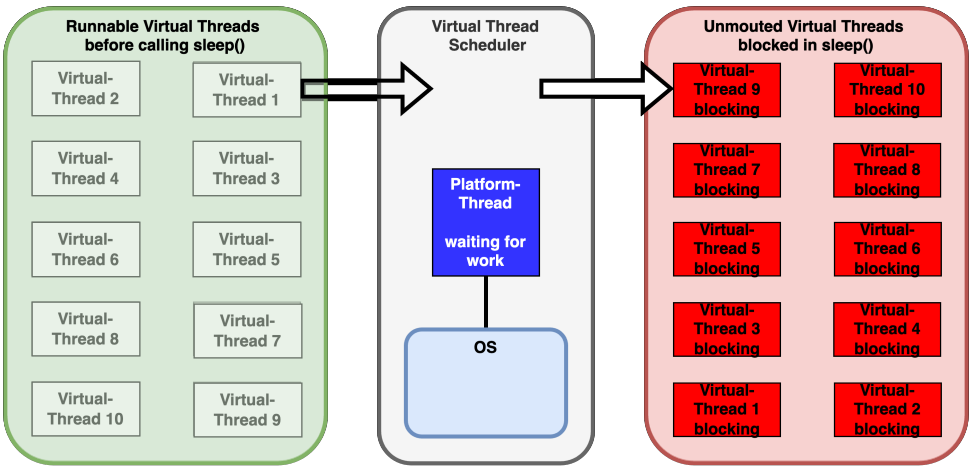
```
Virtual-Thread Nr.  979 VirtualThread[#1017]/runnable@ForkJ…-1-worker-6 at: …llis
Virtual-Thr…  Nr.  994 VirtualThread[#1032]/runnable@ForkJo…-1-worker-4 at: 15 mi…llis
Virtual-Threa…  Nr.  995 VirtualThread[#1033]/runnable@ForkJoi…-1-worker-10 at: 15 millis
Virtual-Thread Nr.  997 VirtualThread[#1035]/runnable@ForkJoinPool-1-worker-2 at: 15 millis
Virtual-Thread Nr.  999 VirtualThread[#1037]/runnable@ForkJoinPool-1-worker-11 at: 15 millis
Virtual-Thread Nr.  998 VirtualThread[#1036]/runnable@ForkJoinPool-1-worker-7 at: 15 millis
Virtual-Thread Nr.  991 VirtualThread[#1029]/runnable@ForkJoinPool-1-worker-14 at: 15 millis
Virtual-Thread Nr.  985 VirtualThread[#1023]/runnable@ForkJoinPool-1-worker-1 at: 15 millis
```

Page 39

39

Scheduler Dynamik: Blocking

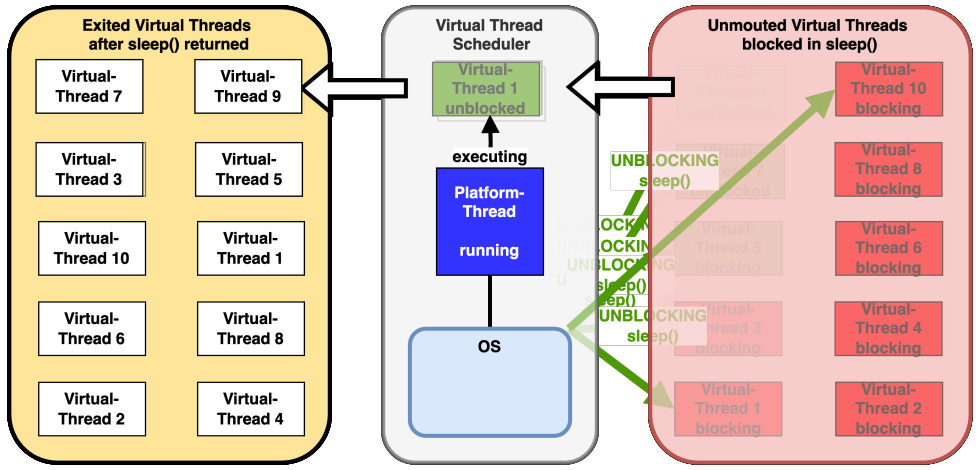
SIEMENS



40

SIEMENS

Scheduler Dynamik: Unblocking



41

SIEMENS

Agenda

Teil I: Virtual Threads

1 Ziel: Thread-per-Request Programmiermodell skalierbar machen
2 Konzept und Architektur: Implizite Freigabe des Carrier-Threads bei Blocking-I/O
3 Anwendung: API und Programmiermodell
4 Verhalten: Was passiert bei einem blockierenden Aufruf?
5 Performance und Skalierbarkeit: Thread-per-Request Server

TEIL II: Sprachfeatures zur Unterstützung von Virtual Threads

6 Structured Concurrency: Lösung der Struktur-Probleme von Threads
7 Scoped Values: Datenzuweisung an Threads

Teil III: Architekturen auf Basis von Virtual Threads

8 Applikationen und Architekturen, **die von Virtual Threads profitieren**
9 **Nicht geeignete** Applikationen und Architekturen
10 **Entscheidungskriterien** für Migration zu Virtual **Threads**

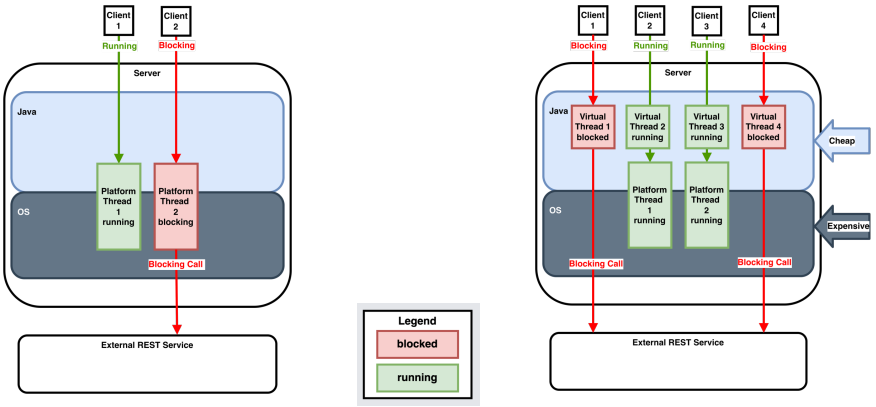
42

SIEMENS

Skalierbare Server im Thread-Per-Request Stil

Klassisches Thread Modell
Teure Platform-Threads blockieren bei I/O
Sind untätig, belegen aber Speicher
=> Wenige gleichzeitige Clients

Virtual Thread Modell
Platform-Threads laufen immer
Nur Virtual Threads blockieren
=> Viele gleichzeitige Clients



Page 43

43

SIEMENS

Theoretischer Hintergrund: Little's Law

- **Hauptmotivation für Virtual Threads: Skalierbarkeit von Server-Applikationen**
  - Skalierbarkeit: Schlüssel für Performance im Sinne von Durchsatz
  - Effekt: Performance-Vorteile und mehr Stabilität im Vergleich zu Platform-Threads
- **Littles Law: „Skalierbarkeit erhöht Performance"**
  - Durchsatz := Anzahl paralleler Prozesse / durchschnittliche Antwortzeit
  - **Erhöhte Nebenläufigkeit** bei gleichbleibender Antwortzeit **=> Durchsatz steigt**
  - Genau das tun Virtual Threads im Falle blockierender OS-Aufrufe

T = N / d

T := Throughput
N := Number of concurrent tasks
d := Duration of processing a single request

John Little 1954: Statistischer Beweis für diesen intuitiv offensichtlichen Zusammenhang

Siehe: Little's Law (https://en.wikipedia.org/wiki/Little%27s_law)

Page 44

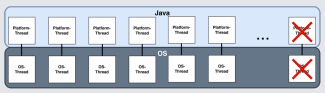
44

SIEMENS

Beispiel: Skalierbarkeit

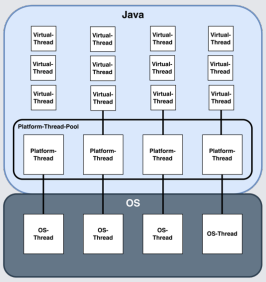
- **100.000 Platform-Threads starten**
  - 12.000 Threads Absturz: **OutOfMemoryError**
    (auf meinem Rechner: MacBook Pro, M4 Pro, 48 GB)

Platform-Threads: Crash



- **Alternative: 1.000.000 Virtual Threads (1 Mio!)**
  - Starten problemlos und schnell
  - Existieren simultan in der JVM
- **Entscheidend: Blockierender Aufruf**
  - z.B. sleep()
- **Erfolg der Virtual Threads Architektur**
  - Massiver Anstieg der Anzahl simultaner Tasks
  - Konstante Response Time
  - Little's Law **=>** Durchsatz-Steigerung

Virtual Threads: Skalieren



Page 45

45

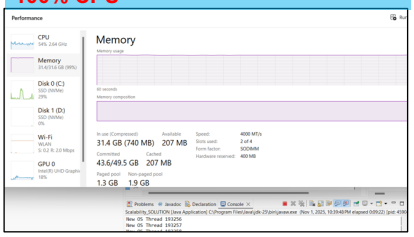
Code zu dem Beispiel:
100.000 Platform-Threads: Absturz

Tatsächliche Grenzen hängen ab von OS, HW, Konfiguration

- Platform-Threads starten
- `sleep()`: Blockierender Aufruf

```
for(int i = 0; i < 100_000; ++i) {
  final int cnt = i;
  // launch platform thread
  Thread thread = new Thread(()->{
     System.out.println("New OS Thread"+ cnt);
        // sleep (Blocking-Call)
      Thread.sleep(10000);
  });
  thread.start();
} // exceptionhandling omitted
```

PC, 32 GB bei 250.000 Threads
100% Speicherauslastung
100% CPU



Verhalten Mac M4 Pro / 48 GB:
Nach 12.000 Threads OutOfMemoryError

```
Console
Performance_BASE [Java A
Job 12252 started
Job 12253 started
Job 12254 started
Job 12255 started
[1.826s][warning][os,thread] Failed to start thread "Unknown thread" - pthread_create failed (EAGAIN) for attributes: stacksize: 2048k, guardsize: 16k, det
[1.826s][warning][os,thread] Fa
Exception in thread "main" java.lang.OutOfMemoryError: unable to create native thread: possibly out of memory or process/resource limits reached
        at java.base/java.lang.Thread.start0(Native Method)
        at java.base/java.lang.Thread.start(Thread.java:1444)
        at java.base/java.lang.System$1.start(System.java:2230)
        at java.base/jdk.internal.vm.SharedThreadContainer.start(SharedThreadContainer.java:147)
        at java.base/java.util.concurrent.ThreadPoolExecutor.addWorker(ThreadPoolExecutor.java:899)
        at java.base/java.util.concurrent.ThreadPoolExecutor.execute(ThreadPoolExecutor.java:1327)
```

Page 46

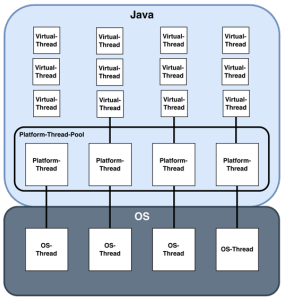
46

1 Million Virtual Threads
Existieren problemlos simultan (!)

SIEMENS

```
final long start = System.currentTimeMillis();
// 1 million virtual threads
for(int i = 0; i < 1_000_000; ++i) {
  final int cnt = i;
  // starting new virtual thread
  Thread.startVirtualThread(()->{
    // blocking call 10 seconds
     Thread.sleep(10000);
    // wake up
    System.out.println("Virtual Thread " + cnt +" woke up at:
      " + (System.currentTimeMillis() - start));
  });
} // exceptionhandling omitted
```

sleep(): alle leben simultan



Console

<terminated> VirtualThreadIntro (1) [Java Application] /Library/Java

```
Virtual Thread 999991 woke up at: 14332 millis
Virtual Thread 999999 woke up at: 14332 millis
Virtual Thread 999919 woke up at: 14332 millis
Virtual Thread 999977 woke up at: 14332 millis
Virtual Thread 999891 woke up at: 14332 millis
Virtual Thread 999998 woke up at: 14332 millis
Virtual Thread 999996 woke up at: 14332 millis
```

1.000.000 Virtual Threads
- gestartet
- simultan im Programm existent
- blockiert
- wieder aufgewacht
- < 5 Sekunden Overhead!!!

Page 47

47

Performance-Steigerung durch Virtual Threads
Faktor 10!!!

SIEMENS

```
// Variante 1 Platform-Threads: ca. 100 Sekunden
//ExecutorService pool = Executors.newFixedThreadPool(10_000); // 10.000 Platform-Threads

// Variante 2 Virtual Threads: ca. 10 Sekunden
ExecutorService pool = Executors.newVirtualThreadPerTaskExecutor();

long start = System.currentTimeMillis();

for(int i = 0 ; i < 100_000; ++i) {
  final int loop_cnt = i;
  pool.submit(() -> {
    // Blockierender Aufruf
    Thread.sleep(10_000);
    System.out.println("Job " + loop_cnt + " woke up " +
        (System.currentTimeMillis() - start) / 1000 + " Seconds ");
  } (Exception Handling nicht gezeigt)
}
```

Platform-Thread Variante

Virtual Thread Variante

ca. 100 Sekunden mit Platform-Thread-Pool

```
Job 99986 woke up 103 Seconds
Job 99987 woke up 103 Seconds
Job 99998 woke up 103 Seconds
Job 99999 woke up 103 Seconds
Job 99997 woke up 103 Seconds
Job 99982 woke up 103 Seconds
```

ca. 10 Sekunden mit Virtual Threads

```
Job 99979 woke up 11 Seconds
Job 99971 woke up 11 Seconds
Job 99970 woke up 11 Seconds
Job 99219 woke up 11 Seconds
Job 99982 woke up 11 Seconds
Job 99996 woke up 11 Seconds
```

Page 48

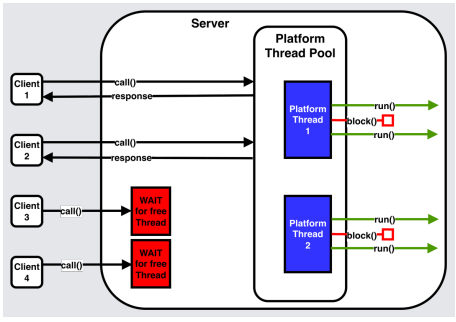
48

Architektur-Optionen im Thread-per-Request Server
Platform-Thread-Pool vs. Virtual Threads

SIEMENS

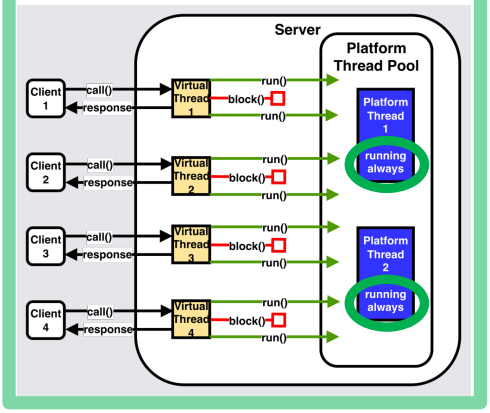
Platform-Threadpool: LANGSAM

- Begrenzte Skalierbarkeit
- Platform-Threads blockieren bei I/O
- Clients warten



Virtual Threads: SCHNELL

- Nur Virtual Threads blockieren
- Platform-Threads laufen immer
- Indirektion => Skalierbarkeit



Page 50

50

Virtual Threads Fazit
Fokus: Skalierbare Server

SIEMENS

Positiv

- **Extrem hohe Skalierbarkeit**
  - Viele Hunderttausende gleichzeitig
- **Können Performance massiv erhöhen**
  - Durchsatz
  - Little‘s Law
- **Sparen Ressourcen**
  - Memory und CPU
  - Platform-Threads
- **Einfaches Programmiermodell**
  - Wie klassische Threads
  - Einfache Umstellung
  - Sequentiell

Negativ

- **Nur effektiv, wenn viel Blocking**
  - Nur für I/O-lastige Applikationen
  - Nur wenn Blocking-Zeit signifikant
  - Für rechenintensive Applikation ungeeignet
- **Nur effektiv bei extrem viel Nebenläufigkeit**
  - Dimension in die Platform-Threads nicht skalieren können (Tausende, evtl. Mio.)
- **Nicht ideal für lang laufende Aufgaben**
  - Stack kann dabei groß werden
  - Performance-Vorteil schwindet
- **Nicht-preemptives Scheduling**
  - Kann Semantik der Applikation verändern

Page 52

52

SIEMENS

Agenda

Teil I: Virtual Threads

1 **Ziel:** Thread-per-Request Programmiermodell skalierbar machen
2 **Konzept und Architektur**: Implizite Freigabe des Carrier-Threads bei Blocking-I/O
3 **Anwendung:** API und Programmiermodell
4 **Verhalten**: Was passiert bei einem blockierenden Aufruf?
5 **Performance und Skalierbarkeit:** Skalierbare Server

TEIL II: Sprachfeatures zur Unterstützung von Virtual Threads

6 Structured Concurrency: Lösung der Struktur-Probleme von Threads
7 Scoped Values: Datenzuweisung an Threads

Teil III: Architekturen auf Basis von Virtual Threads

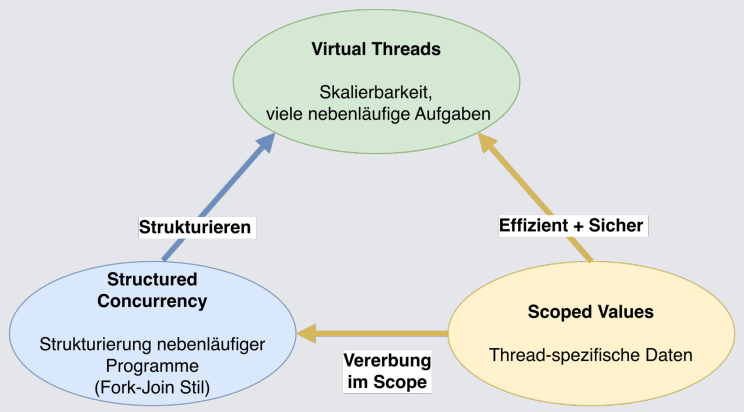
8 Applikationen und Architekturen, die von **Virtual Threads profitieren**
9 **Nicht geeignete** Applikationen und Architekturen
10 **Entscheidungskriterien** für Migration zu Virtual **Threads**

Page 53

53

SIEMENS

Drei Technologien des Project Loom im Zusammenspiel



Seit 2017 ongoing
- Ron Pressler, Alan Bateman et. al.
- Java 19, 2021 erste Preview
- Java 21, 2023 erstes Release

Page 54

54

SIEMENS

Agenda

Teil I: Virtual Threads

1 **Ziel:** Thread-per-Request Programmiermodell skalierbar machen
2 **Konzept und Architektur**: Implizite Freigabe des Carrier-Threads bei Blocking-I/O
3 **Anwendung:** API und Programmiermodell
4 **Verhalten**: Was passiert bei einem blockierenden Aufruf?
5 **Performance und Skalierbarkeit:** Skalierbare Server

TEIL II: Sprachfeatures zur Unterstützung von Virtual Threads

6 Structured Concurrency: Lösung der Struktur-Probleme von Threads
7 Scoped Values: Datenzuweisung an Threads

Teil III: Architekturen auf Basis von Virtual Threads

8 Geeignete Architekturen: Systeme, die von Virtual Threads profitieren
9 Ungeeignete Architekturen: Systeme, die nicht von Virtual Threads profitieren
10 Entscheidungskriterien für Migration zu Virtual Threads

Page 55
© Marwan Abu-Khalil

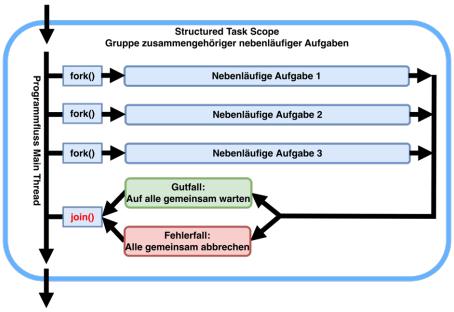
55

SIEMENS

Structured Concurrency

Lösung für (einige) Komplexitäts-Probleme der nebenläufigen Programmierung

- **Ziel: Gruppierung nebenläufiger Tasks**
  - Verständlichkeit nebenläufiger Abläufe
  - Handhabung der Gruppe als Ganzes
  - Verschachtelte Subscopes
- **Exceptionhandling**
  - Falls eine Task Exception wirft:
  - Abbruch der gesamten Gruppe
- **`Joiner`**
  - Definieren genaues Verhalten
  - Anpassung an verschiedene Use-Cases
  - Vordefinierte / selbstgeschriebene `Joiner`



Java 25: JEP 505, 5. Preview
https://openjdk.org/jeps/505 (Static open Method)
Java 26: JEP 525, 6. Preview: (allSuccessful: List)
Java 27: JEP 533, 7. Preview: (awaitAll removed)

Page 57
© Marwan Abu-Khalil

57

StructuredTaskScope API

SIEMENS

- `StructuredTaskScope.open()`: Fasst mehrere Threads zu einer Einheit zusammen
- `StructuredTaskScope.fork()`: Startet neuen Thread
- `StructuredTaskScope.join()`: Wartet auf alle Threads im Scope, wirft im Fehlerfall Exception

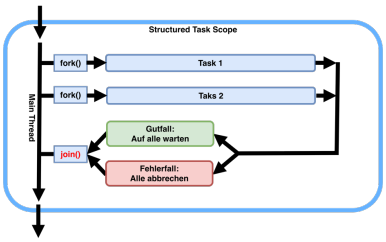
Rahmen-Code für einen Scope

```
// Scope öffnen
try(var scope = StructuredTaskScope.open()){

  // Subtask 1 starten
  Subtask<Void> subTask1 = scope.fork(() -> {
     // Code innerhalb der Task 1
  });

  // Subtask 2 starten
  Subtask<Void> subTask2 = scope.fork(() -> {
    // Code innerhalb der Task 2
  });

  // Auf Beendigung beider Subtasks warten
  scope.join();
}
```



Page 60

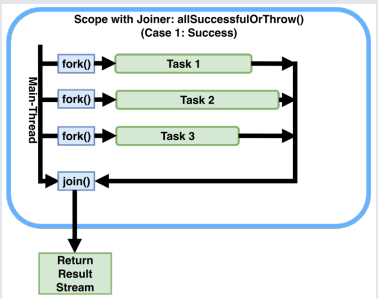
60

Joiner definieren Verhalten (bzgl. Warten / Beendigung) Bsp: allSuccsessfulOrThrow()

SIEMENS

Fall 1: Erfolg

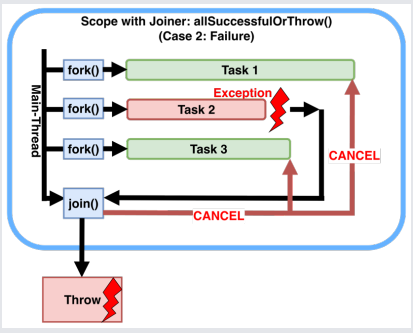
- Alle Tasks beenden sich ohne Exception
- Tasks haben identischen Ergebnistyp
- Return: Stream der Ergebnisse aller Tasks



(Ab JDK 26 : `List` statt `Stream`)

Fall 2: Fehler

- Eine der Tasks wirft eine Exception
- Joiner beendet alle anderen Tasks
- `join()` wirft Exception weiter



(Kontrollfluss vereinfacht dargestellt)

Page 62

62

Vordefinierte Joiner im JDK 25

SIEMENS

`awaitAllSuccessfulOrThrow()` Default Joiner
Warte auf alle
- Für **unterschiedliche Ergebnistypen**
- `join()` gibt keine Werte zurück

`anySuccessfulResultOrThrow()`
Nimm den ersten
- Ergebnis der **ersten erfolgreichen** Subtask
- Andere Subtasks canceln sobald eine fertig ist

`allUntil(Predicate<Subtask> isDone …)`
Bedingung prüfen
- **Bricht Scope ab**, sobald **Predicate true** ist
- Greift auf Interna der Subtasks zu

`allSuccessfulOrThrow()`
Warte auf alle
- Für **identische Ergebnistypen**
- Beendet Scope, sobald eine Subtask scheitert
- `join()` Gibt `Stream` zurück (Java 26: `List`)

`awaitAll()`
- Cancelt Scope im Fehlerfall nicht
- (entfällt in Java 27)

Page 63

63

Custom-Joiner: Methoden für Start, Beendigung, Ergebnislieferung

SIEMENS

Beispiel-Implementierung für `onComplete()`

```
class CustomJoiner<T> implements Joiner<T, Stream<T>> {

  // onComplete() must be threadsafe
  private final Queue<T> results = new ConcurrentLinkedQueue<>();


  public boolean onComplete(Subtask<? extends T> subtask) {
    if (subtask.state() == Subtask.State.SUCCESS) {
      results.add(subtask.get());
    }

    // true to cancel the scope, otherwise false
    return false;
  }

 //…
}
```

Bei Beendigung einer Subtask

Zugriff auf Subtask State

Ergebnisaufbereitung

Verhaltenssteuerung

Weitere Interface-Methoden hier weggelassen

Page 71

71

SIEMENS

Agenda

Teil I: Virtual Threads

1 **Ziel:** Thread-per-Request Programmiermodell skalierbar machen
2 **Konzept und Architektur**: Implizite Freigabe des Carrier-Threads bei Blocking-I/O
3 **Anwendung:** API und Programmiermodell
4 **Verhalten**: Was passiert bei einem blockierenden Aufruf?
5 **Performance und Skalierbarkeit:** Thread-per-Request Server

TEIL II: Sprachfeatures zur Unterstützung von Virtual Threads

6 Structured Concurrency: Lösung der Struktur-Probleme von Threads
7 Scoped Values: Datenzuweisung an Threads

Teil III: Architekturen auf Basis von Virtual Threads

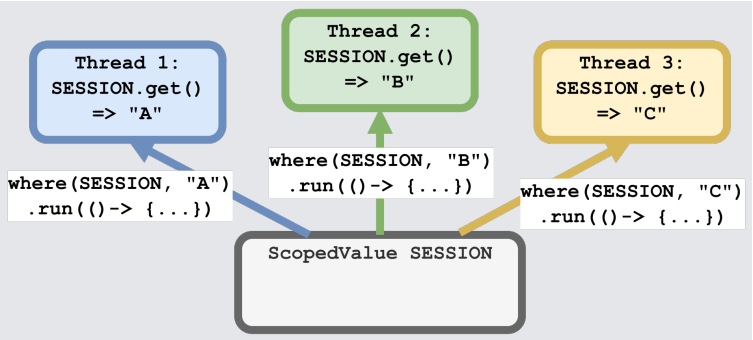
8 Geeignete Architekturen: Systeme, die von Virtual Threads profitieren
9 Ungeeignete Architekturen: Systeme, die nicht von Virtual Threads profitieren
10 Entscheidungskriterien für Migration zu Virtual Threads

Scoped Values Grundkonzept

SIEMENS

Semantik: Threadspezifische Variable
- Meist `static final` Variable
- Aber: Jeder Thread sieht anderen Wert
- Wert an entfernten Code “durchreichen”

Anwendung bei Thread-per-Request
- DB-Connection pro Thread
- Client-Session pro Thread



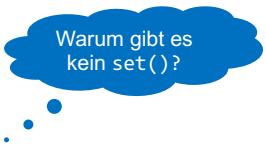
Scoped Values JEP 506 https://openjdk.org/jeps/506

Scoped Values API

SIEMENS

Definierter Gültigkeitsbereich (Scope)

- `where`(…) bindet den Wert
- `run`(…) definiert den Scope
- Nur im Scope ist der Wert vorhanden



- **Deklaration und Erzeugung**
  ```
  static final ScopedValue<...> S_V = ScopedValue.newInstance();
  ```
- **Wert setzen**
  ```
  ScopedValue.where(S_V, <value>)
  ```
- **Scope definieren** und darin `Runnable` ausführen (Startet keinen Thread!)
  ```
  ScopedValue.Carrier.run(() -> { ... S_V.get(); ... });
  ```
- **Auf Wert zugreifen**
  ```
  S_V.get()
  ```

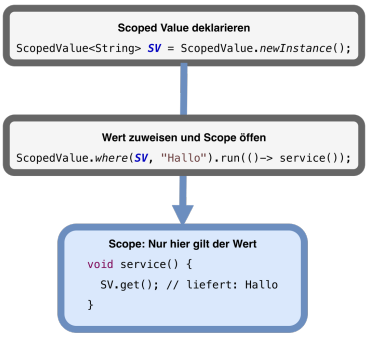
Scope

78

Scoped Values Features

SIEMENS

- **Vererbung im `StructuredTaskScope`**
  - Skaliert gut
  - Bisher: `ThreadLocal<T>` (skaliert nicht gut)
- **Immutable**
  - Der Wert kann nicht verändert werden
  - Robustes Programmiermodell
  - Wesentlicher Unterschied zu Thread-Local
- **Nested dynamic scopes**
  - Innerer Scope
  - Wert kann überschrieben werden
  - Nach Ende des Scopes gilt vorheriger Wert
- **Caching (Performance-Optimierung)**
  - Pro Thread existiert ein kleiner `ScopedValue` Cache
  - Speed / Memory Tradeoff konfigurierbar

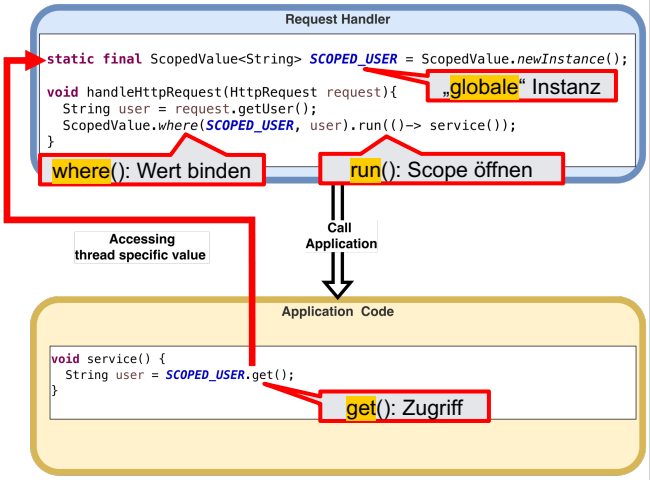


79

Anwendungsbeispiel: User aus Request an Anwendungscode übermitteln

SIEMENS

Request Handler Code
- User aus Request extrahieren
- User an Thread binden
- Request in Thread behandeln



Anwendungscode (weit entfernt)
- `User` von überall zugreifbar
- Thread-spezifischer Wert
- Nicht in Methoden-Signatur nötig

Page 80

80

Fazit Structured Concurrency und Scoped Values Stuktur, Robustheit, Skalierbarkeit

SIEMENS

Positiv

- **Strukturierung** nebenläufiger Aufgaben im Fork-Join Stil steigert die Klarheit
- Hohe **Skalierbarkeit** der Scoped Values
- **Virtual Threads,** Scoped Values, Structured Concurrency: Aufeinander abgestimmt
- Gutes **Programmiermodell** für Nebenläufigkeit

Negativ

- Structured Concurrency leider noch immer im **Preview-Stadium (6. Vorlage!)**
- Andere Sprachen erreichen **klareres Programmiermodell** für Nebenläufigkeit
  - Z.B Go: Goroutines und Channels
- Viele und überlagernde **Begrifflichkeiten**
  - Scope des `ScopedValue`: `run()`
  - Scope des `StructuredTaskScopes`: `open()`

Page 86

86

SIEMENS

Agenda

Teil I: Virtual Threads

1 **Ziel:** Thread-per-Request Programmiermodell skalierbar machen
2 **Konzept und Architektur**: Implizite Freigabe des Carrier-Threads bei Blocking-I/O
3 **Anwendung:** API und Programmiermodell
4 **Verhalten**: Was passiert bei einem blockierenden Aufruf?
5 **Performance und Skalierbarkeit:** Thread-per-Request Server

TEIL II: Sprachfeatures zur Unterstützung von Virtual Threads

6 **Structured Concurrency**: Lösung der Struktur-Probleme von Threads
7 **Scoped Values**: Datenzuweisung an Threads

Teil III: Architekturen auf Basis von Virtual Threads

8 Geeignete Architekturen: Systeme, die von Virtual Threads profitieren
9 Ungeeignete Architekturen: Systeme, die nicht von Virtual Threads profitieren
10 Entscheidungskriterien für Migration zu Virtual Threads

Page 87 © Marwan Abu-Khalil

87

SIEMENS

Notwendige Voraussetzungen um Performance-Vorteil durch Virtual Thread Migration zu erzielen

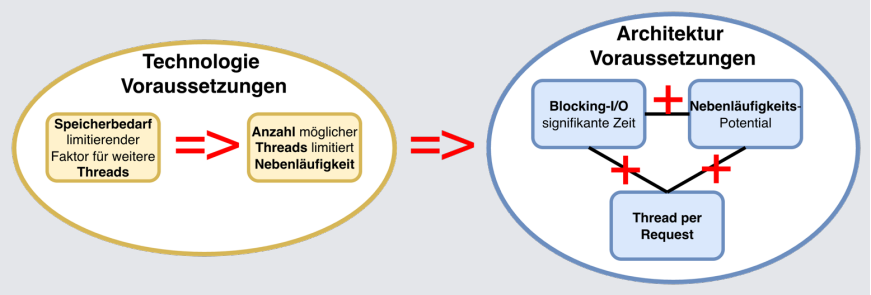
1. **Performance ist durch die Anzahl verfügbarer Threads limitiert**
   - D.h. Applikation könnte schneller laufen, wenn sie mehr Threads zur Verfügung hätte
   - Virtual Threads steigern nur Nebenläufigkeit, weder Parallelität noch Rechengeschwindigkeit
2. **Signifikante Zeit wird mit blockierenden Aufrufen verbracht**
   - D.h. Applikation ist „I/O-bound“
   - Grund: Virtual Threads ändern lediglich das Systemverhalten bei blockierenden OS-Calls

Page 89 © Marwan Abu-Khalil

89

SIEMENS

Argumentationskette für erfolgreiche Migration



93

SIEMENS

Kategorien geeigneter Systeme

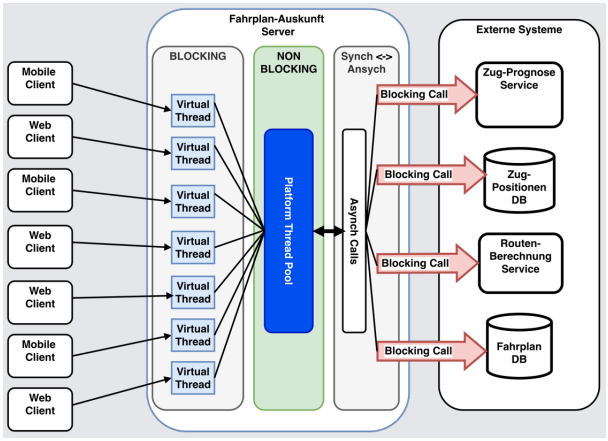
- **Skalierbare Server (Bsp. Fahrgast-Informationssystem)**
  - Steigende **Anzahl an Clients**
  - **Thread-per-Request**: Client-Request von Anfang bis Ende im selben Thread
  - **Blockierende Backend-Aufrufe**: DB, Customer Repository, Kreditkarten-System
- **Verteilte Systeme mit vielen Remote Calls (Bsp. MCP-Server)**
  - Microservice-Landschaft: **Interne blockierende Aufrufe**
  - REST-Calls zwischen Services
- **Batch Systeme mit blockierenden Aufrufen (Bsp. Prognose-System für Flugbuchungen)**
  - **Datenbeschaffung** aus unterschiedlichen Quellen

Migration nur aussichtsreich,
wenn Anzahl Threads limitierender Faktor (viele 10.000)!!!

95

Bsp.: Fahrgast-Informationssystem Skalierbarer Server

- **Backend Calls**
  - Blockieren
- **Java Virtual Threads Runtime**
  - Asynch-IO des OS transparent
- **Platform-Threads**
  - Laufen kontinuierlich
- **Virtual Threads**
  - Bedienen Clients
  - Ohne Kosten beim Blockieren



96

Agenda

98

Eigenschaften die gegen Einsatz von Virtual Threads sprechen

SIEMENS

- **CPU-bound Applikationen profitieren nicht**
  - Virtual Thread Vorteil lediglich: Kein Speicherplatz-Verbrauch bei blockierenden Aufrufen

Virtual Thread rechnet nicht schneller!

- **Wenige Aufgaben nebenläufig ausführbar**
  - Anzahl Clients begrenzt
  - Wenige gleichartige Aufgaben (z.B. ein UI-Thread, ein Hintergrund-Thread für Datenbeschaffung)
  - Sequentieller Algorithmus
  - Daten nicht disjunkt partitionierbar

- **Andere Ressourcen Bottlenecks als Speicherplatz**
  - Backend-Systeme nicht skalierbar: Z.B. Daten aus DB-Instanz auf schwachem Server
  - CPU ist bereits ausgelastet
  - Sockets, Network-Bandwidth
  - Konfiguration des OS, der Frameworks

Page 100

100

Kategorien und Beispiele ungeeigneter Systeme

SIEMENS

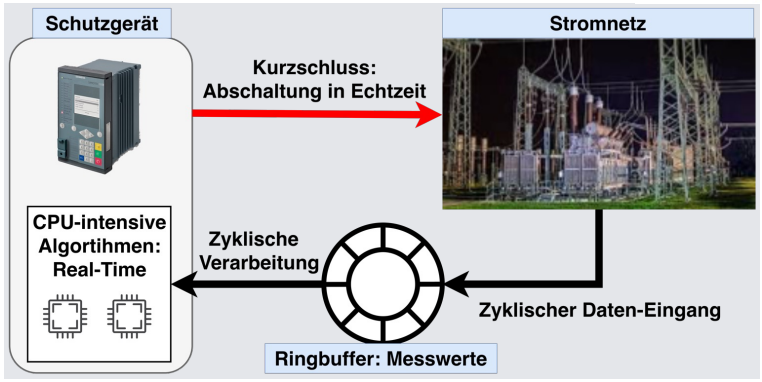
$$\begin{pmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{pmatrix} \cdot \begin{pmatrix} 8 & 9 \\ 10 & 11 \\ 12 & 13 \end{pmatrix}$$

Numbercruncher: CPU-intensiv rechnen	
Kategorien	**Beispiele**
Mathematische Berechnungen	• Optimierungsalgorithmen • Numerik • Matrix-Multiplikation
Simulationen / Prognosen	• Wetterprognosen • Digital-Twin: Simulation von Produktionsanlagen • (Daten Import / Export profitiert von Virtual Threads)
Realtime Systeme	• Automatisierungstechnik (z.B. Energienetze) • Latenz ist wichtiger als Durchsatz
Algorithmen, Suchen, Sortieren, …	• Kein Blocking-I/O, falls Daten im RAM
Kompression, Kryptographie	• Falls Fokus auf Berechnung • (Für Daten-Import / Export evtl. Virtual Threads)
LLM	• Training • Inference

Page 101

101

Bsp. Embedded Realtime System: OS-Threads bieten mehr Kontrolle als Virtual Threads



Echtzeit: Latenz wichtiger als Durchsatz
=>
Virtual Threads nicht sinnvoll

102

Alternativen zu Virtual Threads innerhalb JVM / JDK

API / Usecase	Virtual Thread	Platform-Thread	Fork-Join-Task	Parallel-Stream	Reactive-Stream
Blocking-IO	Ja: Perfekter Use-Case	Ja	Nein	Nein	Ja
System Architektur Struktur	Ja	Ja: Perfekter Use-Case	Nein	Nein	Ja
Rekursiver Algorithmus	Ja	Eher nicht	Ja: Perfekter Use-Case	Nein	Eher nicht
Endliche Pipeline Verarbeitung	Eher nicht	Eher nicht	Nein	Ja: Perfekter Use-Case	Ja
Unendliche Streams	Nein	Ja	Nein	Nein	Ja: Perfekter Use-Case

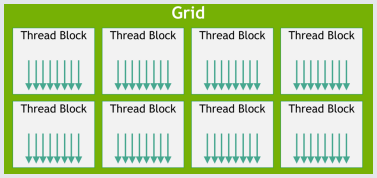
103

SIEMENS

Alternativen zu Virtual Threads außerhalb der JVM

- **GPU: Massiv paralleles Rechnen**
  - LLM-Training und Inference
  - Matrix-Operationen, Numerik, …
  - z.B. mit Tornado VM
  - **Virtual Threads hier nutzlos**

- **Viele 1000 GPU-Threads rechnen gleichzeitig!**
- **Virtual Threads: Nur gleichzeitiges Blockieren**



NVIDIA CUDA

- **Scale out**
  - Cluster / Cloud
  - z.B. Container mit Kubernetes
  - **Virtual Threads reduzieren Memory Bedarf**

kubernetes

Page 104

104

SIEMENS

Agenda

Teil I: Virtual Threads

1 **Ziel:** Thread-per-Request Programmiermodell skalierbar machen
2 **Konzept und Architektur**: Implizite Freigabe des Carrier-Threads bei Blocking-I/O
3 **Anwendung:** API und Programmiermodell
4 **Verhalten**: Was passiert bei einem blockierenden Aufruf?
5 **Anwendung und Programmierung:** API und Frameworks

TEIL II: Sprachfeatures zur Unterstützung von Virtual Threads

6 **Structured Concurrency**: Lösung der Struktur-Probleme von Threads
7 **Scoped Values**: Datenzuweisung an Threads

Teil III: Architekturen auf Basis von Virtual Threads

8 **Geeignete Architekturen:** Systeme, die von Virtual Threads profitieren
9 **Ungeeignete Architekturen:** Systeme, die nicht von Virtual Threads profitieren
10 Entscheidungskriterien für Migration zu Virtual Threads

Page 120

120

Schnelltest Checkliste
Virtual Threads Migration aussichtsreich?

SIEMENS

Geeignete Architektur

Notwendig
- Anzahl Threads limitiert Skalierbarkeit
- Signifikante Zeit in Blocking-I/O
- Speicherbedarf limitiert weitere Threads

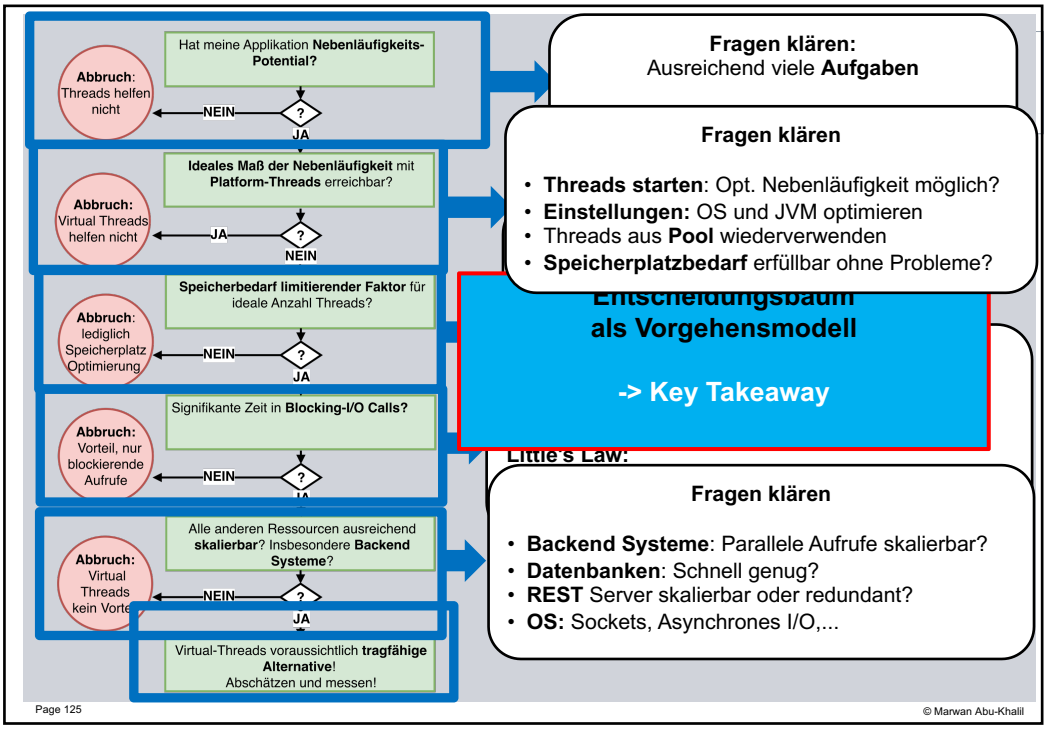
Unterstützend
- Skalierbarkeits-Anforderung (Viele Clients)
- Begrenzter verfügbarer Speicher
- Verteiltes System mit Kommunikation
- Backend-Systeme sind skalierbar
- Kurzlaufende Aufgabe pro Thread

Ungeeignete Architektur

Ausschließend
- CPU-bound Applikation
- Kein Blocking-I/O

Fragwürdig
- Begrenztes Potential für Nebenläufigkeit
- Mehr Threads: Keine Perfomance-Steigerung
- Wenige Clients
- Keine unabhängigen Aufgaben / Daten
- Langlaufende Aufgabe pro Thread

121



125

SIEMENS

Agenda im Rückblick

Teil I: Virtual Threads

1 Ziel: Thread-per-Request Programmiermodell skalierbar machen

2 Konzept und Architektur: Implizite Freigabe des Carrier-Threads bei Blocking-I/O

3 Anwendung: API und Programmiermodell

4 Verhalten: Was passiert bei einem blockierenden Aufruf?

5 Performance und Skalierbarkeit: Thread-per-Request Server

TEIL II: Sprachfeatures zur Unterstützung von Virtual Threads

6 Structured Concurrency: Lösung der Struktur-Probleme von Threads

7 Scoped Values: Datenzuweisung an Threads

Teil III: Architekturen auf Basis von Virtual Threads

8 Geeignete Architekturen: Systeme, die von Virtual Threads profitieren

9 Ungeeignete Architekturen: Systeme, die nicht von Virtual Threads profitieren

10 Entscheidungskriterien für Migration zu Virtual Threads

Page 127

127

SIEMENS

Agenda im Rückblick

Teil I: Virtual Threads

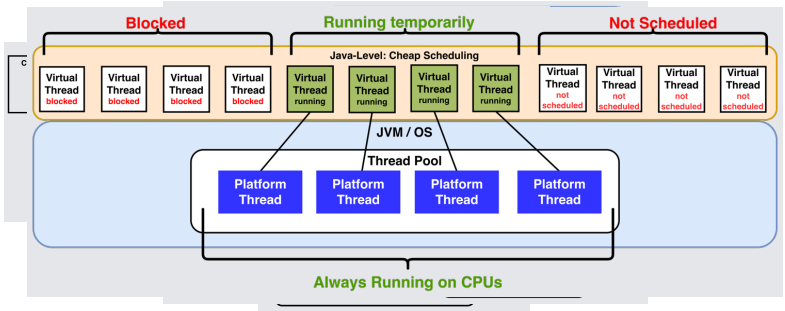
1 Ziel: Thread-per-Request Programmiermodell skalierbar machen

2 Konzept und Architektur: Implizite Freigabe des Carrier-Threads bei Blocking-I/O

3 Anwendung: API und Programmiermodell

4 Verhalten: Was passiert bei einem blockierenden Aufruf?

5 Performance und Skalierbarkeit: Thread-per-Request Server



Page 128

128

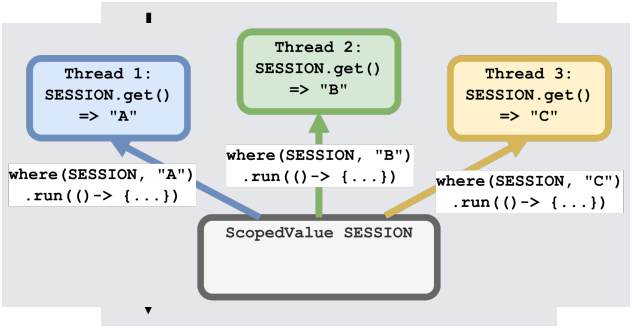
SIEMENS

Agenda im Rückblick

TEIL II: Sprachfeatures zur Unterstützung von Virtual Threads

6 Structured Concurrency: Lösung der Struktur-Probleme von Threads

7 Scoped Values: Datenzuweisung an Threads



Page 129

129

SIEMENS

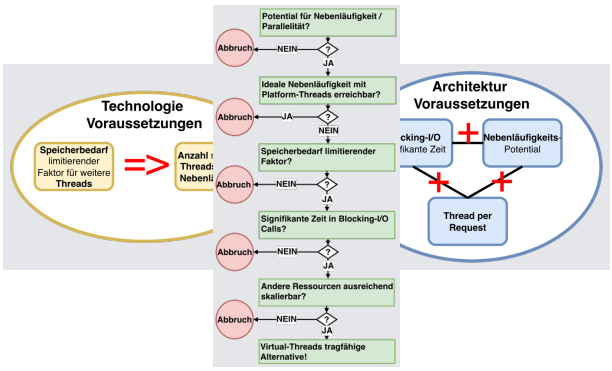
Agenda im Rückblick

Teil III: Architekturen auf Basis von Virtual Threads

8 Geeignete Architekturen: Systeme, die von Virtual Threads profitieren

9 Ungeeignete Architekturen: Systeme, die nicht von Virtual Threads profitieren

10 Entscheidungskriterien für Migration zu Virtual Threads



Page 130

130

SIEMENS

Marwan Abu-Khalil

Senior Software Architect
Siemens AG, Berlin

Kontakt
seminar@abu-khalil.de

www.linkedin.com/in/marwan-abu-khalil

Vorträge auf der W-JAX, November 2026

- GPU und Java, eine aussichtsreiche Liaison?
- Virtual Threads: Architekturoptionen durch Skalierbarkeit in neuer Dimension

Seminare

- Parallele Programmierung in Java
- Virtual Threads in Java
- Software Architektur

heise academy

GEDOPLAN

TAE

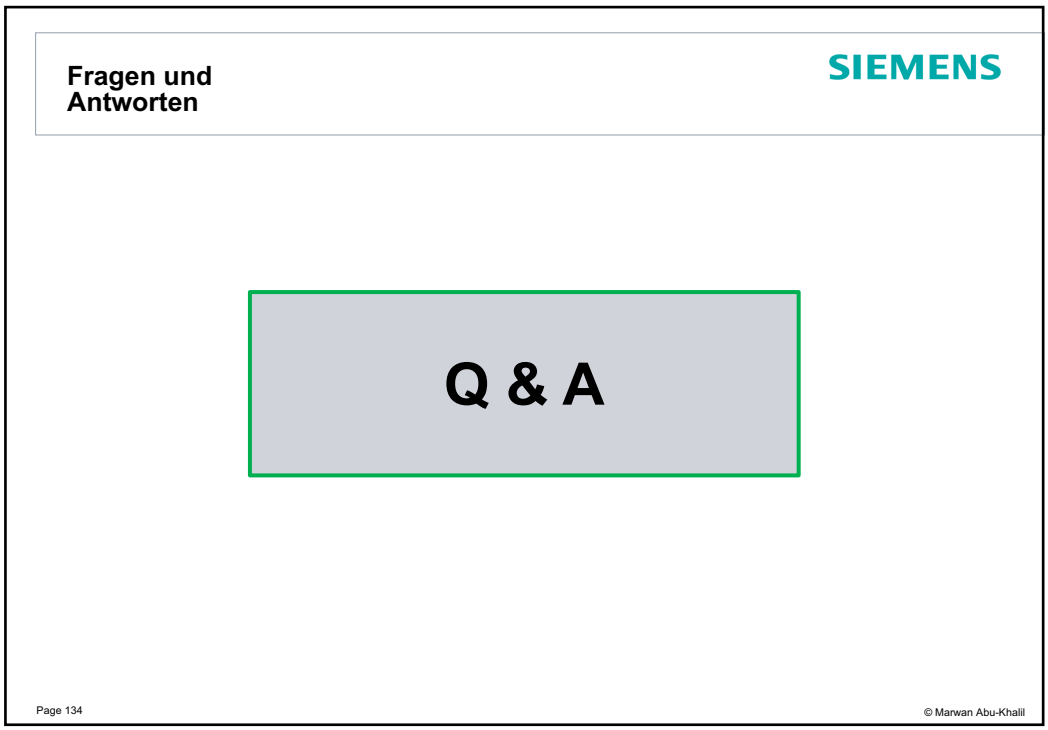
131

SIEMENS

Literatur und Weblinks

- **Brian Goetz**, 2022: New Foundations for Java High Scale Applications https://www.infoq.com/articles/java-Virtual Threads/
- **JEP Virtual Threads** https://openjdk.org/jeps/425 and https://openjdk.org/jeps/444
- **JEP Structured Concurrency**: JEP 525 (Java 26, preview) https://openjdk.org/jeps/525
- **JEP 506: Scoped Values**, https://openjdk.org/jeps/506
- **Lutz Hühnken**, entwickler.de 2020 https://entwickler.de/java/blick-in-die-fernere-zukunft
- **Marwan Abu-Khalil**, 2025, Virtual Threads in Java: Performance und Skalierbarkeit in neuer Dimension, https://www.sigs.de/artikel/Virtual Threads-in-java-performance-und-skalierbarkeit-in-neuer-dimension/
- **Michael Inden**, 2025: Java 25 LTS: Modernes Java leicht gemacht https://www.amazon.de/Java-25-LTS-Modernes-gemacht/dp/B0FRYWGDJ7
- **Nicolai Parlog**, Oracle, 2022: https://blogs.oracle.com/javamagazine/post/java-loom-Virtual Threads-platform-threads
- **Oracle** Java Core Libraries Documentation: https://docs.oracle.com/en/java/javase/21/core/Virtual Threads.html
- **Reinald Menge-Sonntag**, heise online 2022: https://www.heise.de/news/Java-19-verbessert-die-Nebenlaeufigkeit-mit-virtuellen-Threads-aus-Project-Loom-7269453.html
- **Ricardo Cardin**, 2023, good Examples about Threadpool and Scheduling: https://blog.rockthejvm.com/ultimate-guide-to-java-Virtual Threads/#3-how-to-create-a-Virtual Thread
- **Ron Presler**, QCon 2019: https://www.infoq.com/presentations/continuations-java/
- **Sven Woltmann**, 2022, nice little examples: https://www.happycoders.eu/de/java/Virtual Threads/
- **Thomasz Nurkiewitz**, QCon 2022: https://www.infoq.com/presentations/loom-java-concurrency/

133



134