Hochschule
Bremerhaven

Fachbereich II
Management und Informationssysteme
Informatik – Vertrauenswürdige Systeme M.Sc.

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science

Nebenläufigkeitsparadigmen und ihre Auswirkungen auf Laufzeiteigenschaften

Eine vergleichende Analyse von asynchroner Programmierung, Thread-per-Request- und User-Space-Thread-Modellen

---

Vorgelegt von: Julia Kaeten
Fabian Lignitz

Vorgelegt am: 05. Juli 2026

Erstgutachter: Prof. Dr.-Ing. Oliver Radfelder
Zweitgutachterin: Prof. Dr.-Ing. Karin Vosseberg

Abstract

Moderne Serveranwendungen müssen zehntausende gleichzeitige Verbindungen bedienen, von denen ein großer Teil überwiegend auf externe Systeme wartet. Wie eine Laufzeitumgebung diese Nebenläufigkeit repräsentiert und auf das Betriebssystem sowie die darunterliegende Hardware abbildet, entscheidet damit unmittelbar über Skalierbarkeit, Ressourcenverbrauch und Antwortverhalten. Zwei Antworten auf dieses Skalierungsproblem stehen sich gegenüber. Das ereignisgetriebene Modell verlangt eine explizit asynchrone Strukturierung des Programmcodes, während das User-Space-Scheduling die Verwaltung der Wartesituationen in die Laufzeitumgebung verlagert und das klassische blockierende Programmiermodell erhält.

Die vorliegende Arbeit untersucht, ob folgende Modelle ihr Versprechen auf der Ebene messbarer Systemeffekte halten können. Verglichen werden vier Laufzeitsysteme, die drei Modellklassen abdecken, nämlich das Thread-per-Request-Modell der Java-Plattform-Threads, das ereignisgetriebene Modell von Node.js sowie das M:N-Scheduling der Go-Runtime und der Java Virtual Threads. Der Fokus liegt bewusst nicht auf Durchsatz und Latenz auf Anwendungsebene, sondern auf den zugrundeliegenden Mechanismen wie Kontextwechseln, Thread-Migrationen zwischen Kernen, Cache-Miss-Raten und Scheduling-Overhead. Für die Untersuchung wurde eine reproduzierbare Messumgebung auf einem dedizierten Zwei-Sockel-Server mit NUMA-Architektur aufgebaut. Vier synthetische Workloads bilden eine rechengebundene, eine dateibasierte, eine gemischte und eine netzwerkbasierte Last ab. Die Erhebung kombiniert Hardware- und Betriebssystemzähler der `perf`-Familie mit prozessinternen Kennzahlen aus dem `/proc`-Dateisystem, sodass jede Kennzahl nach Möglichkeit aus mehreren Quellen bestätigt wird.

Die Ergebnisse zeigen deutliche strukturelle Unterschiede. Die Plattform-Threads erzeugen unter hoher Nebenläufigkeit die mit Abstand höchsten Kosten, sichtbar an den meisten Kontextwechseln, der höchsten Scheduling-Verzögerung, den stärksten Belastungen von Cache und TLB sowie dem größten Speicherbedarf. Go und die Virtual Threads halten die Zahl der Betriebssystem-Threads dagegen nahezu konstant, wahren dadurch die Cache-Lokalität und erreichen einen deutlich höheren Durchsatz. Node.js nimmt eine Sonderstellung ein. Auf einen einzelnen Thread bezogen ist es bei der netzwerkbasierten Last das wirksamste Modell, während seine fehlende Parallelität den Durchsatz bei rechengebundenen und gemischten Lasten begrenzt.

Darüber hinaus hat sich gezeigt, dass die beobachteten Leistungsunterschiede weniger aus der reinen Zahl der Kontextwechsel als aus deren Wirkung auf die Speicherhierarchie folgen. Das User-Space-Scheduling erreicht die Leistung des ereignisgetriebenen Modells nicht durch das Einsparen der Kontextwechsel selbst, sondern durch den Erhalt der Cache-Lokalität. Bei der gemischten Last erreichen die Virtual Threads sogar den höchsten Durchsatz aller Modelle und übertreffen damit das ereignisgetriebene Modell. Für die Praxis folgt daraus eine nach der Ziellast differenzierte Empfehlung. Für I/O gebundene und gemischte Lasten bieten Virtual Threads und Goroutinen die günstigste Kombination aus hohem Durchsatz, sparsamer Cache-Nutzung und unverändertem Programmiermodell, für rein netzwerkgebundene Lasten bleibt das ereignisgetriebene Modell je Thread am effizientesten und Go erweist sich über alle Szenarien hinweg als die ausgewogenste Wahl. Damit belegt die Arbeit, dass eine systemnahe Betrachtung Ursachen sichtbar macht, die eine reine Analyse von Durchsatz und Latenz verborgen ließe.

Inhaltsverzeichnis

1 Einleitung

Die vorliegende Arbeit untersucht verschiedene Nebenläufigkeitsmodelle moderner Laufzeitumgebungen. Ausgangspunkt der Untersuchung ist die zunehmende Bedeutung hochskalierender Serveranwendungen und die damit verbundene Frage, wie sich unterschiedliche Modelle der Nebenläufigkeit auf die Effizienz moderner Softwaresysteme auswirken. Die Einleitung führt zunächst in die Problemstellung und Motivation der Arbeit ein, formuliert die Zielsetzung und Forschungsfragen, leitet daraus die zu untersuchenden Hypothesen ab und beschreibt abschließend den Aufbau der Arbeit.

1.1 Relevanz und Motivation

Die Verlagerung weiter Teile moderner Softwaresysteme in netzwerkbasierte, verteilte Architekturen hat die Anforderungen an Nebenläufigkeit deutlich verändert. Serveranwendungen müssen heute Zehntausende bis Millionen gleichzeitig bestehender Verbindungen bedienen, wobei jeder einzelne Aufruf einen erheblichen Anteil seiner Lebensdauer blockierend auf Ergebnisse externer Systeme, typischerweise Datenbanken oder weitere Microservices, wartet [1, 2]. Die Frage, wie diese Nebenläufigkeit in einer Laufzeitumgebung repräsentiert und auf die zugrundeliegende Hardware abgebildet wird, entscheidet damit unmittelbar über Skalierbarkeit, Ressourcenverbrauch und Antwortverhalten solcher Systeme.
Das historisch prägende Programmiermodell für Serveranwendungen ist das Thread-per-Request-Modell, in dem jeder eingehende Aufruf einem dedizierten Betriebssystem-Thread zugewiesen wird [1, 2]. Mit wachsender Verbindungszahl stößt dieses Modell an strukturelle Grenzen, die zum Ende der 1990er Jahre unter dem Begriff des C10k-Problems systematisch beschrieben und damit zum Wendepunkt in der Bewertung serverseitiger Nebenläufigkeit wurden [3]. Eine detaillierte Auseinandersetzung mit den Mechanismen und Folgekosten dieses Modells findet sich in Kapitel 2.1.

Als Reaktion auf diese Grenze haben sich zwei alternative Modellklassen etabliert, die in dieser Arbeit verglichen werden. Die ereignisgetriebene Architektur, wie sie nginx und Node.js verfolgen, fordert von der Anwendungsentwicklung eine explizite Strukturierung von Wartesituationen im Programmcode [4]. Das User-Space-Scheduling, wie es Go mit Goroutinen und Java mit den in JDK Enhancement Proposal (JEP) 444 und 491 finalisierten Virtual Threads anbieten, verlagert die Verwaltung dieser Wartesituationen in die Laufzeitumgebung und erhält damit das klassische blockierende Programmiermodell [1, 2]. Beide Wege adressieren das gleiche Skalierungsproblem, stellen aber grundlegend unterschiedliche Anforderungen an die Entwickler:innen.

Aus diesen unterschiedlichen Anforderungen ergibt sich der zentrale Konflikt der Arbeit. Die ereignisgetriebene Architektur skaliert gut, verlangt jedoch eine Umstrukturierung der Anwendungslogik, die Lesbarkeit und Wartbarkeit messbar verschlechtert [1, 5]. Das User-Space-

Scheduling verspricht hingegen die Skalierbarkeit der ereignisgetriebenen Modelle bei unverändertem Programmiermodell [1, 2]. Ob dieses Versprechen sich auch auf der Ebene messbarer Systemeffekte erfüllt, ist die übergeordnete Leitfrage dieser Arbeit.

Die hier dargestellte Entwicklung beschränkt sich nicht auf klassische Webanwendungen, sondern prägt jeden Bereich, in der zahlreiche unabhängige Ereignisströme unter engen Latenzanforderungen verarbeitet werden müssen. Im Hochfrequenzhandel entscheiden bereits Differenzen im Mikrosekundenbereich über die Profitabilität einer Strategie [6], in der Medizintechnik können verspätete Antworten in der Echtzeitverarbeitung intraoperativer Sensordaten oder in der Steuerung chirurgischer Robotik unmittelbar die Patient:innensicherheit gefährden [7]. Wie eine Laufzeitumgebung Nebenläufigkeit auf das Betriebssystem und die darunterliegende Hardware abbildet, berührt damit Konstruktionsentscheidungen in einer Reihe regulierter und sicherheitsrelevanter Anwendungsgebiete. Diese Arbeit untersucht solche Bereiche nicht direkt, die Frage nach dem Einfluss des Nebenläufigkeitsmodells auf die Hardware ist dort aber genauso relevant.

Vor diesem Hintergrund stellt sich eine Frage, die in der Praxisliteratur bislang nur unzureichend systematisch beantwortet wird. Inwiefern sind explizite asynchrone, ereignisgetriebene Programmiermodelle weiterhin notwendig, wenn moderne Laufzeitumgebungen die zugrundeliegende Komplexität durch User-Space-Scheduling vollständig in den Hintergrund verlagern können? Bestehende Vergleichsstudien zeigen zwar, dass User-Level-Threading-Ansätze unter geeigneten Bedingungen eine zu reinen Event-Driven-Systemen vergleichbare Leistung erreichen können, sie wurden jedoch mit speziellen Forschungslaufzeitsystemen erhoben, nicht mit den breit eingesetzten Produktionssystemen Go-Runtime und Java Virtual Machine (JVM) mit Virtual Threads und fokussieren primär auf Durchsatz- und Latenzkennzahlen auf Anwendungsebene [2]. Aus einer betriebssystem- und systemnahen Perspektive bleibt damit ungeklärt, wie sich die untersuchten Modelle in Bezug auf Kontextwechsel, Thread-Migrationen zwischen CPU-Kernen, Cache-Miss-Raten und den daraus resultierenden Scheduling-Overhead unterscheiden. Diese Ebene ist nicht nebensächlich, da Cache-Lokalität, Non-Uniform Memory Access (NUMA)-Effekte und die Häufigkeit von Übergängen zwischen User Space und Kernel Space auf moderner Hardware wesentliche Teile der tatsächlichen Laufzeit bestimmen und die Effekte des Programmiermodells auf Anwendungsebene dabei dominieren können [8]. Eine methodische Bewertung dieser Effekte verlangt einen Vergleich auf Basis kontrollierter Workloads, geeigneter Hardware-Performance-Counter und einer reproduzierbaren Messumgebung [9]. Die vorliegende Arbeit nimmt diese Lücke zum Ausgangspunkt.

1.2 Zielsetzung und Forschungsfragen

Ziel der vorliegenden Arbeit ist eine systemnahe, empirisch unterlegte Analyse und ein Vergleich verschiedener Nebenläufigkeitsmodelle hinsichtlich ihres Einflusses auf Betriebssystem- und systemnahe Aspekte. Im Mittelpunkt stehen dabei nicht primär Durchsatz- oder Latenzwerte auf Anwendungsebene, sondern die zugrundeliegenden Mechanismen wie Kontextwechsel, Thread-Migrationen zwischen Kernen, Cache-Miss-Raten und Scheduling-Overhead.
Untersucht werden vier Laufzeitsysteme, die drei grundlegende Modellklassen abdecken. Das klassische Thread-per-Request-Modell auf Basis von Java-Plattform-Threads (1:1-Mapping zwischen Anwendungs- und Kernel-Thread), das ereignisgetriebene Modell von Node.js (Single-Threaded Event Loop über `libuv`) sowie das M:N-User-Space-Scheduling, das mit Goroutinen (Go-Runtime, G-M-P-Modell) und Java Virtual Threads (JVM, kontinuationsbasiert auf einem Carrier-Thread-Pool) durch zwei unabhängig entstandene und sich in den Implementierungsdetails unterscheidende Vertreter abgedeckt wird. [1, 2, 4, 10].

Die Arbeit adressiert folgende Forschungsfragen:

F1. Welche systemnahen Kosten entstehen durch klassische Betriebssystem-Threads im Vergleich zu asynchronen und virtualisierten Nebenläufigkeitsmodellen? Im Fokus stehen dabei Unterschiede in der Anzahl der Kontextwechsel, der Häufigkeit von CPU-Migrationen und den Cache-Miss-Raten unter identischen Workloads.

F2. In welchem Maße reduzieren ereignisgetriebene Architekturen Kontextwechsel und Cache-Invalidierungen im Vergleich zu thread-basierten Modellen und unter welchen Bedingungen kehrt sich dieser strukturelle Unterschied in einen Nachteil um?

F3. Wie effektiv abstrahieren Goroutinen und Java Virtual Threads blockierendes Input/Output (I/O) gegenüber den Anwendungsprogrammierer:innen und welche verbleibenden Kosten sind auf Systemebene messbar, etwa durch das Pinning virtueller Threads an ihren Carrier-Thread, durch Stack-Verwaltung oder durch die Sättigung des darunterliegenden Pools an Kernel-Threads?

Die Arbeit ist hinsichtlich ihres Untersuchungsgegenstands bewusst eng abgegrenzt. Nicht Bestandteil sind die Bewertung von Benutzerfreundlichkeit oder subjektiver Programmierergonomie, ein Vergleich vollständiger Web-Frameworks oder Anwendungen, die Analyse verteilter Systeme und Netzwerkprotokolle sowie Aspekte des Energieverbrauchs. Der Fokus liegt auf einer einzelnen Maschine und der Interaktion zwischen Laufzeitumgebung, Betriebssystem und CPU-Architektur. Diese Eingrenzung folgt der Argumentation von Gregg, wonach systemnahe Performanceuntersuchungen nur dann belastbare Aussagen liefern, wenn sie die Anzahl der gleichzeitig variierenden Faktoren konsequent klein halten [9].

1.3 Hypothesen und erwartete Ergebnisse

Aus den dargestellten Vorarbeiten und den charakteristischen Eigenschaften der vier untersuchten Laufzeitsysteme lassen sich vier Arbeitshypothesen formulieren. Jede Hypothese benennt eine erwartete strukturelle Differenz zwischen den Modellen sowie deren theoretische Grundlage und ist so formuliert, dass sie in der empirischen Evaluation falsifizierbar ist.

H1: Speicherbedarf pro Ausführungseinheit Die untersuchten Modelle unterscheiden sich systematisch im Speicherbedarf pro Ausführungseinheit. Virtual Threads und Goroutinen, die ihren Stack dynamisch und bedarfsorientiert auf dem verwalteten Speicher des Laufzeitsystems halten, weisen einen signifikant geringeren Speicherbedarf pro Ausführungseinheit auf als klassische Plattform-Threads, deren Stack-Bereiche ca. ein Megabyte (JVM) bzw. acht Megabyte (Linux), vorab reservieren [1, 11]. Die Grundlage dieser Hypothese bilden zum einen die Angaben der OpenJDK-Spezifikation zu Stack-Größen von Plattform-Threads [1], zum anderen die empirischen Messungen von Veen und Vlijmincx, die für Skalierungen von eintausend bis hunderttausend Instanzen ein deutliches Auseinanderlaufen der Speicherbelegung zeigen [10].

H2: Skalierungsverhalten unter hoher Nebenläufigkeit Die untersuchten Modelle zeigen bei zunehmender Nebenläufigkeit qualitativ unterschiedliches Skalierungsverhalten. Das klassische Thread-per-Request-Modell bindet pro Aufgabe einen Betriebssystem-Thread und ist damit hinsichtlich der maximalen Anzahl gleichzeitig aktiver Aufgaben durch die Ressourcen des Betriebssystems begrenzt. M:N-basierte Modelle sowie das Event-Loop-Modell vermeiden diese harte Schranke, indem sie viele Ausführungseinheiten auf wenige Kernel-Threads multiplexen bzw. rein Single Threaded arbeiten. Es ist zu erwarten, dass sich dieser strukturelle Unterschied ab einer Größenordnung von etwa zehntausend gleichzeitig aktiven Aufgaben in messbaren Abweichungen bei Durchsatz und Antwortzeitverteilung niederschlägt. Die Grundlage dieser Hypothese sind die empirischen Ergebnisse von Karsten und Barghi, die für ein vergleichbares Setup zeigen, dass thread-basierte Server bei vielen tausend Verbindungen deutlich abweichendes Skalierungsverhalten zu User-Level- und Event-driven-Laufzeitsystemen aufweisen [2], sowie die in JEP 444 und 491 dokumentierten architektonischen Eigenschaften des Plattform-Thread-Modells [1, 12].

H3: Scheduling-Overhead Die untersuchten Modelle erzeugen strukturell unterschiedliche Mengen an Kontextwechseln und CPU-Migrationen. Da im Event-Loop-Modell Scheduling-Entscheidungen vollständig im User Space getroffen werden und ein einzelner Thread alle Callbacks sequentiell abarbeitet, ist eine deutlich geringere Anzahl kernel-vermittelter Kontextwechsel und CPU-Migrationen zu erwarten als bei thread-basierten Modellen [4]. Diese Differenz gilt jedoch nicht uneingeschränkt, denn bei CPU gebundenen Lasten wird die

Single-Threaded-Architektur der Event-Loop zum Engpass, da CPU intensive Operationen die Verarbeitung weiterer Ereignisse zurückhalten [4, 13]. Die Grundlage dieser Hypothese ist die Architektur von `libuv` sowie der dokumentierte Ablauf einer Event-Loop-Iteration in der Node.js-Spezifikation [4, 13].

H4: Abstraktionsleistung der Virtual Threads Virtual Threads erreichen unter I/O gebundenen und gemischten Lastszenarien eine Performance, die der des ereignisgetriebenen Event-Loop-Modells nahekommt, bei einem für Java Entwickler:innen nahezu unveränderten sequentiellen Programmiermodell. Diese Hypothese greift den zentralen Konflikt der Arbeit auf. Wenn User-Space-Scheduling das Blockieren des Betriebssystem-Threads verhindert, sollte explizite asynchrone Programmierung keinen messbaren Vorteil mehr bieten. Verbleibende Kostenquellen sind insbesondere das Pinning virtueller Threads an ihren Carrier-Thread bei Aufrufen innerhalb von `synchronized`-Blöcken oder nativem Code, ein Effekt, der in JDK Enhancement Proposal 444 explizit dokumentiert und in nachfolgenden Versionen sukzessive reduziert wurde [1, 12]. Die Grundlage dieser Hypothese bilden ebenfalls die Ergebnisse von Karsten und Barghi, die zeigen, dass User-Level-basierte Laufzeitsysteme unter geeigneten Bedingungen den Durchsatz event-driven-basierter Systeme erreichen können [2], sowie die Architektur von Project Loom, wie sie in JDK Enhancement Proposal 444 und JEP 491 spezifiziert ist [1, 12].

Aus der Bestätigung oder Widerlegung dieser Hypothesen erwartet die Arbeit eine differenzierte, systemnah belegte Einordnung der vier Laufzeitsysteme. Insbesondere soll deutlich werden, ob das Anwendungsversprechen virtueller Threads, die Vorteile asynchroner Ausführung bei einem sequentiellen Programmiermodell zu vereinen, erfüllt werden kann.

1.4 Aufbau der Arbeit

In der vorliegenden Arbeit werden zunächst die konzeptionellen und systemtechnischen Grundlagen geschaffen, die für das Verständnis der zu untersuchenden Laufzeitsysteme erforderlich sind. Dazu gehören insbesondere die historischen Entwicklungen rund um das C10k-Problem, die Mechanismen moderner Betriebssysteme zur Prozess- und Thread-Verwaltung sowie die Grundlagen von Scheduling, Kontextwechseln und der Trennung zwischen Kernel- und User Space. Darauf aufbauend werden die vier zu betrachteten Laufzeitsysteme systematisch eingeführt und hinsichtlich ihrer Architekturprinzipien sowie ihrer Zugehörigkeit zu den drei Modellklassen eingeordnet. Ergänzend werden hardwarenahe Aspekte wie CPU-Caches, Cache-Lokalität und NUMA-Architekturen betrachtet, da diese einen wesentlichen Einfluss auf das Laufzeitverhalten hochnebenläufiger Systeme besitzen.
Anschließend wird der methodische Aufbau der empirischen Untersuchung beschrieben. Dies umfasst die Spezifikation der Testumgebung, die Konzeption synthetischer Workloads für CPU gebundene, I/O gebundene, gemischte Lastszenarien und Netzwerklasten sowie die Auswahl

der eingesetzten Mess- und Analysewerkzeuge. Ein besonderer Fokus liegt dabei auf der Reproduzierbarkeit der Experimente und der kontrollierten Variation einzelner Einflussgrößen. Für die Erhebung hardwarenaher Leistungsdaten werden unter anderem Werkzeuge der `perf`-Familie [14] sowie laufzeitspezifische Analysetools eingesetzt.

Darauf aufbauend erfolgt die Durchführung der Experimente und die Erfassung der Messdaten. Die erhobenen Ergebnisse werden anschließend vergleichend ausgewertet und hinsichtlich der untersuchten Laufzeitsysteme sowie ihrer Zugehörigkeit zu den drei Modellklassen analysiert. Abschließend werden die Ergebnisse in Bezug auf die formulierten Forschungsfragen und Hypothesen diskutiert. Dabei werden sowohl die Aussagekraft als auch die Grenzen der Untersuchung betrachtet und Implikationen für den praktischen Einsatz der untersuchten Modelle in hochskalierenden Softwaresystemen abgeleitet. Die Arbeit endet mit einer Zusammenfassung der zentralen Erkenntnisse sowie einem Ausblick auf mögliche weiterführende Untersuchungen.

2 Theoretische Grundlagen

Dieses Kapitel legt die konzeptionellen und systemtechnischen Grundlagen, auf denen die spätere Untersuchung aufbaut. Abschnitt 2.1 beschreibt zunächst die historische Entwicklung von der sequentiellen Programmierung über nebenläufige Thread-Modelle bis hin zu ereignisgetriebenen Architekturen, wobei das C10k-Problem als Wendepunkt eingeordnet wird. Darauf aufbauend führt Abschnitt 2.2 die für die weitere Argumentation notwendigen Grundlagen der Prozess- und Thread-Verwaltung ein, so u. a. die Trennung zwischen Kernel und User Space, die Repräsentation von Betriebssystem-Threads und Kontextwechseln sowie das Scheduling im Kernel. Anschließend behandelt Abschnitt 2.3 die hardwarenahen Aspekte CPU-Caches, Cache-Lokalität und NUMA-Architekturen, da diese das Laufzeitverhalten hochnebenläufiger Systeme wesentlich beeinflussen. Abschnitt 2.4 grenzt darauf aufbauend blockierendes und nicht blockierendes I/O voneinander ab. Abschließend folgt in Abschnitt 2.5 eine Einführung der vier untersuchten Laufzeitsysteme, also der Java-Plattform-Threads, des ereignisgetriebenen Modells von Node.js, der Goroutinen in der Go-Runtime sowie der Virtual Threads im Rahmen von Project Loom.

2.1 Historischer Kontext: Das C10k-Problem

Die Frage, wie Software mit Asynchronität umgeht, ist im Bereich der serverseitigen Programmierung sehr relevant. Sobald ein Programm auf langsamere Komponenten warten muss, etwa auf Datenträger, Netzwerk oder andere Prozesse, stellt sich die Wahl zwischen zwei Grundhaltungen. Entweder verbirgt die Laufzeitumgebung diese Wartezeit hinter einem sequentiellen, blockierenden Programmiermodell, oder sie zwingt die Entwickler:innen dazu, Wartesituationen explizit als asynchrone Ereignisse abzubilden [5, 15]. Der folgende Abschnitt soll aufzeigen, wie die Informatik in mehreren Etappen mit diesem Problem umgegangen ist. Ausgangspunkt ist die rein sequentielle Programmierung der frühen Rechnergeneration, gefolgt von der Entwicklungsstufe Thread-per-Request-Modells der ersten Webserver Generation, bis hin zu dem von Dan Kegel im Jahr 1999 formulierte C10k-Problem, das die Grenzen dieses Modells sichtbar machte und ereignisgetriebene Architekturen zu einer Art Mainstream Lösung werden ließ.

Frühe Rechner kannten keine Nebenläufigkeit auf Anwendungsebene. Aufgaben wurden gesammelt eingereicht und vom Rechner in der Eingangsreihenfolge sequentiell abgearbeitet, wobei eine Aufgabe vollständig terminiert sein musste, bevor die nächste beginnen konnte. Diese Betriebsart wird als Stapelverarbeitung oder Batch-Betrieb bezeichnet [8]. Aus Sicht der Programmierung existierte in diesem Modell keine echte Parallelität, da der Rechner genau eine Aufgabe ausführte, bevor er die nächste annahm.
Mit der Einführung des Mehrprogrammbetriebs in den 1960er Jahren wurde diese strikt sequentielle Abarbeitung aufgelöst. Mehrere Programme konnten nun zeitlich verschränkt im Speicher gehalten und vom Betriebssystem so verwaltet werden, dass die CPU während der

I/O Wartezeiten eines Programms ein anderes weiterführen konnte. Damit trat Nebenläufigkeit erstmals als eigenständiges Phänomen auf der Ebene des Betriebssystems auf, allerdings noch nicht innerhalb einzelner Anwendungen [8].
Theoretisch erforscht wurde die Nebenläufigkeit zwischen Prozessen durch eine Reihe Arbeiten ab Mitte der 1960er Jahre. Dijkstra zeigte mit den Cooperating Sequential Processes, wie zwei voneinander unabhängige sequentielle Programme über Semaphore koordiniert werden können [16]. Hoare führte mit den Communicating Sequential Processes ein mathematisches Modell ein, in dem Nebenläufigkeit nicht durch geteilten Speicher, sondern durch synchrone Nachrichtenkommunikation zwischen Prozessen beschrieben wird [17]. Diese theoretischen Grundlagen prägen die Sprachgestaltung von Nebenläufigkeitskonstrukten bis heute und finden sich bspw. in den Channels und Goroutinen der Programmiersprache Go wieder [8].

Auf der Ebene der einzelnen Anwendung wurde Nebenläufigkeit erst mit der Einführung von Threads als leichtgewichtige Ausführungseinheiten innerhalb eines Prozesses zu einem allgemein verfügbaren Werkzeug. Ein Thread besteht aus einem eigenen Registersatz und einem eigenen Stack. Den Adressraum, die geöffneten Dateideskriptoren und weitere Prozessressourcen teilt er sich mit den übrigen Threads desselben Prozesses, was Erzeugung und Wechsel im Vergleich zu vollwertigen Prozessen deutlich günstiger macht. [8]. Aus dieser Bauweise leitet sich die Vorstellung ab, jede unabhängige Aufgabe einer Anwendung in einen eigenen Thread auszulagern und so Nebenläufigkeit direkt im Programmiermodell abzubilden.

Mit der Verbreitung des World Wide Web in den 1990er Jahren wurde diese Idee zur vorherrschenden Architekturentscheidung für Serveranwendungen. Der bis heute weit verbreitete Apache Server implementierte ein Prozess- bzw. Thread-per-Connection-Modell, in dem jeder eingehenden Verbindung ein eigener Prozess oder Thread zugeordnet wird, der die Anfrage von der Annahme bis zur Antwort sequentiell und blockierend abarbeitet [18, 19, 20]. Das Problem dieses Modells zeigt sich, sobald die Anzahl gleichzeitiger Verbindungen die Anzahl der CPU-Kerne deutlich überschreitet. Jeder zusätzliche Thread bindet Ressourcen des Betriebssystems, konkret einen vorab reservierten Stack-Bereich in der Größenordnung von mehreren Megabyte sowie Verwaltungsstrukturen im Kernel [8, 21].
Bereits 1996 formulierte Ousterhout die These, dass Threads für die meisten Anwendungszwecke ein schlechtes Werkzeug seien, da Synchronisation, Race Conditions und Deadlocks die kognitive Last für die Programmierung unangemessen erhöhten und das Modell auf vielen gleichzeitigen Verbindungen nicht sinnvoll skaliert [15]. Diese Aussage wurde durch empirische Beobachtungen an den Webservern jener Zeit gestützt.

Im Jahr 1999 fasste dann Dan Kegel diese Beobachtungen unter dem Begriff des C10k-Problems in einem Blogeintrag zusammen. Der Begriff bezeichnet die Aufgabe, auf einem einzelnen Server Zehntausend gleichzeitige Client-Verbindungen zu bedienen, ohne dass die Anwendung am Verwaltungsaufwand des Betriebssystems scheitert. Kegel verband seine Problemstellung mit einer praktischen Beobachtung. Der FTP-Server cdrom.com bediente im selben Jahr bereits zehntausend gleichzeitige Verbindungen über eine Gigabit Ethernet Leitung, was zeigte, dass die Hardware prinzipiell ausreichte, das traditionelle Programmiermodell jedoch nicht. Seine

Analyse listete eine Reihe von Techniken auf, mit denen sich diese Skalierungsgrenze überwinden lassen könne, darunter nicht blockierende Sockets, effiziente Ereignisbenachrichtigung über `epoll` unter Linux, `kqueue` unter BSD und `IOCP` unter Windows sowie asynchrones I/O auf Kernel-Ebene [3]. Mit dem fortschreitenden Hardwareausbau wurde das Problem ab 2013 als C10M-Problem auf mehrere Millionen gleichzeitige Verbindungen neu formuliert [2, 22]. Die zugrundeliegende Argumentation gegen das klassische Thread-Modell blieb dabei im Prinzip unverändert [2]. Das C10k-Problem markiert damit weniger ein einzelnes technisches Hindernis als vielmehr einen Paradigmenwechsel in der Bewertung von Nebenläufigkeitsmodellen. Sichtbar wurde, dass die 1:1 Abbildung zwischen Anwendungslogik und Kernel-Thread bei wachsender Verbindungsanzahl nicht sauber skaliert und ein neues Modell benötigt wird.

Dieses Modell wurde unabhängig von mehreren Autoren entwickelt. Auf akademischer Seite formulierten Welsh, Culler und Brewer im Jahr 2001 mit der Staged Event-Driven Architecture (SEDA) einen Vorschlag, in dem Anwendungen aus einem Netz ereignisgetriebener Stufen bestehen, die über explizite Warteschlangen verbunden sind. Jede Stufe verfügt über einen dynamisch verwalteten Pool weniger Threads, was die Konstruktion hochnebenläufiger Internetdienste ohne die Synchronisationsproblematik klassischer thread-basierter Modelle erlauben soll [23]. Auf praktischer Seite griff Igor Sysoev mit dem 2004 veröffentlichten Webserver nginx das C10k-Problem unmittelbar auf und implementierte eine ereignisgetriebene Architektur, in der einzelne Worker-Prozesse (typisch einer pro CPU-Kern) über eine Event Loop tausende gleichzeitige Verbindungen verwaltet [19]. Diese Architektur löste das Engpassproblem des prozess- und thread-basierten Apache Modells und bestimmt bis heute die Architektur moderner Reverse Proxies und Load Balancer.

Die Verlagerung des ereignisgetriebenen Modells in die Anwendungsentwicklung zeigte sich 2009 mit der Veröffentlichung von Node.js. Ryan Dahl präsentierte auf der JSConf EU eine an JavaScript gebundene Laufzeitumgebung, die alle elementaren I/O-Operationen konsequent ereignisgetrieben gestaltet und heute auf der Bibliothek `libuv` aufsetzt [4]. Wartesituationen werden in dieser Architektur nicht durch blockierte Threads, sondern durch in Datenstrukturen verwaltete Callbacks repräsentiert, was eine sehr hohe Verbindungsdichte bei vergleichsweise geringem Speicherverbrauch ermöglicht [4, 13]. Mit Node.js etablierte sich das ereignisgetriebene Modell der serverseitigen Anwendungsentwicklung und diente als Antwort auf das C10k-Problem. Mit der Verbreitung ereignisgetriebener Modelle entstanden jedoch neue Herausforderungen gerade für Entwickler:innen. Sequentiell formulierte Logik muss in eine Folge von Callbacks zerlegt werden, was den Kontrollfluss undurchsichtig macht und Fehlerbehandlung, Stack-Traces sowie Debugging deutlich erschwert [1]. In der Praxis hat sich dafür die Bezeichnung *callback hell* etabliert, die die tief verschachtelten Rückruffunktionen beschreibt, die in komplexeren ereignisgetriebenen Programmen entstehen [2].

Als Reaktion hierauf schrieb bspw. Bob Nystrom den 2015 veröffentlichten Aufsatz *What Color Is Your Function*. Nystrom entwarf darin eine fiktive Programmiersprache, in der jede Funktion eine von zwei Farben trägt und in der synchrone (blaue) Funktionen keine asynchronen (rote) aufrufen dürfen, die Umkehrung jedoch erlaubt ist. Der Aufsatz macht deutlich, wie

sehr diese Trennung den Umgang mit Synchronität und Asynchronität in vielen modernen Programmiersprachen prägt. Jede explizite Asynchronität färbt die aufrufenden Schichten einer Anwendung mit ihrer Farbe ein und breitet sich auf diesem Wege bis in tiefe Teile der Codebasis aus [5].

Eine alternative Antwort auf das C10k-Problem verfolgte parallel zu Node.js die im Jahr 2009 bei Google von Griesemer, Pike und Thompson veröffentlichte Sprache Go [24]. Anstatt die Asynchronität an die Anwendungsentwicklung zurückzugeben, verlagert Go diese in die Laufzeitumgebung. Eine Goroutine ist eine leichtgewichtige Ausführungseinheit, die wie eine gewöhnliche Funktion aufgerufen wird, zur Laufzeit jedoch von einem Scheduler der Sprachimplementierung auf eine kleinere Menge von Betriebssystem-Threads gemultiplext wird. Dieses Muster wird als M:N-Scheduling bezeichnet. Blockiert eine Goroutine auf I/O, erkennt die Laufzeitumgebung diesen Zustand und koppelt die Goroutine vom tragenden Kernel-Thread ab, ohne diesen selbst zu blockieren, sodass der Thread weitere Goroutinen ausführen kann [2]. Aus Sicht der Entwickler:innen bleibt der Code sequentiell und blockierend, während die Laufzeitumgebung intern auf nicht blockierende Systemschnittstellen zurückgreift. Damit verspricht Go, das einfache Programmiermodell der Thread-per-Request Ära mit der Skalierbarkeit ereignisgetriebener Architekturen zu verbinden.

Die Java Virtual Machine vollzog diesen Schritt später mit Project Loom. Mit JDK Enhancement Proposal 444 wurden in Java 21 sog. Virtual Threads als finalisiertes Sprachmittel eingeführt. Diese sind weiterhin Instanzen der Klasse `java.lang.Thread`, jedoch nicht mehr dauerhaft an einen Betriebssystem-Thread gebunden. Die JVM hält einen kleinen Pool von Plattform-Threads, die als Carrier-Threads bezeichnet werden und multiplext virtuelle Threads über ein M:N Modell auf diese [1, 10]. Wenn ein virtueller Thread eine blockierende Operation ausführt, etwa einen Lesezugriff auf einen Socket, wird er von seinem Carrier abgehängt, sodass der Plattform-Thread unmittelbar für andere virtuelle Threads zur Verfügung steht [1]. Aus Sicht der Anwendungsentwicklung bleibt die Programmierung im klassischen Thread-per-Request Stil möglich, während die Skalierungsgrenze nicht mehr durch die Anzahl der Betriebssystem-Threads, sondern durch die verfügbaren Speicher- und CPU-Ressourcen bestimmt wird [1]. Empirische Messungen von Veen und Vlijmincx zeigen diesen Unterschied, indem sie die Speicherbelegung von Plattform- und Virtual Threads in der Größenordnung von tausend bis hunderttausend Instanzen gegenüberstellen [10].

Betrachtet man diesen historischen Kontext, dann zeigen sich zwei zunächst getrennte Entwicklungslinien. Auf der einen Seite steht Node.js mit einer einzelnen Ereignisschleife und einem explizit asynchronen Programmiermodell. Auf der anderen Seite stehen Go und die JVM mit Virtual Threads, die das blockierende Thread-per-Request Schema durch User-Space-Scheduling skalierbar machen. Beide Linien beanspruchen, das C10k-Problem zu lösen, unterscheiden sich aber grundlegend in der Art und Weise, wie sie Wartesituationen repräsentieren und auf die zugrundeliegende Hardware abbilden.

2.2 Grundlagen der Prozess- und Thread-Verwaltung

Die im vorangegangenen Abschnitt geschilderte Entwicklung von der sequentiellen über die thread-basierte hin zur ereignisgetriebenen Nebenläufigkeit lässt sich nur dann untersuchen und beurteilen, wenn die zugrundeliegenden Mechanismen moderner Betriebssysteme zur Prozess- und Thread-Verwaltung verstanden worden sind. Diesem Verständnis widmen sich die folgenden Abschnitte. Kapitel 2.2.1 grenzt zunächst den Kernel Space vom User Space ab und beschreibt den Übergang zwischen beiden über Systemaufrufe. Abschnitt 2.2.2 behandelt anschließend die Repräsentation von Threads im Betriebssystem sowie die direkten und indirekten Kosten von Kontextwechseln. Abschnitt 2.2.3 schließt mit der Funktionsweise des Schedulers im Kernel, also der zentralen Instanz, die über die Zuteilung der CPU zwischen lauffähigen Threads entscheidet. Die dabei wiederholt angesprochenen hardwarenahen Strukturen wie CPU-Caches und NUMA-Architekturen werden anschließend in Abschnitt 2.3 vertieft, da sie das Laufzeitverhalten hochnebenläufiger Systeme wesentlich beeinflussen und die indirekten Kosten der hier eingeführten Mechanismen erst erklären.

2.2.1 Kernel- und User Space

Moderne Multiuser-Betriebssysteme trennen die Ausführung von Anwendungscode strikt von der Ausführung des Kernels. Diese Trennung dient zwei Zielen, dem Schutz des Systems vor fehlerhaftem oder bösartigem Anwendungscode und der Isolation der Prozesse untereinander, damit kein Prozess auf den Speicher oder die Betriebsmittel eines anderen Prozesses zugreifen kann [8, 21]. Realisiert wird die Trennung durch zwei voneinander unabhängige Mechanismen, einen hardwaregestützten Schutzring auf Prozessorebene und eine Aufteilung des virtuellen Adressraums auf Speicherebene [8].
Auf Prozessorebene definiert die Hardware verschiedene Privilegstufen, die jeweils festlegen, welche Maschinenbefehle ein Programm in diesem Zustand ausführen darf und auf welche Speicherbereiche es zugreifen darf. Die x86-64 Architektur, die auf den heute verbreiteten 64-Bit Systemen von Intel und AMD zum Einsatz kommt, stellt vier solcher Stufen bereit, die als Ring 0 bis Ring 3 bezeichnet werden (siehe Abbildung 2.1). Ring 0 entspricht dem Kernel Mode und gestattet uneingeschränkten Zugriff auf den gesamten Speicher sowie auf privilegierte Maschinenbefehle, etwa solche, die die Speicherverwaltungseinheit umkonfigurieren oder direkten Zugriff auf Hardwareregister erlauben. Ring 3 entspricht dem User Mode, in dem ein Prozess weder auf den Adressraum fremder Prozesse noch auf privilegierte Befehle Zugriff hat [8, 21]. Auf praxisrelevanten Linux Systemen werden ausschließlich diese beiden Stufen verwendet, sodass sich praktisch eine Zwei-Stufen-Architektur ergibt [8].

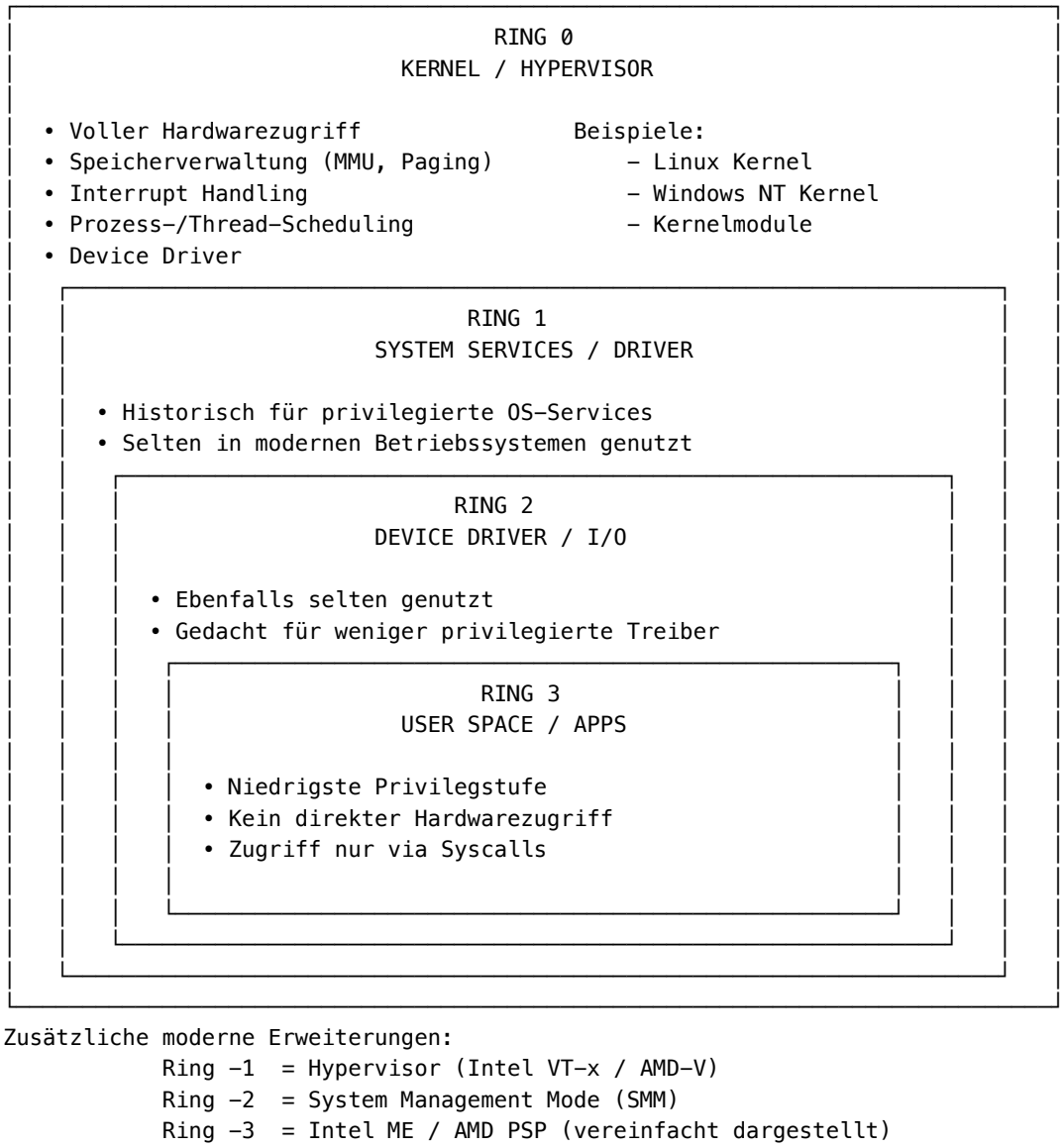

Abbildung 2.1: Schematische Darstellung der CPU-Ring Architektur

Auf Speicherebene wird der virtuelle Adressraum jedes Prozesses in zwei Bereiche aufgeteilt, den Kernel Space und den User Space. Unter Linux nimmt der Kernel Space den oberen Adressbereich ein, der User Space den unteren [8]. Ein Anwendungsprozess kann ausschließlich auf den User Space seines eigenen Adressraums zugreifen. Der Kernel Space ist zwar in jedem virtuellen Adressraum identisch eingeblendet, sodass der Kernel unabhängig vom aktiven Prozess immer denselben Code und dieselben Datenstrukturen vorfindet, bleibt aber nur im Kernel Mode erreichbar [8]. Versucht ein Anwendungsprozess auf eine Adresse im Kernel Space zuzugreifen, löst die Speicherverwaltungseinheit eine Schutzverletzung aus und der Prozess wird beendet.

Aus dieser strikten Trennung folgt, dass Anwendungen auf sämtliche Betriebsmittel des Rechners nur indirekt zugreifen können, also über den Kernel als vermittelnde Instanz. Der Kontrollpfad, über den eine Anwendung den Kernel anspricht, ist der Systemaufruf. Im Programmcode wirkt ein Systemaufruf wie ein gewöhnlicher Funktionsaufruf, intern muss jedoch ein architekturspezifischer Mechanismus ausgelöst werden, der den Prozessor in den Kernel Mode versetzt [8, 21].

Historisch wurde dieser Übergang auf 32-Bit Linux Systemen über einen Software Interrupt mit der Instruktion `int 0x80` realisiert. Diese Instruktion durchläuft die Interrupt Descriptor Table, um die Einstiegsadresse des Kernels zu finden und teilt damit denselben Hardwaremechanismus mit regulären Hardware-Interrupts [8, 21]. Mit der Einführung der x86-64 Architektur wurden die dedizierten Instruktionen `syscall` und `sysret` eingeführt, die den Umweg über die Interrupt Tabelle vermeiden. Die Einstiegsadresse des Kernels wird stattdessen aus einem dafür vorgesehenen Model Specific Register gelesen, was den Übergang deutlich beschleunigt [25]. Auf modernen x86-64 Linux Systemen ist dieser schnellere Pfad der Standard, während `int 0x80` aus Kompatibilitätsgründen weiterhin verfügbar bleibt. Systemaufrufe sind damit keine klassischen Interrupts im modernen Design, sondern kontrollierte Privileg Übergänge über einen dedizierten Maschinenbefehlspfad.

Unabhängig vom konkreten Mechanismus läuft der Übergang nach einem festen Muster ab (siehe Abbildung 2.2). Der Prozessor sichert die aktuelle Befehlsadresse sowie das Prozessorstatuswort, lädt das Statuswort des Kernels und springt an die hinterlegte Einstiegsadresse innerhalb des Kernel-Codes [8]. Erst nach diesem Übergang darf der Code des Kernels die angeforderte Operation ausführen, etwa das Lesen von einem Dateideskriptor, den Empfang eines Netzwerkpakets oder die Anforderung weiteren Speichers. Nach Abschluss kehrt die Kontrolle in den User Mode zurück, der Prozessor wechselt das Statuswort zurück und setzt die Ausführung der aufrufenden Anwendung an der gespeicherten Befehlsadresse fort. Dieser Übergang zwischen User Mode und Kernel Mode nennt man Mode Switch. Seine Kosten lassen sich in zwei Komponenten aufteilen. Unter den direkten Kosten eines Mode Switch werden ausschließlich die explizit vom Prozessor ausgeführten Operationen verstanden, also das Sichern und Wiederherstellen der Register, das Umschalten des Stack-Zeigers auf den Kernel-Stack des betroffenen Prozesses sowie das Setzen des Privilegbits [8]. Unter den indirekten Kosten werden hingegen alle nachgelagerten Effekte zusammengefasst, die sich nicht im Mode Switch selbst, sondern in der nachfolgenden Ausführung zu finden sind. Sie entstehen dadurch, dass der Kernel-Code während seiner Ausführung zentrale Prozessorstrukturen mit kernelbezogenem Zustand überschreibt, namentlich L1-, L2- und L3-Caches sowie den Translation Lookaside Buffer (TLB) und auf diese Weise die Arbeitsdaten der Anwendung aus diesen Strukturen löscht [25].

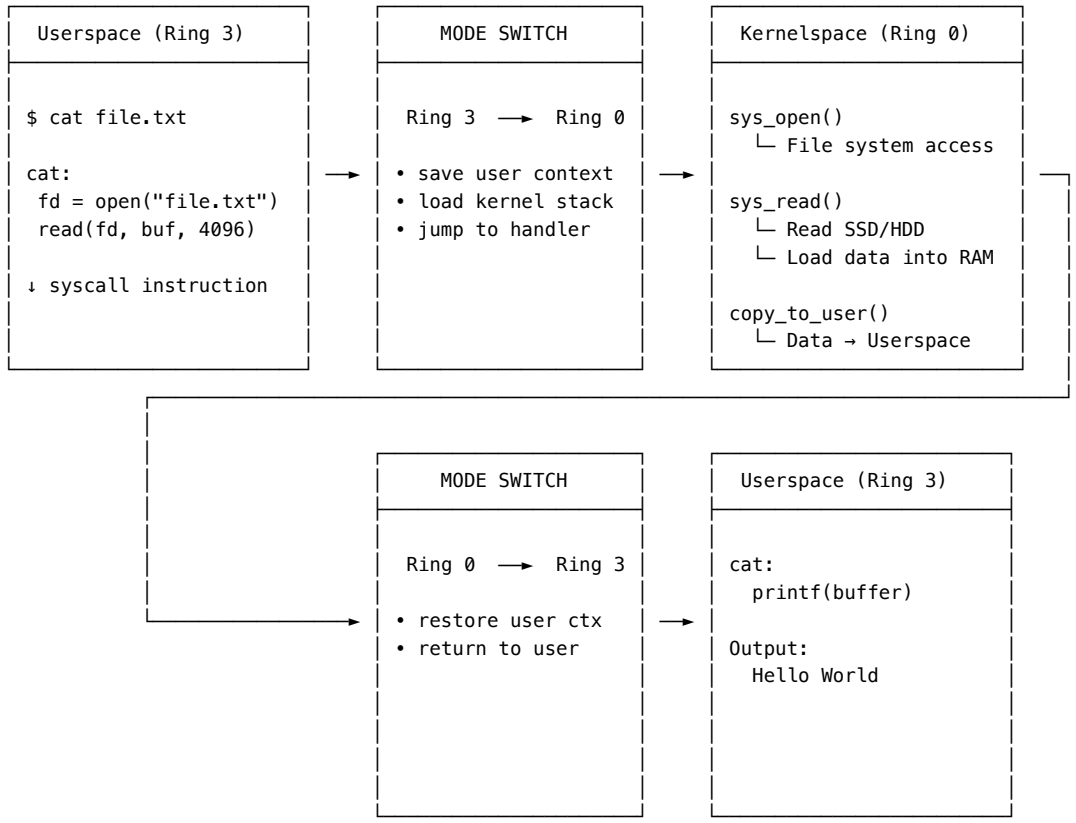

Abbildung 2.2: Vereinfachte Darstellung eines Mode Switch

Soares und Stumm zeigen diesen Effekt in ihrer Arbeit für verschiedene Systemaufrufe auf einem Intel-Core-i7-Prozessor und weisen dabei die Anzahl verdrängter Einträge getrennt für i-cache, d-cache, L2, L3 und d-Translation Lookaside Buffer aus. Sie stellen zusätzlich fest, dass der Effekt auf L2 und L3 größer ausfällt als auf L1, da höhere Cache-Ebenen intensiver prefetchen und dass der TLB durch die meisten Systemaufrufe signifikant belastet wird [25]. Für systemaufrufintensive Workloads wie Apache, MySQL und BIND können diese indirekten Effekte die direkten Kosten des Mode Switch deutlich übersteigen [25].

2.2.2 Betriebssystem-Threads und Kontextwechsel

Ein Prozess kapselt sämtliche Zustands- und Steuerinformationen eines laufenden Programms, insbesondere den virtuellen Adressraum, die geöffneten Dateideskriptoren, die Speicherreferenzen für Code-, Daten- und Stack-Segmente sowie die Konfiguration der Signalbehandlung [8, 21]. Ein Prozess ist damit das schwergewichtigste Konstrukt zur Strukturierung nebenläufiger Ausführung, da seine Erzeugung das Anlegen eines neuen Adressraums, einer neuen Seitentabelle und eines vollständigen Prozesskontrollblocks erfordert [8].

Aus diesem Umstand ergibt sich der Bedarf nach einer leichteren Ausführungseinheit innerhalb eines Prozesses. Ein Thread teilt sich mit den übrigen Threads desselben Prozesses den Adressraum, die offenen Dateideskriptoren und die übrigen Prozessressourcen. Eigene Zustandsinformationen besitzt ein Thread lediglich in Form eines eigenen Stacks, eines eigenen Registersatzes und einiger weniger zustandsbeschreibender Felder [8, 21]. Aus dieser Bauweise ergibt sich die Bezeichnung des Threads als Leichtgewichtsprozess. Erzeugung, Vernichtung und Wechsel zwischen Threads desselben Prozesses sind deutlich kostengünstiger als die entsprechenden Operationen auf Prozessebene, da weder ein neuer Adressraum angelegt noch eine neue Seitentabelle erzeugt werden muss [8].

Threads können grundsätzlich auf drei verschiedene Weisen realisiert werden, die sich darin unterscheiden, wie eine Menge von Anwendungsthreads auf die vom Kernel verwalteten Ausführungseinheiten abgebildet wird. Diese drei Varianten werden als N:1-, 1:1- und M:N-Mapping bezeichnet [2, 8, 21].
Beim N:1-Mapping liegt die gesamte Thread-Verwaltung in einer Anwendungsbibliothek und damit vollständig im User Space des Prozesses (siehe Abbildung 2.3). Alle N Anwendungsthreads laufen auf genau einem Kernel-Thread, sodass der Kernel den enthaltenden Prozess als eine einzelne Ausführungseinheit wahrnimmt und die innerhalb dieses Prozesses verwalteten Threads nicht kennt [8, 21].

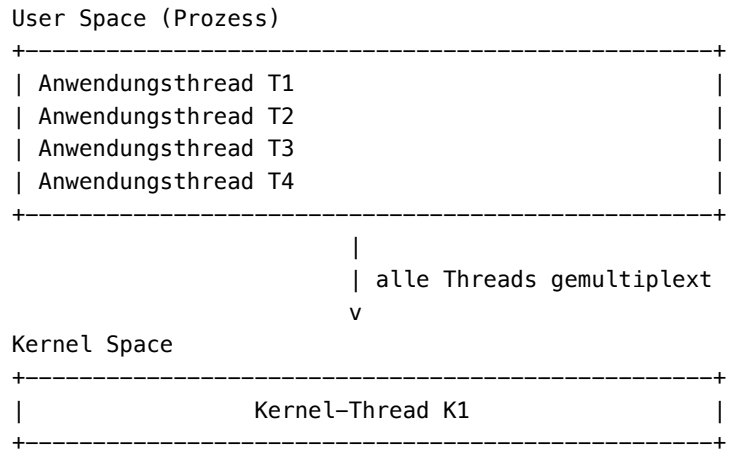

Abbildung 2.3: Darstellung N:1 Mapping

Aus dieser Bauweise ergeben sich sowohl Vor- als auch Nachteile. Die Umschaltung zwischen den Threads erfolgt vollständig im User Space, ohne Systemaufruf und ohne Mode Switch und ist damit sehr schnell. Zusätzlich entfällt der vom Kernel reservierte Verwaltungsaufwand pro Thread, sodass auch der Speicherbedarf gering bleibt [8]. Dem stehen jedoch zwei strukturelle Schwächen gegenüber. Da der Kernel die einzelnen Threads nicht sieht, blockiert ein einziger blockierender Systemaufruf, etwa ein Lesezugriff auf einen Socket ohne verfügbare Daten, den gesamten Prozess samt aller anderen Threads [8, 21]. Außerdem kann der Prozess auf einem Multi-Core-System nicht parallel auf mehreren CPU-Kernen ausgeführt werden, da der Kernel

ihm nur einen einzigen Kernel-Thread zur Ausführung zuteilt. Reine N:1-Implementierungen finden sich vor allem in historischen Thread-Bibliotheken und in modernen Systemen nur noch in Spezialfällen ohne blockierende I/O-Operationen [8].

Beim 1:1-Mapping wird jeder Anwendungsthread auf genau einen vom Kernel verwalteten Thread abgebildet (siehe Abbildung 2.4). Der Kernel kennt damit jeden einzelnen Thread, kann ihn unabhängig von den anderen blockieren und ihn auf einen beliebigen verfügbaren CPU-Kern legen [8, 21]. Diese Sichtbarkeit bedeutet für das Modell jedoch höhere Kosten je Umschaltung, da jeder Wechsel zwischen Threads über den Scheduler des Kernels und damit über einen Mode Switch erfolgt. Zusätzlich entsteht pro Thread ein vom Kernel verwalteter Verwaltungsaufwand inklusive eines vorab reservierten Stack-Bereichs [8, 21]. Linux setzt seit der Native POSIX Thread Library (NPTL) Implementierung konsequent auf dieses Modell. Die klassischen Java-Plattform-Threads sind ihrerseits ein dünner Wrapper um diese Kernel-Threads und folgen damit ebenfalls dem 1:1-Mapping [1, 2].

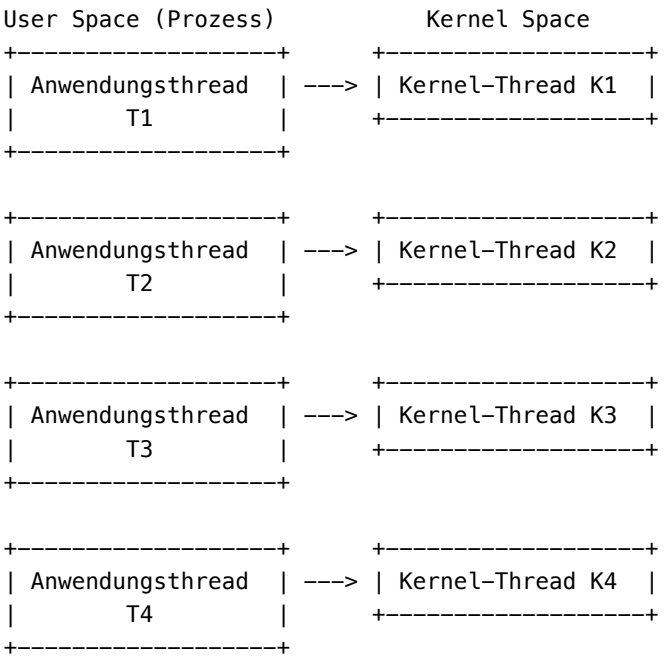

Abbildung 2.4: Darstellung 1:1 Mapping

Beim M:N-Mapping multiplext eine Laufzeitumgebung eine größere Menge von M Anwendungsthreads auf eine deutlich kleinere Menge von N Kernel-Threads (siehe Abbildung 2.5), die in diesem Kontext häufig als Träger- oder Carrier-Threads bezeichnet werden [1, 2]. Das Modell verbindet die Vorteile der beiden vorherigen Ansätze. Die Umschaltung zwischen Anwendungsthreads findet im User Space statt und kommt damit ohne Systemaufruf aus, während der Kernel weiterhin die N Carrier-Threads unabhängig auf den verfügbaren CPU-Kernen einplanen kann und damit echte Parallelausführung erlaubt [2]. Anders als beim N:1-Mapping führt ein blockierender Systemaufruf nicht zum Anhalten des gesamten Prozesses, da die

Laufzeitumgebung den betroffenen Anwendungsthread vom Carrier-Thread abkoppelt und auf demselben Carrier-Thread einen anderen Anwendungsthread ausführt [1, 2]. Diese Architektur prägt die Goroutinen der Programmiersprache Go und die Virtual Threads der Java Virtual Machine, deren konkrete Realisierung in Abschnitt 2.5 im Detail behandelt wird [1, 2, 10].

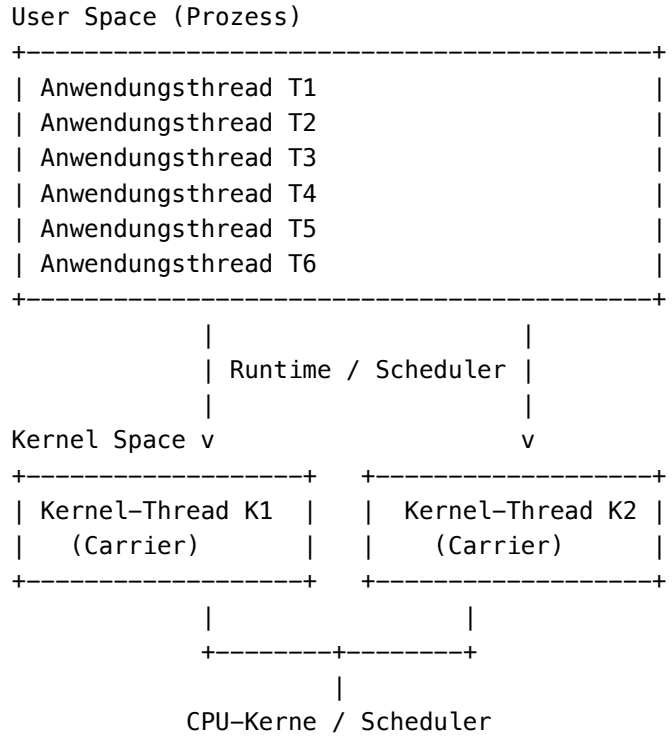

Abbildung 2.5: Darstellung M:N Mapping

Aus diesen Erklärungen folgt, dass die Wahl des Thread-Mappings unmittelbar bestimmt, wie sich Anwendungen auf der Hardware verhalten. Jeder Anwendungsthread bindet im 1:1-Modell einen vollständigen Kernel-Thread, der seinerseits einen vom Kernel reservierten Stack-Bereich benötigt. Die Stack-Größe ist plattform- und konfigurationsabhängig. Der typische Linux-Default für den Stack eines NPTL basierten glibc-Threads liegt auf den verbreiteten Distributionen bei 8 MB pro Thread [26]. Andere Thread-Bibliotheken verwenden deutlich kleinere Werte. Auf der JVM werden Plattform-Threads von der Oracle Dokumentation als ressourcenintensiv und mit einem großen, vom Betriebssystem unterhaltenen Stack charakterisiert [11]. Bei zehntausend gleichzeitig aktiven Threads ergibt sich allein für die Stack-Bereiche rechnerisch ein virtuell reservierter Speicherbedarf in der Größenordnung mehrerer Gigabyte, wobei der Anteil tatsächlich physisch belegten Speichers durch Lazy-Allocation der Stack-Seiten deutlich darunter liegen kann [26].

Eine zentrale Operation des Kernels im Umgang mit Threads ist der Kontextwechsel. Darunter wird der Vorgang verstanden, bei dem die Ausführung eines aktuell laufenden Threads unterbrochen und durch einen anderen lauffähigen Thread fortgesetzt wird. Ein Kontextwechsel umfasst

eine fest definierte Abfolge von Operationen. Der Scheduler sichert zunächst den Registersatz und den Stack-Zeiger des laufenden Threads in dessen Verwaltungsstruktur, lädt anschließend den entsprechenden Zustand des nachfolgenden Threads und übergibt die Kontrolle an dessen letzte Befehlsadresse [8, 21]. Findet der Wechsel zwischen zwei Threads desselben Prozesses statt, bleibt der virtuelle Adressraum unverändert. Findet er hingegen zwischen Threads unterschiedlicher Prozesse statt, muss zusätzlich der in der Speicherverwaltungseinheit aktive Adressraum gewechselt und der TLB entwertet werden, sofern dieser keine prozessbezogenen Identifikationen unterstützt [8]. Auslöser für einen Kontextwechsel lassen sich in zwei Klassen unterteilen. Ein freiwilliger Kontextwechsel tritt ein, wenn ein Thread eine blockierende Operation ausführt und damit seine verbleibende Laufzeit selbst aufgibt. Typische Auslöser sind ein Lesezugriff auf einen Socket ohne verfügbare Daten, ein Warten auf einen Mutex, der Versuch, einen leeren Buffer zu lesen, oder ein expliziter Aufruf von `sched_yield` [8]. Ein unfreiwilliger Kontextwechsel wird vom Scheduler erzwungen, sobald die zugeteilte Laufzeit eines Threads abgelaufen ist oder ein anderer Thread höherer Priorität lauffähig wird [8, 21]. Beide Varianten verursachen auf der Hardware identische unmittelbare Kosten, unterscheiden sich jedoch hinsichtlich ihrer Ursache.
Die Kosten eines Kontextwechsels lassen sich analog zu den Kosten eines Mode Switches in eine direkte und eine indirekte Komponente zerlegen. Unter den direkten Kosten wird auch hier ausschließlich die explizite Arbeit des Schedulers verstanden, also das Sichern und Wiederherstellen des Registersatzes sowie das Umschalten der Verwaltungsstrukturen. Diese Kosten liegen unter Linux je nach Hardware in der Größenordnung weniger Mikrosekunden [27]. Die indirekten Kosten umfassen wiederum alle nachgelagerten Effekte und entstehen dadurch, dass der nachfolgende Thread einen anderen Befehls- und Datenpfad ausführt und auf diese Weise die zuvor in den CPU-Caches und im TLB gehaltenen Arbeitsdaten des verdrängten Threads überschreibt. Bei der Rückkehr des verdrängten Threads müssen dessen Daten dann erneut aus den gemeinsam genutzten Cache-Ebenen oder dem Hauptspeicher geladen werden, was viele zusätzliche Speicherzugriffe nach sich zieht. Die indirekten Kosten hängen stark von der Größe des Arbeitsdatensatzes des verdrängten Threads ab und können die direkten Kosten je nach Workload übersteigen [27].

Eine zusätzliche Kostenquelle entsteht dann, wenn ein Thread zwischen verschiedenen CPU-Kernen wandert. Da L1- und L2-Cache in modernen Prozessoren je Kern organisiert sind, verfügt ein Thread nach einiger Laufzeit auf einem bestimmten CPU-Kern über entsprechende Cache-Inhalte in dessen lokalen Speicher-Hierarchien. Wandert der Thread auf einen anderen CPU-Kern, gehen diese Cache-Inhalte verloren und müssen aus gemeinsam genutzten Cache-Ebenen oder dem Hauptspeicher nachgeladen werden [8]. Ein solcher Switch trägt allgemein die Bezeichnung CPU-Migration und ist auf Mehrkernarchitekturen unter Last die Regel, da der Scheduler versucht, die Last gleichmäßig über alle CPU-Kerne zu verteilen.

2.2.3 Scheduling im Kernel

Der Scheduler ist der Teil des Kernels, der aus einer Menge lauffähiger Threads jeweils den nächsten auszuwählenden Thread bestimmt und ihm einen CPU-Kern zuteilt (siehe Abbildung 2.6) [8, 21]. Auf einem Multi-Core-System trifft er diese Entscheidung pro CPU-Kern, zusätzlich entscheidet er über die Verteilung der Threads über die verfügbaren CPU-Kerne.
Grundsätzlich werden non präemptive und präemptive Scheduling-Verfahren unterschieden. Im non präemptiven Fall läuft ein Thread so lange, bis er von sich aus auf ein Ereignis wartet oder sich beendet. Diese Variante ist nur dann unproblematisch, wenn alle laufenden Threads bekannt und ausreichend kooperativ sind, etwa in eingebetteten Systemen mit festem Aufgabensatz [8]. Im präemptiven Fall darf der Scheduler einen laufenden Thread auch dann unterbrechen, wenn dieser nicht freiwillig die Kontrolle abgibt, etwa nach Ablauf einer zugewiesenen Laufzeit oder wenn ein Thread mit höherer Priorität lauffähig wird [8, 21]. Moderne Multiuser-Betriebssysteme arbeiten ausnahmslos präemptiv, da nur dieses Modell garantiert, dass kein einzelner Thread die CPU dauerhaft blockieren kann [8].

Die klassische präemptive Strategie ist das time-slice basierte Verfahren in Verbindung mit einer First In, First Out (FIFO) Warteschlange, das allgemein als Round-Robin bezeichnet wird [8, 21]. Jeder lauffähige Thread erhält einen festen time slice und wird nach dessen Ablauf an das Ende der Warteschlange zurückgestellt. Die Wahl der time-slice Größe entscheidet dabei über das Verhältnis zwischen Antwortverhalten und Verwaltungsaufwand. Wird sie zu klein gewählt, verschlechtert sich das Verhältnis zwischen Nutzarbeit und Umschaltzeit, im Extremfall schaltet der Prozessor lediglich noch zwischen Threads um, ohne diese tatsächlich auszuführen. Brause nennt für diesen klassischen Ansatz einen Richtwert von etwa einhundert Millisekunden, was oberhalb des typischen CPU-Bursts der meisten Prozesse liegt [8]. Dieser Wert gilt jedoch ausschließlich für das beschriebene klassische Round-Robin-Verfahren und ist nicht auf moderne adaptive Scheduler übertragbar.
Reines Round Robin reicht auf modernen Multiuser-Systemen nicht aus, da unterschiedliche Threads verschiedene Prioritäten besitzen und reines Round Robin keine Priorisierung erlaubt. Daher haben sich Verfahren mit mehreren Warteschlangen unterschiedlicher Priorität entwickelt. Beim Multi-Level-Scheduling wird jede Prioritätsklasse durch eine eigene Warteschlange repräsentiert, die in fester Reihenfolge abgearbeitet wird. Beim Multi-Level-Feedback-Scheduling können Threads zusätzlich zwischen den Warteschlangen wechseln, etwa dadurch, dass die Priorität eines wartenden Threads mit zunehmender Wartezeit ansteigt und der Thread auf diese Weise in eine höhere Warteschlange aufrückt [8, 21]. Dieses Schema entkoppelt Threads mit langer Wartezeit von solchen mit kontinuierlicher CPU-Beanspruchung und verhindert das sogenannte Verhungern einzelner Threads.

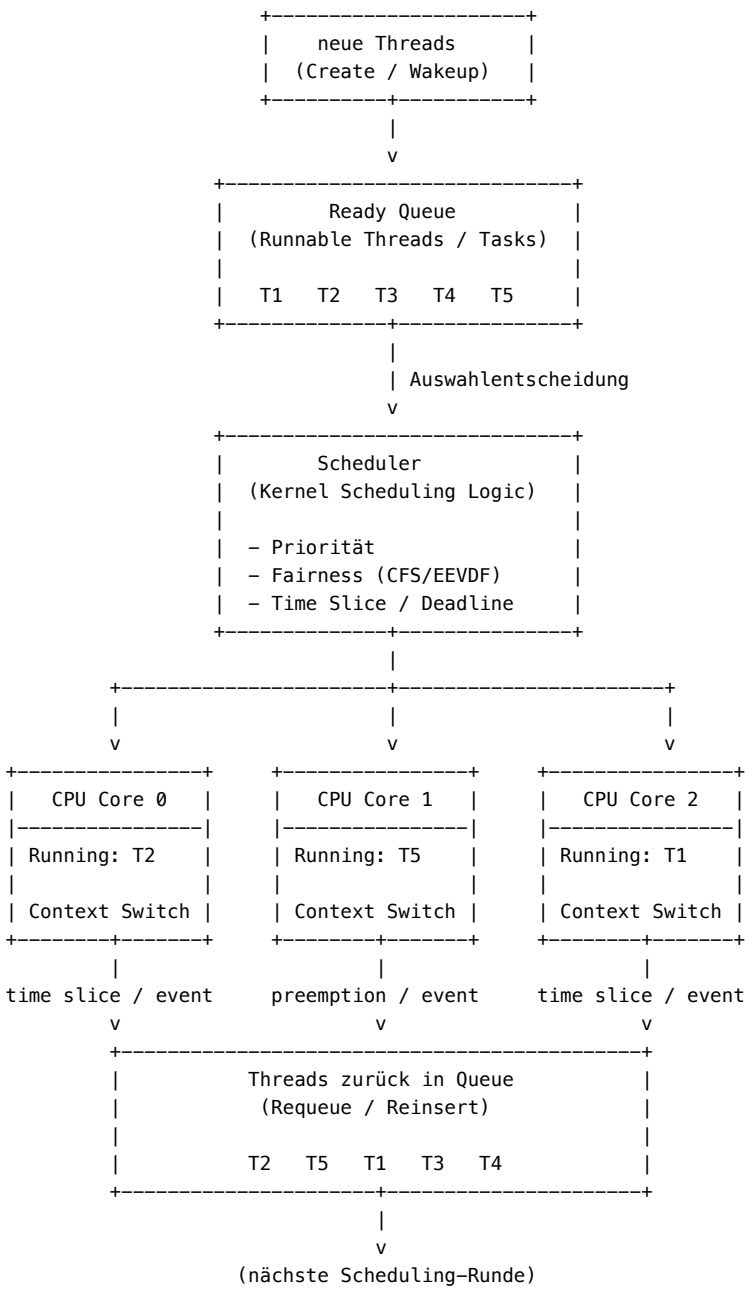

Abbildung 2.6: Darstellung Scheduling-Runde

Linux verwendet seit Kernel-Version 6.6, veröffentlicht im Oktober 2023, den Earliest Eligible Virtual Deadline First (EEVDF) [28]. Er löst den Completely Fair Scheduler (CFS) ab, der seit Version 2.6.23 aus dem Jahr 2007 als Standard für reguläre Anwendungsprozesse diente [29, 30]. EEVDF verfolgt dasselbe Grundziel wie sein Vorgänger, nämlich die faire Verteilung der CPU-Zeit auf alle lauffähigen Threads. Es ergänzt das Prinzip jedoch um einen Mechanismus,

der das Antwortverhalten gegenüber latenzsensitiven Anwendungen deutlich verbessert und im CFS noch über separate Erweiterungen nachgerüstet werden musste [28, 31].

Das Modell nutzt zwei Kennzahlen pro Thread. Die erste ist der Lag (Differenz), der ausdrückt, wie weit ein Thread von seinem rechnerisch fairen Anteil an der CPU-Zeit entfernt ist. Ein positiver Lag bedeutet, dass dem Thread noch Zeit zusteht, ein negativer Lag, dass er mehr erhalten hat als ihm zugestanden hätte. Nur Threads mit positivem Lag gelten als berechtigt, in der aktuellen Auswahlrunde überhaupt eingeplant zu werden. Die zweite Kennzahl ist die virtual Deadline, also der Zeitpunkt, bis zu dem ein Thread seine aktuelle zugewiesene Laufzeit (Quantum) abgearbeitet haben sollte. Aus der Menge der berechtigten Threads wählt der Scheduler stets denjenigen mit der frühesten virtuellen Deadline aus [28, 31].
Die Stärke dieses Ansatzes liegt darin, dass Threads mit kurzen Quanten automatisch frühere virtual Deadlines erhalten und damit bevorzugt zum Zuge kommen. Latenzsensitive Anwendungen, die typischerweise nur kurze CPU-Bursts benötigen, profitieren davon, ohne dass das System ihnen explizit eine höhere Priorität zuweisen müsste. Auf Datenstrukturebene behält der EEVDF ebenso das Prinzip seines Vorgängers bei und alle lauffähigen Threads werden in einem Rot-Schwarz-Baum verwaltet. Diese Struktur erlaubt das Einfügen, Entfernen und Auslesen jedes Threads in logarithmischer Zeit, sodass auch sehr große Mengen lauffähiger Threads effizient verwaltet werden können [28, 31].

Für hochnebenläufige Anwendungen bleibt eine zentrale Eigenschaft des Schedulings erhalten. Mit wachsender Anzahl lauffähiger Threads verteilt sich die verfügbare CPU-Zeit auf immer mehr Aufgaben und die Frequenz der Kontextwechsel steigt proportional an. Der akkumulierte Scheduling-Overhead nimmt damit unabhängig vom konkreten Scheduler direkt mit der Thread-Anzahl zu.

2.3 CPU-Caches, Cache-Lokalität und NUMA-Architekturen

Die in den vorangegangenen Abschnitten beschriebenen direkten und indirekten Kosten von Mode Switches und Kontextwechseln verweisen immer wieder auf hardwarenahe Strukturen wie L1-, L2- und L3-Caches, ohne dass deren Funktionsweise bisher im Detail erklärt worden ist. Ohne ein Verständnis dieser Strukturen lässt sich nicht beurteilen, warum die indirekten Kosten eines Kontextwechsels die direkten Kosten in vielen Fällen übersteigen und warum eine CPU-Migration messbare Folgen für die Laufzeit eines Threads hat.

Moderne Prozessoren stehen vor einem grundlegenden Geschwindigkeitsproblem zwischen CPU und Hauptspeicher. Während die Taktfrequenz und die Anzahl der Recheneinheiten pro Prozessor über mehrere Jahrzehnte stetig zugenommen haben, ist die Zugriffszeit auf dynamischen Hauptspeicher vergleichsweise wenig gesunken, was zu einer wachsenden Lücke zwischen verfügbarer Rechenleistung und nutzbarer Speicherbandbreite führt [32, 33]. Hardwareseitig wird dieser Lücke durch eine Reihe von Pufferspeichern entgegengewirkt, die zwischen CPU und Hauptspeicher liegen und die als Caches bezeichnet werden [8, 33]. Ein Cache nutzt die

Beobachtung aus, dass Programmcode überwiegend lokale Bezüge zu Daten und kurze Sprungweiten aufweist, sodass ein in einen schnellen Hilfsspeicher kopierter Programmabschnitt die Abarbeitungsgeschwindigkeit deutlich erhöht [8].

Auf den x86-64 Prozessoren sind diese Pufferspeicher typischerweise dreistufig aufgebaut (siehe Abbildung 2.7). Der L1-Cache wird pro CPU-Kern bereitgestellt und in einen Daten- und einen Instruktionsteil aufgeteilt, was der Harvard-Architektur folgt und parallele Zugriffe auf Code und Daten ermöglicht [8, 32]. Der L2-Cache ist ebenfalls pro CPU-Kern organisiert und vereint Daten und Instruktionen, der L3-Cache wird hingegen von allen Kernen eines Prozessors gemeinsam genutzt [32, 33]. Mit jeder höheren Stufe nehmen Größe und Zugriffszeit zu, während die nutzbare Bandbreite abnimmt. Die Latenzunterschiede zwischen den Stufen liegen in der Größenordnung mehrerer Faktoren, also wenige Taktzyklen für L1, etwa eine Größenordnung mehr für L2 und L3 und mehrere hundert Taktzyklen für einen Zugriff auf den Hauptspeicher [32, 33]. Ein Zugriff, der bereits im L1-Cache beantwortet werden kann, ist auf moderner Hardware damit zwei bis drei Größenordnungen schneller als ein Zugriff auf den Hauptspeicher [32].

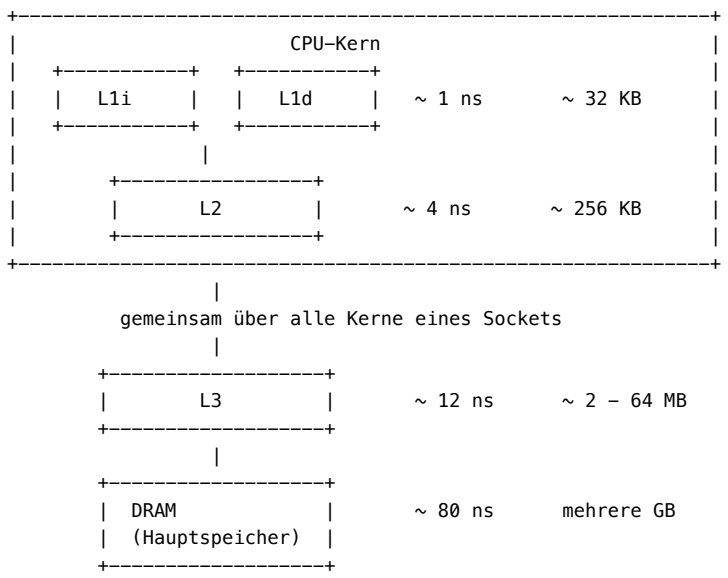

Abbildung 2.7: Schematische Darstellung der Cache-Hierarchie

Die Cache-Verwaltung erfolgt nicht byteweise, sondern in festen Blöcken, die als Cache Lines bezeichnet werden. Auf gängigen x86-64 Prozessoren beträgt die Größe einer Cache Line einheitlich 64 Byte [32, 33]. Fordert ein Programm eine Speicheradresse an, deren Inhalt nicht im Cache vorhanden ist, spricht man von einem *Cache Miss*, kann der Zugriff hingegen unmittelbar aus einer Cache-Ebene beantwortet werden, liegt entsprechend ein *Cache Hit* vor [33]. Im Fall eines Cache Miss lädt der Prozessor nicht nur das angeforderte Byte, sondern die gesamte enthaltende Cache Line aus dem nächsthöheren Cache oder dem Hauptspeicher [32].

Da die Hierarchie schrittweise durchsucht wird, unterscheidet man weiter zwischen Misses, die noch auf einer höheren Cache-Ebene beantwortet werden können (z. B. einem L1 Miss bei gleichzeitigem L2 Hit) und solchen, die bis zum Hauptspeicher führen, den sogenannten *Last-Level-Cache Misses* (LLC Misses). Letztere verursachen die erwähnten Latenzen von mehreren hundert Taktzyklen und sind damit für die Laufzeit eines Threads besonders teuer [32, 33].

Diese Eigenschaften sind bewusst so implementiert und eine direkte Antwort auf zwei seit langem belegte Eigenschaften des Speicherzugriffsverhaltens laufender Programme [8, 33]. Die erste Eigenschaft ist die räumliche Lokalität, dies beschreibt die Beobachtung, dass auf Adressen in unmittelbarer Nachbarschaft einer kürzlich referenzierten Adresse mit hoher Wahrscheinlichkeit ebenfalls nach kurzer Zeit zugegriffen wird, etwa beim sequentiellen Durchlaufen eines Arrays, beim Abarbeiten eines Stack-Frames oder bei der Ausführung einer Befehlsfolge [8, 32]. Genau diese Eigenschaft macht das blockweise Laden ganzer Cache Lines effizient, weil der einmalige Transport eines 64-Byte-Blocks die für die folgenden Zugriffe benötigten Daten mit großer Wahrscheinlichkeit bereits enthält und damit weitere Cache Misses und Zugriffe auf den langsamen Hauptspeicher erspart [32]. Die zeitliche Lokalität, ist die zweite Eigenschaft und bezeichnet die ergänzende Beobachtung, dass eine kürzlich referenzierte Adresse mit hoher Wahrscheinlichkeit innerhalb kurzer Zeit erneut referenziert wird, was sich typischerweise bei Schleifenvariablen, häufig genutzten Datenstrukturen oder wiederholt ausgeführten Codeabschnitten zeigt [8]. Sie rechtfertigt, dass eine einmal in den Cache geladene Cache Line dort möglichst lange gehalten und erst dann verdrängt wird, wenn der Platz für andere Daten benötigt wird. Erst aus der Kombination beider Eigenschaften ergibt sich der praktische Nutzen eines Caches. Ohne räumliche Lokalität wäre das blockweise Laden ineffizient, ohne zeitliche Lokalität bleibt das Halten bereits geladener Daten ohne Effekt [8, 33].

Zusätzlich muss beachtet werden, dass moderne Prozessoren mehrere Rechenkerne mit jeweils eigenen L1- und L2-Caches besitzen. Da derselbe Speicherinhalt gleichzeitig in mehreren Caches vorliegen kann, müssen die Prozessoren sicherstellen, dass eine Änderung in einem Cache allen anderen Kernen mitgeteilt wird, bevor diese auf veraltete Daten zugreifen [8, 33]. Realisiert wird diese Garantie über ein Cache-Kohärenzprotokoll, das jede Cache Line mit einem Statuswort versieht und Änderungen zwischen den Caches synchronisiert.

Das MESI-Protokoll wurde 1984 von Papamarcos und Patel an der University of Illinois at Urbana-Champaign publiziert [33, 34]. Es wurde später in einer Reihe kommerzieller Prozessorarchitekturen umgesetzt, unter anderem in den Pentium-Prozessoren von Intel [8, 33]. Das Protokoll weist jeder Cache Line genau einen von vier Zuständen zu [8, 33]. Im Zustand *Modified* hält genau ein Cache eine Kopie der Cache Line vor, deren Inhalt gegenüber dem Hauptspeicher verändert wurde. Der Hauptspeicher ist in diesem Zustand nicht aktuell, sodass die geänderten Daten irgendwann zurückgeschrieben werden müssen [8]. Im Zustand *Exclusive* hält ebenfalls nur ein einziger Cache eine Kopie, deren Inhalt jedoch mit dem Hauptspeicher übereinstimmt. Damit kann der haltende Kern die Cache Line ohne Rückfrage bei anderen Caches modifizieren und in den Zustand *Modified* überführen. Im Zustand *Shared* liegen in

potenziell mehreren Caches identische Kopien der Cache Line vor, die ebenfalls mit dem Hauptspeicher übereinstimmen. Schreibzugriffe sind in diesem Zustand erst nach entsprechender Koordination mit den übrigen Caches möglich. Der Zustand *Invalid* beschreibt schließlich eine Cache Line, deren Inhalt als ungültig markiert ist und die bei einem Zugriff zunächst aus einer höheren Cache-Ebene oder aus dem Hauptspeicher nachgeladen werden muss (siehe Abbildung 2.8) [8, 33].

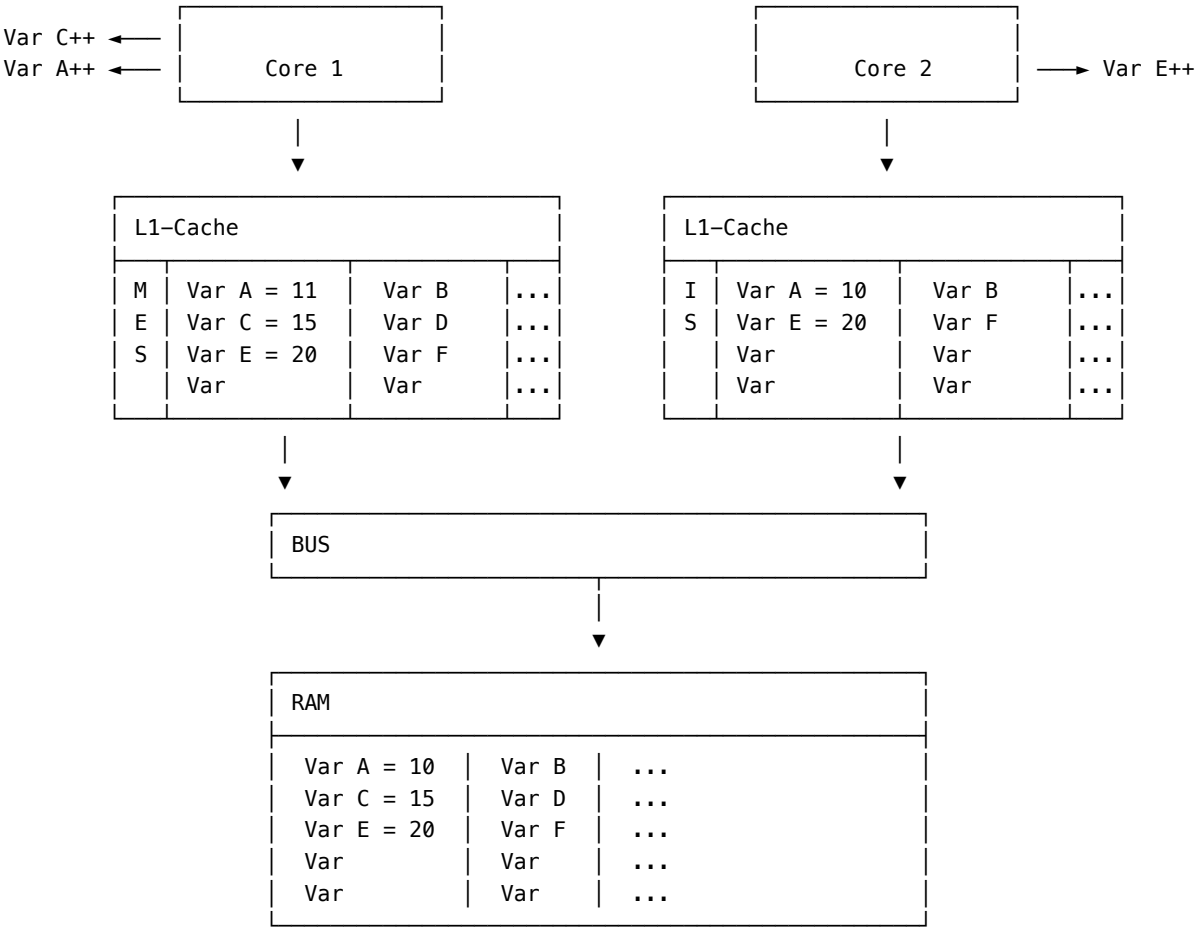

Abbildung 2.8: MESI-Protokoll: Cache-Kohärenz zwischen zwei Cores

Die Übergänge zwischen diesen Zuständen werden durch lokale Speicherzugriffe des haltenden Kerns sowie durch beobachtete Speicherzugriffe anderer Kerne ausgelöst. Beobachtet werden diese fremden Zugriffe über einen sog. Snooper, der den Bus oder die zwischen den Caches genutzten Verbindungen überwacht und Statusänderungen anderer Caches registriert [8]. Liest der haltende Kern eine Cache Line, die zuvor nicht vorhanden war, prüft das Protokoll, ob eine Kopie in einem anderen Cache existiert. Existiert keine, wird die Cache Line in den Zustand *Exclusive* überführt. Existiert eine, gehen beide Kopien in den Zustand *Shared* über [33]. Schreibt der haltende Kern in eine Cache Line, die sich im Zustand *Shared* befindet, müssen

sämtliche Kopien in anderen Caches durch einen Invalidierungsbefehl auf *Invalid* gesetzt werden, bevor die Schreiboperation lokal als *Modified* abgeschlossen werden kann [8, 33]. Liest ein anderer Kern eine Cache Line, die sich in einem fremden Cache im Zustand *Modified* befindet, muss der haltende Cache die Daten zunächst in den Hauptspeicher zurückschreiben, beide Kopien wechseln anschließend in den Zustand *Shared*. Schreibt ein anderer Kern in eine solche Cache Line, geht die lokale Kopie auf *Invalid* über. Bei zuvor *Modified* gehaltenen Daten wird der Inhalt zunächst in den Hauptspeicher zurückgeschrieben [8, 33].

Aus dieser Synchronisation auf Cache-Line-Ebene ergibt sich ein für nebenläufige Anwendungen besonders relevanter Effekt, das sog. False Sharing. Liegen zwei voneinander logisch unabhängige Variablen zufällig in derselben Cache Line und werden sie von zwei Threads auf unterschiedlichen CPU-Kernen gleichzeitig beschrieben, behandelt das Kohärenzprotokoll die Cache Line als geteiltes Datum und invalidiert sie bei jeder Schreiboperation des jeweils anderen Kerns [32]. Die beteiligten Kerne arbeiten zwar auf logisch getrennten Daten, müssen sich aber dennoch permanent über das Kohärenzprotokoll abstimmen, wodurch die betreffende Cache Line wiederholt zwischen den Kernen übertragen werden muss. Auf Basis des MESI-Protokolls wechselt der Zustand bei jeder Schreiboperation eines Kerns auf *Modified*, beim nächsten Schreibzugriff des anderen Kerns über *Invalid* zurück auf *Modified* im fremden Cache usw. Der dadurch entstehende Bandbreitenverbrauch und die zusätzlichen Cache Misses können die Leistung betroffener Codepfade um eine Größenordnung verschlechtern, ohne dass die Anwendung selbst einen Synchronisationsfehler aufweist [32]. False Sharing tritt in hochnebenläufigen Anwendungen insbesondere dann auf, wenn unabhängige Zustandsfelder pro Thread innerhalb derselben Datenstruktur abgelegt werden, etwa pro Thread geführte Zähler in einem gemeinsamen Array.

Zusätzliche Komplexität kommt in Systemen hinzu, in denen der Hauptspeicher nicht mehr zentral an einen einzigen Bus angebunden, sondern in mehrere lokal an einzelne Prozessoren angeschlossene Bänke aufgeteilt wird. [8, 35]. Diese Architektur wird als Non-Uniform Memory Access bezeichnet und ist auf heutigen Mehrsockel-Servern der Regelfall, da die Anbindung des gesamten Hauptspeichers an einen einzelnen zentralen Bus bei steigender Kernzahl zum Engpass wird [35]. Ein Prozessor erreicht seinen lokalen Speicher direkt über den eigenen Speichercontroller, während Zugriffe auf den Speicher eines anderen Sockels über eine Verbindung zwischen den Prozessoren laufen müssen, etwa über Intel QuickPath Interconnect oder AMD Infinity Fabric. Die Latenz eines solchen entfernten Zugriffs liegt dadurch je nach System um einen Faktor von etwa 1,5 bis 2 höher als die eines lokalen Zugriffs und die nutzbare Bandbreite ist entsprechend reduziert [32, 35].

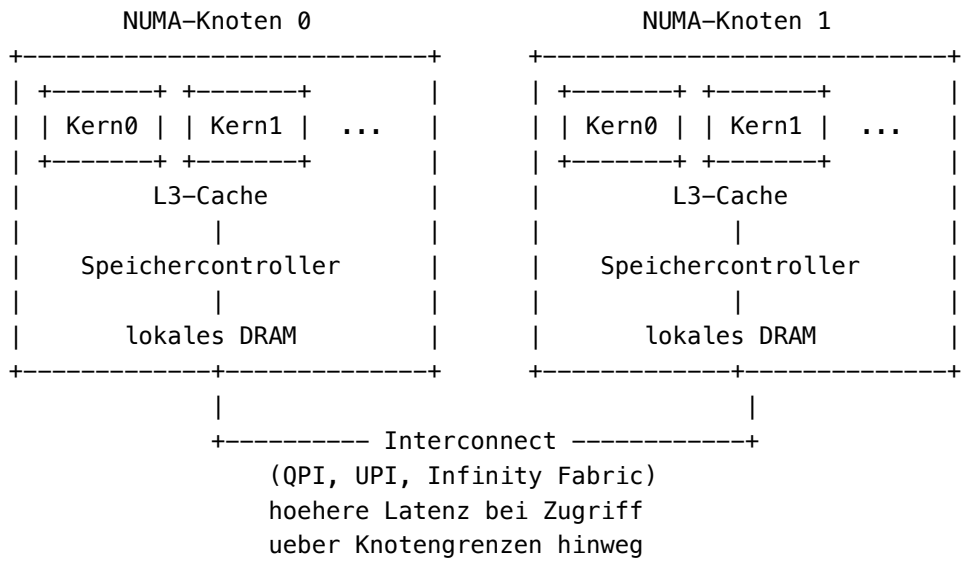

Abbildung 2.9: Darstellung einer einfachen NUMA-Topologie mit zwei Knoten

Damit Anwendungscode transparent auf einer NUMA-Architektur laufen kann, ohne dass die Programmierung explizit zwischen lokalem und entferntem Speicher unterscheiden muss, werden heutige Systeme als cache-kohärente NUMA (ccNUMA)-Systeme realisiert (siehe Abbildung 2.9). Im weiteren Verlauf wird der Begriff NUMA jedoch synonym verwendet. [8, 35]. Das oben beschriebene MESI-Protokoll wird in diesem Fall über die Sockelgrenzen hinweg ausgedehnt, sodass eine Statusänderung einer Cache Line in einem prozessorlokalen Cache auch in den Caches der übrigen Sockel sichtbar wird [8]. Aus Sicht der Anwendungen ist der gesamte Speicher damit einheitlich adressierbar, aus Sicht der Performance bleiben die Latenzunterschiede zwischen lokalem und entferntem Zugriff jedoch bestehen [35].

Für das Betriebssystem ergeben sich aus NUMA zwei miteinander verbundene Entscheidungen. Erstens muss bei der Allokation einer Speicherseite festgelegt werden, an welchen NUMA-Knoten diese Seite physisch gebunden wird. Linux verfolgt hierfür die Strategie der first-touch-Allokation, also die Bindung der Seite an den Knoten des Threads, der zuerst schreibend auf die Seite zugreift [35]. Zweitens muss der Scheduler bei der Wahl eines CPU-Kerns für einen lauffähigen Thread berücksichtigen, auf welchem Knoten dessen Speicherseiten liegen, da eine Migration des Threads auf einen entfernten Knoten den Anteil entfernter Zugriffe sofort erhöht. Seit Linux Kernel-Version 3.8 enthält der Kernel hierfür eine als *Automatic NUMA Balancing* bezeichnete Komponente, die aktiv genutzte Seiten näher an den zugreifenden Thread verschiebt, um die Kosten entfernter Zugriffe zu reduzieren [35]. Die Erkennung erfolgt indirekt über sog. NUMA hinting faults, eine besondere Form des Page Faults. Ein Page Fault bezeichnet allgemein eine durch die Speicherverwaltungseinheit der CPU ausgelöste Ausnahme, die immer dann auftritt, wenn ein Thread auf eine virtuelle Adresse zugreift, deren Übersetzung in eine physische Adresse nicht oder nicht mit den geforderten Rechten in der Seitentabelle des Prozesses hinterlegt ist. Die weitere Behandlung übernimmt das Betriebssystem, das anschließend entscheidet, ob

die betroffene Seite eingelagert, neu zugewiesen oder als Zugriffsverletzung an den auslösenden Prozess signalisiert wird [8]. Das Automatic NUMA Balancing nutzt diesen Mechanismus gezielt aus. Der Kernel hebt dafür periodisch die Zugriffsrechte auf einen Teil des Adressraums eines Tasks auf, sodass der nächste Zugriff auf eine betroffene Seite einen kontrollierten Page Fault auslöst, ohne dass tatsächlich Daten nachgeladen werden müssten. Über diesen Fault wird festgehalten, von welchem CPU-Kern und damit von welchem NUMA-Knoten aus die Seite gerade benutzt wird. Seiten, die wiederholt von einem entfernten Knoten aus referenziert werden, werden anschließend auf diesen Knoten verschoben, wobei nach einer ersten Migration ein Filter weitere Verschiebungen nur dann zulässt, wenn erneut mehrfach entfernte Zugriffe registriert worden sind, um ein ständiges Hin- und Herschieben derselben Seite zwischen Knoten zu vermeiden [35]. Die Wirksamkeit dieses Verfahrens hängt jedoch davon ab, wie oft Threads zwischen Knoten migrieren und wie regelmäßig deren Speicherzugriffsmuster bleiben. Lameter weist darauf hin, dass eine optimale NUMA-Verteilung in der Praxis häufig ein explizites Pinning der Threads an bestimmte CPU-Kerne über `taskset` oder `cpusets` erfordert [35].

2.4 Blockierendes und nicht-blockierendes I/O

Die im vorangegangenen Abschnitt beschriebenen Mechanismen von Scheduling und Kontextwechsel werden in serverseitigen Anwendungen vor allem dann ausgelöst, wenn Threads auf den Abschluss von I/O-Operationen warten. Wie ein Programm auf eine solche Wartesituation reagiert, hängt davon ab, in welcher Variante der Systemaufruf erfolgt. Die Unterscheidung zwischen einem blockierenden und einem nicht blockierenden Systemaufruf prägt damit nicht nur die Struktur des Programmcodes, sondern auch die Häufigkeit der bisher diskutierten Mode Switches, Kontextwechsel und CPU-Migrationen. Im Kontext serverseitiger Anwendungen ist dabei in erster Linie Netzwerk-I/O relevant, also die Zugriffe auf TCP-Sockets, über die eingehende Verbindungen angenommen und Daten ausgetauscht werden [3, 8]. Reguläre Dateizugriffe weisen dabei ein komplett anderes Verhalten auf und stellen einen Sonderfall dar.

Ein blockierender Systemaufruf hält den aufrufenden Thread so lange im Wartezustand, bis das erwartete Ereignis eingetreten ist. Im Serverbetrieb tritt dieser Zustand vor allem an zwei Stellen auf. Erstens beim Annehmen eingehender TCP-Verbindungen über `accept`, das blockiert, solange kein Client verbunden ist. Zweitens beim Lesen von Daten einer bestehenden Verbindung über `read` oder `recv`, das blockiert, solange die Gegenseite noch keine Daten gesendet hat oder diese noch nicht im kernelseitigen Socket-Empfangspuffer angekommen sind [8]. Der Kernel nimmt den Thread in dieser Situation aus der Run-Queue des Schedulers, hinterlegt ihn in der zum betroffenen Dateideskriptor gehörenden Warteliste und führt einen Kontextwechsel auf einen anderen lauffähigen Thread aus [8, 21]. Trifft ein TCP-Segment mit Daten ein, benachrichtigt der Kernel den wartenden Thread, versetzt ihn in die Run-Queue zurück und teilt ihm die Anzahl der im Puffer verfügbaren Bytes mit [8]. Während der gesamten

Wartezeit verbraucht der Thread keine CPU-Zeit, belegt aber weiterhin die in Abschnitt 2.2.2 beschriebenen Strukturen und einen vorab reservierten Stack-Bereich [8, 26].

Aus dieser Bauweise ergibt sich die enge Verbindung zwischen blockierendem Netzwerk-I/O und dem Thread-per-Request Modell. Da der Kernel die Wartesituation an einen einzelnen Thread hängt, ist pro gleichzeitig bestehender TCP-Verbindung mindestens ein Thread erforderlich, der diese Verbindung von der Annahme bis zur abschließenden Antwort begleitet. Genau dieser Zusammenhang machte das C10k-Problem auf der Ebene der Programmiermodelle sichtbar (s. Abschnitt 2.1). Die Skalierungsgrenze entstand nicht durch die blockierende Systemschnittstelle selbst, sondern durch die Notwendigkeit, für jede der zehntausend gleichzeitig offenen TCP-Verbindungen einen vom Kernel verwalteten Thread vorzuhalten, was in der Summe an die Grenzen der verfügbaren Speicher- und Scheduling-Ressourcen stößt [2, 3].

Ein nicht blockierender Systemaufruf kehrt hingegen sofort zurück, unabhängig davon, ob die angeforderte Operation ausgeführt werden konnte. Unter Linux wird dieses Verhalten durch das Setzen des Flags `O_NONBLOCK` am betroffenen Dateideskriptor aktiviert. Im Anschluss liefert ein `read` auf einen leeren Socket-Empfangspuffer den Fehlercode `EAGAIN` bzw. `EWOULDBLOCK` zurück, anstatt den Thread zu blockieren [8]. Damit wird die Verantwortung für das Warten von der Kernel-Ebene in die Anwendungsentwicklung verschoben. Eine Umsetzung eben dieser wäre das aktive Pollen der Dateideskriptoren, also das wiederholte Absetzen nicht blockierender Aufrufe bis zur erfolgreichen Ausführung. Dessen Einsatz nur dann vertretbar ist, wenn die Wartezeit absehbar wenige CPU Takte beträgt und ein Kontextwechsel teurer wäre als das aktive Abfragen [8]. Für die üblichen Netzwerkwartezeiten, die von der Round-Trip-Time zwischen Client und Server abhängen und typischerweise im Bereich mehrerer Millisekunden liegen, scheidet aktives Pollen als Variante aus. Das Modell der nicht blockierenden I/O-Operationen funktioniert in dieser Form ausschließlich für Dateideskriptoren, deren Bereitschaft sich sinnvoll abfragen lässt, also für Sockets, Pipes und ähnliche Stream-basierte Konstrukte. Reguläre Dateien verhalten sich hier grundlegend anders, ihr Sonderfall wird weiter unten gesondert behandelt.

Praktisch nutzbar wird nicht blockierendes Netzwerk-I/O erst in Verbindung mit einem Mechanismus zur Ereignisbenachrichtigung, der dem aufrufenden Thread mitteilt, welche aus einer Menge beobachteter Socket-Deskriptoren tatsächlich lese- oder schreibbereit sind. Die ersten Schnittstellen dieser Art waren die POSIX-Aufrufe `select` und `poll`, die einen vom Anwendungsprozess übergebenen Satz von Dateideskriptoren auf I/O-Bereitschaft prüfen [2]. Beide Aufrufe weisen jedoch eine architektonische Schwäche auf. Pro Aufruf muss der gesamte Satz beobachteter Deskriptoren vom User Space in den Kernel kopiert und im Kernel durchlaufen werden, sodass ihr Aufwand mit der Anzahl beobachteter Verbindungen linear wächst [3].

Als Reaktion auf diese Skalierungsgrenze haben die großen Server-Plattformen jeweils eigene zustandsbehaftete Schnittstellen entwickelt, die den beobachteten Satz von Socket-Deskriptoren dauerhaft im Kernel halten und ausschließlich über tatsächlich bereite Deskriptoren berichten. Linux führte mit `epoll`, FreeBSD mit `kqueue` und Windows mit I/O Completion Ports (IOCP) jeweils Mechanismen ein, deren Aufwand pro Abfrage nicht mehr linear mit der Gesamtzahl

beobachteter Deskriptoren, sondern lediglich mit der Anzahl tatsächlich bereiter Deskriptoren wächst [2, 3]. Durch diese Schnittstellen wurden ereignisgetriebene Server wie nginx und Node.js auf dem zuvor beschriebenen Skalierungsniveau praktisch umsetzbar [4, 19].

Linux unterscheidet bei `epoll` zusätzlich zwischen einer level-triggered und einer edge-triggered Benachrichtigung. Im level-triggered Modus wird eine Bereitschaftsbenachrichtigung so lange wiederholt geliefert, wie der entsprechende Socket-Deskriptor lesbar oder schreibbar bleibt. Im edge-triggered Modus wird hingegen nur ein einzelnes Ereignis beim Wechsel des Bereitschaftszustands ausgelöst, sodass die Anwendung den zugehörigen Socket-Puffer in einer Schleife so lange leeren oder füllen muss, bis ein nicht blockierender Aufruf `EAGAIN` zurückgibt [2]. Der edge-triggered Modus reduziert die Anzahl der vom Kernel zugestellten Benachrichtigungen und ist in hochnebenläufigen Implementierungen die übliche Wahl, verlangt jedoch eine sorgfältige Implementierung im User Space [2].

Allen bisher vorgestellten Schnittstellen zur Ereignisbenachrichtigung `select`, `poll`, `epoll` und `kqueue`, liegt ein bereitschaftsbasiertes Modell zugrunde. Der Kernel meldet hierbei nicht den Abschluss einer Operation, sondern lediglich, dass ein anschließender nicht blockierender Zugriff auf den betreffenden Deskriptor voraussichtlich ohne Warten ausgeführt werden kann. Dieses Modell setzt voraus, dass ein Deskriptor überhaupt einen nicht bereiten Zustand einnehmen kann, der sich später in einen bereiten Zustand wandelt und dadurch ein beobachtbares Ereignis erzeugt. Für Sockets ist diese Voraussetzung erfüllt, da empfangene Daten erst über das Netzwerk eintreffen müssen, bevor sie gelesen werden können. Reguläre Dateien kennen hingegen keinen vergleichbaren Zustandsübergang, weshalb sich das bereitschaftsbasierte Modell auf sie nicht sinnvoll anwenden lässt [36].

Konkret zeigt sich diese Eigenheit bereits am Flag `O_NONBLOCK`, das auf regulären Dateien praktisch wirkungslos bleibt. Ein `read` auf eine Datei kehrt nicht mit `EAGAIN` zurück, sondern führt den Lesezugriff aus und blockiert dabei genau dann, wenn die angeforderten Daten nicht bereits im Page-Cache vorliegen und zunächst vom Speichermedium nachgeladen werden müssen [21, 36]. Da der Page-Cache häufig benötigte Dateiinhalte im Hauptspeicher vorhält, kehrt der überwiegende Teil der Lesezugriffe unmittelbar zurück. Ob ein einzelner Zugriff jedoch sofort aus dem Cache bedient werden kann oder den Thread für die Dauer einer Geräteoperation in den Wartezustand versetzt, ist aus Sicht der Anwendung nicht vorhersehbar. Genau aus diesem Grund meldet der Kernel reguläre Dateien gegenüber `select`, `poll` und `epoll` unabhängig vom tatsächlichen Cache-Zustand stets als lese- und schreibbereit, obwohl der nachfolgende Zugriff dennoch blockieren kann [36].

Ereignisgetriebene Laufzeitsysteme begegnen diesem Umstand in den hier betrachteten Versionen, indem sie blockierende Dateizugriffe nicht in die `epoll`-basierte Ereignisschleife einreihen, sondern an einen Pool dedizierter Worker-Threads auslagern. Diese führen den blockierenden Aufruf stellvertretend aus, während der Thread der Ereignisschleife ungehindert weitere Netzwerkereignisse bedienen kann. Der Abschluss der Dateioperation wird der Schleife anschließend über einen internen Benachrichtigungsmechanismus zurückgemeldet. Node.js setzt dieses Verfahren über die Thread-Pool-Komponente von libuv um [4]. Damit kehrt für den

Sonderfall der Dateizugriffe gewissermaßen das in Abschnitt 2.1 beschriebene Prinzip eines eigenen Threads je blockierender Operation zurück, nun jedoch begrenzt auf einen kleinen, festen Pool im Inneren der ansonsten ereignisgetriebenen Architektur.

Eine jüngere Entwicklung ist die 2019 mit Linux Kernel-Version 5.1 eingeführte Schnittstelle `io_uring`, die zwischen Kernel und User Space zwei gemeinsam in den Speicher abgebildete ringförmige Warteschlangen einrichtet, eine Submission Queue für einzureichende Operationen und eine Completion Queue für deren Abschlussmeldungen [36]. Beide Seiten arbeiten direkt auf den geteilten Ringen, sodass für die Übermittlung einer Operation kein Mode Switch pro Aufruf mehr erforderlich ist und mehrere Operationen in einem einzigen Systemaufruf zusammengefasst werden können [36]. Anders als die zuvor beschriebenen Schnittstellen folgt `io_uring` damit einem abschlussbasierten Modell, bei dem die Anwendung eine Operation einreicht und über deren tatsächlichen Abschluss benachrichtigt wird, statt lediglich deren Ausführbarkeit abzufragen. Eben dieser Unterschied schließt die oben beschriebene Lücke zwischen Netzwerk- und Datei-I/O, da sich der Abschluss einer Operation im Gegensatz zur bloßen Bereitschaft auch für reguläre Dateizugriffe sinnvoll melden lässt, `io_uring` bietet somit echte Asynchronität auch dort, wo `epoll` strukturell ungeeignet ist [36]. Da die in dieser Arbeit untersuchten Laufzeitsysteme ihr I/O-Subsystem in den betrachteten Versionen weiterhin primär auf `epoll` bzw. `kqueue` stützen, wird `io_uring` im weiteren Verlauf nicht eigens evaluiert.

2.5 Nebenläufigkeitsmodelle und untersuchte Laufzeitsysteme

Die in den vorangegangenen Abschnitten eingeführten Konzepte des Thread-Mappings, der Kontextwechsel sowie der I/O-Behandlung beschreiben den allgemeinen Rahmen, in dem moderne Laufzeitsysteme operieren. Wie diese Mechanismen in der Praxis umgesetzt werden, unterscheidet sich zwischen den hier untersuchten Varianten jedoch deutlich. Dieser Abschnitt soll die vier Laufzeitsysteme, also Java Platform Threads als Vertreter des klassischen 1:1-Modells, Node.js als Vertreter der ereignisgetriebenen Architektur sowie die Go-Runtime und Java Virtual Threads als zwei unabhängig entstandene Implementierungen des M:N-Modells, beschreiben. Im Mittelpunkt steht dabei die konkrete Realisierung jedes Modells in der Laufzeitumgebung bzw. der Sprachimplementierung. Da die spätere Untersuchung systemnah unterhalb der Laufzeitumgebung ansetzt, sind insbesondere die Stellen relevant, an denen die jeweilige Laufzeitumgebung mit dem Betriebssystem-Kernel interagiert, also die Erzeugung von Threads, die Behandlung blockierender Aufrufe sowie die Verteilung des Workloads auf die verfügbaren CPU-Kerne.

2.5.1 Das Thread-per-Request-Modell und Java Platform Threads

Das Thread-per-Request-Modell bildet jede eingehende Aufgabe auf einen eigenen Betriebssystem-Thread ab und stellt damit das weitverbreitetste Modell serverseitiger Nebenläufigkeit dar [1, 2]. In Java wird dieses Modell durch die Klasse `java.lang.Thread` realisiert. Lasić et al. charakterisieren diese Klasse als dünne Wrapper-Schnittstelle um die vom Betriebssystem verwalteten Threads, sodass die Effizienz von Java-Threads eng mit der Effizienz des Kernel-Schedulers verbunden ist [37]. Auf Linux Systemen ist der unterliegende Kernel-Thread durch die NPTL-Implementierung der glibc realisiert, die das 1:1-Mapping zwischen Anwendungs- und Kernel-Thread standardisiert hat [1, 26]. Jeder mit `new Thread().start()` erzeugte Plattform-Thread entspricht damit einem vom Kernel verwalteten Scheduling-Objekt, das der EEVDF-Scheduler unabhängig auf einen verfügbaren CPU-Kern legt und bei Bedarf präemptiv unterbricht.

Aus dieser Architektur ergeben sich einige Auswirkungen. Da die JVM den Stack eines Plattform-Threads nicht selbst verwaltet, sondern diesen vollständig dem Betriebssystem überlässt, ist die Stack-Größe nicht dynamisch wachsend, sondern muss bei der Thread-Erzeugung festgelegt werden [1, 11]. Die JVM weist hierfür standardmäßig einen Bereich von einem Megabyte zu, der über die Kommandozeilenoption `-Xss` konfigurierbar ist [11]. Auch wenn der tatsächliche physische Speicherverbrauch durch die in Abschnitt 2.2.2 beschriebene Lazy-Allocation der Stack-Seiten geringer ausfallen kann, ist die Anzahl der gleichzeitig erzeugbaren Plattform-Threads dadurch bereits aus Speichergründen begrenzt [1].
Die zweite Konsequenz betrifft das Scheduling. Da der Kernel jeden Plattform-Thread unabhängig kennt und einplant, werden sämtliche Scheduling Entscheidungen über den Kernel getroffen. Blockiert ein Plattform-Thread auf einer I/O-Operation, etwa auf dem Lesen eines TCP-Sockets, führt der Kernel einen freiwilligen Kontextwechsel durch, versetzt den Thread in den Wartezustand und teilt dem CPU-Kern einem anderen lauffähigen Thread zu. Trifft das erwartete Ereignis ein, versetzt der Kernel den Thread zurück in die Run-Queue, woraufhin er im Rahmen einer nachfolgenden Scheduling-Runde erneut einem Kern zugewiesen wird [8, 21]. Jeder dieser Übergänge ist mit den in Abschnitt 2.2.2 beschriebenen direkten und indirekten Kosten verbunden. Wächst die Anzahl der Verbindungen, wächst proportional die Anzahl der vom Kernel zu verwaltenden Scheduling-Objekte und damit auch die Häufigkeit der Kontextwechsel. Lasić et al. weisen ergänzend darauf hin, dass dem Betriebssystem zudem die spezifischen Speicheranforderungen einzelner Java-Threads nicht bekannt sind, weshalb häufig unnötig große Stack-Bereiche reserviert werden [37].

In der Praxis wird die Erzeugung eines neuen Plattform-Threads pro Aufgabe selten direkt verwendet. Stattdessen kapselt das `ExecutorService`-Interface einen Pool wiederverwendbarer Threads, an den Aufgaben in Form von `Runnable`- oder `Callable`-Objekten übergeben werden [10]. Diese Indirektion vermeidet die wiederholten Kosten der Thread-Erzeugung, verändert die grundsätzliche Architektur jedoch nicht. Jeder Thread im Pool bleibt ein Plattform-Thread und blockiert bei einer I/O-Operation ebenso wie ein direkt erzeugter Thread, sodass die maximale

Anzahl gleichzeitig in Bearbeitung befindlicher Aufgaben durch die Pool-Größe begrenzt bleibt [10, 37].

Aus systemnaher Sicht erzeugt das Thread-per-Request-Modell mit Java Platform Threads damit das in der Literatur am ausführlichsten dokumentierte Lastprofil. Pro Verbindung existiert ein eigener vom Kernel verwalteter Thread, dessen Scheduling vollständig im Kernel stattfindet und dessen blockierende I/O-Aufrufe je Wartesituation mindestens einen Mode Switch und einen Kontextwechsel verursachen.

2.5.2 Das ereignisgetriebene Modell und Node.js

Node.js verfolgt einen entgegengesetzten Ansatz. Die Laufzeitumgebung führt JavaScript auf einem einzelnen Thread aus und gibt damit die in Abschnitt 2.1 eingeführte Annahme auf, dass Nebenläufigkeit über Threads abgebildet werden muss. Stattdessen werden alle ereignisbezogenen Wartesituationen über eine zentrale Schleife verwaltet, die als Event Loop bezeichnet wird und durch die zugrundeliegende C-Bibliothek `libuv` bereitgestellt wird [4, 13]. Die Trennung zwischen Sprachimplementierung und I/O-Subsystem ist hier deutlich zu sehen, da V8 ausschließlich die Ausführung des JavaScript-Codes übernimmt, während `libuv` sämtliche Interaktionen mit dem Betriebssystem kapselt [4].

Eine Iteration der Event Loop läuft in einer fest definierten Folge von Phasen ab, von denen jede eine eigene First In, First Out-Warteschlange für die in dieser Phase auszuführenden Callbacks besitzt [13]. Die Node.js-Dokumentation nennt folgende Phasen in dieser Reihenfolge [13].

- **timers**: Führt Callbacks aus, deren über `setTimeout` oder `setInterval` gesetzter Schwellwert abgelaufen ist.
- **pending callbacks**: Führt aus der vorhergehenden Iteration verschobene I/O-Callbacks aus, etwa für bestimmte TCP-Fehlerfälle.
- **idle, prepare**: Wird ausschließlich intern von Node.js verwendet.
- **poll**: Bestimmt zunächst, wie lange auf neue I/O-Ereignisse gewartet werden soll und arbeitet anschließend die in der Poll-Queue eingetragenen Callbacks ab. Diese Phase blockiert den Event-Loop-Thread im typischen Serverbetrieb im Aufruf von `epoll_wait` und stellt damit den zentralen Berührungspunkt mit dem Kernel dar.
- **check**: Führt Callbacks aus, die über `setImmediate` registriert wurden.
- **close callbacks**: Führt Aufräum-Callbacks für geschlossene Handles aus.

Ergänzend zu diesen Phasen existieren zwei Microtask Warteschlangen, die nicht Teil der Phasenstruktur sind, sondern zwischen je zwei Phasen geleert werden. Die eine umfasst die über `process.nextTick` registrierten Callbacks, die andere die Continuations aufgelöster JavaScript Promises [13]. Da diese Microtasks vor dem Übergang in die nächste Phase abgearbeitet werden,

können sie die übrige Verarbeitung verzögern, wenn sie selbst rechenintensive Operationen enthalten.

Auf der Systemebene unterscheidet `libuv` bei der Behandlung von I/O-Operationen zwei grundsätzlich verschiedene Pfade. Netzwerk-I/O, also der Zugriff auf TCP- und UDP-Sockets, wird über die in Abschnitt 2.4 beschriebenen ereignisbasierten Schnittstellen abgewickelt [2, 4]. In diesem Pfad existiert kein Pool zusätzlicher Threads. Der einzelne Event-Loop-Thread blockiert in der Poll-Phase auf `epoll_wait` und arbeitet anschließend alle bereiten Deskriptoren der Reihe nach ab. Tausende gleichzeitige Verbindungen werden auf diesem Wege durch einen einzelnen Betriebssystem-Thread bedient, ohne dass pro Verbindung ein eigenes Scheduling-Objekt im Kernel entsteht [4].
Reguläre Dateizugriffe stellen jedoch einen Sonderfall dar, da `epoll` für sie ungeeignet ist, weil sich die Bereitschaft regulärer Dateien nicht sinnvoll abfragen lässt [4, 36]. Für diese Klasse von Operationen verwaltet `libuv` einen separaten Thread-Pool, in den blockierende Aufrufe ausgelagert werden, ohne den Event-Loop-Thread zu beeinträchtigen [38]. Über diesen Pool werden sämtliche Dateisystemoperationen, `getaddrinfo`- und `getnameinfo`-Anfragen sowie ausgewählte Crypto- und Zlib-Operationen abgewickelt [38, 39]. Die Standardgröße des Pools beträgt vier Threads und kann über die Umgebungsvariable `UV_THREADPOOL_SIZE` bis auf 1024 erhöht werden [38]. Aus systemnaher Sicht ist diese Trennung sehr relevant, da reine Netzwerk-Lasten ohne Datei-Beteiligung mit einem einzigen Kernel-Thread auskommen, gemischte Lasten mit Dateizugriffen hingegen die volle Mode-Switch- und Kontextwechsel-Methodik eines kleinen Thread-Pools aufweisen.

Relevant für das Verständnis des Modells ist, dass das Scheduling der einzelnen Callbacks vollständig im User Space stattfindet. Innerhalb einer Phase werden die in der zugehörigen Warteschlange enthaltenen Callbacks der Reihe nach auf demselben Thread ausgeführt, ohne dass zwischen ihnen ein Kontextwechsel des Kernels stattfindet [4, 13]. Eine notwendige Folge dieser Architektur ist, dass ein einzelner langer Callback alle weiteren Phasen und Callbacks zurückhalten kann. Solange der Event-Loop-Thread mit der Ausführung eines CPU-intensiven Callbacks beschäftigt ist, kann er weder neue I/O-Ereignisse aus dem Kernel entgegennehmen noch weitere bereits eingereihte Callbacks abarbeiten [4, 13].

2.5.3 Goroutinen und die Go-Runtime

Die Go-Runtime realisiert das M:N-Modell über eine Architektur, die als G-M-P-Modell bezeichnet wird [2, 40]. Die drei Buchstaben bezeichnen drei Datenstrukturen, deren Zusammenspiel das Scheduling der Goroutinen vollständig im User Space organisiert.

- Eine Goroutine (G) ist die Ausführungseinheit auf Anwendungsebene und enthält ihren Stack, den aktuellen Befehlszeiger sowie die übrigen Zustandsinformationen, die zur Wiederaufnahme der Ausführung erforderlich sind [40].

- Eine Machine (M) repräsentiert einen Betriebssystem-Thread, auf dem Go-Code tatsächlich ausgeführt wird und steht damit für eine 1:1-Abbildung auf einen Kernel-Thread [40].
- Ein Processor (P) ist eine reine Abstraktion der Go-Runtime und bündelt die Ressourcen, die eine Machine zur Ausführung von Go-Code benötigt, insbesondere eine lokale Run-Queue lauffähiger Goroutinen [40]. Eine Machine kann nur dann Go-Code ausführen, wenn sie an einen Processor gebunden ist, sodass die Anzahl der Processor die maximale Parallelität der Anwendung festlegt. Diese Anzahl ist über die Variable `GOMAXPROCS` konfigurierbar und entspricht standardmäßig der Anzahl der dem Prozess zur Verfügung stehenden logischen CPU-Kerne, also der Hardware-Threads inklusive Simultaneous Multithreading (SMT) bzw. Hyperthreading [41].

Eine Machine kann nur dann Go-Code ausführen, wenn ihr ein Processor zugeordnet ist. Über die Anzahl der Processor wird auf diese Weise die maximale Anzahl gleichzeitig Go-Code ausführender Threads begrenzt, unabhängig davon, wie viele Goroutinen oder Machines insgesamt existieren [41]. Lauffähige Goroutinen werden zunächst in die lokale Run-Queue desjenigen Processor eingereiht, auf dem sie erzeugt wurden. Reicht der Platz in der lokalen Queue nicht aus, werden überzählige Goroutinen in eine über alle Processor geteilte globale Run-Queue verschoben [40].
Hat ein Processor seine lokale Queue abgearbeitet, prüft er zunächst die globale Queue und greift anschließend auf einen Mechanismus zurück, der als Work Stealing bezeichnet wird. Dabei sucht der Processor in den lokalen Queues anderer Processor nach lauffähigen Goroutinen und übernimmt typischerweise die Hälfte der dort gefundenen Einträge in die eigene Queue [40]. Dieses Verfahren verteilt die Last ohne zentrale Koordination über die verfügbaren Processor und vermeidet die in Abschnitt 2.2.2 beschriebene Verschiebung einzelner Aufgaben durch den Kernel.

Für Netzwerk-I/O unterhält die Runtime eine eigene Komponente, die als Netpoller bezeichnet wird und intern die in Abschnitt 2.4 beschriebenen Schnittstellen `epoll`, `kqueue` oder IOCP verwendet. Ruft eine Goroutine eine vermeintlich blockierende Netzwerkoperation auf, übersetzt die Runtime diese intern in einen nicht blockierenden Systemaufruf, parkt die Goroutine und registriert den Dateideskriptor im Netpoller, sodass die tragende Machine sofort eine andere Goroutine ausführen kann [40]. Wird der Dateideskriptor später als bereit gemeldet, hebt der Netpoller das Parken der Goroutine auf und gibt sie in eine Run-Queue zurück. Aus Sicht der Entwickler:innen bleibt der Aufruf sequentiell und scheinbar blockierend, ohne dass ein vollständiger Kontextwechsel auf Kernel-Ebene erforderlich wäre.

Anders verhält sich die Runtime bei echten blockierenden Systemaufrufen, also etwa bei regulärem Datei-I/O oder bei Aufrufen über `cgo`. In diesem Fall blockiert die tragende Machine zwangsläufig im Kernel. Die Runtime erkennt diesen Zustand, koppelt den Processor von der blockierten Machine ab und übergibt ihn an eine andere, gegebenenfalls neu erzeugte Machine,

sodass die übrigen Goroutinen dieses Processor weiterlaufen können. Beendet sich der blockierende Systemaufruf, versucht die ursprüngliche Machine, einen neuen Processor zu erwerben und stellt die zugehörige Goroutine ansonsten in die globale Run-Queue zurück [40].

Auf der Ebene des Speicherverbrauchs ist Go sparsam. Eine neu erzeugte Goroutine erhält einen initialen Stack von lediglich 2 KB. Dieser Startwert wurde mit Go 1.4 von ursprünglich 8192 Bytes auf 2048 Bytes gesenkt, was die gleichzeitig eingeführte Umstellung auf zusammenhängende, kopierende Stacks erst ermöglichte. Da kein Stack-Segment mehr zwischen alten und neuen Bereichen wechselt, entfällt der zuvor notwendige Sicherheitspuffer, der einen größeren Mindeststapel erzwungen hatte [42]. Der Stack wird vom Runtime-Garbage-Collector verwaltet und wächst bei Bedarf durch Kopieren des gesamten Inhalts auf einen größeren zusammenhängenden Bereich [43]. Ab Go 1.19 kann die Runtime den initialen Stack einer neuen Goroutine darüber hinaus adaptiv auf Basis der beobachteten durchschnittlichen Stack-Nutzung gleichartiger Goroutinen anpassen, sodass der tatsächliche Startwert in laufenden Programmen vom Mindestwert von 2 KB nach oben abweichen kann [44]. Diese Konstruktion unterscheidet sich grundlegend vom vorab reservierten Stack eines Plattform-Threads und macht es praktisch, hunderttausende oder gar Millionen Goroutinen innerhalb eines einzelnen Prozesses zu nutzen [43].

Hinsichtlich der Präemption arbeitete der Go-Scheduler bis einschließlich Version 1.13 vollständig kooperativ. Eine Goroutine konnte ausschließlich an den vom Compiler eingefügten sog. Safepoints unterbrochen werden, also typischerweise bei Funktionsaufrufen, bei Channel-Operationen oder bei expliziten Aufrufen von `runtime.Gosched` [45]. Mit Go 1.14 wurde diese Beschränkung durch eine asynchrone, signalbasierte Präemption ergänzt, die einer dauerhaft rechnenden Goroutine über ein `SIGURG`-Signal mitteilt, dass sie ihre Ausführung an einer geeigneten Stelle unterbrechen soll [45, 46]. Diese Erweiterung verhindert, dass eine einzige CPU-intensive Goroutine die tragende Machine und damit einen Processor dauerhaft blockiert, ohne dass die übrigen Goroutinen Fortschritt machen.

2.5.4 Virtual Threads und Project Loom

Mit Project Loom und der in Java 21 finalisierten JDK Enhancement Proposal 444 hat die OpenJDK-Plattform ebenfalls eine M:N-Architektur eingeführt, allerdings auf einem grundlegend anderen Konstruktionsprinzip als die Go-Runtime [1]. Ein Virtual Thread ist weiterhin eine Instanz von `java.lang.Thread` und verhält sich aus Sicht der Anwendungsentwicklung wie ein gewöhnlicher Thread, läuft jedoch nicht direkt auf einem Kernel-Thread, sondern wird von einem Pool aus Plattform-Threads ausgeführt, die in diesem Kontext als Carrier-Threads bezeichnet werden [1, 10]. Diese Carrier-Threads sind ihrerseits gewöhnliche Plattform-Threads und damit dünne Wrapper um Kernel-Threads im Sinne von Abschnitt 2.5.1.

Der zugrundeliegende Träger ist ein dedizierter `ForkJoinPool`, der im First In, First Out-Modus betrieben wird. Die Anzahl der Carrier-Threads in diesem Pool entspricht standardmäßig

der Anzahl der verfügbaren CPU-Kerne und ist über die Eigenschaft `jdk.virtualThreadScheduler.parallelism` konfigurierbar [10].

Technisch beruht die M:N-Abbildung auf dem in der JVM neu eingeführten Mechanismus der Continuation. Eine Continuation kapselt den vollständigen Ausführungszustand einer Methode, also die Stack-Frames sowie die zugehörigen Register und erlaubt es der JVM, diesen Zustand zu speichern und zu einem späteren Zeitpunkt wieder fortzusetzen [1, 37]. Ein Virtual Thread ist damit im Wesentlichen eine Continuation zusammen mit den üblichen Thread-Metadaten. Die Stack-Frames eines Virtual Thread werden nicht in einem vorab reservierten Bereich gehalten, sondern als Objekte auf dem von der JVM verwalteten Heap, was die unter JDK Enhancement Proposal 444 dokumentierte erhebliche Reduktion des Speicherbedarfs pro Thread ermöglicht [1, 10].

Der Lebenszyklus eines Virtual Thread folgt einem festen Ablauf. Beim Start wird der Virtual Thread in eine zentrale Warteschlange eingereiht und gemäß First In, First Out-Reihenfolge einem freien Carrier-Thread zugewiesen [10]. Diesen Vorgang nennt man *Mount*. Anschließend führt der Carrier-Thread den Code des Virtual Thread aus, als wäre dieser ein gewöhnlicher Plattform-Thread. Erreicht der Code eine blockierende Operation, etwa einen Lesezugriff auf einen Socket, übersetzt die JVM diesen intern in eine nicht blockierende Operation und entfernt den Virtual Thread vom Carrier-Thread, was als *Unmount* bezeichnet wird [1, 10]. Die Continuation wird dabei in den Heap geschrieben, der Carrier-Thread steht unmittelbar für andere wartende Virtual Threads zur Verfügung und der ursprüngliche Virtual Thread wird erst dann erneut einem Carrier zugewiesen, wenn das erwartete Ereignis eingetreten ist. Auf Systemebene entspricht dieses Modell damit weitgehend dem Vorgehen der Go-Runtime, allerdings mit dem Unterschied, dass die Trägerarchitektur ein bestehender `ForkJoinPool` und kein eigens für diesen Zweck entworfener Scheduler ist.

JDK Enhancement Proposal 444 unterscheidet zwei Klassen von Situationen, in denen ein Virtual Thread seinen Carrier-Thread blockiert, ohne unmounten zu können. Die erste Klasse, im Folgenden als Carrier-Capture bezeichnet, umfasst Operationen, bei denen Einschränkungen auf Betriebssystem- oder JDK-Implementierungsebene ein Unmount verhindern, darunter Aufrufe von `Object.wait` sowie bestimmte Dateisystemoperationen. In diesen Fällen blockieren Virtual Thread und Carrier-Thread gemeinsam, bis das erwartete Ereignis eintritt. Die JVM kompensiert diesen Effekt, indem sie vorübergehend zusätzliche Carrier-Threads in den Pool aufnimmt und so sicherstellt, dass die übrigen wartenden Virtual Threads weiter ausgeführt werden können. Die maximale Anzahl temporär erzeugbarer Carrier-Threads ist über die Eigenschaft `jdk.virtualThreadScheduler.maxPoolSize` konfigurierbar und beträgt standardmäßig 256 [1, 10].

Die zweite Klasse ist das eigentliche Pinning. Es tritt in zwei Szenarien auf. Zum einen wenn ein Virtual Thread Code innerhalb eines `synchronized`-Blocks oder einer `synchronized`-Methode ausführt zum anderen bei Aufrufen über das Java Native Interface in nativen Code [1, 10]. Pinning unterscheidet sich von Carrier-Capture darin, dass der Scheduler in diesem

Fall keine Kompensation durch zusätzliche Carrier-Threads vornimmt. Pinnt ein Virtual Thread während einer blockierenden Operation, belegt er seinen Carrier-Thread für die gesamte Blockierdauer und reduziert damit die effektiv verfügbare Parallelität des Pools dauerhaft. In Anwendungen mit häufiger Verwendung von `synchronized`-Konstruktionen kann dies die Skalierbarkeit erheblich beeinträchtigen [10]. Mit JDK Enhancement Proposal 491 wurde diese Einschränkung für `synchronized`-Blöcke weitgehend aufgehoben, sodass auch Operationen innerhalb solcher Blöcke ein reguläres Unmount erlauben. Pinning durch nativen Code bleibt jedoch bestehen [12].

Aus systemnaher Sicht erzeugt Project Loom damit ein Lastprofil, das dem der Go-Runtime nahekommt, jedoch eine andere Implementierungsrichtung verfolgt. Während Go einen eigens entwickelten Scheduler mit lokalen Run-Queues und Work Stealing einsetzt, baut Project Loom auf einer bestehenden Plattform-Komponente auf und überträgt das Multiplexen an die ohnehin in der JVM vorhandene `ForkJoinPool`-Infrastruktur. Beide Wege erreichen das gemeinsame Ziel, blockierende Operationen ohne proportionale Inanspruchnahme von Kernel-Threads zu unterstützen, unterscheiden sich aber in den Details der Stack-Verwaltung, der Präemption und der verbleibenden Stellen, an denen sich die Abstraktion bis auf die Systemebene durchschlägt. Genau diese Unterschiede sind es, deren empirische Auswirkungen die spätere Untersuchung sichtbar machen soll.

3 Versuchsaufbau & Reproduzierbarkeit

Dieses Kapitel überträgt die in den vorangegangenen Kapiteln gelegten Grundlagen in einen konkreten Versuchsaufbau. Ziel ist eine Messumgebung, in der sich die in Kapitel 2 beschriebenen systemnahen Effekte der vier Laufzeitsysteme realitätsnah und wiederholbar messen lassen, ohne dass andere Einflüsse das Ergebnis verzerren. Dabei gilt der Grundsatz von Gregg, dass eine Messung nur dann aussagekräftig ist, wenn möglichst wenige Faktoren gleichzeitig verändert werden und jede Messung nur eine Frage beantwortet [9]. Abschnitt 3.1 beschreibt zuerst das Testsystem und seine Konfiguration einschließlich der Einstellungen, die für wiederholbare Messungen nötig sind. Abschnitt 3.2 erklärt anschließend, worauf beim Entwurf der Testprogramme für die drei Lastklassen zu achten ist. Abschnitt 3.3 stellt schließlich die Messverfahren und Werkzeuge vor, mit denen die Kennzahlen erhoben werden. Darauf aufbauend wird die konkrete Umsetzung der Testprogramme im Abschnitt 3.4 dargestellt und erläutert. Abschließend wird beschrieben, welche Kennzahlen auf welchem Weg erfasst und in welchem Format sie ausgegeben werden.

3.1 Testsystem und Umgebungskonfiguration

Die lokalen Lasttests, also die rechengebundene, die dateibasierte und die gemischte Last, laufen auf einem einzelnen, dedizierten Server vom Typ Lenovo ThinkSystem SR650. Die Beschränkung auf eine Maschine folgt der in Abschnitt 2.1 und 2.3 dargestellten Überlegung, dass sich das Zusammenspiel von Laufzeitumgebung, Betriebssystem und CPU-Architektur nur dann sauber beurteilen lässt, wenn keine Netzwerk- und Verteilungseffekte hinzukommen. Das System ist mit zwei Prozessoren vom Typ Intel Xeon Silver 4110 der Skylake-SP-Generation bestückt. Jeder Prozessor hat acht physische Kerne und stellt über SMT jeweils sechzehn logische Hardware-Threads bereit, sodass dem Betriebssystem insgesamt sechzehn physische und zweiunddreißig logische Kerne zur Verfügung stehen. Der Basistakt liegt bei 2,10 GHz, der maximale Turbotakt bei 3,0 GHz. Die Cache-Hierarchie folgt dem in Abschnitt 2.3 beschriebenen dreistufigen Aufbau. Jeder Kern hat einen eigenen, in Daten- und Instruktionsteil aufgeteilten L1-Cache und einen eigenen L2-Cache von einem Megabyte, während der L3-Cache mit elf Megabyte von allen Kernen eines Sockels gemeinsam genutzt wird. Der Arbeitsspeicher umfasst 128 Gigabyte registrierten DDR4-2666-Speicher mit Fehlerkorrektur, der auf beide Sockel verteilt ist, sodass das System die in Abschnitt 2.3 eingeführte NUMA-Architekur mit zwei Knoten besitzt. Das ist für die Untersuchung wichtig, weil sich nur dadurch CPU-Migrationen zwischen Kernen und Zugriffe über Knotengrenzen hinweg beobachten lassen [35]. Als Massenspeicher dient ein, über einen RAID-Controller, angebundener Solid-State-Datenträger mit ext4-Dateisystem, die Netzwerkanbindung erfolgt über Gigabit-Ethernet.
Als Betriebssystem dient Debian GNU/Linux 13 (Trixie) mit einem Kernel der Version 6.12. Dieser Kernel nutzt den in Abschnitt 2.2.3 beschriebenen EEVDF als Standardscheduler. Die Laufzeitsysteme liegen in folgenden Versionen vor, OpenJDK in Version 25.0.2, die Go-Runtime

in Version 1.24.4 und Node.js in Version 20.19.2. Diese Versionen sind nicht beliebig gewählt, sondern hängen an den in Kapitel 2 beschriebenen Eigenschaften. OpenJDK 25 enthält die in JDK Enhancement Proposal 444 finalisierten Virtual Threads und zusätzlich die mit JDK Enhancement Proposal 491 in OpenJDK 24 ausgelieferte Neuimplementierung von `synchronized`, durch die das in Abschnitt 2.5.4 beschriebene Pinning über `synchronized`-Blöcke weitgehend wegfällt [1, 12]. Das ist bei der Bewertung von Hypothese H4 zu beachten, da die früher wichtigste Ursache für Pinning in dieser Version nicht mehr greift und verbleibende Kosten daher anderen Ursachen zuzuordnen sind. Die Go-Runtime 1.24 besitzt die seit Version 1.14 vorhandene asynchrone, signalbasierte Präemption und die seit Version 1.19 mögliche adaptive Anpassung der anfänglichen Stack-Größe [43, 45]. Node.js 20 ist eine Version mit Langzeitunterstützung und baut auf dem in Abschnitt 2.5.2 beschriebenen, auf `libuv` aufsetzenden Event-Loop-Modell auf [4].

Neben der Hardware ist die Konfiguration der Umgebung für die Aussagekraft der Messungen wichtig. Ein zentraler Punkt ist der variable CPU-Takt. Moderne Prozessoren passen ihre Taktfrequenz an die aktuelle Last und an die Temperatur an, sodass dieselbe Last je nach den genannten Umgebungsbedingungen mit unterschiedlicher Frequenz läuft. Bliebe diese Anpassung aktiv, ließe sich nicht mehr sagen, ob ein Unterschied vom Nebenläufigkeitsmodell oder nur von einem anderen Taktzustand kommt. Deshalb wird der CPU-Frequenzregler des Kernels für alle Experimente auf den Governor `performance` gesetzt, der die Kerne fest auf die höchste verfügbare Frequenz bindet und die dynamische Anpassung abschaltet [47]. Zusätzlich wird der Turbotakt deaktiviert, damit die Kerne während der gesamten Messung mit einer festen Frequenz arbeiten. Das SMT bleibt über alle Läufe gleich eingestellt, sodass die Standardparallelität der Laufzeitsysteme für alle Kandidaten auf dieselben logischen Kerne trifft [41].
Bewusst wird darauf verzichtet, die Threads fest an einzelne Kerne oder NUMA-Knoten zu binden. Stattdessen überlässt der Aufbau die Verteilung über die Maschine dem Scheduler. Damit wird ein realistisches Szenario nachgebildet, in dem eine Anwendung in der Regel keine feste Bindung vorgibt. Wie stark der Scheduler die Threads zwischen Kernen und NUMA-Knoten verschiebt, ist deshalb kein Störfaktor, den es zu vermeiden gilt, sondern Teil des untersuchten Verhaltens und wird über die in Abschnitt 3.3 beschriebenen Kennzahlen mit erfasst.

Schließlich ist zu beachten, dass ein Serverbetriebssystem nie völlig untätig ist, sondern immer Hintergrundprozesse wie `systemd`, Kernel-Threads und Verwaltungsdienste ausführt. Für eine saubere Messung kommt es nicht darauf an, diese Aktivität ganz zu vermeiden, sondern darauf, dass sie über die Läufe gleich bleibt und nicht mal stärker und mal schwächer in einzelne Messungen hineinwirkt. Dazu wird das System vor jeder Messreihe in einen definierten Leerlauf gebracht, nicht benötigte regelmäßige Aufgaben wie Paketaktualisierungen, Indexierungsdienste und geplante Wartungsjobs werden abgeschaltet und es wird geprüft, dass keine weiteren Sitzungen auf dem System laufen. Wie wichtig das ist, zeigen Mytkowicz et al., die nachweisen, dass schon scheinbar nebensächliche Eigenschaften der Umgebung die Messergebnisse spürbar verfälschen können [48].

3.2 Entwurf der Testprogramme

Beim Entwurf der Testprogramme geht es nicht um deren konkrete Implementierung, sondern um die Frage, welche Eigenschaften ein Messprogramm haben muss, damit es wirklich den Einfluss des Nebenläufigkeitsmodells zeigt und nicht zufällige Eigenheiten der Sprache, des Compilers oder der Hardware. Einige Anforderungen gelten für alle Lastklassen, andere ergeben sich aus der jeweils belasteten Ressource, also aus der CPU-Last, der dateibasierten I/O-Last, der aus CPU- und I/O-Last gemischten Last und der netzwerkbasierten I/O-Last

Für alle Lasten muss zuerst sichergestellt sein, dass der Workload in allen vier Laufzeitsystemen vergleichbar ist. Da Go, die beiden Java-Varianten und Node.js unterschiedliche Compiler beziehungsweise JIT-Compiler verwenden, dürfen nur Operationen genutzt werden, die in allen Sprachen gleich definiert sind, etwa eine Ganzzahlarithmetik auf vorzeichenlosen 32-Bit-Werten mit festgelegtem Überlaufverhalten. Solche Operationen laufen auf der Hardware bitgenau gleich ab und geben keiner Sprache einen Vorteil, sodass Unterschiede dem Modell und nicht einem sprachspezifischen Algorithmus zuzuschreiben sind. Außerdem muss verhindert werden, dass die Compiler Teile der Arbeit wegoptimieren. Erkennt ein Compiler oder JIT-Compiler, dass das Ergebnis einer Berechnung nie verwendet wird, darf er sie als Dead Code entfernen, sodass nicht die Last selbst, sondern deren Wegfall gemessen würde. Die Programme müssen daher so gebaut sein, dass ihr Ergebnis am Ende zwingend ausgegeben oder weiterverwendet wird, etwa über eine fortlaufend gebildete Prüfsumme. Eine solche Prüfsumme kann zusätzlich dazu verwendet werden, die bitgenaue Abarbeitung des Workloads in allen Laufzeitsystemen zu prüfen. Ist die Prüfsumme in allen getesteten Laufzeitmodellen, bei gleichen Parametern identisch, dann ist sichergestellt, dass Überlaufverhalten, Algorithmik und Shift-Operationen identisch vorliegen. [48, 49].

Schließlich muss die Initialisierung der Testprogramme berücksichtigt werden. Beim Start entstehen einmalige Kosten, da unter anderem die Threads angelegt sowie Heap und Stack eingerichtet werden müssen. Hinzu kommt, dass die JIT-Kompilierung der JVM erst nach einer Aufwärmphase einen stabilen Zustand erreicht, was bei den JIT-basierten Laufzeitsystemen deutlich stärker ins Gewicht fällt als bei der vorab übersetzten Go-Runtime [49, 50]. Da in dieser Arbeit jeweils der gesamte Prozess samt Initialisierung gemessen wird, laufen die Testprogramme bewusst über einen möglichst langen Zeitraum. Die Anlaufkosten fallen dann zwar weiterhin an, machen aber nur noch einen kleinen Teil der über die Laufzeit gesammelten Messwerte aus, sodass die Messung im Wesentlichen das eingeschwungene Verhalten der Laufzeitsysteme zeigt. Die lange Laufzeit sorgt außerdem dafür, dass die größere Aufwärmphase der JIT-basierten Systeme den Vergleich nicht zugunsten der vorab übersetzten Go-Runtime verzerrt.

Die Alternative, die Messung erst nachträglich an einen schon laufenden Prozess zu hängen, hat mehrere Nachteile. Zum einen entsteht zwangsläufig eine Lücke zwischen dem Ende der Initialisierung und dem Start der Messung und damit eine nicht bestimmbare Ungenauigkeit.

Zum anderen müsste `perf` die Werte für jede Wiederholung einzeln erfassen, wodurch die automatische Zusammenfassung über mehrere Wiederholungen und weitere Auswertungsfunktionen wegfielen.

Für die rechengebundene Last ist wichtig, dass das Programm nicht nur die CPU, sondern auch den Weg zum Arbeitsspeicher belastet, weil gerade die Speicherzugriffe die in Abschnitt 2.3 beschriebenen indirekten Kosten von Kontextwechseln und Migrationen sichtbar machen. Eine reine Schleife auf wenigen Registern würde nur die Rechenleistung der Hardware messen und erfasst nicht das Verhalten von Scheduler und Cache. Damit die Speicherzugriffe die Latenzen der Speicherhierarchie wirklich zeigen, darf das Zugriffsmuster von der Hardware nicht vorhersagbar sein. Ein streng der Reihe nach durchlaufenes Feld würde von der Prefetch-Logik erkannt, die die Daten vorab in die schnellen Cache-Ebenen lädt und die zu messenden Latenzen damit verdeckt [32]. Sinnvoll ist daher ein festes, aber für den Prefetcher nicht vorhersagbares Zugriffsmuster, dessen Sprungweite sich zum Beispiel aus dem Goldenen Schnitt ableiten lässt, sodass der Prozessor die echten Latenzen des L3-Caches und des Hauptspeichers in Kauf nehmen muss. Genauso wichtig ist die Größe des Speicherbereichs, den jede Einheit nutzt, im Verhältnis zum L2-Cache eines Kerns. Ist der Bereich klein genug, bleibt er bei wenig Last im schnellen Cache, während er bei hoher Nebenläufigkeit durch die ständig wechselnden Einheiten verdrängt wird. So entsteht der in Abschnitt 2.3 beschriebene Effekt der gegenseitigen Cache-Verdrängung und die Unterschiede im Affinitäts- und Migrationsverhalten der Scheduler treten hervor. Diese Größe wird relativ zur L2-Kapazität des Testsystems von einem Megabyte gewählt und gezielt verändert. Bei dieser Last ist außerdem zu erwarten, dass das Single-Threaded-Event-Loop-Modell von Node.js keinen Vorteil hat und bei hoher Nebenläufigkeit hinter den parallel arbeitenden Modellen zurückbleibt, weil eine rechenintensive Operation die Verarbeitung weiterer Ereignisse aufhält. Genau das soll im Sinne von Hypothese H3 sichtbar werden [4, 39].

Für die dateibasierte Last ist die wichtigste Anforderung, dass die Zugriffe tatsächlich auf dem Massenspeicher blockieren und nicht schon aus dem Arbeitsspeicher beantwortet werden. Die blockierende Wartesituation wird dabei bewusst über Datei-I/O statt über Netzwerk-I/O erzeugt, da Letzteres einen zweiten, über das Netzwerk angebundenen Rechner erfordert hätte und damit zusätzliche, systemübergreifende Effekte in die Messung eingebracht hätte, die sich nicht mehr eindeutig dem untersuchten Nebenläufigkeitsmodell zuordnen ließen. Datei-I/O erzeugt dieselbe Grundsituation, nämlich einen blockierenden Wartezustand des ausführenden Threads, lässt sich aber vollständig auf dem Testsystem selbst und ohne zusätzliche Infrastruktur nachbilden. Das Betriebssystem hält einmal gelesene Dateiinhalte im Page-Cache, sodass wiederholte Zugriffe auf dieselben Daten ohne den Datenträger bedient werden [8]. Wäre der Bereich der Testdaten kleiner als der verfügbare Page-Cache, würde man nicht das I/O-Verhalten der Laufzeitumgebung messen, sondern die Geschwindigkeit des Caches. Der gesamte Datensatz darf daher nicht schon im Page-Cache sein, damit ein großer Teil der Zugriffe aus dem Massenspeicher beantwortet wird. Das ist gerade deshalb wichtig, weil sich reguläre Dateien nicht über die ereignisbasierten Schnittstellen auf Bereitschaft abfragen lassen und ein Lesezugriff auch dann blockieren kann, wenn die Schnittstelle die Datei als lesbar meldet,

die Seite aber nicht im Page-Cache liegt [36]. Genau hier unterscheiden sich die Modelle, denn Node.js lagert solche Zugriffe in den Thread-Pool von `libuv` aus, während die übrigen Modelle den ausführenden Kernel-Thread blockieren [2, 38]. Damit die Messung wiederholbar bleibt, wird der Cache vor jedem Lauf in einen definierten Ausgangszustand gebracht und die Dateigrößen sowie die Zugriffsmuster bleiben für alle Testmodelle gleich.

Die netzwerkbasierte Last ist anders angelegt als die beiden vorherigen. Sie zielt weniger auf die detaillierte Erfassung von Hardware-Zählern, sondern greift den in Abschnitt 2.1 beschriebenen historischen Kontext des C10k-Problems auf und prüft, wie die vier Modelle skalieren, wenn die Zahl der gleichzeitig offenen Verbindungen sehr groß wird. Im Mittelpunkt steht die Frage, wie viele gleichzeitige Verbindungen ein Modell verarbeitet und wie viele Anfragen es dabei pro Sekunde beantwortet.

Dazu beantwortet das Testsystem eingehende Anfragen mit einer einfachen, immer gleichen Antwort, während die Last von einer zweiten Maschine über das Netzwerk kommt. Auf dieser zweiten Maschine läuft ein Lastgenerator mit hoher Nebenläufigkeit. Der Generator öffnet eine große und schrittweise erhöhte Zahl gleichzeitiger Verbindungen und hält die Last, bis sich ein stabiler Zustand einstellt. Auf dem Testsystem werden dann die Zahl der gleichzeitig offenen TCP-Verbindungen, der Durchsatz an beantworteten Anfragen pro Sekunde und die Verteilung der Antwortzeiten erfasst, außerdem der Zustand des Systems wie die Zahl der vom Prozess gehaltenen Threads und die Aktivität des Schedulers. So wird sichtbar, wann das thread-basierte Modell an die in Abschnitt 2.5.1 beschriebene Grenze stößt und ab wann die ereignisgetriebenen und M:N-basierten Modelle ihren Vorteil ausspielen.

Weil die Last von einer eigenen Maschine kommt, konkurriert der Generator nicht um Rechenzeit, Cache und Speicherbandbreite des Testsystems und verfälscht die Messung dadurch nicht. Bei der Wahl des Generators ist zu beachten, ob er neue Anfragen erst nach Abschluss der vorherigen stellt oder unabhängig davon, da sich beide Varianten im Betrieb deutlich unterschiedlich verhalten und das Ergebnis beeinflussen. Klassische Closed-Loop-Generatoren wie `hey` messen Antwortzeiten nicht unter realen Bedingungen, weil ein langsamer Server den Anfragestrom drosselt, statt dass sich Anfragen stauen. Aus diesem Grund kommt der Closed-Loop-Generator `oha` mit Latenzkorrektur zum Einsatz, diese Latenzkorrektur kompensiert das Coordinated-Omission-Problem [51].

3.3 Messverfahren und Werkzeuge

Die rechengebundene, die dateibasierte und die gemischte Last werden im Kern auf zwei sich ergänzenden Ebenen erhoben, einer äußeren Ebene über Hardware- und Betriebssystemcounter und einer prozessinternen Ebene über das `/proc`-Dateisystem. Die netzwerkbasierte Last bildet demgegenüber einen eigenständigen Testfall mit eigenen Kennzahlen, die sich aus den Werten des externen Lastgenerators, etwa Durchsatz und Antwortzeiten, sowie aus prozessinternen Zählern über das `/proc`-Dateisystem zusammensetzen, jedoch ohne die Hardware-Counter von

`perf`. Dass mehrere Quellen genutzt werden, folgt der Empfehlung von Gregg, Kennzahlen nach Möglichkeit aus mehreren Quellen zu bestätigen, um einzelne Werte nicht falsch zu deuten [9]. Welche Größen im Einzelnen erhoben und wie sie den Hypothesen zugeordnet werden, ist Gegenstand der späteren Auswertung. An dieser Stelle steht das Vorgehen im Vordergrund und nicht die einzelne Messgröße.

Die äußere Ebene stützt sich auf das Werkzeug `perf`. Es führt die Testprogramme direkt aus und liest dabei über mehrere Wiederholungen sowohl Hardware-Performance-Counter als auch Software-Events des Kernels aus. Auf diese Weise lassen sich systemnahe Vorgänge erfassen, die der Anwendung selbst verborgen bleiben, etwa das Verhalten der Caches, des TLB und der Speicheranbindung sowie die Häufigkeit von Kontextwechseln und von Übergängen zwischen User- und Kernel-Mode. Über die reinen Counter hinaus erlaubt `perf` zusätzlich eine Analyse des Schedulings, also der Wartezeiten lauffähiger Threads und ihrer Verteilung über die CPU-Kerne. Das Werkzeug läuft dabei außerhalb der Testprogramme, sodass die Messung den gemessenen Code nicht verändert [14].

Die prozessinterne Ebene erhebt das Testprogramm selbst. Es liest die passenden Einträge des `/proc`-Dateisystems an zwei Punkten aus, einmal bevor der eigentliche Workload beginnt und einmal nachdem er beendet ist. Über diesen Weg werden Größen sichtbar, die `perf` nicht oder nur ungenau liefert, vor allem der vom Prozess belegte Speicher, die Zahl seiner Threads, die über alle Threads zusammengefassten Kontextwechsel, die Statistik des Kernel-Schedulers sowie die Verteilung der Speicherseiten über die NUMA-Knoten. Letztere ergibt sich erst aus der bewusst freien Platzierung der Threads und zeigt, wie stark der Scheduler die Seiten über die Knotengrenzen hinweg verschiebt. Ein Teil dieser Werte dient zugleich als Kontrolle, dass im Messfenster kein unerwünschter Zugriff auf Swap oder Datenträger das Ergebnis verfälscht hat. Ergänzend geben die Testprogramme selbst einige laufzeitinterne Größen aus, die sich weder über `perf` noch über `/proc` ablesen lassen, etwa die Zahl der aktiven Goroutinen beziehungsweise der Plattform- und Virtual Threads. Diese Angaben stammen aus den Standardschnittstellen der jeweiligen Laufzeitumgebung und werden vom Programm selbst abgerufen und ausgegeben. Laufzeitspezifische Profiler oder Analysewerkzeuge kommen dabei nicht zum Einsatz.

Die Antwortzeiten werden vollständig erfasst und nicht nur als Mittelwert, sondern als Verteilung ausgewertet, da gerade das Verhalten am oberen Rand der Verteilung für die in Abschnitt 2.1 genannten latenzsensitiven Anwendungsfelder entscheidend ist. Dass eine Verteilung der Latenz mehr aussagt als ein einzelner Mittelwert, ist in der Literatur ausführlich begründet [52]. Jede Konfiguration wird mehrfach wiederholt und neben dem typischen Wert werden auch die Schwankungen zwischen den Läufen angegeben, wobei die Zahl der Wiederholungen so gewählt wird, dass sich diese Schwankung abschätzen lässt. Da ein stabiler Zustand der JVM nicht immer einem festen Leistungsmaximum entspricht, wird er nicht vorausgesetzt, sondern anhand der Messreihen überprüft. Dieses Vorgehen folgt den gängigen Empfehlungen zur statistisch belastbaren Bewertung von Programmen auf virtuellen Maschinen [49, 50].

Damit sich die Experimente nachvollziehen lassen, wird der gesamte Ablauf so weit wie möglich über Skripte gesteuert und protokolliert. Zusammen mit der in Abschnitt 3.1 beschriebenen festen

Taktfrequenz und dem ruhig gehaltenen Hintergrund sowie den in Abschnitt 3.2 beschriebenen, in allen Sprachen gleich aufgebauten Programmen lassen sich die Messungen unter denselben Bedingungen wiederholen.

3.4 Die Testprogramme

Abschnitt 3.2 hat festgelegt, welche Eigenschaften ein Messprogramm besitzen muss, damit es den Einfluss des Nebenläufigkeitsmodells und nicht zufällige Eigenheiten von Sprache, Compiler oder Hardware zeigt. Dieser Abschnitt überträgt diese Anforderungen in konkrete Programme. Für jeden Baustein wird erklärt, was er zur Laufzeit tut, an welcher Stelle die jeweilige Last entsteht und welche Wirkung das auf das System hat.

Die Darstellung zeigt die Umsetzung in Go. Die Programme für die drei weiteren Laufzeitsysteme folgen demselben Aufbau und unterscheiden sich nur in sprachspezifischen Details, vor allem in der Art, wie die vorzeichenlose 32-Bit-Arithmetik mit festgelegtem Überlaufverhalten nachgebildet wird. Da sich die Implementierungen dadurch inhaltlich nicht unterscheiden, werden im Folgenden ausschließlich die Go-Programme im Detail gezeigt und erläutert, die vollständigen Implementierungen der drei übrigen Laufzeitsysteme finden sich im Anhang.

Die Funktionen und Variablen in den folgenden Listings sind bewusst sehr sprechend benannt, etwa `erzeugeZugriffskette` oder `pruefsummeGesamt`. Die Listings werden dadurch etwas länger als üblich, lassen sich aber leichter lesen, weil sich die Aufgabe jeder Funktion und jeder Variable bereits am Namen ablesen lässt. Alle Programme folgen demselben Grundgerüst. Zuerst liest das Programm seine Parameter von der Kommandozeile. Danach startet es viele gleichzeitige Ausführungseinheiten, die jeweils ihren Arbeitsbereich vorbereiten und dann an einer gemeinsamen Barriere warten, dem sog. Gate. Sind alle Einheiten bereit, erfasst das Programm einen ersten Messpunkt, öffnet das Gate und lässt damit alle Einheiten zeitgleich beginnen. Diese Zeitspanne bildet das Messfenster. Haben alle Einheiten ihre Arbeit abgeschlossen, erfasst das Programm einen zweiten Messpunkt. Aus dem Unterschied beider Messpunkte und aus den gemessenen Laufzeiten ergeben sich die Kennzahlen des Durchlaufs.

Die eigentliche Last steht im Vordergrund dieses Abschnitts. Die Erfassung der Messwerte ist in allen Programmen gleich und in eine eigene Quelldatei ausgelagert, die das jeweilige Programm nur an den beiden genannten Punkten aufruft. Wie in Abschnitt 3.3 dargestellt, laufen die Programme direkt unter `perf`, sodass der gesamte Prozess samt seiner Initialisierung gemessen wird.

3.4.1 Die rechengebundene Last

Das Programm für die rechengebundene Last beschäftigt die Prozessorkerne mit einer kurzen Schleife, die jede Goroutine sehr oft durchläuft. In jedem Durchlauf liest die Schleife einen Wert

aus einem Puffer im Arbeitsspeicher und führt mit diesem Wert einfache Rechenoperationen aus. Die Last entsteht aus der enormen Menge dieser Durchläufe. Bei vielen Goroutinen und vielen Millionen Schritten je Goroutine summieren sich die Durchläufe zu einer Vielzahl an Operationen, sodass die Kerne über die gesamte Messdauer ausgelastet sind. Ein relevanter Teil dieser Zeit soll dabei nicht auf das Rechnen, sondern auf die Latenz von Speicherzugriffen entfallen.

Die Ausführungseinheit ist die Goroutine. Das Programm startet viele Goroutinen, die alle denselben Workload unabhängig voneinander abarbeiten. Nach dem in Abschnitt 2.5.3 beschriebenen M:N-Modell verteilt die Go-Runtime diese Goroutinen auf eine kleinere Zahl von Betriebssystem-Threads. `GOMAXPROCS` begrenzt dabei, wie viele dieser Threads gleichzeitig Go-Code ausführen dürfen, legt also die maximale Parallelität fest. Sind mehr Goroutinen als Kerne aktiv, müssen sich mehrere von ihnen einen Kern teilen. Der Scheduler schaltet dann laufend zwischen ihnen um und verschiebt sie über die Kerne. In den Programmen der drei übrigen Laufzeitsysteme entspricht die Ausführungseinheit jeweils dem in Kapitel 2 beschriebenen Modell der jeweiligen Sprache, also Plattform-Threads bzw. Virtual Threads bei den beiden Java-Varianten und der Event-Loop bei Node.js und verhält sich entsprechend den dort erläuterten Eigenschaften.
Listing 3.1 zeigt die Parameter eines Laufs. `AnzahlAufgaben` ist die Zahl der gleichzeitigen Goroutinen. `PufferBytes` ist die Größe des privaten Puffers je Goroutine. `Schritte` ist die Zahl der Durchläufe, die jede Goroutine im Messfenster ausführt, `Gomaxprocs` hält den tatsächlich aktiven Wert von `GOMAXPROCS` fest und `AlsCsvAusgeben` steuert, ob das Programm nach dem Messlauf eine maschinenlesbare, Semikolon getrennte Zeile auf die Standardausgabe schreibt, mit der sich mehrere Läufe außerhalb des Programms zu einer Gesamttabelle zusammenfassen lassen.

```go
type Config struct {
  AnzahlAufgaben int // gleichzeitige Goroutinen
  PufferBytes    int // privater Puffer pro Aufgabe in Byte
  Schritte       int // Chase- bzw. Mix-Schritte pro Aufgabe
  Gomaxprocs     int // tatsächlich aktive GOMAXPROCS
  AlsCsvAusgeben bool // maschinenlesbare CSV-Zeile auf stdout
}
```

Listing 3.1: Parameter eines Laufs der rechengebundenen Last (Go)

Der Puffer jeder Goroutine enthält keine unabhängigen Datenwerte, sondern bildet eine Zugriffskette. Jedes Element speichert den Index des nächsten Elements, auf das zugegriffen werden soll. Während der Ausführung liest die Schleife zunächst den aktuellen Index und verwendet dessen Inhalt unmittelbar als Position des nächsten Speicherzugriffs. Da die Adresse des folgenden Zugriffs damit erst nach Abschluss des vorherigen Lesezugriffs bekannt ist, kann der Prozessor die Zugriffe nicht im Voraus parallel vorbereiten. Bei jedem Zugriff, dessen Zieldaten nicht im Cache liegen, muss der Kern anhalten und auf den langsameren Speicher warten. Dieses Verfahren wird als Pointer Chasing bezeichnet.

Listing 3.2 zeigt den Aufbau dieser Zugriffskette. Die Zugriffskette wird als zyklische Permutation der Pufferindizes implementiert, um eine möglichst gleichmäßige Traversierung des Puffers zu erreichen. Eine Permutation ist dabei eine Umordnung einer Menge, bei der jedes Element genau einmal vorkommt. Ausgehend vom Startindex besucht die Traversierung die Positionen in festen Schritten der Sprungweite, also nach dem Muster $0, s, 2s, 3s, \ldots$ modulo der Pufferlänge.
Entscheidend für den angestrebten Effekt ist, dass die Adresse jedes Zugriffs erst aus dem Ergebnis des unmittelbar vorausgehenden Lesezugriffs hervorgeht. Diese Datenabhängigkeit verhindert, dass der Prozessor mehrere Zugriffe derselben Kette überlappend ausführt. Der nächste Index ist erst bekannt, wenn der aktuelle Lesezugriff abgeschlossen ist, sodass die Out-of-Order-Ausführung nicht vorauslaufen kann und zu jedem Zeitpunkt höchstens ein Cache Miss der Kette offen ist. Jeder Zugriff, dessen Zieldaten nicht im Cache liegen, hält den Kern damit für die volle Latenz der nächsthöheren Speicherebene an, ohne dass diese Latenz durch nebenläufige Zugriffe überlagert werden kann [32].

```go
func groessterGemeinsamerTeiler(a, b int) int {
  for b != 0 {
    a, b = b, a%b
  }
  return a
}

func erzeugeZugriffskette(anzahlElemente int) []uint32 {
  if anzahlElemente < 16 {
    anzahlElemente = 16
  }

  sprungweite := int(float64(anzahlElemente) * 0.6180339887) // Goldener Schnitt
  if sprungweite < 1 {
    sprungweite = 1
  }
  for groessterGemeinsamerTeiler(sprungweite, anzahlElemente) != 1 {
    sprungweite++
  }

  kette := make([]uint32, anzahlElemente)
  for i := 0; i < anzahlElemente; i++ {
    kette[i] = uint32((i + sprungweite) % anzahlElemente)
  }
  return kette
}
```

Listing 3.2: Aufbau der Zugriffskette (Go)

Es ist dennoch möglich, dass der Hardware-Prefetcher die benötigten Cache Lines vorab lädt und die Latenz auf diesem Wege dennoch ausgeglichen wird. Ein Prefetcher sagt Adressen anhand des Befehlszeigers und der beobachteten Schrittweite voraus und ist dabei nicht an die Datenabhängigkeit der Kette gebunden. Die Wahl der Sprungweite zielt deshalb gezielt darauf ab, diese Vorhersage zu unterlaufen. Mit einer aus dem Goldenen Schnitt abgeleiteten Schrittweite von etwa $0{,}618$ der Pufferlänge liegen aufeinanderfolgende Zugriffe weit auseinander, landen verlässlich in unterschiedlichen Cache Lines und überschreiten bei den hier verwendeten Puffergrößen mit nahezu jedem Schritt eine Speicherseitengrenze. Der L2-Streamer, der benachbarte Cache Lines innerhalb einer Speicherseite vorab lädt, kann unter diesen Bedingungen nicht greifen, da weder eine räumliche Nachbarschaft der Zugriffe vorliegt noch aufeinanderfolgende Zugriffe in dieselbe Seite fallen.

Gegenüber einem ip-basierten Stride-Prefetcher, der eine konstante Schrittweite je Lade-Instruktion auch über Seitengrenzen hinweg erkennen kann, ist zusätzlich die modulare Arithmetik relevant. Da die besuchten Positionen dem Muster $k \cdot s \bmod n$ folgen, springt die tatsächliche Adresse je nachdem, ob die Summe die Pufferlänge überschreitet, um $+s$ oder um $s-n$ Elemente weiter. Im Adressraum liegt damit keine einzelne konstante Schrittweite vor, sondern eine zwischen zwei Werten wechselnde, quasi-periodische Folge, auf die sich eine Stride-Vorhersage nur unzuverlässig einstellen kann [33].

Der Goldene Schnitt sorgt darüber hinaus für eine gleichmäßige Verteilung der besuchten Positionen über den gesamten Puffer, sodass die Zugriffe zu jedem Zeitpunkt der Traversierung breit gestreut sind und nicht zunächst nur einen zusammenhängenden Teilbereich belegen. Damit der gesamte Puffer als Arbeitsbereich genutzt wird und die Zugriffskette nicht in mehrere kurze Teilzyklen zerfällt, muss die Sprungweite teilerfremd zur Pufferlänge sein. Ist diese Bedingung erfüllt, durchläuft die Kette alle Elemente des Puffers genau einmal, bevor sie zum Ausgangspunkt zurückkehrt. Andernfalls würde die Schleife nur eine kleine Teilmenge der Elemente immer wieder besuchen, die unter Umständen klein genug ist, um vollständig im schnellen L1- bzw. L2-Cache zu liegen.
Wie schnell ein einzelner Zugriff ist, hängt, wie bereits erwähnt, davon ab, ob das gesuchte Element im Cache liegt. Hier wird die Größe des Puffers relevant, die im Verhältnis zum L2-Cache eines Kerns gewählt werden muss, der auf dem Testsystem ein Megabyte groß ist (Abschnitt 3.1). Solange nur wenige Goroutinen laufen, hat jede ihren Kern weitgehend für sich. Ist ihr Puffer kleiner als der L2-Cache, liegt er nach dem ersten Durchlauf vollständig in diesem Cache und alle weiteren Zugriffe werden von dort bedient. Sobald der Puffer den Cache übersteigt, müssen Zugriffe aus dem langsameren L3-Cache oder dem Hauptspeicher geholt werden. Laufen dagegen viele Goroutinen gleichzeitig, teilen sich mehrere von ihnen einen Kern und damit denselben L2-Cache. Passen ihre Puffer zusammen nicht mehr in den Cache, verdrängen sie sich gegenseitig und auch Puffer, die einzeln noch in den Cache gepasst hätten, müssen häufiger aus langsameren Speicherebenen nachgeladen werden. Verschiebt der Scheduler eine Goroutine zusätzlich auf einen anderen Kern, sind ihre Daten weder im dortigen L1- noch im L2-Cache vorhanden und müssen neu geladen werden. Das ist der in Abschnitt 2.3 beschriebene Effekt der gegenseitigen Cache-Verdrängung.

Listing 3.3 zeigt die Schleife selbst. Die erste Zeile innerhalb der Schleife, in der der gelesene Wert sofort zur neuen Position wird, ist der beschriebene Pointer Chase und damit die Stelle, an der der Kern auf den Speicher wartet. Die beiden folgenden Zeilen verknüpfen den gelesenen Wert mit einer laufenden Prüfsumme. Sie halten die Recheneinheiten beschäftigt und sorgen dafür, dass der gelesene Wert tatsächlich verwendet wird.
Diese Prüfsumme erfüllt zwei Zwecke. Zum einen wird ihr Ergebnis am Ende ausgegeben. Dadurch hat die Schleife eine sichtbare Wirkung. Ohne eine solche Wirkung dürfte ein Compiler oder JIT-Compiler erkennen, dass das Ergebnis nie gebraucht wird und die ganze Schleife als wirkungslos entfernen. Dann würde nicht die Last gemessen, sondern ihr Wegfall [48, 49]. Zum anderen bestätigt die Prüfsumme, dass die Operationen in allen vier Laufzeitsystemen bitgenau gleich ablaufen, sodass ein gemessener Unterschied dem Nebenläufigkeitsmodell zuzuschreiben ist und nicht einem sprachspezifischen Rechenverhalten.

```go
func verarbeite(kette []uint32, schritte int) uint32 {
  var index, pruefsumme uint32
  for s := 0; s < schritte; s++ {
    index = kette[index]
    pruefsumme = (pruefsumme + index) * streuFaktor
    pruefsumme ^= pruefsumme >> 15
  }
  return pruefsumme
}
```

Listing 3.3: Die Messschleife der rechengebundenen Last (Go)

Listing 3.4 zeigt den Ablauf einer Messung. Zur Laufzeit werden zuerst alle Goroutinen gestartet. Jede erzeugt ihre private Zugriffskette. Anschließend meldet sich die Goroutine über die Wartegruppe `alleBereit` als bereit und wartet am Gate. Das Gate ist ein Channel, auf dessen Schließen alle Goroutinen warten. Das Schließen eines Channels weckt alle Wartenden gleichzeitig. Dadurch beginnen alle Goroutinen ihre Arbeit im selben Augenblick.

```go
func fuehreMesslaufAus(config Config) {
  anzahlElemente := config.PufferBytes / 4 // 4 Byte je uint32

  var (
    startSignal      = make(chan struct{})
    alleBereit       sync.WaitGroup
    alleFertig       sync.WaitGroup
    pruefsummeGesamt int64
    latenzen         = make([]time.Duration, config.AnzahlAufgaben)
    messbeginn       time.Time
  )

```

```
13    alleBereit.Add(config.AnzahlAufgaben)
14    alleFertig.Add(config.AnzahlAufgaben)
15
16    for id := 0; id < config.AnzahlAufgaben; id++ {
17      go func(id int) {
18        defer alleFertig.Done()
19        kette := erzeugeZugriffskette(anzahlElemente)
20
21        alleBereit.Done()
22        <-startSignal
23
24        pruefsumme := verarbeite(kette, config.Schritte)
25
26        latenzen[id] = time.Since(messbeginn)
27        atomic.AddInt64(&pruefsummeGesamt, int64(pruefsumme))
28      }(id)
29    }
30
31    alleBereit.Wait()
32    snapshotVorMessung := erfasseSnapshot()
33    messbeginn = time.Now()
34    close(startSignal)
35    alleFertig.Wait()
36    wallclock := time.Since(messbeginn)
37    snapshotNachMessung := erfasseSnapshot()
38
39    sort.Slice(latenzen, func(i, j int) bool { return latenzen[i] < latenzen[j] })
40
41    ergebnis := Laufergebnis{
42      Wallclock:      wallclock,
43      AnzahlAufgaben: config.AnzahlAufgaben,
44      Pruefsumme:     pruefsummeGesamt,
45      Latenzen:       latenzen,
46      Gomaxprocs:     config.Gomaxprocs,
47    }
48
49    if config.AlsCsvAusgeben {
50      druckeCsvZeile(config, snapshotVorMessung, snapshotNachMessung, ergebnis)
51    }
52  }
```

Listing 3.4: Ablauf eines Messlaufs der rechengebundenen Last (Go)

Weil alle Goroutinen denselben Startzeitpunkt haben, lässt sich für jede die Latenz von diesem Start bis zu ihrem Abschluss bestimmen. Diese Latenzen werden gesammelt und sortiert, sodass sie sich nicht nur als Mittelwert auswerten lassen, sondern als Verteilung mit kleinstem Wert, Median und oberen Perzentilen. Das ist relevant, weil gerade das Verhalten am oberen Rand für die in Abschnitt 2.1 genannten latenzsensitiven Anwendungen entscheidend ist. Das Aufsummieren der einzelnen Prüfsummen geschieht über eine atomare Operation, damit die gleichzeitig schreibenden Goroutinen keinen Konflikt erzeugen.

3.4.2 Die dateibasierte I/O-Last

Im Testfall für die dateibasierte Last führt das Programm viele gleichzeitige Goroutinen aus, die jeweils eine Zahl zufällig verteilter Lesezugriffe auf einen zuvor angelegten Datensatz ausführen. Die Last auf der Hardware entsteht im Warten auf den Datenträger. Jeder Lesezugriff, der nicht bereits aus dem Page-Cache des Betriebssystems bedient werden kann, zwingt die ausführende Einheit dazu, bis zum Abschluss der Hardwareoperation zu warten. Wie diese Wartesituation auf das Betriebssystem abgebildet wird, ist der Unterschied zwischen den vier Modellen und damit der Gegenstand dieser Messung. Die Programme der übrigen Laufzeitsysteme folgen demselben Aufbau und unterscheiden sich nur in sprachspezifischen Details, etwa in der Art, wie der Lesezugriff abgesetzt und die vorzeichenlose 32-Bit-Arithmetik der Prüfsumme nachgebildet wird.

Der Datensatz wird einmalig erzeugt und ist nicht Teil der Messung. Grund hierfür ist, dass das Anlegen der Dateien das Schreiben vieler Gigabyte bedeuten und wenn es in das Messfenster fiele, die einmaligen Schreibkosten mit dem zu untersuchenden Messlauf vermischen und den ersten Lauf von allen weiteren unterscheiden würde. Im Messmodus öffnet das Programm ausschließlich die bereits vorhandenen Dateien und bricht ab, falls der Datensatz fehlt.

Die wichtigste Anforderung an die dateibasierte Last ist, dass die Zugriffe tatsächlich den Datenträger erreichen und nicht aus dem Hauptspeicher geladen werden. Das Betriebssystem hält einmal gelesene Dateiinhalte im Page-Cache, aus diesem Grund leert das Programm den Page-Cache zu Beginn jedes Laufs, bevor es den Datensatz öffnet und in das Messfenster eintritt, sodass der erste Zugriff auf einen Block nicht im Arbeitsspeicher liegen kann. Da jeder Durchlauf als eigener Prozess gestartet wird und jede Wiederholung von `perf stat -r` einen neuen Prozessstart bedeutet, beginnt auf diese Weise jede Wiederholung mit einem geleerten Cache.
Das Leeren des Caches ist technisch ein Schreibzugriff auf die Kontrolldatei `/proc/sys/vm/drop_caches`, dem ein `sync` vorausgeht. Beides sind gewöhnliche Dateioperationen und keine besonderen Systemaufrufe, sodass alle vier Laufzeitsysteme den Cache am Prozessstart ohne ein zusätzliches Hilfsprogramm selbst leeren können. Lediglich für den globalen `sync` besitzen die JVM und Node.js keine eigene Standardschnittstelle, was für die rein lesende Last jedoch irrelevant ist, da der Kernel saubere Cache-Seiten auch ohne vorheriges `sync` verwirft und der Datensatz bereits beim Anlegen vollständig auf den Datenträger geschrieben wurde.

Da das Leeren zu Beginn des Prozesses und damit vor der ersten Momentaufnahme erfolgt, fließt es nicht in die vom Programm über das `/proc`-Dateisystem gebildeten Differenzen ein. Diese erfassen ausschließlich das Geschehen zwischen dem Gate und dem Ende des Messfensters. Die von `perf` erhobenen Zähler messen hingegen den gesamten Prozess und enthalten damit auch den geringen Aufwand des Cache-Leerens. Dieser Aufwand ist klein gegenüber der eigentlichen, vom Warten auf den Datenträger geprägten Last und fällt für alle vier Kandidaten in gleicher Weise an, sodass er den Vergleich zwischen den Modellen nicht in relevantem Ausmaß einschränkt.

Aus dem kalten Start ergibt sich auch die Wahl der Datensatzgröße. Damit die Zugriffe tatsächlich den Datenträger erreichen, muss der Datensatz deutlich größer als die je Messlauf gelesene Datenmenge sein. Wäre der Datensatz nur wenig größer, würde innerhalb eines einzelnen Laufs ein erheblicher Teil der Blöcke mehrfach besucht und beim zweiten Zugriff bereits aus dem inzwischen gefüllten Page-Cache bedient. Die je Lauf gelesene Datenmenge ergibt sich aus der Gesamtzahl der Lesezugriffe multipliziert mit der Blockgröße.

```go
type Config struct {
  AnzahlAufgaben     int    // gleichzeitige Goroutinen
  Verzeichnis        string // Ablageort des Datensatzes
  AnzahlDateien      int    // Anzahl der Dateien im Datensatz
  Dateigroesse       int64  // Größe je Datei in Byte
  Blockgroesse       int    // Größe je einzelnem Lesezugriff in Byte
  AnzahlLesezugriffe int    // abgeleitete Lesezugriffe pro Aufgabe
  Seed               int64  // Basis-Seed für das Zugriffsmuster
  Gomaxprocs         int    // tatsächlich aktive GOMAXPROCS
}
```

Listing 3.5: Parameter eines Laufs der dateibasierten Last (Go)

Da die Untersuchung die Nebenläufigkeit über die Aufgabenzahl skaliert, würde eine feste Zahl an Lesezugriffen je Aufgabe diese Datenmenge mit der Aufgabenzahl ansteigen lassen und damit auch den benötigten Datensatz proportional vergrößern. Um dies zu vermeiden, hält der Aufbau die Gesamtzahl der Lesezugriffe über alle Aufgaben konstant und leitet die Zahl der Lesezugriffe je Aufgabe daraus ab. Bei eintausend Aufgaben sind das zweitausend Lesezugriffe je Aufgabe, bei zehntausend Aufgaben entsprechend zweihundert. Die je Lauf gelesene Datenmenge bleibt damit auf jeder Nebenläufigkeitsstufe bei etwa 7,6 Gigabyte, unabhängig davon, wie viele Aufgaben diese Zugriffe gemeinsam ausführen. Damit ändert sich beim Skalieren der Aufgabenzahl allein die Zahl der gleichzeitig blockierenden Aufgaben und nicht die Datenmenge. Der Datensatz muss entsprechend nur einmal groß genug gewählt werden. Bei 64 Gigabyte, verteilt auf 64 Dateien zu je einem Gigabyte, bleibt der Anteil der innerhalb eines Messlaufes wiederholten Zugriffe im Bereich weniger Prozent, sodass nahezu jeder Zugriff den Datenträger erreicht. Dass die Zugriffe wirklich den Datenträger erreichen,

wird in jedem Lauf über das Verhältnis der vom Block-Device geholten Bytes zu den logisch gelesenen Bytes kontrolliert (Abschnitt 3.3).

Listing 3.6 zeigt das Erzeugen der Dateiinhalte. Ein einfacher Xorshift-Generator beschreibt den Schreibpuffer fortlaufend mit pseudozufälligen Bytes, deren Zustand zwischen den Blöcken weitergereicht wird. Aufeinanderfolgende Blöcke erhalten dadurch unterschiedlichen Inhalt, sodass das Dateisystem die Datei vollständig und ohne optimierten Speicherbereiche anlegt.

```go
func fuelleMitZufall(puffer []byte, zustand *uint64) {
  wert := *zustand
  if wert == 0 {
    wert = 0x9E3779B97F4A7C15
  }
  for i := 0; i+8 <= len(puffer); i += 8 {
    wert ^= wert << 13
    wert ^= wert >> 7
    wert ^= wert << 17
    binary.LittleEndian.PutUint64(puffer[i:], wert)
  }
  *zustand = wert
}
```

Listing 3.6: Erzeugen nicht komprimierbarer Dateiinhalte (Go)

Für das Zugriffsmuster im Messlauf wird ein eigener, sprachunabhängig definierter Generator benötigt. Die Standardgeneratoren der Sprachen, also `math/rand` in Go und `java.util.Random` in Java, liefern bei gleichem Seed unterschiedliche Zahlenfolgen, sodass die vier Modellvarianten ohne Eingriff verschiedene Blöcke des Datensatzes lesen. Die Prüfsumme wäre dann zwischen den Laufzeitsystemen nicht vergleichbar und könnte die bitgenaue Gleichheit des Lastprofils nicht belegen, die in Abschnitt 3.2 als Anforderung formuliert ist. Deshalb verwenden alle vier Programme einen xorshift64*-Generator mit fest definierten 64-Bit-Operationen. Dessen Verschiebungen und die Multiplikation laufen in Go, Java und Node.js modulo 2^{64} identisch ab, sodass `naechste()` bei gleichem Startwert in allen Sprachen dieselbe Folge liefert.

```go
type Zufall struct{ zustand uint64 }

func neuerZufall(seed uint64) *Zufall {
  if seed == 0 {
    seed = 0x9E3779B97F4A7C15
  }
  return &Zufall{zustand: seed}
}
```

```
9
10 func (z *Zufall) naechste() uint64 {
11   z.zustand ^= z.zustand >> 12
12   z.zustand ^= z.zustand << 25
13   z.zustand ^= z.zustand >> 27
14   return z.zustand * 0x2545F4914F6CDD1D
15 }
```

Listing 3.7: Portabler Zufallsgenerator für das Zugriffsmuster (Go)

Listing 3.7 zeigt die Implementierung des portablen Zufallsgenerators. Der Generator darf nicht im Nullzustand starten, weshalb ein von null verschiedener Initialwert erzwungen wird. Jede Aufgabe führt die ihr zugeteilte Zahl zufälliger Lesezugriffe aus, wählt dabei für jeden Zugriff eine zufällige Datei und einen an der Blockgrenze ausgerichteten Offset und liest jeweils einen Block fester Größe über `ReadAt`, Listing 3.8 zeigt diesen Lesepfad. Die zufällige und über den gesamten Datensatz gestreute Auswahl unterläuft den Readahead-Mechanismus des Kernels, der bei sequentiellem Zugriff die folgenden Seiten vorab lädt und die Gerätelatenz damit verbergen würde. Da die Zugriffe weit auseinanderliegen und zufällig über den gesamten Datensatz verteilt sind, läuft das Prefetchen ins Leere und ein großer Teil der Zugriffe muss aus dem Datenträger bedient werden [32]. Die an der Blockgröße ausgerichteten Offsets entsprechen zugleich der Seitengröße des Systems, sodass jeder Zugriff genau eine Seite anfordert. Die gelesenen Bytes werden über `aktualisierePruefsumme` in eine fortlaufende Prüfsumme geschrieben. Diese erfüllt denselben Zweck wie in der rechengebundenen Last, ihr Ergebnis wird am Ende ausgegeben, sodass der Compiler die Schleife nicht als wirkungslos entfernen kann und sie bestätigt bei identischem Wert über alle vier Laufzeitsysteme, dass dieselben Blöcke in derselben Reihenfolge gelesen wurden.

```
1  func aktualisierePruefsumme(pruefsumme uint32, daten []byte) uint32 {
2    for _, b := range daten {
3      pruefsumme = (pruefsumme + uint32(b)) * streuFaktor
4      pruefsumme ^= pruefsumme >> 15
5    }
6    return pruefsumme
7  }
8
9  func leseZufaellig(datensatz *Datensatz, puffer []byte,
10   anzahl int, zufall *Zufall) (uint32, int64) {
11
12   var pruefsumme uint32
13   var gelesen int64
14   blockgroesse := int64(len(puffer))
15
```

```
  for i := 0; i < anzahl; i++ {
    dateiIndex := int(zufall.naechste() % uint64(len(datensatz.Dateien)))
    dateigroesse := datensatz.Groessen[dateiIndex]

    maxBloecke := (dateigroesse - blockgroesse) / blockgroesse
    if maxBloecke < 1 {
      maxBloecke = 1
    }
    offset := int64(zufall.naechste()%uint64(maxBloecke)) * blockgroesse

    gelesenJetzt, err := datensatz.Dateien[dateiIndex].ReadAt(puffer, offset)
    if err != nil && gelesenJetzt == 0 {
      continue
    }
    pruefsumme = aktualisierePruefsumme(pruefsumme, puffer[:gelesenJetzt])
    gelesen += int64(gelesenJetzt)
  }
  return pruefsumme, gelesen
}
```

Listing 3.8: Zufälliger Lesezugriff und Prüfsumme (Go)

Listing 3.9 zeigt den Ablauf eines Messlaufs. Er folgt demselben Schema wie die rechengebundene Last. Jede Aufgabe erhält einen festen Basis-Seed und eine eigene Aufgaben-ID. Aus diesen Werten wird ein Startwert abgeleitet, welcher ein stabiles Zugriffsmuster über alle Wiederholungen ergibt. Durch diese Umsetzung entsteht kein gemeinsamer Sperrpunkt und die Aufgaben blockieren sich nicht gegenseitig. Da alle vier Kandidaten dieselbe Gesamtzahl an Lesezugriffen derselben Größe gleichverteilt über denselben Datensatz ausführen, ist die erzeugte Last äquivalent, sodass ein Unterschied im Ergebnis auf das Nebenläufigkeitsmodell und nicht auf ein abweichendes Lastprofil zurückgeht. Wie auch bei der rechnergebunden Last, starten alle Aufgaben über ein Gate, so ist sichergestellt, dass das einmalige Öffnen der Dateien und das Anlegen der Puffer nicht in das intern gemessene Fenster fallen.

```
func fuehreMesslaufAus(config Config, datensatz *Datensatz) {
  var (
    startSignal      = make(chan struct{})
    alleBereit       sync.WaitGroup
    alleFertig       sync.WaitGroup
    pruefsummeGesamt int64
    gelesenGesamt    int64
    latenzen         = make([]time.Duration, config.AnzahlAufgaben)
    messbeginn       time.Time
```

```
10  )
11
12  alleBereit.Add(config.AnzahlAufgaben)
13  alleFertig.Add(config.AnzahlAufgaben)
14
15  for id := 0; id < config.AnzahlAufgaben; id++ {
16    go func(id int) {
17      defer alleFertig.Done()
18
19      zufall := neuerZufall(uint64(config.Seed) + uint64(id))
20      puffer := make([]byte, config.Blockgroesse)
21
22      alleBereit.Done()
23      <-startSignal
24
25      pruefsumme, gelesen := leseZufaellig(
26        datensatz, puffer, config.AnzahlLesezugriffe, zufall)
27
28      latenzen[id] = time.Since(messbeginn)
29      atomic.AddInt64(&pruefsummeGesamt, int64(pruefsumme))
30      atomic.AddInt64(&gelesenGesamt, gelesen)
31    }(id)
32  }
33
34  alleBereit.Wait()
35  snapshotVorMessung := erfasseSnapshot()
36
37  messbeginn = time.Now()
38  close(startSignal)
39  alleFertig.Wait()
40  wallclock := time.Since(messbeginn)
41
42  snapshotNachMessung := erfasseSnapshot()
43  sort.Slice(latenzen, func(i, j int) bool { return latenzen[i] < latenzen[j] })
44
45  ergebnis := Laufergebnis{
46    Wallclock:          wallclock,
47    LesezugriffeGesamt: float64(config.AnzahlAufgaben) *
    ↪  float64(config.AnzahlLesezugriffe),
48    GeleseneBytes:      gelesenGesamt,
49    Pruefsumme:         pruefsummeGesamt,
50    Latenzen:           latenzen,
```

```
51      Gomaxprocs:         config.Gomaxprocs,
52    }
53
54    if config.AlsCsvAusgeben {
55      druckeCsvZeile(config, snapshotVorMessung, snapshotNachMessung, ergebnis)
56    }
57  }
```

Listing 3.9: Ablauf eines Messlaufs der dateibasierten Last (Go)

3.4.3 Die gemischte Last

Die beiden vorangegangenen Testprogramme isolieren jeweils eine Ressource. Die rechengebundene Last (Abschnitt 3.4.1) hält die Kerne durchgehend beschäftigt, die dateibasierte Last (Abschnitt 3.4.2) zwingt sie überwiegend zum Warten auf den Datenträger. Reale Serveranwendungen bleiben jedoch selten in nur einem dieser Zustände. Ein typischer Aufruf rechnet zunächst, fordert dann Daten vom Datenträger oder von einem weiteren Dienst an und rechnet anschließend weiter. Das Programm für die gemischte Last bildet diesen Wechsel nach und ist damit der Testfall, der einem realen Szenario am nächsten kommt.

Entscheidend für die Aussagekraft ist, dass die beiden Phasen nicht simpel nacheinander, sondern verschränkt ausgeführt werden. Ein Programm, das zuerst die gesamte Rechenlast und danach die gesamte I/O-Last abarbeitet, wäre lediglich eine Hintereinanderausführung der beiden vorangegangenen Tests und brächte keine neuen Erkenntnisse. Der eigentliche Gegenstand der gemischten Last ist die Vermischung der Workloads. Während eine Ausführungseinheit auf den Datenträger wartet, kann die Laufzeitumgebung die Rechenphase einer anderen Einheit ausführen. Wie gut ein Laufzeitsystem diese Überlappung erreicht, ist eine zentrale Frage dieser Arbeit. Aus diesem Grund durchläuft jede Aufgabe eine feste Zahl von Iterationen, von denen jede eine kurze rechengebundene Phase und im Anschluss genau einen Lesezugriff ausführt. Das Programm verwendet die bereits eingeführten Mechanismen weiter. Die Zugriffskette der Rechenphase und der portable xorshift64*-Generator der I/O-Phase sind gegenüber den Abschnitten 3.4.1 und 3.4.2 unverändert und werden hier nicht erneut erläutert. Auch das Leeren des Page-Caches sowie der Datensatz und sein Aufbau entsprechen der dateibasierten Last (Abschnitt 3.4.2), sodass ein dort bereits erzeugter Datensatz unverändert weiterverwendet werden kann.

```
1  type Config struct {
2    AnzahlAufgaben int    // gleichzeitige Goroutinen
3    PufferBytes    int    // privater CPU-Puffer pro Aufgabe in Byte
4    Iterationen    int    // CPU+IO-Zyklen pro Aufgabe (== Lesezugriffe pro Aufgabe)
5    CpuSchritte    int    // Pointer-Chase-Schritte pro Iteration
```

```
  Verzeichnis    string // Ablageort des Datensatzes
  AnzahlDateien  int    // Anzahl der Dateien im Datensatz
  Dateigroesse   int64  // Größe je Datei in Byte
  Blockgroesse   int    // Größe je einzelnem Lesezugriff in Byte
  Seed           int64  // Basis-Seed für das Zugriffsmuster
  Gomaxprocs     int    // tatsächlich aktive GOMAXPROCS
}
```

Listing 3.10: Parameter eines Laufs der gemischten Last (Go)

Listing 3.10 zeigt die Parameter eines Laufs. `AnzahlAufgaben` ist die Zahl der gleichzeitigen Goroutinen. `PufferBytes` ist die Größe des privaten CPU-Puffers je Aufgabe und wird wie bei der rechengebundenen Last relativ zur L2-Kapazität gewählt. `Iterationen` ist die Zahl der CPU-und-I/O-Zyklen je Aufgabe und zugleich die Zahl der Lesezugriffe je Aufgabe, da je Iteration genau ein Block gelesen wird. `CpuSchritte` ist die Zahl der Pointer-Chase-Schritte je Rechenphase. Die übrigen Parameter stimmen mit der dateibasierten Last überein.

Aus der Verschränkung der Workloads ergeben sich dieselben Überlegungen zur Skalierung wie bei der dateibasierten Last. Hält der Aufbau die Gesamtzahl der Lesezugriffe über alle Aufgaben konstant und leitet die Zahl der Iterationen je Aufgabe aus der Aufgabenzahl ab, bleibt nicht nur die je Lauf gelesene Datenmenge konstant, sondern wegen der festen Schrittzahl je Iteration auch die Rechenarbeit. Beim Skalieren der Aufgabenzahl ändert sich damit allein die Zahl der gleichzeitig rechnenden und blockierenden Einheiten und nicht der Umfang der Arbeit. Listing 3.11 zeigt die verschränkte Schleife. Jede Iteration führt zunächst `cpuSchritte` Schritte des Pointer Chase aus, also die Rechenphase, in der der Kern auf den Speicher wartet und stößt anschließend genau einen zufälligen Lesezugriff an, also die I/O-Phase, in der die Einheit auf den Datenträger wartet. Die Ergebnisse beider Phasen fließen in dieselbe fortlaufende Prüfsumme ein. Die Reihenfolge der Generatoraufrufe, also erst die Wahl der Datei und dann die des Offsets, stimmt mit der dateibasierten Last überein, sodass das Zugriffsmuster und mit ihm die Prüfsumme über alle vier Laufzeitsysteme bitgenau gleich ist. Der Index der Zugriffskette und die Prüfsumme werden über alle Iterationen weitergetragen. Die gemeinsame Prüfsumme über beide Phasen erfüllt denselben Zweck wie in den beiden vorangegangenen Tests.

```
func verarbeiteGemischt(kette []uint32, datensatz *Datensatz, puffer []byte,
  iterationen, cpuSchritte int, zufall *Zufall) (uint32, int64) {

  var index, pruefsumme uint32
  var gelesen int64
  blockgroesse := int64(len(puffer))

  for it := 0; it < iterationen; it++ {
    // Rechengebundene Phase: Pointer Chase entlang der Zugriffskette.
```

```
    for s := 0; s < cpuSchritte; s++ {
      index = kette[index]
      pruefsumme = (pruefsumme + index) * streuFaktor
      pruefsumme ^= pruefsumme >> 15
    }

    // I/O-Phase: ein zufälliger, über den Datensatz gestreuter Lesezugriff.
    dateiIndex := int(zufall.naechste() % uint64(len(datensatz.Dateien)))
    dateigroesse := datensatz.Groessen[dateiIndex]

    maxBloecke := (dateigroesse - blockgroesse) / blockgroesse
    if maxBloecke < 1 {
      maxBloecke = 1
    }
    offset := int64(zufall.naechste()%uint64(maxBloecke)) * blockgroesse

    gelesenJetzt, err := datensatz.Dateien[dateiIndex].ReadAt(puffer, offset)
    if err != nil && gelesenJetzt == 0 {
      continue
    }
    pruefsumme = aktualisierePruefsumme(pruefsumme, puffer[:gelesenJetzt])
    gelesen += int64(gelesenJetzt)
  }
  return pruefsumme, gelesen
}
```

Listing 3.11: Die verschränkte Messschleife der gemischten Last (Go)

Listing 3.12 zeigt den Ablauf eines Messlaufs. Er folgt demselben Schema wie die beiden vorangegangenen Lasten. Jede Aufgabe legt ihre Zugriffskette, ihren Generator und ihren Lesepuffer bereits vor dem Gate an, sodass das einmalige Vorbereiten nicht in das intern gemessene Fenster fällt. Über das Gate beginnen alle Aufgaben zeitgleich.

```
func fuehreMesslaufAus(config Config, datensatz *Datensatz) {
  anzahlElemente := config.PufferBytes / 4 // 4 Byte je uint32

  var (
    startSignal      = make(chan struct{})
    alleBereit       sync.WaitGroup
    alleFertig       sync.WaitGroup
    pruefsummeGesamt int64
    gelesenGesamt    int64
```

```
  latenzen         = make([]time.Duration, config.AnzahlAufgaben)
  messbeginn       time.Time
)

alleBereit.Add(config.AnzahlAufgaben)
alleFertig.Add(config.AnzahlAufgaben)

for id := 0; id < config.AnzahlAufgaben; id++ {
  go func(id int) {
    defer alleFertig.Done()

    kette := erzeugeZugriffskette(anzahlElemente)
    zufall := neuerZufall(uint64(config.Seed) + uint64(id))
    puffer := make([]byte, config.Blockgroesse)

    alleBereit.Done()
    <-startSignal

    pruefsumme, gelesen := verarbeiteGemischt(
      kette, datensatz, puffer, config.Iterationen, config.CpuSchritte, zufall)

    latenzen[id] = time.Since(messbeginn)
    atomic.AddInt64(&pruefsummeGesamt, int64(pruefsumme))
    atomic.AddInt64(&gelesenGesamt, gelesen)
  }(id)
}

alleBereit.Wait()
snapshotVorMessung := erfasseSnapshot()
messbeginn = time.Now()
close(startSignal)
alleFertig.Wait()
wallclock := time.Since(messbeginn)
snapshotNachMessung := erfasseSnapshot()
sort.Slice(latenzen, func(i, j int) bool { return latenzen[i] < latenzen[j] })

ergebnis := Laufergebnis{
  Wallclock:          wallclock,
  AnzahlAufgaben:     config.AnzahlAufgaben,
  LesezugriffeGesamt: float64(config.AnzahlAufgaben) * float64(config.Iterationen),
  GeleseneBytes:      gelesenGesamt,
```

```
51      CpuSchritteGesamt:  float64(config.AnzahlAufgaben) * float64(config.Iterationen)
        ↪  * float64(config.CpuSchritte),
52      Pruefsumme:         pruefsummeGesamt,
53      Latenzen:           latenzen,
54      Gomaxprocs:         config.Gomaxprocs,
55    }
56
57    if config.AlsCsvAusgeben {
58      druckeCsvZeile(config, snapshotVorMessung, snapshotNachMessung, ergebnis)
59    }
60  }
```

Listing 3.12: Ablauf eines Messlaufs der gemischten Last (Go)

Die Messung erfasst dabei die Kennzahlen beider vorangegangenen Lasten in einem Lauf. Neben den Kontextwechseln, der Scheduling-Statistik und der NUMA-Verteilung wird über das Verhältnis der vom Block-Device geholten zu den logisch gelesenen Bytes weiterhin kontrolliert, dass die Zugriffe tatsächlich den Datenträger erreichen. Damit lässt sich für die vier Laufzeitsysteme nicht nur ablesen, welcher Durchsatz und welche Antwortzeitverteilung sich unter gemischter Last einstellen, sondern auch, wie unterschiedlich die Modelle die Wartezeit der I/O-Phase mit der Rechenphase anderer Einheiten überlappen.

3.4.4 Die netzwerkbasierte Last

Die drei vorangegangenen Programme isolieren jeweils eine Ressource auf einer einzelnen Maschine bzw. deren Kombination. Die netzwerkbasierte Last kehrt demgegenüber zum historischen Ausgangspunkt dieser Arbeit zurück, dem in Abschnitt 2.1 beschriebenen C10k-Problem und untersucht, wie die vier Modelle skalieren, wenn sehr viele Verbindungen gleichzeitig geöffnet werden. Anders als bei den bisherigen Tests muss hier kein Prefetcher unterlaufen und kein Page-Cache geleert werden. Im Mittelpunkt stehen Netzwerk, Durchsatz, Antwortzeit und Concurrency. Die Hardware-Counter und die Speicherbelegung werden nicht erfasst, da hier nicht die mikroarchitektonische Ebene, sondern das Verbindungsverhalten über die Zeit im Vordergrund steht. Der Test folgt damit bewusst einem klassischen, praxisnahen Aufbau, in dem ein Server unter realistischer Last steht und über die TCP-Statistik des Betriebssystems beobachtet wird.
Der Aufbau verteilt sich auf zwei Maschinen. Auf dem Testsystem läuft der Server, der jede eingehende Anfrage mit einer einfachen, immer gleichen Antwort beantwortet. Die Last kommt von einer zweiten Maschine über das Netzwerk. Diese Trennung folgt derselben Überlegung wie in Abschnitt 3.1 und 3.2, denn nur so konkurriert der Lastgenerator nicht um Rechenzeit, Cache und Speicherbandbreite des Testsystems und verfälscht die Messung dadurch nicht.

Auf der zweiten Maschine öffnet ein Lastgenerator mit hoher Nebenläufigkeit eine große und schrittweise erhöhte Zahl gleichzeitiger Verbindungen und hält die Last, bis sich ein stabiler Zustand einstellt. Als Generator dient `oha`, das eine fest vorgegebene Zahl paralleler Verbindungen über einen festgelegten Zeitraum bedient. Im Gegensatz zu klassischen Closed-Loop-Generatoren unterstützt `oha` über den Parameter `--latency-correction` eine Korrektur des Coordinated-Omission-Problems. Mit aktivierter Korrektur misst `oha` die Latenzen aus Sicht eines offenen Ankunftsprozesses und liefert damit für Tail-Latenz-Aussagen belastbare Werte. Da die Verbindungen über HTTP-Keepalive bestehen bleiben, hält im thread-basierten Modell jede offene Verbindung einen eigenen Betriebssystem-Thread, der zwischen zwei Anfragen blockierend auf der Verbindung wartet. Genau dieser Zusammenhang machte das C10k-Problem auf der Ebene der Programmiermodelle sichtbar.
Jede Anfrage durchläuft auf dem Server eine kurze, fest eingestellte Wartezeit. Diese steht stellvertretend für eine blockierende Anfrage an ein nachgelagertes System, wie sie in der Einleitung als typischer Anteil der Lebensdauer serverseitiger Aufrufe beschrieben wurde und hält die Verbindung für ihre Dauer belegt. Erst dadurch entsteht die hohe gleichzeitige Belegung, die das thread-basierte Modell an seine Grenze führt. Die Wartezeit ist deterministisch und über alle vier Laufzeitsysteme identisch, sodass ein Unterschied im Ergebnis auf das Nebenläufigkeitsmodell und nicht auf ein schwankendes nachgelagertes System zurückgeht. Jedes Laufzeitsystem behandelt das Warten dabei nach seinem Modell. Wird die Wartezeit auf null gesetzt, misst der Test den reinen Bearbeitungsdurchsatz ohne diese Überlagerung.

Listing 3.13 zeigt die Parameter eines Laufs. `Adresse` ist die Listen-Adresse des Servers. `Wartezeit` ist die je Anfrage simulierte Wartezeit auf ein nachgelagertes System. `AntwortGroesse` ist die Größe der festen Antwort in Byte und `Gomaxprocs` hält den tatsächlich aktiven Wert von `GOMAXPROCS` fest.

```go
type Config struct {
  Adresse        string        // Listen-Adresse, etwa ":5431"
  Wartezeit      time.Duration // simulierte Wartezeit je Anfrage
  AntwortGroesse int           // Größe der festen Antwort in Byte
  Gomaxprocs     int           // tatsächlich aktive GOMAXPROCS
}
```

Listing 3.13: Parameter eines Laufs der netzwerkbasierten Last (Go)

Listing 3.14 zeigt den Kern des Servers. Die Funktion `erzeugeHandler` liefert die je Anfrage aufgerufene Funktion, die `net/http` in der Goroutine der jeweiligen Verbindung ausführt. Sie zählt die gleichzeitig in Bearbeitung befindlichen Anfragen hoch, wartet die simulierte Zeit ab, sendet die feste Antwort und zählt anschließend wieder herunter. Die Funktion `aktualisiereMaximum` hält über eine Compare-and-Swap-Schleife das beobachtete Maximum gleichzeitig aktiver Anfragen fest, da mehrere Goroutinen diesen Wert nebenläufig fortschreiben. Der Zähler `anfragenGesamt` wird erst nach dem Senden der Antwort erhöht und zählt damit die tatsächlich abgeschlossenen Anfragen.

```go
var (
  aktiveAnfragen int64
  maxAktiv       int64
  anfragenGesamt int64
)

func aktualisiereMaximum(aktuell int64) {
  for {
    bisher := atomic.LoadInt64(&maxAktiv)
    if aktuell <= bisher {
      return
    }
    if atomic.CompareAndSwapInt64(&maxAktiv, bisher, aktuell) {
      return
    }
  }
}

func erzeugeHandler(config Config, antwort []byte) http.HandlerFunc {
  return func(schreiber http.ResponseWriter, anfrage *http.Request) {
    aktuell := atomic.AddInt64(&aktiveAnfragen, 1)
    aktualisiereMaximum(aktuell)

    if config.Wartezeit > 0 {
      time.Sleep(config.Wartezeit)
    }

    schreiber.Header().Set("Content-Type", "text/plain")
    schreiber.Write(antwort)

    atomic.AddInt64(&anfragenGesamt, 1)
    atomic.AddInt64(&aktiveAnfragen, -1)
  }
}
```

Listing 3.14: Zähler und Handler der netzwerkbasierten Last (Go)

3.5 Die Messwerte

Abschnitt 3.3 hat begründet, warum die Kennzahlen auf zwei sich ergänzenden Ebenen erhoben werden, einer äußeren Ebene über `perf` und einer prozessinternen Ebene über das `/proc`-Dateisystem. Dieser Abschnitt macht konkret, welche Größe an welcher Stelle und auf welchem Weg erfasst wird und ordnet jede Kennzahl ihrer Quelle zu. Die Erfassung ist für die rechengebundene, die dateibasierte und die gemischte Last identisch und in eine eigene Quelldatei ausgelagert, die das jeweilige Messprogramm nur an den beiden bereits in Listing 3.4 sichtbaren Punkten aufruft. Die netzwerkbasierte Last folgt einem abweichenden Schema, das am Ende des Abschnitts gesondert behandelt wird.
Die prozessinterne Ebene beruht auf zwei Momentaufnahmen, die das Programm selbst erstellt. Die Erste wird erfasst, sobald alle Ausführungseinheiten am Gate bereitstehen und ihre Puffer angelegt und eingelagert haben, also unmittelbar bevor der eigentliche Workload beginnt. Die zweite folgt, nachdem alle Einheiten ihre Arbeit abgeschlossen haben. Aus der Differenz beider Aufnahmen ergeben sich die Kennzahlen des Messfensters, während die einmaligen Vorbereitungskosten außerhalb des Fensters liegen und nicht in die prozessinternen Differenzen einfließen. Eine einzelne Momentaufnahme bündelt dabei alle zu einem Zeitpunkt lesbaren systemnahen Zähler des eigenen Prozesses. Felder, die sich auf dem Testsystem nicht lesen lassen, behalten einen Sentinelwert und werden bei der Ausgabe übersprungen, sodass eine fehlende Quelle die übrigen Werte nicht verfälscht.

Listing 3.15 zeigt, wie eine Momentaufnahme aus den einzelnen `/proc`-Quellen zusammengesetzt wird. Jede der aufgerufenen Funktionen liest genau eine Quelle. Der Speicherverbrauch und die Threadzahl stammen aus `/proc/self/status`, die Minor und Major Major Page Faults sowie die im User- und im Kernel-Mode verbrachte CPU-Zeit aus `/proc/self/stat`. Die Kontextwechsel und die Scheduling-Statistik liefert der Kernel nur je Thread, weshalb das Programm über alle Einträge in `/proc/self/task` iteriert und die Werte aufsummiert.

```
1  func erfasseSnapshot() Snapshot {
2    snap := Snapshot{ /* alle Zaehler zunaechst auf -1 (nicht verfuegbar) */ }
3
4    liesSpeicherUndThreads(&snap)              // /proc/self/status
5    liesKontextwechsel(&snap)                  // /proc/self/task/*/status
6    liesStatZaehler(&snap)                     // /proc/self/stat
7    liesSchedZaehler(&snap)                    // /proc/self/task/*/schedstat
8    liesIoZaehler(&snap)                       // /proc/self/io
9    snap.NumaSeitenJeKnoten = liesNumaVerteilung() // /proc/self/numa_maps
10
11   return snap
12 }
```

Listing 3.15: Zusammensetzen einer Momentaufnahme aus den `/proc`-Quellen (Go)

Die drei Felder der Datei `schedstat` stehen dabei für die auf der CPU verbrachte Zeit, die Wartezeit in der Run-Queue und die Zahl der zugeteilten Timeslices. Ergänzend liest das Programm `/proc/self/io`, das die logisch gelesenen Bytes, die tatsächlich vom Block-Device geholten Bytes und die Zahl der Lese-Systemaufrufe enthält. Dieses Verhältnis dient als zentrale Kontrolle dafür, dass die Lesezugriffe bei der dateibasierten und der gemischten Last den Datenträger wirklich erreichen und nicht aus dem Page-Cache bedient werden. Bei der rechengebundenen Last sind diese Zähler nahe null und bestätigen damit, dass keine unerwarteten I/O-Zugriffe stattgefunden haben. Die Verteilung der anonymen Speicherseiten über die NUMA-Knoten ergibt sich schließlich aus `/proc/self/numa_maps`.

Aus den beiden Momentaufnahmen leitet das Programm einige Verhältniszahlen ab, die den Vergleich der Modelle erleichtern. Aus der im User- und der im Kernel-Mode verbrachten Zeit bildet es einen Kernel-Anteil, der angibt, welcher Teil der CPU-Zeit im Kernel verbracht wurde. Aus der CPU-Zeit und der Wartezeit in der Run-Queue bildet es einen Warteanteil, der die Sättigung der CPU-Kerne sichtbar macht. Aus der Verschiebung der Seitenverteilung zwischen den beiden Aufnahmen ergibt sich die Zahl der über die Knotengrenzen migrierten Seiten. Die Major Page Faults dienen zugleich als Kontrolle. Tritt im Messfenster der rechengebundenen Last ein Major Page Fault auf, deutet das auf einen unerwünschten Zugriff auf Swap oder Datenträger hin, der die Messung unbrauchbar macht, weshalb hierauf in der Ausgabe geachtet werden muss. Die Ausgabe erfolgt in einem CSV-Format sodass sich mehrere Läufe außerhalb des Programms zusammenfassen lassen. Die Tabelle 3.1 zeigt alle erfassten Messwerte über die verschiedenen Testszenarien und einer kurzen Erklärung.

Tabelle 3.1: Erfasste Messwerte über die Testszenarien

CSV-Spalte	Einheit	Szenario	Bedeutung
proc Metriken			
`sprache`	Text	alle	Programmiersprache der Implementierung
`modell`	Text	alle	Nebenläufigkeitsmodell der Implementierung, etwa goroutinen, threads, virtual-threads oder event-loop
`tasks`	Anzahl	alle	Zahl der gleichzeitig ausgeführten Aufgaben des Laufs
`bufbytes`	Byte	cpu, mixed	Größe des privaten Puffers je Aufgabe
`steps`	Anzahl	cpu	Rechenschritte je Aufgabe

Fortsetzung auf nächster Seite

Tabelle 3.1: Erfasste Messwerte über die Testszenarien

CSV-Spalte	Einheit	Szenario	Bedeutung
`iters`	Anzahl	mixed	über alle Aufgaben konstant gehaltene Gesamtzahl der CPU- und IO-Zyklen
`cpusteps`	Anzahl	mixed	Rechenschritte je Iteration
`blocksize`	Byte	fileio	Größe eines Lese-Blocks
`reads`	Anzahl	fileio	über alle Aufgaben konstant gehaltene Gesamtzahl der Lesezugriffe
`os_threads`	Threads	alle	Zahl der Betriebssystem-Threads am Gate, also vor dem Messfenster
`parallelism`	Anzahl	alle	konfigurierte Parallelität des Laufzeitsystems wie GOMAXPROCS oder Scheduler-Parallelism, also die Zahl der Carrier-Threads
`vmrss_kb`	kB	alle	am Gate belegter physischer Speicher, VmRSS
`vmswap_kb`	kB	alle	am Gate in den Swap ausgelagerter Speicher, dient als Kontrolle gegen Auslagerung
`vmhwm_kb`	kB	alle	im Lauf maximal belegter physischer Speicher, VmHWM
`vmswap_post_kb`	kB	fileio, mixed	nach dem Messfenster ausgelagerter Speicher
`os_threads_post`	Threads	fileio, mixed	Zahl der Betriebssystem-Threads nach dem Messfenster
`minflt_gate`	Anzahl	cpu, mixed	Stand der Minor Page Faults am Gate
`majflt_gate`	Anzahl	cpu, mixed	Stand der Major Page Faults am Gate
`minflt_delta`	Anzahl	cpu, mixed	Minor Page Faults im Messfenster
`majflt_delta`	Anzahl	cpu, mixed	Major Page Faults im Messfenster, Kontrolle gegen Swap- und Datenträgerzugriffe

Fortsetzung auf nächster Seite

Tabelle 3.1: Erfasste Messwerte über die Testszenarien

CSV-Spalte	Einheit	Szenario	Bedeutung
`ctx_vol_delta`	Anzahl	alle	freiwillige Kontextwechsel im Messfenster, über alle Threads summiert
`ctx_nonvol_delta`	Anzahl	alle	unfreiwillige Kontextwechsel im Messfenster, über alle Threads summiert
`utime_s`	s	alle	im User-Mode verbrachte CPU-Zeit im Messfenster
`stime_s`	s	alle	im Kernel-Mode verbrachte CPU-Zeit im Messfenster
`kernel_ratio_pct`	%	alle	Anteil der im Kernel-Mode verbrachten an der gesamten CPU-Zeit
`sched_cpu_gate_s`	s	alle	bis zum Gate auf der CPU verbrachte Zeit, über alle Threads summiert
`sched_wait_gate_s`	s	alle	bis zum Gate in der Run-Queue gewartete Zeit, über alle Threads summiert
`slices_gate`	Anzahl	alle	bis zum Gate zugeteilte Zeitscheiben, über alle Threads summiert
`sched_cpu_delta_s`	s	alle	im Messfenster auf der CPU verbrachte Zeit, über alle Threads summiert
`sched_wait_delta_s`	s	alle	im Messfenster in der Run-Queue gewartete Zeit, über alle Threads summiert
`slices_delta`	Anzahl	alle	im Messfenster zugeteilte Zeitscheiben, über alle Threads summiert
`wait_ratio_pct`	%	alle	Anteil der Wartezeit an der Summe aus CPU-Zeit und Wartezeit, ein Maß für die Sättigung der Kerne
`io_idle_pct`	%	fileio, mixed	Anteil der Messfensterzeit, in der die Kerne nicht ausgelastet waren, ein Maß für die Wartezeit auf I/O

Fortsetzung auf nächster Seite

Tabelle 3.1: Erfasste Messwerte über die Testszenarien

CSV-Spalte	Einheit	Szenario	Bedeutung
`rchar_gate`	Byte	fileio, mixed	bis zum Gate logisch gelesene Bytes
`read_bytes_gate`	Byte	fileio, mixed	bis zum Gate tatsächlich vom Block-Device geholte Bytes
`syscr_gate`	Anzahl	fileio, mixed	bis zum Gate abgesetzte Lese-Systemaufrufe
`rchar_delta`	Byte	fileio, mixed	im Messfenster logisch gelesene Bytes
`read_bytes_delta`	Byte	fileio, mixed	im Messfenster tatsächlich vom Block-Device geholte Bytes
`syscr_delta`	Anzahl	fileio, mixed	im Messfenster abgesetzte Lese-Systemaufrufe
`device_ratio_pct`	%	fileio, mixed	Anteil der logisch gelesenen Bytes, der bis auf den Datenträger durchschlägt, Kontrolle gegen Bedienung aus dem Page-Cache
`numa_total_gate`	Seiten	alle	am Gate über alle NUMA-Knoten verteilte anonyme Seiten
`numa_migrated`	Seiten	alle	zwischen Gate und Ende über die Knotengrenzen migrierte anonyme Seiten
`wall_s`	s	alle	Wanduhrzeit des Messfensters
`throughput`	Schritte/s	cpu	Durchsatz des rechengebundenen Laufs, verarbeitete Rechenschritte je Sekunde
`tasks_per_s`	Aufgaben/s	mixed	Durchsatz an abgeschlossenen Aufgaben
`reads_per_s`	Lesezugriffe/s	fileio, mixed	Durchsatz an Lesezugriffen
`throughput_mib_s`	MiB/s	fileio, mixed	gelesener Datendurchsatz
`checksum`	ohne	alle	über alle Aufgaben gebildete Prüfsumme, über alle Modelle identisch und damit eine Korrektheitskontrolle
`lat_min_us`	μs	alle	kürzeste Aufgabenlaufzeit, Minimum der Latenzen

Fortsetzung auf nächster Seite

Tabelle 3.1: Erfasste Messwerte über die Testszenarien

CSV-Spalte	Einheit	Szenario	Bedeutung
`lat_p50_us`	μs	alle	Median der Aufgabenlaufzeiten, 50. Perzentil
`lat_p99_us`	μs	alle	99. Perzentil der Aufgabenlaufzeiten
`lat_max_us`	μs	alle	längste Aufgabenlaufzeit, Maximum der Latenzen
perf stat Metriken			
`model`	Text	alle	Kürzel des Nebenläufigkeitsmodells, go, java, loom oder js
`instructions`	Anzahl	alle	vom Prozessor ausgeführte Instruktionen über den gesamten Prozess
`cycles`	Anzahl	alle	Taktzyklen über den gesamten Prozess
`cpu_migrations`	Anzahl	alle	vom Kernel gezählte Wechsel eines Threads auf einen anderen CPU-Kern
`context_switches`	Anzahl	alle	vom Kernel gezählte Kontextwechsel
`task_clock_msec`	ms	alle	tatsächlich auf der CPU verbrachte Zeit
`raw_syscalls`	Anzahl	alle	Zahl der abgesetzten Systemaufrufe und damit der Mode Switches, Tracepoint `raw_syscalls:sys_enter`
`sched_switch`	Anzahl	alle	Scheduling-Wechsel-Ereignisse des Kernels
`sched_wakeup`	Anzahl	alle	Aufweck-Ereignisse lauffähig gewordener Threads
`elapsed_s`	s	alle	von `perf` gemessene Gesamtlaufzeit des Prozesses aus dem ersten Durchlauf
`elapsed_stddev_s`	s	alle	Standardabweichung der Gesamtlaufzeit über die Wiederholungen

Fortsetzung auf nächster Seite

Tabelle 3.1: Erfasste Messwerte über die Testszenarien

CSV-Spalte	Einheit	Szenario	Bedeutung
`l1d_misses`	Anzahl	alle	Cache-Misses beim Laden aus dem L1-Datencache
`llc_misses`	Anzahl	alle	Cache-Misses im Last-Level-Cache
`dtlb_misses`	Anzahl	cpu, mixed	Misses im TLB für Datenzugriffe
`itlb_misses`	Anzahl	cpu, mixed	Misses im TLB für Instruktionszugriffe
`minor_faults`	Anzahl	alle	Minor Page Faults über den gesamten Prozess
`major_faults`	Anzahl	alle	Major Page Faults über den gesamten Prozess
`block_rq_issue`	Anzahl	fileio, mixed	an das Block-Device abgesetzte Requests
`block_rq_complete`	Anzahl	fileio, mixed	vom Block-Device abgeschlossene Requests
`*_variance`	%	alle	relative Streuung des gleichnamigen `perf`-Zählers über die Wiederholungen, je Zähler eine eigene Spalte wie `instructions_variance` oder `cycles_variance`, bei nur einer Wiederholung NULL
perf sched Metriken			
`run`	Anzahl	alle	laufende Nummer der Sched-Aufzeichnung, im Messplan stets 1
`filter_task`	Text	alle	Name des betrachteten `comm`, etwa node, libuv-worker oder ForkJoinPool, auf den `perf sched latency` gefiltert wurde
`runtime_ms`	ms	alle	gesamte Laufzeit dieses `comm` auf der CPU
`switches`	Anzahl	alle	Zahl der Scheduling-Wechsel dieses `comm`

Fortsetzung auf nächster Seite

Tabelle 3.1: Erfasste Messwerte über die Testszenarien

CSV-Spalte	Einheit	Szenario	Bedeutung
`avg_delay_ms`	ms	alle	mittlere Wartezeit in der Run-Queue, bis der lauffähige Thread die CPU erhielt
`max_delay_ms`	ms	alle	maximale solche Wartezeit

Die äußere Ebene stützt sich auf `perf`, das die Messprogramme direkt ausführt und dabei den gesamten Prozess samt seiner Initialisierung erfasst. Da moderne Prozessoren nur eine begrenzte Zahl an Hardware-Zählern besitzen und `perf` bei zu vielen gleichzeitig angeforderten Ereignissen auf zeitlich verschränktes Zählen ausweicht [14], was die Genauigkeit verschlechtert, werden die Ereignisse auf zwei getrennte Durchläufe verteilt. Der erste Durchlauf erfasst die allgemeinen Kennzahlen wie Instruktionen, Taktzyklen, CPU-Migrationen, Kontextwechsel, Systemaufrufe und Scheduling-Ereignisse, der zweite die lastspezifischen Cache- und TLB-Misses sowie bei den dateibasierten Lasten zusätzlich die an das Block-Device gerichteten Requests. Jeder Durchlauf wird über `perf stat -r` mehrfach wiederholt, sodass sich neben dem typischen Wert auch die Schwankung zwischen den Läufen abschätzen lässt. Ergänzend zeichnet `perf sched` bei bestimmten Läufen das Scheduling auf und macht über `perf sched` die Wartezeiten der lauffähigen Threads und ihre Verteilung über die CPU-Kerne sichtbar. Da `perf` außerhalb der Messprogramme läuft, verändert die Messung den gemessenen Code nicht.

Die beiden Ebenen ergänzen sich. Die von `perf` erhobenen Zähler messen den gesamten Prozess und enthalten damit auch den geringen Aufwand der Vorbereitung, während die über `/proc` gebildeten Differenzen genau das Geschehen zwischen dem Gate und dem Ende des Messfensters abbilden. Kennzahlen, die in beiden Ebenen erscheinen, etwa die Kontextwechsel, lassen sich auf diese Weise gegeneinander prüfen, was der Empfehlung folgt, einzelne Werte nach Möglichkeit aus mehreren Quellen zu bestätigen [9].

Die netzwerkbasierte Last bildet die Ausnahme von diesem Schema. Da der Server ein langlaufender, von außen getriebener Prozess ist und das relevante Signal das Verbindungsverhalten über die Zeit ist, wird er nicht in ein eng abgegrenztes Messfenster gefasst und weder unter `perf` noch über eine `/proc/self`-Momentaufnahme beobachtet. Stattdessen hält der Server nur wenige anwendungsnahe Zähler, die gerade in Bearbeitung befindlichen Anfragen, deren beobachtetes Maximum und die insgesamt beantworteten Anfragen und gibt diese gemeinsam mit Laufzeit und Durchsatz beim Empfang von `SIGINT` oder `SIGTERM` aus. Die systemweiten TCP-Kennzahlen erfasst parallel ein Steuerskript, das in festem Intervall `/proc/net/snmp`, `/proc/net/netstat` und `/proc/net/sockstat` ausliest und fortlaufend in eine Datei schreibt. Die zentrale Kennzahl ist dabei die Zahl der aktuell etablierten TCP-Verbindungen, die den Verlauf der Concurrency über die Zeit abbildet. Damit liefert die netzwerkbasierte Last für jeden Kandidaten den erreichten Durchsatz, die Verteilung der Antwortzeiten aus Sicht des Lastgenerators und den über die Zeit beobachteten Verlauf der gleichzeitig getragenen Verbindungen [2].

4 Durchführung der Experimente & Darstellung der Ergebnisse

Nachdem Abschnitt 3 das Testsystem, die für wiederholbare Messungen nötige Konfiguration sowie die in Abschnitt 3.4 umgesetzten Testprogramme beschrieben und die dabei zu berücksichtigenden Seiteneffekte dargestellt hat, überträgt dieses Kapitel den Versuchsaufbau in die konkrete Durchführung. Beschrieben wird, mit welchen Parametern die Programme gestartet wurden, wie oft jede Konfiguration wiederholt wurde und aus welchen Überlegungen die gewählten Werte entstanden sind. Abschnitt 4.1 behandelt zuerst die drei lokalen Lastszenarien, also die rechengebundene, die dateibasierte und die gemischte Last. Abschnitt 4.2 behandelt anschließend die netzwerkbasierte Last, die als Sonderfall auf einem eigenen System und mit einem eigenen Messplan läuft. Abschließend werden die Ergebnisse in den letzten Abschnitten dargestellt.

4.1 Die lokalen Lastszenarien

Die drei lokalen Szenarien laufen alle auf dem in Abschnitt 3.1 beschriebenen Server und folgen demselben Ablauf. Für jede Kombination aus Nebenläufigkeitsmodell, Szenario und Aufgabenzahl wird das zugehörige Programm mit einem festen Satz an Parametern gestartet und über `perf` mehrfach wiederholt gemessen. Tabelle 4.1 fasst die Parameter der drei Szenarien zusammen, sodass sich für jeden Lauf genau ablesen lässt, mit welcher Aufgabenzahl, mit welchen Workload-Parametern und mit wie vielen Wiederholungen er gestartet wurde.

Tabelle 4.1: Messplan der drei lokalen Lastszenarien

Szenario	Aufgabenstufen (`tasks`)	Feste Workload-Parameter	Wiederholungen (`reps`)
cpu	100, 500, 2500, 10000	`steps`=20000000, `bufbytes`=1048576	go, java und loom je 10, js 3 bei 100/500/2500 und 1 bei 10000
fileio	100, 500, 2500, 10000	`reads`=2000000 (gesamt), `blocksize`=4096	go, java und loom je 10, js 3 bei 100/500/2500 und 1 bei 10000
mixed	100, 500, 2500, 10000	`iters`=2000000 (gesamt), `cpusteps`=10000, `bufbytes`=1048576, `blocksize`=4096	go, java und loom je 10, js 3 bei 100/500/2500 und 1 bei 10000

Einige Werte sind über alle drei Szenarien gleich und daher nicht je Szenario aufgeführt. Die konfigurierte Parallelität der Laufzeitsysteme, also `GOMAXPROCS` bei Go und die Scheduler-Parallelität der Virtual Threads ist an die 32 logischen Kerne des Testsystems gebunden. Die Zahl der `libuv`-Threads bei Node.js, wird auf dem Default-Wert von vier belassen. Dies schränkt die Vergleichbarkeit über die absoluten Zahlen ein, ist jedoch bewusst so belassen worden, da Node.js konzeptionell auf einem Single-Threaded-Event-Loop-Modell basiert und nicht durch eine künstliche Skalierung des internen libuv-Threadpools an ein Multi-Threading-Paradigma angepasst werden sollte. Zudem entspricht der Default-Wert von vier Threads dem Standardverhalten in der Praxis, da dieser Parameter in produktiven Implementierungen nicht unbedingt modifiziert wird. Die dateibasierte und die gemischte Last nutzen denselben Datensatz aus 64 Dateien mit je einem Gigabyte und der Zugriffs-Seed bleibt mit dem Wert 1 über alle Läufe fest, sodass die Zugriffsfolge reproduzierbar ist. Die Aufzeichnung des Schedulings über `perf sched` wird nur bei den beiden oberen Aufgabenstufen 2500 und 10000 aktiviert, da die Grenzen der Modelle erst bei hoher Nebenläufigkeit sichtbar werden und die Aufzeichnung bei jeder Stufe die Laufzeit und den Speicherbedarf der Trace deutlich erhöhen würde.
Bei der dateibasierten und der gemischten Last ist zu beachten, dass `reads` und `iters` nicht die Last je Aufgabe, sondern die über alle Aufgaben konstant gehaltene Gesamtzahl angeben. Das Programm leitet daraus die Arbeit je Aufgabe ab, indem es die Gesamtzahl durch die Aufgabenzahl teilt. Dadurch bleibt die insgesamt gelesene Datenmenge und bei der gemischten Last auch die gesamte Rechenlast über alle Aufgabenstufen gleich, sodass sich die Stufen fair vergleichen lassen und allein die Nebenläufigkeit variiert. Bei der rechengebundenen Last bleibt `steps` dagegen die Schrittzahl je Aufgabe, sodass die Gesamtlast hier mit der Aufgabenzahl wächst.

Die konkreten Werte in Tabelle 4.1 sind das Ergebnis mehrerer Vorüberlegungen und kurzer Testläufe, bei denen die Laufzeit der Messreihe und die Kapazität des Testsystems gegen die Menge der auswertbaren Messdaten abgewogen wurden. In einem ersten Durchlauf lief die Festplatte mit den Aufzeichnungen von `perf sched` voll, sodass die Messung abbrach. In einem zweiten Durchlauf waren die Parameter so groß gewählt, dass die reine Rechenzeit auf dem Testsystem bei etwa acht Wochen gelegen hätte, weshalb auch dieser Lauf abgebrochen werden musste. Die endgültigen Werte sind somit ein Kompromiss zwischen einer ausreichenden Auslastung und Beanspruchung von Testsystem und Programm auf der einen Seite und einer vertretbaren Laufzeit sowie einer handhabbaren Menge an Messdaten auf der anderen Seite. Mit diesen Werten liegt die geschätzte Gesamtlaufzeit der lokalen Messreihe bei etwa 38 Stunden, wobei die gemischte Last mit rund 27 Stunden den größten Anteil trägt.

4.2 Das netzwerkbasierte Lastszenario

Die netzwerkbasierte Last ist, wie bereits in Abschnitt 3 dargestellt, ein Sonderfall und läuft mit einem eigenen Aufbau und eigenen Testszenarien. Anders als bei den lokalen Szenarien wird die Last nicht auf dem Testsystem selbst erzeugt, sondern von einer zweiten Maschine über das Netzwerk. Beide Maschinen laufen dabei bewusst im selben Rechenzentrumsnetz bei Hetzner auf jeweils einem dedizierten Server mit dedizierter CPU. In einem ersten Aufbau mit eigener Hardware waren zwischen Testsystem und Lastgenerator eine Firewall bzw. weitere Netzwerkkomponenten zwischengeschaltet, die Teile des Datenverkehrs blockierten und die Messergebnisse dadurch unbrauchbar machten. Um eine möglichst störungsfreie und direkte Netzwerkanbindung zwischen beiden Maschinen sicherzustellen, wurde der Aufbau daher auf zwei baugleiche Server im selben Hetzner-Netz umgestellt.
Die Last erzeugt auf der zweiten, baugleichen Maschine der Generator `oha`, der wie in Abschnitt 3.3 beschrieben mit Latenzkorrektur arbeitet und dessen Anzeige über `--no-tui` abgeschaltet wird, damit sie die Messung nicht stört. Der Messplan besteht aus zwei Gruppen von Läufen, die Tabelle 4.2 zusammenfasst.

Als System unter Last dient ein dedizierter Server vom Typ Hetzner CCX43 mit 16 AMD-Kernen, 64 Gigabyte Arbeitsspeicher und einer 360 Gigabyte großen NVMe-SSD. Auf diesem Server läuft der jeweilige Testserver eines Modells. Der Go-Server wird beispielhaft mit `--addr :5431 --delay 10ms --respbytes 128 --gomaxprocs 16` gestartet, die Server der übrigen Modelle folgen demselben Aufbau.

Die erste Gruppe erhöht schrittweise die Zahl der gleichzeitig offenen Verbindungen von 100 über 500 und 2500 bis auf 10000, während der Generator die Anfragen ungebremst stellt. Damit wird sichtbar, bis zu welcher Verbindungszahl ein Modell den Durchsatz noch hält und ab wann das thread-basierte Modell an die in Abschnitt 2.1 beschriebene Grenze stößt. Jeder Lauf dauert 60 Sekunden, was lang genug ist, damit sich ein stabiler Zustand einstellt und kurz genug, dass die gesamte Messreihe in vertretbarer Zeit durchläuft. Diese Dauer wurde in mehreren Vorversuchen mit variierender Laufzeit ermittelt.

Die zweite Gruppe hält die Zahl der Verbindungen fest bei 10000 und steuert stattdessen die Anfragerate über `-q`. Als Bezugspunkt dient der Sättigungswert, also die höchste Rate, die das jeweilige Modell im ungebremsten Lauf mit 10000 Verbindungen dauerhaft beantworten konnte. Ausgehend von diesem Wert wird die Rate auf 50 Prozent für den Normalbetrieb, auf 80 Prozent für die Hochlast, auf 100 Prozent für den Kipppunkt und auf 110 Prozent für die Überlast gesetzt. Auf diese Weise lässt sich das Verhalten der Modelle nicht nur an der Grenze, sondern auch im geregelten Normal- und Hochlastbereich sowie im Überlastbereich beobachten, in dem die Anfragerate die Kapazität des Servers bewusst übersteigt.

Tabelle 4.2: Testszenarien der netzwerkbasierten Last mit `oha`

Szenario	Dauer (`-z`)	Verbindungen (`-c`)	Rate (`-q`)
Skalierung der Verbindungen			
100 gleichzeitige Verbindungen	60 s	100	ungebremst
500 gleichzeitige Verbindungen	60 s	500	ungebremst
2500 gleichzeitige Verbindungen	60 s	2500	ungebremst
10000 gleichzeitige Verbindungen	60 s	10000	ungebremst
Sättigungsstufen (mit `--latency-correction`)			
Normalbetrieb	60 s	10000	50 % des Sättigungswerts
Hochlast	60 s	10000	80 % des Sättigungswerts
Kipppunkt	60 s	10000	100 % des Sättigungswerts
Überlast	60 s	10000	110 % des Sättigungswerts

4.3 Ergebnisse der rechengebundenen Last

Dieser Abschnitt stellt die Messergebnisse der rechengebundenen Last dar. Zuerst werden die Werte der vier Modelle einzeln aufgeführt (Abschnitt 4.3.1), anschließend werden sie einander gegenübergestellt (Abschnitt 4.3.2). Die Werte stammen aus den beiden in Abschnitt 3.3 beschriebenen Ebenen, den prozessinternen Größen aus `/proc` und den Zählern von `perf`. Die Latenzen sind in Sekunden angegeben, der Speicherbedarf als maximal belegter physischer Speicher (VmHWM) in Mebibyte. Für jede Aufgabenstufe stimmten die Prüfsummen aller vier Modelle überein, etwa der Wert 367868081800 bei 100 Aufgaben, sodass über alle Modelle derselbe Workload vorlag. Die relative Streuung der perf-Zähler über die Wiederholungen blieb überwiegend unter zwei Prozent. Der Lauf von Node.js bei 10000 Aufgaben wurde als Einzelanker ohne Wiederholung gemessen, sodass hier keine Streuung vorliegt.

4.3.1 Ergebnisse der einzelnen Modelle

Für jedes Modell werden im Folgenden die Laufzeit des Messfensters, der Median und das 99. Perzentil der Aufgabenlatenz, der Speicherbedarf und die von `perf stat` gezählten Kontext-

wechsel über die vier Aufgabenstufen aufgeführt. Ergänzend wird jeweils die Zahl der am Gate gezählten Betriebssystem-Threads genannt, da sie sich zwischen den Modellen unterscheidet.

Go führt die Läufe über alle vier Stufen in der jeweils kürzesten Laufzeit aller Modelle aus. Die Laufzeit des Messfensters steigt von 0,73 Sekunden bei 100 Aufgaben auf 62,15 Sekunden bei 10000 Aufgaben, wobei der Median und das 99. Perzentil der Latenz auf jeder Stufe nahe beieinander liegen und bei 10000 Aufgaben 59,95 gegenüber 62,12 Sekunden betragen. Die Zahl der Kontextwechsel steigt über die Stufen von 1307 auf 38838, während die Zahl der Betriebssystem-Threads mit 33 bis 34 nahezu konstant bleibt und nicht mit der Aufgabenzahl wächst. Der maximal belegte physische Speicher wächst von 109 auf 10053 MiB. Tabelle 4.3 enthält die Werte im Einzelnen.

Tabelle 4.3: Rechengebundene Last, Ergebnisse für Go (Goroutinen)

Aufgaben	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)	Kontextwechsel
100	0,73	0,53	0,72	109	1307
500	3,38	2,11	3,36	512	3592
2500	15,76	14,90	15,65	2519	12846
10000	62,15	59,95	62,12	10053	38838

Bei den Plattform-Threads von Java folgt die Zahl der Betriebssystem-Threads der Aufgabenzahl. Sie steigt von 142 bei 100 Aufgaben auf 10045 bei 10000 Aufgaben und liegt damit jeweils rund vierzig über der Aufgabenzahl. Parallel dazu wächst die Zahl der Kontextwechsel von 9342 auf 994037 und übertrifft auf jeder Stufe die der drei übrigen Modelle. Zwischen Median und 99. Perzentil der Latenz liegt ein größerer Abstand als bei Go, bei 10000 Aufgaben stehen 63,01 Sekunden gegenüber 124,20 Sekunden. Der maximal belegte Speicher erreicht bei 10000 Aufgaben 14957 MiB und übertrifft damit den der übrigen Modelle. Tabelle 4.4 führt die Werte auf.

Tabelle 4.4: Rechengebundene Last, Ergebnisse für Java (Plattform-Threads)

Aufgaben	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)	Kontextwechsel
100	1,06	0,88	1,06	286	9342
500	5,87	4,23	5,84	1395	49322
2500	29,12	15,92	29,07	3627	250643

Fortsetzung auf nächster Seite

Tabelle 4.4: Rechengebundene Last, Ergebnisse für Java (Plattform-Threads)

Aufgaben	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)	Kontextwechsel
10000	124,50	63,01	124,20	14957	994037

Die Virtual Threads führen die Läufe in Laufzeiten aus, die nahe an denen von Go liegen, bei 10000 Aufgaben 70,17 gegenüber 62,15 Sekunden. Die Zahl der Betriebssystem-Threads bleibt mit 75 bis 82 über alle Stufen annähernd konstant und folgt nicht der Aufgabenzahl, während die Kontextwechsel von 2486 auf 24852 steigen. Der Median der Latenz liegt auf jeder Stufe unter dem 99. Perzentil, bei 10000 Aufgaben 35,14 gegenüber 69,55 Sekunden, wobei der Abstand zwischen beiden größer ist als bei Go. Der Speicherbedarf erreicht 13184 MiB bei 10000 Aufgaben. Tabelle 4.5 zeigt die Werte.

Tabelle 4.5: Rechengebundene Last, Ergebnisse für Java (Virtual Threads)

Aufgaben	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)	Kontextwechsel
100	0,93	0,69	0,93	229	2486
500	3,49	1,79	3,47	629	3843
2500	17,57	8,88	17,47	3286	8151
10000	70,17	35,14	69,55	13184	24852

Node.js läuft mit einer konfigurierten Parallelität von eins und hält die Zahl der Betriebssystem-Threads über alle Stufen konstant bei 11. Die Laufzeit des Messfensters steigt von 17,22 Sekunden bei 100 Aufgaben auf 1595,74 Sekunden bei 10000 Aufgaben und ist damit auf jeder Stufe die längste aller Modelle. Der Median der Latenz liegt bei 10000 Aufgaben bei 804,62 Sekunden und das 99. Perzentil bei 1579,93 Sekunden. Die Kontextwechsel steigen über die Stufen von 321 auf 18876 und der Speicherbedarf von 157 auf 10071 MiB. Tabelle 4.6 zeigt die Werte.

Tabelle 4.6: Rechengebundene Last, Ergebnisse für Node.js (Event-Loop)

Aufgaben	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)	Kontextwechsel
100	17,22	8,86	17,05	157	321

Fortsetzung auf nächster Seite

Tabelle 4.6: Rechengebundene Last, Ergebnisse für Node.js (Event-Loop)

Aufgaben	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)	Kontextwechsel
500	82,65	41,00	81,82	556	1064
2500	408,07	208,55	404,12	2561	4853
10000	1595,74	804,62	1579,93	10071	18876

4.3.2 Vergleich der Modelle

Die Gegenüberstellung fasst die vier Modelle über alle Aufgabenstufen zusammen und stützt sich auf den Durchsatz, die Latenzverteilung, die Kontextwechsel, die Seitenmigration über die NUMA-Knoten, ausgewählte Hardware-Zähler und die aufgezeichnete Scheduling-Latenz.

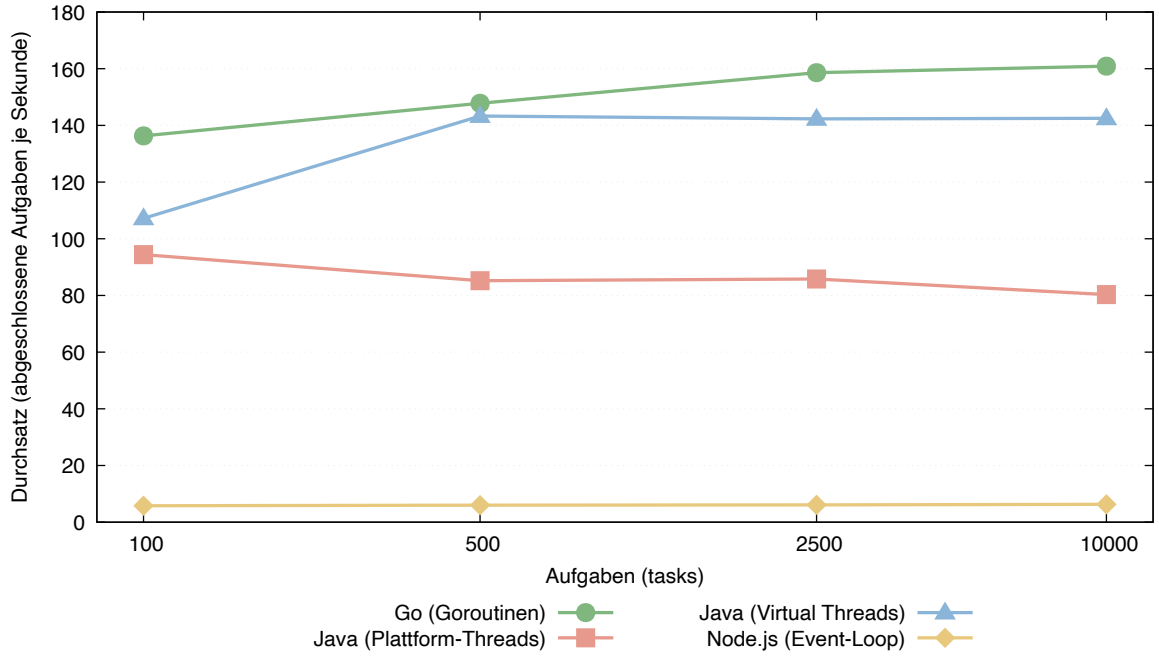

Abbildung 4.1: Rechengebundene Last, Durchsatz in abg. Aufgaben je Sekunde über die Aufgabenstufen

Abbildung 4.1 stellt den Durchsatz in abgeschlossenen Aufgaben je Sekunde über die Aufgabenstufen dar. Der Durchsatz von Go steigt von 136,3 auf 160,9 und der der Virtual Threads von 107,2 auf 142,5, sodass beide Verläufe eng beieinander liegen. Die Plattform-Threads gehen von 94,4 auf 80,3 zurück und liegen damit unter den beiden erstgenannten Verläufen. Node.js verläuft zwischen 5,8 und 6,3 und bildet über alle Stufen den untersten Verlauf der Abbildung.

Die Latenz zeigt Abbildung 4.2 anhand des 99. Perzentils. Für alle vier Modelle steigt das 99. Perzentil mit der Aufgabenzahl. Die Verläufe von Go, den Virtual Threads und den Plattform-Threads liegen dabei enger beieinander, während der Verlauf von Node.js über den gesamten Bereich um mehr als eine Größenordnung darüber liegt und bei 10000 Aufgaben 1579,93 Sekunden erreicht.

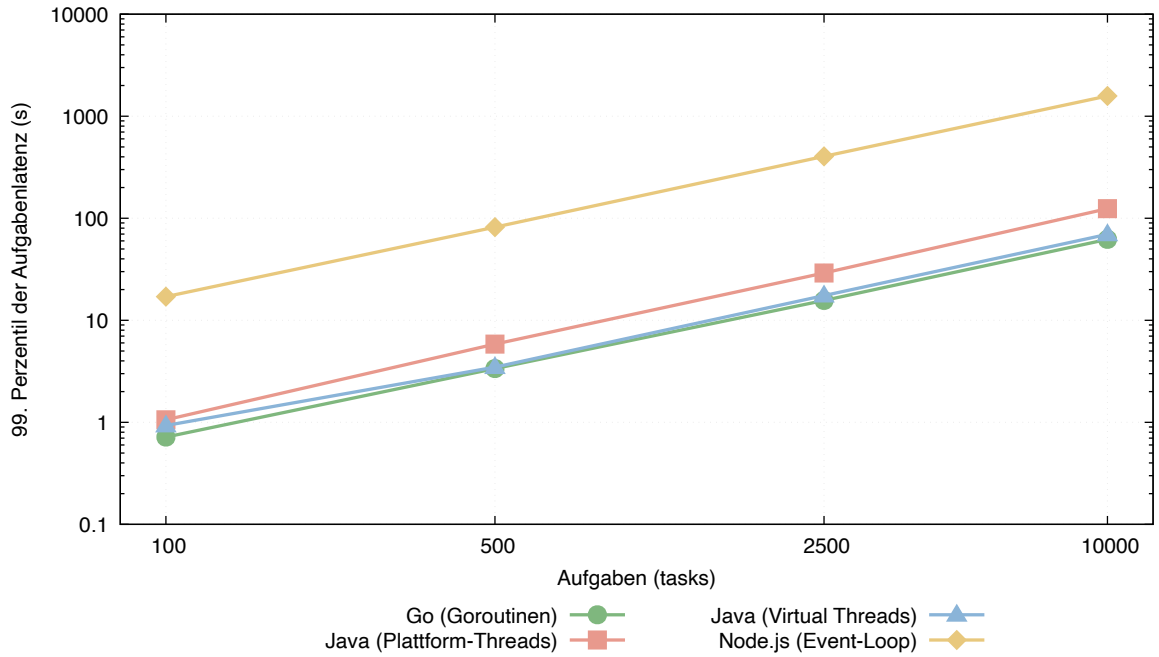

Abbildung 4.2: Rechengebundene Last, 99. Perzentil der Aufgabenlatenz über die Tasks

Auf der Hardware-Ebene stellt Tabelle 4.7 ausgewählte perf-Zähler bei der höchsten Stufe von 10000 Aufgaben gegenüber. Node.js führt mit 4482,9 Milliarden die meisten Instruktionen aus und erreicht mit 0,93 die höchste Zahl an Instruktionen je Zyklus, während die Plattform-Threads mit 0,19 die niedrigste erreichen und Go sowie die Virtual Threads mit 0,46 und 0,31 dazwischen liegen.

Tabelle 4.7: Rechengebundene Last, ausgewählte perf-Zähler bei 10000 Aufgaben

Modell	Instruktionen (10^9)	Zyklen (10^9)	IPC	CPU-Migrationen	LLC-Misses (10^6)	dTLB-Misses (10^6)
Go (Goroutinen)	2236,8	4847,7	0,46	371	3409,6	10,6
Java (Plattform-Threads)	1699,8	9038,3	0,19	50871	18516,1	188,5

Fortsetzung auf nächster Seite

Tabelle 4.7: Rechengebundene Last, ausgewählte perf-Zähler bei 10000 Aufgaben

Modell	Instruktionen (10^9)	Zyklen (10^9)	IPC	CPU-Migrationen	LLC-Misses (10^6)	dTLB-Misses (10^6)
Java (Virtual Threads)	1710,5	5484,2	0,31	903	2813,9	9,9
Node.js (Event-Loop)	4482,9	4811,4	0,93	161	222,4	16,5

Die Zahl der CPU-Migrationen reicht von 161 bei Node.js über 371 bei Go und 903 bei den Virtual Threads bis 50871 bei den Plattform-Threads. Bei den Misses im Last-Level-Cache reicht die Spannweite von 222,4 Millionen bei Node.js bis 18516,1 Millionen bei den Plattform-Threads, bei den dTLB-Misses von 9,9 Millionen bei den Virtual Threads bis 188,5 Millionen bei den Plattform-Threads.

Die Kontextwechsel vergleicht Abbildung 4.3. Die Plattform-Threads bilden auf jeder Stufe den höchsten Balken und erreichen bei 10000 Aufgaben 994037 Kontextwechsel. Die drei übrigen Modelle liegen auf derselben Stufe zwischen 18876 bei Node.js und 38838 bei Go.
Ein anderes Bild zeigt sich bei der Seitenmigration über die NUMA-Knoten. Hier migrieren bei 10000 Aufgaben bei den Virtual Threads 360011 Speicherseiten, bei Go 28626, bei den Plattform-Threads 10044 und bei Node.js 3907.

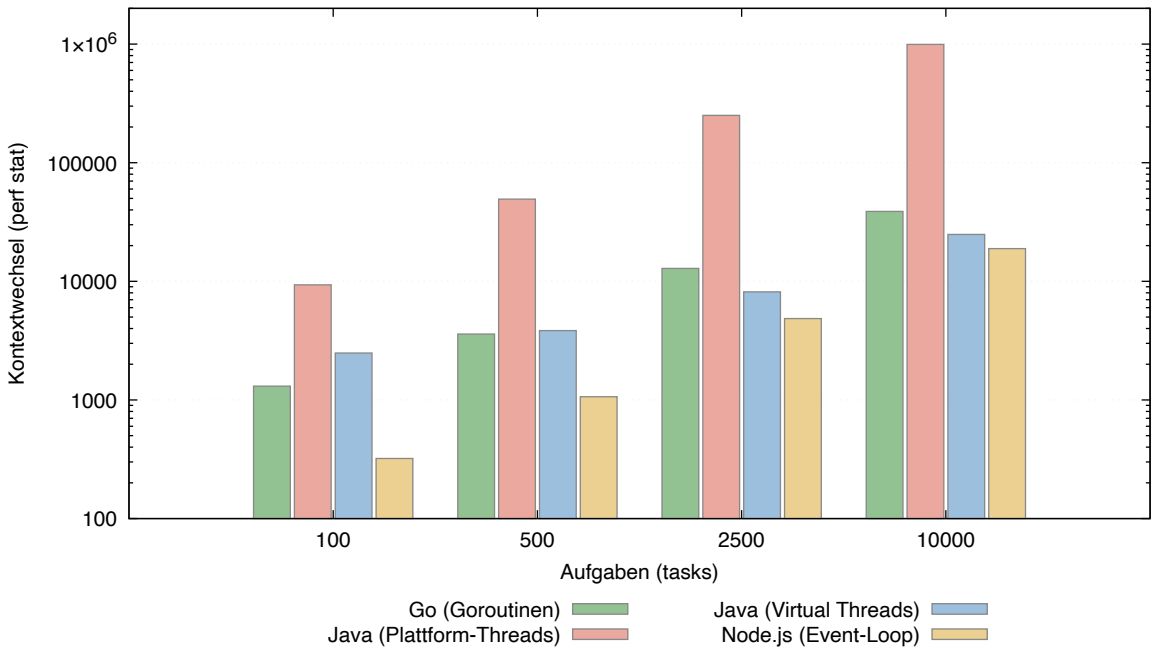

Abbildung 4.3: Rechengebundene Last, Kontextwechsel über die Aufgabenstufen

Die aufgezeichnete Scheduling-Latenz für die beiden oberen Aufgabenstufen fasst Tabelle 4.8 zusammen, Abbildung 4.4 zeigt die mittlere Verzögerung zusätzlich als Balkendiagramm. Die mittlere Verzögerung der Plattform-Threads beträgt 19,245 Millisekunden bei 2500 und 19,861 Millisekunden bei 10000 Aufgaben, während die drei übrigen Modelle auf beiden Stufen unter 0,3 Millisekunden bleiben. Die maximale Verzögerung reicht bei 10000 Aufgaben von 0,898 Millisekunden bei Node.js bis 64,001 Millisekunden bei den Plattform-Threads. Auch die Zahl der aufgezeichneten Wechsel ist bei den Plattform-Threads mit 240591 bei 2500 und 1026403 bei 10000 Aufgaben größer als bei den übrigen Modellen.

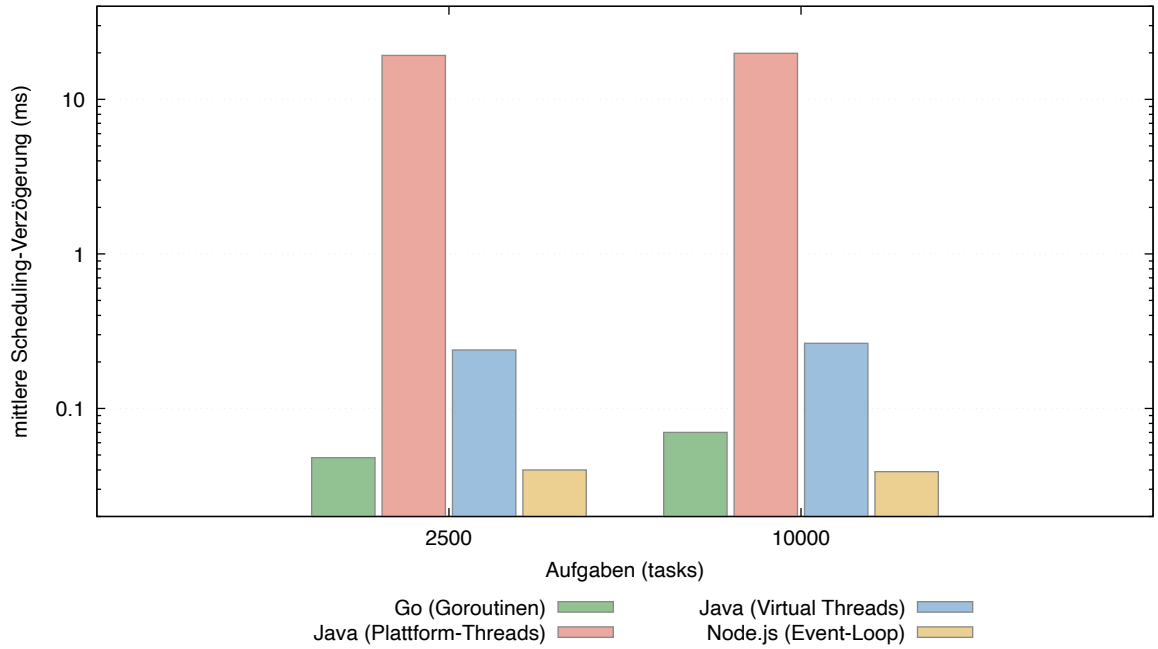

Abbildung 4.4: Rechengebundene Last, mittlere Scheduling-Verzögerung

Tabelle 4.8: Rechengebundene Last, Scheduling (perf sched) bei 2500 und 10000 Aufgaben

Modell	Aufgaben	Wechsel	avg (ms)	max (ms)
2500 Aufgaben				
Go (Goroutinen)	2500	10443	0,048	4,339
Java (Plattform-Threads)	2500	240591	19,245	59,971
Java (Virtual Threads)	2500	3110	0,239	11,375
Node.js (Event-Loop)	2500	4833	0,040	0,603

Fortsetzung auf nächster Seite

Tabelle 4.8: Rechengebundene Last, Scheduling (perf sched) bei 2500 und 10000 Aufgaben

Modell	Aufgaben	Wechsel	avg (ms)	max (ms)
10000 Aufgaben				
Go (Goroutinen)	10000	38847	0,070	8,006
Java (Plattform-Threads)	10000	1026403	19,861	64,001
Java (Virtual Threads)	10000	12009	0,264	11,457
Node.js (Event-Loop)	10000	18753	0,039	0,898

Um die Werte unabhängig von der Zahl der gleichzeitig genutzten Kerne vergleichen zu können, zeigt Tabelle 4.9 und Abbildung 4.5 die je Aufgabe aufgewendete CPU-Zeit. Sie ergibt sich aus der Summe der im User- und im Kernel-Mode verbrachten CPU-Zeit geteilt durch die Zahl der Aufgaben und ist damit unabhängig davon, auf wie vielen Kernen die Aufgaben parallel liefen. Node.js weist mit 159,5 bis 172,3 Millisekunden die kürzeste CPU-Zeit je Aufgabe auf, gefolgt von Go mit 198,7 bis 210,7 Millisekunden und den Virtual Threads mit 219,4 bis 277,0 Millisekunden, während die Plattform-Threads mit 315,6 bis 398,0 Millisekunden die längste CPU-Zeit je Aufgabe aufweisen. Über die Aufgabenstufen bleibt dieser Wert bei Go annähernd konstant und fällt bei Node.js leicht von 172,3 auf 159,5 Millisekunden. Bei den Virtual Threads liegt die erste Stufe mit 277,0 Millisekunden höher, ab 500 Aufgaben pendelt der Wert um 220 Millisekunden. Bei den Plattform-Threads steigt er über die Stufen von 315,6 auf 398,0 Millisekunden.

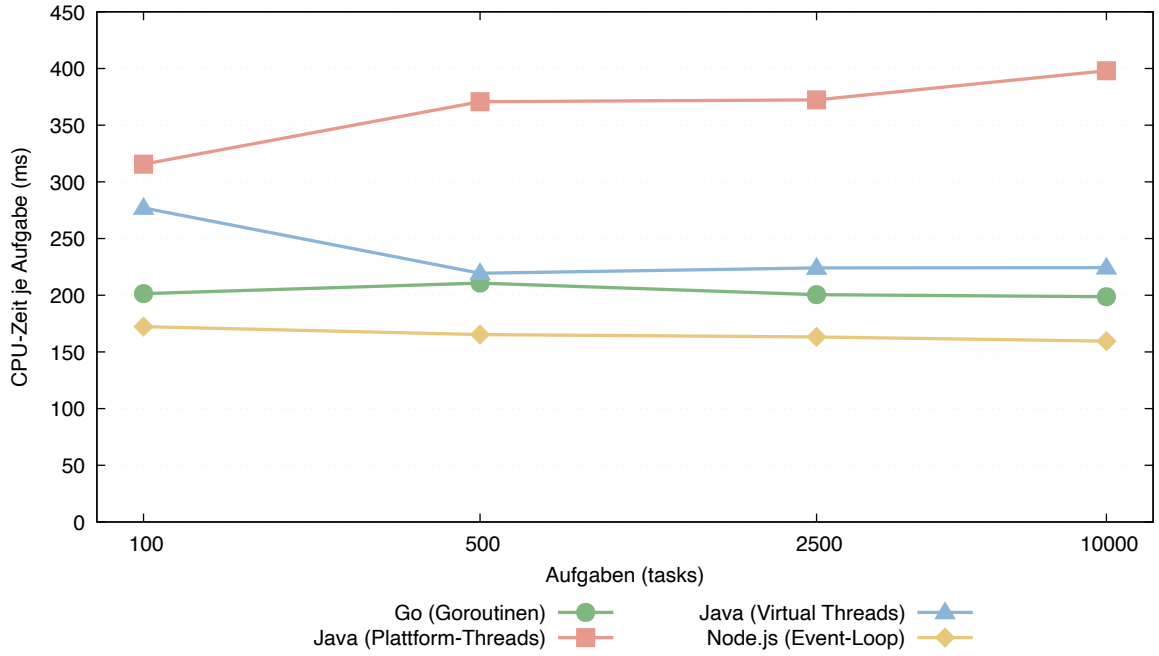

Abbildung 4.5: Rechengebundene Last, CPU-Zeit je Aufgabe über die Aufgabenstufen

Tabelle 4.9: Rechengebundene Last, CPU-Zeit je Aufgabe in Millisekunden

Aufgaben	Go	Java (Plattform)	Java (Virtual)	Node.js
100	201,4	315,6	277,0	172,3
500	210,7	370,7	219,4	165,3
2500	200,5	372,3	224,1	163,2
10000	198,7	398,0	224,3	159,5

Nicht alle erfassten Kennzahlen sind hier in Tabellen oder Abbildungen aufgeführt. Ein Teil beschreibt die feste Konfiguration und ist bereits im Messplan enthalten, etwa `bufbytes` und `steps`. Ein weiterer Teil dient als Kontrolle und blieb über alle Läufe bei den erwarteten Festwerten. So lag der ausgelagerte Speicher `vmswap_kb` durchgehend bei null und die Zahl der Major Page Faults (`major_faults`) ebenfalls, sodass im Messfenster kein Zugriff auf den Auslagerungsspeicher stattfand und der Anteil der im Kernel-Mode verbrachten CPU-Zeit `kernel_ratio_pct` lag für alle Modelle unter 0,35 Prozent. Andere Größen liegen aus mehreren Quellen vor, sodass jeweils nur eine dargestellt wird. Für die Kontextwechsel und die Scheduling-Wartezeiten wurden die prozessweiten Zähler von `perf` verwendet, da die über `/proc` je Thread gebildeten Differenzen `ctx_vol_delta`, `ctx_nonvol_delta`, `sched_cpu_delta_s` und `sched_wait_delta_s` bei den Plattform-Threads durch früh beendete Threads teils negative Werte annehmen und der Zähler `sched_switch` entspricht dem bereits gezeigten `context_switches`. Aus der Latenzverteilung wurden der Median und das 99. Perzentil dargestellt, während das Minimum `lat_min_us` und das Maximum `lat_max_us` entfielen, da das Maximum praktisch der Laufzeit des Messfensters entspricht. Bei den Hardware-Zählern wurden der Last-Level-Cache und der Daten-TLB gezeigt, während die eng verwandten L1-Daten-Cache-Misses `l1d_misses` und die Instruktions-TLB-Misses `itlb_misses` nur zur Kontrolle mitgeführt wurden. Die Zähler `task_clock_msec`, `raw_syscalls` und `sched_wakeup` sowie die block-bezogenen Zähler wurden für die rechengebundene Last nicht erfasst, da sie für diese Last nicht im Fokus standen und bewusst nicht bei jedem Testszenario vollständig erhoben wurden..

4.4 Ergebnisse der dateibasierten Last

Dieser Abschnitt stellt die Messergebnisse der dateibasierten Last dar. Der Aufbau folgt dem der rechengebundenen Last, zuerst werden die vier Modelle einzeln aufgeführt (Abschnitt 4.4.1), anschließend werden sie gegenübergestellt (Abschnitt 4.4.2). Gegenüber der rechengebundenen Last werden hier mehrere zusätzliche Kennzahlen erhoben, die erst bei I/O anfallen, insbesondere der Durchsatz in Mebibyte je Sekunde, der Anteil der auf Ein- und Ausgabe gewarteten Zeit (io_idle), der auf den Datenträger durchschlagende Anteil der gelesenen Bytes (device_ratio) und die an das Block-Device gerichteten Requests (block_rq). Umgekehrt entfallen die TLB-Zähler, die für dieses Szenario nicht aufgezeichnet wurden. Über alle Läufe

wurde je Aufgabenstufe dieselbe Gesamtzahl von zwei Millionen Lesezugriffen ausgeführt, was einer vom Block-Device gelesenen Datenmenge von rund 7,25 Gibibyte je Lauf entspricht. Die Prüfsummen aller vier Modelle stimmten je Stufe überein, etwa der Wert 227405683415 bei 100 Aufgaben. Die relative Streuung der perf-Zähler blieb überwiegend unter zwei Prozent. Der Lauf von Node.js bei 10000 Aufgaben wurde erneut als Einzelanker ohne Wiederholung gemessen.

4.4.1 Ergebnisse der einzelnen Modelle

Für jedes Modell werden im Folgenden der Durchsatz, die Laufzeit des Messfensters, der Median und das 99. Perzentil der Aufgabenlatenz und der Speicherbedarf über die vier Aufgabenstufen aufgeführt. Ergänzend werden im Text der Anteil der Wartezeit auf I/O, der Anteil der im Kernel-Mode verbrachten Zeit und die Zahl der Betriebssystem-Threads genannt.

Go erreicht über alle Stufen den höchsten Durchsatz aller Modelle, er liegt zwischen 654,5 und 675,4 MiB/s bei Laufzeiten zwischen 11,57 und 11,94 Sekunden. Der Median und das 99. Perzentil der Latenz liegen auf jeder Stufe nahe beieinander und betragen bei 10000 Aufgaben 11,30 gegenüber 11,92 Sekunden. Der Anteil der Wartezeit auf I/O fällt von 69,5 Prozent bei 100 Aufgaben auf rund 13,5 Prozent ab 500 Aufgaben, während der Anteil der im Kernel-Mode verbrachten Zeit ab 500 Aufgaben bei rund 92 Prozent liegt. Die Zahl der Betriebssystem-Threads bewegt sich zwischen 8 und 33 und der Speicher zwischen 18 und 94 MiB. Tabelle 4.10 enthält die Werte im Einzelnen.

Tabelle 4.10: Dateibasierte Last, Ergebnisse für Go (Goroutinen)

Aufgaben	Durchsatz (MiB/s)	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)
100	675,4	11,57	11,55	11,57	18
500	657,2	11,89	11,80	11,89	30
2500	654,5	11,94	11,61	11,92	46
10000	654,9	11,93	11,30	11,92	94

Die Plattform-Threads von Java erreichen einen Durchsatz nahe an Go, zwischen 644,3 und 668,0 MiB/s bei Laufzeiten zwischen 11,70 und 12,13 Sekunden. Die Zahl der Betriebssystem-Threads folgt der Aufgabenzahl und steigt von 141 bei 100 Aufgaben auf 10040 bei 10000 Aufgaben. Der Median der Latenz fällt über die Stufen von 11,66 auf 6,84 Sekunden, während das 99. Perzentil bei rund 12 Sekunden bleibt, sodass der Abstand zwischen beiden mit der Aufgabenzahl wächst. Der Anteil der Wartezeit auf I/O fällt von 68,5 Prozent auf 10,5 Prozent, der Kernel-Anteil liegt ab 500 Aufgaben bei rund 90 Prozent. Der verwendete Speicher steigt von 181 auf 1598 MiB. Tabelle 4.11 führt die Werte auf.

Tabelle 4.11: Dateibasierte Last, Ergebnisse für Java (Plattform-Threads)

Aufgaben	Durchsatz (MiB/s)	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)
100	668,0	11,70	11,66	11,69	181
500	648,0	12,06	11,89	12,04	376
2500	646,7	12,08	10,19	12,06	744
10000	644,3	12,13	6,84	12,11	1598

Die Virtual Threads erreichen einen Durchsatz zwischen 471,6 und 541,8 MiB/s, der von der ersten zur zweiten Stufe steigt und danach annähernd konstant bleibt. Die Laufzeit fällt von 16,57 auf 14,42 Sekunden. Die Zahl der Betriebssystem-Threads bewegt sich zwischen 53 und 86 und folgt nicht der Aufgabenzahl. Zwischen Median und 99. Perzentil der Latenz liegt auf jeder Stufe ein Abstand, bei 10000 Aufgaben 7,47 gegenüber 14,29 Sekunden. Der Anteil der Wartezeit auf I/O bleibt über alle Stufen zwischen 72,4 und 76,0 Prozent, der verwendete Speicher zwischen 142 und 249 MiB. Tabelle 4.12 zeigt die Werte.

Tabelle 4.12: Dateibasierte Last, Ergebnisse für Java (Virtual Threads)

Aufgaben	Durchsatz (MiB/s)	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)
100	471,6	16,57	9,51	16,55	145
500	528,8	14,78	7,87	14,57	142
2500	540,8	14,45	7,50	14,34	150
10000	541,8	14,42	7,47	14,29	249

Node.js läuft mit einer konfigurierten Parallelität von vier, die dem libuv-Threadpool entspricht und hält die Zahl der Betriebssystem-Threads über alle Stufen konstant bei 11. Der Durchsatz liegt zwischen 41,2 und 45,7 MiB/s und ist auf jeder Stufe der niedrigste aller Modelle, die Laufzeit liegt zwischen 171,03 und 189,85 Sekunden. Der Median und das 99. Perzentil der Latenz liegen auf jeder Stufe dicht beieinander. Der Anteil der ungenutzten Maschinenkapazität liegt zwischen 94,2 und 95,1 Prozent, der Kernel-Anteil zwischen 26,1 und 29,4 Prozent. Der verwendete Speicher steigt von 87 auf 325 MiB. Tabelle 4.13 zeigt die Werte.

Tabelle 4.13: Dateibasierte Last, Ergebnisse für Node.js (Event-Loop)

Aufgaben	Durchsatz (MiB/s)	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)
100	41,2	189,85	189,85	189,85	87
500	45,7	171,03	171,02	171,03	110
2500	42,8	182,71	182,64	182,71	158
10000	45,1	173,31	172,96	173,30	325

4.4.2 Vergleich der Modelle

Die Gegenüberstellung fasst die vier Modelle über alle Aufgabenstufen zusammen und stützt sich auf den Durchsatz, die Latenzverteilung, den Anteil der Wartezeit auf I/O, ausgewählte Hardware- und Block-Counter und die aufgezeichnete Scheduling-Latenz.

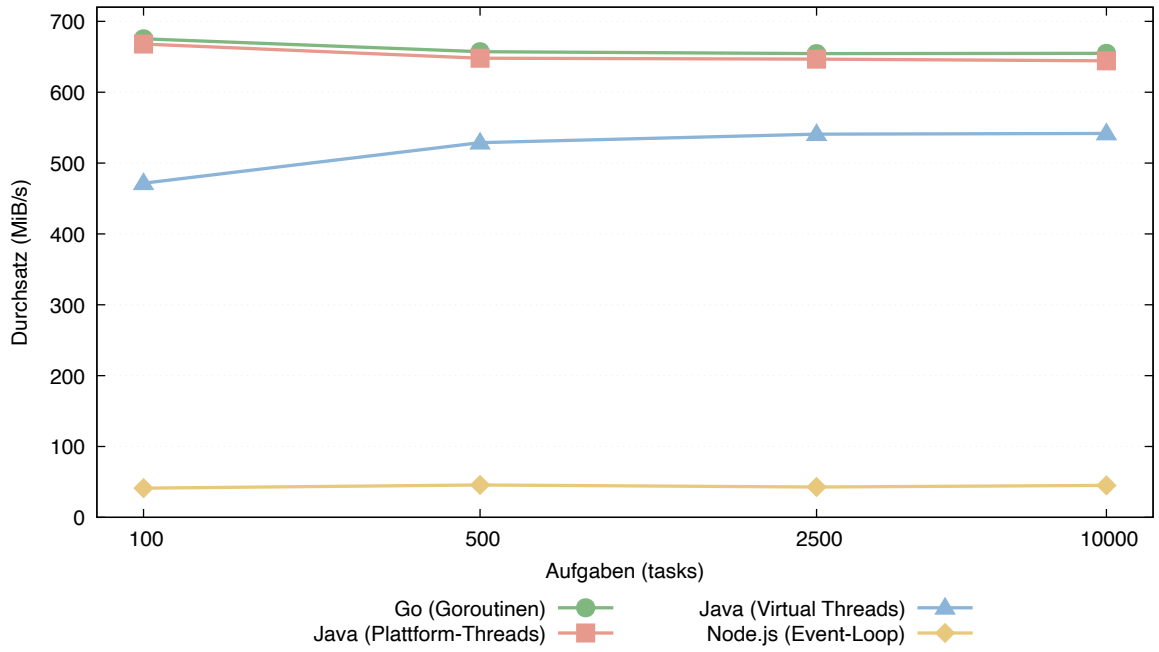

Abbildung 4.6: Dateibasierte Last, Durchsatz in Mebibyte je Sekunde über die Aufgabenstufen

Abbildung 4.6 stellt den Durchsatz in Mebibyte je Sekunde über die Aufgabenstufen dar. Go und die Plattform-Threads liegen mit rund 644 bis 675 MiB/s eng beieinander und bilden die beiden oberen Verläufe. Die Virtual Threads liegen mit 471,6 bis 541,8 MiB/s darunter, wobei ihr Durchsatz von der ersten zur zweiten Stufe steigt. Node.js verläuft zwischen 41,2 und 45,7 MiB/s und bildet über alle Stufen den untersten Verlauf.

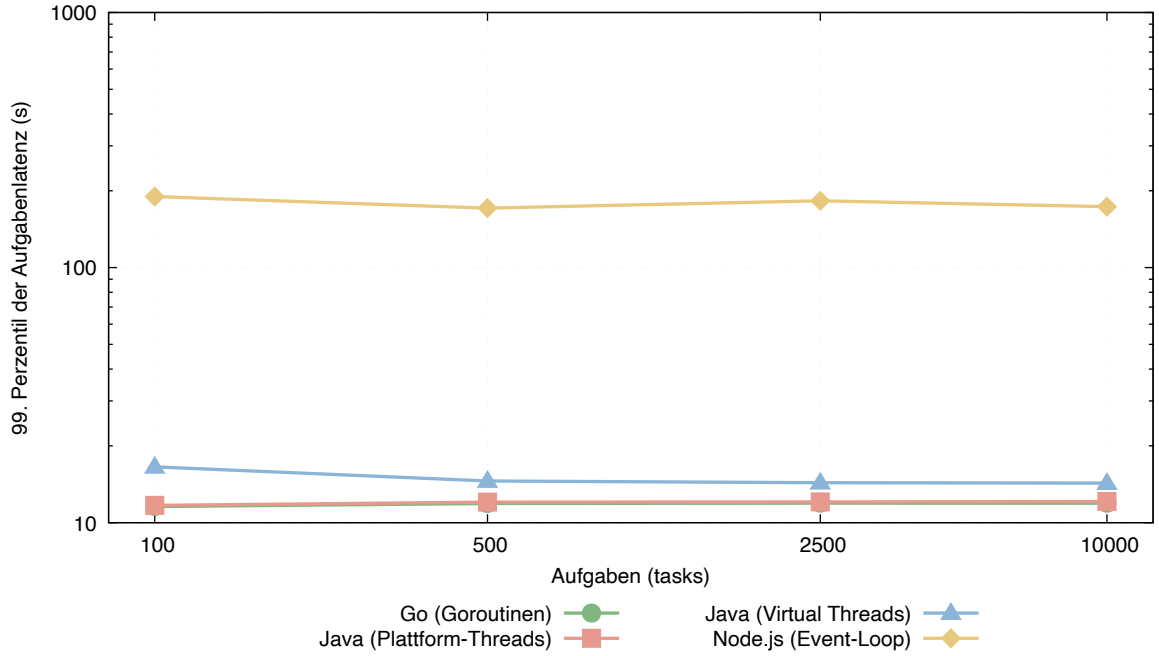

Abbildung 4.7: Dateibasierte Last, 99. Perzentil der Aufgabenlatenz über die Aufgabenstufen

Die Latenz zeigt Abbildung 4.7 anhand des 99. Perzentils. Die Verläufe von Go, den Plattform-Threads und den Virtual Threads liegen zwischen 11,57 und 16,55 Sekunden eng beieinander, während der Verlauf von Node.js über den gesamten Bereich um mehr als eine Größenordnung darüber liegt und zwischen 171,03 und 189,85 Sekunden verläuft.

Einen Blick auf die Hardware- und Block-Layer erlaubt Tabelle 4.14, die ausgewählte Zähler bei der höchsten Stufe von 10000 Aufgaben gegenüberstellt. Node.js erreicht mit 0,99 die höchste Zahl an Instruktionen je Zyklus, gefolgt von den Virtual Threads mit 0,63, während Go und die Plattform-Threads mit 0,23 und 0,19 darunter liegen. Die Zahl der CPU-Migrationen reicht von 957 bei Node.js über 40297 bei den Virtual Threads und 791310 bei den Plattform-Threads bis 1334175 bei Go. Die Misses im Last-Level-Cache liegen mit rund 133 Millionen bei den Virtual Threads am niedrigsten, gefolgt von rund 286 Millionen bei Go, rund 314 Millionen bei den Plattform-Threads und rund 358 Millionen bei Node.js, während der Daten-TLB für dieses Szenario nicht aufgezeichnet wurde. Die an das Block-Device gerichteten Requests (block_rq_issue) liegen mit 1,75 bis 1,89 Millionen für alle Modelle nahe beieinander, während die abgeschlossenen Requests (block_rq_complete) von 0,01 Millionen bei Node.js über 0,25 Millionen bei den Virtual Threads bis 1,65 und 1,71 Millionen bei Go und den Plattform-Threads reichen.

Tabelle 4.14: Dateibasierte Last, ausgewählte perf-Zähler bei 10000 Aufgaben

Modell	Instr. (10^9)	IPC	CPU-Migr.	Kontext-wechsel	LLC-Misses (10^6)	block_rq_issue (10^6)	block_rq_complete (10^6)
Go (Goroutinen)	185,2	0,23	1334175	3251994	286	1,75	1,65
Java (Plattform-Threads)	166,7	0,19	791310	2711004	314	1,76	1,71
Java (Virtual Threads)	108,1	0,63	40297	1892891	133	1,89	0,25
Node.js (Event-Loop)	599,1	0,99	957	3942375	358	1,89	0,01

Den Anteil der ungenutzten Maschinenkapazität zeigt Abbildung 4.8. Der Wert bezieht alle vier Modelle auf dieselbe Grundlage, nämlich die 32 logischen Kerne des Testsystems. Bei Go und den Plattform-Threads fällt dieser Anteil von rund 69 Prozent bei 100 Aufgaben auf rund 11 bis 14 Prozent ab 500 Aufgaben. Bei den Virtual Threads bleibt er über alle Stufen zwischen 72,4 und 76,0 Prozent, bei Node.js zwischen 94,2 und 95,1 Prozent.

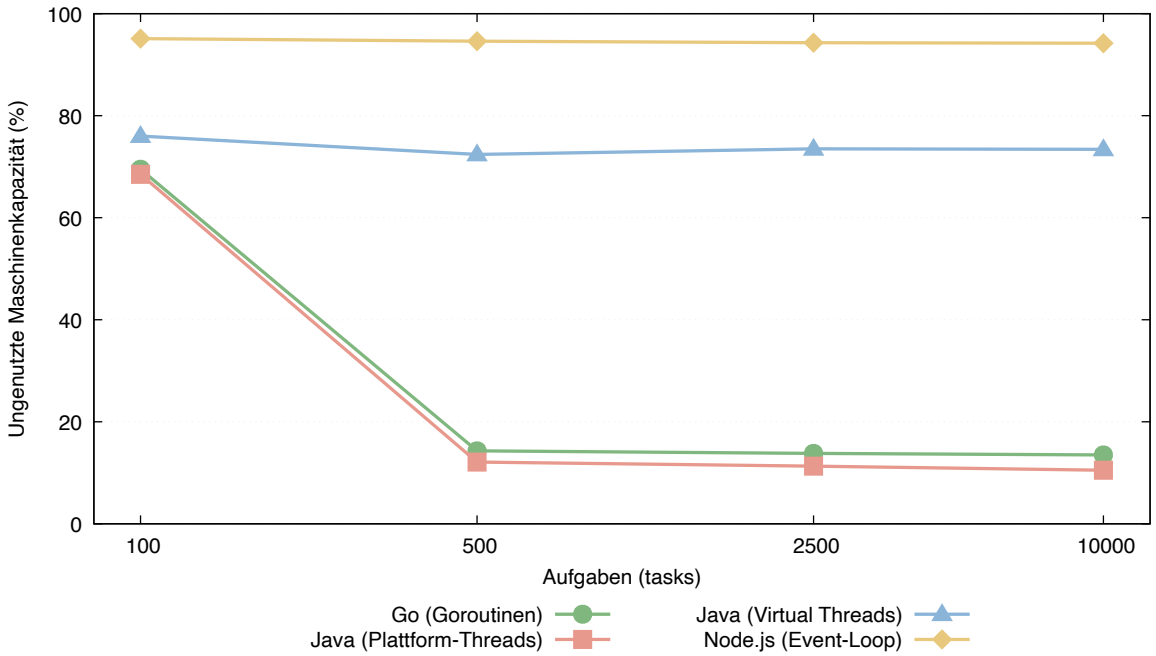

Abbildung 4.8: Dateibasierte Last, Anteil der ungenutzten Maschinenkapazität über die Tasks

Die aufgezeichnete Scheduling-Latenz für die beiden oberen Aufgabenstufen fasst Tabelle 4.15 zusammen, Abbildung 4.9 zeigt die mittlere Verzögerung zusätzlich als Balkendiagramm. Die mittlere Verzögerung der Plattform-Threads beträgt 5,634 Millisekunden bei 2500 und 2,614 Millisekunden bei 10000 Aufgaben, während die drei übrigen Modelle auf beiden Stufen unter 0,2 Millisekunden bleiben. Die maximale Verzögerung reicht bei 2500 Aufgaben von 0,401 Millisekunden bei Node.js bis 278,446 Millisekunden bei den Plattform-Threads. Die Zahl der aufgezeichneten Wechsel liegt bei Node.js mit rund 3,84 bis 3,87 Millionen am höchsten und bei den Virtual Threads mit rund 1,89 Millionen am niedrigsten.

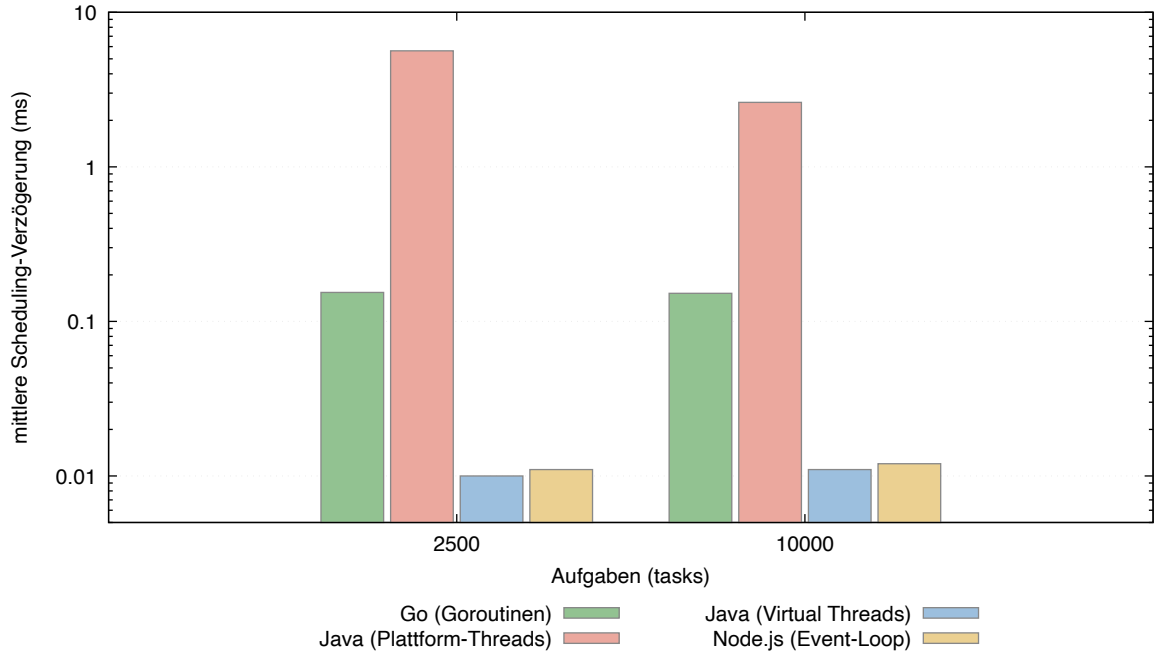

Abbildung 4.9: Dateibasierte Last, mittlere Scheduling-Verzögerung

Tabelle 4.15: Dateibasierte Last, Scheduling bei 2500 und 10000 Aufgaben

Modell	Aufgaben	Wechsel	avg (ms)	max (ms)
2500 Aufgaben				
Go (Goroutinen)	2500	2749853	0,154	5,369
Java (Plattform-Threads)	2500	2314118	5,634	278,446
Java (Virtual Threads)	2500	1886251	0,010	5,310
Node.js (Event-Loop)	2500	3868762	0,011	0,401

Fortsetzung auf nächster Seite

Tabelle 4.15: Dateibasierte Last, Scheduling bei 2500 und 10000 Aufgaben

Modell	Aufgaben	Wechsel	avg (ms)	max (ms)
10000 Aufgaben				
Go (Goroutinen)	10000	2740419	0,152	6,321
Java (Plattform-Threads)	10000	2369067	2,614	190,868
Java (Virtual Threads)	10000	1886608	0,011	7,823
Node.js (Event-Loop)	10000	3843990	0,012	0,186

Wie bei der rechengebundenen Last lässt sich auch hier eine von der Zahl der gleichzeitig genutzten Kerne unabhängige Kennzahl bilden. Da die dateibasierte Last am Block-Device hängt und in jedem Lauf dieselbe Datenmenge gelesen wird, wird dazu nicht die Aufgabe, sondern die gelesene Datenmenge als Bezug gewählt. Tabelle 4.16 und Abbildung 4.10 zeigen die CPU-Zeit je gelesenem Gibibyte, also die Summe der im User- und im Kernel-Mode verbrachten CPU-Zeit geteilt durch die je Lauf gelesenen rund 7,25 Gibibyte. Die Virtual Threads weisen mit 16,9 bis 18,0 Sekunden je Gibibyte über alle Stufen den niedrigsten und annähernd konstanten Wert auf. Bei Go und den Plattform-Threads liegt der Wert bei 100 Aufgaben mit 15,6 und 16,2 Sekunden je Gibibyte in derselben Größenordnung, steigt ab 500 Aufgaben jedoch auf rund 45 beziehungsweise rund 47 Sekunden je Gibibyte. Node.js liegt zwischen 40,9 und 46,2 Sekunden je Gibibyte.

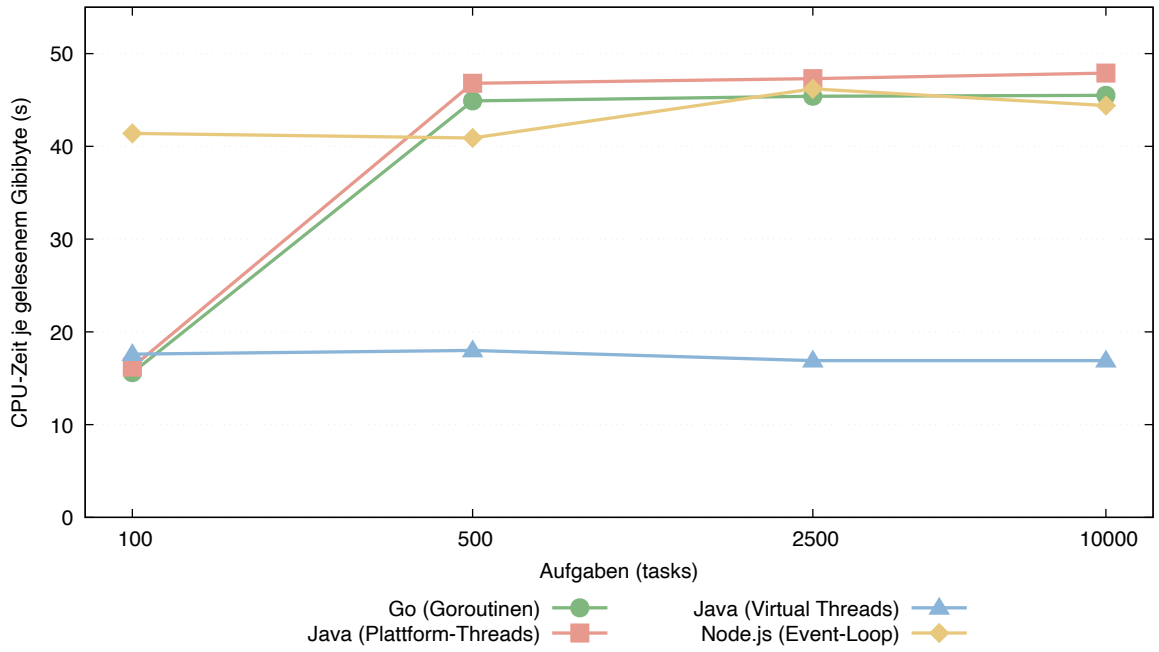

Abbildung 4.10: Dateibasierte Last, CPU-Zeit je gelesenem Gibibyte über die Aufgabenstufen

Tabelle 4.16: Dateibasierte Last, CPU-Zeit je gelesenem Gibibyte in Sekunden

Aufgaben	Go	Java (Plattform)	Java (Virtual)	Node.js
100	15,6	16,2	17,6	41,4
500	44,9	46,8	18,0	40,9
2500	45,4	47,3	16,9	46,2
10000	45,5	47,9	16,9	44,4

Auch in diesem Abschnitt werden nicht alle erfassten Kennzahlen in die Tabellen oder Abbildungen aufgenommen. Ein Teil beschreibt, wie bereits erwähnt, die feste Konfiguration und ist bereits im Messplan enthalten. Die Kontrollwerte blieben über alle Läufe im erwarteten Bereich. So lag der ausgelagerte Speicher `vmswap_kb` durchgehend bei null, die vom Block-Device gelesene Datenmenge `read_bytes_delta` je Lauf bei rund 7,25 Gibibyte, die Zahl der Lese-Systemaufrufe `syscr_delta` bei rund zwei Millionen und der auf den Datenträger durchschlagende Anteil der gelesenen Bytes `device_ratio_pct` bei rund 95 Prozent, was als Kontrolle gegen eine Bedienung aus dem Page-Cache dient. Die Zahl der Major Page Faults (`major_faults`) blieb mit höchstens 79 klein. Wie im vorherigen Abschnitt beschrieben liegen bestimmte Metriken aus mehreren Quellen vor wovon nur jeweils eine dargestellt wird.

4.5 Ergebnisse der gemischten Last

Dieser Abschnitt stellt die Messergebnisse der gemischten Last dar. Der Aufbau folgt dem der beiden vorherigen Szenarien, zuerst werden die vier Modelle einzeln aufgeführt (Abschnitt 4.5.1), anschließend werden sie gegenübergestellt (Abschnitt 4.5.2). Die gemischte Last vereint rechengebundene und dateibasierte Arbeit, da jede Iteration sowohl Rechenschritte als auch einen Lesezugriff ausführt. Deshalb fallen hier die Kennzahlen beider vorherigen Szenarien gemeinsam an, also sowohl die TLB-Zähler der rechengebundenen Last als auch der Durchsatz, der Anteil der Wartezeit auf Ein- und Ausgabe (io_idle), der auf den Datenträger durchschlagende Anteil der gelesenen Bytes (device_ratio) und die Block-Requests (block_rq) der dateibasierten Last. Zusätzlich liegt neben dem Read-Throughput auch die Zahl der abgeschlossenen Aufgaben je Sekunde vor. Über alle Läufe wurde je Aufgabenstufe dieselbe Gesamtzahl von zwei Millionen Iterationen ausgeführt, was einer vom Block-Device gelesenen Datenmenge von rund 7,25 Gibibyte je Lauf entspricht. Die Prüfsummen aller vier Modelle stimmten je Stufe überein, etwa der Wert 171445596215 bei 100 Aufgaben. Die relative Streuung der perf-Zähler blieb überwiegend unter zwei Prozent. Der Lauf von Node.js bei 10000 Aufgaben wurde erneut als Einzelanker ohne Wiederholung gemessen.

4.5.1 Ergebnisse der einzelnen Modelle

Für jedes Modell werden im Folgenden der Read-Throughput, die Laufzeit des Messfensters, der Median und das 99. Perzentil der Aufgabenlatenz und der Speicherbedarf über die vier Aufgabenstufen aufgeführt. Ergänzend werden im Text die Zahl der abgeschlossenen Aufgaben je Sekunde, der Anteil der Wartezeit auf I/O, der Anteil der im Kernel-Mode verbrachten Zeit und die Zahl der Betriebssystem-Threads genannt.

Go liest über die vier Stufen mit 149,5 bis 174,6 MiB/s bei Laufzeiten zwischen 44,74 und 52,26 Sekunden. Der Median und das 99. Perzentil der Latenz liegen auf jeder Stufe nahe beieinander und betragen bei 10000 Aufgaben 49,45 gegenüber 51,53 Sekunden. Der Anteil der Wartezeit auf I/O liegt über alle Stufen zwischen 15,4 und 17,3 Prozent, der Kernel-Anteil zwischen 4,0 und 5,0 Prozent. Die Zahl der Betriebssystem-Threads bleibt zwischen 33 und 34, die Zahl der abgeschlossenen Aufgaben je Sekunde steigt von 2,2 auf 194,0 und der verwendete Speicher steigt auf 10221 MiB. Tabelle 4.17 enthält die Werte im Einzelnen.

Tabelle 4.17: Gemischte Last, Ergebnisse für Go (Goroutinen)

Aufgaben	Durchsatz (MiB/s)	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)
100	174,6	44,74	44,43	44,70	120
500	152,9	51,11	50,65	51,09	521
2500	149,5	52,26	51,06	52,25	2548
10000	151,6	51,55	49,45	51,53	10221

Die Plattform-Threads von Java lesen mit 109,2 bis 184,1 MiB/s, wobei der Durchsatz von der ersten zur zweiten Stufe zurückgeht. Die Zahl der Betriebssystem-Threads folgt der Aufgabenzahl und steigt von 142 bei 100 Aufgaben auf 10045 bei 10000 Aufgaben. Der Median der Latenz fällt über die Stufen von 38,69 auf 33,41 Sekunden, während das 99. Perzentil bei 10000 Aufgaben 67,57 Sekunden beträgt. Der Anteil der Wartezeit auf I/O liegt zwischen 0,4 und 4,4 Prozent. Die Zahl der abgeschlossenen Aufgaben je Sekunde steigt auf 147,9 und der verwendete Speicher auf 15513 MiB, den höchsten Wert aller Modelle. Tabelle 4.18 führt die Werte auf.

Tabelle 4.18: Gemischte Last, Ergebnisse für Java (Plattform-Threads)

Aufgaben	Durchsatz (MiB/s)	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)
100	184,1	42,43	38,69	42,28	286
500	120,8	64,67	59,17	64,55	1448
2500	109,2	71,57	57,30	71,53	3639
10000	115,5	67,63	33,41	67,57	15513

Die Virtual Threads lesen mit 274,2 bis 306,3 MiB/s und damit über alle Stufen am schnellsten, bei den kürzesten Laufzeiten zwischen 25,50 und 28,50 Sekunden. Die Zahl der Betriebssystem-Threads bewegt sich zwischen 74 und 82 und folgt nicht der Aufgabenzahl. Median und 99. Perzentil der Latenz betragen bei 10000 Aufgaben 13,68 gegenüber 26,65 Sekunden. Der Anteil der Wartezeit auf I/O liegt zwischen 26,5 und 36,4 Prozent. Die Zahl der abgeschlossenen Aufgaben je Sekunde steigt auf 371,9, den höchsten Wert aller Modelle und der verwendete Speicher erreicht 12740 MiB. Tabelle 4.19 zeigt die Werte.

Tabelle 4.19: Gemischte Last, Ergebnisse für Java (Virtual Threads)

Aufgaben	Durchsatz (MiB/s)	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)
100	274,2	28,50	16,90	28,06	239
500	306,3	25,50	13,80	25,05	634
2500	296,5	26,35	13,63	26,15	3287
10000	290,6	26,89	13,68	26,65	12740

Node.js läuft mit einer konfigurierten Parallelität von vier, die dem libuv-Threadpool entspricht und hält die Zahl der Betriebssystem-Threads über alle Stufen konstant bei 11. Der Read-Throughput liegt zwischen 4,1 und 4,4 MiB/s und ist auf jeder Stufe der niedrigste aller Modelle, die Laufzeit liegt zwischen 1784,08 und 1919,89 Sekunden. Der Median und das 99. Perzentil der Latenz liegen auf jeder Stufe dicht beieinander. Der Anteil der ungenutzten Maschinenkapazität liegt zwischen 96,4 und 96,5 Prozent, der Kernel-Anteil zwischen 7,6 und 10,4 Prozent. Die Zahl der abgeschlossenen Aufgaben je Sekunde steigt auf 5,2 und der verwendete Speicher auf 10218 MiB. Tabelle 4.20 zeigt die Werte.

Tabelle 4.20: Gemischte Last, Ergebnisse für Node.js (Event-Loop)

Aufgaben	Durchsatz (MiB/s)	Laufzeit (s)	Latenz p50 (s)	Latenz p99 (s)	VmHWM (MiB)
100	4,4	1784,08	1784,08	1784,08	197
500	4,3	1826,25	1826,25	1826,25	601
2500	4,1	1896,15	1896,09	1896,15	2627
10000	4,1	1919,89	1919,55	1919,88	10218

4.5.2 Vergleich der Modelle

Die Gegenüberstellung fasst die vier Modelle über alle Aufgabenstufen zusammen. Da die gemischte Last beide Anteile verbindet, stützt sie sich sowohl auf die dateibasierten Größen, also den Durchsatz und den Anteil der Wartezeit auf I/O, als auch auf die rechengebundenen Größen der Hardware-Ebene, ergänzt um die Block-Counter und die aufgezeichnete Scheduling-Latenz.

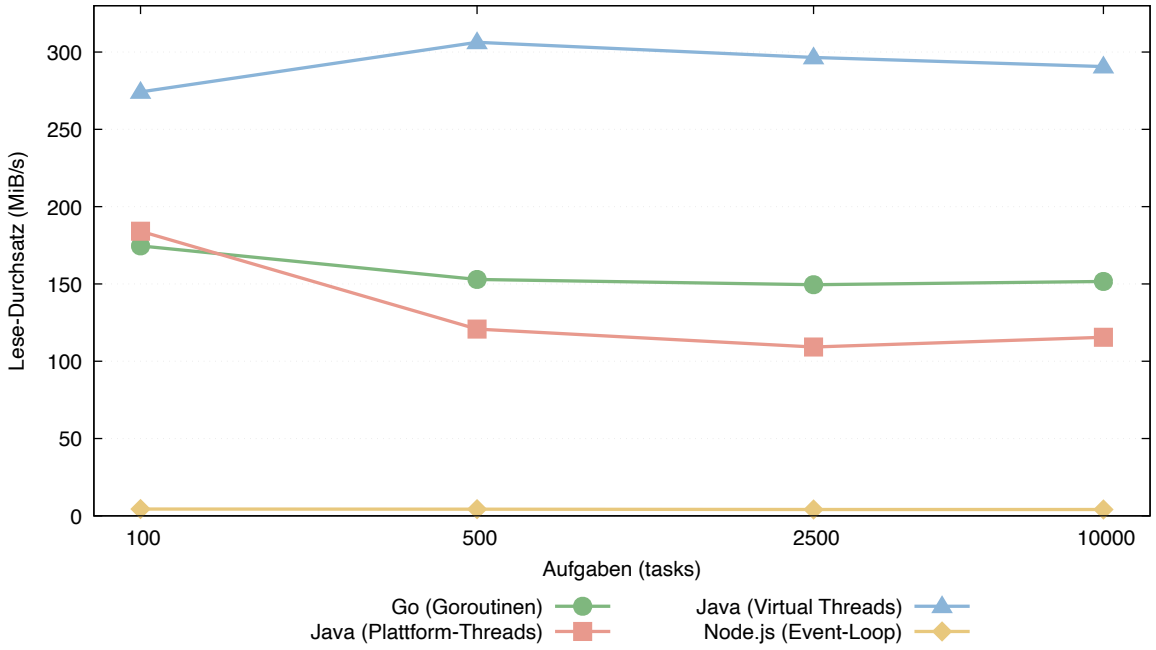

Abbildung 4.11: Gemischte Last, Read-Throughput in Mebibyte je Sekunde über die Tasks

Abbildung 4.11 stellt den Read-Throughput in Mebibyte je Sekunde über die Aufgabenstufen dar. Die Virtual Threads bilden mit 274,2 bis 306,3 MiB/s über alle Stufen den obersten Verlauf. Go liegt mit 149,5 bis 174,6 MiB/s darunter, die Plattform-Threads mit 109,2 bis 184,1 MiB/s

in derselben Größenordnung wie Go, wobei ihr Durchsatz von der ersten zur zweiten Stufe zurückgeht. Node.js verläuft zwischen 4,1 und 4,4 MiB/s und bildet über alle Stufen den untersten Verlauf.

Die Latenz zeigt Abbildung 4.12 anhand des 99. Perzentils. Die Verläufe der Virtual Threads (25,05 bis 28,06 Sekunden), von Go (44,70 bis 52,25 Sekunden) und den Plattform-Threads (42,28 bis 71,53 Sekunden) liegen enger beieinander, während der Verlauf von Node.js über den gesamten Bereich um mehr als eine Größenordnung darüber liegt und zwischen 1784,08 und 1919,88 Sekunden verläuft.

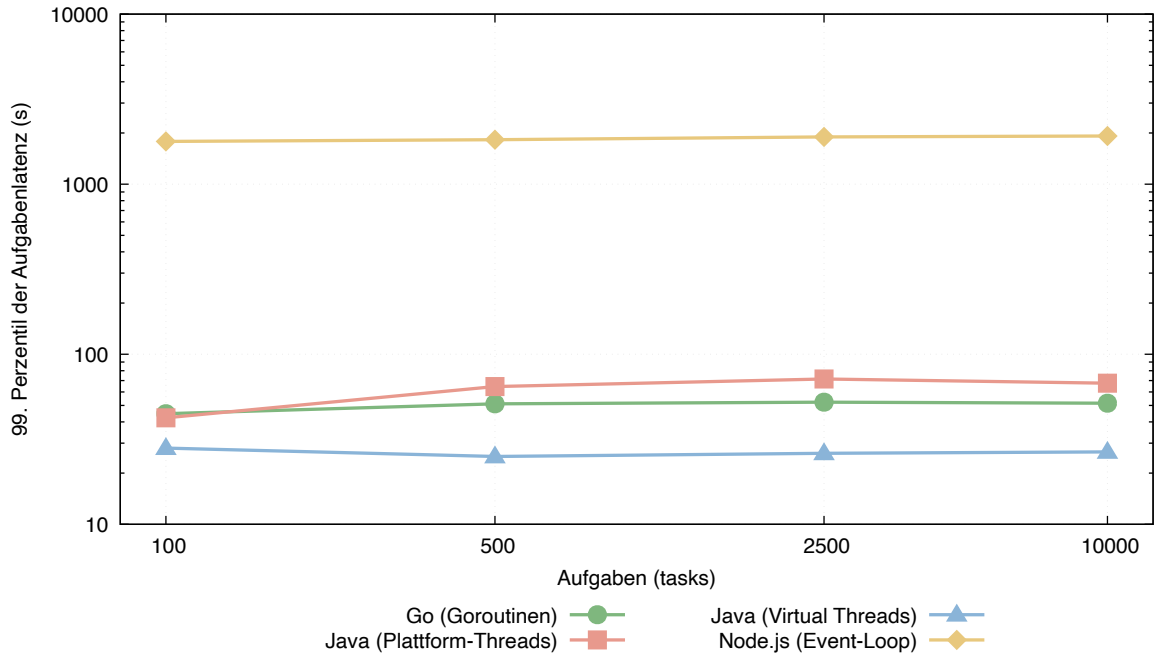

Abbildung 4.12: Gemischte Last, 99. Perzentil der Aufgabenlatenz über die Aufgabenstufen

Die Hardware- und Block-Layer fasst Tabelle 4.21 bei der höchsten Stufe von 10000 Aufgaben zusammen. Node.js führt mit 2325,9 Milliarden die meisten Instruktionen aus und erreicht mit 0,39 die höchste Zahl an Instruktionen je Zyklus, gefolgt von den Virtual Threads mit 0,19, Go mit 0,12 und den Plattform-Threads mit 0,09. Die Misses im Last-Level-Cache reichen von 1610,7 Millionen bei den Virtual Threads bis 17132,4 Millionen bei Node.js, die dTLB-Misses von 43,3 Millionen bei den Virtual Threads bis 1057,8 Millionen bei Node.js. Die Zahl der CPU-Migrationen reicht von 807 bei Node.js bis 624684 bei Go. Die ausgegebenen Block-Requests (block_rq_issue) liegen mit rund 1,89 Millionen für alle Modelle gleichauf, während die abgeschlossenen Requests (block_rq_complete) von rund 200 bei Node.js über 0,66 Millionen bei den Virtual Threads und 0,94 Millionen bei Go bis 1,88 Millionen bei den Plattform-Threads reichen.

Tabelle 4.21: Gemischte Last, ausgewählte perf-Zähler bei 10000 Aufgaben

Modell	Instr. (10^9)	IPC	CPU-Migr.	LLC-Misses (10^6)	dTLB-Misses (10^6)	block_rq_issue (10^6)	block_rq_complete (10^6)
Go (Goroutinen)	421,1	0,12	624 684	8 161,1	78,0	1,89	0,94
Java (Plattform-Threads)	343,6	0,09	82 544	11 031,5	258,1	1,89	1,88
Java (Virtual Threads)	312,7	0,19	12 565	1 610,7	43,3	1,89	0,66
Node.js (Event-Loop)	2325,9	0,39	807	17 132,4	1 057,8	1,89	<0,01

Den Anteil der ungenutzten Maschinenkapazität zeigt Abbildung 4.13. Bei den Plattform-Threads liegt dieser Anteil über alle Stufen zwischen 0,4 und 4,4 Prozent, bei Go zwischen 15,4 und 17,3 Prozent, bei den Virtual Threads zwischen 26,5 und 36,4 Prozent und bei Node.js zwischen 96,4 und 96,5 Prozent.

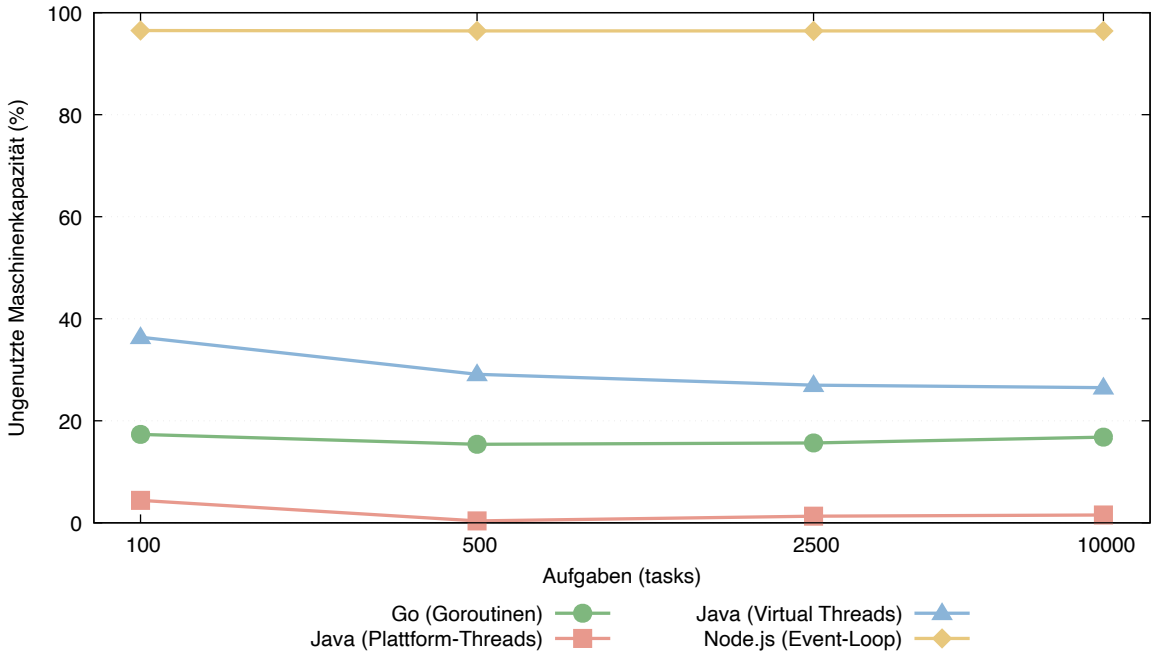

Abbildung 4.13: Gemischte Last, Anteil der ungenutzten Maschinenkapazität über die Tasks

Die aufgezeichnete Scheduling-Latenz für die beiden oberen Aufgabenstufen fasst Tabelle 4.22 zusammen, Abbildung 4.14 zeigt die mittlere Verzögerung zusätzlich als Balkendiagramm. Für Node.js wurden zwei Arbeits-Threads aufgezeichnet, der libuv-Threadpool und der Haupt-Thread des Event-Loops, die Tabelle führt den libuv-Threadpool auf. Die mittlere Verzögerung der Plattform-Threads beträgt 30,855 Millisekunden bei 2500 und 42,064 Millisekunden bei 10000 Aufgaben, während die drei übrigen Modelle auf beiden Stufen unter 0,1 Millisekunden bleiben. Die maximale Verzögerung reicht bei 10000 Aufgaben von 0,434 Millisekunden bei Node.js bis 806,873 Millisekunden bei den Plattform-Threads.

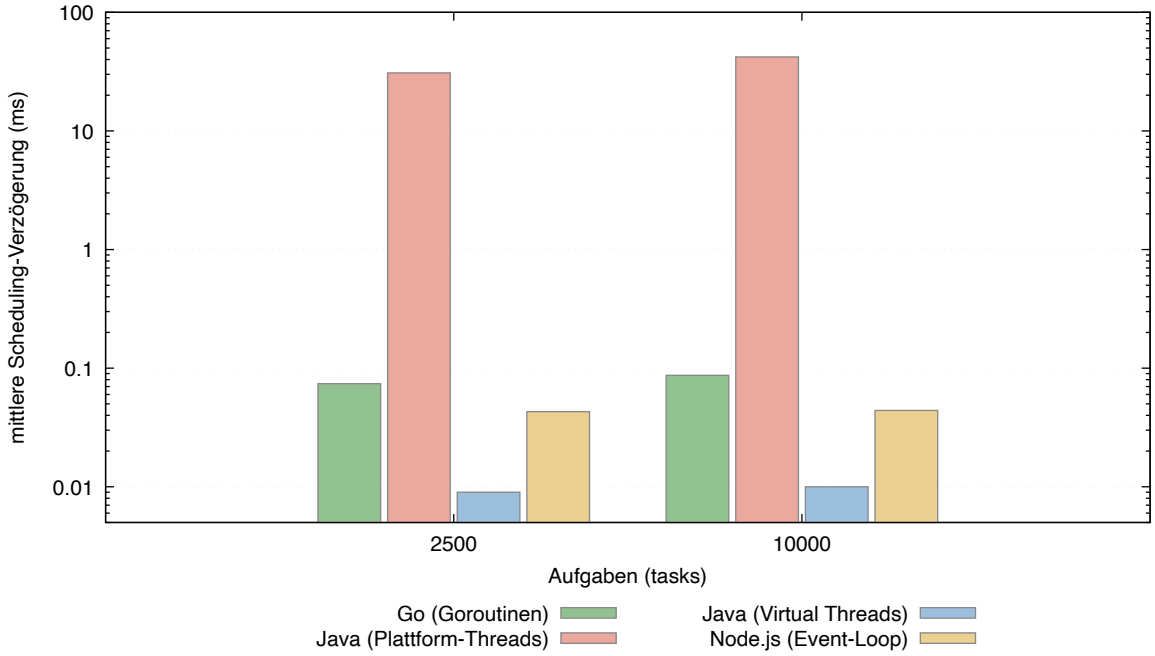

Abbildung 4.14: Gemischte Last, mittlere Scheduling-Verzögerung

Tabelle 4.22: Gemischte Last, Scheduling-Kennzahlen bei 2500 und 10000 Aufgaben

Modell	Aufgaben	Wechsel	avg (ms)	max (ms)
2500 Aufgaben				
Go (Goroutinen)	2500	2967147	0,074	5,937
Java (Plattform-Threads)	2500	2097390	30,855	322,498
Java (Virtual Threads)	2500	1894198	0,009	8,177
Node.js (Event-Loop)	2500	3881349	0,043	1,366

Fortsetzung auf nächster Seite

Tabelle 4.22: Gemischte Last, Scheduling-Kennzahlen bei 2500 und 10000 Aufgaben

Modell	Aufgaben	Wechsel	avg (ms)	max (ms)
10000 Aufgaben				
Go (Goroutinen)	10000	2926732	0,087	21,461
Java (Plattform-Threads)	10000	2167802	42,064	806,873
Java (Virtual Threads)	10000	1894656	0,010	9,997
Node.js (Event-Loop)	10000	3881600	0,044	0,434

Wie bei den beiden vorherigen Szenarien lässt sich auch hier eine von der Zahl der gleichzeitig genutzten Kerne unabhängige Kennzahl bilden. Da die gemischte Last je Lauf dieselbe Datenmenge liest, wird die insgesamt verbrauchte CPU-Zeit auf die gelesene Datenmenge bezogen. Tabelle 4.23 und Abbildung 4.15 zeigen die CPU-Zeit je gelesenem Gibibyte, also die Summe der im User- und im Kernel-Mode verbrachten CPU-Zeit geteilt durch die je Lauf gelesenen rund 7,25 Gibibyte. Die Virtual Threads weisen mit 79,7 bis 87,2 Sekunden je Gibibyte über alle Stufen den niedrigsten und annähernd konstanten Wert auf. Go liegt zwischen 163,2 und 194,5 Sekunden je Gibibyte, die Plattform-Threads zwischen 179,0 und 311,6 Sekunden je Gibibyte und Node.js zwischen 276,1 und 303,8 Sekunden je Gibibyte.

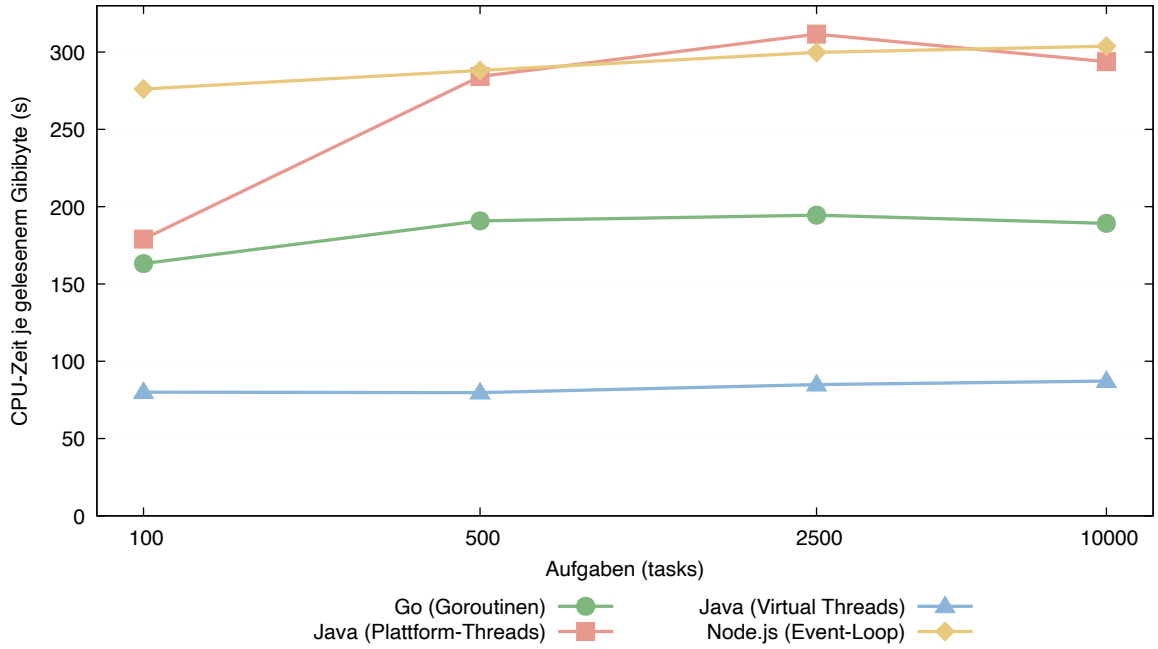

Abbildung 4.15: Gemischte Last, CPU-Zeit je gelesenem Gibibyte über die Aufgabenstufen

Tabelle 4.23: Gemischte Last, CPU-Zeit je gelesenem Gibibyte in Sekunden

Aufgaben	Go	Java (Plattform)	Java (Virtual)	Node.js
100	163,2	179,0	80,0	276,1
500	190,8	284,2	79,7	288,1
2500	194,5	311,6	84,9	299,8
10000	189,2	293,8	87,2	303,8

Da die gemischte Last rechnende und lesende Anteile verbindet, lässt sich diese CPU-Zeit je Gibibyte weiter in den im User-Mode und den im Kernel-Mode verbrachten Anteil aufteilen. Abbildung 4.16 zeigt beide Anteile bei der höchsten Stufe von 10000 Aufgaben. Der User-Anteil überwiegt bei allen Modellen, er reicht von 80,6 Sekunden je Gibibyte bei den Virtual Threads bis 282,4 Sekunden je Gibibyte bei den Plattform-Threads. Der Kernel-Anteil ist kleiner und reicht von 6,6 Sekunden je Gibibyte bei den Virtual Threads bis 31,7 Sekunden je Gibibyte bei Node.js.

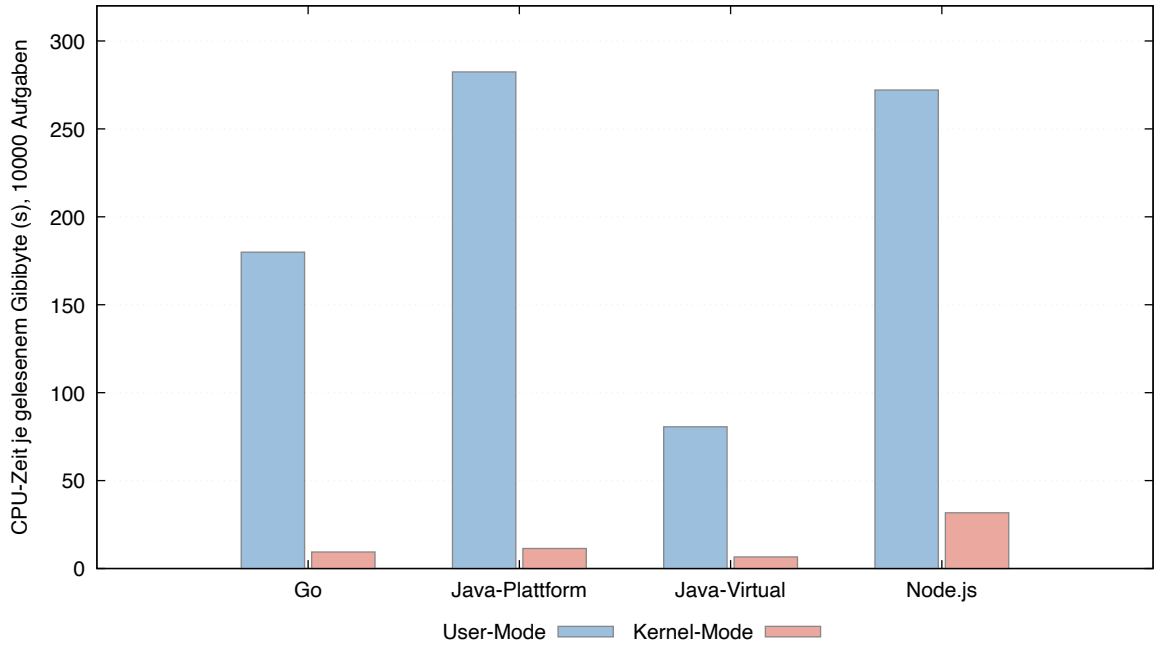

Abbildung 4.16: Gemischte Last, CPU-Zeit je gelesenem Gibibyte bei 10000 Aufgaben, aufgeteilt in User- und Kernel-Mode

Wie in den beiden vorherigen Abschnitten werden auch hier nicht alle erfassten Kennzahlen in Tabellen oder Abbildungen aufgenommen. Ein Teil beschreibt wie bisher die feste Konfiguration und ist bereits im Messplan enthalten, etwa `bufbytes`, `iters` und `cpusteps`. Die Kontrollwerte blieben über alle Läufe im erwarteten Bereich, so lag der ausgelagerte Speicher

vmswap_kb durchgehend bei null, die gelesene Datenmenge read_bytes_delta je Lauf bei rund 7,25 Gibibyte und der auf den Datenträger durchschlagende Anteil der gelesenen Bytes device_ratio_pct bei rund 95 Prozent. Wie in den vorherigen Abschnitten beschrieben liegen bestimmte Größen aus mehreren Quellen vor, wovon jeweils nur eine dargestellt wird und die block_rq_issue-Werte lagen mit rund 1,89 Millionen über alle Modelle gleichauf.

4.6 Ergebnisse der netzwerkbasierten Last

Dieser Abschnitt stellt die Messergebnisse der netzwerkbasierten Last dar. Der Test läuft gemäß dem in Abschnitt 4.2 beschriebenen Aufbau, bei dem der Lastgenerator oha von einer zweiten Maschine aus Anfragen an den Server richtet. Der Server beantwortet jede Anfrage nach einer festen Wartezeit von zehn Millisekunden mit 128 Byte und lief für Go, die Plattform-Threads und die Virtual Threads mit 16 Worker-Threads, für Node.js mit einem Thread. Der Aufbau folgt dem der lokalen Szenarien insofern, als zuerst die vier Modelle einzeln aufgeführt werden (Abschnitt 4.6.1) und anschließend der Vergleich erfolgt (Abschnitt 4.6.2). Der Messplan besteht aus zwei Gruppen. Die erste Gruppe, die Sättigungssuche, erhöht die Zahl gleichzeitiger Verbindungen von 100 auf 10000 ohne Ratenbegrenzung. Die zweite Gruppe, die Sättigungsstufen, hält 10000 Verbindungen und regelt die Anfragerate auf 50, 80, 100 und 110 Prozent des zuvor gefundenen Sättigungswerts. Gegenüber den lokalen Szenarien treten neue Kennzahlen auf, insbesondere die clientseitig gemessene Antwortzeitverteilung von oha, die Erfolgsquote und die Verbindungsfehler sowie systemweite TCP-Kennzahlen wie die Zahl gleichzeitig bestehender Verbindungen, die Überläufe und Verwerfungen der Accept-Queue, die TCP-Retransmits und die Timeouts. Als Durchsatz wird die clientseitig über das 60-Sekunden-Fenster gemessene Rate oha_req_s verwendet. Die Erfolgsquote lag außer bei Node.js unter Last durchgehend bei 100 Prozent.

4.6.1 Ergebnisse der einzelnen Modelle

Für jedes Modell wird im Folgenden die Sättigungssuche über die Verbindungszahl mit dem Durchsatz, dem Median und dem 99. Perzentil der Antwortzeit, der Zahl gleichzeitig bestehender Verbindungen und den Verbindungsfehlern aufgeführt. Ergänzend wird im Text das Verhalten unter geregelter Anfragerate genannt.

Go steigert den Durchsatz mit der Verbindungszahl bis auf rund 135800 Anfragen je Sekunde bei 2500 Verbindungen und liegt bei 10000 Verbindungen bei rund 128500. Das 99. Perzentil der Antwortzeit steigt dabei von 12,95 auf 513,02 Millisekunden. Bis 10000 Verbindungen werden alle Verbindungen angenommen, es bestehen 10002 gleichzeitige Verbindungen und es treten keine Verbindungsfehler auf. Unter geregelter Rate liefert Go bis zur Hochlast die volle Zielrate und erreicht am Kipppunkt sowie in der Überlast rund 127500 Anfragen je Sekunde. Tabelle 4.24 zeigt die Sättigungssuche im Einzelnen.

Tabelle 4.24: Netzwerkbasierte Last, Sättigungssuche für Go (Goroutinen)

Verbindungen	Durchsatz (10^3 req/s)	p50 (ms)	p99 (ms)	bestehende Verb.	Fehler (conn)
100	8,9	11,15	12,95	103	0
500	40,7	11,94	17,07	501	0
2500	135,8	18,02	25,10	2502	0
10000	128,5	37,69	513,02	10002	0

Die Plattform-Threads von Java steigern den Durchsatz bis auf rund 102000 Anfragen je Sekunde ab 2500 Verbindungen und halten diesen Wert bis 10000 Verbindungen. Das 99. Perzentil steigt von 11,04 auf 929,29 Millisekunden, alle Verbindungen werden angenommen. Unter geregelter Rate erreichen sie in der Überlast rund 94300 Anfragen je Sekunde bei einer Zielrate von rund 112100. Tabelle 4.25 führt die Werte auf.

Tabelle 4.25: Netzwerkbasierte Last, Sättigungssuche für Java (Plattform-Threads)

Verbindungen	Durchsatz (10^3 req/s)	p50 (ms)	p99 (ms)	bestehende Verb.	Fehler (conn)
100	9,5	10,43	11,04	102	0
500	47,6	10,35	11,35	501	0
2500	101,9	20,10	246,70	2502	0
10000	102,0	24,52	929,29	10001	0

Die Virtual Threads erreichen mit rund 130000 Anfragen je Sekunde bei 2500 Verbindungen einen ähnlichen Höchstwert wie Go und liegen bei 10000 Verbindungen bei rund 116700. Das 99. Perzentil steigt von 11,47 auf 555,87 Millisekunden, alle Verbindungen werden angenommen. Unter geregelter Rate liefern sie bis zur Überlast rund 122600 Anfragen je Sekunde. Tabelle 4.26 zeigt die Werte.

Tabelle 4.26: Netzwerkbasierte Last, Sättigungssuche für Java (Virtual Threads)

Verbindungen	Durchsatz (10^3 req/s)	p50 (ms)	p99 (ms)	bestehende Verb.	Fehler (conn)
100	9,3	10,67	11,47	106	0

Fortsetzung auf nächster Seite

Tabelle 4.26: Netzwerkbasierte Last, Sättigungssuche für Java (Virtual Threads)

Verbindungen	Durchsatz (10^3 req/s)	p50 (ms)	p99 (ms)	bestehende Verb.	Fehler (conn)
500	47,5	10,45	11,56	506	0
2500	130,0	18,04	36,26	2503	0
10000	116,7	35,88	555,87	10004	0

Node.js erreicht seinen höchsten Durchsatz mit rund 33500 Anfragen je Sekunde bereits bei 500 Verbindungen und liegt bei höheren Verbindungszahlen zwischen 27400 und 30100. Ab 2500 Verbindungen treten Verbindungsfehler auf, 44 bei 2500 und 387 bei 10000 Verbindungen und bei 10000 Verbindungen werden nur 3340 der Verbindungen aufgebaut. Das 99. Perzentil der angenommenen Anfragen bleibt bei 10000 Verbindungen mit 139,65 Millisekunden vergleichsweise klein, während das 99,99. Perzentil auf 52190 Millisekunden steigt. Unter geregelter Rate treten in der Hochlast und in der Überlast erneut Verbindungsfehler auf. Tabelle 4.27 zeigt die Werte.

Tabelle 4.27: Netzwerkbasierte Last, Sättigungssuche für Node.js (Event-Loop)

Verbindungen	Durchsatz (10^3 req/s)	p50 (ms)	p99 (ms)	bestehende Verb.	Fehler (conn)
100	8,8	11,29	13,75	102	0
500	33,5	14,57	20,30	501	0
2500	27,4	69,45	125,33	2501	44
10000	30,1	77,52	139,65	3340	387

4.6.2 Vergleich der Modelle

Die Gegenüberstellung fasst die vier Modelle zusammen und stützt sich auf den Durchsatz und die Antwortzeit über die Verbindungszahl, das Verhalten unter geregelter Anfragerate, die TCP-Kennzahlen unter Überlast und den auf die Worker-Threads bezogenen Durchsatz.

Abbildung 4.17 stellt den Durchsatz über die Verbindungszahl dar. Go und die Virtual Threads erreichen ihren Höchstwert von rund 135800 beziehungsweise 130000 Anfragen je Sekunde bei 2500 Verbindungen und gehen bei 10000 Verbindungen auf rund 128500 und 116700 zurück. Die Plattform-Threads steigen bis rund 102000 Anfragen je Sekunde und halten diesen Wert. Node.js erreicht mit rund 33500 Anfragen je Sekunde bereits bei 500 Verbindungen den höchsten Wert und liegt darüber zwischen 27400 und 30100.

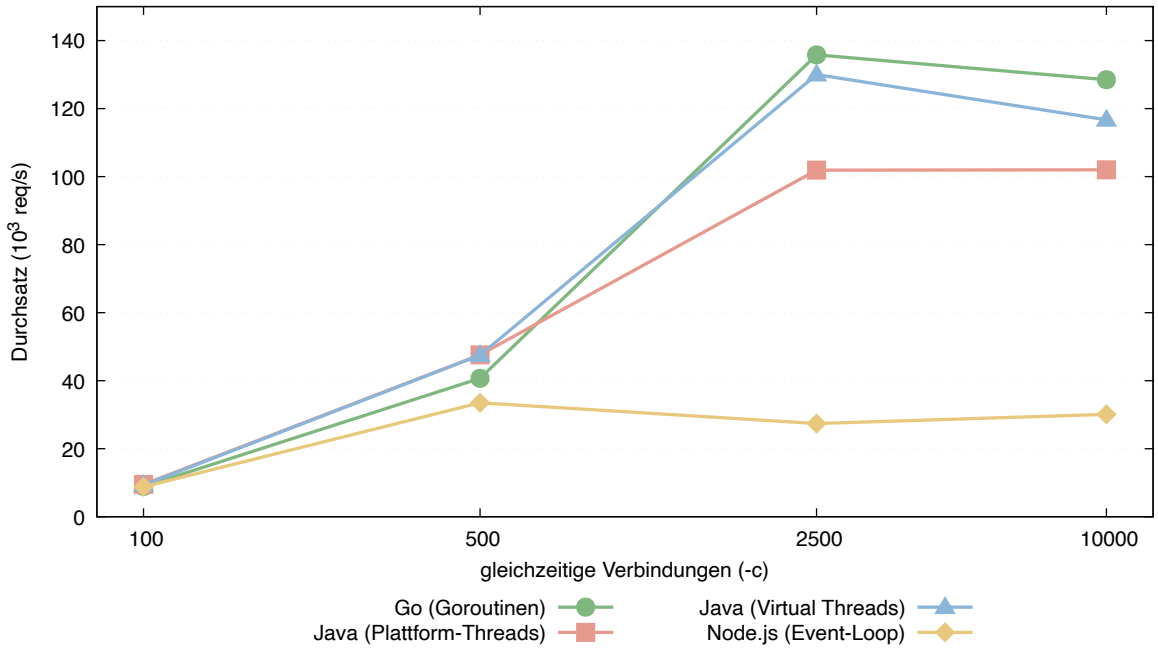

Abbildung 4.17: Netzwerkbasierte Last, Durchsatz über die Verbindungszahl

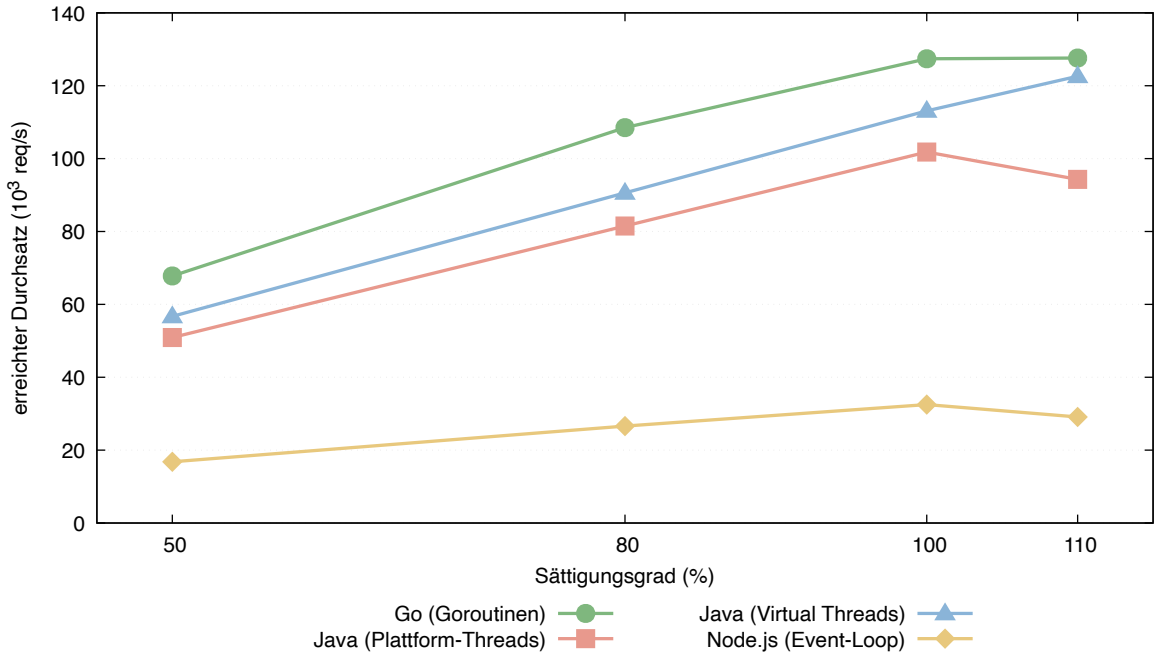

Abbildung 4.18: Netzwerkbasierte Last, erreichter Durchsatz über den Sättigungsgrad

Das Verhalten unter geregelter Anfragerate fasst Tabelle 4.28 zusammen, Abbildung 4.18 zeigt den erreichten Durchsatz über den Sättigungsgrad. Im Normalbetrieb bei 50 Prozent liefern alle vier Modelle die Zielrate bei einem 99. Perzentil zwischen 15,32 und 37,73 Millisekunden. Ab der Hochlast steigt das 99. Perzentil bei allen Modellen über 250 Millisekunden. Go, die Plattform-Threads und die Virtual Threads bleiben mit dem erreichten Durchsatz nahe an der Zielrate, bis diese am Kipppunkt die Sättigung erreicht. Bei Node.js treten in der Hochlast 488 und in der Überlast 590 Verbindungsfehler auf und der erreichte Durchsatz geht in der Überlast von 32500 am Kipppunkt auf 29100 Anfragen je Sekunde zurück.

Tabelle 4.28: Netzwerkbasierte Last, Verhalten unter geregelter Anfragerate

Phase	Sättigung (%)	Zielrate (10^3 req/s)	Durchsatz (10^3 req/s)	p99 (ms)	Fehler (conn)
Go (Goroutinen)					
Normalbetrieb	50	67,9	67,8	24,65	0
Hochlast	80	108,6	108,5	471,09	0
Kipppunkt	100	135,8	127,4	786,69	0
Überlast	110	149,3	127,6	927,58	0
Java (Plattform-Threads)					
Normalbetrieb	50	51,0	50,9	19,29	0
Hochlast	80	81,6	81,5	476,12	0
Kipppunkt	100	101,9	101,8	494,61	0
Überlast	110	112,1	94,3	952,04	0
Java (Virtual Threads)					
Normalbetrieb	50	56,7	56,7	15,32	0
Hochlast	80	90,7	90,6	492,68	0
Kipppunkt	100	113,4	113,1	934,25	0
Überlast	110	124,7	122,6	936,67	0
Node.js (Event-Loop)					
Normalbetrieb	50	16,8	16,8	37,73	0
Hochlast	80	26,8	26,6	258,67	488
Kipppunkt	100	33,5	32,5	426,04	0
Überlast	110	36,9	29,1	138,27	590

Die Antwortzeit zeigt Abbildung 4.19 anhand des 99. Perzentils. Bis 500 Verbindungen liegen alle vier Modelle zwischen 11,04 und 20,30 Millisekunden. Ab 2500 Verbindungen steigt das 99. Perzentil bei Go, den Plattform-Threads und den Virtual Threads deutlich an und erreicht bei 10000 Verbindungen 513,02, 929,29 und 555,87 Millisekunden. Bei Node.js bleibt es mit 125,33 und 139,65 Millisekunden niedriger, was mit den ab 2500 Verbindungen auftretenden Verbindungsfehlern und den nur teilweise aufgebauten Verbindungen einhergeht.

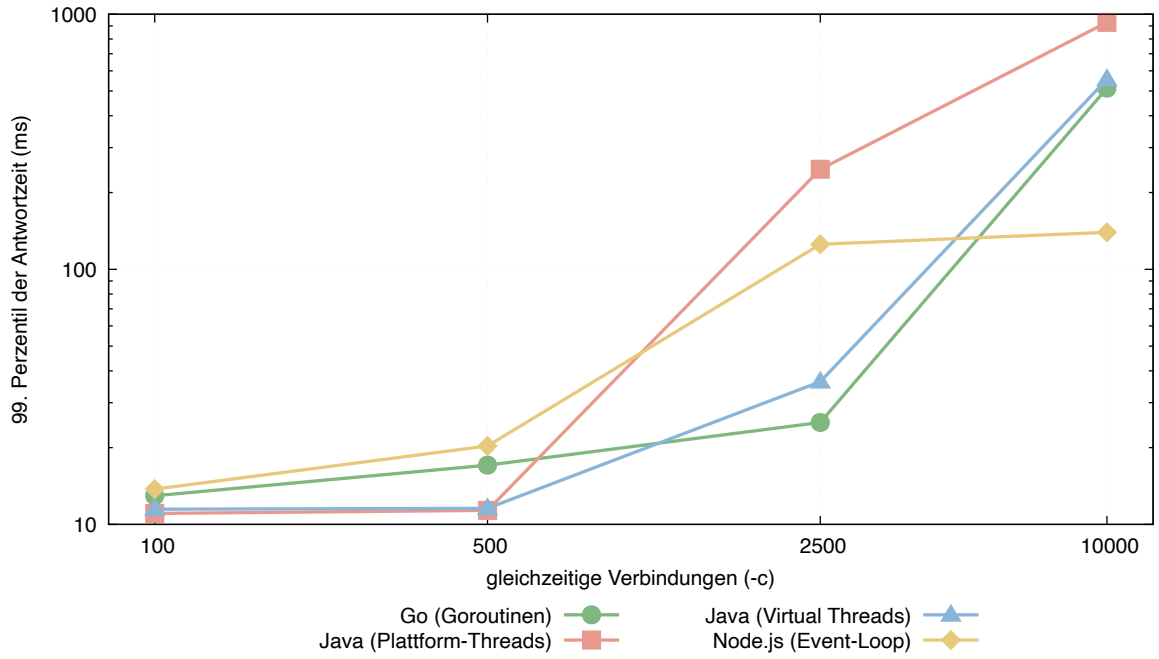

Abbildung 4.19: Netzwerkbasierte Last, 99. Perzentil der Antwortzeit über die Verbindungszahl

Die TCP-Ebene unter Überlast fasst Tabelle 4.29 bei 10000 Verbindungen der Sättigungssuche zusammen. Go, die Plattform-Threads und die Virtual Threads nehmen alle rund 10000 Verbindungen an und zeigen dabei hohe Zahlen an TCP-Retransmits von rund 906000 bis 1163000 sowie an Timeouts von rund 100000 bis 243000. Node.js baut dagegen nur 3340 Verbindungen auf und verwirft 89695 Verbindungen aus der Accept-Queue, bei zugleich niedrigen Retransmits von 4236. Ein ähnlicher Kontrast zeigt sich zwischen der Zahl der auf Anwendungsebene gleichzeitig in Bearbeitung befindlichen Anfragen und den bestehenden TCP-Verbindungen. Diese Zahl liegt bei Go mit 4333 und bei den Plattform-Threads mit 2090 deutlich unter den jeweils rund 10000 bestehenden Verbindungen.

Tabelle 4.29: Netzwerkbasierte Last, TCP-Kennzahlen bei 10000 Verbindungen

Modell	bestehende Verb.	Accept-Drops	Retransmits	Timeouts	Fehler (conn)
Go (Goroutinen)	10002	0	1163354	243434	0
Java (Plattform-Threads)	10001	14457	905546	100564	0
Java (Virtual Threads)	10004	5224	1058193	181738	0
Node.js (Event-Loop)	3340	89695	4236	2846	387

Da Node.js mit einem Thread und die übrigen Modelle mit 16 Worker-Threads liefen, mischt der absolute Durchsatz die Effizienz des Modells mit der Zahl der Worker-Threads. Um beides zu trennen, zeigt Tabelle 4.30 und Abbildung 4.20 den auf einen Worker-Thread bezogenen Durchsatz, also die Rate `oha_req_s` geteilt durch die Zahl der Worker-Threads. Node.js erreicht auf seinem einen Thread zwischen 8765 und 33548 Anfragen je Sekunde und liegt damit über alle Verbindungsstufen über den drei übrigen Modellen, deren Wert je Thread zwischen 555 und 8485 Anfragen je Sekunde liegt.

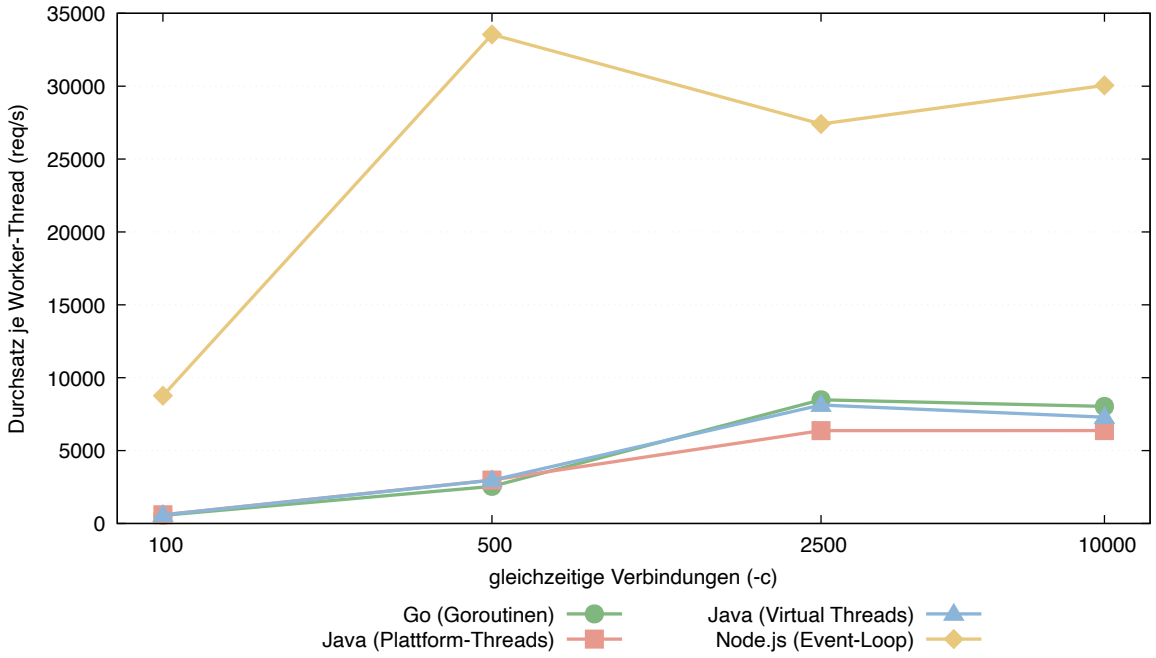

Abbildung 4.20: Netzwerkbasierte Last, Durchsatz je Worker-Thread über die Verbindungszahl

Tabelle 4.30: Netzwerkbasierte Last, Durchsatz je Worker-Thread in Anfragen je Sekunde

Verbindungen	Go	Java (Plattform)	Java (Virtual)	Node.js
100	555	597	582	8765
500	2543	2977	2966	33548
2500	8485	6369	8127	27406
10000	8032	6373	7292	30056

Wie in den vorherigen Abschnitten werden auch hier nicht alle erfassten Kennzahlen in Tabellen oder Abbildungen aufgenommen. Als Durchsatz wurde die clientseitige Rate `oha_req_s` verwendet, die serverseitige Rate `srv_req_s` entfällt daher, da ihre Bezugszeit kurze Leerlaufphasen vor und nach der Last enthält. Aus der Antwortzeitverteilung wurden der Median und das 99. Perzentil dargestellt, die übrigen Perzentile sowie der Mittelwert werden nur im Text erwähnt. Die Kennzahlen zum Verbindungsaufbau `oha_dns_dialup` und `oha_dns_lookup` bleiben nahe null, da der Client über eine IP-Adresse zugreift. Die Zähler `net_active_opens` und `net_tcp_tw` bleiben ebenfalls nahe null, da der Server keine Verbindungen aktiv öffnet. Der Zähler `oha_err_deadline` wird nicht als Fehler geführt, da er die beim Ablauf der 60-Sekunden-Deadline noch offenen Anfragen zählt und damit die zum Messende in Bearbeitung befindlichen Anfragen beschreibt.

5 Diskussion der Ergebnisse

Die Messergebnisse der vier Szenarien wurden in den Abschnitten 4.3 bis 4.6 dargestellt. Dieses Kapitel ordnet die Werte nun ein und leitet aus ihnen Erklärungen ab. Dafür werden die vier Lastszenarien zunächst einzeln und der Speicherbedarf übergeordnet betrachtet. Abschnitt 5.6 stellt die vier Modelle über die Szenarien hinweg gegenüber. Abschnitt 5.7 verbindet die Beobachtungen mit den theoretischen Grundlagen aus Kapitel 2, Abschnitt 5.8 benennt die Grenzen der Messung und Abschnitt 5.9 beantwortet die Forschungsfragen und Hypothesen.

5.1 Rechengebundene Last

Das auffälligste Messergebnis der rechengebundenen Last ist der große Leistungsunterschied zwischen den untersuchten Nebenläufigkeitsmodellen. Dabei setzen sich Go und Virtual Threads, also Modelle, die ihre Ausführungseinheiten auf eine feste Anzahl von Betriebssystem-Threads binden, deutlich von den Plattform-Threads und insbesondere von Node.js ab. Go und die Virtual Threads erreichen den höchsten Durchsatz (Abbildung 4.1), während die Plattform-Threads darunter liegen und Node.js über alle Stufen den untersten Verlauf bildet. Die Ursache für diese Reihenfolge liegt nicht in der pro Aufgabe geleisteten Rechenarbeit, sondern im Verwaltungsaufwand, den die Modelle bei hoher Nebenläufigkeit erzeugen. Am deutlichsten zeigt sich dies bei den Plattform-Threads. Da das Thread-per-Request-Modell wie in Abschnitt 2.5.1 beschrieben je Aufgabe einen Betriebssystem-Thread anlegt, laufen bei 10000 Aufgaben rund 10045 Threads auf 32 logischen Kernen. Diese Diskrepanz um mehr als das Dreihundertfache erzwingt ein ständiges Time-Slicing durch den Scheduler, das sich in den 994037 Kontextwechseln aus Tabelle 4.4 und in der mittleren Scheduling-Verzögerung von rund 19 Millisekunden aus Tabelle 4.8 niederschlägt.

Die eigentliche Wirkung dieses Time-Slicing wird dabei auf der Hardware-Ebene in Tabelle 4.7 sichtbar. Die Plattform-Threads weisen mit rund 18516 Millionen die mit weitem Abstand höchste Zahl an Misses im Last-Level-Cache auf, also mehr als das Fünffache der rund 3410 Millionen bei Go und mehr als das Sechsfache der rund 2814 Millionen bei den Virtual Threads. Ein ähnliches Bild ergibt sich beim Daten-TLB, dessen Misses mit rund 189 Millionen bei den Plattform-Threads etwa das Neunzehnfache der rund 9,9 Millionen bei den Virtual Threads betragen. Beide Werte sind Folge desselben Vorgangs, denn jeder Thread muss nach seiner Unterbrechung die von ihm zuvor genutzten Cache Lines und Adressübersetzungen neuladen. Grund ist, dass die zwischenzeitlich gelaufenen Threads diese Werte verdrängt haben. Da ein Zugriff auf den Hauptspeicher dort mehrere hundert Taktzyklen kostet, während ein Zugriff auf den L1-Cache mit wenigen Zyklen auskommt, schlägt sich diese Verdrängung direkt in der Zahl der Instruktionen je Zyklus nieder, die bei den Plattform-Threads mit 0,19 den niedrigsten Wert aller Modelle erreicht. Ein großer Teil der Zyklen entfällt somit nicht auf Rechenarbeit, sondern auf das Warten der Recheneinheiten auf Cache und TLB. Die hohe Zahl an Misses ist

dabei nicht die Ursache des Leistungsabfalls, sondern lediglich Symptom der Kontextwechsel, welche die geringe Recheneffizienz verursachen.

Go und die Virtual Threads umgehen diesen Aufwand durch das in Abschnitt 2.5 beschriebene M:N-Modell, das viele leichte Ausführungseinheiten auf eine kleine, an die Kernzahl gebundene Menge von Processoren/Carriern abbildet. Beide halten die Zahl der Betriebssystem-Threads über alle Stufen nahezu konstant, Go bei 33 bis 34 und die Virtual Threads bei 75 bis 82, sodass die wenigen langlebigen Carrier ihre Cache Lines über längere Zeit behalten. Entsprechend niedriger fallen die Misses im Last-Level-Cache und im Daten-TLB aus und die Instruktionen je Zyklus liegen mit 0,46 bei Go und 0,31 bei den Virtual Threads deutlich über dem Wert 0,19 der Plattform-Threads. Dass Go dabei vor den Virtual Threads liegt, könnte sich auf den geringeren Aufwand der Go-Runtime zurückführen lassen, dessen Scheduler die Goroutinen mit weniger Verwaltungsschritten verwaltet als der ForkJoinPool die Virtual Threads, was auch die geringfügig niedrigere Zahl an Cache-Misses bei Go erklären würde.

Node.js nimmt eine Sonderstellung ein, an der sich der Gegensatz zwischen der Verarbeitung einer einzelnen Aufgabe und der fehlenden Parallelität besonders deutlich zeigt. Der niedrige Durchsatz ist die direkte Folge der Parallelität von eins, denn ein einzelner Thread kann die rechengebundenen Aufgaben nur nacheinander abarbeiten. Die übrigen Kennzahlen zeigen jedoch, dass nicht die Verarbeitung der einzelnen Aufgabe die Schwäche von Node.js ist. So weist Node.js in Tabelle 4.9 mit 159,5 bis 172,3 Millisekunden die niedrigste je Aufgabe aufgewendete CPU-Zeit auf und verursacht in Tabelle 4.7 mit rund 222 Millionen die mit Abstand niedrigste Zahl an Misses im Last-Level-Cache, also nur etwa ein Fünfzehntel des Wertes von Go und rund 1,2 Prozent des Wertes der Plattform-Threads.

Dieser Befund lässt sich jedoch nicht als eine grundsätzlich überlegene Ausführung deuten, sondern ist vor allem eine Folge der seriellen Abarbeitung. Da Node.js die Aufgaben nacheinander auf einem Kern ausführt, ist zu jedem Zeitpunkt nur ein einziger Puffer von einem Mebibyte aktiv, der genau in den ein Mebibyte großen L2-Cache eines Kerns passt, wie er in Abschnitt 3.1 beschrieben ist. Node.js profitiert damit von einer nahezu perfekten Cache-Lokalität, während bei den parallel arbeitenden Modellen mehrere solcher Puffer gleichzeitig um denselben L2-Cache und den geteilten L3-Cache konkurrieren, was ihre höhere Zahl an Cache-Misses erklärt. Zu dieser Lokalität kommt, dass bei nur einem Thread weder Kontextwechsel zwischen konkurrierenden Aufgaben noch Kohärenzkosten zwischen mehreren Kernen entstehen.

Auch die hohe Zahl an Instruktionen je Zyklus reicht nicht als Beleg einer überlegenen Ausführung aus, denn diese Kennzahl ist über unterschiedliche Instruktionstypen hinweg kein gutes Maß für Effizienz. Node.js führt mit rund 4483 Milliarden die meisten Instruktionen aus, also etwa doppelt so viele wie Go dies könnte den von der JIT-Kompilierung erzeugten Instruktionen geschuldet sein, dies wurde allerdings nicht gemessen und bildet somit nur eine Vermutung. Aussagekräftiger ist die Zahl der Taktzyklen und diese liegt bei Node.js mit rund 4811 Milliarden nahezu gleichauf mit den rund 4848 Milliarden bei Go. Node.js benötigt also trotz der doppelten Instruktionszahl etwa gleich viele Zyklen wie Go. V8 ist zudem kein reiner Interpreter, sondern

ein JIT-Compiler mit vorgeschalteter Interpreterstufe, sodass die höhere Instruktionszahl nicht einfach der Tatsache geschuldet sein kann, das Node.js eine Interpretersprache ist.

Für die rechengebundene Last lässt sich damit festhalten, dass nicht die Ausführung von Node.js langsam ist, sondern allein die auf einen Kern beschränkte Parallelität den Durchsatz begrenzt. Der scheinbare Vorsprung bei der CPU-Zeit je Aufgabe beruht dabei wesentlich auf der perfekten Cache-Lokalität der seriellen Abarbeitung und nicht auf einer grundsätzlich besseren Ausführung.

5.2 Dateibasierte Last

Bei der dateibasierten Last verschiebt sich der begrenzende Faktor von der CPU auf den Massenspeicher. Der auf den Datenträger durchschlagende Anteil der gelesenen Bytes (`device_ratio`) liegt über alle Modelle und Läufe bei rund 95 Prozent. Dieser Wert ist keine Angabe zur Auslastung des Block-Devices, sondern eine Kontrolle gegen eine Bedienung aus dem Page-Cache und belegt allein, dass die Zugriffe kalt waren und nahezu vollständig den Datenträger erreichten. Dass das Block-Device nicht in jedem Fall der begrenzende Faktor ist, zeigt schon der Vergleich der Modelle bei identischer Datenmenge, denn Go liest mit rund 654 MiB/s ein Vielfaches der rund 45 MiB/s von Node.js. Der Durchsatz hängt daher nicht allein von der Geschwindigkeit des Datenträgers ab, sondern davon, wie viele Lesezugriffe ein Modell gleichzeitig verarbeiten kann. Grundsätzlich liegen die Laufzeiten näher beieinander als bei der rechengebundenen Last. Innerhalb dieser Beobachtung zeigen sich jedoch deutliche Unterschiede zwischen erreichtem Durchsatz und dafür aufgewendeter CPU-Zeit, was die Modelle klar voneinander abgrenzt.

Go und die Plattform-Threads erreichen mit rund 644 bis 675 MiB/s den höchsten Durchsatz (Abbildung 4.6). Beide nutzen für jede Aufgabe einen eigenen blockierenden Lesezugriff, sei es über eine Goroutine oder über einen Plattform-Thread, sodass zu jedem Zeitpunkt viele Anfragen gleichzeitig beim Block-Device ausstehen. Diese hohe Zahl gleichzeitiger Zugriffe hält das Device beschäftigt, erzeugt aber viele Systemaufrufe, was sich im Anteil der im Kernel-Mode verbrachten Zeit von rund 92 Prozent und in der hohen CPU-Zeit je gelesenem Gibibyte aus Tabelle 4.16 niederschlägt, die ab 500 Aufgaben bei rund 45 Sekunden liegt.

Die Virtual Threads verhalten sich hier grundsätzlich anders. Sie erreichen mit 471,6 bis 541,8 MiB/s einen geringeren Durchsatz, benötigen dafür aber mit 16,9 bis 18,0 Sekunden je Gibibyte die mit Abstand niedrigste CPU-Zeit. Der zugleich über alle Stufen hohe Anteil der ungenutzten Maschinenkapazität von 72,4 bis 76,0 Prozent zeigt, dass die Kerne dabei überwiegend auf den Abschluss der Lesezugriffe warten und weniger Anfragen gleichzeitig ausstehen als bei Go und den Plattform-Threads. Der Grund dafür liegt im Verhalten der Carrier-Threads bei blockierenden Dateizugriffen. Wie in Abschnitt 2.5.4 beschrieben, fällt ein blockierender Dateizugriff unter das dort dargestellte Carrier-Capture, das die JVM durch vorübergehend zusätzliche Carrier-Threads bis zu einer Obergrenze von standardmäßig 256 kompensiert. Dass diese Kompensation greift, zeigt die Zahl der Betriebssystem-Threads, die bei den Virtual

Threads über die Stufen zwischen 53 und 86 liegt. Dieser Wert übersteigt die Grundparallelität des ForkJoinPool von 32 Carriern zuzüglich der JVM-Basisthreads erkennbar, bleibt jedoch weit unterhalb der zulässigen Obergrenze von 256. Die Zahl gleichzeitig ausstehender Lesezugriffe wächst somit durch die Kompensation über die Grundparallelität hinaus, bleibt aber weit unter dem Wert des Modells mit einem dedizierten Betriebssystem-Thread je Aufgabe. Warum sich die Kompensation auf dieser Stufe einpendelt und nicht bis zur Obergrenze ausbaut, lässt sich aus den erhobenen Kennzahlen nicht eindeutig belegen und kann nur als Deutung benannt werden. Nahe liegt, dass der ForkJoinPool zusätzliche Carrier nur reaktiv und schrittweise erzeugt und sich für diese Last ein Gleichgewicht deutlich unterhalb der Obergrenze einstellt. Der reduzierte Aufwand hat hier also zugleich einen geringeren Durchsatz zur Folge. Da aber auch schon die rechnengebundene Last eine ähnliche Anzahl an Carrier Threads aufweist, könnte der Anstieg ebenso auf den JIT-Compiler zurückzuführen sein.

Dass die geringe Zahl gleichzeitiger Zugriffe der Virtual Threads mit einem sparsamen Umgang mit der Speicherhierarchie einhergeht, zeigt auch der für dieses Szenario erfasste Last-Level-Cache aus Tabelle 4.14. Die Virtual Threads verursachen bei 10000 Aufgaben mit rund 133 Millionen die niedrigste Zahl an Misses, während Go mit rund 286 Millionen und die Plattform-Threads mit rund 314 Millionen darüber liegen. Der Daten-TLB wurde für die dateibasierte Last nicht aufgezeichnet, sodass sich die Betrachtung hier auf den Last-Level-Cache beschränkt. Insgesamt sind die Unterschiede zwischen den Modellen bei den Cache-Misses jedoch weit kleiner als bei der rechengebundenen Last, was zum verschobenen Engpass passt, denn wenn die Läufe auf das Block-Device warten, tritt die Belastung der Caches als Ursache weniger in Erscheinung.

Der Verlauf des Anteils der ungenutzten Maschinenkapazität aus Abbildung 4.8 lässt sich vor dem Hintergrund des Zugriffsmusters deuten. Wie in Abschnitt 3.4.2 beschrieben, wählt der dateibasierte Test für jeden Zugriff neu eine zufällige Datei und einen zufälligen Versatz, sodass die Zugriffe auf jeder Aufgabenstufe über den gesamten Datensatz gestreut sind und das Readahead des Kernels keine langen zusammenhängenden Bereiche zu wenigen großen Block-Requests zusammenfassen kann. Der Rückgang des Anteils bei Go und den Plattform-Threads von rund 69 Prozent bei 100 Aufgaben auf rund 11 bis 14 Prozent ab 500 Aufgaben ist daher nicht auf ein solches Zusammenfassen zurückzuführen. Näher liegt die Vermutung, dass bei nur 100 Aufgaben wenige Zugriffe gleichzeitig ausstehen, sodass das Block-Device nicht ausgelastet ist und die Kerne überwiegend auf den Abschluss der Zugriffe warten. Mit steigender Aufgabenzahl stehen mehr Zugriffe gleichzeitig aus, der Verwaltungsaufwand im Kernel für das Absetzen und Abschließen der vielen kleinen Requests wächst und füllt die zuvor wartenden Kerne. Diese Deutung deckt sich mit der CPU-Zeit je Gibibyte aus Tabelle 4.16, die bei Go und den Plattform-Threads von rund 16 Sekunden bei 100 Aufgaben auf rund 45 Sekunden ab 500 Aufgaben steigt. Bei den Virtual Threads bleibt der Anteil über alle Stufen zwischen 72,4 und 76,0 Prozent, was erneut zur begrenzten Zahl gleichzeitiger Zugriffe passt.

Node.js liegt bei dieser Last mit 41,2 bis 45,7 MiB/s deutlich unter allen anderen Modellen und die Gründe dafür lassen sich hier weniger eindeutig benennen als bei der rechengebundenen

Last. Für die Interpretation, dass nicht die Verarbeitung der einzelnen Zugriffe, sondern die geringe Parallelität der begrenzende Faktor ist, spricht zunächst, dass Node.js in Tabelle 4.14 mit 0,99 die höchste Zahl an Instruktionen je Zyklus erreicht. Anders als bei der rechengebundenen Last endet die Übereinstimmung der Kennzahlen an dieser Stelle jedoch. Node.js weist mit rund 40,9 bis 46,2 Sekunden je gelesenem Gibibyte einen der höchsten Werte für die aufgewendete CPU-Zeit auf und verursacht mit rund 358 Millionen zugleich die höchste Zahl an Misses im Last-Level-Cache. Beide Werte zeigen, dass der Weg eines Lesezugriffs durch die JavaScript-Ebene, den libuv-Threadpool und die zugehörigen Callbacks mehr Arbeit je Zugriff erfordert als der direkte Systemaufruf bei Go. Der niedrige Durchsatz von Node.js bei der dateibasierten Last geht somit nur zum Teil auf die begrenzte Parallelität des Threadpools zurück, während der höhere Aufwand des ereignisgetriebenen Zugriffspfads als zweite Ursache hinzukommt. Das bei der rechengebundenen Last deutliche Ergebnis, dass allein die Kernzahl begrenzt, gilt für die dateibasierte Last also nur in abgeschwächter Form.

5.3 Gemischte Last

Die gemischte Last kombiniert rechen- und leseintensive Anteile, was in einem bestimmten Aspekt zu einer deutlichen Verschiebung der ursprünglichen Rangfolge führt. Die Virtual Threads erreichen hier über alle Stufen den höchsten Durchsatz und die kürzesten Laufzeiten (Abbildung 4.11 und Tabelle 4.19) und zugleich mit 79,7 bis 87,2 Sekunden je Gibibyte die niedrigste CPU-Zeit (Tabelle 4.23). Sie sind damit im gemischten Szenario sowohl das schnellste als auch das sparsamste Modell, obwohl sie bei der reinen dateibasierten Last hinter Go lagen.

Da jede Iteration zuerst rechnet und dann liest, ist der Ablauf bei der gemischten Last nicht mehr allein durch den Durchsatz des Block-Devices bestimmt, sondern durch die Fähigkeit eines Modells, die Rechenarbeit der einen Aufgabe mit der Wartezeit einer anderen zu überlappen. Ein auf einen Lesezugriff wartende Virtual Thread gibt seinen Carrier frei, der daraufhin einen rechenbereiten Virtual Thread übernimmt, sodass die Kerne beschäftigt bleiben, ohne dass die für die Plattform-Threads typische Flut an Kontextwechseln entsteht. Die gute Cache-Nutzung der Virtual Threads bleibt dabei erhalten, denn sie verursachen in Tabelle 4.21 mit rund 1611 Millionen die niedrigste Zahl an Misses im Last-Level-Cache und mit rund 43 Millionen die niedrigste Zahl an dTLB-Misses aller Modelle. Dass Go, das bei der reinen rechengebundenen und der reinen dateibasierten Last jeweils vor den Virtual Threads lag, hier mit rund der doppelten Laufzeit zurückfällt, ist das auffälligste Ergebnis dieses Szenarios. Eine plausible Erklärung könnte am ineffizienteren Management des Go-Schedulers liegen, welcher die logischen Processors kontinuierlich von den blockierten Threads lösen und an andere Threads übergeben muss. Bei hoher Nebenläufigkeit erzeugt dieses ständige Umkoppeln einen erheblichen internen Verwaltungsaufwand, der die parallele Verarbeitung der rechengebundenen Last massiv drosselt.

Ein entsprechendes Bild zeigen die Plattform-Threads. Sie halten den Anteil der ungenutzten Maschinenkapazität mit 0,4 bis 4,4 Prozent sehr niedrig (Abbildung 4.13), ihre Kerne sind also fast durchgehend beschäftigt, doch die hohe CPU-Zeit je Gibibyte von bis zu 311,6 Sekunden sowie die mit rund 11031 Millionen zweithöchsten Cache-Misses und die rund 258 Millionen dTLB-Misses zeigen, dass ein großer Teil dieser Beschäftigung erneut auf den Verwaltungsaufwand der vielen Threads entfällt und nicht auf nutzbare Arbeit.

Node.js zeigt bei der gemischten Last das ungünstigste Ergebnis aller Szenarien und benötigt mit 1784 bis 1920 Sekunden ein Vielfaches der Laufzeit der übrigen Modelle. Auch hier lohnt die Unterscheidung zwischen Ausführungseffizienz und Parallelität, denn beide Ursachen lassen sich in den Werten getrennt nachweisen. Für die begrenzende Wirkung der Parallelität spricht am deutlichsten der Anteil der ungenutzten Maschinenkapazität von rund 96 Prozent, dieser Wert zeigt, dass Node.js fast die gesamte Kapazität der 32 Kerne ungenutzt lässt, während die rechnenden Anteile auf dem einen Haupt-Thread des Event-Loops nacheinander abgearbeitet werden. Da die Rechenschritte im Gegensatz zu den Dateizugriffen nicht an den libuv-Threadpool ausgelagert werden, laufen sie seriell und die übrigen Kerne stehen still. Node.js führt zwar wie schon in den anderen Szenarien die Zahl der Instruktionen je Zyklus mit 0,39 auch bei der gemischten Last an, doch belegt diese Kennzahl für sich genommen keine überlegene Ausführung, wie in Abschnitt 5.1 dargelegt. Zugleich zeigen die Werte, dass Node.js den rechnenden Anteil mit hohem Aufwand bewältigt, denn es führt mit rund 2326 Milliarden die meisten Instruktionen aller Modelle aus, also etwa das Fünfeinhalbfache der rund 421 Milliarden bei Go und verursacht mit rund 17132 Millionen Cache-Misses und rund 1058 Millionen dTLB-Misses die mit weitem Abstand höchsten Werte der Speicherhierarchie. Die gemischte Last belastet Node.js also auf zwei Ebenen gleichzeitig, denn der rechnende Anteil ist sowohl seriell auf einen Kern beschränkt als auch instruktions- und cacheintensiver als bei den übrigen Modellen. Der Befund, dass allein die Kernzahl begrenzt, trifft damit für die gemischte Last am wenigsten von allen Szenarien zu, auch wenn der hohe Anteil ungenutzter Maschinenkapazität zeigt, dass ungenutzte Parallelität einen wesentlichen Anteil daran hat.

5.4 Netzwerkbasierte Last

Die netzwerkbasierte Last unterscheidet sich von den lokalen Szenarien dadurch, dass die feste Wartezeit von zehn Millisekunden je Anfrage die Last stark an I/O bindet und der Durchsatz eng an die Zahl der gleichzeitig in Bearbeitung befindlichen Anfragen gebunden ist. Bei einer festen Bearbeitungszeit je Anfrage steigt der erreichbare Durchsatz näherungsweise proportional mit der Zahl der gleichzeitig offenen Bearbeitungen, sodass ein Modell umso mehr Anfragen je Sekunde beantwortet, je mehr Anfragen es gleichzeitig in Bearbeitung halten kann.

Diesen Zusammenhang zeigt die Sättigungssuche aus Abbildung 4.17. Go und die Virtual Threads halten bei 2500 Verbindungen die meisten Anfragen gleichzeitig in Bearbeitung und erreichen mit rund 135800 beziehungsweise 130000 Anfragen je Sekunde ihren Höchstwert. Die Plattform-Threads bleiben mit rund 102000 Anfragen je Sekunde darunter, was sich mit

der niedrigeren Nebenläufigkeit deckt, denn ihr höchster Wert liegt bei 2090 gegenüber 4333 bei Go. Node.js erreicht seinen Höchstwert von rund 33500 Anfragen je Sekunde bereits bei 500 Verbindungen und fällt danach ab. Dieser frühe Sättigungspunkt markiert die Grenze des einen Threads, denn die Event-Loop kann die zur Annahme, zur Verarbeitung und zur Beantwortung einer Anfrage nötige Arbeit nur seriell leisten und diese Arbeit begrenzt die Zahl der Beantwortungen je Sekunde unabhängig davon, wie viele Verbindungen offen sind.

Für die netzwerkbasierte Last lässt sich im Bezug auf Node.js wieder sagen, dass nicht die Ausführung, sondern die Zahl der Threads begrenzend wirkt. Der Durchsatz je Worker-Thread aus Abbildung 4.20 zeigt, dass Node.js auf seinem einen Thread zwischen 8765 und 33548 Anfragen je Sekunde beantwortet und damit deutlich über den rund 555 bis 8485 Anfragen je Sekunde und Thread der übrigen Modelle liegt. Auf einen einzelnen Thread bezogen ist das ereignisgetriebene Modell für diese an Ein- und Ausgabe gebundene Last also das mit Abstand wirksamste, wie es die Betrachtung des in Abschnitt 2.1 beschriebenen C10k-Problems erwarten lässt. Die Bearbeitung einer Anfrage besteht im Wesentlichen aus dem Warten auf die feste Verzögerung und dem Senden einer kleinen Antwort. Node.js kann diesen Vorgang lediglich auf einen Thread ausführen und stößt so auf dem Gesamtdurchsatz früher an eine Begrenzung.

Ebenso interessant ist das Verhalten unter Überlast, das Tabelle 4.29 zusammenfasst. Go, die Plattform-Threads und die Virtual Threads nehmen bei 10000 Verbindungen alle rund 10000 Verbindungen an und zeigen dabei hohe Zahlen an TCP-Retransmits und Timeouts. Sie nehmen die Last also vollständig an und geben die Überlastung in Form steigender Antwortzeiten weiter, was sich im starken Anstieg des 99. Perzentils auf über 500 Millisekunden aus Abbildung 4.19 zeigt. Node.js verhält sich grundlegend anders, es baut nur 3340 der 10000 Verbindungen auf und verwirft 89695 Verbindungen aus der Accept-Queue, während seine Zahl an Retransmits mit 4236 gering bleibt. Node.js weist die Überlast also bereits am Verbindungsaufbau ab, statt sie anzunehmen und zu verzögern. Genau dieses unterschiedliche Überlastverhalten erklärt das auf den ersten Blick widersprüchliche 99. Perzentil von Node.js, das mit 139,65 Millisekunden bei 10000 Verbindungen niedriger ausfällt als bei den übrigen Modellen. Der Wert bezieht sich nur auf die angenommenen Anfragen, die schnell genug beantwortet werden, während die abgewiesenen Verbindungen nicht in die Antwortzeit eingehen. Dass die wenigen dennoch angenommenen, aber verzögerten Anfragen eine sehr langen Peak bilden, zeigt das 99,99. Perzentil von 52190 Millisekunden. Unter geregelter Anfragerate (Tabelle 4.28) zeigt sich ein ähnlicher Befund, denn Node.js zeigt in der Hochlast und in der Überlast erneut Verbindungsfehler, während die übrigen Modelle die Last bis zum jeweiligen Sättigungswert ohne Verbindungsfehler tragen.

5.5 Speicherbedarf pro Ausführungseinheit

Der Speicherbedarf ist die einzige der zentralen Kennzahlen, die sich nicht innerhalb eines einzelnen Szenarios, sondern erst im Vergleich der Szenarien sinnvoll interpretieren lässt,

weshalb sie hier gesondert betrachtet wird. Erfasst wurde der Speicherverbrauch über den maximal belegten physischen Speicher (VmHWM) in den drei lokalen Szenarien. Bei der rechengebundenen und der gemischten Last hält jede Aufgabe einen eigenen Puffer von einem Mebibyte vor, sodass bei 10000 Aufgaben allein die Puffer rund zehn Gibibyte belegen und den Gesamtwert prägen. In diesen beiden Szenarien liegen daher alle Modelle bei 10000 Aufgaben nahe beieinander, die Plattform-Threads belegen mit 14957 MiB bei der rechengebundenen und 15513 MiB bei der gemischten Last jeweils den höchsten Wert, gefolgt von den Virtual Threads, während Go und Node.js mit rund 10000 bis 10200 MiB am niedrigsten liegen.

Deutlich klarer treten die Effekte bei der dateibasierten Last hervor, da diese ohne große Puffer je Aufgabe auskommt und der belegte Speicher damit nahezu vollständig auf das Laufzeitsystem und die Ausführungseinheiten entfällt. Die Werte aus den Tabellen 4.10 bis 4.13 zeigen bei 10000 Aufgaben eine Spanne von 94 MiB bei Go über 249 MiB bei den Virtual Threads und 325 MiB bei Node.js bis zu 1598 MiB bei den Plattform-Threads. Die Plattform-Threads belegen damit rund das Siebzehnfache des Speichers von Go. Ebenso interessant ist das Wachstum über die Aufgabenstufen, denn der Speicherbedarf der Plattform-Threads steigt von 181 MiB bei 100 Aufgaben auf 1598 MiB bei 10000 Aufgaben und damit annähernd um den Faktor neun, während er bei Go nur um den Faktor fünf und bei den Virtual Threads lediglich um den Faktor von knapp zwei zunimmt.

Diese Beobachtung deckt sich mit den in Abschnitt 2.2.2 beschriebenen Eigenschaften der Modelle. Ein Plattform-Thread reserviert einen festen Stack, der im Fall der JVM in der Größenordnung eines Megabyte liegt [1] und da diese Reservierung im Verlauf der Ausführung zunehmend tatsächlich belegt wird, wächst der Speicherbedarf annähernd proportional zur Zahl der Threads. Die Virtual Threads und die Goroutinen halten ihre Stacks dagegen auf dem verwalteten Speicher des Laufzeitsystems und vergrößern sie nur nach Bedarf, sodass der Speicherbedarf je Ausführungseinheit weit geringer ausfällt und langsamer wächst. Dass die Virtual Threads dabei über den Goroutinen liegen, ist dabei schlüssig, denn beide vermeiden zwar die feste Reservierung der Plattform-Threads, doch verwalten sie ihre Stacks mit unterschiedlichem Aufwand. Der Speicherbedarf bestätigt damit die zu erwartende Abstufung, nach der die Goroutinen am sparsamsten sind, die Virtual Threads darüber liegen und die Plattform-Threads den mit Abstand höchsten Bedarf je Ausführungseinheit aufweisen [10].

5.6 Gegenüberstellung der untersuchten Modelle

Zusammenfassend lässt sich sagen, kein Modell ist über alle Lastarten hinweg überlegen, vielmehr bestimmt die Art der Last, welches Modell besonders Vorteilshaft im Hinblick auf die systemnahen Laufzeiteigenschaften agiert. Die folgende Gegenüberstellung fasst für jedes Modell zusammen, unter welchen Bedingungen es welche Vorteile besitzt und wodurch seine Grenzen bestimmt sind.

Go zeigt das ausgewogenste Verhalten. Bei der rechengebundenen und der reinen dateibasierten Last erreicht es jeweils den höchsten oder nahezu höchsten Durchsatz bei geringem Verwaltungsaufwand und bei der netzwerkbasierten Last gehört es zu den beiden Modellen mit dem höchsten Sättigungsdurchsatz. Auch bei den Kennzahlen der Speicherhierarchie liegt es durchgehend im mittleren Bereich und beim Speicherbedarf ist es das sparsamste Modell. Seine Schwäche wird erst bei der gemischten Last sichtbar, wo es hinter die Virtual Threads zurückfällt.

Die Virtual Threads erweisen sich als das Modell mit dem größten Vorteil bei gemischten Lasten und mit der durchgehend besten Cache-Nutzung. Sie verbinden den geringen Verwaltungsaufwand des M:N-Modells mit der Fähigkeit, blockierende Wartezeiten sinnvoll zu überbrücken und erreichen dadurch bei der gemischten Last den höchsten Durchsatz bei zugleich niedrigster CPU-Zeit und niedrigsten Cache- und TLB-Misses. Der Speicherbedarf liegt zwar über dem der Goroutinen, aber weit unter dem der Plattform-Threads. Bei der reinen dateibasierten Last weisen die Virtual Threads zwar einen Nachteil beim Durchsatz auf, der jedoch mit einem sehr geringen Aufwand einhergeht. Bei der rechengebundenen und der netzwerkbasierten Last liegen sie nahe an Go.

Die Plattform-Threads sind das Modell, dessen Kosten am stärksten mit der Nebenläufigkeit steigen. Solange die Zahl der Aufgaben klein bleibt, liegen sie gleichauf mit den übrigen Modellen, etwa bei 100 Verbindungen der netzwerkbasierten Last. Mit wachsender Aufgabenzahl treten jedoch die Kosten des Thread-per-Request-Modells in den Vordergrund, also die vielen Kontextwechsel, die hohe Scheduling-Verzögerung und die in dieser Diskussion mehrfach belegte Belastung von Cache und TLB, die bei der rechengebundenen Last die höchsten Werte aller Modelle erreicht. Der höchste Speicherbedarf über alle Szenarien ist eine weitere unmittelbare Folge des einen Threads je Aufgabe.

Node.js zeigt über alle lokalen Szenarien hinweg die höchste Zahl der Instruktionen je Zyklus. Diese Kennzahl belegt für sich genommen keine überlegene Ausführung, wie in Abschnitt 5.1 erläutert, da sie über unterschiedliche Instruktionsmischungen hinweg kein Maß für Effizienz ist. Bei der rechengebundenen Last verursacht Node.js zudem die niedrigsten Cache-Misses und die niedrigste CPU-Zeit je Aufgabe, was jedoch wesentlich auf die perfekte Cache-Lokalität der seriellen Abarbeitung zurückgeht und nicht auf eine grundsätzlich schnellere Ausführung. Der geringe Durchsatz bei drei der vier Szenarien ist vielmehr eine Folge der geringen Parallelität, die bei der rechengebundenen und der netzwerkbasierten Last die alleinige Ursache bildet und sich bei der dateibasierten und der gemischten Last mit dem höheren Aufwand des ereignisgetriebenen Zugriffspfads kombiniert. Bei der an I/O gebundenen netzwerkbasierten Last ist Node.js je Thread das wirksamste Modell und hält die Antwortzeiten der angenommenen Anfragen auch unter hoher Last niedrig, was seinen ursprünglichen Einsatzzweck widerspiegelt. Der Vorteil des einen Threads, also der Wegfall von Kontextwechseln und Kohärenzkosten, ist somit zugleich seine Grenze, denn er lässt sich nicht auf mehrere Kerne ausdehnen.

5.7 Einordnung in den theoretischen Kontext

Die Messergebnisse lassen sich weitgehend auf die in Kapitel 2 beschriebenen Mechanismen zurückführen. Dieser Abschnitt ordnet die zentralen Beobachtungen den entsprechenden theoretischen Konzepten zu und geht dabei auch auf einen Bereich ein, für den die Theorie einen Effekt nahelegt, den die Messung jedoch nicht in nennenswertem Umfang zeigt.

Der deutlichste Bezug besteht zwischen den Kontextwechseln und dem in Abschnitt 2.2.2 sowie Abschnitt 2.2.3 beschriebenen Scheduling. Die dort dargestellte Unterscheidung zwischen den direkten und den indirekten Kosten eines Kontextwechsels findet sich über alle drei lokalen Szenarien hinweg in den Cache- und TLB-Werten wieder. Die durchgehend niedrigsten Cache-Misses der Virtual Threads und die durchgehend höchsten der Plattform-Threads belegen die theoretische Aussage, dass die indirekten Kosten eines Kontextwechsels die direkten in vielen Fällen übersteigen. Nicht die Zahl der Kontextwechsel allein, sondern deren Wirkung auf Cache und TLB bestimmt den Leistungsabfall, wie der gemeinsame Anstieg von Kontextwechseln, Cache- und TLB-Misses und der Rückgang der Instruktionen je Zyklus bei den Plattform-Threads zeigt. Der Vorteil des M:N-Modells von Go und den Virtual Threads besteht damit weniger im Einsparen der Kontextwechsel selbst, sondern im Erhalt der Cache-Lokalität, die durch die geringe Zahl langlebiger Plattform-Threads gewahrt bleibt. Node.js erweitert diese Analyse um einen Grenzfall, denn sein einzelner Thread führt zwangsweise die Cache-Lokalität bei der rechengebundenen Last auf ein Maximum und erreicht die niedrigsten Cache-Misses überhaupt, ohne dass daraus ein hoher Durchsatz folgt, weil die Lokalität nicht über die Parallelität hinweg nutzbar gemacht werden kann.

Ein Bereich, für den die Theorie einen Effekt beschreibt, der in der Darstellung der Ergebnisse jedoch bewusst nicht aufgegriffen wurde, ist die in Abschnitt 2.3 behandelte NUMA-Architektur. Die Messung erfasst mit `numa_migrated` die Zahl der Speicherseiten, die das in diesem Abschnitt beschriebene `Automatic NUMA Balancing` im Verlauf eines Laufs verschoben hat. Diese Zahl unterscheidet sich erheblich zwischen den Modellen, denn bei der rechengebundenen Last mit 10000 Aufgaben verschieben die Virtual Threads mit 360011 Seiten die meisten, während es bei Node.js nur 3907 sind. Trotz dieses Unterschieds besteht kein erkennbarer Zusammenhang zwischen der Zahl verschobener Seiten und dem Durchsatz oder der Latenz, denn die Virtual Threads gehören trotz der höchsten Migrationszahl zu den schnellsten Modellen. Dieses Ausbleiben eines Effekts lässt sich dadurch erklären, dass eine optimale NUMA-Verteilung in der Praxis häufig ein explizites Pinning der Threads erfordert, das in dieser Untersuchung nicht vorgenommen wurde, sodass alle Modelle gleichermaßen auf das automatische Verfahren angewiesen waren. Die hohe Migrationszahl der Virtual Threads ist wahrscheinlich eine Folge des Work-Stealing im ForkJoinPool, das Ausführungseinheiten häufig zwischen Kernen und damit zwischen Knoten verschiebt und so das automatische NUMA-Balancing auslöst, ohne dass daraus ein messbarer Nachteil entsteht. Damit scheint das automatische Balancing kein bestimmender Faktor zu sein, jedenfalls nicht ohne explizites Pinning, während der Aufwand für Kontextwechsel und die Auslastung des Block-Devices es sehr wohl sind.

Die Ergebnisse der netzwerkbasierten Last lassen sich unmittelbar an die in Abschnitt 2.1 und Abschnitt 2.4 beschriebenen Konzepte anschließen. Das ereignisgetriebene Modell wurde dort als Antwort auf das Problem vieler gleichzeitiger, überwiegend wartender Verbindungen dargestellt und genau in dieser Rolle zeigt es je Thread die höchste Wirksamkeit. Zugleich macht die Messung die Kehrseite sichtbar, denn die Bindung an einen Thread begrenzt den Gesamtdurchsatz und führt unter Überlast zum Abweisen von Verbindungen an der Accept-Queue. Die Multithreading-Modelle folgen dagegen dem Muster, dass sie Verbindungen annehmen und die Überlastung in Antwortzeiten übersetzen. Bemerkenswert ist dabei, dass die in der Theorie getrennt behandelten Modelle im Netzwerkfall nicht sich schlicht in schnell und langsam aufteilen, sondern in Modelle, die Last annehmen und verzögern und solche, die Last früh abweisen.

5.8 Grenzen der Untersuchung

Die vorstehende Einordnung beruht auf Messwerten, deren Aussagekraft durch die Bedingungen ihrer Erhebung begrenzt ist. Damit die abgeleiteten Erklärungen angemessen eingeordnet werden können, benennt dieser Abschnitt die wichtigsten methodischen Einschränkungen, die bei der Bewertung der Ergebnisse zu berücksichtigen sind.

Eine erste Einschränkung betrifft die Vergleichbarkeit der Parallelität. Node.js lief bei der rechengebundenen Last mit einem Thread und bei der dateibasierten sowie der gemischten Last mit einem libuv-Threadpool von vier, während die übrigen Modelle an die 32 logischen Kerne gebunden waren bzw. 16 logischen Kernen bei der netzwerkbasierten Last. Diese Wahl wurde in Abschnitt 4.1 bewusst getroffen, weil sie dem üblichen Einsatz von Node.js entspricht, sie führt jedoch dazu, dass der absolute Durchsatz von Node.js nicht unmittelbar mit dem der anderen Modelle verglichen werden kann. Die auf die einzelne Aufgabe beziehungsweise auf den einzelnen Thread bezogenen Kennzahlen federn diese Einschränkung etwas ab, sie heben sie aber nicht vollständig auf. Insbesondere lässt sich aus den vorliegenden Daten nicht ablesen, wie sich Node.js mit einem größeren libuv-Threadpool bei der dateibasierten Last verhalten würde.

Eine zweite Einschränkung liegt in der Bedeutung der Latenzperzentile bei den lokalen Szenarien. Die dort ausgewiesene Latenz einer Aufgabe umfasst die Zeit von deren Start bis zu deren Abschluss und damit auch die Wartezeit unter Konkurrenz. Sie beschreibt somit das Verhalten unter der jeweiligen Nebenläufigkeit und nicht die Dauer einer isoliert ausgeführten Aufgabe, was bei der Interpretation der Perzentile zu beachten ist. Ähnlich verhält es sich mit der auf die gelesene Datenmenge bezogenen CPU-Zeit, die als von der Kernzahl unabhängige Kennzahl gewählt wurde, weil bei der dateibasierten und der gemischten Last die pro Aufgabe geleistete Arbeit mit der Aufgabenzahl abnimmt und eine auf die einzelne Aufgabe bezogene Betrachtung hier keinen einheitlichen Vergleich erlaubt.

Eine dritte Einschränkung betrifft die Zahl der Wiederholungen. Der Lauf von Node.js bei 10000 Aufgaben wurde in den lokalen Szenarien ohne Wiederholung gemessen, sodass für diesen Punkt keine Streuung vorliegt. Auch die aufwendige Aufzeichnung des Schedulings über `perf sched record` wurde nur bei den beiden oberen Aufgabenstufen aktiviert, sodass die Aussagen zur Scheduling-Verzögerung auf diese Stufen beschränkt sind. Ebenso wurde der Daten-TLB bei der dateibasierten Last nicht aufgezeichnet, sodass die Betrachtung der Adressübersetzung dort entfällt. Diese Entscheidungen wurden in Abschnitt 4.1 mit der Laufzeit der Messreihe und der Kapazität des Testsystems begründet.

Eine vierte Einschränkung betrifft die Bedingungen, unter denen die `/proc`-Kennzahlen der einzelnen Aufgabenstufen erhoben wurden. Wie in Abschnitt 4.1 beschrieben, wird die Aufzeichnung des Schedulings über `perf sched record` nur bei den beiden oberen Aufgabenstufen 2500 und 10000 aktiviert und dort mit dem `/proc`-Messlauf desselben Durchlaufs kombiniert. Die `/proc`-Kennzahlen dieser beiden Stufen, also Wallclock, Durchsatz, Latenzperzentile und der Anteil der ungenutzten Maschinenkapazität, entstanden somit unter einer systemweiten Aufzeichnung sämtlicher Scheduling-Ereignisse, während dieselben Kennzahlen der Stufen 100 und 500 ohne diese Aufzeichnung erhoben wurden. Da die Aufzeichnung zusätzliche Rechen- und Schreiblast erzeugt, kann sie die Werte der oberen Stufen gegenüber den unteren geringfügig verschieben und damit die über die Aufgabenstufen gebildeten Verläufe verzerren. Ein systematischer Bruch zwischen den beiden unteren und den beiden oberen Stufen ist in den Messwerten allerdings nicht erkennbar, denn in den beiden Szenarien mit über alle Stufen konstanter Datenmenge, der dateibasierten und der gemischten Last, ändern sich Durchsatz und Laufzeit zwischen der Stufe 500 ohne Aufzeichnung und der Stufe 2500 mit Aufzeichnung nur im Bereich weniger Prozent. Der Effekt der Aufzeichnung auf die `/proc`-Kennzahlen erscheint damit klein, lässt sich aus den vorliegenden Daten aber nicht vollständig von der Wirkung der gestiegenen Nebenläufigkeit trennen und bleibt als möglicher Störeinfluss zu berücksichtigen.

Eine fünfte Einschränkung betrifft den Zähler `block_rq_complete`. Er zählt die vom Block-Device abgeschlossenen Requests, wird aber im Hardware-Interrupt- beziehungsweise Software-Interrupt-Kontext ausgelöst und damit außerhalb des Threads, der den Lesezugriff abgesetzt hat. Bei der prozessgebundenen Messung über `perf stat` wird das Ereignis nur dann gezählt, wenn es einem Thread des gemessenen Prozesses zugerechnet wird, sodass die abgeschlossenen Requests je nach Verteilung der Abschluss-Interrupts unterschiedlich stark untererfasst werden. Das erklärt, warum die abgeschlossenen Requests in Tabelle 4.14 und Tabelle 4.21 weit unter den nahezu gleichen abgesetzten Requests (block_rq_issue) liegen und zwischen den Modellen stark streuen, obwohl jeder abgesetzte Request am Ende abgeschlossen wird. Die absolute Höhe von `block_rq_complete` wird daher nicht als Maß für die Zahl gleichzeitig ausstehender Zugriffe herangezogen. Für diese Aussage stützt sich die Diskussion stattdessen auf den Anteil der ungenutzten Maschinenkapazität und den erreichten Durchsatz.

Eine sechste Einschränkung ergibt sich aus dem Aufbau der netzwerkbasierten Last. Der Netzwerk-Server lief mit 16 Worker-Threads für Go, die Plattform-Threads und die Virtual Threads sowie mit einem Thread für Node.js, sodass die dortige Parallelität von der der

lokalen Szenarien abweicht. Als Durchsatz wurde die clientseitig gemessene Rate verwendet, da die serverseitige Rate kurze Leerlaufphasen vor und nach der Last enthält und die als `oha_err_deadline` gezählten Anfragen wurden nicht als Fehler gewertet, da sie die beim Messende noch in Bearbeitung befindlichen Anfragen beschreiben. Schließlich beziehen sich alle Ergebnisse auf ein einzelnes Testsystem mit einer festen Konfiguration, sodass eine Übertragung auf abweichende Hardware, andere Kernzahlen oder eine andere NUMA-Architektur nur unter Vorbehalt möglich ist.

Zusätzlich liefert diese Untersuchung keinen Beleg für das historische C10k-Problem, da das Testsystem mit 10000 Verbindungen bereits an der unteren Grenze der damals beschriebenen Größenordnung endet. Die Messung zeigt jedoch, dass sich die untersuchten Modelle bereits innerhalb dieses Bereichs deutlich unterschiedlich verhalten.

5.9 Beantwortung der Forschungsfragen und Hypothesen

Auf der Grundlage der Interpretation lassen sich die in der Einleitung formulierten Forschungsfragen und Hypothesen beantworten. Dieser Abschnitt greift sie in ihrer ursprünglichen Reihenfolge auf. An einigen Stellen lässt die Datenlage jedoch keine eindeutige Antwort zu, was entsprechend kenntlich gemacht wird.

F1 Die erste Forschungsfrage nach den systemnahen Kosten klassischer Betriebssystem-Threads gegenüber den asynchronen und virtualisierten Modellen lässt sich anhand der drei genannten Kennzahlen klar beantworten. Bei der rechengebundenen Last mit 10000 Aufgaben erzeugen die Plattform-Threads mit 994037 Kontextwechseln, 50871 CPU-Migrationen und rund 18516 Millionen Cache-Misses auf jeder der drei Messachsen die mit weitem Abstand höchsten Werte, während die virtualisierten Modelle Go und die Virtual Threads sowie das asynchrone Modell von Node.js deutlich darunter liegen. Die systemnahen Kosten des Thread-per-Request-Modells sind damit nicht nur vorhanden, sondern übersteigen die der übrigen Modelle um ein Vielfaches und sie wachsen mit der Zahl der Aufgaben, wie die Verläufe über die Aufgabenstufen zeigen.

F2 Die zweite Forschungsfrage nach der Reduktion von Kontextwechseln und Cache-Invalidierungen durch ereignisgetriebene Architekturen und nach den Bedingungen, unter denen sich dieser Vorteil umkehrt, wird durch die Messungen differenziert beantwortet. Einerseits weist Node.js bei der reinen rechengebundenen Last mit 18876 Kontextwechseln und rund 222 Millionen Cache-Misses die niedrigsten Werte aller Modelle auf, sodass die erwartete Reduktion auf dem reinen Event-Loop-Pfad eintritt. Andererseits zeigt sich bei der dateibasierten und der gemischten Last ein gegenteiliger Befund, hier verzeichnet Node.js mit rund 3,9 Millionen beziehungsweise 4,0 Millionen die höchsten Kontextwechsel sowie die höchsten LLC-Misses aller Modelle. Ursächlich hierfür ist der libuv-Threadpool, der pro Lesezugriff Kernel-Wechsel

erzeugt, womit sich die Hypothese für den Datei-I/O-Pfad umkehrt. Der Vorteil der Architektur schlägt somit unter zwei Bedingungen in einen Nachteil um, zum einen, sobald blockierende I/O-Operationen auf den Threadpool ausgelagert werden und die Kontextwechsel massiv ansteigen. Zum anderen, sobald nennenswerte Rechenarbeit anfällt, die sich nur über mehrere Kerne verteilen ließe, wodurch der einzelne Thread den Durchsatz begrenzt und Node.js in diesen Szenarien den untersten Verlauf bildet.

F3 Die dritte Forschungsfrage betrachtet Goroutinen und Virtual Threads gemeinsam als die beiden Modelle, die blockierendes I/O vor den Anwendungsprogrammierer:innen verbergen und fragt einerseits, wie wirksam diese Abstraktion ist und andererseits, welche Kosten auf Systemebene messbar bleiben. Für die Bewertung der Abstraktionsleistung dient das explizite asynchrone Modell von Node.js als Maßstab, da dort dieselbe blockierende Wartezeit nur mit Callbacks, Promises oder `async/await`-Konstrukten im Quellcode bearbeitet werden kann. Goroutinen und Virtual Threads verbergen dieses, denn bei beiden erscheint ein blockierender Aufruf im Quellcode wie ein gewöhnlicher synchroner Aufruf. Die Wirksamkeit dieser Abstraktion zeigt sich dadurch, dass sie keine Reduktion des Durchsatzes verursacht. Bei der gemischten Last erreichen die Virtual Threads den höchsten Durchsatz aller vier Modelle und übertreffen damit sogar das ereignisgetriebene Modell (Abbildung 4.11) und auch die Goroutinen ordnen sich im vorderen Bereich ein. Bei der netzwerkbasierten Last liegt der auf einen einzelnen Thread bezogene Durchsatz von Node.js zwar über dem der beiden abstrahierenden Modelle (Abbildung 4.20), dieser Vorsprung lässt sich jedoch, wie in Abschnitt 5.8 dargelegt, nicht von der unterschiedlichen Parallelität trennen und daher nicht allein auf das Programmiermodell zurückführen. Für die Abstraktionsleistung ergibt sich damit ein positiver Befund, denn beide Modelle verbergen die Wartezeiten wirksam, ohne dass das sequentielle Programmiermodell einen erkennbaren Durchsatznachteil mit sich bringt.
Die Frage benennt drei mögliche Kostenquellen auf Systemebene, von denen sich zwei in den Messwerten belegen lassen und eine nicht. Die erste ist die Stack-Verwaltung. Der in Abschnitt 5.5 gezeigte Speicherbedarf für die Stacks der Ausführungseinheiten fällt bei beiden Modellen weit geringer aus als bei den Plattform-Threads, liegt bei den Virtual Threads jedoch über dem der Goroutinen, sodass die Verwaltung der dynamischen Stacks als verbleibende Kostenquelle nachweisbar ist. Die Zweite ist die Sättigung des darunterliegenden Carrier-Pools. Sie tritt bei der reinen dateibasierten Last auf, wo die Virtual Threads einen geringeren Durchsatz erreichen als Go und die Plattform-Threads. Wie in Abschnitt 5.2 dargelegt, kompensiert die JVM das Carrier-Capture blockierender Dateizugriffe zwar durch zusätzliche Carrier-Threads, deren Zahl mit 53 bis 86 jedoch deutlich unterhalb der möglichen Obergrenze von 256 bleibt, sodass die Zahl gleichzeitig ausstehender Lesezugriffe nicht das Niveau der Modelle mit einem Betriebssystem-Thread je Aufgabe erreicht. Der zugleich hohe Anteil der ungenutzten Maschinenkapazität von über 70 Prozent bestätigt, dass die Kerne dabei überwiegend warten. Der Carrier-Pool begrenzt den Durchsatz somit auf einem Niveau, das oberhalb der Grundparallelität, aber unterhalb der vollen Kompensation liegt, was die in der Frage genannte Sättigung abbildet. Die dritte genannte Kostenquelle, das Pinning virtueller Threads an ihren Carrier-Thread, lässt

sich aus den erhobenen Kennzahlen dagegen nicht nachweisen, da hierfür kein geeigneter Zähler erfasst wurde und sich ein solcher Effekt nicht eindeutig von den beiden übrigen Kostenquellen trennen ließe. Dieser Kostenfaktor kann daher nur als mit den Daten verträgliche Deutung benannt werden, nicht aber als eigenständig gemessener Effekt.

H1 Die erste Hypothese zum Speicherbedarf pro Ausführungseinheit wird durch die Messungen bestätigt. Wie in Abschnitt 5.5 gezeigt, belegen die Plattform-Threads bei der dateibasierten Last mit 10000 Aufgaben rund das Siebzehnfache des Speichers von Go und die Virtual Threads liegen mit 249 MiB klar zwischen den Goroutinen und den Plattform-Threads. Die erwartete Abstufung, nach der die dynamisch verwalteten Stacks der virtualisierten Modelle einen geringeren Bedarf aufweisen als die vorab reservierten Stacks der Plattform-Threads, tritt damit ein.

H2 Die zweite Hypothese zum Skalierungsverhalten unter hoher Nebenläufigkeit wird ebenfalls bestätigt. Der strukturelle Unterschied zwischen dem an die Ressourcen des Betriebssystems gebundenen Thread-per-Request-Modell und den übrigen Modellen wird ab der Größenordnung von 10000 Aufgaben in mehreren Kennzahlen sichtbar. Deutlich wird dies anhand des fallenden Durchsatzes und der Latenzverteilung der Plattform-Threads, die hinter die der übrigen Modelle zurückfallen und bei der netzwerkbasierten Last markiert dieselbe Größenordnung den Punkt, an dem sich das Verhalten der Modelle unter Überlast grundlegend unterscheidet.

H3 Die dritte Hypothese zum Scheduling-Overhead muss auf Basis der Messungen differenziert bewertet werden, da sie sich nur teilweise bestätigt. Einerseits erzeugt Node.js als ereignisgetriebenes Modell bei der rein rechengebundenen Last mit 18.876 Kontextwechseln und 161 CPU-Migrationen die niedrigsten Werte, was die erwartete geringe Zahl kernel-vermittelter Wechsel für den reinen Event-Loop-Pfad belegt. Zugleich wird bei dieser Last die in der Hypothese benannte Einschränkung sichtbar, da der einzelne Thread zum Engpass wird und den Durchsatz begrenzt, sodass der Vorteil des geringen Scheduling-Aufwands nicht in einen hohen Durchsatz umgesetzt werden kann. Andererseits weist Node.js bei der dateibasierten Last mit 3942375 Kontextwechseln den höchsten Wert aller Modelle auf, ebenso bei der gemischten Last mit rund 4,0 Millionen Wechseln. Ursächlich hierfür ist der libuv-Threadpool, der pro Lesezugriff Kontext-Wechsel erzwingt. Die Annahme eines reduzierten Scheduling-Overheads gilt somit ausschließlich für den reinen Event-Loop-Pfad und muss für Szenarien, in denen blockierende Datei-I/O-Operationen auf den Threadpool ausgelagert werden, verworfen werden.

H4 Die vierte Hypothese zur Abstraktionsleistung der Virtual Threads bildet den zentralen Konflikt der Arbeit ab und lässt sich nur mit einer Einschränkung beantworten. Im Kern bestätigen die Ergebnisse die Hypothese deutlich, denn die Virtual Threads erreichen unter der

gemischten und der netzwerkbasierten Last einen hohen Durchsatz bei einem sequentiellen Programmiermodell und übertreffen das ereignisgetriebene Modell bei der gemischten Last sogar. Das sequentielle Modell bringt somit keinen erkennbaren Durchsatznachteil mit sich, was das Anwendungsversprechen virtueller Threads stützt. Die Einschränkung betrifft den direkten Vergleich mit dem Event-Loop-Modell, denn Node.js lief mit einem einzelnen Thread beziehungsweise einem libuv-Threadpool von vier, während die Virtual Threads die volle Kernzahl nutzten. Ein Vergleich der absoluten Durchsätze wird dadurch verzerrt und je Thread bleibt das ereignisgetriebene Modell bei der rein an I/O gebundenen Last das wirksamste. Die Aussage, dass die Virtual Threads die Leistung des Event-Loop-Modells erreichen, lässt sich daher nur unter diesem Vorbehalt treffen. Für die praktisch bedeutsame Frage, ob das sequentielle Modell der Virtual Threads bei gemischten und I/O gebundenen Lasten eine vergleichbare Leistung erbringt, geben die Daten jedoch eine klare und positive Antwort.

6 Ergebnisbetrachtung und kritische Reflexion

Das in Abschnitt 1.2 formulierte Ziel einer systemnahen, empirisch unterlegten Analyse und eines Vergleichs der Nebenläufigkeitsmodelle hinsichtlich ihres Einflusses auf betriebssystem- und systemnahe Aspekte wurde im Kern erreicht. Die zugrundeliegenden Mechanismen wie Kontextwechsel, Thread-Migrationen, Cache-Miss-Raten und Scheduling-Overhead wurden für alle vier Modelle erhoben, über die vier Lastszenarien hinweg verglichen und auf die in Kapitel 2 beschriebenen theoretischen Konzepte zurückgeführt.

Auf die Forschungsfragen konnten evidenzbasierte Antworten gefunden und die Hypothesen überprüft werden. Zugleich wurden nicht alle Teilziele gleichermaßen erreicht und die belastbaren Aussagen der Arbeit sind durch die in Abschnitt 5.8 benannten Testbedingungen begrenzt. Die wichtigste Einschränkung betrifft die Vergleichbarkeit der Parallelität, denn Node.js lief mit einem Thread beziehungsweise einem libuv-Threadpool von vier, während die übrigen Modelle an die volle Kernzahl gebunden waren. Der absolute Durchsatz von Node.js lässt sich dadurch nicht unmittelbar mit dem der anderen Modelle vergleichen. Diese Entscheidung wurde zwar bewusst getroffen, dies mindert nichtsdestotrotz die Vergleichbarkeit.
Durch die auf die einzelne Aufgabe oder den einzelnen Thread bezogenen Kennzahlen soll diesem Nachteil entgegengewirkt werden, heben ihn aber nicht vollständig auf.
Ebenso ließ sich das in der dritten Forschungsfrage genannte Pinning virtueller Threads aus den erhobenen Kennzahlen nicht einzeln nachweisen, da hierfür kein geeigneter Zähler erfasst wurde. Darüber hinaus lies sich der Effekt nicht aus den übrigen Kennzahlen ableiten. Auch für das historische C10k-Problem liefert die Untersuchung keinen Beleg, da das Testsystem mit 10000 Verbindungen bereits an der unteren Grenze der namensgebenden Größenordnung endet. Schließlich beziehen sich alle Ergebnisse auf ein einzelnes Testsystem mit einer festen Konfiguration, sodass eine Übertragung auf abweichende Hardware, andere Kernzahlen oder eine andere NUMA-Architektur nur unter Vorbehalt möglich ist.

Bei einer erneuten Untersuchung sollte zusätzlich die Parallelität vergleichbar gemacht werden, so ließe sich Node.js zusätzlich mit einem über das Cluster-Modul auf mehrere Kerne verteilten Aufbau sowie mit wechselnden Größen des libuv-Threadpools messen, sodass sich der Einfluss der Parallelität vom Einfluss des Programmiermodells trennen ließe. Um das Pinning als eigenständigen Effekt zu erfassen, könnten die `jdk.VirtualThreadPinned`-Ereignisse des Java Flight Recorder aufgezeichnet werden.
Die Zahl der Wiederholungen müsste insbesondere für den nur einmal gemessenen Lauf von Node.js bei 10000 Aufgaben erhöht werden, um auch dort eine Standardabweichung ausweisen zu können. Ebenso sollten die Daten-TLB auch für die dateibasierte Last erfasst werden, um die Betrachtung der Adressübersetzung über alle lokalen Szenarien hinweg zu ermöglichen. Schließlich ließe sich die netzwerkbasierte Last zu deutlich höheren Verbindungszahlen hin ausdehnen, um den Bereich des C10k-Problems zu überschreiten und das Verhalten der Modelle unter echter Massenlast sichtbar zu machen. Keine dieser Einschränkungen stellt die zentralen Aussagen der Arbeit infrage, da sie sich auf die absolute Vergleichbarkeit einzelner Kennzahlen

und nicht auf die durchgehend beobachteten strukturellen Unterschiede zwischen den Modellen beziehen. Ein Teil der Kritikpunkte ist dabei von im Vorhinein bekannt gewesen und auf die nötige Begrenzung der Testlaufzeiten bzw. auf die begrenzte Anzahl der Hardware-Counter zurückzuführen.

6.1 Implikationen für Praxis und Forschung

Für die Praxis lässt sich aus den Ergebnissen eine differenzierte Empfehlung formulieren, die gemäß der Ziellast variiert. Für die in der Serverpraxis häufigen I/O gebundenen und gemischten Lasten, bei denen ein großer Teil der Bearbeitungszeit aus dem Warten auf externe Systeme besteht, bieten die Virtual Threads und die Goroutinen die günstigste Kombination aus hohem Durchsatz, sparsamer Cache-Nutzung und einem unveränderten sequentiellen Programmiermodell. Sie erlauben es den Entwickler:innen, blockierenden Code zu schreiben und dennoch eine Leistung zu erreichen, die früher eine explizite asynchrone Strukturierung des Programmcodes erfordert hätte. Der wesentliche praktische Vorteil dieser Modelle liegt somit darin, dass die in Kapitel 2 beschriebene Umstrukturierung der Anwendungslogik entfällt, ohne dass ein messbarer Nachteil bei der systemnahen Effizienz entsteht.

Für rein netzwerkgebundene Lasten mit vielen überwiegend wartenden Verbindungen bleibt das ereignisgetriebene Modell je Thread das effizienteste und hält die Antwortzeiten der angenommenen Anfragen auch unter hoher Last niedrig. Der Nachteil besteht darin, dass es sich nicht auf mehrere Kerne ausdehnen lässt, weshalb es in der Praxis auf eine Skalierung über mehrere Prozesse angewiesen ist, sobald der Gesamtdurchsatz einen einzelnen Kern übersteigt oder nennenswerte Rechenarbeit hinzukommt. Die Plattform-Threads sind demgegenüber nur bei geringer Nebenläufigkeit effizient, da ihre Kosten mit der Zahl gleichzeitig aktiver Aufgaben am stärksten steigen und sie bei hoher Nebenläufigkeit die höchsten Werte für Speicherbedarf, Kontextwechsel und die stärkste Belastung von Cache und TLB aufweisen. Go stellt über alle Szenarien hinweg die ausgewogenste Wahl dar und eignet sich damit besonders für Systeme, deren Lastprofil im Voraus nicht eindeutig festgelegt ist.

Für die bestehende Forschung ergibt sich zunächst, dass die systemnahen Kennzahlen einen signifikanten Effekt besitzen, der über die reine Betrachtung von Durchsatz und Latenz hinausgeht. Erst die Sicht auf die Gesamtheit der Kennzahlen wie Kontextwechseln, Cache- und TLB-Misses sowie der Instruktionen je Zyklus macht sichtbar, dass die beobachteten Leistungsunterschiede weniger aus der reinen Zahl der Kontextwechsel als aus deren Wirkung auf die Speicherhierarchie folgen. Vergleiche von Nebenläufigkeitsmodellen, die allein auf Kennzahlen der Anwendungsebene beruhen, können diese Ursache nicht aufzeigen und so zu Fehldeutungen führen.

6.2 Einordnung in die bestehende Forschung

Die Ergebnisse dieser Arbeit lassen sich unmittelbar an die in der Einleitung benannten Vorarbeiten anschließen und erweitern diese. Karsten und Barghi haben gezeigt, dass User-Level-Threading-Ansätze unter geeigneten Bedingungen eine zu reinen ereignisgetriebenen Systemen vergleichbare Leistung erreichen können, ihre Untersuchung wurde jedoch mit einem speziellen Forschungslaufzeitsystem erhoben und fokussierte auf Durchsatz- und Latenzkennzahlen auf Anwendungsebene. Die vorliegende Arbeit bestätigt diese Aussage für die breit eingesetzten Produktionssysteme Go-Runtime und JVM mit Virtual Threads und ergänzt sie um eine systemnahe Erklärung, denn sie führt die vergleichbare Leistung auf den Erhalt der Cache-Lokalität durch die geringe Zahl langlebiger Betriebssystem-Threads zurück. Damit wird nicht nur bestätigt, dass User-Space-Scheduling die Leistung ereignisgetriebener Modelle erreichen kann, sondern auch begründet, warum dies der Fall ist.

Die Ergebnisse zur gemischten und dateibasierten Last stehen zudem im Einklang mit empirischen Untersuchungen zu Virtual Threads in datenbankgetriebenen Serveranwendungen, die deren Eignung für I/O gebundene Lasten belegen. Über die inhaltliche Einordnung hinaus liefert die Arbeit einen methodischen Beitrag. Die entwickelte Messumgebung kombiniert die kontrollierte Erzeugung synthetischer Workloads für rechengebundene, dateibasierte, gemischte und netzwerkbasierte Lasten mit der Erfassung systemnaher Kennzahlen über die `/proc`-Schnittstelle und die Werkzeuge der `perf`-Familie. Dieser Aufbau ist nicht auf die vier untersuchten Laufzeitsysteme beschränkt, sondern lässt sich auf weitere Laufzeitumgebungen und Lastprofile übertragen, sodass die Testumgebung selbst ein wiederverwendbares Werkzeug für vergleichbare systemnahe Analysen darstellt.
Gleichzeitig ist die vorliegende Arbeit kein Produkt langjähriger Forschung, sondern als Abschlussarbeit eine Arbeit von wenigen Wochen und kann nur als ein erster Schritt im wissenschaftlichen Forschungsbereich gesehen werden.

Literaturverzeichnis

[1] R. Pressler und A. Bateman. „JEP 444: Virtual Threads,“ OpenJDK, besucht am 14. Mai 2026. Adresse: https://openjdk.org/jeps/444

[2] M. Karsten und S. Barghi, „User-level Threading: Have Your Cake and Eat It Too,“ *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, Jg. 4, Nr. 1, 2020. DOI: 10.1145/3379483

[3] D. Kegel. „The C10K Problem.“ Aktualisierungen bis 2014, besucht am 14. Mai 2026. Adresse: https://www.kegel.com/c10k.html

[4] libuv contributors. „libuv Documentation – Design Overview,“ besucht am 14. Mai 2026. Adresse: https://docs.libuv.org/en/v1.x/design.html

[5] B. Nystrom, *What Color Is Your Function?* https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/, Online, abgerufen am 16.05.2026, 2015.

[6] J.-H. Liu, „Design and Implementation of a Low-Latency High-Frequency Trading System for Cryptocurrency Markets,“ 2025, Preprint und Open-Source-Referenzimplementierung, https://medium.com/@gwrx2005/design-and-implementation-of-a-low-latency-high-frequency-trading-system-for-cryptocurrency-markets-a1034fe33d97.

[7] W. Zhao und M. Q. Yang, „Dependability enhancing mechanisms for integrated clinical environments,“ *The Journal of supercomputing*, Jg. 73, Nr. 10, S. 4207–4220, 2017.

[8] R. Brause, *Betriebssysteme: Grundlagen und Konzepte*, 4. Aufl. Berlin und Heidelberg: Springer Vieweg, 2017, ISBN: 978-3-662-54099-2. DOI: 10.1007/978-3-662-54100-5

[9] B. Gregg, „Thinking Methodically about Performance,“ *ACM Queue*, Jg. 10, Nr. 12, 2012, Erweiterte Fassung in Communications of the ACM, Vol. 56, No. 2, 2013. DOI: 10.1145/2405116.2413037

[10] R. Veen und D. Vlijmincx, *Virtual Threads, Structured Concurrency, and Scoped Values – Explore Java's New Threading Model*. Berkeley, CA: Apress, 2024, ISBN: 979-8-8688-0499-1.

[11] Oracle Corporation. „Virtual Threads – Java Platform, Standard Edition Core Libraries,“ Oracle Corporation, besucht am 20. Mai 2026. Adresse: https://docs.oracle.com/en/java/javase/21/core/virtual-threads.html

[12] P. C. Mateo und A. Bateman. „JEP 491: Synchronize Virtual Threads without Pinning,“ OpenJDK, besucht am 20. Mai 2026. Adresse: https://openjdk.org/jeps/491

[13] OpenJS Foundation. „The Node.js Event Loop, Timers, and process.nextTick(),“ besucht am 14. Mai 2026. Adresse: https://nodejs.org/learn/asynchronous-work/event-loop-timers-and-nexttick

[14] Linux Kernel Community. „perf: Linux profiling with performance counters,“ besucht am 14. Mai 2026. Adresse: https://perfwiki.github.io/main/

[15] J. Ousterhout, *Why Threads Are A Bad Idea (for most purposes)*, Invited talk, USENIX Annual Technical Conference, San Diego, CA, 1996. Adresse: https://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf

[16] E. W. Dijkstra, „Cooperating Sequential Processes,“ *Technical Report EWD-123*, 1965, Nachdruck in F. Genuys (Hrsg.): *Programming Languages*, Academic Press, London, 1968.

[17] C. A. R. Hoare, „Communicating Sequential Processes,“ *Communications of the ACM*, Jg. 21, Nr. 8, S. 666–677, 1978. DOI: 10.1145/359576.359585

[18] Apache Software Foundation, *Apache MPM Common Directives und Multi-Processing Modules (MPMs)*, https://httpd.apache.org/docs/2.4/mpm.html, Online, abgerufen am 16.05.2026, Apache HTTP Server Project.

[19] W. Reese, *nginx: The High-Performance Web Server and Reverse Proxy*, https://www.linuxjournal.com/magazine/nginx-high-performance-web-server-and-reverse-proxy, 2008.

[20] D. Pariag, T. Brecht, A. Harji, P. Buhr und A. Shukla, „Comparing the Performance of Web Server Architectures,“ in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, ACM, 2007, S. 231–243. DOI: 10.1145/1272996.1273021

[21] A. S. Tanenbaum und H. Bos, *Modern Operating Systems*, 4. Aufl. Boston, MA: Pearson, 2014, ISBN: 978-0133591620.

[22] R. Graham, *The C10M Problem*, http://c10m.robertgraham.com/p/manifesto.html, Online, abgerufen am 16.05.2026, 2013.

[23] M. Welsh, D. Culler und E. Brewer, „SEDA: An Architecture for Well-Conditioned, Scalable Internet Services,“ in *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, 2001, S. 230–243. DOI: 10.1145/502034.502057

[24] The Go Authors. „Frequently Asked Questions (FAQ) – The Go Programming Language (Origins),“ The Go Programming Language, besucht am 4. Juli 2026. Adresse: https://go.dev/doc/faq#Origins

[25] L. Soares und M. Stumm, „FlexSC: Flexible System Call Scheduling with Exception-Less System Calls,“ in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, BC, Canada: USENIX Association, 2010, S. 33–46, ISBN: 978-1-931971-79-9.

[26] M. Kerrisk. „pthread_create(3) – Linux manual page.“ Linux man-pages 6.16, The Linux man-pages project, besucht am 20. Mai 2026. Adresse: https://man7.org/linux/man-pages/man3/pthread_create.3.html

[27] C. Li, C. Ding und K. Shen, „Quantifying the Cost of Context Switch,“ in *Proceedings of the 2007 Workshop on Experimental Computer Science* (*ExpCS '07*), San Diego, CA, USA: ACM, 2007. DOI: 10.1145/1281700.1281702

[28] Linux Kernel Community. „EEVDF Scheduler – Linux Kernel Documentation,“ besucht am 19. Mai 2026. Adresse: https://docs.kernel.org/scheduler/sched-eevdf.html

[29] Linux Kernel Community. „CFS Scheduler – Linux Kernel Documentation,“ besucht am 18. Mai 2026. Adresse: https://docs.kernel.org/scheduler/sched-design-CFS.html

[30] M. T. Jones, „Inside the Linux 2.6 Completely Fair Scheduler,“ *IBM developerWorks Technical Library*, 2009. besucht am 18. Mai 2026. Adresse: https://developer.ibm.com/tutorials/l-completely-fair-scheduler/

[31] I. Stoica und H. Abdel-Wahab, „Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation,“ Old Dominion University, Technical Report TR-95-22, 1995.

[32] U. Drepper, *What Every Programmer Should Know About Memory*, https://people.freebsd.org/~lstewart/articles/cpumemory.pdf, Online, abgerufen am 19.05.2026, 2007.

[33] J. L. Hennessy und D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6. Aufl. Cambridge, MA: Morgan Kaufmann, 2019, ISBN: 978-0128119051.

[34] M. S. Papamarcos und J. H. Patel, „A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories,“ in *Proceedings of the 11th Annual International Symposium on Computer Architecture* (*ISCA '84*), Ann Arbor, MI, USA: ACM, 1984, S. 348–354. DOI: 10.1145/800015.808204

[35] C. Lameter, „NUMA (Non-Uniform Memory Access): An Overview,“ *ACM Queue*, Jg. 11, Nr. 7, S. 40–51, 2013. DOI: 10.1145/2508834.2513149

[36] J. Axboe, *Efficient IO with io_uring*, https://kernel.dk/io_uring.pdf, Online, abgerufen am 21.05.2026, Okt. 2019.

[37] L. Lasić, D. Beronić, B. Mihaljević und A. Radovan, „Assessing the Efficiency of Java Virtual Threads in Database-Driven Server Applications,“ in *47th International Convention on Information, Communication and Electronic Technology* (*MIPRO*), IEEE, 2024, S. 2046–2051.

[38] libuv contributors. „libuv Documentation – Thread pool work scheduling,“ besucht am 26. Mai 2026. Adresse: https://docs.libuv.org/en/v1.x/threadpool.html

[39] OpenJS Foundation. „Don't Block the Event Loop (or the Worker Pool),“ besucht am 26. Mai 2026. Adresse: https://nodejs.org/en/learn/asynchronous-work/dont-block-the-event-loop

[40] D. Vyukov, *Scalable Go Scheduler Design Doc*, https://docs.google.com/document/d/1TTj4T2JO42uD5ID9e89oa0sLKhJYD0Y_kqxDv3I3XMw, Go 1.1 Design Document, abgerufen am 26.05.2026, 2012.

[41] The Go Authors. „Package `runtime` – `GOMAXPROCS`," The Go Programming Language, besucht am 26. Mai 2026. Adresse: https://pkg.go.dev/runtime#GOMAXPROCS

[42] The Go Authors. „Go 1.4 Release Notes," The Go Programming Language, besucht am 26. Mai 2026. Adresse: https://tip.golang.org/doc/go1.4

[43] The Go Authors. „Frequently Asked Questions (FAQ) – Goroutines," The Go Programming Language, besucht am 26. Mai 2026. Adresse: https://go.dev/doc/faq%5C#goroutines

[44] The Go Authors, *Go 1.19 Release Notes*, https://go.dev/doc/go1.19, Besucht am 25. Juni 2026, 2022.

[45] The Go Authors. „Go 1.14 Release Notes – Runtime," The Go Programming Language, besucht am 26. Mai 2026. Adresse: https://go.dev/doc/go1.14%5C#runtime

[46] A. Clements. „Proposal: Non-cooperative goroutine preemption," besucht am 26. Mai 2026. Adresse: https://go.googlesource.com/proposal/+/master/design/24543-non-cooperative-preemption.md

[47] Linux Kernel Community. „CPU Performance Scaling – Linux Kernel Documentation," besucht am 12. Juni 2026. Adresse: https://docs.kernel.org/admin-guide/pm/cpufreq.html

[48] T. Mytkowicz, A. Diwan, M. Hauswirth und P. F. Sweeney, „Producing Wrong Data Without Doing Anything Obviously Wrong!" In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XIV)*, Washington, DC, USA: ACM, 2009, S. 265–276. DOI: 10.1145/1508244.1508275

[49] A. Georges, D. Buytaert und L. Eeckhout, „Statistically Rigorous Java Performance Evaluation," in *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '07)*, Montreal, Quebec, Canada: ACM, 2007, S. 57–76. DOI: 10.1145/1297027.1297033

[50] E. Barrett, C. F. Bolz-Tereick, R. Killick, S. Mount und L. Tratt, „Virtual Machine Warmup Blows Hot and Cold," *Proceedings of the ACM on Programming Languages*, Jg. 1, Nr. OOPSLA, S. 1–27, 2017. DOI: 10.1145/3133876

[51] B. Schroeder, A. Wierman und M. Harchol-Balter, „Open Versus Closed: A Cautionary Tale," in *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, USA: USENIX Association, 2006.

[52] B. Gregg, „Visualizing System Latency," *ACM Queue*, Jg. 8, Nr. 5, S. 30–42, 2010. DOI: 10.1145/1794514.1809426

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Abkürzungsverzeichnis

Erklärung zur Nutzung von KI

Im Sinne der wissenschaftlichen Redlichkeit und in Anbetracht der Tatsache, dass es sich bei der vorliegenden Arbeit um eine benotete Prüfungsleistung handelt, möchten wir an dieser Stelle ausführlich auf die Nutzung von künstlicher Intelligenz bei der Erstellung dieser Arbeit eingehen.

Diese Arbeit ist unseres Erachtens auch unter sehr kritischer Betrachtung ein von uns eigenständig verfasstes Werk, dies muss es natürlich für eine Masterarbeit auch sein, dennoch hat KI während verschiedener Prozesse eine Rolle gespielt. So kam künstliche Intelligenz als Reviewer einzelner Abschnitte und Kapitel zum Einsatz. Sie sollte beurteilen, ob Mechanismen, Aussagen, Überlegungen oder Erklärungen korrekt und gut dargelegt sind und ob die Literaturauswahl stimmig ist. Ebenso wurde sie genutzt, um Formulierungen und den allgemeinen Sprachgebrauch zu prüfen.

Dabei sollte die KI die Rolle der Prüfenden einnehmen und ein ausführliches Feedback erstellen. Dieses Feedback wurde von uns gegen Literatur geprüft, besprochen und floss, wenn wir dies für sinnvoll erachtet haben, in unsere Arbeit ein. Dabei wurden keine Textpassagen generiert, welche eins zu eins übernommen wurden, und dennoch sind wir überzeugt, dass sich ein gewisser Einfluss, sei es sprachlich, methodisch oder fachlich, in der Arbeit wiederfinden lässt.

Besondere Erwähnung müssen dabei die Testprogramme finden, deren Erstellung zwar nicht das Ziel der Arbeit war, aber ohne diese keine aussagekräftigen Ergebnisse möglich gewesen wären. Dieser Prozess wäre ohne künstliche Intelligenz zumindest nicht in der zur Verfügung stehenden Zeit möglich gewesen. Moderne CPUs und Betriebssysteme sind hochoptimiert und darauf ausgelegt, systemnahe Effekte maximal zu reduzieren. Programme, die genau diese Effekte aufzeigen können, mussten folglich diese Optimierungen unterlaufen und zusätzlich noch in allen Programmiersprachen äquivalent und korrekt umgesetzt werden.
KI kam in diesem Abschnitt der Arbeit sicherlich am stärksten zum Einsatz und hat nicht nur einen Reviewprozess durchgeführt, sondern auch aktiv Programmcode korrigiert, optimiert und erweitert, um die von uns gestellten Anforderungen zu erfüllen. Auch hierbei wurden keine Bestandteile ohne Prüfung übernommen, dennoch sind hierbei auch generierte Codeabschnitte in der endgültigen Abgabe zu finden. Da die Programme, wie bereits erwähnt, nicht das Ziel der Arbeit waren, sondern ein notwendiger Bestandteil zur Beantwortung der Forschungsfragen, erachten wir diesen Einsatzzweck als vertretbar und sinnvoll. Die abschließende Beurteilung obliegt dabei natürlich den Prüfenden.

Selbstständigkeitserklärung

Wir versichern, die von uns vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, haben wir als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die wir für die Arbeit benutzt haben, sind angegeben. Die Arbeit haben wir mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegt. Für die Nutzung von künstlicher Intelligenz verweisen wir ausdrücklich auf die separate Erklärung zur Nutzung von KI.

Bremerhaven, den 25. Juli 2026 Unterschrift: 

Bremerhaven, den 25. Juli 2026 Unterschrift: