Betriebssysteme
Übungsblatt 2

Kilian Bartz (Mknr.: 1538561)

Hinweis

Der zugehörige Code befindet sich im folgenden Github Repository: `https://github.com/kilianbartz/bs_ueb2`.

Vorab: Verteilungsmodell und Metriken für Latenzen

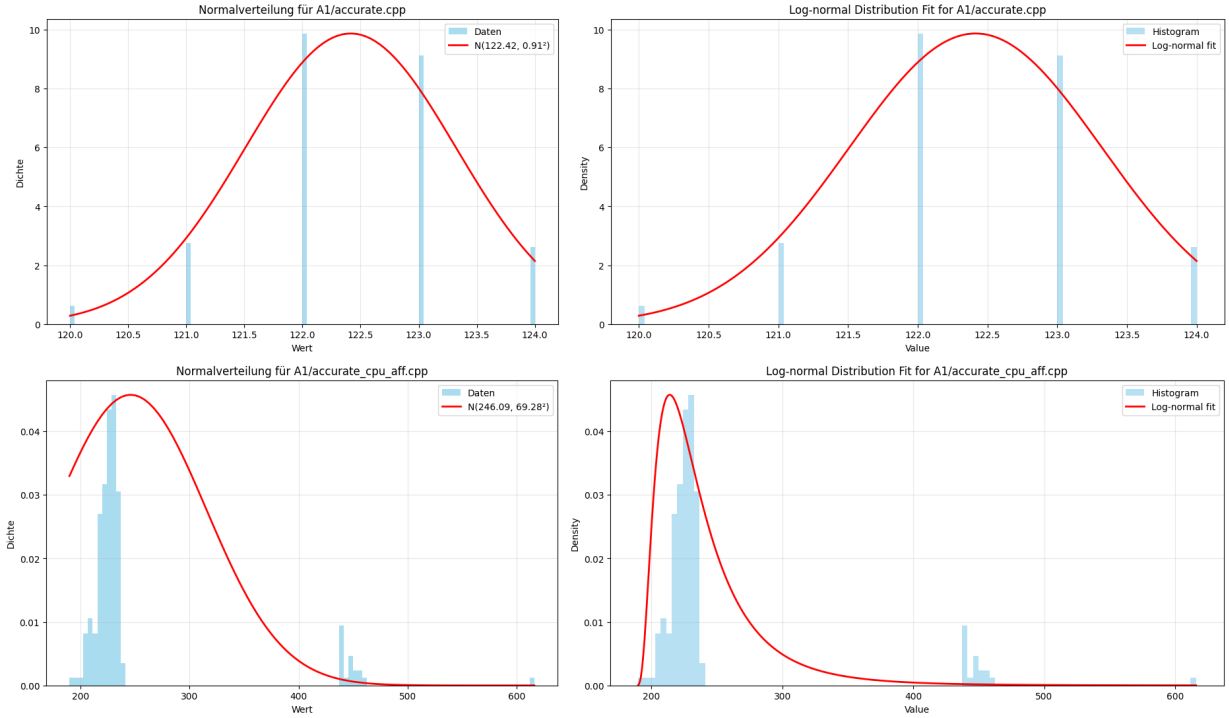

Abbildung 1: Modellierung zweier Latenzmessungen als Normalverteilung (links) und logarithmische Normalverteilung (rechts)

In der Praxis nimmt man für viele Zufallsvariablen eine Normalverteilung an. Dies liegt vor allem an der Erkenntnis des zentralen Grenzwertsatzes[1], dass Verteilungen, die durch die **additive** Überlagerung von vielen unabhängigen Einflüssen entstehen, gut durch eine Normalverteilung angenähert werden können. Denkt man an einen Rechner, so hat man viele Einflüsse auf IPC Latenzen, z. B. Scheduling, Speicherverwaltung, Speicherzugriffszeiten, Kontextwechsel und eventuell sogar der Netzwerk-Stack und der I/O-Stack.
Für eine grobe Approximation mag eine Normalverteilung ausreichen (siehe Abbildung 1), jedoch scheint sie für mich keine gute Näherung zu sein, da

- die Normalverteilung symmetrisch ist, wohingegen Latenzen oft eine stark rechtsschiefe Verteilung annehmen,
- die Einflüsse nicht unabhängig voneinander sind (Scheduling und Speicherzugriff beeinflussen die meisten der anderen Faktoren),

[1] `https://de.wikipedia.org/wiki/Zentraler_Grenzwertsatz`

- die Annahme, dass sich die Einflüsse addieren nicht realistisch ist.

Ein Netzartikel zu dem Thema [3] beschreiben noch weitere Schwächen der Normalverteilung zur Modellierung von Latenzen. Um zumindest zwei dieser Schwächen zu adressieren, habe ich mich für die Modellierung im Folgenden für eine logarithmische Normalverteilung[2] entschieden. Einerseits kann man mit ihr rechtsschiefe Prozesse modellieren, andererseits ergibt sie sich als multiplikative Überlagerung viele unabhängiger Einflüsse. Während die Unabhängigkeit noch immer eine beschönigende Annahme darstellt, ist die multiplikative Überlagerung realistisch. Betrachtet man den Rechner als eine Pipeline an Komponenten, um IPC umzusetzen, ergibt sich die gesamte Effizienz als ein Produkt der Effizienzen der einzelnen Komponenten (hat bspw. Komponente A eine Effizienz von 90% und Komponente B eine Effizienz von 80%, beläuft sich die Gesamteffizienz auf 72%). Aufgrund dieser Überlegungen scheint die logarithmische Normalverteilung für mich die geeignetste univariate Verteilung zu sein. Für das realistischste Modell müsste man vermutlich auf einigermaßen komplexe multivariate Verteilungen zurückgreifen.

Metriken zur Beschreibung der Latenzen

Wie diese zwei interessanten Berichte [3, 4] beschreiben, sind Mittelwerte und Standardabweichung nur bedingt geeignet, um ein realistisches Bild von Latenzen zu schaffen. Allem voran liefern sie kein gutes Bild davon, wie häufig man mit welchen Ausreißern rechnen muss, weshalb Quantile die zugrundeliegenden Daten besser beschreiben. Selbst bei der Angabe eines 99% Quantils (also nur 1% der Messungen dauerte länger als der angegebene Wert) tritt die Latenz im restliche Prozent der Zeit (bei so etwas elementarem wie IPC) immer noch oft genug auf, um eine spürbare Auswirkung auf die Benutzererfahrung zu haben.
Auch wenn es in erwähnten Berichten eher um maximale Latenzen und ihre Auswirkungen auf die Benutzererfahrung geht, würde ich dafür argumentieren, dass diese Punkte auch für die minimale Latenz zutreffen und somit ein akkurateres Bild liefern, welche Latenzen in der Praxis mit den unterschiedlichen Methoden zu erwarten sind. Daher gebe ich neben dem Mittelwert und dem 95% Konfidenzintervall auch das 1% und das 99% Quantil der Latenzen an.

Aufgabe 1: Spin Locks

Die Implementierungen für Aufgabe 1 - 4 folgen alle der gleichen Vorlage: Ein Thread (**Server** bzw. Main-Thread) möchte mit dem zweiten Thread kommunizieren und setzt dazu den Wert einer `shared_data` Variable und speichert in der Variablen `start` den aktuellen Wert des High-Performance Counters `rtdsc`, welchen ich wie im ersten Übungsblatt für die genaust-möglichen Messungen verwende.
Nach der Freigabe des kritischen Bereichs ist es nun die Aufgabe des zweiten Threads (**Client**), den Wert von `shared_data` zu lesen, aufzuzeichnen, wie viel Zeit zwischen dem Setzen der Variablen bis jetzt vergangen ist (`__rtdsc()-start`) und `shared_data` abschließend wieder zurückzusetzen. Die vergangene Zeit in Zyklen wird dabei über eine feste Anzahl k von Iterationen gemessen und gemittelt ausgegeben.

Besonderheiten von Spinlocks

Spinlocks (zumindest solche, die man ohne viele Bauchschmerzen in der Praxis einsetzen würde) beruhen auf *busy waiting* mittels einer `test_and_set` Instruktion, welche atomar ein Flag innerhalb eines gemeinsamen Speicherbereichs liest und auf einen festgelegten Wert setzt. Durch diese Instruktion werden alle anderen Threads, die zeitgleich auf den gleichen kritischen Abstand zugreifen wollen, solange am Betreten gehindert, bis der erste Thread mittels einer `clear` Instruktion das Flag zurücksetzt.
Dabei erlauben Spin-Locks im Gegensatz zu den anderen Verfahren keine genaue Kontrolle darüber, welcher Thread als nächstes den kritischen Bereich betritt. Eventuell könnte so der Server m mal drankommen, bevor der Client den Wert von `shared_data` lesen und verarbeiten kann. Da es jedoch um eine tatsächliche Kommunikation geht, wird die Zeit nur gestoppt, wenn der Client den kritischen Bereich betritt, nicht ein beliebiger Thread. Die eventuell auftretenden m zusätzlichen Loops des Servers verschwenden in diesem Fall möglicherweise viel Performance (andernfalls hätten vielleicht einige abwechselnde Server/Client Iterationen

[2] `https://en.wikipedia.org/wiki/Log-normal_distribution`

in derselben Zeit stattfinden können) zählen meiner Ansicht nach jedoch zur Latenz und sollten somit mitgemessen werden.

Testmethodologie

Alle Messungen wurden auf meinem Rechner mit einem Ryzen 7 7700X und 32GB RAM durchgeführt. Als Betriebssystem kam ein frisches Ubuntu 24.10 zum Einsatz. Während der Messungen wurden keine anderen Benutzerprogramme ausgeführt. Sofern nicht anders notiert, habe ich für jede der folgenden Messugnen die Zeit (in CPU-Zyklen) habe über $k = 1.000.000$ Iterationen gemessen. Jede Messung wurde $n = 200$ mal mithilfe des Skripts `measure.py` wiederholt, um eine statistische Relevanz zu erreichen. Zudem wurde in für die Messungen in Aufgabe 1 und 2 mittels `chrt -r 90` eine sehr hohe *Realtime Priorität* der gemessenen Prozesse, sodass diese vor (fast) allen anderen Systemprozessen vom Scheduler priorisiert werden sollten. Die Implementierung für alle Aufgaben ist in C++ als systemnahe Programmiersprache erfolgt.

Ergebnisse

Implementierung	Mittelwert	Untere Grenze	Obere Grenze	1% Quantil	99% Quantil
A1/accurate.cpp	122.41	122.28	122.54	120.00	124.00
A1/accurate_cpu_aff.cpp	239.50	232.52	246.70	198.96	458.03
A1/shared_mem.cpp	**111.97**	**111.67**	**112.27**	**110.00**	**114.00**

Tabelle 1: Gemessene Metriken und Konfidenzintervalle für meine drei Spinlock Implementierungen in CPU-Zyklen: Mittelwert, untere und obere Grenze des 95% Konfidenzintervalls, sowie das 1% und 99% Quantil meiner Messungen.

Betrachtet man die Ergebnisse in Tabelle 1, so finden sich wie zu erwarten in allen Fällen niedrige Latenzen von ca. 122 CPU-Zyklen bzw. 22.5 ns (siehe Tabelle 5). Die Zeit, die benötigt wird, um mittels *busy waiting* kritische Bereiche zu betreten wäre eigentlich kürzer (nach einer früheren Rechnung etwa 6 ns), jedoch verlängert diese Latenz durch Forcierung des abwechselnden Betretens knapp auf das Vierfache. Somit lässt sich vermuten, dass ein Thread im Schnitt 3 mal hintereinander in den kritischen Bereich läuft, bevor der andere Thread Erfolg beim Betreten hat. Gibt es eine höhere *Spinlock Contention* und mehr Threads versuchen zeitgleich auf das gleiche Spinlock zuzugreifen, verringert sich die produktiv genutzte Zeit beider Threads sicherlich weiter. Auch, wenn dies etwas ineffizient erscheint, ist es trotzdem mit deutlichem Abstand die schnellste Methode der Synchronisierung zweier Threads. Der Umstand, dass alle Messungen sehr dicht beieinander liegen (kleine Differenz zwischen 99% Quantil und 1% Quantil) spricht dafür, dass Spinlocks zudem sehr robust und vorhersehbar sind und ihre Effizienz nicht stark von äußeren Einflüssen beeinträchtigt wird.
Auffällig ist, dass die Latenz bei gesetzter CPU-Affinität (`A1/accurate_cpu_aff.cpp`) im Mittel etwa doppelt so groß und in schlimmen Fällen (99% Quantil) sogar 4 mal so groß ist. Für die Begründung dessen habe ich zwei Ideen:

- Die CPU-Affinität selbst ist das Problem (ein anderer Thread, der ebenfalls auch nur auf demselben Kern laufen kann, könnte ihm die CPU-Zeit streitig machen).
- Die physischen Kerne (0 und 1) waren schlecht gewählt und könnten z. B. auf unterschiedlichen *CPU-Dies* liegen. Somit wird die Kommunikation zwischen den Kernen bereits durch die CPU-Architektur (z. B. AMDs Infinity Fabric) verlangsamt.

Schließlich ist überraschenderweise die *Shared Memory* Implementierung mit zwei Prozessen in allen Metriken etwas schneller als `accurate.cpp`, was zwei Threads im selben Prozess verwendet. Dies könnte damit zusammenhängen, dass die zwei Threads denselben Adressraum teilen, während jeder Prozess einen separaten Adressraum erhält. Eventuell könnte dies zu gelegentlichen Konflikten führen, die etwas teurer sind.

Aufgabe 2: Semaphore (Mutex)

Meine erste Idee für diese Aufgabe war es, einen Mutex (für die Sicherung der kritischen Bereiche) zusammen mit einer `condition_variable` (um zu signalisieren, dass der andere Thread an der Reihe ist) zu verwenden. In Anlehnung an das *Reader-writers Problem* ist mir jedoch dann eine elegantere Lösung eingefallen, welche eine Kommunikation zwischen den beiden Threads ausschließlich mittels zwei binären Semaphoren realisiert. Der Server gibt dabei `semaphore_client` frei, nachdem er die Iteration fertig bearbeitet hat (also `shared_data` und `start` gesetzt wurde) und wartet bis die nächste Iteration mit einer Freigabe der `semaphore_server` beginnt. Der Client wartet derweil auf die Freigabe von `semaphore_client` und gibt am Ende seiner Arbeit `semaphore_server` frei.

Ergebnisse

Implementierung	Mittelwert	Untere Grenze	Obere Grenze	1% Quantil	99% Quantil
A2/semaphore.cpp	671.07	642.52	700.90	153.00	1141.00

Tabelle 2: Gemessene Metriken und Konfidenzintervalle für meine Semaphore-Implementierung in CPU-Zyklen: Mittelwert, untere und obere Grenze des 95% Konfidenzintervalls, sowie das 1% und 99% Quantil meiner Messungen.

Wie zu erwarten, ist die mittlere Latenz bei Kommunikation über die Semaphore, welche vom Betriebssystem bereitgestellt werden, deutlich höher als über Spinlocks (etwa Faktor 6) und liegt somit im Mittel schon über 100ns. Zusätzlich ist hier der Einfluss von äußeren Faktoren (z. B. dem Scheduler) viel deutlicher zu sehen, so kann die Kommunikation über Semaphore im schlechtesten Fall etwa 7.5 mal länger dauern als in den schnellsten 1% der Fälle.
Trotzdem fällt diese Form der Kommunikation je nach Einsatzszenario wahrscheinlich kaum ins Gewicht (wenn die Threads die meiste Zeit mit Berechnungen und nur wenig Zeit mit Kommunikation verbringen) und verzichtet außerdem auf *busy waiting*.

Aufgabe 3

ZeroMQ ist eine Library für asynchrone Nachrichten und Nebenläufigkeit. Sie bietet dabei eine Socket-basierte Kommunikation zwischen Threads zur Verfügung, welche

- innerhalb des selben Prozesses leben (`inproc://`)
- in zwei unterschiedlichen Prozessen leben (`ipc://`)
- auf zwei unterschiedlichen Machinen leben (z. B. `tcp://`, `multicast://`).

Dabei unterstützt ZeroMQ eine Vielzahl von Protokollen und *Messaging Patterns* (u. a. Request-reply oder pub-sub). ZeroMQ verspricht außerdem "zero latency, zero cost and zero admission". Zum einen kann dies die Kommunikation zwischen Threads deutlich simplifizieren (über Nachrichteninhalte nachzudenken fällt deutlich einfacher als über Zustände von geteilten Variablen und Semaphoren); zum anderen lässt sich einem Ansatz, der für die Kommunikation lokaler Threads funktioniert, auf ein verteiltes System ausweiten, ohne, dass man dazu neue Werkzeuge bräuchte.

Hinweis

ZeroMQ verarbeitet die I/O, die für die nachrichtenbasierte Kommunikation anfällt, in einem oder mehreren Hintergrundthreads. Die Zahl der Hintergrundthreads ist dabei ein konfigurierbarer Parameter. Da ein I/O Thread jedoch für bis zu einem GB an Daten pro Sekunde ausreichen sollte[2], habe ich für jede der folgenden ZeroMQ Lösungen nur einen I/O Thread verwendet.

Um eine Baseline zu schaffen und die Overheads der verschiedenen Protokolle miteinander vergleichen zu können, habe ich im Folgenden sowohl `inproc` (in `threads.cpp`), `ipc` und `tcp` (`client.cpp` und `server.cpp`) ausprobiert.

Ergebnisse

Implementierung	Mittelwert	Untere Grenze	Obere Grenze	1% Quantil	99% Quantil
A3_A4/threads.cpp	58380.93	54368.14	62689.91	9775.43	178870.71
A3_A4/server_client.cpp	57145.57	56970.80	57320.87	54882.42	59572.21
A3_A4/server_client_tcp.cpp	84733.47	79705.51	90078.61	81202.72	82859.96

Tabelle 3: Gemessene Metriken und Konfidenzintervalle für meine ZeroMQ-Implementierungen in CPU-Zyklen: Mittelwert, untere und obere Grenze des 95% Konfidenzintervalls, sowie das 1% und 99% Quantil meiner Messungen.

Erwartungsgemäß ist die nachrichtenbasierte Kommunikation deutlich langsamer als eine Kommunikation mittels geteilter, primitiver Variablen und Semaphoren. Trotzdem hat mich überrascht, wie deutlich der Unterschied (Faktor 85 im Mittel) ausgefallen ist. So liegt die Latenz meiner Implementierungen im Mittel bereits im niedrigen μs Bereich. Neben dem gesteigerten Overhead durch Nachrichten und die Übermittlungsprotokolle (`tcp` steigert die Latenz nochmals um ca. 46% ggü. `ipc`) wäre es durchaus wahrscheinlich, dass ein größerer Teil der Latenz Optimierungen seitens ZeroMQ geschuldet sind (z. B. Puffern von Nachrichten, verzögertes Versenden, o. ä.).
Zudem ist auch hier die Implementierung mit zwei separaten Prozessen marginal (etwa 2%) schneller, aber vor allem deutlich stabiler. Das Histogramm der Messungen in Abbildung 2 deckt sich deutlich besser mit der logarithmischen Normalverteilung und hat weniger Ausreißer mit sehr kleiner Latenz, aber auch deutlich weniger Ausreißer mit viel höherer Latenz. Hier spielen vermutlich auch die Überlegungen aus Abschnitt (c) hinein, erkärt aber sicherlich nicht die sehr große Varianz, die hier bei der Messung von `threads.cpp` zustande kam.

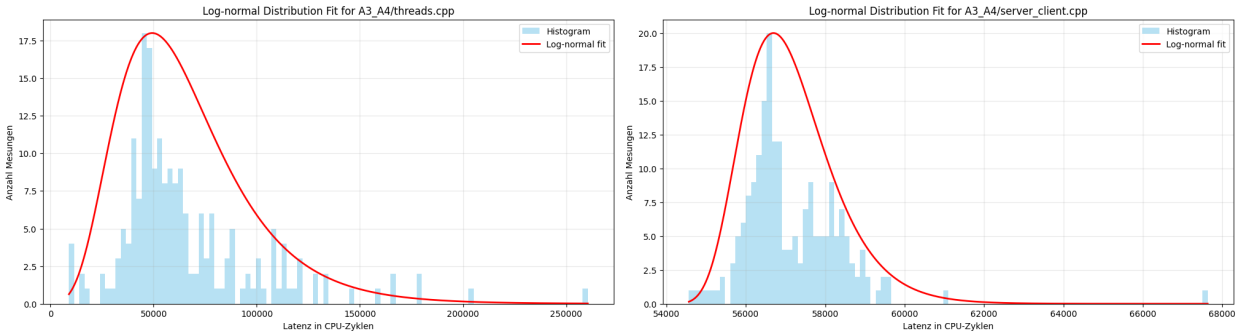

Abbildung 2: Vergleich der Latenzen der Kommunikation zweier Threads im selben Prozess (`inproc`) links ggü. zwei separaten Prozessen (`ipc`) `rechts.`

Aufgabe 4

Docker erlaubt es, Programme in unterschiedlichen Containern zu starten, welche dann über *TCP/IP Sockets* oder *Shared Memory* miteinander kommunizieren können. Als Basis setze ich hier auf die ZeroMQ Lösungen aus Aufgabe 3, da sie flexibel beide Protokolle unterstützt und ohne eine Veränderung eingesetzt werden kann. Zudem bietet dieser Ansatz die Möglichkeit, ziemlich genau die zusätzliche Latenz, welche durch die OS-Level Virtualisierung durch Docker zustande kommt, zu messen.

In den folgenden Messungen wurden zwei *Ubuntu:latest* Container verwendet, welche mittels eines *TCP*-Ports miteinander kommunizieren konnten. Die Kommunikation mittels *Shared Memory* konnte ich leider nicht ausprobieren, da mir trotz `ipc=host` keine funktionierende Konfiguration geglückt ist.

Ergebnisse

Implementierung	Mittelwert	Untere Grenze	Obere Grenze	1% Quantil	99% Quantil
A3_A4/tcp_docker	80187.21	78836.24	81561.33	78796.68	80468.54

Tabelle 4: Gemessene Metriken und Konfidenzintervalle für meine ZeroMQ-Implementierungen in separaten Docker-Containern in CPU-Zyklen: Mittelwert, untere und obere Grenze des 95% Konfidenzintervalls, sowie das 1% und 99% Quantil meiner Messungen.

Wie zu erwarten ist die Kommunikationslatenz über Docker Container hinweg sehr nahe an der *bare metal* Messung. Ein entsprechendes Paper [1] fand bereits 2014 keinen nennenswerten Unterschied der Performance und durch die kontinuierliche Verbesserung der Docker-Runtime und des Linux Kernels als Open Source Projekte wurde sie vermutlich noch weiter optimiert.
Interessanterweise ist die Latenz zwischen den beiden *Ubuntu:latest* Containern sogar noch etwa 5% geringer als bei der *bare metal* Messung. Dies liegt vermutlich daran, dass das Ubuntu im Docker Container etwas optimaler für diesen Anwendungsfall (ZeroMQ + TCP) konfiguriert ist, als Ubuntu LiveOS, wie es standardmäßig in der aktuellsten ISO vorzufinden ist.

Literatur

[1] Felter, W., Ferreira, A., Rajamony, R., and Rubio, J. An updated performance comparison of virtual machines and linux containers. In 2015 IEEE international symposium on performance analysis of systems and software (ISPASS) (2015), IEEE, pp. 171–172.

[2] Hintjens, P. ZeroMQ. O'Reilly Media, Sebastopol, CA, Mar. 2013.

[3] Taylor, B. The Second Law of Latency: Latency distributions are NEVER NORMAL — medium.com. `https://medium.com/engineers-optimizely/laws-of-latency-latency-distributions-are-never-normal-a368c1624340`, 2022. [Accessed 11-01-2025].

[4] Treat, T. Everything You Know About Latency Is Wrong — bravenewgeek.com. `https://bravenewgeek.com/everything-you-know-about-latency-is-wrong/`, 2015. [Accessed 11-01-2025].

Implementierung	Mittelwert	Untere Grenze	Obere Grenze	1% Quantil	99% Quantil
A1/accurate.cpp	22.50	22.48	22.53	22.06	22.79
A1/accurate_cpu_aff.cpp	44.03	42.74	45.35	36.57	84.20
A1/shared_mem.cpp	20.58	20.53	20.64	20.22	20.96
A2/semaphore.cpp	123.36	118.11	128.84	28.12	209.74
A3_A4/threads.cpp	10731.79	9994.14	11523.88	1796.95	32880.65
A3_A4/server_client.cpp	10504.70	10472.57	10536.92	10088.68	10950.77
A3_A4/server_client_tcp.cpp	15576.01	14651.75	16558.57	14926.97	15231.61
A3_A4/docker	14740.30	14491.96	14992.89	14484.68	14792.01

Tabelle 5: Approximierte Latenzen aus Tabellen 1, 2, 3, 4 in ns mit einer mittleren gemessenen Taktrate von 5.44 GHz