

Mechanical Sympathy

Mechanical Sympathy ist ein Begriff, den Martin Thompson ursprünglich vom Rennfahrer Jackie Stewart entlehnt hat, der sagte, dass die Maschine verstanden werden muss, um in Harmonie mit ihr zu arbeiten. Übertragen auf die Softwareentwicklung heißt das: Guter Code versteht die Hardware, auf der dieser läuft und orientiert sich an ihren physikalischen Grenzen.

Zum Beispiel dauert ein Zugriff auf den Arbeitsspeicher etwa hundert Nanosekunden, während der Zugriff auf den CPU-Cache nur 0.5-7 Nanosekunden benötigt. Der Weg über das Netzwerk ist schnell mehrere Größenordnungen langsamer und über mehrere Availability Zones hinweg fällt das noch einmal deutlich stärker ins Gewicht [1]. Wer diese Zahlen im Kopf hat, trifft beim Entwurf eines Systems von vornherein bessere Entscheidungen.

Aus dem Verständnis der physikalischen Grenzen der Hardware folgt eine wichtige Erkenntnis: Ein Hochleistungssystem muss nicht zwangsläufig auf viele Maschinen verteilt sein. Wenn geringe und stabile Latenzen gefordert sind, muss auf Netzwerkverkehr verzichtet werden und eine einzige Maschine bis an die Grenzen der Hardware ausgereizt werden. Um dabei einen möglichst hohen Durchsatz zu erreichen, wird die Arbeit auf mehrere Threads verteilt. Der Datenaustausch und die Synchronisation dieser Threads kann, je nach gewählter Methode, starke Performance Unterschiede verursachen.

Kosten der Synchronisation

Gerade bei hohem Datendurchsatz und gleichzeitigen Zugriffen führen notwendige Synchronisationsmechanismen wie synchronized, Compare-and-Swap (CAS) oder volatile zu Performanceeinbußen. Threads müssen aufeinander oder auf Netzwerk- bzw. Festplattenoperationen warten, was zu Kontextwechseln in der CPU führt. Diese Wechsel sind teuer, da sie die Cache-Lokalität verschlechtern und zusätzliche Speicherzugriffe erforderlich machen.

Thompson et al. haben im Paper zum Disruptor [2] beschrieben, wie teuer diese Synchronisation sein kann. Die Tabelle 1 zeigt die Messergebnisse eines minimalen Experiments.

Tabelle 1: Synchronisationskosten [2]

Methode	Zeit in ms
1 Thread	300
1 Thread und Lock	10.000
2 Threads und Lock	224.000
1 Thread und CAS	5.700
2 Threads und CAS	30.000
1 Thread und volatile	4.700

Eine weitere Quelle von Verzögerungen ist das sogenannte *Cache-Line-Bouncing*, bei dem Schreibzugriffe mehrerer Kerne auf dieselbe Cache-Zeile dazu führen, dass die Zeile zwischen den Kernen hin- und herwandert und dabei in den jeweils anderen Caches invalidiert wird [3]. Dieses Phänomen kann auch auftreten, wenn zwei Threads in unterschiedliche Variablen schreiben, die sich aber zufällig in derselben Cache-Zeile befinden. Um dieses *False Sharing* zu vermeiden, muss auf die Positionierung im Speicher geachtet werden und Variablen eventuell, umgeben von ungenutzten Bytes, in eine eigene Cache-Zeile gelegt werden.

Mit einem Blick für *Mechanical Sympathy* wurde aus diesen Erkenntnissen ein System gebaut, was sich die Stärken der unterliegenden Maschine zu Nutzen macht, die verschiedenen Cache Level effizient nutzt und teure Operationen vermeidet: der *LMAX Disruptor*.

Disruptor

Der Disruptor ist eine Java-Bibliothek die aus Produzenten, Konsumenten, einem *Ringbuffer* und einem *Sequencer* besteht. Der *Ringbuffer* bildet die zentrale Datenstruktur, in die der Produzent Daten speichert und die Konsumenten diese lesen und verarbeiten. Er wird bei Programmstart in einem fortlaufenden Speicherbereich allokiert, damit der Zugriff effektiv gecached werden kann. Der Lese- und Schreibzugriff ist ohne Locks oder das synchronized-Schlüsselwort in Java implementiert. Im Folgenden wird der Mechanismus erklärt.

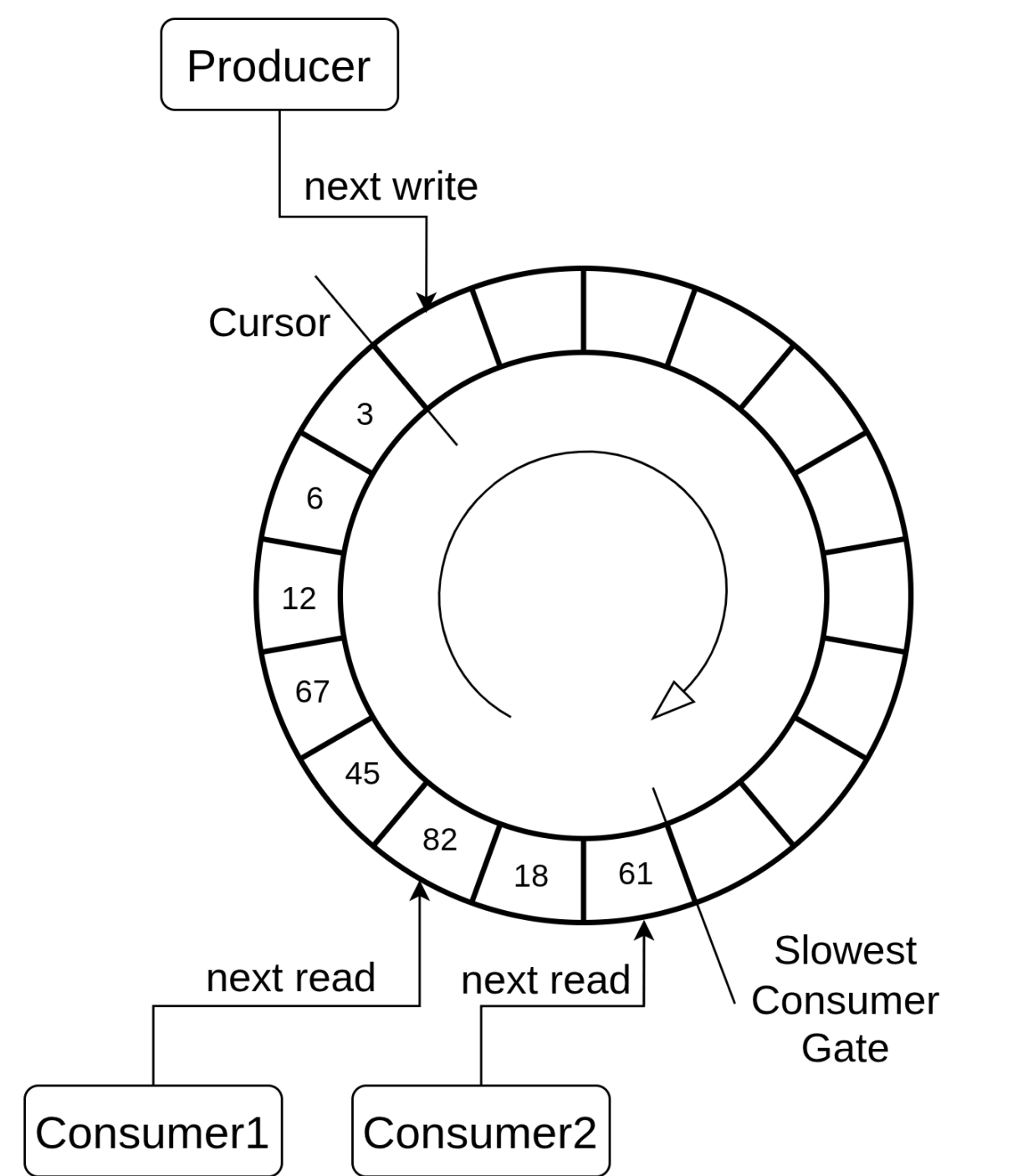


Abbildung 1: Ringbuffer

Um die Datenkorrektheit zu garantieren, müssen zwei Voraussetzungen erfüllt werden:

1. Konsumenten dürfen nur soweit lesen, wie valide Daten vorhanden sind. **Konsumenten dürfen nicht überholen.**
2. Produzenten dürfen keine Felder überschreiben, die noch nicht gelesen wurden. **Produzenten dürfen nicht übrerrunden.**

Die Logik, die diese Bedingungen umsetzt, ist im *Sequencer* abgebildet. Während der Ringbuffer die Daten hält, regelt der Sequencer, welche Slots Produzenten und Konsumenten nutzen dürfen. Zentral für die Koordination sind die *Sequenzen*, die als monoton wachsende Zähler den Fortschritt von Produzenten und Konsumenten beschreiben. Die letzte beschriebene Sequenz ist der Cursor, bis zu dem die Konsumenten lesen dürfen. Die Sequenz des langsamsten Konsumenten bestimmt wie weit der Produzent schreiben darf. So wird sichergestellt, dass Speicherzugriffe in der korrekten Reihenfolge erfolgen, ohne dass Synchronisationsmechanismen vom Betriebssystem notwendig sind. Jede Sequenz gehört zu einem Thread und darf nur von diesem erhöht werden. Das verhindert das Cache-Line-Bouncing durch abwechselndes Schreiben und ist die grundlegende Designentscheidung des Disruptors.

Damit ist das Bild aber noch nicht vollständig, denn Hardware bringt eigene Tücken mit. Ein Prozessor lädt den Speicher stets in Blöcken von 64 Byte, den sogenannten Cache-Lines, sodass eng beieinander liegende Werte unweigerlich gemeinsam im Cache landen. Teilen sich zwei Sequenzen dieselbe Cache-Line genügt bereits die Änderung der einen, um die zwischengespeicherte Kopie der anderen zu invalidieren. Dieses, als *False Sharing* bezeichnete Phänomen, frisst einen Großteil des Vorteils wieder auf, den der Lock-freie Zugriff eigentlich verspricht. Der Disruptor begegnet dem, indem er jede Sequenz gezielt mit ungenutztem Speicher umgibt, sodass sie eine Cache-Line vollständig ausfüllt und kein zweiter Zähler mehr in denselben Block fällt. Auf diese Weise kann das Schreiben einer Sequenz die Caches der übrigen Threads nicht länger stören, und die Kerne arbeiten nahezu ungestört nebeneinander [2].

Weil die Sequenzen unaufhörlich wachsen und niemals zurückgesetzt werden, drängt sich die Frage auf, ob sie nicht irgendwann überlaufen. Eine Sequenz wird als vorzeichenbehafteter 64-Bit-Ganzzahlwert geführt und reicht damit bis in die Größenordnung von rund $9,2 \cdot 10^{18}$. Bei einem im Benchmark gemessenen Durchsatz von 11,2 Millionen Events pro Sekunde wäre dieser Wertebereich erst nach etwa 26.000 Jahren erschöpft. Ein Überlauf ist damit zwar theoretisch möglich, praktisch aber irrelevant.

Offen bleibt schließlich, wo der Ringbuffer seine Daten vorhält. Wir haben dazu zwei Varianten gegenübergestellt, eine On-Heap-Variante im verwalteten Speicher der JVM und eine Off-Heap-Variante in einem nativen Speicherbereich außerhalb des Garbage Collectors. Ein nennenswerter Leistungsunterschied zeigte sich dabei allerdings nicht, denn schon die On-Heap-Variante vermeidet den Garbage Collector durch die vollständige Vorallokierung des Ringbuffers, sodass in beiden Fällen während der Verarbeitung keine Objekte mehr erzeugt oder abgeräumt werden müssen.

Anwendungsgebiete

Der Disruptor wurde für den Bereich des Hochfrequenzhandels entwickelt und wird beispielsweise bei LMAX Exchange und Garanti BBVA eingesetzt, um eine hohe Anzahl von Ereignissen wie Marktpreisänderungen oder Kaufaufträge mit möglichst niedriger und stabiler Latenz zu verarbeiten. Abbildung 2 verdeutlicht die Architektur so eines Systems, bei dem Produzenten und Konsumenten als eigene Threads agieren und mehrere Disruptoren als synchronisierende Datenstruktur verwendet werden.

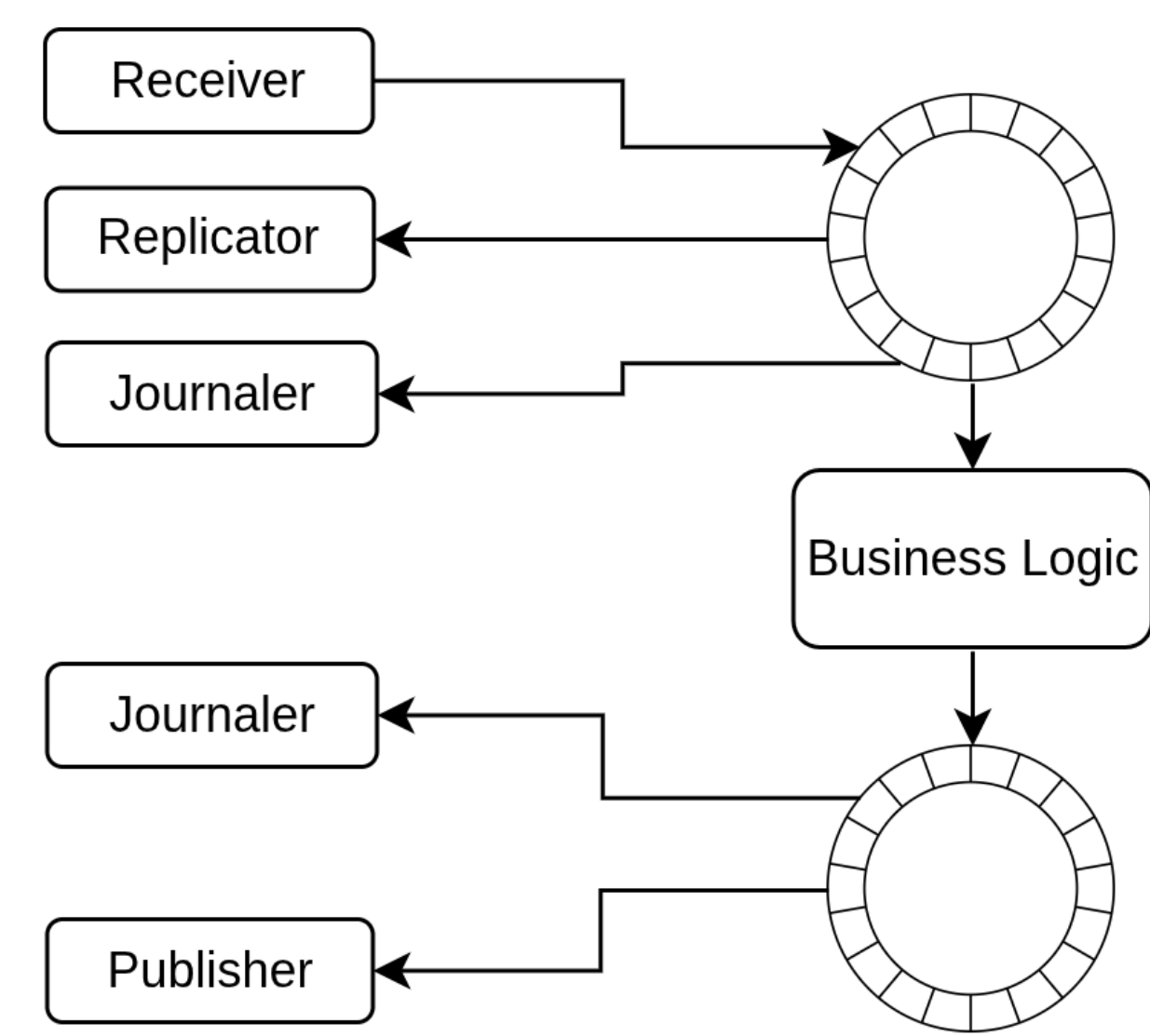


Abbildung 2: Zusammenspiel von Threads und Disruptor

Darüber hinaus können auch andere Echtzeitanwendungen, wie z. B. Game Server oder Digital Audio Workstations, von einer Architektur wie dem Disruptor profitieren. Wo viele Ereignisse mit möglichst niedriger und stabiler Latenz verarbeitet werden müssen, können Lock-freie, cache-effiziente Ringbuffer-Strukturen dabei helfen, Synchronisationskosten zu reduzieren und die Reaktionszeit vorhersehbar zu halten.

Der Fokus auf Performanz und effiziente Nutzung der Hardware lässt sich auch aus einer Nachhaltigkeitsperspektive nutzen. Gegenüber verteilten Systemen, die ihren zusätzlichen Rechenaufwand durch mehr Hardware ausgleichen, können lokale Systeme die Energiekosten und Hardwareanforderungen reduzieren. Martin Thompsons *Mechanical Sympathy* und nachhaltige Softwareentwicklung haben zwar eine andere Motivation, doch die Ziele und Werkzeuge sind ähnlich genug um voneinander zu lernen.

Benchmarks

Für die Benchmarks wurde ein Lenovo ThinkPad P14s Gen 3 mit einer Intel Core i7-1260P CPU und 32GB RAM verwendet [4]. Der Prozessor verfügt über vier Performance-Kerne mit Hyperthreading und acht Effizienzkerne. Als Betriebssystem wurde Debian Trixie in einer minimalen Version ohne GUI installiert. Für die Java Virtual Machine wurde OpenJDK 25.0.5 verwendet.

Um ungewollte Einflüsse auf die Benchmarks zu vermeiden, wurde das System so eingestellt, dass die Kerne 0-3 isoliert werden. Dadurch verwendet das Betriebssystem diese Kerne nicht für eigene Aufgaben, und es werden keine Interrupts auf ihnen ausgeführt. So wird verhindert, dass Hintergrundprozesse oder Systemaktivitäten die Messung beeinflussen und die Ergebnisse verfälschen. Der CPU-Governor wurde vor dem

Start der Benchmarks auf *performance* gestellt. Das Benchmarkprogramm weist sich selbst mithilfe der Bibliothek Java-Thread-Affinity die Kerne zu, welche nicht durch das Betriebssystem verwendet werden, und pinnt die einzelnen Threads auf jeweils einen dieser Kerne. So wird sichergestellt, dass die Threads nicht zwischen den Kernen wechseln und die Cache-Lokalität erhalten bleibt. Da der Prozessor auf den Performance-Kernen Hyperthreading unterstützt, teilen sich die Kerne 0 und 1 sowie die Kerne 2 und 3 jeweils eine physische CPU und damit auch den Level-1- und Level-2-Cache.

Vor der eigentlichen Messung wurde eine Warm-up-Phase durchgeführt. Diese dient dazu, das System unter Last thermisch zu stabilisieren und die CPU in einen reproduzierbaren Betriebszustand zu bringen. Da bei der verwendeten Konfiguration nur wenige Kerne voll ausgelastet werden, kann der Prozessor den hohen Turbotakt unter günstigen thermischen Bedingungen länger halten. Erst danach beginnt die eigentliche Messung, damit die gemessenen Latenzen und der Durchsatz nicht durch das anfängliche Hochregeln von Takt und Temperatur beeinflusst werden [5].

Der Producer erzeugt so schnell wie möglich neue Events. Diese Events enthalten ausschließlich einen Zeitstempel in Nanosekunden. Der erste Konsument macht während des Benchmarks nichts, als einmal pro Event einen Spin-Wait durchzuführen. Der zweite Konsument liest den Zeitstempel aus dem Event, berechnet die Latenz und speichert diesen Wert zur späteren Auswertung in einem Histogramm.

Die Auswertung erfolgt anschließend auf Basis der gesammelten Latenzwerte. Dafür werden die Messwerte sortiert beziehungsweise in eine Verteilung überführt und daraus die gewünschten Perzentile bestimmt. Das P10 beschreibt dabei den Wert, unter dem 10% aller gemessenen Latenzen liegen. Entsprechend gibt das P50 den Median an, also den mittleren Wert der Verteilung. Mit P90, P99 und P999 lassen sich darüber hinaus auch die Randbereiche der Verteilung betrachten, sodass nicht nur ein typischer Wert, sondern auch das Verhalten bei schlechteren Fällen sichtbar wird. Dadurch lässt sich die Performance des Systems nicht nur über einen einzelnen Durchschnittswert beurteilen, sondern auch über die Stabilität der Messwerte. [6]

Mit unserer Implementation konnten wir die Zahlen von Thompson et al. [2] bestätigen. In Tabelle 2 sind Latenzen und Durchsatz einer 1 Producer - 2 Consumer Anwendung dargestellt. Die Messung zeigt dabei, dass die meisten Events mit sehr geringer Latenz verarbeitet werden und nur ein kleiner Teil der Messwerte in höhere Bereiche fällt. Der Durchschnittswert liegt bei 191 ns, während das P10 bereits bei 111 ns liegt und das P999 bei 303 ns. Dadurch wird sichtbar, dass die Verteilung der Latenzen trotz des hohen Durchsatzes sehr stabil bleibt.

Tabelle 2: Latenzen und Durchsatz	
Latenz Ø	191 ns
Latenz P10	111 ns
Latenz P50	143 ns
Latenz P90	207 ns
Latenz P99	239 ns
Latenz P999	303 ns
Durchsatz	11.200.000/s

Vergleich zur Warteschlange

Als Alternative zum Disruptor wurde eine einfache Implementierung mit einer Warteschlange und ohne Cache Optimierung den gleichen Benchmarks unterzogen.

Die Ergebnisse, dargestellt in Abbildung 3, zeigen, dass diese Implementierung deutlich höhere Latenzen mit sich bringt. Besonders fällt die Varianz der Latenz auf: Während 10% der Ereignisse noch innerhalb 500 Nanosekunden verarbeitet werden können, brauchen die langsamsten 0.1% mehrere Millisekunden, ca. ein Faktor von 6.000. Das zeigt, dass eine einfache Warteschlangenlösung zwar funktional ist, bei hoher Parallelität und niedriger Latenz aber deutlich an ihre Grenzen stößt. Die höhere Streuung der Latenzen zeigt, dass der Warteschlangenansatz unter Last weniger stabil ist. Der Durchsatz ist mit 950.000 Ereignissen pro Sekunde fast um den Faktor 10 schlechter als mit dem Disruptor.

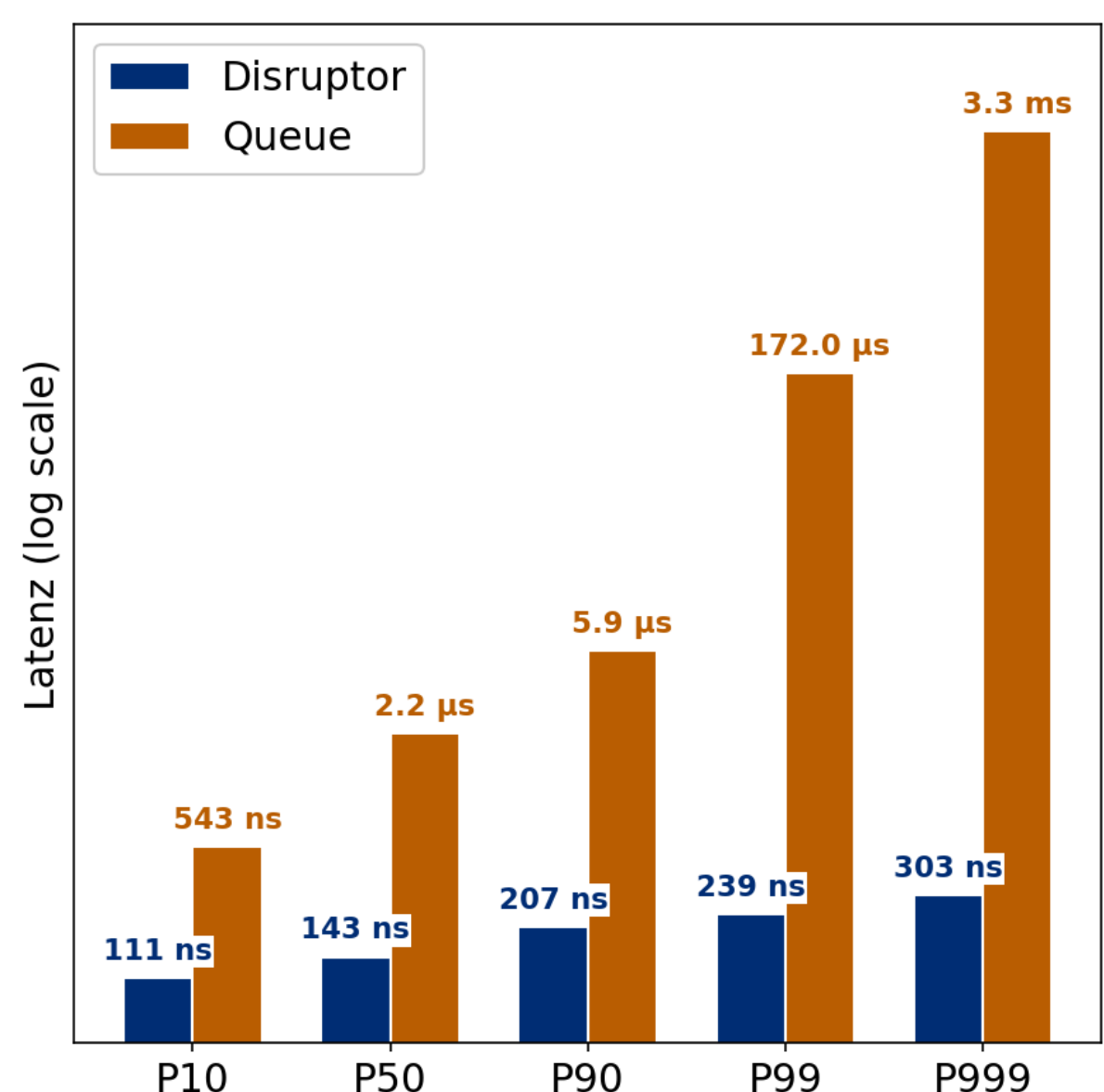


Abbildung 3: Latenzen

Wie schon Thompson et al. [2] anmerken, sind diese Messungen und Erkenntnisse immer von der unterliegenden Hardware und der speziellen Anwendung abhängig und sollten daher auf anderen Systemen neu überprüft werden.

Referenzen

- [1] J. Bonér. „Latency Numbers Every Programmer Should Know.“ Adresse: <http://gist.github.com/jboner/2841832>.
- [2] M. Thompson, D. Farley, M. Barker, P. Gee und A. Stewart, „Disruptor: High Performance Alternative to Bounded Queues for Exchanging Data Between Concurrent Threads,“ LMAX Group, Technical Report, Mai 2011. besucht am 30. Juni 2026. Adresse: <https://lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf>.
- [3] Gernot Heiser, UNSW. „Multiprocessor Operating Systems – Lecture 10: Multiprocessor/Multicore (slides),“ besucht am 28. Juni 2026. Adresse: <https://cgi.cse.unsw.edu.au/~cs9242/12/lectures/10-multiproc-4up.pdf>.
- [4] Intel. „Intel® Core™ i7-1260P Processor (18M Cache, up to 4.70 GHz),“ besucht am 13. Juli 2026. Adresse: <https://www.intel.com/content/www/us/en/products/sku/226254/intel-core-i71260p-processor-18m-cache-up-to-4-70-ghz/specifications.html>.
- [5] Intel. „Intel Turbo Boost Technology 2.0 und Package Power Control,“ besucht am 13. Juli 2026. Adresse: <https://edc.intel.com/content/www/us/en/design/ipia/software-development-platforms/client/platforms/alder-lake-desktop/12th-generation-intel-core-processors-datasheet-volume-1-of-2/003/package-power-control/>.
- [6] M. Kleppmann und C. Riccomini, *Designing Data-Intensive Applications, The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*, 2. Aufl. O'Reilly Media, Inc., 2026.