UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Test und Beurteilung der Echtzeitfähigkeit des Linux-Kernels mit und ohne Preemptive Patch

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Informatik

vorgelegt von
Felix Kloft

Erstgutachter:	Prof. Dr. Dieter Zöbel AG Echtzeitsysteme
Zweitgutachter:	Dipl.-Inform. Andreas Stahlhofen AG Echtzeitsysteme

Koblenz, im Juni 2017

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Abstract

While real-time applications used to be executed on highly specialized hardware and individually developed operation systems, nowadays more often regular off-the-shelf hardware is used, with a variation of the Linux kernel running on top.

Within the scope of this thesis, test methods have been developed and implemented as a real-time application to measure several performance properties of the Linux kernel with regards to its real-time capability.

These tests have been run against three different versions of the Linux kernel. Afterwards, the results of the test series were compared to each other.

Zusammenfassung

Wurden Echtzeitanwendungen früher meist auf hochspezialisierter Hardware und mit eigens dafür entwickelten Betriebssystemen ausgeführt, so wird heutzutage oft auch handelsübliche Hardware eingesetzt, auf dem eine Variante des Linux-Kernels läuft.

Im Rahmen dieser Arbeit wurden Testverfahren aufgestellt und in Form einer Echtzeitanwendung implementiert, welche verschiedene Leistungsmerkmale des Linux-Kernels im Hinblick auf dessen Echtzeitfähigkeit messen.

Diese Tests wurden auf drei verschiedenen Kernel-Versionen ausgeführt. Anschließend wurden die Ergebnisse der Testreihen miteinander verglichen.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

1.1. Motivation

Wenngleich man den Begriff „Echtzeitsystem“ im Alltag praktisch nie und auch im IT-Sektor nur in entsprechend spezialisierten Bereichen hört, so sind solche Systeme doch weiter verbreitet als es zunächst den Anschein hat. Echtzeitsysteme kommen nicht nur in der Industrie zur Steuerung komplexer Vorgänge zum Einsatz, auch im Alltag hat man meist unbewusst Kontakt zu solchen Systemen. Diese finden sich meist in Form von eingebetteten Systemen, d. h. eingebaut in Geräte, denen man auf Anhieb nicht ansieht, dass sie einen Computer enthalten. Ein häufig genanntes Beispiel und womöglich eines der bedeutendsten im Alltag ist das Auto, welches meist nicht nur einen, sondern gleich mehrere Computer zur Steuerung von Motor, Antrieb und Bremse enthält. Diese Systeme sorgen bei normalem Betrieb für ein optimales Fahrgefühl und stellen nicht nur Komfortfunktionen wie Radio, Telefonie und Navigation bereit, sondern schützen auch in Ausnahmesituationen mit Sicherheitsfunktionen wie Airbags, ABS und ESP.

In der Vergangenheit wurden häufig spezialisierte Hard- und Software zur Bereitstellung von Echtzeitsystemen eingesetzt. Diese erlaubt alle Abläufe genauestens zu kontrollieren, so dass keine unvorhergesehenen Bedingungen die Echtzeitfähigkeit beeinträchtigen können. Damit verbunden sind jedoch hohe Kosten für Entwicklung und Wartung sowie eine gewisse Fehleranfälligkeit, da die Systeme nur von einem kleinen Nutzerkreis eingesetzt und getestet werden. Daher ist man zunehmend dazu übergegangen, gängige Computer-Hardware und Betriebssysteme für den Gebrauch in Echtzeitumgebungen anzupassen.[1] Diese enthalten jedoch Bestandteile, die für den allgemeinen Gebrauch nützlich, in Echtzeitumgebungen jedoch unnötig oder gar störend sind. Falls sich diese Bestandteile nicht entfernen lassen, muss jedoch sichergestellt werden, dass ihre Auswirkungen die Echtzeitfähigkeit nicht gefährden.

1.2. Zielsetzung

Wurde der Linux-Kernel zunächst fast ausschließlich auf Server-Systemen eingesetzt, ist er heutzutage aufgrund seiner Flexibilität in den verschiedensten Geräteklassen von Desktop-Rechnern über Mobilgeräte wie Smartphones und Tabletts bis hin zu eingebetteten Geräten zu finden. Gerade der letztgenannte Bereich hat mit dem Aufkommen von Hardware-Plattformen wie dem Raspberry Pi und des „*Internet of Things*“, also der Vernetzung von Alltagsgegenständen über das Internet, ein starkes Wachstum erlebt. Diese Flexibilität sorgte dafür, dass auch Linux-Varianten für den Einsatz als Echtzeitbetriebssystem entwickelt wurden.

Ziel dieser Arbeit ist es, diese Echtzeitfähigkeit zu messen und auszuwerten. Dazu werden zunächst einige Parameter bestimmt, durch welche diese Echtzeitfähigkeit typischerweise eingeschränkt wird. Anschließend werden Testverfahren entwickelt, um diese Parameter zu messen. Zuletzt wird durch Analyse und Auswertung der Messergebnisse eine Aussage über die Echtzeitfähigkeit der Linux-Kernels erstellt.

1.3. Stand der Technik

Es gibt nur wenig Studien, welche die Echtzeitfähigkeiten des Linux-Kernels statistisch untersuchen, z. B. [2]. Diese sind jedoch häufig veraltet, da sich der Kernel in den letzten Jahren stark entwickelt hat. Andere Studien wie [3] untersuchen die Latenz mithilfe von externer Hardware. In dieser Arbeit sollen solche Eigenschaften dagegen aus Sicht einer Anwendung ohne externe Werkzeuge oder Eingriffe in den Kernel-Quelltext untersucht werden.

Im Zusammenhang mit dem PREEMPT_RT-Projekt (siehe Abschnitt 2.3) wurden bereits verschiedene Werkzeuge zum Messen und Testen des Kernels erstellt und unter dem Namen *rt-tests* gesammelt. In Studien über dessen Echtzeitfähigkeiten wie zum Beispiel [4] wird allerdings häufig nur das Programm *cyclictest* benutzt, obwohl eine Vielzahl weiterer Werkzeuge vorhanden sind.

2. Grundlagen

2.1. Echtzeitsysteme

Ein Echtzeitsystem ist ein System, bestehend aus Hardware und Software, das gewisse Garantien erfüllt gegenüber Anwendungen, die auf diesem System ausgeführt werden. Darunter gehört unter anderem die Existenz gewisser Leistungsmerkmale, die das System bereitstellt, wie etwa eine Form von Anwendungspriorisierung und Funktionen zur Synchronisierung mehrerer Tasks. Außerdem sollte das System gewisse zeitliche Grenzen setzen, wie zum Beispiel die maximale Ausführungszeit für Systemaufrufe, die größtmögliche Verzögerung vom Eintreten eines Ereignisses wie etwa eines Interrupts bis zur Ausführung der entsprechenden Behandlungsroutine, sowie die minimale Dauer eines Kontextwechsels. Liegen gleichzeitig mehrere zu erledigende Aufgaben für das System vor, so muss es einen determinierten Ablauf geben, damit sich für jede Aufgabe bestimmen lässt, wie lange sie maximal für die Abarbeitung anderer Aufgaben zurückgestellt wird.

Ein echtzeitfähiges System wird oft fälschlicherweise darauf reduziert, Prozesse möglichst schnell zu verarbeiten. Tatsächlich wird bei Echtzeitsystemen oft ein Geschwindigkeitsverlust in Kauf genommen, um eine deterministische Verarbeitung zu gewährleisten. Bei vielen Echtzeitsystemen wird die Reaktionszeit des Systems üblicherweise in Milli-, Mikro- oder Nanosekunden gemessen, allerdings kann je nach Anwendungsfall auch eine Rechenzeit in der Größenordnung von Stunden oder Tagen normal sein. Das Entscheidende bei Echtzeitsystemen ist, dass Aufgaben innerhalb einer garantierten, berechenbaren Zeit verarbeitet werden. Dies erfolgt z. B. durch Angabe einer Frist, innerhalb derer eine Aufgabe vollständig verarbeitet sein muss.

Um die Echtzeitfähigkeit des Systems zu beschreiben müssen alle Komponenten des Systems betrachtet werden. Dies beinhaltet insbesondere die Hardware, das Betriebssystem, die Treiber, sowie die eigentlichen Echtzeitanwendungen. Eine nicht echtzeitfähige Komponente kann dabei die Echtzeitfähigkeit des gesamten Systems zerstören oder zumindest beeinträchtigen.

Es wird zwischen harter und weicher Echtzeit unterschieden. Bei harter Echtzeit darf zu keinem Zeitpunkt eine Frist verletzt werden, andernfalls gilt das gesamte System als fehlerhaft. Diese Form wird häufig gefordert, wenn die Ergebnisse nach Ablauf der Frist nutzlos sind oder es durch die Fristverletzung zu erheblichem Schaden kommt. Harte Echtzeitsysteme kommen häufig in Sicherheitseinrichtungen wie zum Beispiel KFZ-Steuerungssystemen zum Einsatz.

Weiche Echtzeit dagegen toleriert sporadische Fristverletzungen, solange diese nicht zu häufig eintreten oder bei wiederholter Ausführung eine folgende Instanz der Aufgabe entsprechend schneller bearbeitet werden kann. Je häufiger eine Echtzeit-Garantie verletzt wird, desto geringer ist die Qualität des Systems; allerdings kann es je nach Einsatzgebiet durchaus noch als brauchbar angesehen werden. Diese Art von Echtzeit wird häufig in der Medien-Verarbeitung wie Telefon und Video-Streaming angewendet, wo eine Fristverletzung lediglich zu einem kurzen Aussetzer im Bild- oder Audio-Stream führt.

2.2. Linux-Scheduler

Da diese Arbeit sich auf den Linux-Kernel bezieht, soll zunächst ein Blick auf dessen Scheduler geworfen werden. Der Scheduler ist die Komponente des Kernels, welche festlegt, welcher Prozess gerade Rechenzeit erhält und diesen einem Prozessor zuordnet. Der Linux-Scheduler ist weitgehend darauf ausgelegt, einzelne Threads anstelle von ganzen Prozessen zu verwalten. Ein Prozess wird interpretiert als eine Gruppe von Threads, die sich eine sogenannte *Thread Group ID* (*TGID*) [5] teilen. Diese repräsentiert, was außerhalb des Schedulers allgemein als *Process ID* (*PID*) bezeichnet wird. Innerhalb des Schedulers werden Threads auch als *Task* bezeichnet. Der Scheduler unterstützt präemptives Multitasking, d. h. Tasks können vom Kernel unterbrochen, von anderen Tasks verdrängt und zu einem späteren Zeitpunkt wieder aufgenommen werden. Dabei ist keine Kooperation von Seiten des Tasks nötig; die Unterbrechung (*Preemption*) erfolgt in der Regel als Reaktion auf eine Unterbrechungsanforderung (*Interrupt Request*). Ein Interrupt kann beispielsweise ausgelöst werden durch ein externes Ereignis wie einen Tastendruck, softwareseitig durch den laufenden Task oder durch einen Zeitgeber zu einem vorher vom Prozessor definierten Zeitpunkt.

Der Linux-Scheduler besteht aus mehreren Komponenten: dem Scheduling-Framework sowie mehreren Scheduling-Klassen. Das Scheduling-Framework plant selbst keine Tasks, es stellt lediglich die dazu nötigen Funktionen bereit. Für jeden Task existiert im Speicher eine Datenstruktur namens *task_struct*, welche den Kontext und aktuellen Status eines Tasks speichert. Hier wird jedem Task unter anderem neben der Prozess- und Thread-ID auch genau eine Scheduling-Klasse zugeordnet.

Der Scheduler enthält vier Scheduling-Klassen, von denen zwei für interne Verwaltungszwecke verwendet werden und zwei für Anwendungen zur Verfügung stehen (siehe Abbildung 1).

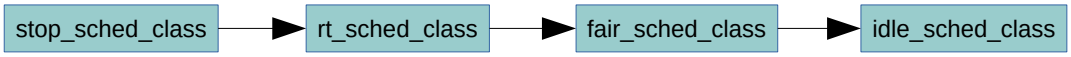

Abbildung 1: Scheduling-Klassen im Linux-Kernel

rt_sched_class implementiert dabei weiches Echtzeitscheduling, *fair_sched_class* den *Completely Fair Scheduler* (CFS), den Standardscheduler. Wenn das Scheduling-Framework den nächsten Task zur Ausführung auf dem Prozessor aussucht, werden die verschiedenen Klassen nacheinander zurate gezogen, bis ein Task ausgewählt wurde. Auf diese Weise werden Echtzeittasks immer bevorzugt gegenüber normalen Tasks eingeplant.

Innerhalb der Scheduling-Klassen werden die Tasks weiter durch Prioritäten unterschieden. Während der Standardscheduler darauf ausgelegt ist auch niedrig priorisierten Tasks irgendwann Rechenzeit einzuräumen, führt der Echtzeitscheduler immer denjenigen rechenbereiten Task aus, welchem die höchste Priorität zugeordnet wurde.

2.3. PREEMPT_RT

Seit einer Reimplementierung des Schedulers und der Einführung der Schedulingklassen in Linux 2.2 [6] unterstützt der Kernel natives Soft-Echtzeitscheduling. Der PREEMPT_RT-Patch [7] dagegen zielt darauf ab, den Kernel auch hart echtzeitfähig zu machen. Einige Änderungen wie etwa der *High Resolution Timer* [8] aus dem PREEMPT_RT-Projekt wurden im Laufe der Zeit auch in den Hauptzweig des Linux-Kernels integriert. Die wesentlichen Änderungen des Patches bestehen dabei aus den folgenden Punkten:

- Ersetzen von *spinlock_t* durch eine verdrängbare Variante: In einem normalen Linux-Kernel kann ein Kernel-Thread eine CPU vollständig blockieren, bis eine gesperrte Ressource verfügbar wird. Der Thread kann in dieser Zeit nicht unterbrochen oder verdrängt werden. Mit PREEMPT_RT sind auch diese Threads unterbrechbar.
- Implementierung von Prioritätsvererbung für Kernel-Threads: Es kann vorkommen, dass ein hoch priorisierter Prozess blockiert wird durch eine Ressource, welche gerade einem niedriger priorisiertem Prozess zugeordnet ist. Der Kernel kann dem zweiten Prozess die Ressource allerdings nicht einfach entziehen. Durch Prioritätsvererbung erhält dieser stattdessen vorübergehend die Priorität des ersten Prozesses. So wird eine möglichst schnelle Freigabe der Ressource erreicht.
- Umwandlung von Unterbrechungsroutinen (*interrupt handler/interrupt service routine)* durch Kernel-Threads: Unterbrechungsroutinen laufen nicht innerhalb eines Thread-Kontexts und lassen sich daher nicht unterbrechen oder zurückstellen, verdrängen ihrerseits allerdings alle anderen Threads. Eine komplexere Unterbrechungsroutine kann daher auch hoch priorisierte Threads um einen beträchtlichen Zeitraum verzögern. Um die Echtzeitfähigkeit des Systems zu gewährleisten werden in PREEMPT_RT diese Ro*bottom half*utinen daher in zwei Teile gespalten. Die sogenannte untere Hälfte („*bottom half*“) läuft im Kontext eines Kernel-Threads und lässt sich wie alle andere Threads priorisieren, unterbrechen und verdrängen. Die eigentliche Unterbrechungsroutine bildet die obere Hälfte („*top half*“), nimmt den Interrupt entgegen, speichert die damit verbundenen Daten und weckt den Kernel-Thread mit der unteren Hälfte auf.

In früheren Versionen unterstützte der Kernel Preemption nur für Anwendungsprozesse, spätere Versionen fügten Unterbrechungspunkte in den Kernel-Quelltext ein, um längere Kernel-Tasks unterbrechen zu können. Hierbei kann ein Kernel-Thread jedoch nur an einigen wenigen, explizit definierten Punkten unterbrochen werden. Mithilfe von PREEMPT_RT ist fast der gesamte Kernel-Code unterbrechbar [9]. Zu den wenigen nicht unterbrechbaren Stellen gehören die Unterbrechungsroutinen, die allerdings wie o.g. auf das nötigste reduziert wurden.

3. Testumgebung

3.1. Testfälle

3.1.1. Preemption

Zunächst soll die Grundfunktion des Schedulers, das Unterbrechen eines Tasks durch einen höher priorisierten Task, getestet werden. Dazu werden zwei Threads unterschiedlicher Prioritäten erzeugt. Der höher priorisierte Thread (P_1) legt sich zunächst für eine gewisse Zeit schlafen. Der andere Thread (P_2) läuft nun zunächst in eine Endlosschleife. In jedem Schleifendurchgang wird die aktuelle Zeit bestimmt und in einer Variablen gespeichert. Wacht nun P_1 auf, wird P_2 von diesem verdrängt. P_1 verbringt nun eine festgelegte, ausreichend lange Zeit mit aktivem Warten, d. h. er läuft selbst in einer Endlosschleife, bis die Systemzeit einen bestimmten Wert überschreitet. Danach beendet er sich selbst. P_2 wird wieder aktiv und stellt fest, dass seit seinem letzten Schleifendurchlauf mindestens die von P_1 mit aktivem Warten verbrachte Zeit vergangen ist. Es wird nun die Zeit des letzten Schleifendurchlaufs von P_2 als t_1 und die aktuelle Zeit als t_4 gespeichert. P_1 speichert darüber hinaus beim Aufwachen die Zeit t_2 sowie die Zeit t_3 unmittelbar, bevor er sich selbst beendet.

Bei der Ausführung des Tests muss darauf geachtet werden, dass die Prozessor-Affinitäten beider Prozesse so festgesetzt sind, dass sie auf dem gleichen Prozessor ausgeführt werden.

Der Test gilt als fehlgeschlagen, wenn nicht gilt $t_1 < t_2 < t_3 < t_4$ oder wenn nach einem hinreichend großen Timeout P_2 noch nicht von P_1 unterbrochen wurde (siehe Abbildung 2).

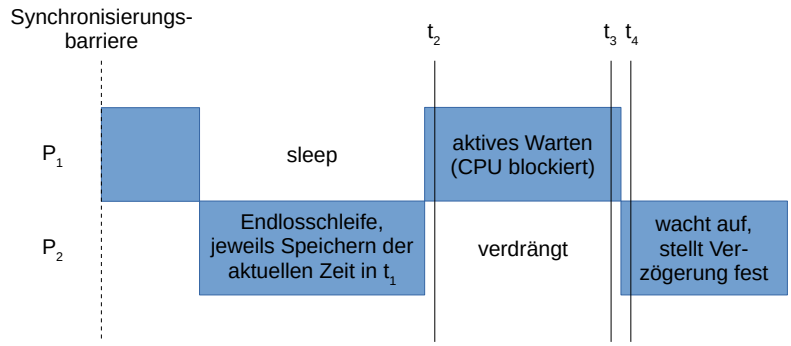

Abbildung 2: Zeitleiste

3.1.2. Auflösung der Uhr

Das Bestimmen der aktuellen Systemzeit nimmt selbst einen gewissen Zeitraum in Anspruch. Zwei unmittelbar aufeinander folgende System-Aufrufe können daher nicht die gleichen Zeitpunkte zurückgeben. Dieser Test misst die minimale Differenz zwischen zwei Zeitmessungen. Um möglichst genaue Ergebnisse zu erhalten sollten zwischen zwei solchen Aufrufen so wenig CPU-Anweisungen wie möglich stattfinden. Daher wird zunächst eine gewisse Menge von Zeitpunken bestimmt und im Speicher abgelegt. Erst im Anschluss wird die minimale Differenz zwischen zwei aufeinander folgenden Messwerten bestimmt.

Die Ergebnisse dieses Tests erlauben eine Aussage über die Genauigkeit der folgenden Tests zu erstellen.

3.1.3. Genauigkeit des Timers

In Echtzeitanwendungen sollen häufig Routinen zu berechneten Zeitpunkten in der Zukunft ausgeführt werden. Dabei bestimmt die Anwendung den Zeitpunkt, zu dem die Routine ausgeführt werden soll, teilt diesen dem System mit und legt sich anschließend selbst schlafen. Das System weckt die Anwendung nach Ablauf des Timers wieder auf. Hier soll nun gemessen werden, wie stark der Zeitpunkt des Wiedereintritts in die Routine vom geplanten Zeitpunkt abweicht. Es werden zwei Varianten der Zeitplanung gemessen.

Bei der Verwendung absoluter Zeitangaben wird ein Zeitpunkt angegeben, nach dessen Ablauf die Anwendung wieder aktiv werden soll. Als erste Anweisung nach dem Aufwachen wird die aktuelle Systemzeit bestimmt. Die Differenz zum geplanten Aufwachzeitpunkt entspricht der gesuchten Abweichung.

Bei der Verwendung relativer Zeitangaben wird anstatt eines Zeitpunktes ein Zeitraum definiert. Die Anwendung wird für die Länge dieses Zeitraums inaktiv. Es muss daher nicht nur nach dem Wiedereintritt in die Routine, sondern auch unmittelbar vor der Aktivierung des Timers die aktuelle Zeit bestimmt werden. Aus der Differenz dieser beiden Zeitpunkte lässt sich die tatsächliche Schlafdauer und damit die Abweichung vom geplanten Zeitraum bestimmen.

3.1.4. Umschaltzeit

Die Umschaltzeit ist die Zeit, die das System für einen Kontextwechsel benötigt. Bei einem Kontextwechsel wird der Zustand des aktiven Tasks gespeichert, der Task in den Wartezustand gebracht und der Zustand eines anderen Tasks wiederhergestellt, um ihn fortzusetzen. Während der eigentliche Kontextwechsel in der Regel relativ schnell ausgeführt werden kann, wird der aufgeweckte Task häufig nicht sofort mit maximaler Geschwindigkeit ausgeführt, da zunächst die Caches des Prozessors neu aufgebaut werden müssen.

Die Umschaltzeit wird gemessen, indem zwei Threads ausgeführt werden, welche sich durch zwei gemeinsame Mutexe immer wieder gegenseitig blockieren, so dass das System ständig zwischen beiden Threads wechseln muss. Das Sperren und Entsperren der Mutexe erfolgt in einer

Reihenfolge, die eine bestimmte Anzahl von Kontextwechseln erfordert, aber keinen Deadlock verursacht. Die Dauer zur Abarbeitung beider Threads wird anschließend durch die Anzahl der Kontextwechsel geteilt.

3.1.5. Auslastung durch SMIs

Ein System Management Interrupt (SMI, auch Hardware-Interrupt genannt) ist ein Interrupt, der nicht von einer Anwendung oder dem Betriebssystem behandelt wird, sondern durch die Firmware des BIOS. Das Betriebssystem kann daher das Auftreten eines SMIs nicht feststellen. Die Verzögerung durch die Ausführung eines SMIs kann mitunter im Bereich von Millisekunden liegen, was bei zeitkritischen Echtzeitsystemen eine exakte Planung unmöglich macht.

Dieser Test stellt einen Spezialfall dar, da er nicht das Verhalten des Linux-Kernels misst, sondern das der Hardware. Er wurde dennoch in die Testreihe aufgenommen, um abzuschätzen, ob die Hardware ausreichend große Latenzen verursacht, welche eine Echtzeitanwendung merklich stören könnte.

Das Auftreten von SMIs kann nur indirekt gemessen werden. Dazu wird das TSC-Register (*Time Stamp Counter*) des Prozessors ausgelesen. Hier wird die Anzahl von Zyklen seit dem letzten Reset des Prozessors gespeichert. Zur Bestimmung von SMIs werden alle Prozessoren für einen bestimmten Zeitraum vom Betriebssystem angehalten. In diesem Zustand können sie lediglich von einem Interrupt wieder aufgeweckt werden, es kann also kein Code in einem Thread-Kontext ausgeführt werden. Wird der Messprozess wieder aktiv und der TSC-Wert ist signifikant höher als vor dem Anhalten, muss in der Zwischenzeit eine Hardware-Unterbrechungsroutine ausgeführt worden sein.

3.2. Implementierung

Die Testumgebung wird in C++ implementiert. Zur Bestimmung der gesuchten Daten kommen Funktionen aus dem POSIX-Standard, der C++-Standardbibliothek sowie einige der *boost*-Bibliotheken zum Einsatz. Zur Messung der SMIs kommt ein Kernel-Modul aus dem PREEMPT_RT-Patch zum Einsatz (siehe Abschnitt 3.2.6). Zum Kompilieren wird weiterhin *CMake* und ein dazu kompatibler Compiler benötigt.

Für die Auswertung der Ergebnisse wird das Programm *gnuplot* benutzt. Dies muss allerdings nicht notwendigerweise auf dem zu testenden System installiert sein.

3.2.1. Konfiguration

Die Testumgebung wird mithilfe einer Textdatei gesteuert, über welche alle relevanten Laufzeitparameter festgelegt werden können. Diese Datei wird in mehreren Ordern gesucht: dem Programmordner, dem Home-Verzeichnis des angemeldeten Benutzers sowie dem aktuellen Arbeitsverzeichnis. Existiert eine Konfiguration an mehreren Orten, wird sie nacheinander eingelesen; später definierte Werte überschreiben frühere. Die Konfiguration im Programmordner enthält Standardwerte für alle möglichen Optionen.

3.2.2. Klassenstruktur

Es existiert eine abstrakte Klasse `Testcase`, welche einen einzelnen Testfall darstellt. Für jeden Testfall ist eine eigene Implementierung dieser Klasse vorhanden (siehe Abbildung 3).

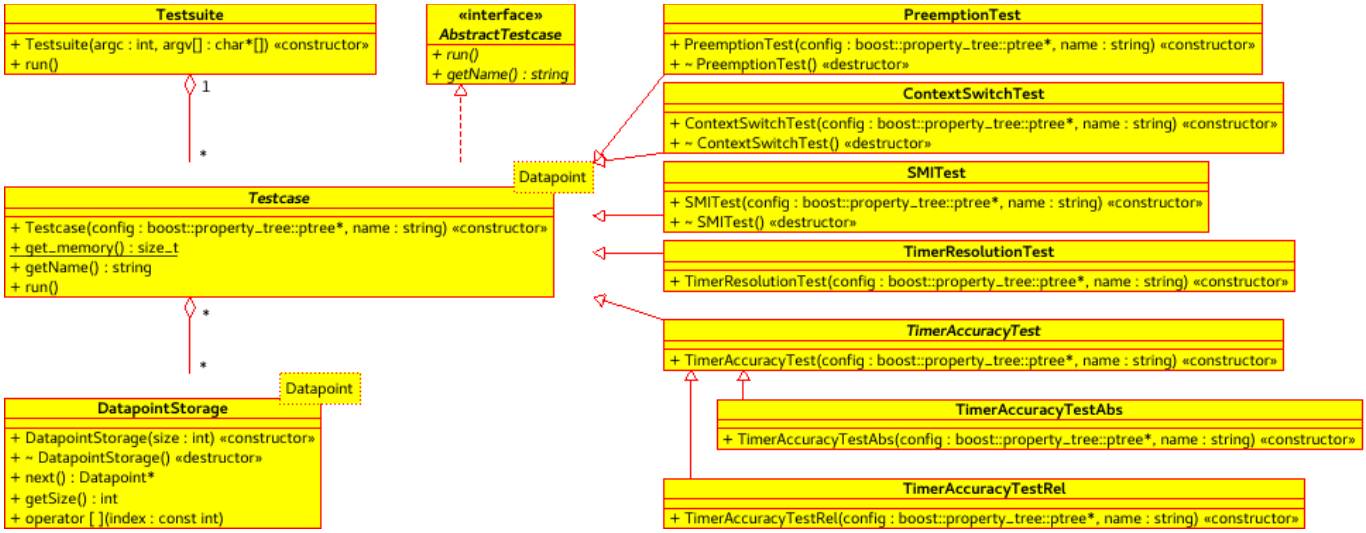

Abbildung 3: Klassenstruktur

Jeder Testfall generiert dabei eine Serie von Messpunkten, welche in Form eines Typparameters in der Klasse `Testcase` verwaltet werden. Die Testumgebung allokiert mittels der Klasse `DatapointStorage` bereits vor der Ausführung eines jeden Tests eine ausreichende Menge Arbeitsspeicher, in welchem die Testergebnisse gespeichert werden (siehe auch Abschnitt 3.2.3). Nachdem alle Messpunkte bestimmt wurden, werden die Ergebnisse auf die Festplatte geschrieben und können später ausgewertet werden. Neben den Messpunkten werden auch die für die Messung verwendeten Konfigurationsparameter gespeichert, da diese in einigen Fällen für die Interpretation der Messwerte relevant sind.

3.2.3. Speicher-Allokation

Speicher-Allokationen blockieren die Programmausführung erheblich und sind daher während des Echtzeitbetriebs dringend zu vermeiden. Es ist deshalb üblich, den benötigten Speicher bereits vor der Ausführung von zeitkritischen Programmanweisungen zu allokieren. Auf den gängigen Systemen werden die Speicherseiten erst beim ersten Schreibzugriff auf dem Arbeitsspeicher reserviert. Dies kann z. B. mit `memset()` erfolgen. Weiterhin muss ausgeschlossen werden, dass das System Teile des Speichers auslagert, da es sonst zu Seitenfehlern und damit zu Verzögerungen kommt. Dies kann mithilfe von `mlockall(MCL_CURRENT | MCL_FUTURE)` ausgeschlossen werden. Dabei werden sowohl aktuell reservierte, sowie alle weiteren noch zu allokierenden Speicherseiten in den Arbeitsspeicher geladen und dort gesperrt.

Linux bietet darüber hinaus mittels `mallopt(M_TRIM_THRESHOLD, -1)` die Möglichkeit, nicht mehr benötigen Speicher nicht an das System zurückzugeben, sondern für kommende Allokierungen in der privaten Speicherzuordnung zu belassen.

3.2.4. Echtzeitpriorität

Neue Prozesse werden standardmäßig dem CFS-Scheduler zugeordnet. Dem System muss mittels `sched_setscheduler(0, SCHED_FIFO, &param)` mitgeteilt werden, dass der aktuelle Prozess stattdessen mit dem Echtzeitscheduler zu planen ist. Über den Parameter `param` wird u.a. auch die Priorität des Prozesses festgelegt.

3.2.5. Prozessorzuordnung

Gängige Hardwaresysteme bieten Prozessoren mit mehreren Kernen, so dass Programme echt parallel ausgeführt werden können. Prozesse können dabei auch während der Ausführung von einem Prozessor zu einem anderen verschoben werden. Dieser Vorgang kann allerdings weitere Verzögerungen verursachen. Daher können Prozesse mithilfe der Funktion `sched_setaffinity` festlegen, dass sie nur einem bestimmten Prozessor (oder einer definierten Menge von möglichen Prozessoren) ausgeführt werden dürfen.

3.2.6. Messung der SMIs

Mit Ausnahme der SMI-Bestimmung messen alle Tests einen Parameter des Betriebssystems aus Sicht einer Echtzeitanwendung. Für den genannten Test dagegen wird die Hardware aus Sicht des Betriebssystems betrachtet. Für diesen Test wird daher das Kernel-Modul *hwlat_detector* benutzt, welches im Rahmen des rt-tests-Projekts entwickelt wurde und Teil des PREEMPT_RT-Patches ist. Das Modul stellt eine Schnittstelle über das virtuelle *debugfs*-Dateisystem des Kernels zur Verfügung.

3.2.7. Berechtigungen

Viele der o. g. Schnittstellen stehen aus Sicherheitsgründen üblicherweise nicht für Benutzeranwendungen zur Verfügung. Die Testumgebung muss daher in der Regel unter dem Benutzer *root* ausgeführt werden.

3.2.8. Statistische Analyse

Die Testumgebung besteht aus zwei unabhängigen Programmen. Die eigentliche Testsuite wird auf dem zu testenden System ausgeführt und misst die gesuchten Parameter. Die Messdaten werden in Form mehrere CSV- und Textdateien gespeichert. Darüber hinaus wurde ein Evaluationsprogramm erstellt, welches aus den gesammelten Daten einige statische Eckdaten wie obere und untere Grenze, Mittelwert sowie Standardabweichung berechnet und mithilfe des Tools *gnuplot* Diagramme generiert. Diese Auswertung muss nicht auf dem gleichen System ausgeführt werden, auf welchem die Messwerte bestimmt wurden.

4. Auswertung

4.1. Testumgebung

Die Tests wurden auf einer HP Z240 Tower Workstation ausgeführt. Die Maschine verfügt über 32 GB Arbeitsspeicher, ca. 10 GB Auslagerungsspeicher (Swap). In der Maschine ist ein Prozessor vom Typ Intel Core i5-6600 verbaut, welcher vier physische Rechenkerne enthält. Pro physischem Kern ist exakt ein logischer Kern vorhanden (siehe Abbildung 4). Der Prozessor entspricht der x86_64-Architektur (64 Bit Wortbreite) und hat eine Normtaktung von 3,3 GHz. Es wurden keine Maßnahmen zur Übertaktung vorgenommen.

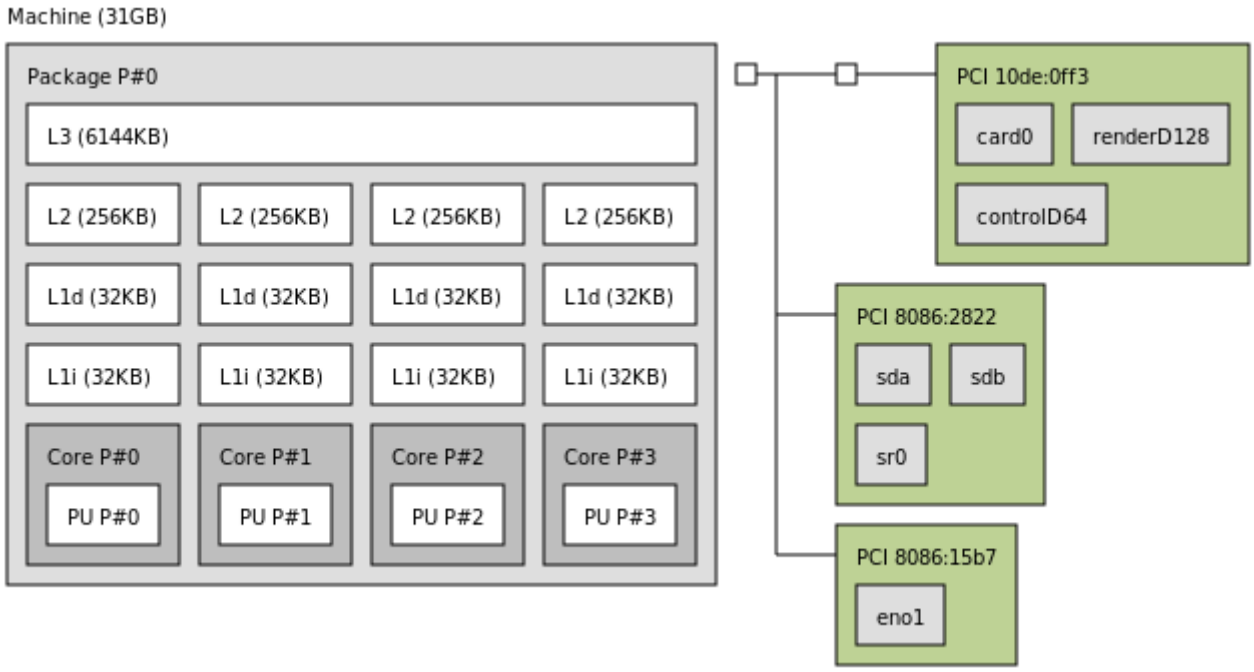

Abbildung 4: Prozessorarchitektur

Auf dem System wurde die Linux-Distribution Ubuntu in der Version 16.04.2 LTS (Codename Xenial Xerus) installiert. Diese Distribution liefert standardmäßig einen *generic*-Kernel über die Paketquellen aus. Darüber hinaus wurde ebenfalls aus den Paketquellen die *lowlatency*-Variante des Kernels installiert, welche den PREEMPT_RT-Patch enthält. Beide Kernel liegen in der Version 4.4.0-78 vor.

Da der *lowlatency*-Kernel jedoch nicht das *hwlat_detector*-Modul umfasst, welches zur Messung der SMIs erforderlich ist, wurde außerdem ein Kernel manuell gepatcht und kompiliert. Bei der Kompilierung wurde die Kernel-Konfiguration des *lowlatency*-Kernels als Vorlage benutzt, allerdings mit Übersetzung des oben genannten Kernel-Moduls. Dieser im folgenden als *custom* bezeichnete Kernel trägt die Version 4.4.27-rt37.

4.2. Durchführung

Die Testsuite wurde nacheinander unter den drei verschiedenen Kerneln ausgeführt. Als Ausnahme davon erfolgte die Bestimmung der SMI-Latenzen lediglich mithilfe des *custom*-Kernels, da in den Kerneln der Paketquellen das *hwlat_detector*-Modul nicht mitgeliefert wird.

Die Tests belasten jeweils nur eine einzige CPU, der Rest des Systems befindet sich während der Ausführung weitgehend im Leerlauf. Die Testreihe wurde daher unter allen drei Kernel-Varianten erneut ausgeführt, während gleichzeitig das Tools *stress* alle CPUs künstlich auslastete. Der SMI-Test war in diesen Durchläufen deaktiviert.

Die als CSV-Dateien gespeicherten Messwerte wurden anschließend mithilfe des Evaluationsprogramms ausgewertet. Die Konfiguration der Testsuite sowie die Ergebnisse des Evaluationsprogramms befinden sich im Anhang dieses Dokuments.

4.3. Interpretation der Testergebnisse

4.3.1. Preemption

Im Folgenden werden die Testergebnisse unter den drei verschiedenen Kerneln gegenübergestellt.

Die Messwerte des Preemption-Tests sind sich sehr ähnlich in Hinblick auf Mittelwert, Standardabweichung und untere Grenze. Auf dem *generic*-Kernel sind allerdings erhöhte obere Grenzwerte festzustellen, das heißt es kommt zu stärkeren Ausreißern in der Datenreihe. Der Kontextwechsel von P_2 zu P_1 dauerte mitunter etwas über 0,1 ms. Eine genauere Analyse wird hier nicht durchgeführt, da dieser Parameter noch gesondert ausgewertet wird.

4.3.2. Auflösung der Uhr

Die Auflösung des Zeitmessers ist sehr stabil. Auf allen drei Kerneln lagen mindestens 15 ns und durchschnittlich ca. 18 ns zwischen zwei Systemaufrufen. In allen Testvarianten konnte jedoch beträchtliche Maxima von bis zu 25 µs festgestellt werden mit Häufungen im Bereich von etwa 1 µs und 10–20 µs. Erstaunlicherweise fanden sich auf dem *generic*-Kernel lediglich 12 Messwerte über 25 ns, auf dem *custom*-Kernel dagegen 42 und auf dem *lowlatency*-Kernel 57 bei jeweils 1.000.000 Messpunkten. Die Standardabweichung (*generic*: 38 ns, *lowlatency*: 84ns, *custom*: 70ns) wird daher nicht durch die Anzahl, sondern die Werte dieser Messspitzen stark ausgelenkt. Rechnet man diese heraus, zeigen alle Messreihen eine Standardabweichung von nur 0,54 ns, welche damit kleiner ist als die Auflösung des Messverfahrens, da der Kernel die Zeiten in ganzen Nanosekunden zurückgibt. Obwohl über 99 % aller Messwerte innerhalb eines äußert kleinen Intervalls liegen, muss dennoch die Existenz von Messspitzen berücksichtigt werden.

4.3.3. Genauigkeit der Timer

Bei der Messung der Timer-Genauigkeit zeigt sich eine Anhäufung von Datenpunkten im Bereich von ca. 1–2 µs mit gewissen Schwankungen nach oben sowie nach unten. Darüber gibt es eine weitere, deutlich kleinere Anhäufung von Datenpunkten im Bereich 10–20 µs. Der direkte Vergleich

zwischen absoluten und relativen Timern auf jeweils dem gleichen Kernel zeigt keine bedeutenden Unterschiede, mit einer Ausnahme: Die oben genannte zweite Häufung tritt auf dem *custom*-Kernel bei der Verwendung absoluter Timer nicht länger auf, was sich in einer um eine Größenordnung reduzierten Standardabweichung äußert.

4.3.4. Umschaltzeit

Die Messung der Umschaltzeiten zeigt auf allen Kernel-Varianten eine harte untere Grenze, diese liegt bei den *generic*- und *lowlatency*-Kerneln jedoch etwas niedriger (550 ns bzw. 559 ns) als beim *custom*-Kernel (694 ns). Auch der Durchschnittswert liegt bei letzterem etwas höher (710 ns gegenüber 557 ns bzw. 570 ns). Alle Systeme zeigen eine gewisse Streuung oberhalb ihres jeweiligen Durchschnittswertes. Da die Standardabweichung bei allen Testläufen vergleichbar ist, kann angenommen werden, dass ein Kontextwechsel auf dem *custom*-Kernel grundsätzlich ca. 140–150 ns länger dauert.

4.3.5. Test unter CPU-Last

Bei Ausführung der Testsuite unter voller CPU-Last wurde in fast allen Ergebnisreihen eine Verschlechterung der Messwerte festgestellt.

Bei der Messung der Umschaltzeiten hat sich die Streuung bei starker CPU-Last deutlich reduziert, obwohl sich die Durchschnittswerte nur verhältnismäßig gering veränderten. Während sich beim *generic*-Kernel zwei dünne Häufungen an Messpunkten ausmachen lassen, zeigen die Echtzeit-Kernel eine breitere Verteilung, dafür eine relative harte obere Grenze mit sehr wenigen Überschreitungen.

Die deutlichsten Unterschiede zeigen sich bei Betrachtung der Timer-Genauigkeit. Während der *generic*-Kernel hier sogar niedrigere Durchschnittswerte bei verminderter Streuung liefert, sind die Messwerte bei den Echtzeit-Kerneln um mehrere Größenordnungen gestiegen.

Eine Erklärung für dieses Verhalten konnte nicht gefunden werden. Das Programm *stress* erhielt eine verringerte Priorität (nice-Wert 10) und wurde dem CFS-Scheduler zugeordnet. Das Messprogramm, welches mit Echtzeitpriorität arbeitete, hätte daher zu jedem Zeitpunkt den Vorrang erhalten müssen.

Weiterhin ist festzustellen, dass die Timer-Genauigkeit bei den Echtzeit-Kerneln eine markante Verteilung aufweist. Es wurden sehr diskrete Abstufungen mit konstantem Abstand zwischen den Spitzen registriert. Der *generic*-Kernel zeigt keine solche Verteilung.

4.3.6. Auslastung durch SMIs

Bei einer Laufzeit von 30 Minuten wurden 1801 Hardware-Interrupts gemessen, dies entspricht ziemlich genau einem Interrupt pro Sekunde. Der überschüssige Interrupt lässt sich durch das Messverfahren erklären, da das Programm nicht exakt nach Ablauf der Messzeit abbricht, sondern erst nach dem nächsten darauf folgenden Interrupt.

Das *hwlat_detector*-Modul misst die Latenzen mit einer Auflösung von 1 µs. Die Interrupts erzeugten im Test eine durchschnittliche Latenz von 26,12 µs bei einer Standardabweichung von lediglich 0,87 µs. Der kürzeste gemessene Interrupt dauerte 22 µs, der längste 33 µs. 80 % aller gemessenen Latenzen wurden mit einer Dauer 26 µs aufgezeichnet, 95 % liegen innerhalb des Intervalls von 25 µs bis 27 µs.

5. Zusammenfassung

Im Rahmen dieser Arbeit wurden zunächst einige Systemparameter bestimmt, welche für die Ausführung von Echtzeitanwendungen relevant sind. Die Kenntnis dieser Parameter soll bei der Entscheidung helfen, ob ein gegebenes System für die Ausführung einer konkreten Echtzeitanwendung geeignet ist.

Im nächsten Schritt sind Messverfahren aufgestellt worden, mit denen diese Parameter auf dem zu testenden System gemessen werden können.

Anschließend erfolgte die Entwicklung der benötigten Werkzeuge, um diese Messungen durchführen zu können. Diese Werkzeuge bestehen aus einer Echtzeitanwendung, welche die aufgezeigten Messverfahren implementiert und auf dem betreffenden System ausgeführt wird, sowie aus einem Auswertungsprogramm, welches eine statistische Untersuchung der Messwerte vornimmt.

Die Analyse der Ergebnisse zeigte, dass der Linux-Kernel durchaus als Echtzeitkernel verwendet werden kann, wenngleich mit Einschränkungen. Die Entscheidung, ob ein Betriebssystem als Echtzeitsystem in Frage kommt, ist letztlich stark von der Echtzeitanwendung und deren zeitlichen Anforderungen abhängig. Die Messergebnisse zeigten in den meisten Fällen keine absoluten Höchstwerte, allerdings ließen sich für die meisten Testfälle Grenzen bestimmen, die nur von einigen wenigen Messpunkten überschritten wurden. Diese Feststellung schließt harte Echtzeitfähigkeit zunächst aus, sofern nicht durch andere Verfahren doch noch entsprechende Obergrenzen bestimmt werden können. Eine Nutzung für weiche Echtzeit erscheint hingegen plausibel.

Es wurde allerdings auch festgestellt, dass eine hohe CPU-Last auch durch Nicht-Echtzeitanwendung die Messwerte drastisch beeinflussen kann. Das Auftreten solche Lasten ist daher auf dem System auf jeden Fall zu verhindern.

6. Ausblick

Die Auswertung der Messdaten zeigte einige Erkenntnisse auf, für die keine Erklärung gefunden wurde. Hier sollten weitere Studien durchgeführt werden, die entweder deren Ursache ergründen oder deren Auswirkungen auf Echtzeitanwendung untersuchen. Insbesondere die Tatsache, dass sich die Performance einer Echtzeitanwendung durch eine niedrig priorisierte Anwendung stark verschlechterte, sollte näher beleuchtet werden.

Das Echtzeitverhalten des Kernels lässt sich durch einige Konfigurationsparameter beeinflussen. Hier könnte untersucht werden, ob sich durch diese das festgestellte Verhalten erklären lässt oder gar verhindert werden kann.

7. Referenzen

[1] Christian Heursch, Arnd Christian (2006): *Using Standard Operating Systems for Time Critical Applications with special emphasis on LINUX*
http://athene-forschung.unibw.de/doc/86162/86162.pdf (Zugriff 29.05.2017)

[2] Abeni, Luca; Goel, Ashvin; Krasic, Charles; Snow, Jim; Walpole, Jonathan (2002): *A Measurement-Based Analysis of the Real-Time Performance of Linux*
http://web.cecs.pdx.edu/~walpole/class/cs533/papers/rtas2002.pdf (Zugriff 29.05.2017)

[3] Regnier, Paul; Lima, George; Barreto, Luciano (2008): *Evaluation of Interrupt Handling Timeliness in Real-Time Linux Operating Systems*
https://pdfs.semanticscholar.org/4e9d/5faa33bfb5b215fa682b1b73a9a5ba712d5a.pdf (Zugriff (29.05.2017)

[4] Cerqueira, Felipe; Brandenburg, Björn B. (2013): *A Comparison of Scheduling Latency in Linux, PREEMPT_RT, and LITMUS*
http://pubman.mpdl.mpg.de/pubman/item/escidoc:2173547/component/escidoc:2173546/ospert13.pdf
(Zugriff 29.05.2017)

[5] Seeker, Volker (2013): *Process Scheduling in Linux*
http://webpages.iust.ac.ir/e_asyabi/osl/osl3/osl3-files/linux_scheduler_notes_final.pdf (Zugriff 29.05.2017)

[6] Jones, M. (2009): *Inside the Linux 2.6 Completely Fair Scheduler. Providing fair access to CPUs since 2.6.23*
https://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/ (Zugriff 29.05.2017)

[7] *RTwiki. Real-Time Linux Wiki*
https://rt.wiki.kernel.org (Zugriff 29.05.2017)

[8] *High resolution timers*
https://rt.wiki.kernel.org/index.php/High_resolution_timers (Zugriff 29.05.2017)

[9] Huang, Jim (2016): *Making Linux do Hard Real-time*
https://de.slideshare.net/jserv/making-linux-do-hard-realtime (Zugriff 29.05.2017)

8. Anhang

8.1. Konfiguration der Testumgebung

```
iterations = 1000000
priority = 80

[PreemptionTest]
iterations = 100000
sleepNs = 200000
highPriority = 80
lowPriority  = 70
processor = 1

[ContextSwitchTest]
processor = 1
numSwitches = 1000

[SMITest]
duration = 1800
threshold = 10
window = 1000000
width = 500000

[TimerAccuracyTestAbs]
sleep = 5000
processor = 1

[TimerAccuracyTestRel]
sleep = 5000
processor = 1

[TimerResolutionTest]
processor = 1
```

Listing 1: Konfiguration der Testumgebung

8.2. Ergebnisse *generic*

```
PreemptionTest
100000 datapoints
    Column        Min        Avg        Max      StDev
       cs1     684.00     825.35  100553.00    1392.04
      wait      54.00     262.11   86309.00    1047.36
       cs2     573.00     647.96   25535.00     330.44
Successfull passes:     100000
Failed passes:               0

TimerResolutionTest
1000000 datapoints
    Column        Min        Avg        Max      StDev
        dt      15.00      17.51   24250.00      38.45

ContextSwitchTest
1000000 datapoints
    Column        Min        Avg        Max      StDev
      time     550.20     557.49     788.95      10.54

TimerAccuracyTestAbs
1000000 datapoints
    Column        Min        Avg        Max      StDev
     delay     757.00    1212.81  101671.00     343.27

TimerAccuracyTestRel
1000000 datapoints
    Column        Min        Avg        Max      StDev
     delay     893.00    1373.18   90653.00     410.24
```

Listing 2: Zusammenfassung – generic

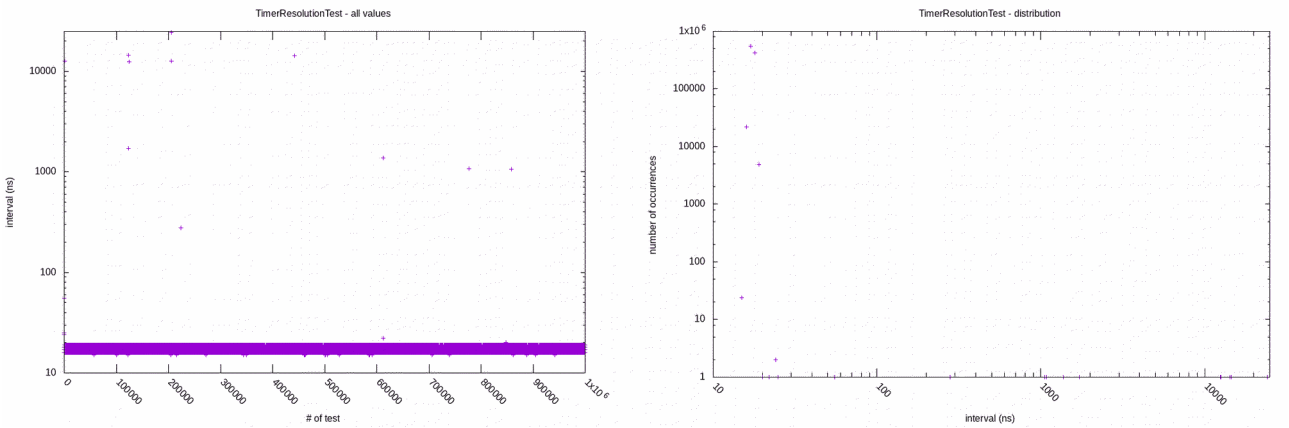

Abbildung 5: Timer-Auflösung – generic

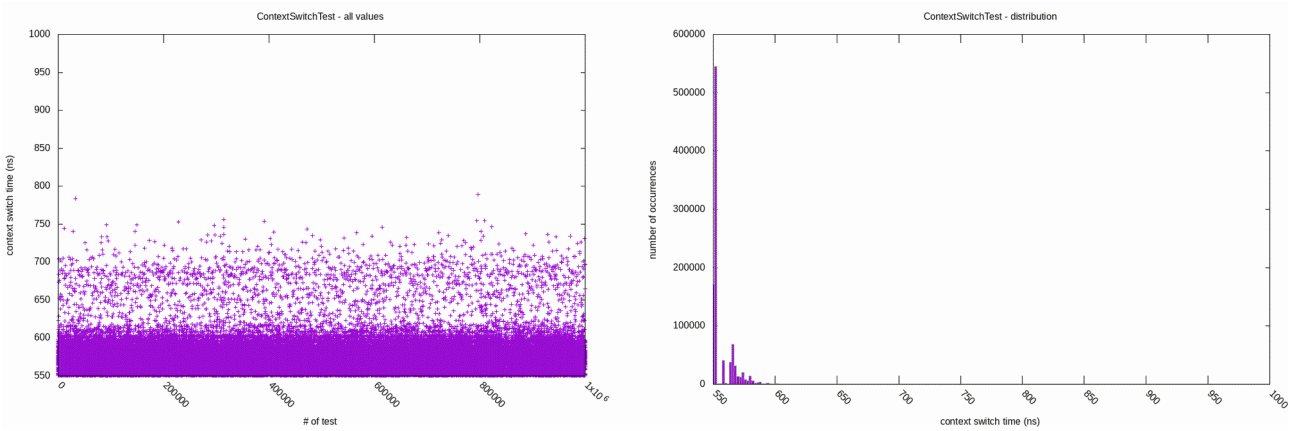

Abbildung 6: Kontextwechsel – generic

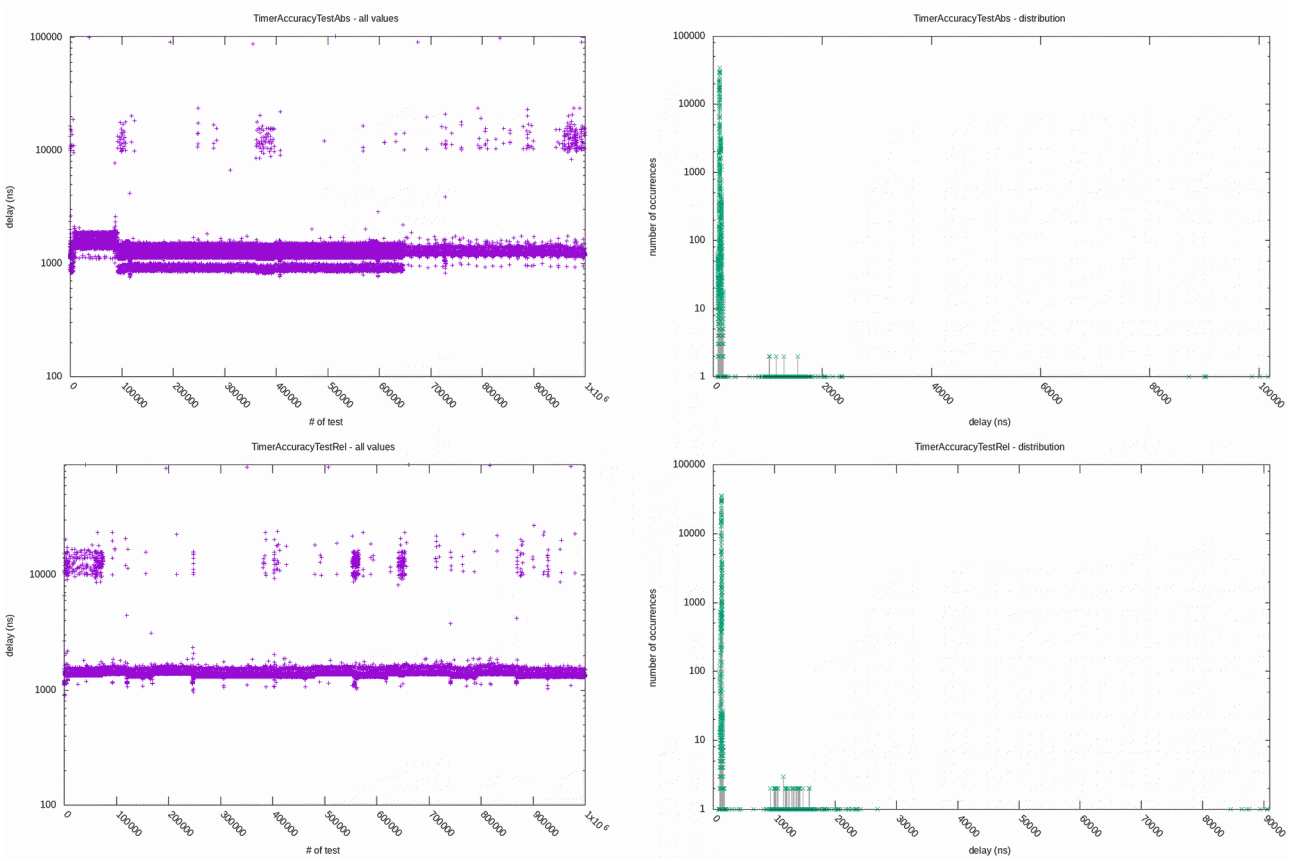

Abbildung 7: Timer-Genauigkeit – generic

8.3. Ergebnisse *lowlatency*

```
PreemptionTest
100000 datapoints
    Column        Min        Avg        Max      StDev
       cs1     699.00     998.69   23748.00    1651.59
      wait      53.00     291.87   19653.00     969.53
       cs2     561.00     702.53   23909.00     407.26
Successfull passes:     100000
Failed passes:               0

TimerResolutionTest
1000000 datapoints
    Column        Min        Avg        Max      StDev
        dt      15.00      17.88   25356.00      83.94

ContextSwitchTest
1000000 datapoints
    Column        Min        Avg        Max      StDev
      time     559.03     569.70     848.30      13.94

TimerAccuracyTestAbs
1000000 datapoints
    Column        Min        Avg        Max      StDev
     delay     698.00    1279.31   65201.00     396.89

TimerAccuracyTestRel
1000000 datapoints
    Column        Min        Avg        Max      StDev
     delay     780.00    1443.47   88265.00     699.03
```

Listing 3: Zusammenfassung – lowlatency

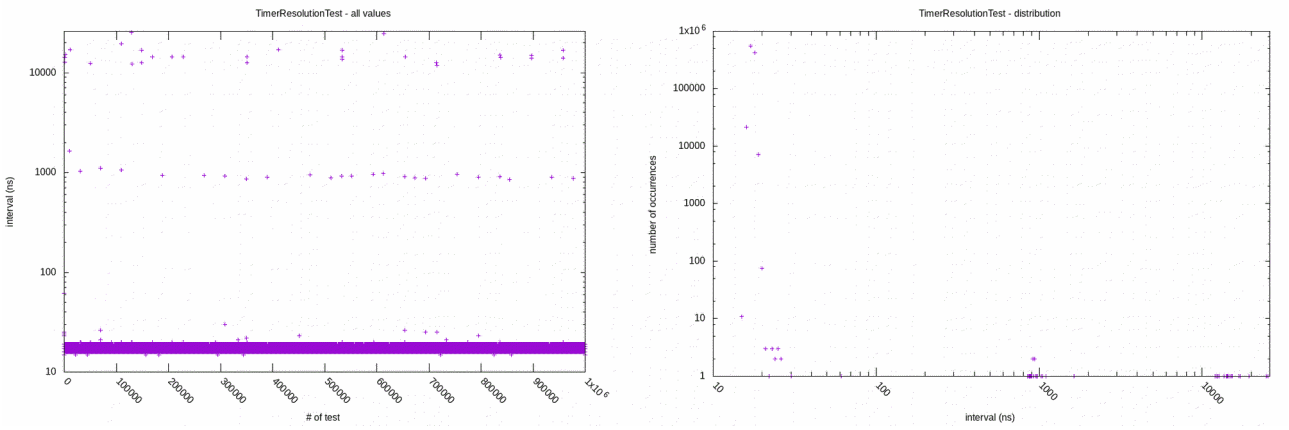

Abbildung 8: Timer-Auflösung – lowlatency

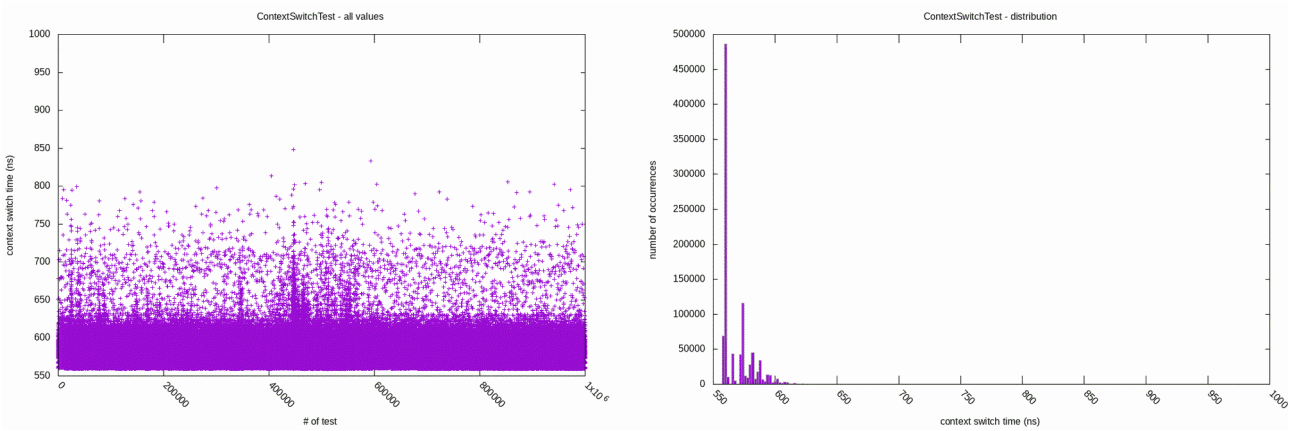

Abbildung 9: Kontextwechsel – lowlatency

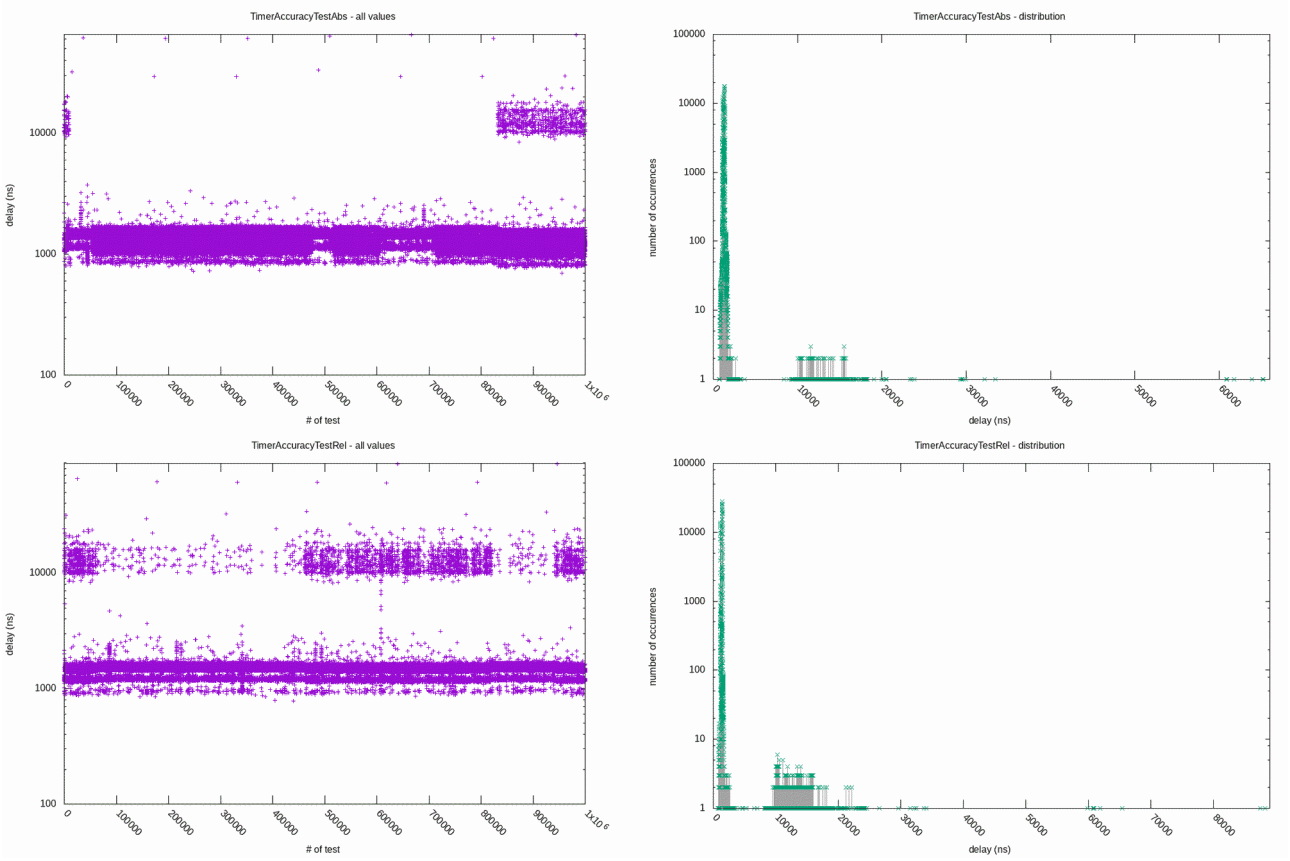

Abbildung 10: Timer-Genauigkeit – lowlatency

8.4. Ergebnisse *custom*

```
PreemptionTest
100000 datapoints
    Column        Min        Avg        Max      StDev
       cs1     787.00    1115.32   23914.00    1830.51
      wait      52.00     321.33   22523.00    1109.73
       cs2     645.00     777.30   20441.00     473.27
Successfull passes:     100000
Failed passes:               0

TimerResolutionTest
1000000 datapoints
    Column        Min        Avg        Max      StDev
        dt      15.00      17.76   22509.00      69.99

ContextSwitchTest
1000000 datapoints
    Column        Min        Avg        Max      StDev
      time     693.91     709.70     991.53      16.46

TimerAccuracyTestAbs
1000000 datapoints
    Column        Min        Avg        Max      StDev
     delay     950.00    1659.37   27641.00      71.70

TimerAccuracyTestRel
1000000 datapoints
    Column        Min        Avg        Max      StDev
     delay     783.00    1359.47   26783.00     854.73
```

Listing 4: Zusammenfassung – custom

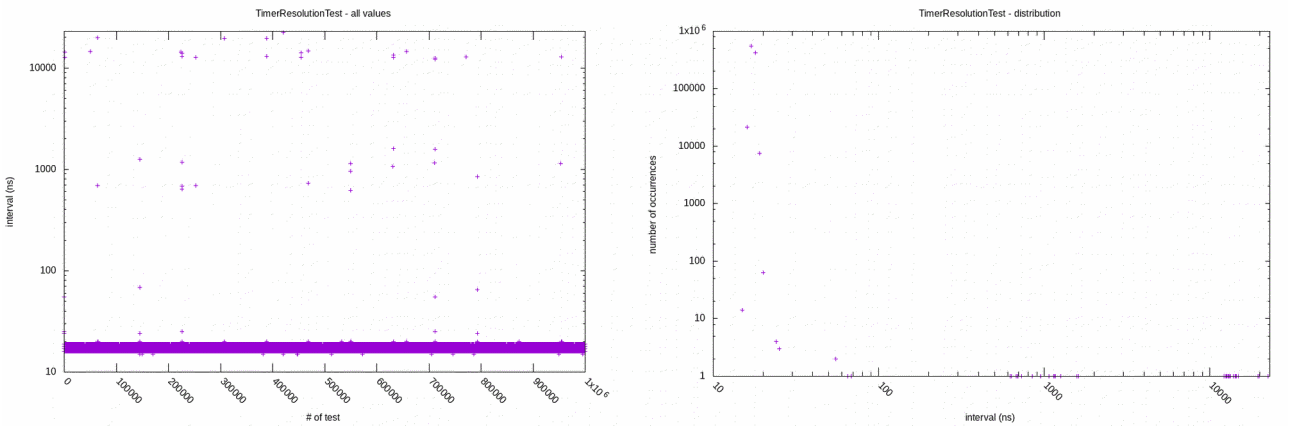

Abbildung 11: Timer-Auflösung – custom

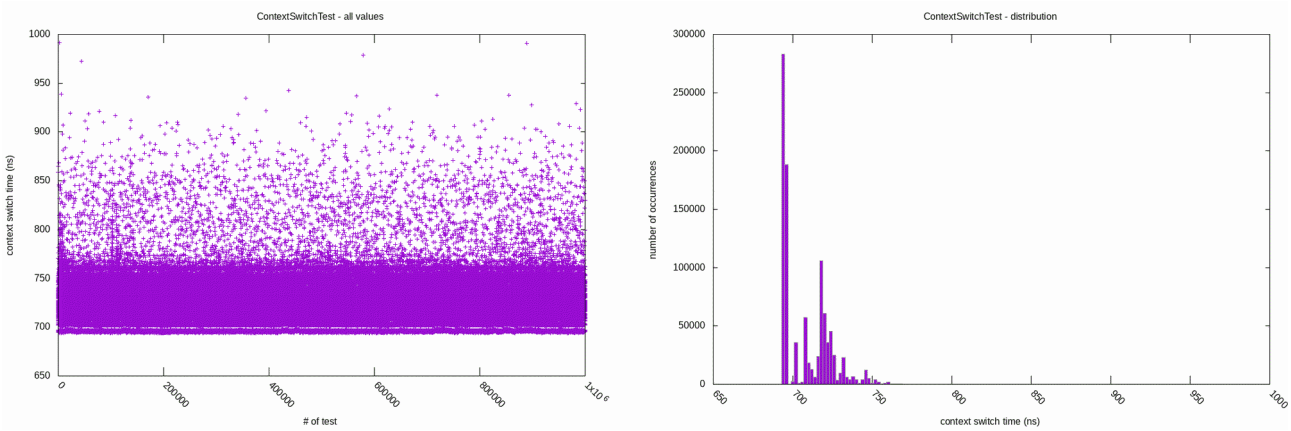

Abbildung 12: Kontextwechsel – custom

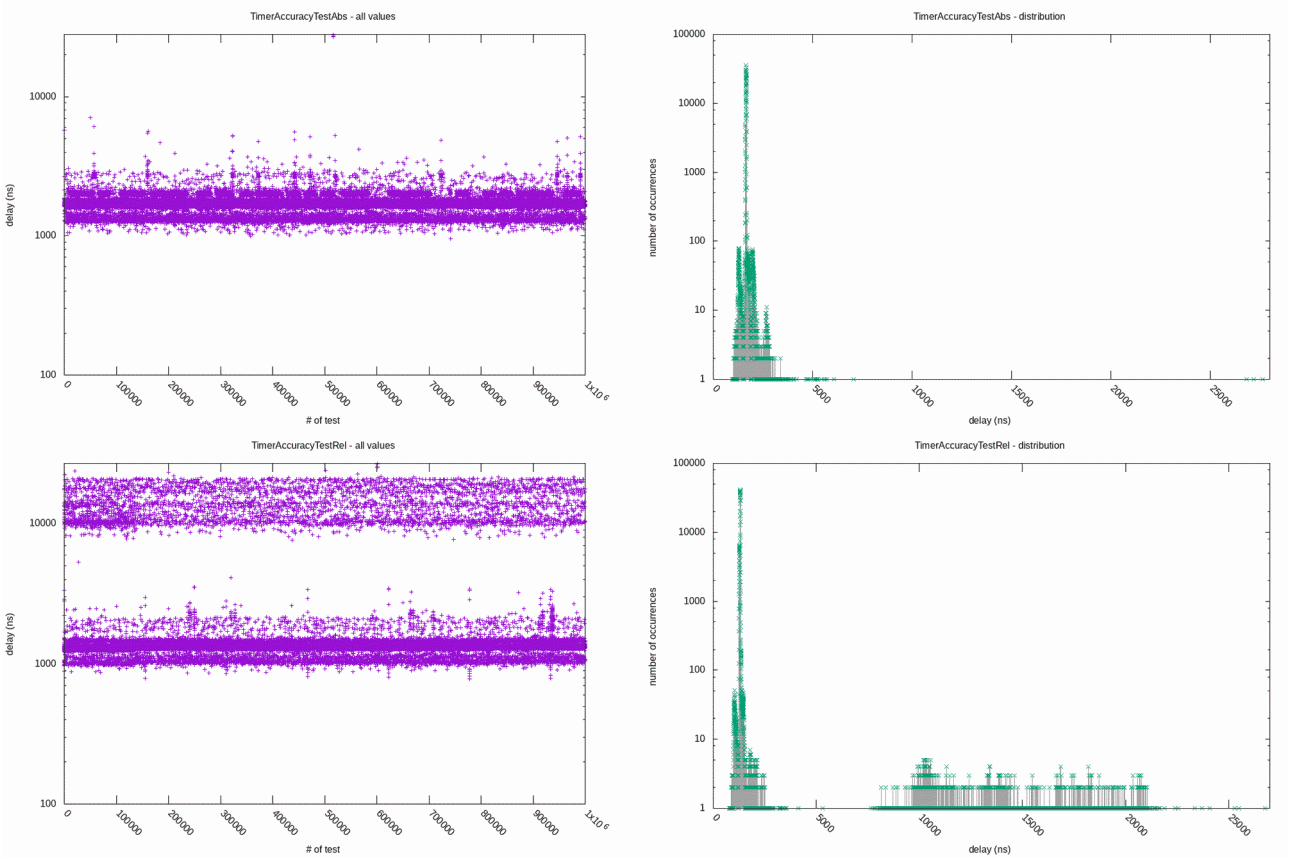

Abbildung 13: Timer-Genauigkeit – custom

8.5. Ergebnisse *generic* unter Last

```
PreemptionTest
100000 datapoints
    Column        Min        Avg        Max      StDev
       cs1     730.00     906.71  100641.00     723.55
      wait      58.00     213.79   89381.00     562.38
       cs2     566.00     670.93    2047.00      25.85
Successfull passes:     100000
Failed passes:               0

TimerResolutionTest
1000000 datapoints
    Column        Min        Avg        Max      StDev
        dt      16.00      18.38    3485.00       3.97

ContextSwitchTest
1000000 datapoints
    Column        Min        Avg        Max      StDev
      time     580.10     581.73     775.92       1.65

TimerAccuracyTestAbs
1000000 datapoints
    Column        Min        Avg        Max      StDev
     delay     753.00     800.02   90331.00     216.27

TimerAccuracyTestRel
1000000 datapoints
    Column        Min        Avg        Max      StDev
     delay     851.00     902.54   89598.00     213.91
```

Listing 5: Zusammenfassung – generic unter Last

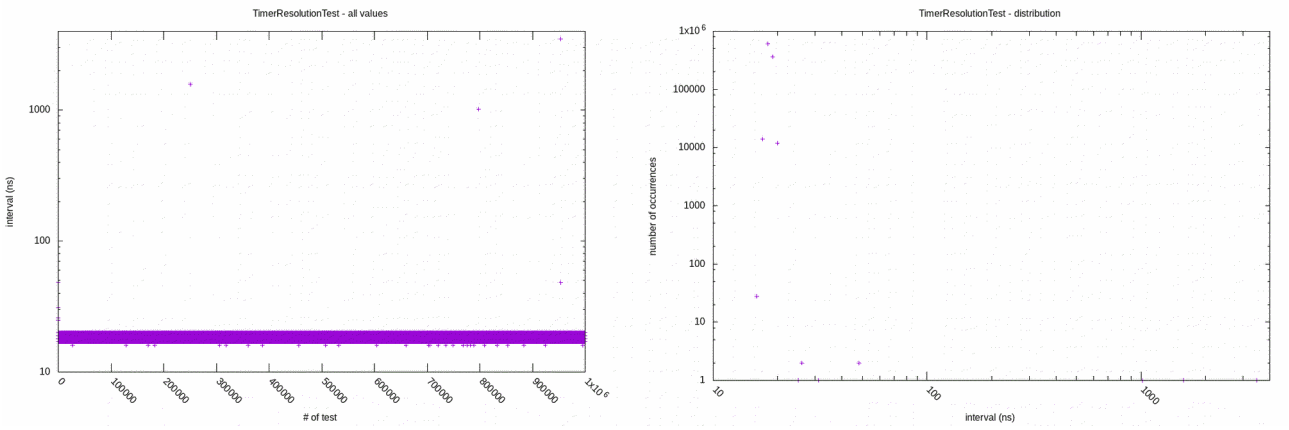

Abbildung 14: Timer-Auflösung – generic unter Last

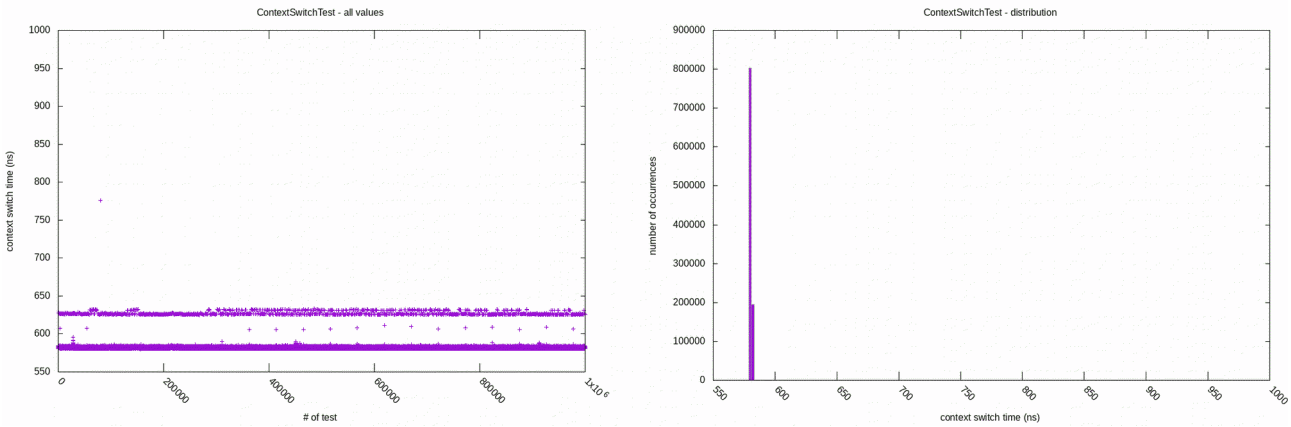

Abbildung 15: Kontextwechsel – generic unter Last

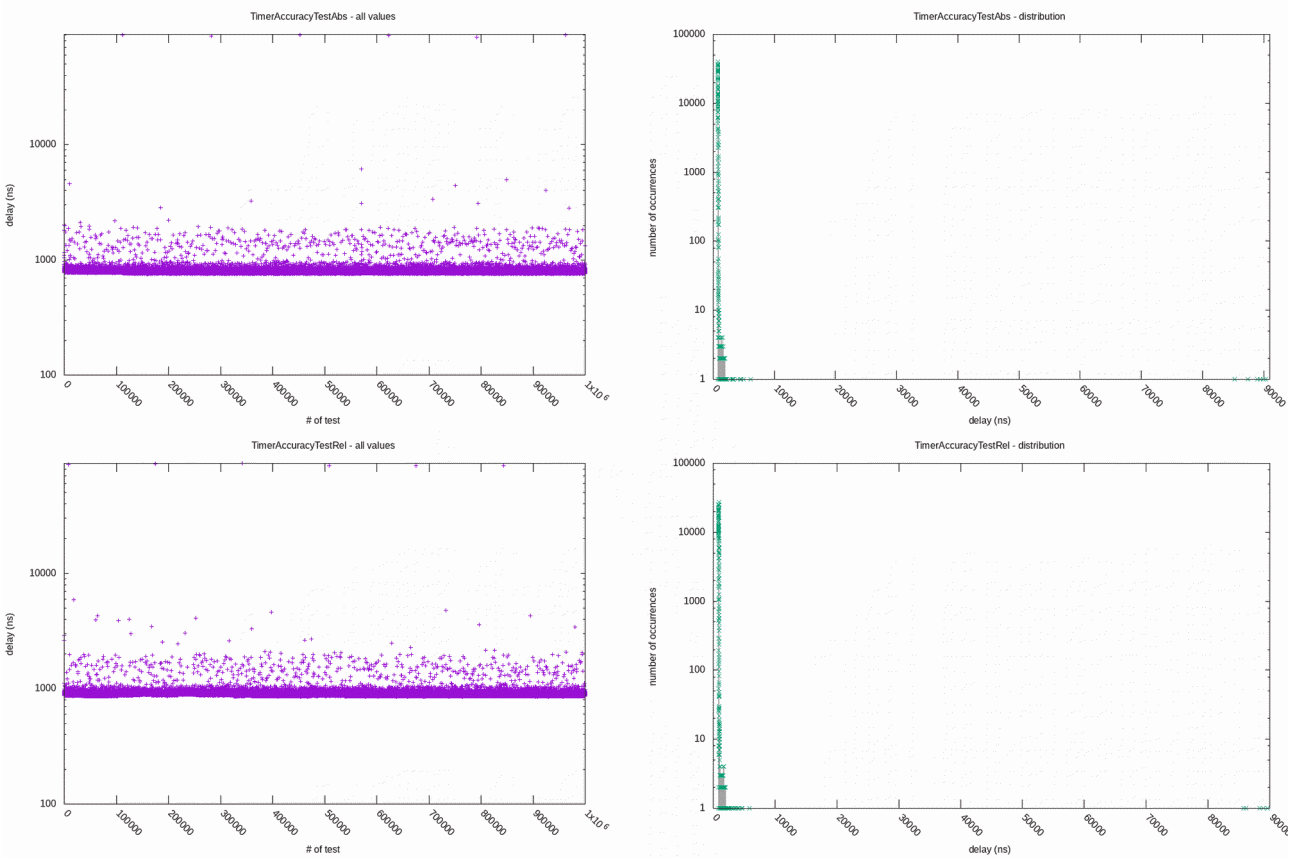

Abbildung 16: Timer-Genauigkeit – generic unter Last

8.6. Ergebnisse *lowlatency* unter Last

```
PreemptionTest
100000 datapoints
    Column        Min        Avg        Max      StDev
       cs1   13416.00   19794.47  121582.00   18550.48
      wait   11163.00   19984.75  137703.00   23028.53
       cs2    4458.00    6434.63  108081.00   10864.87
Successfull passes:     100000
Failed passes:               0

TimerResolutionTest
1000000 datapoints
    Column        Min        Avg        Max      StDev
        dt    4458.00    4915.86  117128.00    5251.62

ContextSwitchTest
1000000 datapoints
    Column        Min        Avg        Max      StDev
      time     603.53     632.82     863.07      16.73

TimerAccuracyTestAbs
1000000 datapoints
    Column        Min        Avg        Max      StDev
     delay   12917.00   14623.69  184548.00   10595.46

TimerAccuracyTestRel
1000000 datapoints
    Column        Min        Avg        Max      StDev
     delay   17417.00   19363.89  194464.00   11261.09
```

Listing 6: Zusammenfassung – lowlatency unter Last

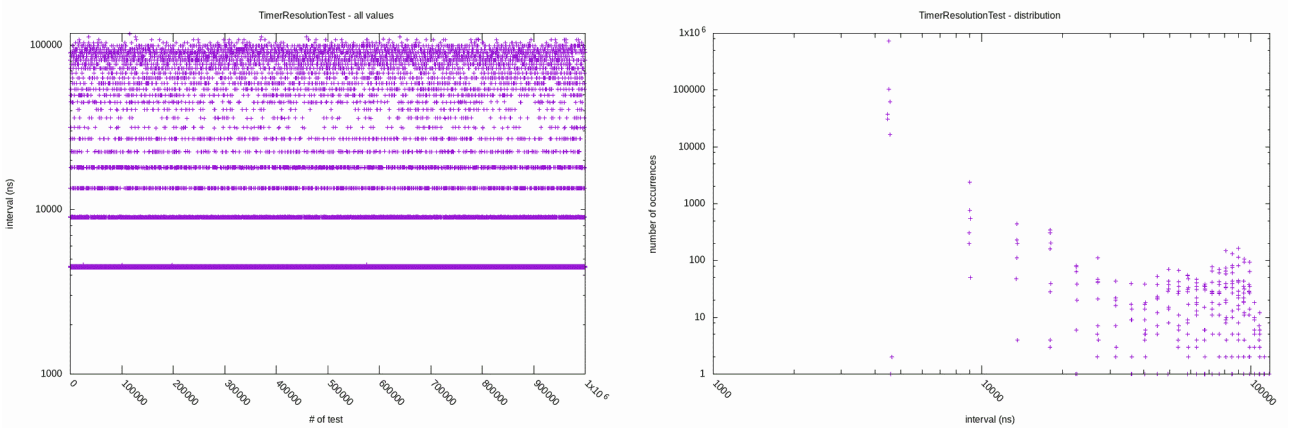

Abbildung 17: Timer-Auflösung – lowlatency unter Last

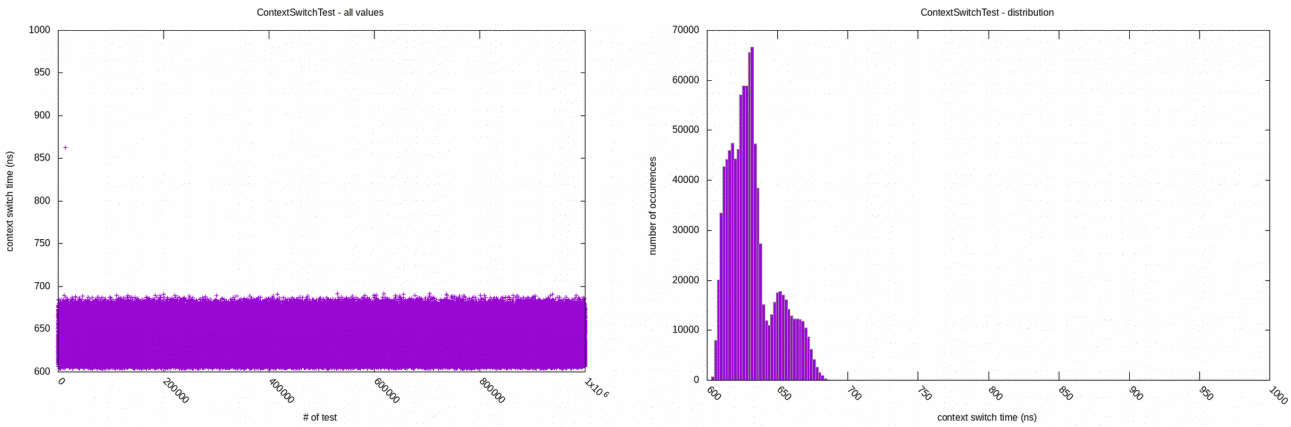

Abbildung 18: Kontextwechsel – lowlatency unter Last

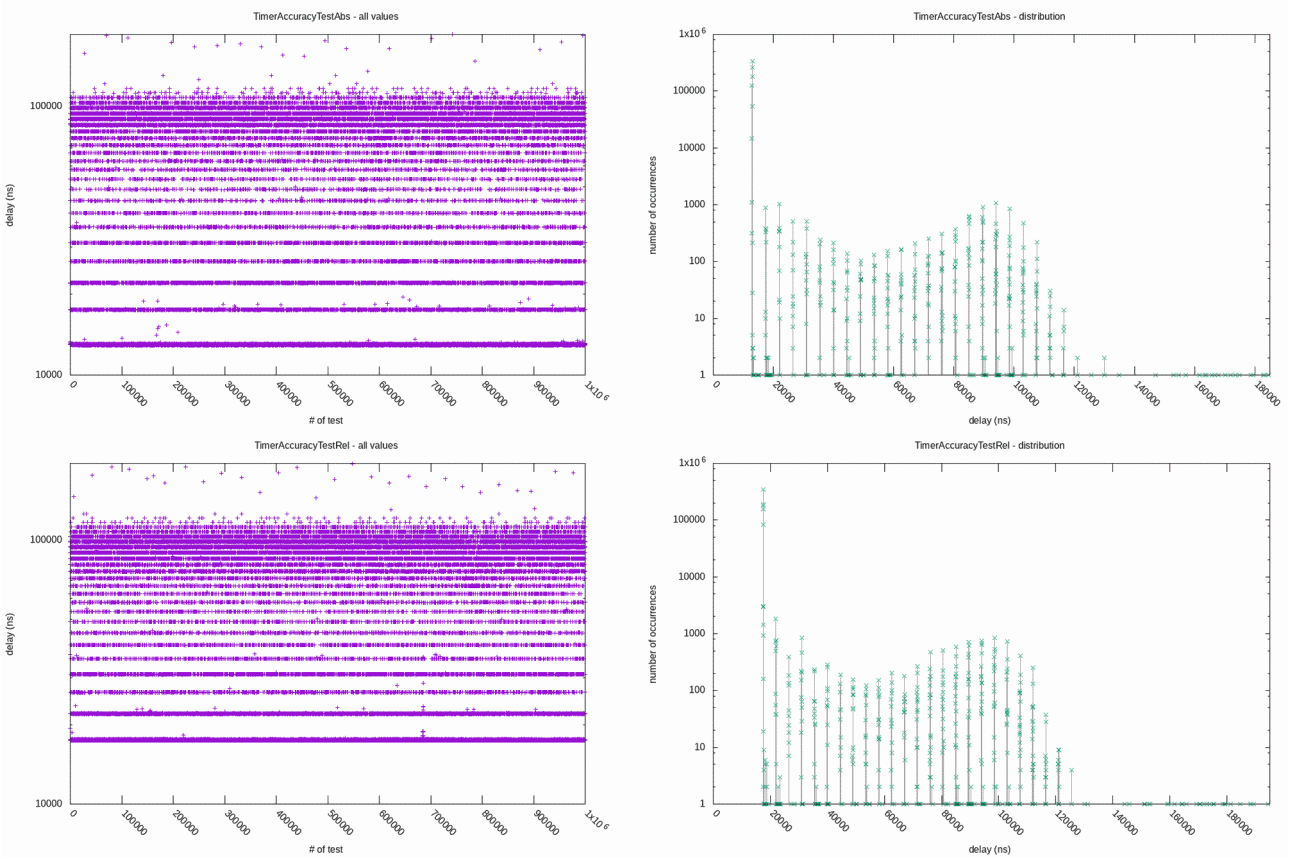

Abbildung 19: Timer-Genauigkeit – lowlatency unter Last

8.7. Ergebnisse *custom* unter Last

```
PreemptionTest
100000 datapoints
    Column        Min        Avg        Max      StDev
       cs1   13416.00   15080.49  130624.00    7628.55
      wait   10956.00   12227.34  115122.00    4534.96
       cs2    4917.00    5994.65  118333.00    2437.94
Successfull passes:     100000
Failed passes:               0

TimerResolutionTest
1000000 datapoints
    Column        Min        Avg        Max      StDev
        dt    4458.00    4732.31  116915.00    2788.38

ContextSwitchTest
1000000 datapoints
    Column        Min        Avg        Max      StDev
      time     752.60     769.45     903.84      17.02

TimerAccuracyTestAbs
1000000 datapoints
    Column        Min        Avg        Max      StDev
     delay   12916.00   13888.12 47073595.00   47447.42

TimerAccuracyTestRel
1000000 datapoints
    Column        Min        Avg        Max      StDev
     delay   17416.00   18746.04  148082.00    6826.90
```

Listing 7: Zusammenfassung – custom unter Last

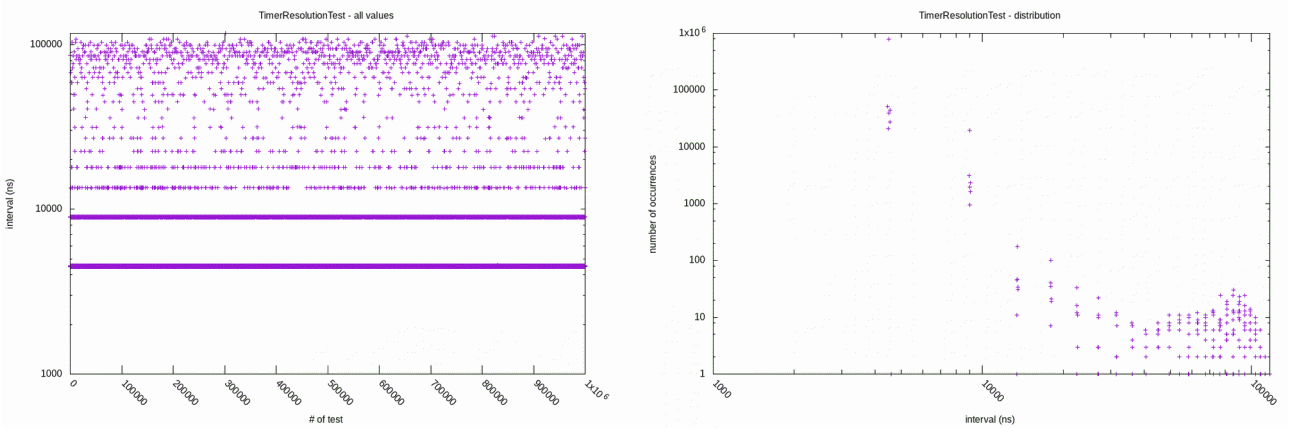

Abbildung 20: Timer-Auflösung – custom unter Last

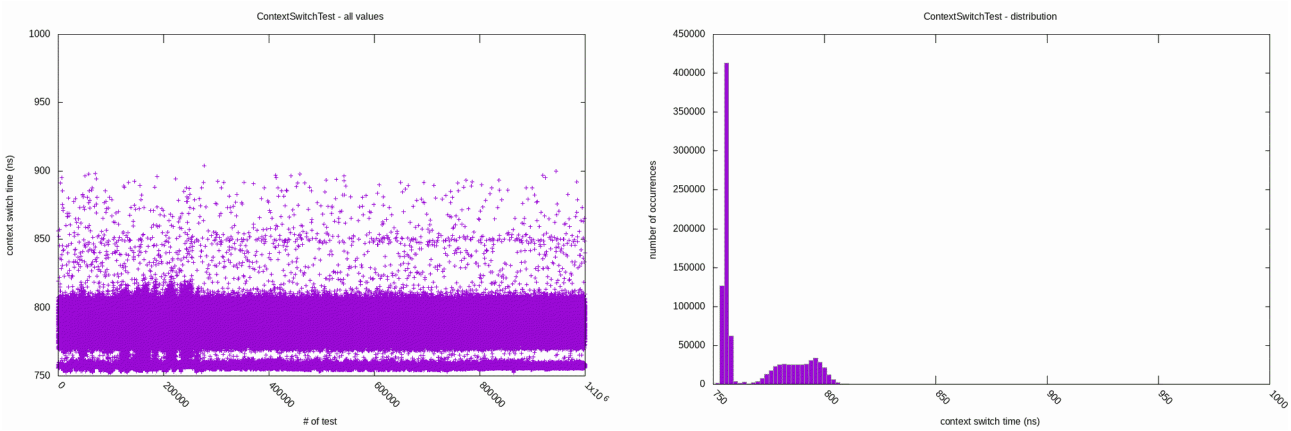

Abbildung 21: Kontextwechsel – custom unter Last

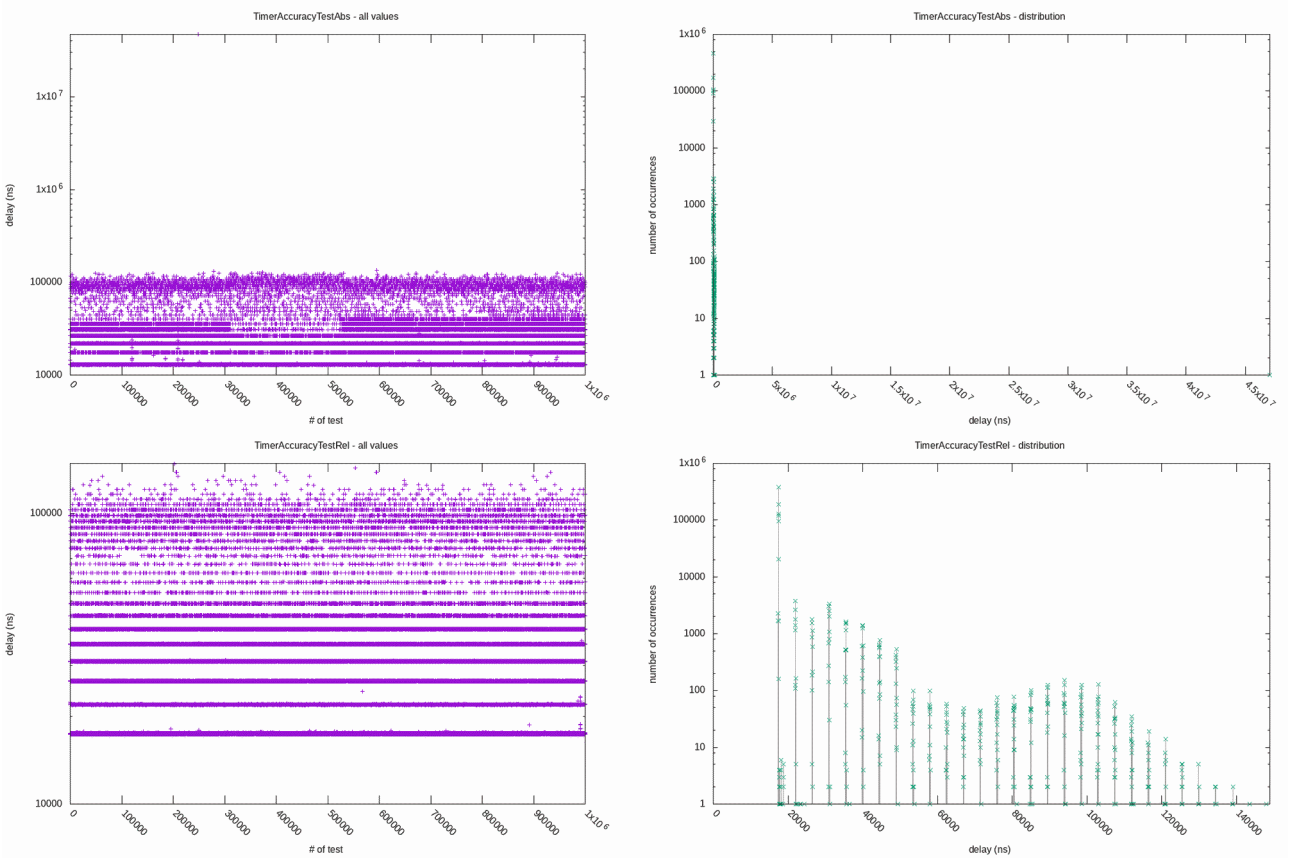

Abbildung 22: Timer-Genauigkeit – custom unter Last

8.8. Ergebnisse SMI-Test

```
SMITest
Test scheduled for  1800.000s and  1000000 interrupts.
Test finished after 1800.471s and     1801 interrupts.
The test was finished because the maximum execution time was reached.
Average timings:     1.000 interrupts/s =   60.018 interrupts/min
Average interval:    1.000 s/interrupt  =  999.706 ms/interrupt

Analysing interrupt durations (times in µs):
1801 datapoints
    Column        Min        Avg        Max      StDev
  duration      22.00      26.12      33.00       0.87
```

Listing 8: Zusammenfassung SMI-Test

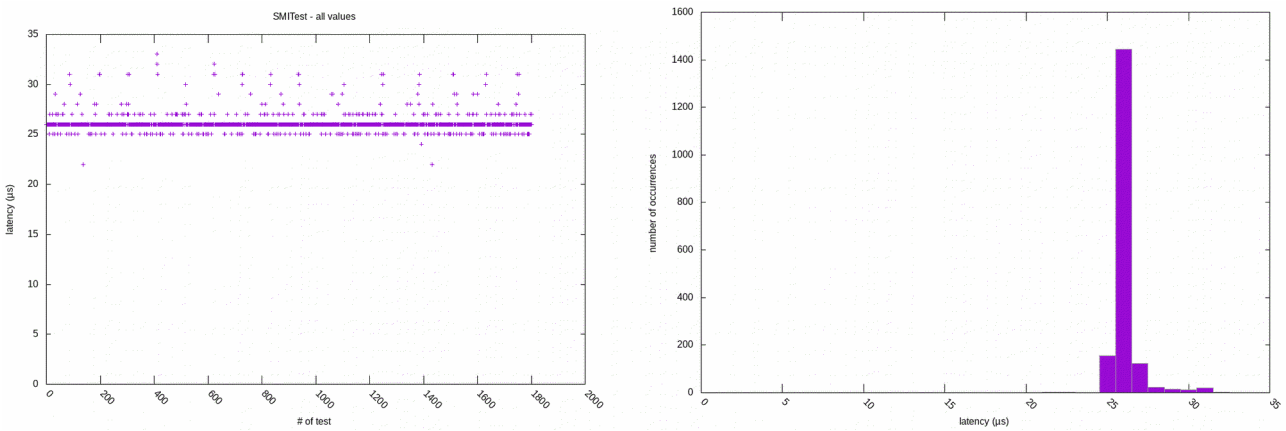

Abbildung 23: SMI-Test