Jül-4302

Jülich Supercomputing Centre (JSC)

Entwicklung eines Werkzeugs zur Analyse des OS-Jitter Effekts auf High-Performance Cluster-Systemen

Stephan Graf

Mitglied der Helmholtz-Gemeinschaft

JÜLICH
FORSCHUNGSZENTRUM

Entwicklung eines Werkzeugs zur Analyse des OS-Jitter Effekts auf High-Performance Cluster-Systemen

Stephan Graf

Berichte des Forschungszentrums Jülich; 4302
ISSN 0944-2952
Jülich Supercomputing Centre (JSC)
Jül-4302

Vollständig frei verfügbar im Internet auf dem Jülicher Open Access Server (JUWEL) unter
http://www.fz-juelich.de/zb/juwel

Zu beziehen durch: Forschungszentrum Jülich GmbH · Zentralbibliothek, Verlag
D-52425 Jülich · Bundesrepublik Deutschland
☎ 02461 61-5220 · Telefax: 02461 61-6103 · e-mail: zb-publikation@fz-juelich.de

Die vorliegende Masterarbeit wurde in Zusammenarbeit mit der Forschungszentrum Jülich GmbH, Jülich Supercomputing Centre angefertigt.

Diese Masterarbeit wurde betreut von:
Prof. Dr. rer. nat. Martin Reißel
Dr. Norbert Eicker

Diese Arbeit ist von mir selbstständig angefertigt und verfasst worden. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

Jülich, Januar 2009

Entwicklung eines Werkzeugs zur Analyse des OS-Jitter Effekts auf High-Performance Cluster-Systemen

Auf dem Gebiet des Höchstleistungsrechnens führt die aktuelle Entwicklung zu Cluster-Systemen, die aus mehreren Tausend Knoten bestehen. Jeder dieser Knoten ist mit vier oder mehr Prozessoren ausgestattet. Um einen Rechner zu betreiben, läuft auf jedem Knoten ein eigenständiges, von den anderen unabhängiges Betriebssystem, welches heutzutage ein UNIX-Derivat des jeweiligen Herstellers, meist jedoch ein optimiertes Linux, ist.

Bei Systemen dieser Art wird in der Literatur oft von einem Leistungsverlust bei steigender Anzahl genutzter Knoten im Zusammenhang mit kollektiven Operationen bei parallelen Anwendungen berichtet. Ein Grund dafür liegt meistens in *Amdahl's Law*. Dieses besagt, dass der nicht zu parallelisierende Anteil einer Applikation dafür sorgt, dass ab einer gewissen Prozessorzahl eine weitere Steigerung der Anzahl zu keinem weiteren signifikanten Leistungsgewinn führt.

Eine weitere Ursache, die für den Leistungsverlust verantwortlich gemacht wird, liegt in Betriebssystem-Aktivitäten. Diese müssen sich die Ressourcen mit der Applikation teilen. Da diese Aktivitäten nicht auf allen Knoten synchron auftreten, kann es bei kollektiven Operationen innerhalb der Applikation zu unnötigen Wartezeiten kommen. Dieser Effekt wird als *OS-Jitter* bezeichnet.

In dieser Arbeit werden zunächst die technischen Grundlagen erklärt. Als Schwerpunkt wird das Linux Betriebssystem betrachtet, speziell dessen aktuelle Weiterentwicklungen im Bereich des Prozess-Scheduling und der Echtzeitfähigkeit.

Im Anschluss wird der *OS-Jitter* beschrieben und dabei die Erkenntnisse aus der Literatur zu Ursache, Visualisierung und Einflussminderung dieses Effekts dargelegt.

Danach werden spezialisierte Benchmarks vorgestellt, die den *OS-Jitter* Effekt sichtbar machen sollen.

Im praktischen Teil der Arbeit werden Messungen, die auf dem *JuRoPA*-Testcluster des *Jülich Supercomputing Centre* durchgeführt wurden, analysiert und dargestellt. Es wird gezeigt, wie sich der *OS-Jitter* auf zwei verschiedenen Varianten des Linux Betriebssystems bemerkbar macht, und welche Ursachen zugrunde liegen.

Aus den gewonnenen Erkenntnissen wird ein Konzept zur Untersuchung des *OS-Jitter* auf einem beliebigen *HPC* System entwickelt. Es wird beschrieben, wie Störungen auf einem einzelnen Prozessor – unabhängig von den anderen – gemessen werden können. Ein zweites Werkzeug misst den Einfluss einer lokalen Störungen auf ein Programm, das viele Prozessoren parallel benutzt. Es wird gezeigt, wie mit Hilfe von der Frequenzanalyse und dem Dichteschätzer Informationen aus den Messdaten der beiden Benchmarks gewonnen werden können.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Auf dem Gebiet des Höchstleistungsrechnens führt die aktuelle Entwicklung zu Cluster-Systemen, die aus mehreren Tausend Knoten bestehen. Jeder dieser Knoten ist mit vier oder mehr Prozessoren ausgestattet. Damit die im System verteilte Rechenleistung möglichst effizient genutzt werden kann, müssen die Knoten sehr schnell miteinander kommunizieren können. Aus diesem Grund werden sie durch möglichst leistungsfähige Netzwerke wie z.B. InfiniBand miteinander verbunden.

Um einen Rechner zu betreiben, läuft auf jedem Knoten ein eigenständiges, von den anderen unabhängiges Betriebssystem, welches heutzutage ein UNIX Derivat des jeweiligen Herstellers – meist jedoch ein optimiertes Linux – ist.

Derzeit basieren ca. 90% der Systeme in der Top 500 Liste der schnellsten Rechner der Welt auf diesem Konzept. Ein alternativer Ansatz – wie er zum Beispiel bei der *IBM Blue Gene/P* realisiert wurde – ist, auf den einzelnen Knoten nur ein stark abgespecktes Betriebssystem laufen zu lassen. Die Arbeit für die Verwaltung des ganzen Systems wird dabei auf alle Knoten verteilt.

Bei Systemen dieser Art wird in der Literatur oft von einem Leistungsverlust bei steigender Anzahl genutzter Knoten im Zusammenhang mit kollektiven Operationen bei parallelen Anwendungen berichtet. Der Grund dafür liegt meistens in *Amdahl's Law*. Dieses besagt, dass der nicht zu parallelisierende Anteil einer Applikation dafür sorgt, dass ab einer gewissen Prozessorzahl eine weitere Steigerung der Anzahl zu keinem weiteren signifikanten Leistungsgewinn führt.

Eine weitere Ursache, die für den Leistungsverlust verantwortlich gemacht wird, sind Betriebssystem-Aktivitäten. Diese müssen sich die Resourcen mit der Applikation teilen. Da diese Aktivitäten nicht auf allen Knoten synchron auftreten, kann es bei kollektiven Operationen innerhalb der Applikation zu unnötigen Wartezeiten kommen. Dieser Effekt wird als *OS-Jitter* bezeichnet.

In dieser Arbeit werden zunächst die technischen Grundlagen erklärt. Als Schwerpunkt wird das Linux Betriebssystem betrachtet, speziell dessen aktuelle Weiterentwicklungen im Bereich des Prozess-Scheduling und der Echtzeitfähigkeit.

Im Anschluss wird der *OS-Jitter* beschrieben und dabei die Erkenntnisse aus der Literatur zu Ursache, Visualisierung und Einflussminderung dieses Effekts dargelegt.

Danach werden spezialisierte Benchmarks vorgestellt, die den *OS-Jitter* Effekt sichtbar machen sollen.

Im Kapitel 5 werden mathematische Verfahren diskutiert, welche zur Analyse mittels der Benchmarks gewonnenen Messdaten eingesetzt werden.

Darauf folgt im praktischen Teil die eigentliche Analyse und Darstellung der Messungen. Diese Messungen wurden auf dem *JuRoPA*-Testcluster des *Jülich Supercomputing*

Centre durchgeführt. Es wird gezeigt, wie sich der *OS-Jitter* auf zwei verschiedenen Varianten des Linux Betriebssystems bemerkbar macht, und welche Ursachen zugrunde liegen.

Aus den gewonnenen Erkenntnissen wird ein Konzept zur Untersuchung des *OS-Jitter* auf einem beliebigen *HPC* System entwickelt. Es wird beschrieben, wie Störungen auf einem einzelnen Prozessor – unabhängig von den anderen – gemessen werden können. Ein zweites Werkzeug misst den Einfluss einer lokalen Störungen auf ein Programm, das viele Prozessoren parallel benutzt. Es wird gezeigt, wie mit Hilfe von der Frequenzanalyse und dem Dichteschätzer Informationen aus den Messdaten der beiden Benchmarks gewonnen werden können. Zum Abschluss wird dieses Konzept auf den beiden derzeit im Forschungszentrum Jülich verfügbaren Supercomputer *JuMP* und *JuGene* demonstriert.

Damit hat man ein Werkzeug zur Hand, um den *OS-Jitter* auf einem System zu minimieren.

Kapitel 2

Prozess-Scheduling unter Linux

In dieser Arbeit wird der *OS-Jitter* auf einem *High Performance Cluster* untersucht. Auf dem überwiegenden Teil dieser Systeme – wie auch auf den Clustern des *Jülich Supercomputing Centre* – wird als Betriebssystem Linux eingesetzt. Deswegen werden in diesem Kapitel die Eigenschaften dieses Betriebssystems näher beleuchtet, speziell wie die Prozesse vom Betriebssystem verwaltet werden. Ob Linux auf einem Desktop PC oder auf einem Cluster-Knoten läuft macht erst einmal keinen Unterschied. Es handelt sich um dasselbe Betriebssystem, nur die verwendete Hardware und die Konfiguration des Linux Kernels sind anders.

Die Aufgabe eines *Operating Systems (OS)* ist es, die Hardwareressourcen des Rechners zu verwalten und den Programmen zur Verfügung zu stellen. Beim Booten des *OS* wird der Betriebssystemkern gestartet und die Kontrolle über das gesamte System an diesen Prozess übergeben.

Das besondere an Linux ist, dass die Quellen des Betriebssystemkerns offengelegt sind. Die aktuellste Version des Linux Kernels steht auf der Homepage `http://www.kernel.org` zur Verfügung. Damit besteht die Möglichkeit, Änderungen am Kernel vorzunehmen und an die persönlichen Bedürfnisse anzupassen. Werkzeuge zum Übersetzen des neuen Kernels sind in jeder Linux Distribution enthalten.

2.1 Prozess-Scheduling

Für jedes Programm, das auf einem Rechner aufgerufen wird, wird mindestens ein eigener Prozess gestartet. Arbeitet man an einem PC mit einem Standard Betriebssystem – z.B. Windows oder Linux, vermittelt das *Look & Feel* den Eindruck, dass alle Programme gleichzeitig, also echt parallel arbeiten. Tatsächlich ist die Anzahl an CPUs in einem einfachen PC jedoch sehr begrenzt. Es ist einleuchtend, dass in Wirklichkeit nur eine entsprechende Zahl an Prozessen wirklich parallel arbeiten können, trotz modernster Technologie wie *Simultaneous Multithreading (SMT)*. Bei *SMT* handelt es sich um die Fähigkeit eines Prozessors, eine Befehlspipeline mehrfach zu benutzen. Dazu müssen die notwendigen Register entsprechend oft vorhanden sein. Im Fall des 2-Wege *SMT* stehen für zwei *Threads*[1] Register zur Verfügung. Sobald der *Thread*, der gerade die Pipeline benutzt, auf Daten warten muss, kann der andere *Thread* die Pipeline verwenden. Das Betriebssystem sieht zwei oder mehr (logische) Prozessoren, obwohl es sich physikalisch um nur einen handelt. Bekanntester Vertreter dieser Technologie ist Intels *Hyper-Threading* [39].

[1] Threads sind Prozesse, die auf denselben Speicher zugreifen

Der Eindruck, dass alle Anwendungen auf einem Computer scheinbar parallel laufen, wird durch die Multitaskingfähigkeit des Betriebssystems realisiert. Ein Scheduler übernimmt dabei die Aufgabe, Prozesse den Prozessoren zuzuweisen. Er unterteilt dafür die Rechenzeit eines jeden Prozessors in Zeitscheiben, die er den einzelnen Prozessen zuteilt. Hat der aktuell rechnende Prozess seine Zeitscheibe verbraucht oder wartet er auf Informationen (z.B. Zugriff auf Daten der Festplatte), dann speichert der Scheduler die anstehenden Befehle mit den aktuellen Registerwerten ab, und teilt den jetzt „freien“ Prozessor einem anderen Prozess zu. Diesen Vorgang bezeichnet man als *Kontext-Wechsel* von Prozessen.

Des weiteren sind die Prozesse so aufgebaut, dass sie vom Scheduler unterbrochen werden können, um sie später wieder fortzusetzen. Der Scheduler muss deswegen *preemptiv* sein, d.h. er kann Prozesse anhalten, den Prozessor freigeben und einem anderen Prozess zuteilen.

Die Zeitscheiben sind so klein gewählt, dass sehr häufig Kontext-Wechsel stattfinden, und jedem Prozess innerhalb kurzer Zeit Rechenzeit zur Verfügung gestellt wird. Das vermittelt für das menschliche Empfinden den Eindruck einer Parallelität der Anwendungen.

Darüber hinaus gibt es Programme, die in einer bestimmten Geschwindigkeit arbeiten müssen, z.B. CD-Brennprogramme oder Echtzeitanwendungen wie das Erfassen von Messdaten. Diese Art von Prozess muss vom Scheduler bevorzugt behandelt werden. Erreicht wird dies durch eine Priorisierung beim Scheduling. Dazu werden alle Prozesse in eine sogenannte *Warteschlange* (*Runqueue*) einsortiert. Dort gibt es für jede Priorität eine eigene Liste. Der Scheduler bevorzugt Prozesse der höheren Prioritätenliste und teilt ihnen mehr Rechenzeit zu.

2.1.1 Vanilla Scheduler

Das zuvor beschriebene Verfahren beschreibt den *Vanilla Scheduler*, der bis Ende 2006 unter Linux verwendet worden ist, bevor er durch den $O(1)$ Scheduler von Ingo Molnar im Linux Kernel 2.6.13 ersetzt wurde. Der wiederrum wurde von seinem Autor nochmal komplett überarbeitet und wurde unter dem Namen *Complete Fair Scheduler (CFS)* (siehe Seite 6) in den Linux Kernel 2.6.23 integriert, der im Oktober 2007 veröffentlicht wurde. Alle Veröffentlichungen zum Thema *OS-Jitter*, die ihre Messungen auf einem Linux System durchgeführt haben, basieren noch auf dem „alten“ Scheduling-Prinzip. Interessant ist, ob mit dem aktuellen Scheduler hier eine Änderung beobachtet werden kann, und inwieweit die in diesen Publikationen angesprochenen Lösungen adaptiert werden können.

Beim *Vanilla Scheduler* wird die Prozessorzeit in Intervalle gleicher Größe unterteilt – man spricht von *Zeitscheiben* (*timeslices*). Zusätzlich existiert für jeden Prozessor eine *Warteschlange*, worin die Prozesse einsortiert werden. Sie besteht aus zwei Listen, einer mit den Prozessen, die noch ein Anrecht auf Rechenzeit haben (*active*), und einer anderen, in der sich diejenigen befinden, deren Zeitkonto momentan leer ist (*expired*).

Hat ein Prozess seine Rechenzeit verbraucht, wechselt er in die *expired*-Liste. Ist die *active*-Liste leer, tauschen die beiden Listen ihre Rolle und jeder Prozess erhält ein gefülltes Zeitkonto. Unter Umständen werden die interaktiven Prozesse anders behandelt, um bei Bedarf mehr oder weniger Zeit zur Verfügung gestellt zu bekommen. Das

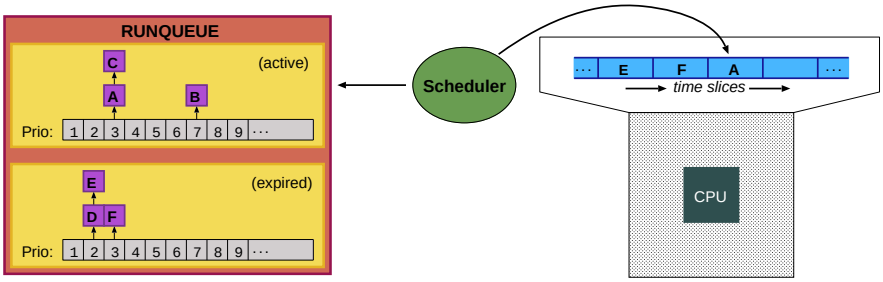


Abbildung 2.1: *Runqueue*-Prinzip des *Vanilla Scheduler*

Grundprinzip wird aber beibehalten.

Über Prioritätenlisten wird entschieden, welches Programm als nächstes auf dem Prozessor eine Zeitscheibe lang rechnen darf. Wenn die Zeit abgelaufen ist oder das Programm nicht mehr rechnen kann (weil es z.B. auf Daten wartet), wird der Scheduler wieder aktiv, passt die Prioritätenlisten an und teilt den Prozessor dem nächsten Programm zu.

Handelt es sich um ein Mehrprozessorsystem, existiert für jeden Prozessor eine eigene *Runqueue*, auf denen die Prozesse gleichmäßig verteilt werden (*Load Balancing*).

Die Größe der Zeitscheiben wird durch zwei Werte festgelegt:

Prozessor Takt: Geschwindigkeitsangabe der CPU

Es ist die Anzahl, wie viele Taktzyklen pro Sekunde ausgeführt werden können. (Frequenz, gemessen in Herz: $1Hz = \frac{1}{\mathrm{S}}$)

System Tick: Zeitgeber im Betriebssystem

Im Linux Kernel wird mit einer Frequenz zwischen $10Hz$ und $1000Hz$ ein *Time-Event* ausgelöst. Dieser wird bei der Kompilation des Kernels angegeben. Die Länge des Intervalls wird in der Literatur als *Jiffy* bezeichnet.

Standardmäßig hat unter Linux der *System Tick* eine Frequenz von $250Hz$. Das ergibt eine Zeitscheibe von $4ms$ Dauer. Besitzt der Prozessor z.B. eine Taktfrequenz von $3GHz$, stehen dem Prozess in einer Zeitscheibe 12.5 Mio. Taktzyklen zur Verfügung.

Interaktiv- oder Batchsystem

Die Frequenz des *System Tick* entscheidet über die Größe der Zeitscheibe. Je kürzer diese ist, desto häufiger gibt es einen Kontextwechsel der Prozesse. Der Prozess-Scheduler wird den Zustand des aktuellen Prozesses speichern und den als nächsten wartenden Prozess laden. Gerade bei interaktiven Arbeiten ist es wichtig, dass diese Prozesse sehr schnell bedient werden. Der Anwender vor dem Rechner hat eine graphische Oberfläche mit einer Vielzahl an Anwendungen, und alle sollen für sein Empfinden parallel arbeiten. Ein Nachteil ist, dass der organisatorische Aufwand, der beim Kontextwechsel entsteht, häufiger auftritt und damit die Nettoleistung des Systems verringert.

Kommt es dagegen auf einem System überwiegend auf den Durchsatz an, z.B. bei einem System im Batch-Betrieb, sind größere Zeitscheiben sinnvoll. Wie weiter oben erwähnt, muss die Größe der Zeitscheibe (des *Jiffy*) zur Kompilationszeit des Kernels festgelegt werden. Das macht eine Änderung sehr aufwendig.

2.1.2 Complete Fair Scheduler (CFS)

Der Linux Kernel wird aufgrund der sich immer wieder ändernden technischen Begebenheiten ständig weiterentwickelt. Man bedenke nur die Einführung der *Multicore*-Prozessoren, die ab 2005 die Einzelkernprozessoren im PC Bereich langsam abgelöst haben. Jährlich entwickeln die Hersteller neue Generationen ihrer CPUs. Weitere Entwicklungen sind die in den letzten Jahren aufgekommenen Stromsparfunktionen der Geräte oder die Anforderung, dass das Linux echtzeitfähig sein soll.

Auch neue Algorithmen, die den Kernel optimieren, finden immer wieder Einzug in die neueren Versionen.

Eine dieser Neuerungen war der Austausch des Schedule-Moduls. Im 2.6er Kernel wurde der *Vanilla Scheduler* durch den *Complete Fair Scheduler* ersetzt.

Das neue Verfahren nutzt keine festen Zeitscheiben im herkömmlichen Sinn wie bei seinem Vorgänger. Vielmehr ist die Länge der *time slices* nun variabel und wird dynamisch bestimmt. Auch ist diese Größe nicht mehr abhängig von der Prozessorfrequenz. Sie wird über Kernelparameter eingestellt, hat eine Granularität im Nanosekundenbereich und kann im laufenden System für jeden einzelnen Prozess geändert werden.

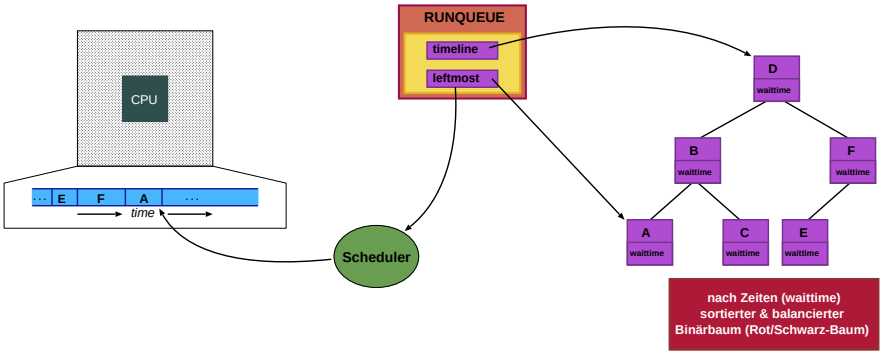


Abbildung 2.2: vereinfachtes Konzept des *Complete Fair Scheduler*

Die Grundidee des *CFS* ist, dass bei n laufenden gleichberechtigten Prozessen jeder genau $\frac{1}{n}$ der Rechenzeit des Prozessors erhält. Damit das funktioniert, wird für jeden Prozess gespeichert, wie lange er gewartet bzw. gearbeitet hat. Intern hat sich die Struktur der *Runqueues* für jeden Prozessor grundlegend geändert. Statt der zwei Prioritätenliste *active* und *expired* werden alle lauffähigen Prozesse in einen zeitorientierten Rot-Schwarz Baum einsortiert. Diese Art von Datenstruktur ist ein binärer Suchbaum, der ausbalanciert ist. Alle Operationen, die auf dieser Struktur definiert sind, benötigen höchstens $O(\log(n))$ Schritte (siehe [27]).

Im Falle von *CFS* wird für jeden Prozess die Wartezeit in dem Wert `wait_runtime` gespeichert. Der Prozess mit der längsten Zeit – der also den höchsten Anspruch bzw. das größte Bedürfnis nach Rechenzeit hat – steht am weitesten links im *Rot-Schwarz Baum*. Sowohl der Baumknoten dieses Prozesses als auch die Wurzel des Baumes werden in der *Runqueue* gespeichert. Das ermöglicht dem Scheduler direkt auf den Prozess mit dem höchsten Bedarf zuzugreifen und ihm den Prozessor zuzuteilen ($O(1)$). Die Größe der Zeitscheibe, die er jetzt rechnen darf, hängt von voreingestellten Parametern und der Anzahl der Prozesse in der *Runqueue* ab. Während dieser Prozess rechnet, wird sein

`wait_runtime`-Wert um die Zeit verringert, und bei allen anderen wird die Zeit zu ihrem Wert aufaddiert. Ändert sich die Reihenfolge in der *Runqueue* hält der Scheduler den laufenden Prozess an und teilt den Prozessor dem nächsten Prozess zu.

Insgesamt werden folgende Vorteile durch das neue Konzept erreicht:

- Der *Complete Fair Scheduler* hat eine Komplexität von $O(1)$ gegenüber $O(n)$ bei den älteren Verfahren.
- Der nächste Prozess ist direkt bekannt. Gerade bei einer großen Anzahl an Prozessen skaliert der neue Scheduler gut.
- Es werden keine komplizierten Heuristiken benötigt, um daraus auf die Interaktivität eines Prozesses zu schließen (eine Weiterentwicklung des *Vanilla Schedulers*). Grundsätzlich erhält jeder Prozess beim *CFS* den gleichen Anteil Rechenzeit. Nur die Länge der *Sleep Time* wird noch berücksichtigt. Die Prozesse mit längeren Schlafzeiten – typischerweise handelt es sich dabei um Interaktive – erhalten ein etwas geringeres Zeitkonto [20].

Eine detaillierte Darstellung ist in [20] zu finden. Dort wurde der *Complete Fair Scheduler* unter dem Linux Kernel 2.6.23 untersucht und neue Erweiterungen vorgestellt, die ab der Version 2.6.24 Einzug in den Linux Kernel gehalten haben.

2.2 Echtzeitsysteme

In vielen Bereichen ist man auf Echtzeitsysteme angewiesen, sei es bei Finanztransaktionen in Banken, Telekommunikation oder Produktionsanlagen in einer Fabrik. Bei allen Vorgängen werden Prozesse ausgelöst, die möglichst direkt bearbeitet werden müssen. Wenn das System eine bestimmte (kurze) Antwortzeit garantiert, spricht man von einem *Echtzeitsystem* (*Real Time*).

Zu den Anforderungen an ein entsprechendes Betriebssystem gehören, dass bestimmte Prozesse bevorzugt, diese sehr schnell den Prozessor zugeteilt bekommen und vor Unterbrechungen geschützt werden können.

Es existieren eine Reihe von Projekten, die das Linux System um die Echtzeitfähigkeit erweitern. Eine Liste ist auf den Internet Seiten der *Real Time Linux Foundation, Inc.* nachzulesen [40].

In Rahmen dieser Arbeit wird das *SuSE Linux Enterprise Real Time 10 (SLERT)* der Firma Novell eingesetzt. Dabei handelt es sich um eine *SuSE Linux Enterprise Server (SLES) 10.2* Version, die um den *RT PREEMPT* Patch erweitert worden ist. Dieses Patch ist von Luotao Fu und Robert Schwebel entwickelt worden. Dabei wird die *Real Time* Fähigkeit wie folgt realisiert: Zum einen kann mit dem *CPU Shielding* ein Prozessor vor Interrupts von Daemonen oder Ähnlichem geschützt werden. Nur Prozesse, die speziell markiert sind, dürfen den auf dem Prozessor laufenden Prozess unterbrechen.

Außerdem wird der *System Tick* mit einer Frequenz von $1000Hz$ ausgelöst. Damit erhält der Scheduler eine sehr feine Granularität und kann auf Ereignisse schneller reagieren.

Eine weitere Änderung ist, dass das Abarbeiten von Interrupts überarbeitet worden ist. Und zwar wird die gesamte Interrupt-Service Routine, die ansonsten zu längeren Latenzzeiten geführt hat, in zwei Teile unterteilt. In den *First Level Interrupt Handler (FLIH)*[2] und dem *Second Level Interrupt Handler (SLIH)*[3]. Wird ein Ereignis ausgelöst, müssen bestimmte Aktionen sofort ausgeführt werden, weil es sich z.B. um Hardware-kritische Informationen handelt. Die Programmierer sind bestrebt, diesen Teil möglichst zeiteffizient zu gestalten. Alle anderen Aktionen (im *SLIH*), die zu dem *Event* gehören, werden als normaler Prozess behandelt und vom Scheduler verwaltet.

Auf diese Weise wird die Zeit der unmittelbaren Unterbrechung durch Interrupts minimiert.

Außerdem werden *Real Time* Prozesse schneller einer CPU zugewiesen, weil alle anderen Prozesse (*Threads*) – auch die des Betriebssystems selber – unterbrechbar (*preemptiv*) sind.

Der Linux Kernel enthält seit Version 2.6.18 die Echtzeitfähigkeit im *Source Code*. Sie muss explizit aktiviert und der Kernel neu übersetzt werden.

2.3 Multiprozessor-Systeme

Aktuelle *High Performance Systeme* enthalten eine Vielzahl an CPUs, die im Verbund miteinander sehr komplexe Probleme bearbeiten sollen. Selbst in einfachen PCs werden schon so genannte *Dual-Core* oder *Quad-Core* Prozessoren verbaut. Das bedeutet, dass ein solches System eine CPU enthält, die zwei oder vier Prozessoren (*Dual-/Quad-Core*) enthält, die parallel arbeiten können. Im HPC Umfeld werden oft noch wesentlich mehr CPUs auf einem *Board* zusammengefasst. Dieses *Board* wird als Knoten (*Node*) bezeichnet.

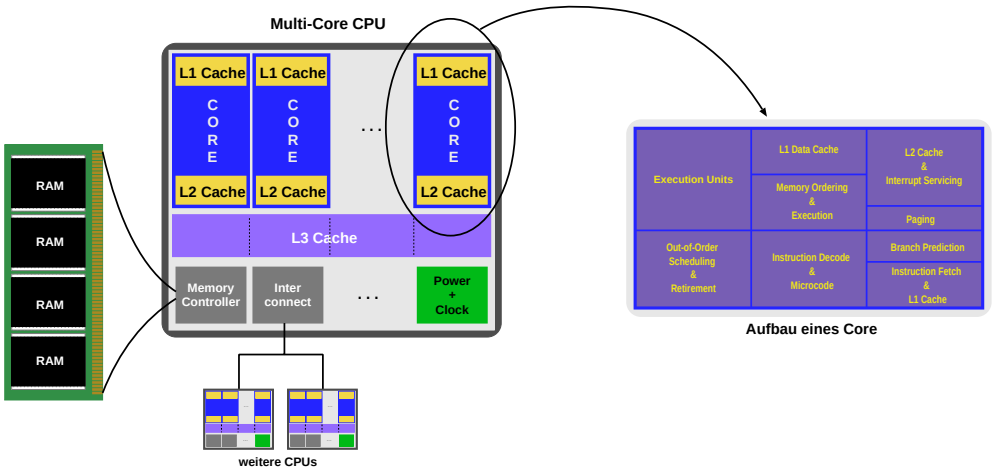


Abbildung 2.3: CPU mit mehreren Kernen

In Abbildung 2.3 ist der schematische Aufbau einer CPU mit mehreren Prozessoren

[2] andere Bezeichnungen: hard interrupt handlers, fast interrupt handlers, top-half of interrupt
[3] andere Bezeichnung: interrupt threads, slow interrupt handlers, bottom-half of interrupt

dargestellt. Das Bild ist angelehnt an den Aufbau der neuen *Nehalem* CPU von Intel. Je nach Variante besteht sie aus zwei, vier oder acht Prozessoren, die auf einem Sockel untergebracht sind.

Das Betriebssystem erkennt jeden Prozessor als selbstständiges Rechenwerk und wird beim Scheduling die Prozesse auf diese verteilen. Für die Programme besteht die Möglichkeit, statt einen Prozessor gleichzeitig mehrere zu nutzen, indem die Arbeit in mehrere Prozesse bzw. *Threads* unterteilt wird.

Die meistbenutzten Standards, mit denen Programme parallel auf mehreren Kernen arbeiten können, sind *OpenMP* und *MPI* (siehe nächstes Kapitel auf Seite 12).

Kapitel 3

Der *OS-Jitter* auf dem *High Performance Cluster*

In diesem Kapitel wird erklärt, was unter einem *High Performance Cluster (HPC)* zu verstehen ist und für welche Art von Programmen es eingesetzt wird. Ein Grund für den Leistungsverlust, der bei steigender Prozessorzahl bei parallelen Anwendungen festgestellt werden kann, wird als *OS-Jitter* bezeichnet. Dabei handelt es sich um Störungen, die durch das Betriebssystem entstehen. Es werden die Ursachen erläutert und Konzepte bzw. Realisierungen vorgestellt, die diesen Effekt verringern.

3.1 High Performance Cluster

Der Begriff *Cluster* leitet sich aus dem Englischen ab und bedeutet Schwarm, Gruppe oder Haufen. Ein *High Performance Cluster* ist im Prinzip eine Ansammlung von Standard-PCs, die miteinander vernetzt sind und aus Nutzersicht als ein Rechner angesprochen werden können. Anstelle von PCs spricht man von Knoten, auf denen jeweils ein autarkes Betriebssystem läuft. Die Knoten selbst bestehen aus mehreren Prozessoren bzw. Kernen. Die Rechenaufgaben – auch als Jobs bezeichnet – werden von einer *Cluster Management Software* auf die einzelnen Prozessoren verteilt. Um eine möglichst hohe Leistung zu erhalten, sind die Knoten über ein sehr schnelles und spezialisiertes Netzwerk miteinander verbunden.[36]

In dieser Arbeit bezeichnen Prozessoren und Kerne das Gleiche. Es ist jeweils eine Einheit auf einem Knoten, die einen Prozess abarbeiten kann.

Diese *HPC*-Systeme sind dafür vorgesehen, dass Programme ihre Aufgabe in viele Teilaufgaben aufteilen, um diese dann auf mehreren Prozessoren parallel zu verarbeiten mit dem Ziel, wesentlich schneller ein Ergebnis zu produzieren. Dabei muss der nicht parallelisierbare Anteil (α) des Algorithmus möglichst gering sein, denn nach *Amdahl's Law* gilt für die Beschleunigung (*Speedup* S) mit gegebener Prozessorzahl (p):

$$S(p) = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

Daraus folgt

$$S(p) \leq \lim_{p \to \infty} S(p) = \frac{1}{\alpha}$$

Das heißt, wie gut Programme auf einem *HPC*-System skalieren, hängt von dem nicht parallelisierbaren Anteil α des Algorithmus ab. Für jeden Wert ungleich Null wird das

Programm nicht beliebig skalieren, sondern bei sehr großen Prozessorzahlen (p) eine Sättigung $\left(\frac{1}{\alpha}\right)$ erfahren. Ist das der Fall, wird eine weitere Verdoppelung der Prozessorzahl nur noch wenige Prozent Beschleunigung erbringen. Nur beim so genannten *Farming*, bei dem die Aufgaben am Anfang auf die einzelnen Prozessoren verteilt werden und dort unabhängig zu Ende gerechnet werden können, ohne dass in irgendeiner Form Daten ausgetauscht werden müssen, gibt es so gut wie keine Beschränkung des *Speedups*. Nur zu Beginn müssen die Daten verteilt, und am Ende die Ergebnisse wieder eingesammelt werden. Diese Art von Anwendungen – bekanntestes Beispiel ist *SETI@home* – sind aber nicht das Ziel der *HPC*-Systeme.

Im Forschungszentrum Jülich verwendet man in diesem Zusammenhang den Begriff *High Performance Computing* und versteht da drunter den Einsatz von Programmen, die eng gekoppelte Verfahren nutzen. Dort ist das Ziel, diese Standardverfahren zu parallelisieren und möglichst effizient das *HPC*-System zu nutzen.

Der nicht zu parallelisierende Anteil α hängt aber nicht nur vom Algorithmus selbst ab. Hinzu kommt noch der Aufwand, der durch die Kommunikation zwischen den Prozessoren entsteht. Das wird beim Parallelisieren automatisch der Fall sein. Wie groß der Kommunikationsaufwand ist, hängt vom Algorithmus selber ab. Ein typisches Beispiel ist die Gebietszerlegung (*Domain Decomposition*). Mit steigender Prozessorzahl kann ein Gebiet immer weiter unterteilt werden, und jeder Prozessor wird ein Teilgebiet berechnen. Dafür müssen an den Rändern die Daten von den Nachbargebieten ausgetauscht werden. Das führt dazu, dass der Kommunikationsaufwand mit steigender Prozessorzahl (= feinere Gebietszerlegung) zunimmt. Eine weitere Nebenbedingung des Algorithmus in diesem Beispiel ist, dass alle beteiligten Prozesse zur gleichen Zeit (synchron) die Daten untereinander austauschen müssen. Damit das funktionieren kann, müssen alle Prozesse diesen Punkt erreichen. Meistens ist aber die Berechnung der einzelnen Gebiete unterschiedlich aufwendig. Das hat zur Folge, dass einige früher fertig werden und auf die anderen warten müssen. Durch gleichmäßiges Verteilen der Last können diese Zeiten minimiert werden.

3.2 Kommunikation im Cluster

Damit bei parallelen Programmen die Daten unter den einzelnen Prozessen ausgetauscht werden können, stehen verschiedene Werkzeuge für die Kommunikation zur Verfügung. Die Standardtypen *MPI* und *openMP* inklusive deren Kombination werden im Folgenden erläutert.

3.2.1 Message Passing Interface (MPI)

Bei diesem Konzept werden auf den einzelnen Prozessoren unabhängige Prozesse gestartet, die jeweils auf einen eigenen Speicher zugreifen. Die Prozesse können über das *Message Passing Interface* miteinander kommunizieren und damit Daten austauschen. Um die Wartezeit bei dieser Kommunikation möglichst gering zu halten, werden in den *HPC*-Systemen spezielle Netzwerke verbaut, die eine sehr geringe Latenz haben, wie z.B. Myrinet, InfiniBand oder 10G Ethernet. Außerdem erkennen die aktuellen Implementierungen des *MPI*, ob Prozesse einen gemeinsamen Speicher haben, und kommunizieren dann über *Shared Memory*.

In Programmen werden spezielle Befehle aus der *MPI*-Bibliothek angewendet, um Nachrichten zwischen den Prozessoren des ganzen Systems auszutauschen. Es gibt verschiedenen Arten der Kommunikation:

$1:1$ ein Prozess sendet, ein anderer empfängt (z.B. *MPI_Send/MPI_Recv*)

$1:n$ ein Prozess sendet an alle anderen (z.B. *MPI_Bcast*)

$n:1$ ein Prozess sammelt Informationen ein (z.B. *MPI_Gather*)

$n:n$ alle Prozesse tauschen Informationen aus (z.B. *MPI_Alltoall*)

Die einzelnen Kommunikationstypen stehen zusätzlich noch in verschiedenen Varianten zur Verfügung. In erster Linie unterscheiden sie sich in *blockierender* und *nicht blockierender* Kommunikation. Der Unterschied ist, dass bei der blockierenden Version alle Teilnehmer die Daten an den Zielprozess versenden und auf eine Empfangsbestätigung warten. Das kann zu Wartezeiten auf den einzelnen Prozessoren führen.

Bei der nicht blockierenden Kommunikation schickt der Sender seine Daten ab, wartet aber nicht auf eine Antwort des Empfängers, sondern fährt in seiner Berechnung fort. Damit wird die Kommunikation etwas flexibler gestaltet und führt im Idealfall zu einem beachtlichen Leistungsgewinn. Bei der Programmentwicklung muss jedoch darauf geachtet werden, dass der Datenaustausch korrekt abläuft. [13]

3.2.2 OpenMP

Haben zwei oder mehrere Prozessoren Zugriff auf denselben Speicher (*Shared Memory*), kann *OpenMP* verwendet werden. Auf den Prozessoren werden Unterprozesse gestartet – sogenannte *Threads* – , die keinen eigenen Speicher zugewiesen bekommen, sondern den des rufenden Prozesses mitbenutzen. Der Programmierer muss darauf achten, welcher *Thread* welche Daten ändert. Über so genannte `#pragma` Anweisungen im Quellcode werden dem Compiler Informationen mitgegeben, die es ihm ermöglichen, für bestimmte Abschnitte des Programms die Aufgaben zu unterteilen und dafür jeweils einen eigenen *Thread* zu starten.[3]

3.2.3 Hybrid Programmierung

Je nach Aufbau des *High Performance Systems* bietet sich auch eine Kombination von *OpenMP* und *MPI* Programmierung an. Wenn man z.B. einen Knoten mit vier *Quad-Core* CPUs hat, kann man innerhalb einer CPU den geteilten Speicher ausnutzen und darüber via *OpenMP* unter den Prozessen kommunizieren, während die Kommunikation für den Datenaustausch über CPU-Grenzen hinweg mittels *MPI* realisiert wird. Diese Art der Programmierung bezeichnet man als *Hybrid*.

Ziel ist es, den Mehraufwand der *MPI*-Kommunikation zu minimieren, indem ausgenutzt wird, dass vier Prozesse auf denselben Speicher zugreifen können. So wird insgesamt mehr Leistung auf dem System erreicht.

Wie oben beschrieben, erkennen die aktuellen *MPI*-Implementierungen inzwischen, wenn die Kommunikation über den geteilten Speicher möglich ist, und nutzen das aus.

3.3 Definition des *OS-Jitter*

Trotz Optimierung der Algorithmen wird auf den meisten *HPC*-Systemen nie die maximale Leistung erreicht. Das liegt zum einen an dem nicht zu parallelisierender Anteil α, der in der Regel immer vorhanden ist. Es wurden aber auch weitere Ursachen entdeckt. So haben F. Petrine, D. Kerbyson und S. Pakin 2003 bei einer Untersuchung auf dem Rechner *ASCI Q* in *Los Alamos National Laboratory* in den USA einen unerwarteten Leistungsverlust gemessen [24]. Bei diesem Rechner handelte es sich um ein 8192 Prozessor System, auf dem 4 Prozessoren immer einen Knoten mit eigenem Betriebssystem bilden. Die Autoren haben Messungen mit dem SAGE[1] Benchmark [19] durchgeführt und dabei festgestellt, dass die gemessene Leistung auf dem System nicht mit der theoretisch erwarteten Leistung übereinstimmt, wenn alle 4 Prozessoren pro Knoten benutzt werden.

Als Ursache haben die Autoren ein Systemrauschen (*OS-Jitter*) identifiziert, das durch das Betriebssystem entsteht.

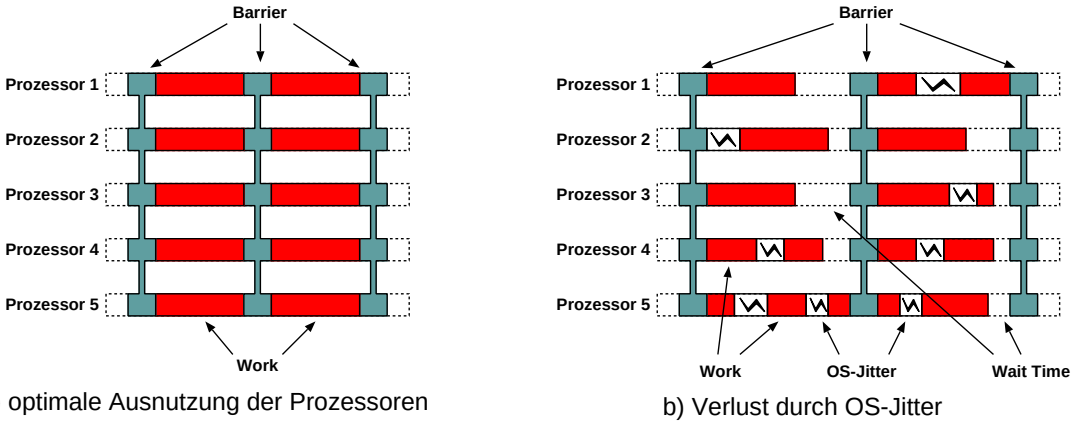


Abbildung 3.1: *OS-Jitter* auf Multiprozessor-Systemen bei parallelen Programmen mit globaler Kommunikation

Für diesen unerwarteten Leistungsverlust in Mehrprozessor-Systemen findet man in der Literatur immer wieder die Begriffe *System Noise* oder *OS-Jitter*. Störungen können in vielen Bereichen auftreten. Es gibt *Jitter* Effekte in der Netzwerkkommunikation, bei dem Zugriff auf Daten aus dem Hauptspeicher oder bei der Implementierung der kollektiven Operationen im *MPI*.

In dieser Arbeit wird nur der *OS-Jitter* betrachtet, der entsteht, weil Hardware oder Dienste (*Daemons*) ein Ereignis (*Event*) auslösen, das vom Betriebssystem abgearbeitet werden muss. Das führt dazu, dass der aktuell laufende Prozess unterbrochen wird, der Prozess-Scheduler den Prozessor einem anderen Prozess zuteilt und erst, wenn dieser abgearbeitet worden ist, kann die ursprüngliche Arbeit wieder fortgesetzt werden.

Eine schematische Illustration dieses Effekts ist in Abbildung 3.1 dargestellt. Ein Job wird auf N Prozessoren – in diesem Fall 5 – verteilt und ist so angelegt, dass auf jedem die gleiche Arbeit verrichtet wird. In regelmäßigen Abständen wird eine globale

[1] SAIC's Adaptive Grid hydrocode

Kommunikation über alle Prozessoren ausgeführt, hier realisiert mittels eines globalen *MPI_ Barrier*. Mit einem *Barrier* erzwingt der Programmierer, dass die beteiligten Prozesse erst dann weiterrechnen dürfen, wenn alle diesen Punkt erreicht haben. Im linken Bild ist der optimale Verlauf dargestellt. Alle Prozessoren rechnen ihren Job und sind gleichzeitig fertig. Es wird die bestmögliche Performance erreicht. Dagegen sieht man im rechten Bild, wie die Prozesse auf den einzelnen Prozessoren unterbrochen werden. Die Ausführung wird verzögert und der Job, der die längsten Unterbrechungen hatte, bestimmt, wie lange die anderen im *Barrier* warten müssen.

Es ist offensichtlich, dass mit zunehmender Anzahl beteiligter Prozesse, die Wahrscheinlichkeit wächst, dass einer durch einen *Event* unterbrochen wird. Ab einer bestimmten Anzahl an beteiligter Knoten, wenn jeweils kein Prozessor frei bleibt, wird der *OS-Jitter* die Jobausführung beeinträchtigen.

3.4 Applikationen mit fein- und grobgranularen Rechenschritten

Nicht jeder Job ist in gleicher Weise anfällig für den *OS-Jitter* Effekt. Es kommt im wesentlichen darauf an, wie häufig kommuniziert wird. Ist das nur selten der Fall, rechnen die einzelnen Prozesse relativ lange, bevor sie Ergebnisse mit den anderen austauschen. Man spricht von grobgranularen im Gegensatz zu feingranularen Programmen. Da die Störungen auf einem gut konfigurierten System, das keine technischen Schwierigkeiten hat, gleichverteilt sein sollten, heben sie sich bei unabhängigen Berechnungen mit langen Sequenzen gegenseitig auf. Erfordert der Algorithmus dagegen sehr häufig einen Datenaustausch, also ist er feingranular, dann führt jede Störung eines einzelnen Prozesses zu Wartezeiten bei allen anderen beteiligten Prozessen.

3.5 Gang Scheduling

Ein erster Ansatz zur Minimierung des *OS-Jitter* ist, die Prozesse in Gruppen einzuteilen, z.B. in *BATCH* und *NORMAL*. Man gibt Zeitintervalle vor, in denen nur die *BATCH*-Jobs rechnen dürfen, und andere Intervalle für Betriebssystemaufgaben, um alle ankommenden *Events* abzuarbeiten bzw. Arbeiten von Diensten zu erledigen. Damit alle Knoten das gleichzeitig erledigen, müssen die Intervalle zeitlich synchronisiert werden. Paul Terry beschreibt in seinem Artikel [31], wie er mit Hilfe einer neuen *Scheduling Policy* für den Linux Scheduler die Prozesse in zwei Gruppen einteilte, ein Zeitfenster von 128 Zeitscheiben definierte und entsprechend zuteilte: 120 für die eigentliche Arbeit, und die restlichen 8 für das so genannte *housekeeping*, also Arbeiten des Betriebssystems. Der Linux Kernel hat zu dem Zeitpunkt, als Paul Terry sein Scheduling Verfahren entwickelt hat, einen Linux Kernel mit dem „alten“ *Vanilla Scheduler* verwendet.

Sein Ergebnis ist, dass – abhängig von der Granularität des Jobs und der Stärke des Rauschen – ein entsprechend starker Leistungsgewinn erzielt wird.

Man bezeichnet dieses globale Synchronisieren der lokalen Scheduler als *Gang Scheduling*.

3.6 Zeitsynchronisation der Knoten

Eine wichtige Voraussetzung, um das eben beschriebene *Gang Scheduling* auf einem Clustersystem einzusetzen, bzw. um eine zuverlässige Kommunikation unter den Prozessen zu gewährleisten, ist, dass die einzelnen Knoten die gleiche Zeit haben. Ziel ist es, die Zeitscheiben möglichst synchron auf dem ganzen System zu starten. Bedenkt man, dass deren Länge im Standard Linux $4ms$ beträgt, ist klar, dass ein Zeitversatz von $100\mu s$ oder weniger notwendig ist.

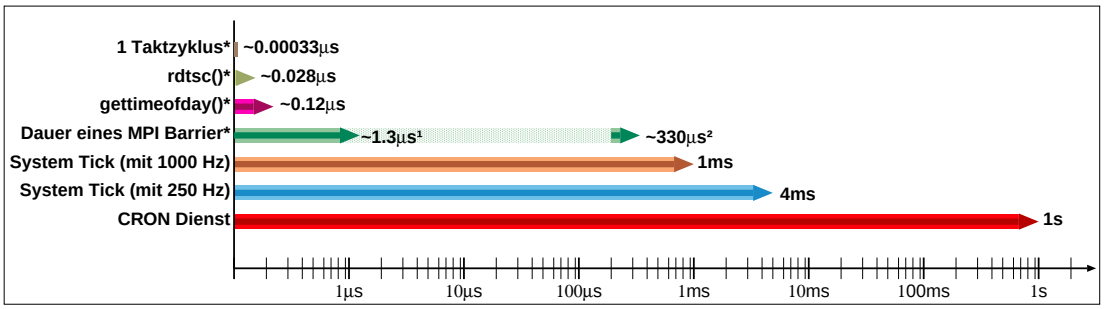


Abbildung 3.2: Zeiten für verschiedene Systemgrößen: Die mit „*“ ausgezeichneten Funktionen sind systemabhängig. Hier liegt ein 3 GHz Prozessor zu Grunde. Bei der Messung der *MPI-Barrier* Zeit ist bei [1] die Zeit für einen *MPI-Barrier* auf einem Knoten und bei [2] die Zeit für einen *MPI-Barrier* auf insgesamt 15 Knoten gemessen worden. Sie fanden auf dem *JuRoPA* Testsystem statt, dass in Kapitel 6 beschrieben wird. (*Intel MPI Benchmark [17]*)

Schon Aufgrund von kleinsten Varianzen in den elektronischen Bauteilen wird es nicht möglich sein, dass auf allen Knoten immer exakt die gleiche Zeit sein wird. Selbst wenn alle Knoten genau zur selben Zeit ein Signal erhalten, nach dem sie sich ausrichten können, läuft darauf jeweils die lokale Uhr und es entstehen kleine Unterschiede in den Zeiten. Diese Zeitdifferenzen dürfen sich höchstens in der Größenordnung von $100\mu s$ bewegen, damit sie für das *Gang Scheduling* nicht störend wirken. Notwendige Voraussetzung für das *Gang Scheduling* ist, dass die verwendeten Zeitscheiben für alle Prozessoren gleich groß sind und sich zeitlich möglichst überdecken. Die Länge der Zeitscheiben beträgt meistens nicht weniger als $1ms$ und kann bis auf $100ms$ oder noch größer ausgedehnt werden. In dem Fall könnte die Zeitdifferenz dementsprechend größer sein.

3.6.1 *Network Time Protokoll (NTP)*

Ein bewährtes Verfahren, die Zeit eines Rechners an die eines anderen anzupassen, ist das *Network Time Protocol (NTP)*. Dabei gibt es zwei Varianten der Konfiguration.

Bei der einen wird auf dem System der NTP-Daemon gestartet und so konfiguriert, dass er sich in regelmäßigen Abständen – die Standardeinstellung ist alle 30 Sekunden – an einen Zeit-Server wendet, um die aktuelle Zeit zu erhalten. Gleichzeitig wird bei Anfrage und Antwort die Latenz des Netzwerkes bestimmt und mit in die Zeitbestimmung eingerechnet. Da zwischen den Synchronisationspunkten der Rechner selbst anhand des

eigenen Zeitgebers die Zeit bestimmt, laufen dessen Uhr und die des Zeit-Servers wahrscheinlich auseinander. Der NTP-Dienst merkt sich aber bei jedem Mal die Änderung und führt eine Anpassung durch, so dass eine höhere Genauigkeit erreicht wird.

Alternativ kann NTP so konfiguriert werden, dass ein *Broadcast* Server proaktiv Zeitsignale aussendet, während die Knoten des Systems als Klienten passiv auf dessen Signale warten. Dieses Verfahren kann per *Multicast* realisiert werden. Beim *Multicast*-Verfahren werden bei n Teilnehmern nicht n Nachrichten vom Sender erzeugt, sondern die Netzwerk-Switches vervielfältigen bei Bedarf die Nachrichten auf dem Weg zum Ziel, wenn diese das *Internet Group Management Protocol (IGMP)* unterstützen. Der Vorteil ist, dass das Netzwerk weniger belastet wird und bei sehr kleinen Latenzzeiten (die bei dieser Art der Kommunikation nicht mehr bestimmt werden können) eine ähnlich genaue Zeit erreicht wird. [5, 6]

Typischerweise wird in einem *WAN*[2] per *Network Time Protocol* eine Genauigkeit von $10ms$ bis $100ms$ erreicht. Letztendlich hängt es stark von der Latenz des Netzwerkes ab.

In einem Clusterverbund wird ein wesentlich schnelleres Netzwerk eingesetzt. Dort kann mittels *NTP* eine Zeitgenauigkeit von $1ms$ oder noch besser erreicht werden.

Eine optimierte Variante für das *Broadcast*-Verfahren von NTP ist das *Reference-Broadcast Synchronization (RBS)*, das in [8] beschrieben wird. Die größte Einschränkung des *RBS* ist, dass ein eigener *Broadcast*-Kanal benötigt wird. Bei Messungen wurde im Vergleich zu einer NTP Lösung ein ca. 8-fach besseres Ergebnis erreicht. Um dabei gleiche Voraussetzungen für *RBS* zu schaffen, wurde die Kommunikation mittels UDP[3] realisiert und auf den Knoten ein Daemon installiert, der im *User Space* lief, und die *Broadcast*-Daten entgegennahm.

Die Firma *Cray Inc.* ging bei ihrem System *Cray XD1* einen anderen Weg. In diesem System wird als Netzwerk das proprietäre *RapidArray* verwendet. Eine Eigenschaft dieses Netzwerkes ist, die Zeitsynchronisation der Knoten zu übernehmen. Die damit erreichte Genauigkeit ist besser als $1\mu s$ ([16]).

Ein anderes Prinzip verwenden die Firma *IBM* auf ihren *BlueGene* Systemen. Es gibt einen zentralen Zeitgeber (*single source clock*), der die Zeit an alle Knoten über eine baumartige Kommunikation (*clock tree*) verteilt ([28]).

Welches Konzept am Besten geeignet ist, kann allgemein nicht beantwortet werden. Je nachdem bringt das im Computercluster verwendete Netzwerk Eigenschaften mit, die eine Realisierung nach dem Vorbild der Firma *Cray Inc.* ermöglichen. Alternativ kann auch eine individuelle Lösung konzipiert werden, die an das Problem angepasst ist und das schnelle Netzwerk ausnutzt.

3.7 Ursachen des *OS-Jitter*

Wie bereits erwähnt, gibt es zwei Klassen von Betriebssystemkomponenten, die als Ursache für das Rauschen im System erkannt wurden.

Zur einen Klasse gehören Dienste des Betriebssystems, die immer wieder aufwachen, um anfallende Arbeit zu erledigen, und sich dann wieder schlafen legen. Typische Dienste

[2] Wide Area Network
[3] User Datagram Protocol

sind der *System Logger (syslog)*, der *Network Time Protocol Daemon (ntpd)*, *Network Filesystem Daemon (nfsd)* oder der *Kernel swap daemon (kswapd)*[4]. In einem *High Performance Cluster* kommen noch Dienste für das Clustermanagement hinzu.

In die zweite Klasse fallen Hardware-Interrupts, die vom Betriebssystem abgearbeitet werden müssen. Auf welche Weise das geschieht, wurde schon im Abschnitt über die Echtzeitfähigkeit auf Seite 8 beschrieben.

Die häufigsten Interrupts sind Netzwerk-*Events* und der periodische *System Tick*.

Im ersten Fall handelt es sich um Ereignisse, die bei der Kommunikation anfallen. Ursache kann die Anwendung selber sein, sei es, dass eine Nachricht kommt, ein Puffer voll ist und geleert werden muss oder ähnliches. Auch verschiedene Dienste des Betriebssystems werden über das Netzwerk kommunizieren und lösen so ein Ereignis aus.

Dagegen ist der *System Tick* ein interner Zeitgeber, der im Linux Kernel voreingestellt ist (zur Kompilationzeit). Wie in Abschnitt über den Linux Scheduler auf Seite 4 beschrieben, kann dieser auf $10Hz$ bis $1000Hz$ eingestellt werden, was gleichbedeutend ist mit einem Zeitintervall zwischen den einzelnen *System Ticks* Ereignissen von $100ms$ bis hinunter zu $1ms$.

Laut einer Studie von *IBM* ist der *System Tick* die Hauptursache für den *OS-Jitter* auf einem Linux System und hat einen Anteil von ca. 60% am gesamten *Jitter* Effekt (Tabelle 1 in [4]). Der Rest verteilt sich auf Netzwerk-Interrupts und die verschiedenen Dienste, die auf dem System laufen.

3.7.1 Der *System Tick*

Fester Bestandteilt des Linux Kernels ist der *System Tick*. Er dient dem Prozess Scheduler als Signalgeber, der mit einer bestimmten Frequenz auftritt.

Im Abschnitt über den *Vanilla Scheduler* auf Seite 4 ist beschrieben, dass über den *System Tick* die Größe der Zeitscheibe definiert wird. Der Wert der Frequenz war bis vor einigen Jahren noch auf $100Hz$ eingestellt. Doch gerade im Desktop Einsatz ist in den letzten Jahren die Frequenz auf $250Hz$ vergrößert worden, weil für Multimedia-Anwendungen $100Hz$ nicht ausreichten. So sind z.B. in Video-Streams zu viele Frames verloren gegangen.

Um dem Linux Kernel *Echtzeitfähigkeit* zu geben, wird sogar eine noch feinere Auflösung vom $1000Hz$ eingestellt.

Durch die Einführung des *Complete Fair Scheduler* ist dieser Zusammenhang nicht mehr direkt gegeben. Die Größe der Zeitscheiben ist variabel und kann zur Laufzeit geändert werden.

Eine weitere Änderung hat die Aufnahme des *Tickless-Patch* in den Linux Kernel 2.6.21 bewirkt, der den statischen *System Tick* gegen ein dynamisches Konzept ersetzt hat [12]. Vorher wurde der *System Tick* bei einer Frequenz von $250Hz$ konstant alle $4ms$ ausgelöst. Jedes Mal muss das Betriebssystem aktiv werden und dieses Ereignis abarbeiten. Das widerspricht dem Ziel, möglichst viel Strom zu sparen, wenn das System nichts zu tun hat (*Idle Time*). Eine andere Motivation kommt aus dem Bereich der Rechner-Virtualisierung. Wenn dort jedes Gastsystem diesen *Event* auslöst, hängt die Belastung der realen CPUs von der Anzahl laufender virtueller Systeme ab.

[4] Speicher Management

Ziel dieser Entwicklungen ist es, das System *Tickless* zu bekommen, d.h. der Kernel sollte möglichst gar keinen *System Tick* mehr auslösen. Der *Tickless-Patch* sorgt dafür, dass sich der *System Tick* der aktuellen Situation anpasst. Auf der einen Seite kann die Frequenz der *Ticks* dynamisch erhöht werden, was gerade von den Echtzeitanwendungen gefordert wird. Auf der anderen Seite wird, wenn die CPU im *Idle*-Zustand ist, der *System Tick* so programmiert, dass er erst dann ausgelöst wird, wenn wieder ein Prozess weiterrechnen kann. So wird die CPU in diesem Zeitintervall nicht unnötig geweckt. Man spricht in diesem Zusammenhang vom *High Resolution Timer (hrTimer)* und den *dynticks*.

Die Idee dazu ist nicht neu. Bereits im Jahr 2005 wurde in einem Artikel das Konzept für den *smart timer* vorgestellt [32]. Es ging um die Analyse des *OS-Jitter* (*noise*) auf einem System, dessen Hauptursache – dem *System Tick* – und wie dieses Problem zu lösen ist. Es wurde ein theoretisches Konzept vorgestellt: Der *smart timer* sollte die folgenden Bedingungen erfüllen:

1. Für die exakte Zeit sorgen.

2. Die Anzahl der *System Ticks* reduzieren, indem direkt aufeinander folgende Ereignisse zusammengehängt werden.

3. Den *System Tick* nicht periodisch ausführen und dadurch deren Anzahl verringern.

Die Entwickler des Linux Kernels verfolgen das Ziel, den *System Tick* letztendlich komplett aus dem Betriebssystem verschwinden zu lassen.

3.8 Messungen des *OS-Jitter*

Viele Artikel, die sich mit der Untersuchung von *OS-Jitter* beschäftigen, beschreiben, wie die Autoren die Störungen im System gemessen und analysiert haben. Es wurden spezialisierte Benchmarks – so genannte *Micro-Benchmark* – und Anwendungs-Benchmark zum Messen der Systemleistung eingesetzt. [14, 32, 4, 24]

Um der Ursache noch genauer auf den Grund zu gehen, werden Analysewerkzeuge eingesetzt, die Ereignisse im Kernel protokollieren und so Rückschlüsse auf die Ursache der Störung gezogen.

3.8.1 Micro-Benchmarks

Micro-Benchmarks ermöglichen es, den im System anfallenden *OS-Jitter* zu messen. Mit ihnen kann man feststellen, ob Maßnahmen zur Reduzierung des *OS-Jitter*, z.B. das Deaktivieren einzelner Dienste, eine Verbesserung bringen.

Ziel dieser Arbeit ist es, ein Konzept für ein Werkzeug zu entwickeln, mit dem der *OS-Jitter* Effekt auf einem *HPC*-System untersucht werden kann. Störungen treten lokal auf einem Prozessor auf und können dann bei parallelen Programmen während der Kommunikation zu unerwünschten Wartezeiten bei den beteiligten Prozessen führen.

Aus diesem Grund können die *Micro-Benchmarks* in zwei Klassen einsortiert werden:

1-Prozessor Benchmarks: Diese Art der *Micro-Benchmarks* untersuchen nur den lokalen *OS-Jitter*, der auf einem Prozessor entsteht, ohne mit anderen zu kommunizieren. Vertreter diesen Typs sind der *Fix Time Quantum* Benchmark und der *Selfish Detour* Benchmark, die in Abschnitt 4 auf Seite 27 vorgestellt werden.

n-Prozessor Benchmarks Bei diesem Typ von *Micro-Benchmarks* wird der *OS-Jitter* bei parallelen Programmen auf Cluster-Systemen untersucht. Es werden n Prozessoren verwendet, die knotenübergreifend kommunizieren müssen. Das im Rahmen dieser Arbeit geschrieben Programm *MPI-Jitter* (siehe Abschnitt 4 auf Seite 27), das sich im Aufbau an das Verfahren aus Abbildung 3.3 anlehnt, ist gehört zu dieser Klasse.

Micro-Benchmark — läuft auf jeder CPU

t_{old} = get_time()
compute phase \\ always the same specified work
$t_{compute}$ = get_time()
Barrier \\ synchronise processors
t_{all} = get_time()
compute time = $t_{compute} - t_{old}$
complete time = $t_{all} - t_{old}$
$t_{old} = t_{all}$
measure count reached

Abbildung 3.3: typischer n-Prozessor *Micro-Benchmark*

Ein *Micro-Benchmark* ist in der Regel ein sehr spezialisiertes Programm, das nur auf einen speziellen Effekt abzielt und ihn sichtbar macht. Es ist kein Benchmark, der direkt etwas über die Leistungsfähigkeit des Systems aussagt, wie z.B. der hier vorgestellte Anwendungs-Benchmark *LINPACK* (siehe Abschnitt 3.8.2 auf Seite 22).

In diesem Fall verwenden die *Micro-Benchmarks* einen einfachen Algorithmus, der speziell darauf abzielt, den *OS-Jitter* sichtbar zu machen. In Abbildung 3.3 ist der typische Aufbau eines n-Prozessor *Micro-Benchmarks* abgebildet. Das Programm läuft parallel auf einer beliebigen Anzahl von Prozessoren. Jede einzelne Messung unterteilt

sich in zwei Phasen. In der *Compute Phase* muss auf allen Prozessoren unabhängig exakt die gleiche Menge an Rechenoperationen ausgeführt werden. Danach erreichen sie die *Wait Phase*, bei der alle Prozesse mit Hilfe eines *Barrier* wieder synchronisiert werden. Der *Barrier* steht dabei für eine globale, blockierende Kommunikation zwischen den Prozessen. In jedem Iterationsschritt werden insgesamt zwei Zeiten gemessen. Die erste Zeit entspricht der Brutto-Rechenzeit für den Rechenaufwand auf jedem Prozessor inklusive der eventuell aufgetretenen Unterbrechungen (`compute time`). Die zweite Zeit beschreibt den Einfluss des lokalen *OS-Jitter* der einzelnen Prozesse auf die globale Kommunikation (`complete time`).

Konnten alle Prozesse ihren Prozessor exklusiv nutzen, sind die Zeiten – bei gleicher Hardware – für die *Compute Phase* auf allen Prozessoren gleich, da die Rechnung und damit die Anzahl der Operationen deterministisch vorgegeben und gleich groß ist. Über die Größe der *Compute Phase* kann die Granularität von Prozessen simuliert werden.

Danach ist die Messung für einen Schleifendurchlauf abgeschlossen und die Zeiten müssen noch ausgewertet werden. Der Anwender entscheidet, wie viele Messungen durchgeführt werden. Die Messdaten können dann auf verschiedene Arten statistisch ausgewertet werden.

Hinweis zum *Barrier*

Die eigentliche Aufgabe eines *Barrier* besteht darin, die beteiligten Prozesse zu synchronisieren. Das heißt aber nicht, dass direkt nach dem *Barrier* alle Prozesse exakt gleich starten, sondern nur, dass alle Prozesse einen gemeinsamen Synchronisationspunkt erreicht haben.

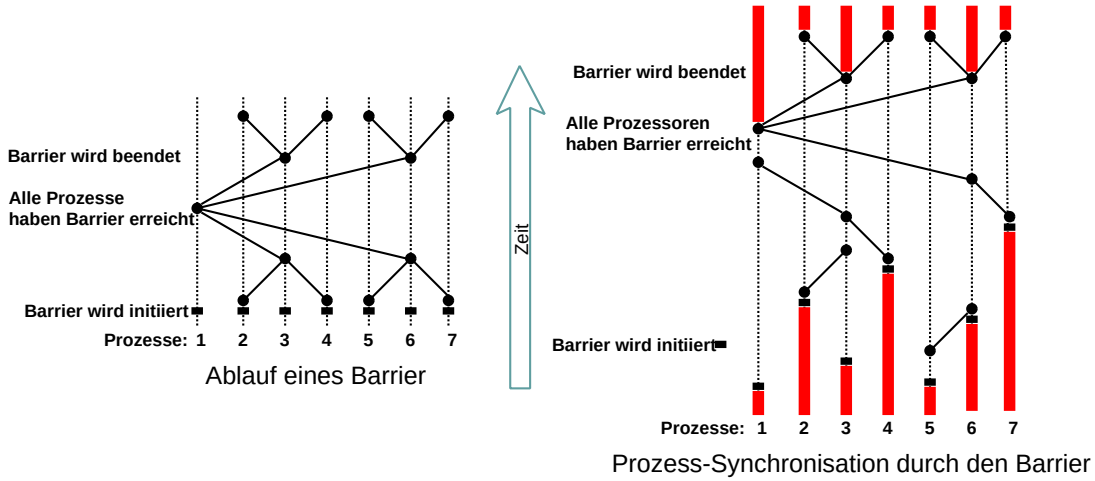


Abbildung 3.4: Kommunikation im *Barrier*: Realisierung mithilfe einer *Binär-Tree* Kommunikation

In Abbildung 3.4 ist eine typische Realisierung für den *Barrier* dargestellt. Die Prozesse tauschen über eine Binärbaum-Struktur ihre Informationen aus. Wie man erkennt, starten die Prozesse leicht zeitversetzt nach der Synchronisation. Die Differenz ist abhängig von der Latenz des verwendeten Netzwerks. In Abbildung 3.2 auf Seite 16 sind

die Zeiten für die Dauer eines *Barrier* eingezeichnet. In diesem Fall wurden die Messungen auf dem *JuRoPA* Testsystem durchgeführt, das später im Kapitel 6 beschrieben wird. Der Zeitaufwand für den *Barrier* variiert dabei von ca. 1,3μs bei Prozessen, die sich nur auf demselben Knoten befinden, und ca. 330μs, wenn 60 Prozessoren auf 15 Knoten beteiligt sind.

3.8.2 Anwendungs-Benchmarks

Ob Änderungen am System eine Wirkung auf ein Programm zeigen, hängt stark von dessen Algorithmus ab, ob er auf viel Kommunikation angewiesen ist und ob diese blockierend abläuft. Im Prinzip ist dabei entscheidend, wie fein granular der Jobablauf ist. In der Literatur werden verschiedene Benchmarks genutzt, um die Leistung eines *HPC*-Systems zu messen. Einer der bekanntesten ist der *LINPACK*-Benchmark. Dabei handelt es sich um eine Programmbibliothek zur Lösung dichtbesetzter linearer Gleichungssysteme ([37]). Die Verfasser der *Top 500 Liste für Supercomputer*, die weltweit großes Ansehen in der Computerbranche hat, setzen den *LINPACK*-Benchmark zur Leistungsmessung der Systeme ein.[21]

Ein weiteres Messwerkzeug – der *SAGE* Benchmark – wurde in [24] verwendet. Er basiert auf einem Eulerschen Hydrodynamik-Code, der bis zu drei Dimensionen berechnen, mehrere Materialien verarbeiten und mit unterschiedlich feinen Netzen umgehen kann ([19]).

Jeder Anwendungs-Benchmark hat seine Vor- bzw. Nachteile, weil selten alle Elemente eines Computersystems gleichzeitig auf ihre Leistungsfähigkeit untersucht werden. Wie bei den realen Anwendungen verwenden diese Benchmarks Verfahren, die oft durch spezielle Eigenschaften des Rechners eine bestimmte Leistung erreichen. So wird der *OS-Jitter* sich unterschiedlich stark bei den verschiedenen Benchmarks bemerkbar machen. Deswegen werden zur Untersuchung des *OS-Jitter*-Effekts spezialisierte *Micro-Benchmarks* verwendet.

3.9 Analyse von Kernel Events

Die gerade vorgestellten Benchmarks stellen zwar fest, dass der *OS-Jitter* existiert und wie stark er ist, sagen aber nicht direkt, was dessen Ursache ist. Gelegentlich überlagern sich die Störungen und es ist gar nicht möglich, diese einem Verursacher genau zuzuordnen. Nur interne Abläufe, die bekannt sind – wie z.B. der *System Tick*, der in festen periodischen Abständen auftritt – können eindeutig identifiziert werden. Im Prinzip hängt die Zuordnung der *OS-Jitter*-Effekte zu ihren Ursachen stark von den Kenntnissen des Analysten über das System ab.

Um den Störungen doch genauer auf die Spur zu kommen, muss das Betriebssystem soweit verändert werden, dass Ereignisse, die den Benchmark potentiell stören, protokolliert werden.

An dieser Stelle soll auf drei Werkzeuge dieser Art hingewiesen werden, die das ermöglichen. Alle drei Produkte erfordern eine Änderung des Linux Kernels und sind nur für spezielle Versionen des Betriebssystemkerns verfügbar.

Im Rahmen dieser Arbeit wurde *KTAU* zur Analyse der Störungen eingesetzt. Deswegen wird im folgenden Abschnitt die Funktionsweise genauer dargelegt. Die anderen

beiden Werkzeuge – *KLogger* und *ClusterOS Toolkit* – werden nur kurz vorgestellt, aber nicht im Detail erklärt. Das Grundprinzip ist dem von *KTAU* sehr ähnlich.

Für *KTAU* wurde sich entschieden, weil es eine weitere Komponente des *TAU* ist, das u.a. von Mitarbeitern des Forschungszentrum Jülich mitentwickelt worden ist.

3.9.1 KTAU

KTAU steht für *Kernel TAU* und ist eine spezielle Erweiterung von *TAU*, den *Tuning and Analysis Utilities*, einem Analyse-Paket für parallele Programme [33]. Es instrumentalisiert den Linux Kernel und erlaubt es, alle möglichen Ereignisse im System zu erfassen, zu messen und zu protokollieren.

In *The Ghost in the Machine* ([23]) wird beschrieben, dass dort *KTAU* zum Erfassen des *OS-Jitter* genutzt wurde.

Zur Zeit ist das Werkzeug *KTAU* für *x86* und *PowerPC*-Architekturen in Kombination mit dem Linux Kernel 2.6.22 verfügbar und ist Teil des *ZeptoOS*-Projekts (siehe Seite 25).

Es handelt sich um ein *Open Source* Produkt, das von der *Argonne National Laboratory (USA)* entwickelt und als Download zur Verfügung gestellt wird. Es darf für nicht kommerzielle Zwecke verwendet werden [22].

Konzept von KTAU

Die hier verwendete Version 1.7.14 kann von der *KTAU* Homepage als Quellcode bezogen werden. Die Software besteht aus drei Teilen.

1. **Kernel Patch**
   Die aktuelle Version liefert einen Patch für den Linux Kernel 2.6.22.1 mit. Für andere Kernel Versionen muss der Patch erst angepasst werden.

   Um den Patch einzuspielen, benötigt man die entsprechenden Kernelquellen. Danach kann ein neuer Kernel erzeugt werden. Über Boot-Parameter werden die verschiedenen Arten der Ereignisprotokollierung aktiviert. Als Schnittstelle dient das Verzeichnis `KTAU` im `/proc`-Verzeichnis.

   Für die anfallenden Daten wird ein spezieller *Ring-Buffer* angelegt. Sobald dieser voll ist, werden die Daten nach dem FIFO Prinzip überschrieben.

2. **Der *KTAU* Daemon `ktaud`**
   Über diesen Dienst können die Informationen des Kernels abgegriffen und in `Log`-Dateien abgespeichert werden. Ob alle Prozesse beobachtet werden oder nur eine bestimmte Auswahl, wird über die Konfigurationsdatei `/etc/ktaud.conf` eingestellt.

3. ***KTAU* Kommandos**
   Es werden einige Befehle zur Verfügung gestellt, um die von `ktaud` angelegten `Log`-Dateien zu bearbeiten.

Auswerten der KTAU Daten

Ist der *KTAU* Patch aktiviert, wird jedes Ereignis, das einen Prozess im System betrifft und vom Kernel gesteuert wird, mitprotokolliert.

```
...
2094147562931883  in   smp_apic_timer_interrupt
2094147562932639  in   __do_softirq
2094147562933017  in   __run_timers
2094147562933377  out  __run_timers
2094147562933620  in   tasklet_action
2094147562934016  out  tasklet_action
2094147562934250  out  __do_softirq
2094147562934457  out  smp_apic_timer_interrupt
...
```

Tabelle 3.1: *KTAU* Protokoll-Datei

In der Protokolldatei (siehe Tabelle 3.1) sind drei Spalten zu sehen. Die erste enthält die Zeit, an der das Ereignis vom Betriebssystemkern ausgelöst wurde. Es handelt sich um den Inhalt des *Time Stamp Counter*-Registers der CPU. Dessen Vorteil ist, dass die später verwendeten Benchmarks dasselbe Register zur Zeitmessung auslesen können. Das erlaubt es, deren Messungen mit denen des *KTAU* Protokolls in eine chronologische Reihenfolge zu vereinen. Die verwendeten Benchmarks und die Qualität der Zeitmessung werden in Kapitel 4 näher erläutert.

Die zweite Spalte des *KTAU* Protokolls gibt an, ob das Ereignis gerade den Prozessor belegt (`in`), oder seine Aufgabe beendet (`out`). Wie man sieht, ist die Reihenfolge dieser Ausgaben geschachtelt. Startet zuerst Ereignis A und als zweites Ereignis B bevor A beendet wird, dann muss erst Ereignis B beendet werden, bevor Ereignis A beendet werden kann.

In der letzten Spalte steht der Funktionsname zu dem Ereignis, unter dem es im Kernel erschienen ist. In den Originaldaten stehen an der Stelle nur Speicheradressen, die mit Hilfe des *KTAU* Kommandos `funcmap` aufgelöst werden können. Der Name und die Zeiten liefern zumindest Anhaltspunkte, die die Suche nach der Ursache einer Störung erleichtern können.

3.9.2 KLogger

Auf der *Kernel Logger* Homepage [9] wird die Version 0.9 des Patches für den Linux Kernel zur Verfügung gestellt. Der Patch kann in die Kernel Quellendateien eingespielt werden. Danach kann ein neuer Kernel konfiguriert und erzeugt werden.

Was genau der *KLogger* protokollieren soll, entscheidet der Anwender über Konfigurationsdateien, so genannten Schemata. So könnte man eine Konfiguration nutzen, die jeden Kontext-Wechsel beim Prozess-Scheduling mitschreibt. Genauer beschrieben wird die Anwendung von *KLogger* in [11].

Diese Erweiterung steht unter der *GNU General Public License (GPLv2)* und ist bis für den Linux Kernel 2.6.20 verfügbar.

3.9.3 ClusterOS Toolkit

Von der Fakultät für Informatik an der Universität Karlsruhe gibt es ein abgeschlossenes Projekt *Scheduling* unter der Leitung von Prof. Walter F. Tichy und Jürgen Reuter. Dieses Projekt hatte zum Ziel, das *Gang Scheduling* Verfahren auf Mehrprozessor-Systemen auf einen losen Verbund von normalen Linux Rechnern abzubilden. Als Vehikel zur Synchronisation der Systeme wird das Netzwerk-Protokoll *ICMP* verwendet.

Um ihre Ergebnisse messen zu können, wurde ein Analysewerkzeug entwickelt, welches über Modifikationen am Kernel erlaubt, verschiedene Informationen – wie z.B. Prozess-Scheduling, Speicherzugriff und paralleler File-I/O – zu protokollieren. Die gesammelten Daten können im Anschluss ausgewertet werden.

Mithilfe des Analysewerkzeugs sollen verschiedene Scheduling-Strategien im Clusterverbund untersucht werden.

Zur Zeit ist diese Software nicht verfügbar. [26]

3.10 Ansätze zur Vermeidung des *OS-Jitter*

Ein erster und intuitiver Ansatz ist das Ausschalten unnötiger Dienste, um das Systemrauschen zu minimieren.

Damit wird sicherlich schon einiges erreicht, aber viele Optimierungen gehen noch weiter, indem sie – wie schon in Abschnitt 3.5 auf Seite 15 beschrieben – die Rechenzeit für den Job und die für das „*Housekeeping*“ des Betriebssystems trennen. Zuerst wird der Job gerechnet und anschließend werden die *Events* des Systems abgearbeitet. Wenn dieses Scheduling auf allen Knoten synchron durchgeführt wird (*Gang Scheduling*, siehe Seite 15), hat man den störenden Effekt des *OS-Jitter* minimiert.

In drei Projekten ist dieser Ansatz schon realisiert, die im Folgenden beschrieben werden.

3.10.1 ZeptoOS I/O Node Kernel

Hierbei handelt es sich um einen Teil des Gesamtprojektes *ZeptoOS*. Der *ZeptoOS I/O Node Kernel* ist ein abgespeckter Linux Kernel, der spezialisiert ist für *HPC*-Systeme wie *IBMs BlueGene* und *Crays XT3*. Es wird in Zusammenarbeit zwischen *Argonne National Laboratory* und der Universität von Oregon entwickelt. [1]

3.10.2 Compute Node Linux

Die Firma *Cray Inc.* hat für ihre älteren *HPC*-Systeme den Betriebssystemkern *Catamount* entwickelt, der seine Wurzeln in dem am *Sandia National Laboratories (USA)* entwickelten *light weight operating system* hat. Laut den Messungen in [34] entsteht mit dem *Catamount* Kernel so gut wie kein *OS-Jitter*, bis auf den periodischen *System Tick*.

Da dem *Catamount* Kernel jedoch die Fähigkeit des *Multithreading* fehlt, hat die Firma *Cray Inc.* sich entschieden, einen Linux Kernel einzusetzen. Dabei wurde sich auf den Kernel bezogen, der im *SLES 10* der Firma *Novell* enthalten ist. Es wurden zuerst alle unnötigen Dienste entfernt und der *System Tick* auf 10 Hz gesetzt. Die Messungen ergaben ein sehr ähnliches Verhalten wie bei dem *Catamount*-System. Zwar unterlag

dieses *Compute Node Linux (CNL)* im direkten Vergleich in der Leistung, aber es wird davon ausgegangen, dass *CNL* bei größeren Prozessorzahlen (> 1024) in Kombination mit *SMT* besser skalieren wird [35].

3.10.3 Compute Node Kernel

Einen ähnlichen Ansatz wie die Firma *Cray Inc.* mit dem *Compute Node Linux* geht die Firma *IBM* bei den *BlueGene* Systemen, wie sie auch im *Jülich Supercomputing Centre* installiert sind.

BlueGene ist eine massiv-parallele Rechnerarchitektur, die von IBM in Zusammenarbeit mit dem *Lawrence Livermore National Laboratory*[5] entwickelt worden ist. Auf den einzelnen Compute Nodes läuft zur Zeit noch ein *Single User/Single Processor* Betriebssystemkern, dem *Compute Node Kernel (CNK)*. Er initialisiert den Prozessor, den Speicher und das Netzwerk, und startet dann die anstehenden Prozesse, immer nur einen zur selben Zeit.

In [28] wird die Möglichkeit untersucht, den *CNK* auf einem *Bluegene/L*-System durch einen Standard Linux Kernel zu ersetzen, um so mehr Funktionen und Schnittstellen zu erhalten. Das Ergebnis ist, dass mit nur kleinen Änderungen am Betriebssystemkern und beim Kompilieren der parallelen Programme eine äquivalente Leistung auf dem System erzielt wird.

Die Ursache für die ursprünglich schlechtere Leistung liegt in der Speicherverwaltung. Unter Linux entstehen relativ viele *TLB*[6]*-Misses* auf den einzelnen Knoten. Um das zu beheben, müssen der *Large Page Support* im Kernel aktiviert werden und die Programme mit der Bibliothek *libhugetlbfs* übersetzt werden.

Da dieses Problem sehr hardwarespezifisch ist, muss untersucht werden, ob für andere Architekturen das Gleiche gilt.

[5] weitere Projektteilnehmer: Department of Energy - National Nuclear Security Administration, Columbia University, San Diego Supercomputing Center, California Institute of Technology

[6] Translation Lookaside Buffer: Element auf der CPU, das logische Speicheradressen in physikalische umwandelt

Kapitel 4

Messung des *OS-Jitter* Effekts

Im letzten Abschnitt auf Seite 19 sind die *1-Prozessor* und die *n-Prozessor* Klasse für *Micro-Benchmarks* zur Erfassung des *OS-Jitter*-Effekts eingeführt worden. In diesem Kapitel werden einige der Benchmarks vorgestellt und dann festgelegt, welche für die Analysen eingesetzt werden.

Bevor aber die Benchmarks betrachtet werden, werden verschiedene Funktionen zur Zeiterfassung untersucht. Denn je nachdem, wie klein und feingranular eine Störung ist, muss die Zeit dementsprechend genau bestimmt werden.

4.1 Qualität der Zeitmessung

Je genauer Zeitmessungen in einem *Micro-Benchmark* sind, desto feinere Störungen können im System erkannt werden. Daher ist die Frage der Zeitmessung in diesem Zusammenhang zentral. Drei Varianten wurden im Rahmen dieser Arbeit getestet und miteinander verglichen.

MPI_Wtime

In parallelen Programmen, wird für die Kommunikation meistens *MPI* (siehe Abschnitt 3.2.1 auf Seite 12) verwendet. Eine Funktion, die der *MPI*-Standard zur Zeitmessung anbietet, ist *MPI_ Wtime*. Wie gut die Auflösung dieser Funktion ist, gibt *MPI_ Wtick* an.

Der Rückgabewert von *MPI_ Wtime* ist eine doppelt genaue Gleitkommazahl, und das Handbuch sagt über sie:

> This is intended to be a high-resolution, elapsed (or wall) clock. See MPI_WTICK to determine the resolution of MPI_WTIME.[7]

Auf dem *JuRoPA* Testsystem, das in Kapitel 6 für die Messung des *OS-Jitter* genutzt worden ist, ist der *MPI_ Wtick* auf $1\mu s$ eingestellt.

gettimeofday (gtod)

Die Standardfunktion für Programme um die aktuelle Zeit zu messen, ist *time*. Eine Alternative dazu ist die *gettimeofday*-Funktion, die laut [10] weniger *Overhead* im Vergleich zur *time*-Funktion hat, und damit schneller ist. Als Ergebnis erhält man die Sekunden und Mikrosekunden, die seit dem 1. Januar 1970, 0 Uhr UTC[1] vergangen sind.

[1] UTC = Universal Time Coordinated, die koordinierte Weltzeit

Time Stamp Counter Register aus der CPU

Seit Intel die Pentium-Prozessoren herausgebracht hat, gibt es eine Möglichkeit, auf Benutzerebene die Anzahl an Taktzyklen seit dem letzten Reset des Prozessors auszulesen. Der Wert steht in einem Register und kann aus einem Programm via der *Inline-Assembler* Routine mit dem Namen *Read Time Stamp Counter* (`rdtsc`, siehe Abbildung 4.1) ausgelesen werden.

```
#include <stdint.h>
extern "C" {
  __inline__ uint64_t rdtsc() {
    uint32_t lo, hi;
    /* We cannot use "=A", since this would use %rax on x86_64 */
    __asm__ __volatile__ ("rdtsc" : "=a" (lo), "=d" (hi));
    return (uint64_t)hi << 32 | lo;
  }
}
```

Abbildung 4.1: rdtsc() für GNU C/C++ von Wikipedia

Allgemein liest diese Funktion zwei 32Bit Register aus, die zu einer 64Bit Integerzahl zusammengefasst werden. Das funktioniert sowohl auf 32Bit als auch auf 64Bit-Maschinen. Inzwischen haben die anderen CPU Hersteller nachgezogen und diese Register mit in ihre *x86*-Linie aufgenommen.

Damit existiert eine Möglichkeit, den aktuellen *Cycle Count* der CPU auszulesen. Laut

Funktion	`Zeit (in Sek.)`	**Erläuterung**
`clock()`	`16.808644`	Standard Routine aus `time.h`
`gettimeofday()`	`12.044037`	Standard Routine aus `time.h`
`rdtsc()`	`2.770133`	Assembler Routine
`MPI_Wtime()`	`12.969749`	*MPI*-Bibliotheksfunktion

Tabelle 4.1: Vergleich von Zeitmessfunktionen: Es wurde der Zeitaufwand für 100 Mio. Funktionsaufrufe gemessen.

[10] verursacht diese Variante den geringsten Overhead und ist somit am effizientesten von den hier vorgestellten Messmethoden.

Um aus den Messwerten der `rdtsc`-Funktion die Zeit in Sekunden zu erhalten, müssen die Werte durch die Taktfrequenz des Prozessors geteilt werden. Unter Linux erhält man diese Information über */proc/cpuinfo*. Ergibt z.B. eine Messung durch `rdtsc`, dass 1.000.000 Taktzyklen vergangen sind, und der zugehörige Prozessor arbeitet mit einer Taktfrequenz von 2,5 GHz, dann erhält man eine Zeit von $\frac{10^6}{2{,}5 \cdot 10^9} s = 4 * 10^{-4} s = 0{,}4 ms$.

In Tabelle 4.1 ist der Aufwand verschiedener Zeitmessfunktionen für die Dauer von 100 Mio. Funktionsaufrufen gemessen worden. Wie man sieht, benötigt die `rdtsc`-Funktion wesentlich weniger Zeit als die anderen getesteten Methoden.

Inzwischen gibt es einige Weiterentwicklungen in der Prozessor-Technologie, die dazu führen können, dass die Anzahl der Prozess-Zyklen pro Sekunde variieren kann. Zum einen verwenden fast alle modernen *x86*-CPUs die *Speedstep*-Technologie, so dass bei niedriger Last der Prozessor niedriger getaktet wird, um Strom zu sparen. Bei einer Intel Xeon CPU mit $3GHz$, wie sie für die vorliegenden Tests verwendet wurde, kann die Frequenz auf bis zu $2GHz$ gesenkt werden. Über das Betriebssystem kann dieser Effekt aber deaktiviert werden. Unter *openSuSE 10.2* mit der Kernel Version 2.6.18 gibt es für jeden Prozessor ein Verzeichnis `/sys/devices/system/cpu/cpu#/cpufreq` mit den Konfigurationsdateien, um das Verhalten einzustellen. In der Datei `scaling_governor` muss der Werte `performance` stehen. Der sorgt dafür, dass die minimale und die maximale Frequenz auf den größten Wert eingestellt werden.

Neben der *Speedstep*-Technologie gibt es noch weitere mögliche Ursachen für eine Frequenzänderung. So greift bei einer Überhitzung des Prozessors der Selbstschutzmechanismus und er taktet sich selbstständig herunter.

Die neuen *Nehalem* Prozessoren bieten sogar einen sogenannte *Turbo Mode* an, der eine Steigerung der Taktfrequenz erlaubt. Wenn zum Beispiel drei der vier Prozessoren nichts zu tun haben, kann der eine Prozessor mit einer bis zu 10% höheren Taktrate laufen.

Interessant bei der *Nehalem* CPU ist, dass es ein weiteres Register zur Messung eines Referenztaktes gibt. Dieses Register ist in der *Performance Monitoring Unit (PMU)* des Prozessor enthalten. Laut [18] kann auf diesen Wert ohne Kernel Unterstützung zugegriffen werden. Weitere Informationen sind dem Prozessorhandbuch zu entnehmen.

Dieser Referenztakt hätte den Vorteil, dass er vom Design her konstant ist.

4.2 1-Prozessor Micro-Benchmark

Wie in Abbschnitt 3.8.1 auf Seite 19 beschrieben gibt es für diese Klasse an *Micro-Benchmarks* viele Varianten. Entscheidend für den Typ dieser Benchmarks ist, dass nur die Störungen des lokalen Systems erfasst werden, und keine Art von globaler Kommunikation mit anderen Prozessoren stattfindet. In vielen Veröffentlichungen, die sich mit dem Thema *OS-Jitter* beschäftigen, haben die Autoren eine eigene Implementierung für ihre Messungen gewählt. Zwei Vertreter aus dieser Klasse – der *Selfish Detour* und der *Fix Time Quantum* Benchmark – werden im Folgenden vorgestellt.

4.2.1 Selfish Detour Benchmark Suite

Auf der Internationalen Cluster Konferenz in Spanien im September 2006 haben Pete Beckman, Kamil Iskra, Kazutomo Yoshii und Susan Coghlan ihren selbst entwickelten *Selfish Detour* Benchmark vorgestellt [2]. Ziel war es – wie in dieser Arbeit auch – Störungen[2] (*OS-Jitter*) sowohl auf einem *High Performance System* wie einer *BlueGene/L* als auch auf einem normalen PC sichtbar zu machen. Der Benchmarks besteht aus einer einfachen Schleife, bei der in jedem Durchlauf die aktuelle Zeit t_n bestimmt wird. Wenn

[2] *selfish detour*: Es sollen Störungen gemessen werden, die von dem Betriebssystem für eigene Zwecke (selfish) erzeugt werden.

$\delta t = t_n - t_{n-1}$ größer einer vorgegebenen Schranke (*threshold*) ist, wird eine Störung registriert, und die beiden Zeiten t_{n-1} und t_n in einem Vektor gespeichert.

```
cnt=0;
min_ticks=INFINITY;
current=rdtsc();

while (cnt < N) {
  prev = current;      /* keep the previous timer value */
  current = rdtsc();   /* obtain the current timer value */

  td = current-prev;
  if (td > threshold) {
    detour[cnt++]=prev;
    detour[cnt++]=current;
  }
  if (td < min_ticks) min_ticks=td;
}
```

Abbildung 4.2: *selfish Detour*: Schleife zur Erfassung der Störungen

Der Benchmark endet entweder, wenn eine bestimmte Anzahl an Störungen registriert wurde, oder wenn die vorher definierte Zeit überschritten (Standardvorgabe: 1000 gemessene Störungen und maximal 120 Sekunden) wurde.

Der Schwellwert für die Schranke (*threshold*) ist mit $1\mu s$. voreingestellt.

Alle drei Werte – Anzahl an Messungen, obere Schranke für die Laufzeit und der Schwellwert zur Erkennung einer Störung – lassen sich beim Aufruf über Parameter angeben.

Als Ergebnis erhält man eine zweispaltige Datei, in deren erster Spalte der Startpunkt der Schleife mit Störung steht, und in der zweiten die Dauer der Schleife. Diese Werte können graphisch dargestellt und statistisch ausgewertet werden.

Für ihre Messungen stand den Autoren ein *BlueGene/L* System zur Verfügung. Dabei hat sich gezeigt, dass auf den *Compute Nodes* des Rechners so gut wie kein *OS-Jitter* zu sehen war. Deswegen bot es sich an, auf diesem System Messungen mit parallelen Programmen durchzuführen, um dann zu sehen, wie sich künstliche Störungen auf den Job auswirken.

Diese Idee wird später für die Untersuchungen in Kapitel 6 aufgenommen.

4.2.2 FTQ – Fix Time Quantum

Ein intuitiver Ansatz für den *Micro-Benchmark* ist die Vorgabe einer bestimmten Menge an Arbeit, um dann die Zeit zu messen, die für dessen Bearbeitung benötigt wird. Ein Beispiel ist in Abbildung 4.3 auf der nächsten Seite dargestellt.

Im September 2004 auf der *Cluster Computing Conference* in San Diego (USA), haben Matthew J. Sottile und Ronald Minnich einen alternativen Ansatz für einen *Micro-Benchmark* vorgestellt: Den *Fix Time Quantum* Benchmark [30]

Deren Hauptkritikpunkt gegenüber den *Fix Work Quantum* Benchmarks war, dass dort die Zeit pro Arbeitseinheit gemessen wird, während beim *Fix Time Quantum* die

```
for (i=0; i< N; i++) {
  timestamp = rdtsc();        /* get the current timer value */
  timeseries[i] = timestamp;
  for (j=0; j<workcount; j++) {
    /* some operation */
  }
}
```

Abbildung 4.3: Beispiel für einen *Fix Work Quantum* Microbenchmark

Arbeit pro Zeiteinheit gemessen wird. Störungen treten im System periodisch auf. Da der *FTQ* feste Zeitintervalle vorgibt, in denen er jeweils die mögliche Arbeit misst, erhält man ein zeitbasiertes Ergebnis. Hiermit lassen sich einfacher Rückschlüsse auf die Ursache der Störung ziehen. Die Dienste lassen sich eher über ihr periodisches Auftreten als über die Länge des benötigten Arbeitsquantums identifizieren. In der Veröffentlichung zum *FTQ*-Benchmark empfehlen die Autoren, frequenz- bzw. zeitbasierte Analyse auf die Messwerte anzuwenden.

Algorithmus des FTQ

In Abbildung 4.4 auf der nächsten Seite ist der Programmabschnitt für die innere Schleife zu sehen. Die Länge des Zeitintervalls wird in der Form 2^n mit $n \in \mathbb{N}$ angegeben. Man gibt die Zeit nicht in Sekunden, sondern in Taktzyklen an. Als Abbruchbedingung der *Workload*-Schleife wird die aktuelle Zeit aus dem *Time Stamp Counter*-Register des Prozessors ausgelesen, und das Zeitintervall hinzu addiert. Von diesem Wert werden die $n-1$ niederwertigen Bits auf „0“ gesetzt. So wird dafür gesorgt, dass der Abstand der Messintervall-Enden immer gleich groß ist.

Wird der Benchmark durch eine Störung unterbrochen, so können drei Fälle auftreten.

1. Fall: Die Störung liegt komplett im Messintervall
Das hat zur Folge, dass die Anzahl der Iterationen (*Workload*) entsprechend geringer ausfällt.

2. Fall: Die Störung liegt am Ende des Messintervalls und reicht noch in das nächste hinein.
Das bedeutet, wenn die Unterbrechung vorbei ist und der Benchmark weiterrechnen darf, wird die aktuelle Iteration sofort beendet und die Nächste wird gestartet. Da aber schon einige Zeit für das neue Messintervall verstrichen ist, werden dementsprechend weniger Iterationen durchgeführt.

3. Fall: Eine Störung dauert so lange, dass mindestens ein Messintervall komplett übersprungen wird.
Da der Benchmark in den ausgelassenen Intervallen nie aktiv war, konnten keine Zeiten erfasst oder gespeichert werden. Es werden keine Einträge in für diese Messungen protokolliert.

Im ersten bzw. zweiten Fall wird eine Messung pro Zeiteinheit erreicht.

```
terminal = 1 << 21; /* sample interval = 2^21 ticks */

while (!endloop) {
  last = rdtsc();    /* get the start time */
                     /* calculate the end_time */
  end_interval = (last + terminal) & (~(terminal-1));

  for (now=last, count=0 ; now<end_interval;) {
    count++;
    now = rdtsc();  /* get the current time */
  }

  totalcount[done][0] = last;  /* sample start time */
  totalcount[done][1] = count; /* sample count      */
  done++;
  if (done > do_total_count)  /* done > number of samples */
    endloop = true;
}
```

Abbildung 4.4: Fix Time Quantum Microbenchmark: die innere `for`-Schleife misst die Arbeit pro Zeitintervall

Messungen mit dem FTQ

Wie gerade beschrieben, wird bei dem *FTQ* Benchmark ein Zeitintervall fest vorgegeben und dann gemessen, wie viele Iterationen in dieser Zeit durchlaufen werden. Als Ergebnis erhält man zwei Ausgabedateien: `ftq_times.dat` und `ftq_counts.dat`

```
⋮
3590
3589
3271
3589
3590
⋮
```

`ftq_counts.dat`:
Anzahl der Schleifendurchläufe (*Workload*)

```
⋮
48275110494315
48275110756503
48275111018610
48275111280798
48275111542905
⋮
```

`ftq_times.dat`:
Werte des CPU *Cycle Counter* vor der *Workload*-Schleife

Abbildung 4.5: Ausgabe des *FTQ* Benchmarks

Die Datei `ftq_counts.dat` enthält für jedes Intervall die Anzahl der Iterationen. In `ftq_times.dat` stehen die jeweils wirklich gemessenen Anfangszeiten. Es handelt sich dabei um den Wert des *Time Stamp Counter* Registers aus dem Prozessor, der direkt vor der *Workload* Schleife ausgelesen wird.

Stellt man die Anzahl der Schleifendurchläufe (*counts*) gegen die Zeit aus `ftq_times.dat` dar, erhält man eine Graphik wie in Abbildung 4.6. Es gibt einen ma-

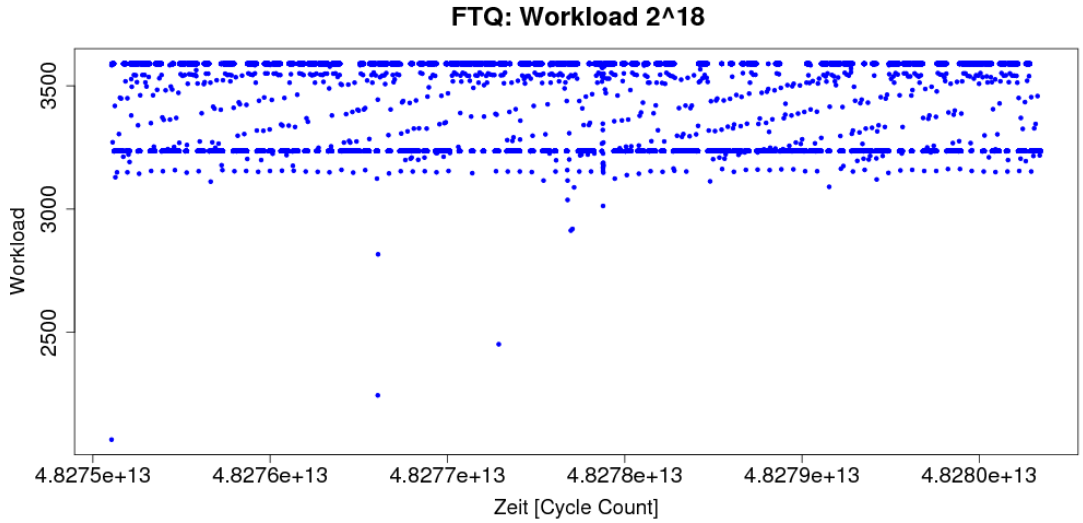


Abbildung 4.6: graphische Darstellung von *FTQ* Messdaten

ximalen Wert – hier sind es 3590 Operationen für eine Messung – der anscheinend eine obere Schranke ist. Diese Linie wird aber immer wieder von Ausreißern nach unten unterbrochen. Obwohl die Zeitintervalle annähernd gleich groß sind, treten dort Messungen mit wesentlich weniger Iterationen auf. Geht man nun davon aus, dass die Prozessoren eine konstante Leistung bringen, muss etwas das Programm gestört haben.

Nimmt man es ganz genau mit der *FTQ* Messung, sind die Intervallgrenzen zwar äquidistant gewählt, doch durch die Zeitmessung an sich und dem Speichern der Messwerte entsteht ein gewisser *Overhead*, der nicht direkt erfasst wird. Es ist zu erwarten, dass dieser Mehraufwand immer gleich groß ist. Ob dem so ist, wird später in dem Kapitel 6 untersucht.

4.3 *n*-Prozessor Benchmark – MPI-Jitter

Für die Klasse der *n-Prozessor Micro-Benchmark* wurde im Rahmen dieser Arbeit ein eigenes Programm mit dem Namen *MPI-Jitter* geschrieben.

Bei parallelen Programmen, die ihre Aufgaben auf vielleicht mehreren Tausend Prozessoren eines *HPC*-System verteilen und deswegen global kommunizieren müssen, ist die Frage, wie sich der lokale *OS-Jitter* auf sie auswirkt. Diesen Effekt soll *MPI-Jitter* erfassen und sichtbar machen.

In dieser Arbeit wird sich an dem Grundkonzept, wie in [4] vorgestellt, orientiert. Der Ablauf der verwendeten Realisierung ist in Abbildung 4.7 auf der nächsten Seite dargestellt. Nach einer Arbeitsphase zufälliger Länge, die für ein beliebiges Programmverhalten steht, werden alle Prozesse mittels eines *Barrier* synchronisiert. So wird sichergestellt, dass die folgende *Compute-Phase* auf allen Prozessoren gleichzeitig startet. Vor dieser Phase und direkt danach wird auf jedem Prozessor die aktuelle Zeit gemessen (t_{start} und $t_{finished}$). Im Anschluss werden die Jobs wieder synchronisiert. Ein *Barrier* steht in diesem Fall wieder für eine globale, blockierende Kommunikation. Es wird

Benchmark-Schleife — läuft auf jedem Prozessor

busy-wait for randomly chosen period	
MPI_Barrier	\\ synchronise nodes
t_{start} = rdtsc()	
compute phase	\\ always the same specified work
$t_{finished}$ = rdtsc()	
MPI_Barrier	\\ synchronise nodes
t_{wait} = rdtsc()	
measure count not reached	

Abbildung 4.7: Innere Schleife des *MPI-Jitter* Benchmark

noch einmal die Zeit festgehalten, um zu sehen, wie lange die einzelnen Prozesse warten mussten (t_{wait}). Danach beginnt die Schleife von vorne und die nächste Messung kann durchgeführt werden.

Dieser Ablauf simuliert ein typisches Programm, wie es auf einem *High Performance System* gerechnet werden soll. Die Aufgaben werden auf die Prozessoren verteilt, dort gerechnet, und anschließend werden die Ergebnisse untereinander ausgetauscht, d.h. es findet eine globale Kommunikation statt. Während der Rechenphase kann jeder Prozess auf seinem Prozessor arbeiten, unabhängig davon, was auf den anderen Prozessoren passiert. Doch erreichen die Prozesse die Kommunikationsphase, die hier durch einen *Barrier* symbolisiert wird, muss jeder Prozess darauf warten, dass auch der Letzte, der am Datenaustausch beteiligt ist, für die Kommunikation bereit ist. Alle müssen auf den letzten – den langsamsten – Prozess warten.

Der Entwickler sollte sein Programm so aufbauen, dass die Last möglichst ausgeglichen ist, so dass bei der parallelen Jobverarbeitung eine möglichst hohe Effizienz erreicht wird. Im *MPI-Jitter* Benchmark ist das der Fall, denn er ist so realisiert, dass alle Prozessoren exakt den gleichen *Workload* rechnen müssen. Es handelt sich also um einen Benchmark vom Typ *Fix Work Quantum*. Der Grund liegt darin, dass es einfacher ist, exakt die gleiche Arbeit vorzugeben, als für alle Prozesse genau dieselbe Zeit zu bestimmen. Denn bei der Zeitmessung kommt es unweigerlich zu Ungenauigkeiten. Ein Grund dafür ist der *Overhead* bei den Zeitmessfunktionen. Da es selbst in dieser Phase zu *OS-Jitter* Störungen kommen kann, werden die Zeiten variieren. Das führt dazu, dass den einzelnen

Prozessen nicht exakt das gleiche Zeitintervall zur Verfügung gestellt werden kann.

Über die Länge der *Compute*-Phase kann der *MPI-Jitter* Benchmark eine unterschiedliche Granularität des Algorithmus simulieren. So kann der Einfluss des *OS-Jitter* auf zwei Klassen von Jobs untersucht werden: Solche, die während der Berechnung längere Phasen rechnen, und erst dann ihre Ergebnisse austauschen und solche, die nur sehr kurze Rechenoperationen ausführen und dann jedes Mal kommunizieren müssen.

Messdaten des *MPI-Jitter* Benchmark

Der *MPI-Jitter* Benchmark führt auf n Prozessoren eine bestimmte Anzahl an Messungen durch. Dabei wird die i-te Messung auf allen Prozessoren durch einen *MPI-Barrier* gleichzeitig gestartet. Es werden jeweils drei Zeiten erfasst:

$t\_start_{i,j}$: Zeitpunkt vor der i-ten *Compute*-Phase des j-ten Prozesses

$t\_finished_{i,j}$: Zeitpunkt nach der i-ten *Compute*-Phase des j-ten Prozesses

$t\_wait_{i,j}$: Zeitpunkt nach dem i-ten *Barrier* des j-ten Prozesses

⋮	⋮	⋮
239904735894495	239904735900912	239904736703505
239904737572572	239904737578926	239904738399114
239904739189278	239904739195776	239904740066256
239904740693295	239904740699730	239904741638331
239904742470444	239904742476906	239904743180319
239904744128037	239904744134445	239904744726384
⋮	⋮	⋮

Abbildung 4.8: Messdaten auf dem j-ten Prozessor: *t_ start, t_ finished* und *t_ wait*. Auszug einer Messung auf dem *JuRoPA* Testsystem, das für die Messungen in Kapitel 6 verwendet wurde.

Als Ergebnis erhält man einen dreispaltigen Datensatz wie er in Abbildung 4.8 zu sehen ist. Es gilt

$$t\_compute_{i,j} = t\_finished_{i,j} - t\_start_{i,j}$$
$$t\_all_{i,j} = t\_wait_{i,j} - t\_start_{i,j}$$

wobei $t\_compute_{i,j}$ die Zeit ist, die in der i-ten Messung auf dem j-ten Prozessor für die *Compute*-Phase benötigt wird. Dementsprechend ist $t\_all_{i,j}$ die Gesamtzeit für eine Messung.

Die sollte, wegen des *Barrier* für alle Prozesse in derselben Iteration gleich groß sein.

$$t\_all_{i,1} = \ldots = t\_all_{i,n} = t\_all_i$$

Die Gesamtzeit einer Messung hängt von dem Prozess ab, der am längsten für die Rechenphase benötigt. Die $t\_compute_{i,j}$ aller Prozesse in einer Iteration können miteinander verglichen werden. Es kann der langsamste Prozess und damit die mittlere Wartezeit t_lost_i eines Prozesses in der i-ten Messung bestimmen werden.

$$\begin{aligned} t\_lost_i &= \frac{1}{n}\sum_{j=1}^{n}(\max(t\_compute_{i,.}) - t\_compute_{i,j}) \\ &= \max(t\_compute_{i,.}) - \frac{1}{n}\sum_{j=1}^{n} t\_compute_{i,j} \\ &= \max(t\_compute_{i,.}) - \bar{t}\_compute_{i,.} \end{aligned}$$

Bestimmt man aus allen Messungen den Mittelwert von t_lost_i, kann man den relativen Verlust ermitteln, eine Kennzahl, die angibt, wie stark der *OS-Jitter* sich auf den Benchmark ausgewirkt hat. Sei m die Anzahl der Messungen, dann gilt:

$$lost_{rel} = \frac{\frac{1}{m}\sum_{i=1}^{m} t\_lost_i}{\frac{1}{m\cdot n}\sum_{i=1}^{m}\sum_{j=1}^{n} t\_compute_{i,j}} = \frac{\bar{t}\_lost}{\bar{t}\_compute}$$

Bei der Messung aus Abbildung 4.8 auf der vorherigen Seite auf einer CPU mit vier Prozessoren entstanden ca. 5,3 % Verlust durch Wartezeiten.

Auf diese Weise können erste statistische Kennzahlen bestimmt werden, die etwas über den *OS-Jitter* im System aussagen.

Kapitel 5

Mathematisches Hilfsmittel zur Datenauswertung

Wendet man die Benchmarks *Fix Time Quantum* und den *MPI-Jitter* auf einem *High Performance Cluster* an, werden große Mengen an Messdaten erzeugt. Um die Daten auszuwerten, werden mathematische Standardverfahren benötigt. Man interessiert sich z.B. dafür, wie die Verteilung der Zeiten für die *Compute*-Phase auf allen Prozessoren ist. Für diesen Zweck kann eine Dichteschätzung durchgeführt werden. Es wird die Herleitung vom Histogramm bis hin zum Kerndichteschätzer gezeigt.

Die Daten des *FTQ* Benchmarks haben die Eigenschaft, dass sie zeitbasiert sind. Um in diesen Messungen Signale einer bestimmten Frequenz zu finden, wird eine Frequenzanalyse mithilfe der *Fast Fourier-Transformation* durchgeführt. Es handelt sich um eine optimierte Version der *diskreten Fourier-Transformation (DFT)*. Um die Ergebnisse zu verbessern, werden sie mit einem speziellen Verfahren der Mittelwertbildung geglättet.

5.1 Dichteschätzer

Die einfachste Art der Dichteschätzung ist das Histogramm. Dabei wird die gesamte Messzeit in n disjunkte Teilintervalle der Breite h – so genannten Klassen – unterteilt. Für die Messdaten des Zeitaufwands der *Compute*-Phase des *MPI-Jitter* Benchmarks kann die Zeit in gleich große Intervalle als Klassen definiert werden. Für die Dichte erhält man dann wie folgt ein Histogramm:

$$\tilde{f}(x) = \frac{1}{nh} \cdot (\text{Anzahl der } t\_compute_i, \text{ die in der gleichen Klasse wie } x \text{ liegen})$$

Dieses Verfahren hat jedoch Nachteile. Zum einen kann die Wahl des Klassenbreite und der Grenzpunkte der Klassen einen erheblichen Einfluss auf das Histogramm ergeben. Andererseits ist $\tilde{f}(x)$ nicht stetig.

Eine erste Verbesserung ist, die Klassen direkt um die Messpunkte zu legen. Dazu betrachtet man die Zahl der Messungen im Intervall der Breite h um den Punkt x.

$$\begin{aligned}\tilde{f}(x) &= \frac{1}{nh} \cdot \left(\text{Anzahl der } t\_compute_i, \text{ die in } \left(x - \tfrac{h}{2}, x + \tfrac{h}{2}\right) \text{ liegen}\right) \\ &= \frac{1}{nh} \sum_{i=1}^{n} G\left(x - t\_compute_i\right) \qquad \text{mit} \qquad G(z) = \begin{cases} 1 & \text{für } \frac{-h}{2} < z < \frac{h}{2} \\ 0 & \text{sonst} \end{cases}\end{aligned}$$

Als verbleibender Nachteil ist dieser Dichteschätzer immer noch nicht stetig, auch wenn die zugrundeliegenden Zufallsereignisse vielleicht eine stetige Dichte besitzen.

Die Ursache für die Unstetigkeit ist die Wahl einer Treppenfunktion für $G(x)$. Um diesen Nachteil zu beheben, werden die einzelnen Messpunkte, die sich in dem Intervall um x befinden, gewichtet. Dazu wird die Funktion $G(x)$ durch die Funktion $K(x)$ ersetzt. Für diese Funktion – auch Kernfunktion genannt – muss gelten:

- $K(x) \geq 0 \qquad$ für $\; x \in (-\infty, \infty)$
- $K(x)$ ist stetig
- $\int_{-\infty}^{\infty} K(x)\, dx \;\; = 1$

In der Regel ist $K(x)$ zusätzlich noch symmetrisch um den Nullpunkt und unimodal[1]. Eine typische Kernfunktion ist z.B. die Dichte der Standardnormalverteilung, dem Gaußkern.

$$\text{Gaußkern} \quad K(x) := \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{1}{2}x^2\right)$$

Als Dichteschätzung erhält man damit:

$$\begin{aligned}\tilde{f}(x) &= \frac{1}{nh}\sum_{i=1}^{m} K\left(\frac{x - t\_compute_i}{h}\right) \\ &= \frac{1}{n}\sum_{i=1}^{m} K_h(x - t\_compute_i) \qquad \text{mit} \qquad K_h(t) = \frac{1}{h}K\left(\frac{t}{h}\right)\end{aligned}$$

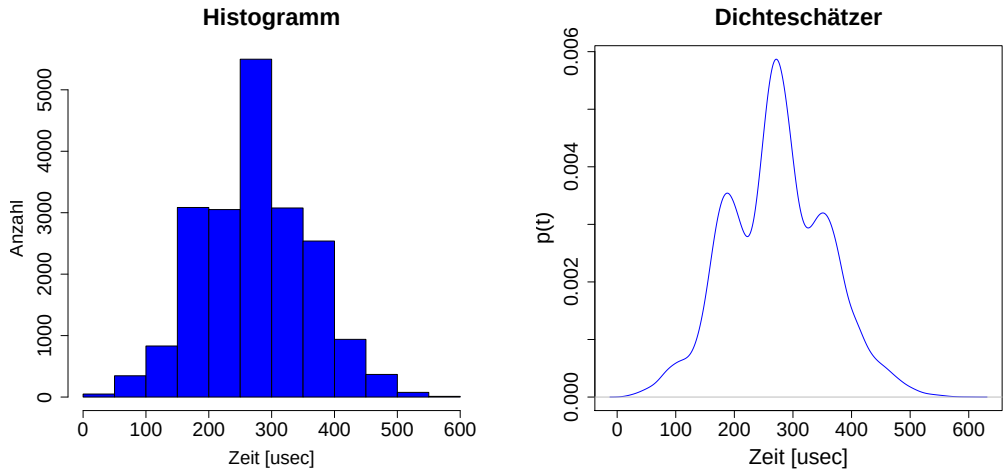


Abbildung 5.1: Ein Histogramm und ein Dichteschätzer zu denselben Daten

Einen entscheidenden Einfluss auf die Qualität des Kerndichteschätzers hat die Wahl des Parameters h, der Bandbreite. Stellt man ihn zu fein ein, kann es sein, dass lokale Ereignisse einen zu starken Einfluss auf den Dichteschätzer nehmen und die Funktion

[1] Eine Funktion $f(x)$ wird als *unimodal* bezeichnet, wenn es ein m gibt, so dass $\max(f(x)) = f(m)$ ist, und für $x < m$ $f(x)$ monoton wachsend und für $x > m$ $f(x)$ monoton fallend ist.

wird sehr unruhig. Ein gröberes h sorgt für eine Glättung der Kurve, es können aber Strukturen und damit Informationen der Originaldaten verloren gehen. Das h wird auch als Glättungsparameter bezeichnet, und ein zu kleiner oder ein zu großer Wert führen zur Unter- bzw. Überglättung (*undersmoothing, oversmoothing*). In Abbildung 5.2 sind „schlechte“ Varianten des Dichteschätzers aus Abbildung 5.1 dargestellt.

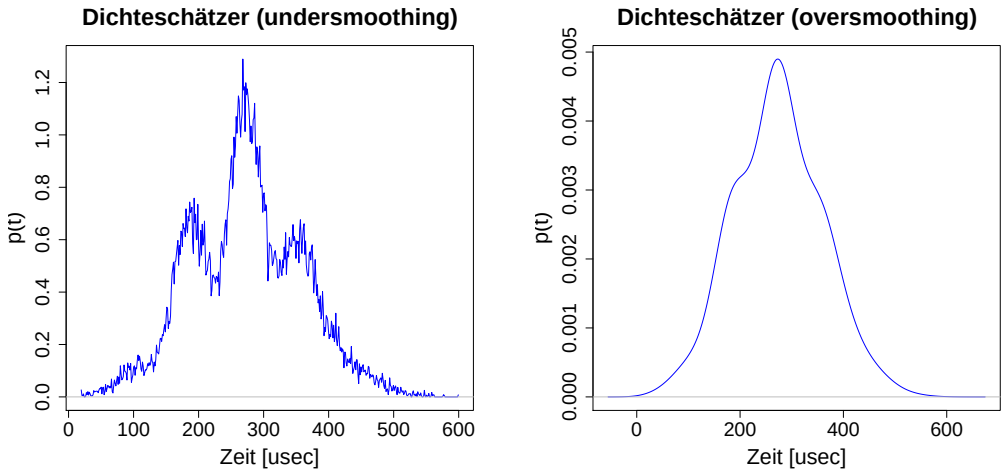


Abbildung 5.2: Einfluss der Bandbreite „h“ auf die Glättung des Dichteschätzers

In der Statistiksoftware R wird für die Bandbreite des Dichteschätzers ein Verfahren von Silverman benutzt [29]. Durch die Messdaten werden folgende Parameter definiert:

$$\begin{aligned} N &:= \text{Stichprobenumfang} \\ S &:= \text{empirische Standardabweichung} \\ \Delta Q &:= \text{interquantilerange}^2\text{: 0.75-Quantil} - \text{0.25-Quantil} \end{aligned}$$

Die Formel zur Bestimmung der Bandbreite h lautet dann:

$$h = N^{-\frac{1}{5}} \cdot 0.9 \cdot \min\left(S, \frac{\Delta Q}{1.34}\right)$$

[2] Beim p-Quantil werden die sortierten Daten in p gleich große Mengen unterteilt. Für $p = 4$ spricht man auch von Quartile (Viertelwerte). In diesem Fall gibt es das 0.25-Quantil, das 0.5-Quantil und das 0.75-Quantil, die Punkte an denen die Menge unterteilt werden.

5.2 Frequenzanalyse mittels *Fast Fourier-Transformation (FFT)*

In Abschnitt 4.2.2 wurde diskutiert, dass die Resultate des *Fix Time Quantum* Benchmarks Arbeit pro Zeit messen werden. Da die Werte zeitbasiert sind, können Werkzeuge für die Frequenzanalyse eingesetzt werden. Ziel ist es, Störungen aufzuspüren, die sich in regelmäßigen Abständen wiederholen, wie z.B. der *System Tick*.

Ein typischer Ansatz für die Frequenzanalyse ist die *Fourier-Transformation*. Sie bildet zeitkontinuierliche Signale in den Frequenzraum ab und findet in vielen Bereichen der Wissenschaft Anwendung. Es gelten folgende Formeln:

$$\text{Fourier-Transformation:} \qquad F(\omega) = \int_{-\infty}^{\infty} f(t)e^{-i\omega t}\,dt \tag{5.1}$$

$$\text{inverse Fourier-Transformation:} \qquad f(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} F(\omega)e^{+i\omega t}\,d\omega \tag{5.2}$$

Hergeleitet wird die *Fourier-Transformation* aus der *Fourierreihe*, mit der beliebige Funktionen durch trigonometrische Funktionen (Sinus und Kosinus) angenähert werden.

$$\text{Fourierreihe:} \qquad f(t) = \sum_{n=-\infty}^{\infty} C_n e^{in\omega t} \tag{5.3}$$

$$\text{mit den Fourierkoeffizienten:} \quad C_n = \tfrac{1}{T}\int_{-\frac{T}{2}}^{\frac{T}{2}} f(t)e^{-in\omega t}\,dt \tag{5.4}$$

Führt man Experimente wie z.B. bei dem *FTQ* Benchmark durch, wird eine feste Abtastrate verwendet, mit der die Daten erfasst werden. Das bedeutet, es steht kein kontinuierliches Signal sondern nur Messwerte zu diskreten Zeitpunkten zur Verfügung. Die gesamte Messzeit T wird in N gleich große Abschnitte unterteilt mit $\Delta t = \frac{T}{N}$, und es wird eine Messreihe $(y_0, y_1, y_2, \ldots, y_{N-1})$ erhoben mit $f(k \cdot \Delta t) = y_k$ mit $k = 0, 1, \ldots, (N-1)$.

Für die diskreten Messwerte nähert man das Integral für die *Fourierkoeffizienten* aus Formel 5.4 über eine Riemannsche Summe an:

$$C_n \approx \tilde{C}_n = \frac{1}{T}\sum_{k=0}^{N-1} f(k\cdot\Delta t)e^{-in\omega k\Delta t}\cdot\Delta t \overset{\omega=\frac{2\pi}{T}}{=} \frac{1}{T}\sum_{k=0}^{N-1} f(k\cdot\Delta t)e^{-in\frac{2\pi}{T}k\Delta t}\cdot\Delta t$$

Setzt man dort $y_k = f(k \cdot \Delta t)$ und $T = N \cdot \Delta t$ ein, erhält man:

$$\tilde{C}_n = \frac{1}{N}\sum_{k=0}^{N-1} y_k e^{-i\frac{n\cdot k}{N}2\pi} \tag{5.5}$$

Die $\tilde{C}_n$ aus Formel 5.5 bezeichnet man als diskrete *Fourierkoeffizienten*.

Bei dem Ausdruck $e^{-i\frac{n\cdot k}{N}2\pi} =: w_N^{-n\cdot k}$ handelt es sich um die N-ten komplexen Wurzeln von $z^N = 1$. Aufgrund deren Periodizität können nur N verschiedene Lösungen für die diskreten *Fourierkoeffizienten* $\tilde{C}_n$ bestimmt werden: $\tilde{C}_0, \tilde{C}_1, \ldots, \tilde{C}_{N-1}$

Damit erhält man für die *Fourier-Transformation* im kontinuierlichen (Formel 5.1) und ihrer Inversen die diskrete Version.

$$\text{diskrete Fourier-Transformation:} \qquad F_n = \frac{1}{N} \sum_{k=0}^{N-1} y_k w_N^{-n \cdot k} \tag{5.6}$$

$$\text{diskrete inverse Fourier-Transformation:} \qquad y_k = \sum_{n=0}^{N-1} F_n w_N^{+n \cdot k} \tag{5.7}$$

Stellt man die Summe 5.7 in Matrixform dar, so erhält man (mit $w_n^0 = 1$):

$$\underbrace{\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w_N^{1\cdot 1} & w_N^{1\cdot 2} & \cdots & w_N^{1\cdot(N-1)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & w_N^{(N-1)\cdot 1} & w_N^{(N-1)\cdot 2} & \cdots & w_N^{(N-1)\cdot(N-1)} \end{pmatrix}}_{\Omega_N} \begin{pmatrix} F_0 \\ F_1 \\ \vdots \\ F_{N-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{N-1} \end{pmatrix}$$

Es gilt:

$$\sum_{k=0}^{N-1} \underbrace{w^{k\cdot n} \cdot \bar{w}^{k\cdot n}}_{\sin^2(kn)+\cos^2(kn)} = N \quad \Rightarrow \qquad \Omega\bar{\Omega} = N \cdot I \quad \Rightarrow \quad \Omega^{-1} = \frac{1}{N}\bar{\Omega}$$

Man erhält folgende Darstellungen für die *diskrete Fourier-Transformation* (Formel 5.6) und deren Inverse (Formel 5.7):

$$\text{diskrete Fourier-Transformation:} \qquad \vec{F} = \frac{1}{N}\bar{\Omega}\vec{y} \tag{5.8}$$

$$\text{diskrete inverse Fourier-Transformation:} \qquad \vec{y} = \Omega\vec{F} \tag{5.9}$$

In beiden Fällen handelt es sich um ein Matrix-Vektor-Produkt, was einer Komplexität von $O(n^2)$ entspricht.

Die schnelle *Fourier-Transformation* (*Fast Fourier Transformation*) ist ein Algorithmus, der sich die spezielle Struktur von Ω zu Nutze macht. Mit Hilfe des *„Teile und Herrsche“* Prinzip wird der Rechenaufwand für dasselbe Ergebnis auf eine Komplexität von $O(n \log(n))$ reduziert. [27]

5.2.1 Anwendung der *Fast Fourier-Transformation*

An einem einfachen Beispiel kann man veranschaulichen, wie die *Fast Fourier-Transformation (FFT)* funktioniert. Sei $f(t)$ eine zusammengesetzte Funktion wie folgt definiert:

$$\begin{aligned} f(t) &= f_1(t) + f_2(t) \\ &\text{mit} \\ f_1(t) &= \sin(2\pi \cdot 900 \cdot t) \\ f_2(t) &= \sin(2\pi \cdot 1300 \cdot t) \end{aligned}$$

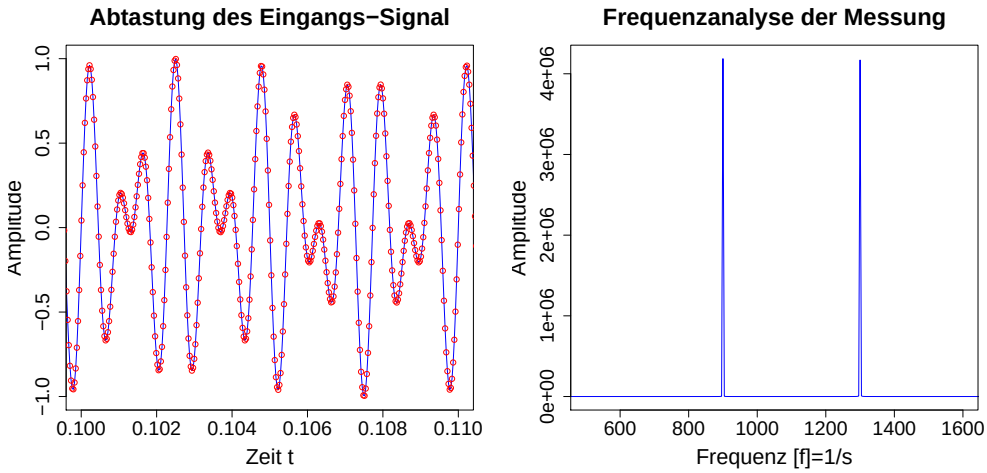


Abbildung 5.3: Beispiel für die Anwendung der *Fast Fourier-Transformation*

Das linke Bild in Abbildung 5.3 zeigt, wie das Eingangssignal $f(t)$ abgetastet wird. Diese Messwerte werden mit Hilfe der *FFT* in den Frequenzraum abgebildet. Als Ergebnis erhält man n *Fourierkoeffizienten*, entsprechend der Anzahl der Messwerte. Zu jedem Wert existiert in der Lösungsmenge auch dessen konjugiert komplexer Wert. Es handelt sich jeweils um dieselbe Frequenz, nur mit negativem Vorzeichen. Aus diesem Grund kann man die Hälfte der Koeffizienten weglassen. Von den restlichen Werten wird der Betrag bestimmt und dann in einem so genannten Periodogramm gegen die Frequenz aufgetragen (siehe rechtes Bild in Abbildung 5.3). Als Ergebnis sieht man bei den Frequenzen $900Hz$ und $1300Hz$ einen sehr großen Peak. Es handelt sich um die Frequenzen, die in dem zusammengesetztem Eingangssignal voreingestellt waren. Das lässt den Schluss zu, dass das abgetastete Signal aus zwei sich überlagernden Schwingungen besteht. Die eine hat eine Frequenz von $900Hz$, die andere von $1300Hz$.

Die Signale, die mit dem *Fix Time Quantum* Benchmark ermittelt werden, haben aber eine andere Struktur.

In Abbildung 5.4 auf der nächsten Seite sind links die Messungen (Workload) gegen die Zeit aufgetragen. Im rechten Bild ist das dazugehörige Periodogramm dargestellt. Dabei ist die Frequenzachse logarithmisch eingeteilt. Das Ergebnis ist leider nicht sehr aussagekräftig. Es sind anscheinend sehr viele verschiedenen Frequenzen zur Darstellung des gemessenen Signals notwendig.

Zwei Ursachen können dafür genannt werden. Zum einen handelt es sich um ein Messsignal, das Ungenauigkeiten enthalten kann. Das führt dazu, dass die Abstände zwischen denselben Ereignissen leicht variieren können und statt einer stark ausgeprägten Frequenz erhält man viele ähnliche Frequenzen. Es kommt zu einer Streuung um die eigentliche Frequenz.

Der zweite und wesentlichere Grund ist, dass das Eingangssignal unstetig ist. Die *Fourierreihe* versucht diese Stelle mithilfe von Sinuskurven darzustellen. Dabei wird automatisch ein Fehler entstehen, der durch Sinuskurven mit einer höheren Frequenz kompensiert werden muss. Das führt dazu, dass statt vielleicht nur einer Frequenz meh-

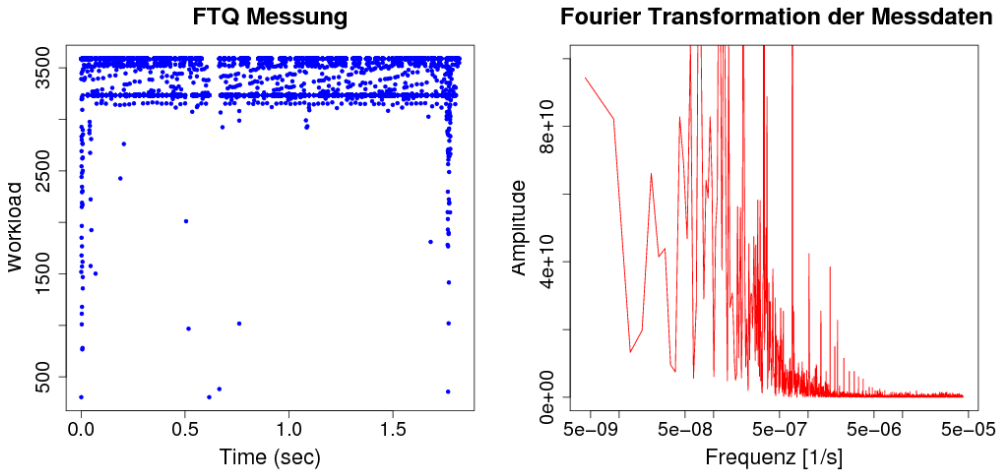


Abbildung 5.4: *FTQ* Messung und *Fourier-Transformation*

rere in der Lösungsmenge enthalten sind.

Glättung des Periodogramms als Maßnahme gegen die Streuung von den Messdaten

Da es sich beim *FTQ* Benchmark um experimentelle Messungen handelt, muss davon ausgegangen werden, dass es zu Ungenauigkeiten beim Erfassen der Daten kommen kann. Die gemessenen Störungen scheinen keine saubere Frequenz zu besitzen, sondern um einen Wert zu streuen. Das führt dazu, dass man in einem Periodogramm nicht einen starken Peak bei der entsprechenden Frequenz findet, sondern viele kleinere um diesen Punkt.

Um diesen Effekt zu kompensieren, kann das Periodogramm geglättet werden. Das geschieht – ähnlich wie bei den Dichteschätzer – durch Mittelwertbildung um einen Punkt.

Für jeden Wert F_i der *diskreten Fourier-Transformation* wird ein Intervall $[F_{i-L}, F_{i+L}]$ festgelegt. Den Mittelwert $\hat{F}_i$ bestimmt man dann wie folgt:

$$\hat{F}_i = \sum_{k=-L}^{L} m_k F_{i+k}$$

Die Gewichte m_k sollten zusammen 1 ergeben und symmetrisch um m_i nach folgender Regel verteilt sein:

$$m_{-L} \leq \ldots \leq m_0 \geq \ldots \geq m_L \quad \text{mit} \sum_{k=-L}^{L} m_k \stackrel{!}{=} 1$$

Diese Vorgehensweise wird als Glättung durch den *gleitenden Mittelwert (moving averages)* bezeichnet.

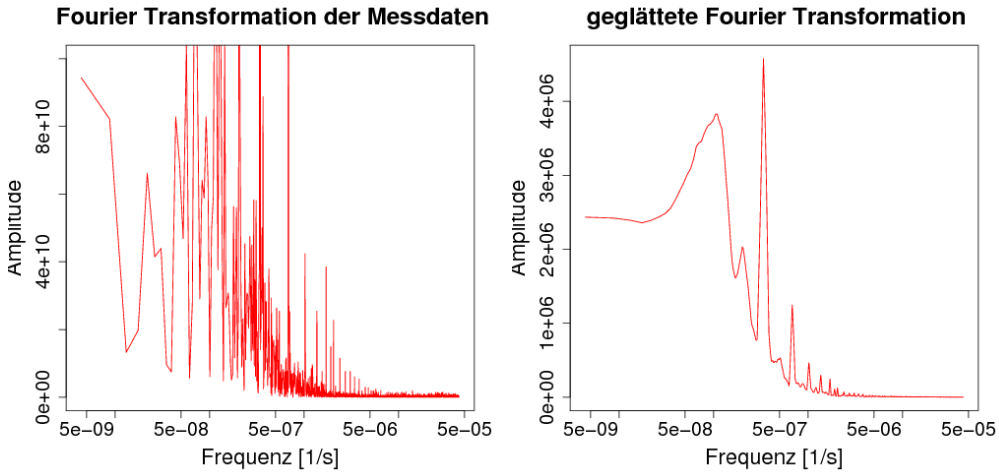


Abbildung 5.5: Glättung des Periodogramms

In Abbildung 5.5 ist dieses Verfahren angewendet worden. Links befindet sich die originale *diskrete Fourier-Transformation* und rechts die geglättete Version. Bei dieser Glättung wurden die Gewichte $\frac{1}{121}, \frac{2}{121}, \frac{3}{121}, \ldots, \frac{9}{121}, \frac{10}{121}, \frac{11}{121}, \frac{10}{121}, \frac{9}{121}, \frac{8}{121}, \ldots, \frac{2}{121}, \frac{1}{121}$ verwendet. Es sind nun wesentlich weniger Peaks zu sehen und über die verbleibenden können mit großer Wahrscheinlichkeit noch Aussagen getätigt werden.

Effekt der unstetigen Stellen

Der zweite kritische Punkt für die *Fourier-Transformation* der *FTQ* Messdaten liegt in den vielen unstetigen Stellen des Eingangssignals. Es besteht aus mehreren horizontalen Linien, zwischen denen die Messwerte immer hin und her springen. Diese Sprungstellen sind sehr schlecht durch eine Sinusschwingung darstellbar. Die bei der *Fourier* Reihenentwicklung entstehenden Fehler an der Stelle müssen durch viele Sinuskurven mit verschiedenen Frequenzen kompensiert werden. Für singuläre Messpunkte gilt das Gleiche.

Den Effekt, den ein Signal dieser Art bei der *diskreten Fourier-Transformation* erzeugt, ist in Abbildung 5.6 auf der nächsten Seite dargestellt. Im linken Bild ist ein künstliches Signal zu sehen, in dem die Messwerte konstant 4000 sind. Nur jede 100. Messung fällt um 10% geringer aus, besitzt damit den Wert 3600. Es liegt also eine absolut saubere Messung vor, bei der es genau eine Störung mit einer vorgegebenen Frequenz gibt.

Im rechten Bild sieht man das dazugehörige Periodogramm. Genau bei der Frequenz 10^{-2} ist ein Ausschlag sichtbar. Das entspricht der Vorgabe, dass alle 100 Zyklen eine Störung auftritt. Aber auch die Vielfachen Frequenzen (200, 300, 400, usw.) scheinen im Eingangssignal enthalten zu sein, obwohl dem nicht so ist. Die Ursache liegt in dem weiter oben beschriebenen Problem der Darstellung von unstetigen Stellen durch die *Fourierreihe*. Um die 3600-er Punkte darzustellen, muss eine Schwingung mit der Frequenz von 0,01/Zyklus erzeugt werden. Das führt aber zu Fehlern zwischen den einzelnen Sprungpunkten. Diese werden mit Vielfachen der eigentlichen Frequenz kompensiert

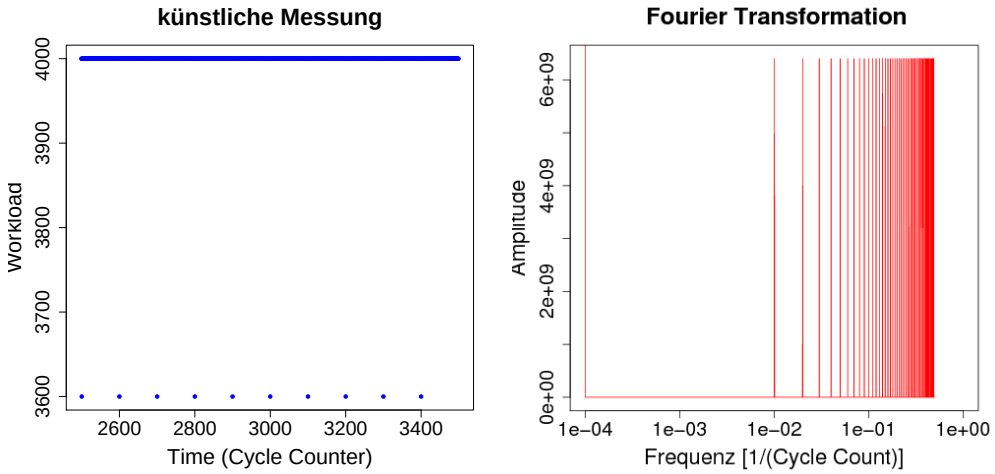


Abbildung 5.6: Ideales Signal: alle 100 Zyklen eine 400er Störung

(Obertöne).

Fazit

Findet man bei der Anwendung der *FFT* auf ein Messsignal des *FTQ* Vielfache einer markanten Frequenz, dann gehören diese Peaks zusammen.

Kapitel 6

Analyse der Benchmarks

In diesem Kapitel werden die Ergebnisse der Benchmark Messungen diskutiert. Ziel ist es, ein Verständnis für den *OS-Jitter* zu erhalten, der auf einem *High Performance Cluster* mit Linux Betriebssystem entsteht. Dabei werden die beiden Varianten *openSuSE* inklusive *KTAU* Patch und *SuSE Linux Enterprise Real Time* eingesetzt.

Es werden zuerst jeweils Messungen mit dem *Fix Time Quantum* Benchmark durchgeführt, um den lokalen *OS-Jitter* zu messen. Als zweites wird die Auswirkung der Störungen auf eine parallele Anwendung mithilfe des *MPI-Jitter* untersucht.

6.1 Testumgebung

Das *Jülich Supercomputing Centre* betreibt das Projekt *JuRoPA*[1] zur Konzipierung und Beschaffung eines neuen Supercomputer-Systems. Es soll als Nachfolgesystem für das „General Purpose“ System *JuMP*[2] dienen. Im Rahmen dieses Projektes wurde in Jülich ein Testsystem aufgebaut, um Hardware- und Softwarekomponenten zu untersuchen.

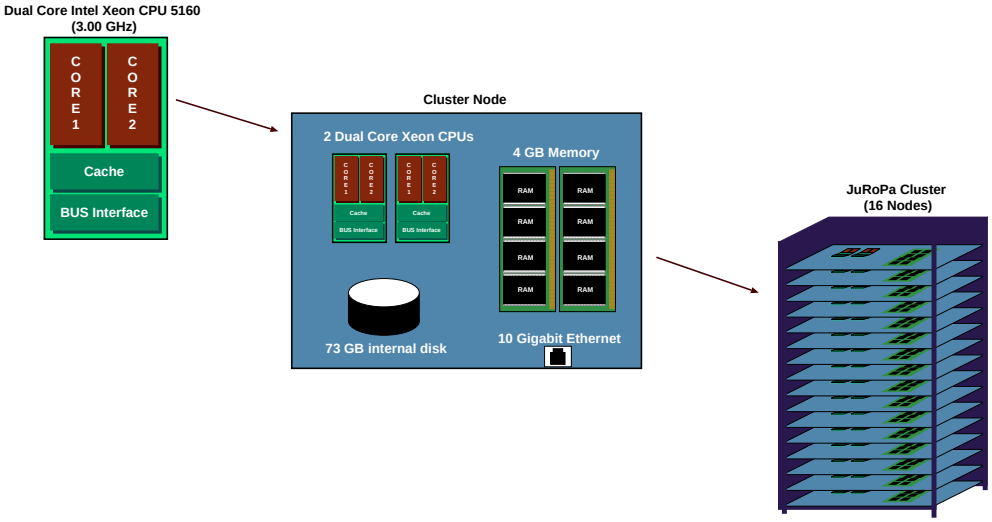


Abbildung 6.1: Aufbau des *JuRoPA* Testcluster

Dieses Cluster besteht aus 16 Knoten mit jeweils 2 *Dual Core Intel Xeon 5160* CPUs. Jede arbeitet mit eine Taktfrequenz von 3,0GHz. Die vier Prozessoren in einem Knoten

[1] *Jülich's Research on Peta-flop Architectures*
[2] *Jülich Multi Processor*

teilen sich 4 GB Hauptspeicher (siehe Abbildung 6.1 auf der vorherigen Seite). Als Verbindungsnetzwerk wird ein *10 GB Ethernet* der Firma *Myricom* eingesetzt. Jeder Knoten für sich ist ein kompletter Rechner mit eigenem Betriebssystem.

Als Cluster Management Software wird *ParaStation* eingesetzt, ein Produkt, das von einem Konsortium bestehend aus dem Forschungszentrum Jülich, der Universität Wuppertal, der Universität Karlsruhe und der Firma *ParTec Cluster Compentence Center GmbH* weiterentwickelt wird. Ursprünglich ist diese Software an der Universität Karlsruhe im Rahmen des DFG[3] Projekts „RESH“ entwickelt worden [25]. Die *ParaStation* sorgt unter anderem dafür, dass bei einem parallel arbeitenden Job jeder Prozess fest einem Prozessor zugeordnet wird (*Process Pinning*).

Alle weiteren Konfigurationen sind abhängig von dem, was gerade mit dem System getestet wird. So sind verschiedene SuSE Linux Versionen als Betriebssystem auf den Knoten installiert. Je nach Bedarf können verschiedenen Konfigurationen „on the fly“ eingespielt werden.

Des weiteren werden noch verschiedene Hardwarekomponenten wie z.B. das *Lustre* Dateisystem der Firma *SUN* untersucht. Auf zwei weiteren Knoten entwickelt und testet die Firma *ParTec* ihre Cluster Management Software *ParaStation*.

Das ganze System ist so flexibel gehalten, dass jederzeit die Konfiguration geändert werden kann, um z.B. ein einheitliches System herzustellen, so dass alle Knoten die gleiche Konfiguration bekommen.

Wenn nicht explizit erwähnt sind die Messungen in den folgenden Abschnitten auf diesem Cluster durchgeführt worden. Die *Speedstep* Technologie wird soweit möglich unterdrückt, damit eine gleichbleibende Frequenz zur Zeiterfassung sichergestellt wird.

6.1.1 openSuSE 10.2 mit KTAU

Für die Analyse des *OS-Jitter* auf dem *JuRoPA* Testcluster wurde auf dem kompletten System ein einheitliches Betriebssystem konfiguriert. In der ersten Konfiguration ist das schon laufende *openSuSE 10.2* System genutzt worden. Dafür mussten die Quellen des Linux Kernel 2.6.22.1 mit dem *KTAU* Patch versehen und kompiliert werden. Danach wurde der neue Kernel auf alle Knoten verteilt. Wichtig war dabei, *Speedstepping* für alle Prozessoren auszuschalten. In dieser Konstellation sieht man bei den Benchmarkläufen nicht nur die Störungen, sondern kann diese mithilfe des *KTAU* Protokolls einem speziellen Kernel Ereignis zuordnen.

6.1.2 SuSE Linux Enterprice Real Time (SLERT)

In der zweiten Konfiguration wurde das *SuSE Linux Enterprice Real Time (SLERT)* genutzt. Es handelt sich um ein *SLES 10* Version[4] der Firma *Novell*, bei der der Linux Kernel mit einem *Real Time* Patch versehen wurde. Grund dafür sind Überlegungen, mittel- bzw. langfristig das *Real Time* System auf dem zukünftigen *JuRoPA*-Cluster einzusetzen und dessen Fähigkeiten zu nutzen, um den *OS-Jitter* zu minimieren. Die Erweiterungen ermöglichen unter anderem, Prozesse exklusiv einem bestimmten Prozessor zuzuordnen und zu schützen, so dass kein anderer diese Resource nutzen kann.

[3] Deutsche Forschungsgemeinschaft
[4] SuSE Linux Enterprise Server

Leider gelang es nicht, den *KTAU* Patch ebenfalls in diesen Kernel zu integrieren. Ursache ist der *Real Time* Patch, der zu gravierenden Änderungen im Prozess-Scheduling des Linux Kernels führt. Die *KTAU* Modifizierungen betreffen zum großen Teil dieselben Stellen und sind deswegen in der aktuellen Version nicht kompatibel.

6.2 FTQ Messungen

Im Folgenden wird der *OS-Jitter* Effekt mit Hilfe des *Fix Time Quantum* Benchmarks auf dem *JuRoPA*-Testsystem untersucht.

Wie schon bei der Beschreibung des *FTQ* Benchmarks auf Seite 30 erwähnt, können bei diesem Benchmark zwei Optionen angeben werden. Zum einen die Anzahl der Messungen, und zum anderen die Dauer einer Messung, einem Δt. Die Anzahl wird immer auf 20.000 gesetzt.

Für das Δt wird nicht die Zeit direkt angegeben, sondern die Anzahl der Taktzyklen. Über die Frequenz des Prozessors kann die reale Zeit bestimmt werden. Bei den Messungen hier wurden 2^{18}, 2^{20}, 2^{22} und 2^{24} Taktzyklen gewählt. Da die Prozessoren mit einer Taktfrequenz von $2{,}997 GHz$ arbeiten, entsprechen diese Angabe den Zeiten $2^{18}/(2{,}997 \cdot 10^9 \frac{1}{s}) = 87{,}4688 \cdot 10^{-6} s = 87{,}4688 \mu s$, $349{,}8752 \mu s$, $1.399{,}501 \mu s$ und $5.598{,}003 \mu s$.

6.2.1 openSuSE 10.2 Kernel mit KTAU Patch

Im ersten Schritt wird das *JuRoPA*-Testcluster mit einem *openSuSE 10.2* Betriebssystem versehen, bei dem der Kernel um den *KTAU* Patch erweitert worden ist.

Bei den *FTQ* Messungen wird in erster Linie auf die 2^{18}-er Messdaten eingegangen. Für die anderen Messungen werden nur noch die Besonderheiten diskutiert. Vergleichbare Abbildung finden sich jeweils im Anhang.

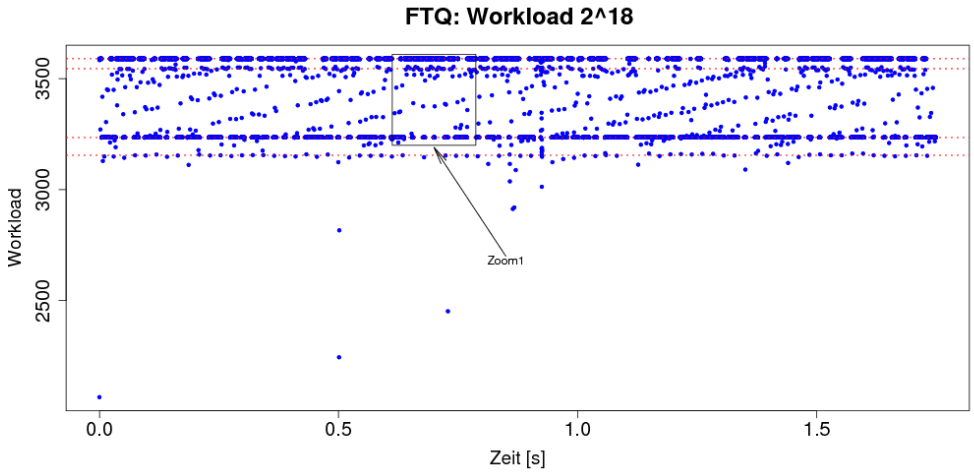


Abbildung 6.2: *FTQ* Messung auf dem *openSuSE* System

In Abbildung 6.2 ist die Messung für eine Intervallgröße von $87{,}4688 \mu s$, also 2^{18} Zyklen pro Messung, zu sehen. Sehr schön zu erkennen ist, dass es 4 horizontale Linien

gibt, jeweils bei dem *Workload* von 3155, 3235, 3545 und 3590. Die Messungen zu einer Linie werden im Folgenden als Klasse bezeichnet. Die oberste Klasse zu dem Wert 3590 kennzeichnet dabei die maximale Leistung, also die maximale Anzahl an Schleifendurchläufen, die der *FTQ* pro Messintervall schaffen kann. In dieser Klasse und in der zu dem Wert 3235 scheint der Hauptanteil der Messungen zu liegen. Das linke Bild in Abbildung 6.3, in dem ein Ausschnitt vergrößert dargestellt wird, zeigt, dass die Leistung hauptsächlich zwischen diesen zwei Werten hin und herspringt. Anscheinend treten in dem System immer wieder Störungen der Art auf, dass das Benchmark Programm in derselben Zeit 355 Iterationen weniger schafft. Das entspricht einem Verlust von ca. 10% an Leistungsfähigkeit.

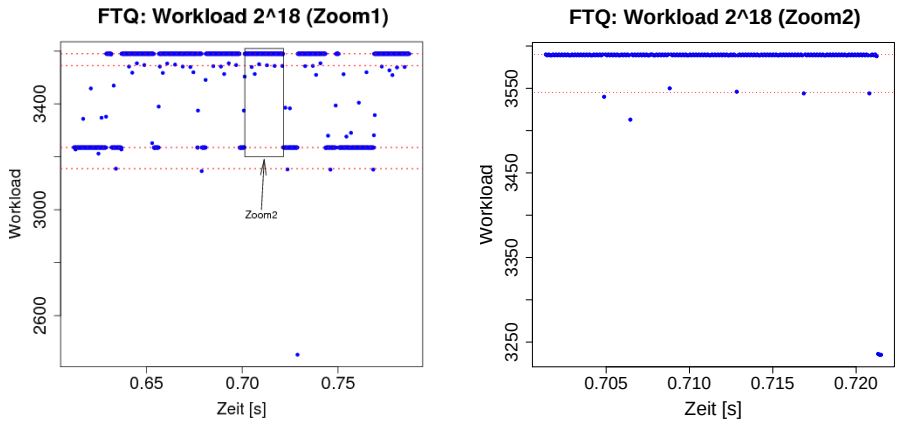


Abbildung 6.3: *FTQ* Benchmark mit 2^{18} Zyklen pro Messung (vergrößert)

Neben den beiden sehr markanten Klassen zu den Werten 3590 und 3235 ist noch eine bei 3545 direkt unterhalb des Maximums zu erkennen. In der zweiten Vergrößerung (siehe rechtes Bild in Abbildung 6.3) sieht man, dass es sich um eine punktuelle – also eine sehr kleine – Störung handelt. Diese scheint sehr regelmäßig aufzutreten.

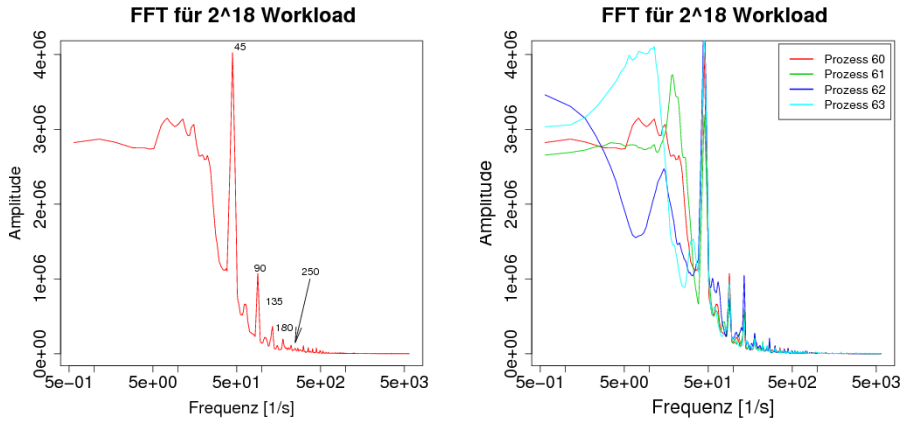


Abbildung 6.4: geglättetes Periodogramm von den vier Prozessen auf einem Knoten

Um die Frequenzen zu bestimmen, kann man mit der *Fast Fourier-Transformation* eine Frequenzanalyse durchführen. In Abbildung 6.4 auf der vorherigen Seite ist das geglättete Ergebnis der *FFT* graphisch dargestellt. Das linke Bild zeigt die Frequenzen der Störungen, die beim Prozess 60 aufgetreten sind. Das rechte Bild verdeutlicht, dass sich alle vier Prozesse auf demselben Knoten sehr ähnlich verhalten. Betrachtet man die Peaks genauer, erkennt man die Frequenz $45Hz$ und alle ihre Vielfachen. Auch eine Störung von $250Hz$ scheint vorhanden zu sein. Es zeigt sich aber, dass es nicht einfach ist, auf diesem Weg die Frequenzen der Störungen zu finden.

Deswegen wird hier ein weiterer Weg gewählt, um an die Frequenz der einzelnen Störungen zu gelangen. Es hat sich gezeigt, dass sich bei den Messungen mit 2^{18} Zyklen bzw. $87{,}4688\mu s$ vier markante Klassen in Form von horizontalen Linien gebildet haben.

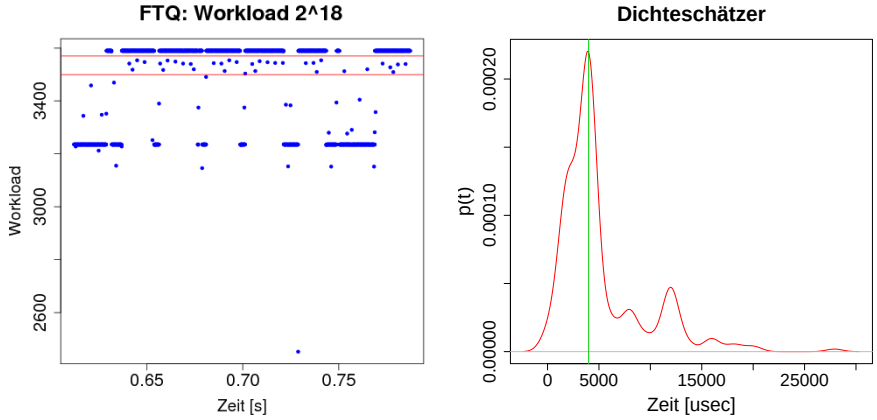


Abbildung 6.5: Analyse der ersten Störung: Links das Eingangssignal mit dem eingezeichneten Band der Klasse zum Wert 3545, rechts die Dichteschätzung für die Zeitabstände der Messwerte in diesem Band

Die zugehörigen Messwerte werden separat analysiert, indem auf die Zeitabstände der Ereignisse eine Dichteschätzung ausgeführt wird. In Abbildung 6.5 ist das Ergebnis für die Klasse zum Wert 3545 dargestellt. Es ist ein eindeutiger Peak zu sehen, der sich bei etwa $4.000\mu s$ befindet. Das entspricht einer Frequenz von $250Hz$. Ein sehr interessantes Ergebnis, wenn man sich daran erinnert, dass der *System Tick* bei dem System auf $250Hz$ eingestellt ist.

Für die nächsten beiden Linien wird genauso verfahren. In Abbildung 6.6 auf der nächsten Seite zeigt das linke Bild die Analyse der Störung mit dem 3235 *Workload*. Der Peak ist bei $18.400\mu s$, was einer Frequenz von ca. $61Hz$ entspricht. Die Graphik zeigt aber, dass die Werte sehr stark zu streuen scheinen. Die Ursache liegt in diesem Fall an dem zu geringen Stichprobenumfang.

Das rechte Bild hat dafür einen eindeutigen Häufungspunkt, und zwar bei $22.400\mu s$, also bei $44{,}6Hz$. Genau diesen Wert hat auch die geglättete *FFT* gezeigt.

Auswertung der KTAU

Aus den Messdaten des *FTQ* Benchmarks konnten bereits Erkenntnisse über die Frequenz der Störungen gewonnen werden. Da aber bei dieser Version des Betriebssystems

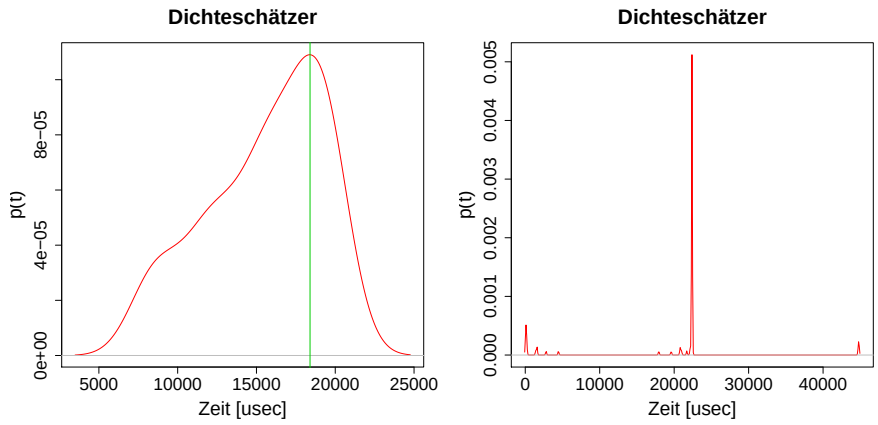
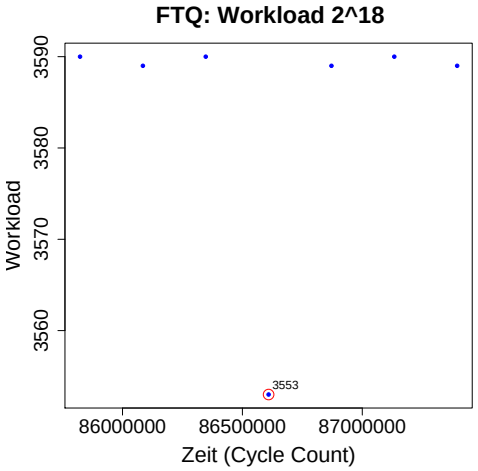


Abbildung 6.6: Links die Analyse der zweiten Störung (Klasse zum Wert 3235), rechts die Analyse der dritten Störung (Klasse zum Wert 3155)

der *KTAU* Patch eingespielt werden konnte, kann man den Kernel und sein Prozess-Scheduling beobachten. Zu jedem beliebigen Prozess können alle Aktivitäten des Scheduler protokolliert werden. Damit sollte es möglich sein, die Ursache der entsprechenden Störung zu ermitteln.



1.) Ausschnitt aus dem *FTQ* Benchmark

```
...
85822831    3590
86085010    3589
86347126    3590
86609305    3553
86783842  in   smp_apic_timer_interrupt
86784400  in   __do_softirq
86784769  in   __run_timers
86785048  out  __run_timers
86785291  out  __do_softirq
86785498  out  smp_apic_timer_interrupt
86871448    3589
87133564    3590
87395743    3589
...
```

2.) *FTQ* Daten + *KTAU* Log

Abbildung 6.7: Störung durch den *System Tick*: korrelierte Daten des *FTQ* Benchmarks mit dem *KTAU* Log

Dazu wird wieder die Messung für 2^{18} Zyklen pro Messung betrachtet. Die Messdaten aus dem *FTQ* Benchmark und die Informationen des *KTAU* Protokolls für den *FTQ* Prozess wurden in einer Datei zusammengefasst und in eine chronologische Reihenfolge gebracht. In Abbildung 6.7 sieht man links einen kleinen Ausschnitt aus der Messung und rechts den entsprechenden Auszug aus den zusammengefassten Daten. In dieser Datei enthält die erste Spalte den *Cycle Counter*, der entweder vom *KTAU* oder dem

FTQ ausgelesen worden ist. Die zweite Spalte enthält entweder den *Workload* einer *FTQ* Messung oder die *in-* bzw. *out*-Marke der protokollierten Ereignisse aus dem Kernel. In den zuletzt genannten steht als drittes Feld noch der zugehörige Funktionsaufruf.

In diesem Beispiel ist gut zu sehen, wie der Benchmark die maximale Leistung von 3589/3590 Operationen erreicht. Wenn ein Ereignis auftritt, in diesem Fall handelt es sich um einen *smp_apic_timer_interrupt*, reduziert sich die Anzahl der Schleifendurchläufe, hier auf den Wert 3553. Die zugehörige Störung ist aber so klein, dass direkt im nächsten Schritt wieder die maximale Leistung gemessen wird. Dieses Symptom wiederholt sich ständig und damit ist die Ursache gefunden für die obere Störungsklasse 3545 aus Abbildung 6.5 auf Seite 51: Der *System Tick*

Wenn man sich weitere Stellen mit demselben Effekt anschaut, wird man feststellen, dass fast immer die Messung vor dem Auftreten des Ereignisses den Leistungseinbruch hat. Das liegt an der Funktionsweise des *FTQ* Benchmarks. Jede Messung besteht immer aus der Startzeit der Schleife und der Anzahl der Schleifendurchläufe, die in ihr erreicht worden ist. Tritt also eine Störung auf, wird in der kombinierten Log-Datei der *FTQ* Messung und den *KTAU* Informationen die gemessene Störung in der Regel vor ihrem Auftreten gelistet werden. Insgesamt hängt das davon ab, wann die Störung beginnt, und in welchem Messintervall sie endet.

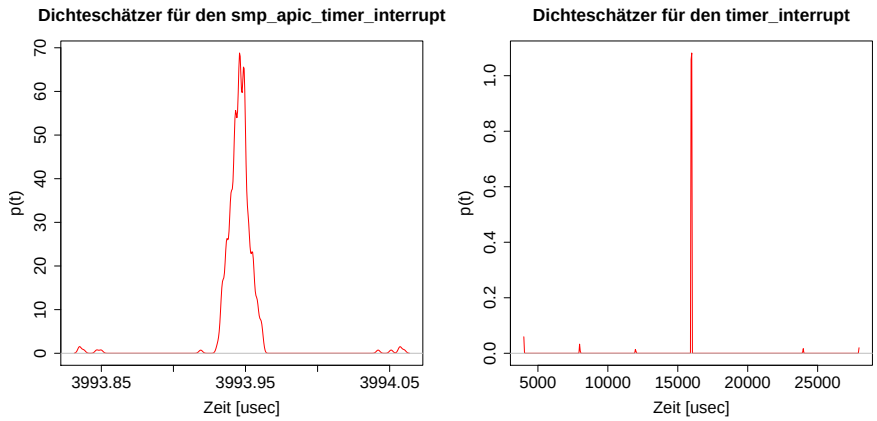


Abbildung 6.8: Verteilung der Zeitabstände zwischen den *timer_interrupts* aus dem *KTAU* Log. Im linken Bild zeigt der Dichteschätzer einen Zeitabstand zwischen den *smp_apic_timer_interrupt* von ca. $3994\mu s$ (ca. $250Hz$). Für den *timer_interrupt* im rechten Bild erhält man ca. $16ms$ (ca. $62{,}5Hz$)

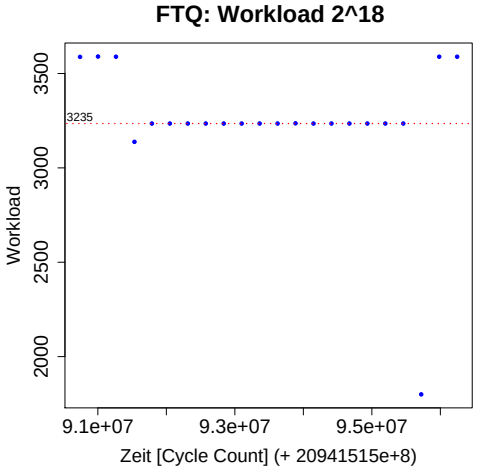
Betrachten man den *smp_apic_timer_interrupt* aus dem *KTAU* Log etwas genauer und untersucht die Zeitabstände zwischen den einzelnen Ereignissen dieser Art mit Hilfe eines Dichteschätzers, stellt man fest, dass sie fast immer mit dem Abstand von $3994\mu s$ voneinander entfernt sind (siehe linkes Bild in Abbildung 6.8). Das entspricht einer Frequenz von $250Hz$, also wieder der voreingestellten Frequenz des *System Tick* im Kernel.

Ein weiteres Ereignis, das immer wieder im *KTAU* Log auftritt, ist der *timer_interrupt*.

```
...
2094147548479656   3589
2094147548741763   3486
2094147548784090  in   timer_interrupt
2094147548788968  out  timer_interrupt
2094147549003924   3589
...
```

Wendet man dieselbe Technik wie bei dem *smp_apic_timer_interrupt* an, dann erhält man den im rechten Bild von Abbildung 6.8 auf der vorherigen Seite präsentierten Dichteschätzer mit einem Zeitabstand von ziemlich genau $16ms$, was einer Frequenz von $62{,}5Hz$ entspricht.

Der *System Tick* des Systems erklärt somit die Störung zur Klasse mit dem Wert 3545. Aber wie sieht es mit der Störung bei dem *Workload* von 3235 des *FTQ* Benchmark aus, die in der hier untersuchten Messung fast genau 50% der Werte enthält? Dazu wird ein entsprechender Abschnitt aus den Messdaten betrachtet. Wie in Abbildung 6.9 zu sehen ist, tritt laut *KTAU* Log ein *do_page_fault* auf. Es handelt sich um eine Systemfunktion des Linux Kernels mit der Aufgabe, einen Speicherseitenfehler (*Memory Page Fault*) abzuhandeln. Es wurde auf einen Speicherbereich zugegriffen, der gerade nicht im Hauptspeicher vorhanden sondern z.B. auf Festplatte ausgelagert war. Die Funktion behebt das Problem, was zu einer Verlangsamung des Prozesses führen kann [38]. Dieser Effekt wird auch bei dem *FTQ* Benchmark sichtbar. Die Leistung sinkt von 3589/3590 Zyklen auf den Wert 3235/3236 ab. Interessant ist, dass der Rücksprung auf die maximale Leistung jedes Mal nach einem Interrupt geschieht. In diesem Fall tritt ein *__sched_text_start* auf, und die Leistung bricht bis auf 1800 ein. Im nächsten Schritt erreicht das Programm wieder den vollen Wert 3589/3590. Leider ist dieser Zusammenhang nicht immer zu sehen. Es gibt auch viele Einträge, bei denen anscheinend ohne ersichtlichen Grund dieser Leistungseinbruch auftritt.



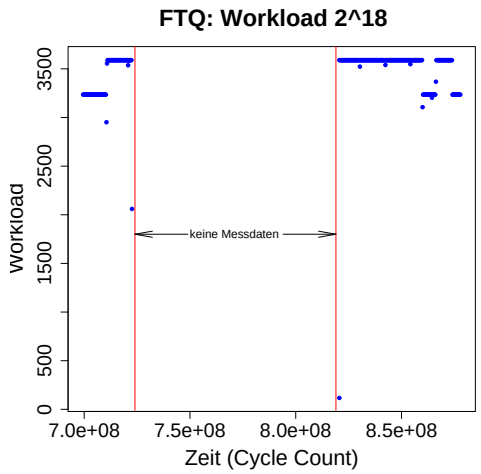
1.) Ausschnitt aus dem *FTQ* Benchmark

```
...
2094151590740133   3588
2094151591002240   3590
2094151591264419   3589
2094151591527381  in   do_page_fault
2094151591533897  out  do_page_fault
2094151591534419   3138
2094151591788696   3235
...    (bleibt immer bei 3235)
2094151595458707   3235
2094151595720850   1800
2094151595814045  in   __sched_text_start
2094151595906763  out  __sched_text_start
2094151595983029   3589
2094151596245145   3589
...
```

2.) *FTQ* Daten + *KTAU* Log

Abbildung 6.9: Störung durch *Memory Page Fault*: korrelierte Daten des *FTQ* Benchmarks mit dem *KTAU* Log

Die Frage ist jetzt, warum die Leistung um etwa 10% einbricht. Wenn ein anderer Prozess parallel auf dem Prozessor rechnet, würde man erwarten, dass der Scheduler jedes Mal einen *KTAU* Eintrag erzeugt, wenn der Prozess gewechselt wird. Man würde auch erwarten, dass der *FTQ* Benchmark in der Zeit gar nicht rechnen kann. Eine weitere Erklärung wäre, dass der Prozessor heruntergetaktet wird, wenn diese Eigenschaft nicht ausgeschaltet worden wäre. Es ist aber davon auszugehen, dass es sich um eine dem *Page Fault* ähnliche Ursache handelt (z.B. einem *Cache Miss*).



1.) Ausschnitt aus dem *FTQ* Benchmark

```
...
252586721804454    3589
252586722066561    3589
252586722328677    3590
252586722590856    2060
252586722741732   in    smp_apic_timer_interrupt
252586722742488   in    __do_softirq
252586722744486   in    __run_timers
252586722744801   out   __run_timers
252586722745062   out   __do_softirq
252586722745296   out   smp_apic_timer_interrupt
252586722745971   in    __sched_text_start
252586820617308   out   __sched_text_start
252586820624013    117
252586820632698    3590
252586820894895    3589
252586821157011    3589
...
```

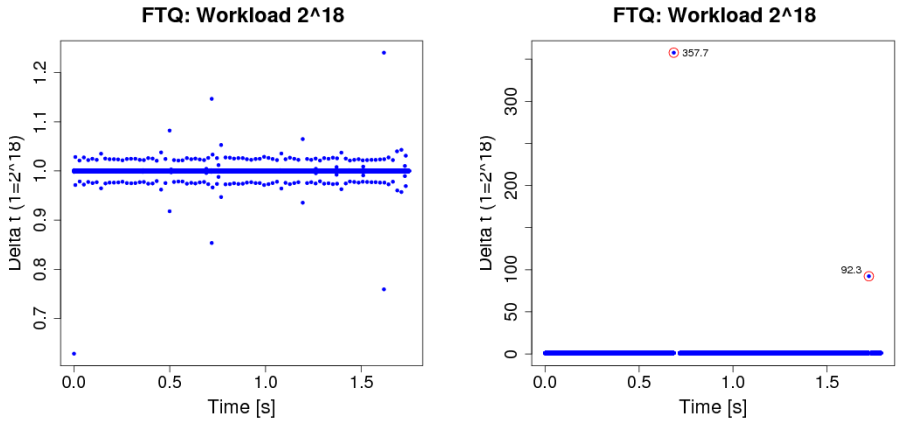
2.) *FTQ* Daten + *KTAU* Log

Abbildung 6.10: Störung über mehrere Messintervalle: korrelierte Daten des *FTQ* Benchmarks mit dem *KTAU* Log

Eine weitere Auffälligkeit gilt für die Funktion *__sched_text_start* bzw. *tasklet_action*. Fast immer, wenn der *Workload* unter den Wert 3000 springt, wird von *KTAU* einer der beiden Funktionsaufrufe mitprotokolliert. Der Scheduler unterbricht den *FTQ* Benchmark für einen anderen Prozess. Welche Auswirkung so eine Störung auf die Messungen hat, sieht man in Abbildung 6.10. In der Graphik ist eine Lücke zu erkennen, in der keine Messergebnisse vorliegen. Schaut man sich den entsprechenden Teil im Protokoll rechts an, kann man den Verlauf genau nachvollziehen. Vor der Störung befand sich die Leistung auf dem Maximum von 3590. Der nächste Wert 2060 zeigt schon eine Störung. In diesem Messintervall treten auch die Ereignisse auf, die von *KTAU* erfasst worden sind. Betrachtet man sich die Zeiten des *__sched_text_start* genau, sieht man, dass zwischen Beginn (*in*) und Ende (*out*) sehr viel Zeit vergeht. Umgerechnet auf das voreingestellte Intervall von 2^{18} wird diese Zeit das 373-fache (ca. 32,6ms) übertroffen. Es fehlen also entsprechend viele Messwerte, was dann zu der Lücke in der Graphik führt. Der Wert in der erste Messung nach der Unterbrechung beträgt 117. Das bedeutet, dass die Störung bis weit in dieses Intervall hineinragt, also die Messung „zu spät“ beginnt.

Fazit für die Funktionsweise des *Fix Time Quantum* Benchmark

Eine wichtige Erkenntnis ist, dass die Intervalllänge von 2^{18} angibt, wie groß Störungen sein dürfen, um von dem *FTQ* Benchmark vollständig erfasst zu werden. Unterbrechungen, die kleiner als diese Intervalllänge sind, starten in der n-ten Messung und enden spätestens in der $(n+1)$-ten Messung. Sind sie größer, kann es passieren, dass eine komplette Messung fehlt. In Abbildung 6.11 sind zwei Graphiken zu sehen, bei denen



alle Störungen $< 2^{18}$ zwei Störungen $>> 2^{18}$

Abbildung 6.11: automatisches Justieren der Zeitintervalle (bei 2^{18}-er Intervallen): Links relativ störungsfrei, rechts mit zwei gravierenden Störungen

zu jeder Messung (Zeit auf der X-Achse) der Zeitabstand zur nächsten Messung (Δt auf der Y-Achse) aufgetragen ist. Im linken Bild ist sehr gut eine horizontale Achsensymmetrie um den Wert „1“ zu erkennen. Man sieht, dass fast immer das Zeitintervall von 2^{18} Zyklen eingehalten wird. Kommt es zu einer Abweichung, d.h., ist das Intervall wegen einer Störung von n Zyklen überschritten worden, so dass das 1,1-fache von 2^{18} erreicht wird, dann wird das nächste Intervall um genau diese Anzahl verkürzt (das 0,9-fache). Das Ende eines Messintervalls ist fest vorgegeben, aber der Startpunkt der Messung kann durch eine Störung verzögert werden.

Die Symmetrie geht verloren, wenn mehr als ein Intervall durch Wartezeiten komplett übersprungen wird. Im rechten Bild ist das zu sehen. Es gibt eine Störung, die über das 350-fache von 2^{18} andauert. Deswegen ist auf der horizontalen Linie auch wirklich eine Unterbrechung zu sehen, weil der nächste Messpunkt entsprechend weiter rechts liegt. Würde man die Linie im rechten Bild entsprechend dem linken vergrößern, würde sich bis auf die beiden Punkt ein sehr ähnlicher Verlauf ergeben.

Weitere Messungen mit größeren Intervallen

Es wurden auch Messungen mit größeren Zeitintervallen (2^{20}, 2^{22} und 2^{24}) durchgeführt. Mit wachsenden Intervallgrößen wird es immer schwieriger, die kleinen Störungen zu unterscheiden und auch bei der Datenanalyse die richtige Frequenz zu extrahieren. Es findet eine Vermischung der Signale statt. Dafür werden die gröberen Störungen besser erfasst. Das liegt auch darin begründet, dass bei den Benchmarkläufen die Anzahl

der Messungen immer konstant auf 20.000 gesetzt wurde, und damit bei den größeren Intervallen ein entsprechend langer Zeitraum erfasst wurde. Der Lauf mit 2^{24} Zyklen pro Messung dauert $2^6 = 64$ mal so lange wie der mit 2^{18} und erhöht die Wahrscheinlichkeit, eine solche (seltene) Störung zu sehen.

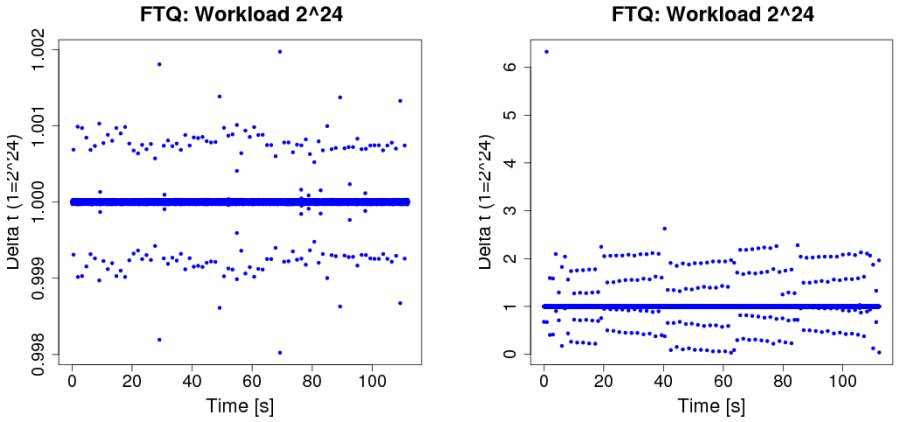


Abbildung 6.12: automatisches Justieren der Zeitintervalle (bei 2^{24}-er Intervallen): Links relativ störungsfrei, rechts mit Störungen

Ein weiteres interessantes Phänomen kristallisiert sich heraus, wenn die Messungen miteinander verglichen werden. Wie bereits beschrieben, besitzt jeder Knoten des Testsystems 4 Prozessoren. Betrachtet man die Messungen, die auf diesen vier parallel stattgefunden haben, verhalten sich immer drei sehr ähnlich, während der vierte einer größeren Störung ausgesetzt ist. Dabei ist es zufällig, welcher Prozessor auf den unterschiedlichen Knoten betroffen ist. In Abbildung 6.12 ist dieser Effekt veranschaulicht. Das linke Bild zeigt die Messung auf einem Prozessor mit wenig Störung, während das Rechte den Prozessor zeigt, auf dem noch etwas anderes immer wieder wesentlich den Benchmark stört. Untersucht man in diesem Fall die Wiederholungsrate, so erhält man eine Frequenz von $1Hz$. Zwar findet man auch auf den anderen Prozessoren eine Störung mit der gleichen Frequenz, die scheint aber nicht so viel Rechenzeit zu beanspruchen. Nimmt man diese Punkte weg und betrachtet die Linie um die „1“ etwas genauer, dann verhält die sich sehr ähnlich zu dem linken Bild.

Es wird sich um eine Störung durch einen Daemon des Betriebssystem handeln, der einmal pro Sekunde aktiv wird, um seine Aufgabe zu erledigen.

6.2.2 SuSE Linux Enterprise Real Time 10 (SLERT)

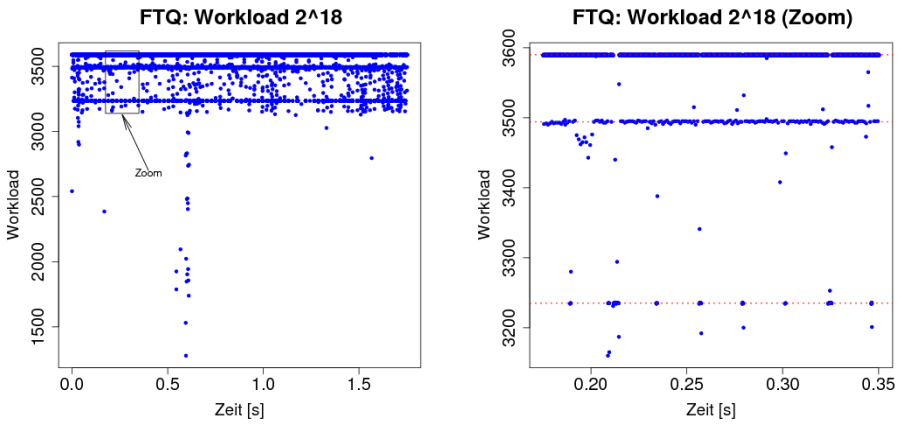
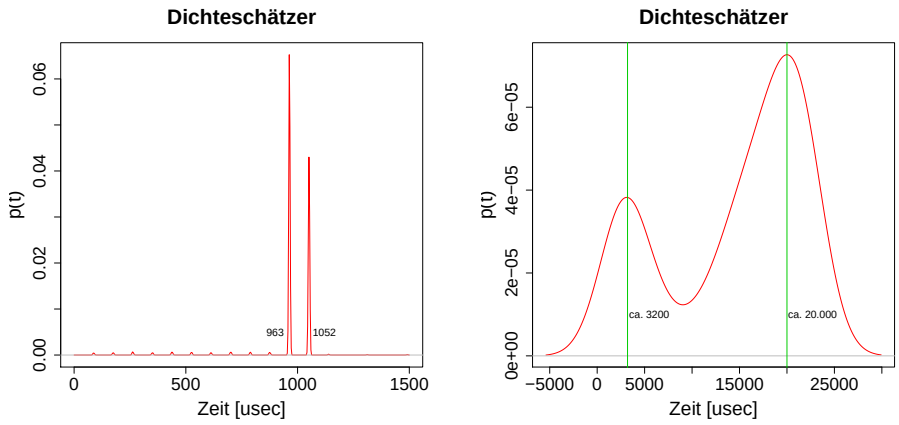


Abbildung 6.13: Messungen auf einem *Real Time* System mit einem Messintervall von 2^{18}

Für das *SuSE Linux Enterprise System* mit eingebautem *Real Time* Patch steht leider keine *KTAU* Version zur Verfügung. Trotzdem lassen sich mit Hilfe der Ergebnisse aus dem letzten Abschnitt einige Rückschlüsse auf die Ursache der Störungen gewinnen. Es wird das Verhalten untersucht, wenn der Benchmark ohne Priorisierung gestartet wird, d.h. es wird ihm keine höhere Priorität gegeben, um ihn vor Unterbrechungen zu schützen. In Abbildung 6.13 ist eine entsprechende *FTQ* Messung dargestellt. Die Länge der Zeitintervalle beträgt wiederum 2^{18} Zyklen und für die Anzahl der Messungen wurde 20.000 gewählt. In dem vergrößerten Ausschnitt rechts sind wieder sehr gut drei horizontale Klassen zu erkennen. Die oberste zeigt die maximale Leistung an und beträgt – genau wie bei der Messung auf dem *openSuSE* System – 3590 Zyklen. Man erkennt, dass der Kernel mit Echtzeitfähigkeiten die Leistungsfähigkeit für die Rechnung nicht beeinflusst, wenn der Prozess ohne spezielle Privilegien arbeitet.

Analysiert man nach demselben Prinzip wie im letzten Abschnitt die zweite und dritte Störungsklasse, erhält man die Dichteschätzer, wie sie in Abbildung 6.14 auf der nächsten Seite zu sehen sind. Dabei scheinen sowohl im linken als auch im rechten Bild jeweils zwei Frequenzen vorhanden zu sein. Links entsprechen sie ca. $959Hz$ (alle $1052\mu s$) und $1038Hz$ (alle $963\mu s$). Das ist interessant, wenn man weiß, dass auf dem *Real Time* System der *System Tick* mit einer Frequenz von $1000Hz$ arbeitet. Das gilt jedoch nur, wenn der Prozessor unter Last ist. Während des Benchmarklaufs wird das der Fall sein.

Der Grund, warum eine Störung, die in einer konstanten Frequenz auftritt, zwei Werte beim *FTQ* Benchmark liefern kann, liegt an der Diskretisierung der Zeit durch die Intervalle. In Abbildung 6.15 ist dieser Effekt exemplarisch dargestellt. Dort wird eine Störung mit einem Abstand gemessen, der 1,5 mal so groß ist wie das Messintervall (X-Achse). Das führt dazu, dass sich die Messdaten in zwei Gruppen unterteilen lassen. Die erste Gruppe enthält die Störungen, die in jedem Intervall auftauchen, und die zweite Gruppe besteht aus denen, die jedes zweite Intervall auftauchen. Das ergibt die Frequenzen von 1 Störung pro Intervall und 0,5 Störung pro Intervall. Der richtige Wert



1. Störung 2. Störung

Abbildung 6.14: Dichteschätzer für die beiden Störungsklassen im *FTQ* Benchmark

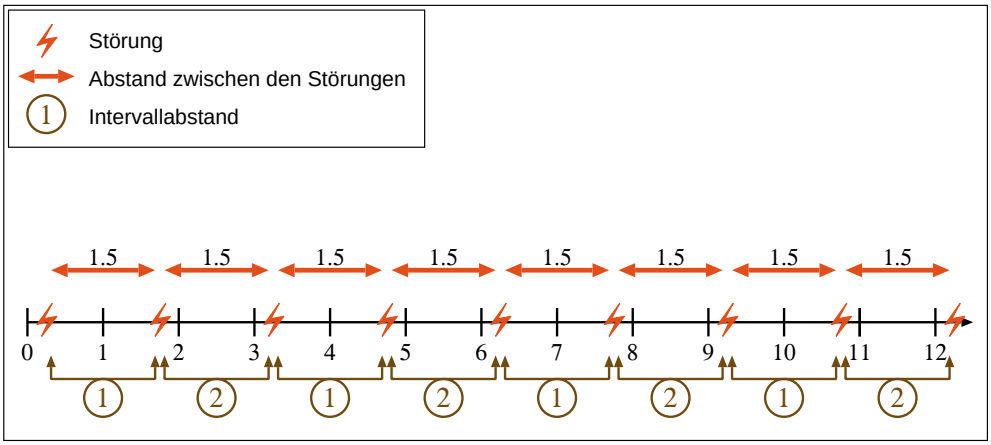


Abbildung 6.15: Entstehung von zwei Frequenzen beim *FTQ* Benchmark durch dieselbe Störung

von 0,75 liegt genau in der Mitte. Das erkennt man auch daran, dass sich die Störungen in genau zwei gleich große Mengen unterteilen.

Derselbe Effekt liegt auch in den Messungen vor. Bildet man die Differenz zweier Störungen mit den Frequenzen $959Hz$ und $1038Hz$, so erhält man ziemlich genau die Länge eines Messintervalls.

$$1052\mu s - 963\mu s = 89\mu s \approx 87,85\mu s = \frac{2^{18}}{2,9925GHz}$$

Daraus folgt, dass die Frequenzen $959Hz$ und $1038Hz$ zu einer einzigen Störung gehören, deren eigentliche Frequenz bei ca. $1000Hz$ liegt. Da die Menge der Störungen vom Typ $1038Hz$ laut Dichteschätzer größer ist, wird die eigentliche Frequenz etwas näher an diesem Wert liegen als an den $959Hz$.

Im rechten Bild von Abbildung 6.14 sind auch zwei Frequenzen zu erkennen, und zwar $312,5Hz$ (alle $3,2ms$) und $50Hz$ (alle $20ms$). Störungen dieser Klasse führen zu

eine Leistung von 3235 statt 3590 Zyklen. Es ist ein ähnlicher Effekt wie bei den *FTQ* Messungen auf dem *openSuSE* System auf Seite 52 zu erkennen.

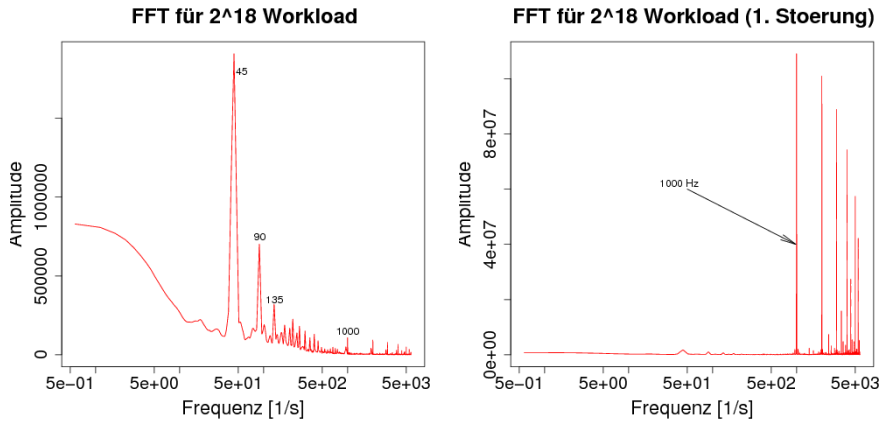


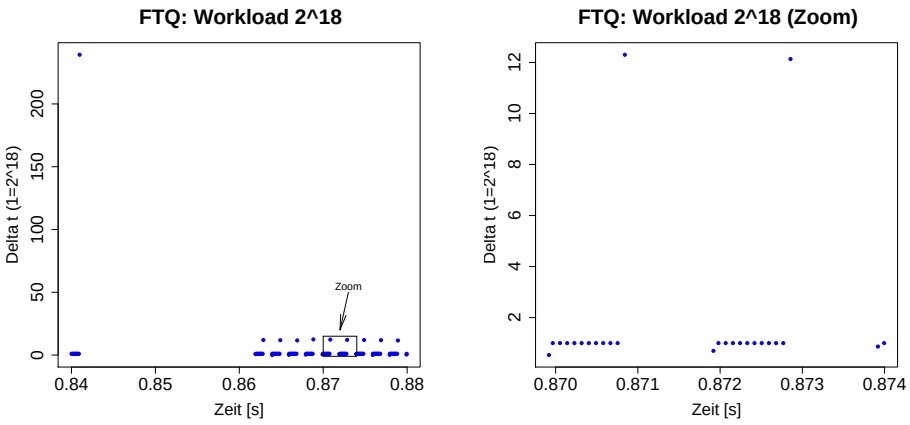
Abbildung 6.16: Periodogramm der *FTQ* Messung auf dem *Real Time* System mit einer *Workload* von 2^{18} Zyklen. Links: komplette Messung, rechts: nur die Messungen mit dem *System Tick* als Störung (1. Störung)

Wendet man die *Fast Fourier-Transformation* auf die Messdaten an, erhält man ein Periodogramm (siehe linkes Bild von Abbildung 6.16). Es taucht die Frequenz $45Hz$ und ihre Vielfachen auf, ähnlich wie bei den Messungen auf dem *openSuSE* System. Ein weiterer markanter Peak ist bei der Frequenz von $1000Hz$ zu sehen. Ursache wird der *System Tick* sein. Beim rechten Periodogramm sind die Messdaten vor der *FFT* gefiltert worden. Alle Messungen, bei denen sich der *Workload* nicht im Intervall $[3480, 3510]$ befand, wurden auf „0“ gesetzt. Wendet man wieder die *FFT* an, erhält man als Ergebnis die Frequenz $1000Hz$ und deren Vielfache (siehe rechtes Bild in Abbildung 6.16).

Im Anhang sind die Messungen auf allen vier Prozessoren eines Knotens dargestellt (siehe Abbildung A.7 auf Seite 103). Dort sieht man das typische Verhalten, wie es auf allen Knoten häufig wiederzufinden war. Im Großen und Ganzen verhalten sich alle vier Prozessoren sehr ähnlich – das Maximum liegt bei 3590 Zyklen, es gibt eine Störungsklasse durch den *System Tick* und eine weitere für den Wert 3235. Nur bei Prozessor 8 tritt eine gravierende Störung in der zweiten Hälfte der Messung auf.

Trägt man wieder gegen die Zeit jeder Messung den Abstand zur nächsten Messung ein (geteilt durch die Länge des Messintervalls, also hier durch 2^{18}), und betrachtet den Anfang der Störung etwas genauer (siehe Abbildung 6.17 auf der nächsten Seite), so erkennt man eine sehr große Unterbrechung zu Beginn der Störung von dem 239-fachen des Messintervalls (ca. $20{,}9ms.$). Direkt danach beginnt ein Bereich, in dem in regelmäßigen Abständen der Benchmark unterbrochen wird. schaut man auf das Verhältnis der Unterbrechung zur Länge der Unterbrechung, sieht man, dass nach 12 Messungen genau eine Unterbrechung von 12 Zeitintervallen folgt.

Erklären lässt sich das damit, dass ein weiterer Prozess gestartet wird und den Prozessor anfordert. Der Scheduler wird aktiv und unterbricht den *FTQ* Benchmark. Es hat den Anschein, dass ein neuer Prozess erst einmal eine gewisse Zeit alleine rechnen darf. Denn die Dauer der ersten Unterbrechung ist wesentlich länger, als der Scheduler



Anfang der Störung — 12 Messungen – Lücke von 12×2^{18}

Abbildung 6.17: automatisches Justieren der Zeitintervalle: große Störung auf Prozessor 8

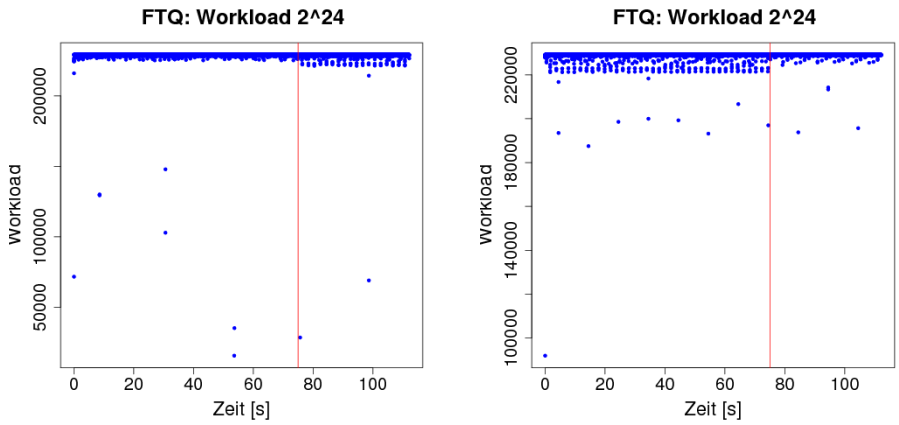
benötigen kann, die zu dem neuen Prozess gehörenden Daten aus dem Hauptspeicher in den Cache zu laden. Nachdem die erste Rechenphase beendet ist, sind die Störung und der *FTQ* Benchmark gleichwertig. Jeder bekommt genau die Hälfte der Zeit zugewiesen.

Im Anhang sind noch die weiteren Messungen für die Messintervalle 2^{20}, 2^{22} und 2^{24} abgebildet. Betrachtet man z.B. die Messungen für 2^{20} genauer, sieht man wieder die gleichen Störungsklassen wie bei 2^{18}. Bei Betrachtung der Zeiten fällt auf, dass an vielen Stellen der Faktor 4 auftaucht. Wird der 2^{20}-er Benchmark durch einen Prozess gestört, werden immer drei Messungen durchgeführt und drei werden ausgelassen. Bei den 2^{18}-er Messungen war das Verhältnis noch 12 : 12.

Das gleiche Verhältnis findet man auch für die 2^{20}-er Messung bei der Störungsklasse zum Wert von 12.944 Zyklen (zweite Störung). Das bedeutet, gegenüber dem Maximum von 14.363 Zyklen gehen 1419 verloren. Bei 2^{18} waren es 355 Zyklen (und $4 \times 355 = 1420$). Bei der Störung durch den *System Tick* stimmt dieser Zusammenhang jedoch nicht. Dort ist bei den 2^{18}-er und den 2^{20}-er Messungen der gleiche Leistungseinbruch von 100-150 Zyklen im Mittel zu beobachten.

Eine weiterer interessanter Effekt ist auf dem Prozessor 1 und dem Prozessor 2 bei der 2^{24}-er Messung in Abbildung 6.18 auf der nächsten Seite zu sehen. Auf Prozessor 2 taucht anfangs alle 2 Sekunden eine Störung auf, die sich bis ca. $\frac{2}{3}$ der Messung wiederholt. Dann verschwindet sie, dafür erscheint zum selben Zeitpunkt auf Prozessor 1 die gleiche Art der Störung. Eine mögliche Erklärung ist ein Prozess, der zunächst auf Prozessor 2 rechnet, und dann auf Prozessor 1 weiterrechnet.

Auf den Abbildungen im Anhang ist auch gut zu sehen, dass auf einem der vier Prozessoren eines Knotens immer wieder eine größere Störung sichtbar wird. Bei 2^{18}-er und 2^{20}-er Messintervallen sind die Abstände noch so groß, dass sie nicht unbedingt während des Benchmarklaufs erfasst werden, aber ab 2^{22} treten sie immer zweimal auf, und bei den 2^{24}-er Messungen dementsprechend 8 Mal. Für ca. 0,7 Sekunden ist dieser Prozess aktiv und konkurriert mit dem *FTQ* Prozess, und wird alle 15,7 bis 16 Sekunden aktiv.



Prozessor 1 Prozessor 2

Abbildung 6.18: wechselnder Prozess bei einem FTQ-Benchmark mit 2^{24}-er Intervallen: Die Störung „wandert“ nach $75s$ von Prozessor 2 auf Prozessor 1 (selber Knoten)

6.2.3 Induzieren einer künstliche Störung

Wie bereits erwähnt ist es nicht immer möglich, den *KTAU* Patch in jede Konfigurationen des Linux Kernels einzupflegen.

Deswegen werden im Folgenden Messungen gezeigt, bei denen eine künstliche Störung in das System eingefügt worden ist. Es wird die Idee aus dem *Selfish Detour* Benchmark aufgenommen (siehe Seite 30). Über diese Störung wissen wir dann, in welchem Abständen sie auftritt und wie lange sie dauert. Damit sollte sie bei der Auswertung der Messdaten identifizierbar sein. Der Benchmark läuft jeweils auf vier Prozessoren eines Knotens, wobei auf dem ersten und dem vierten Prozessor ein Prozess gestartet wird, der immer 0,1 Sekunden arbeitet ($10^5 \mu s$) und sich dann für 10 Sekunden schlafen legt ($10^7 \mu s$).

Der Befehl

```
numactl -physcpubind=0 ./dwork -s 10000000 -w 100000
```

bindet den Prozess an den ersten Prozessor mit der laufenden Nummer „0“. So wird die gerade beschrieben Störung von 10 Sekunden Warten und dann 0,1 Sekunden Arbeiten auf dem 1. Prozessor des Knoten ausgelöst.

In Abbildung 6.19 auf der nächsten Seite sind die *FTQ* Messungen auf den vier Prozessoren zu sehen. Ein Messintervall beträgt dabei 2^{24} Zyklen. Zum einen sieht man wieder den Prozess, der von Prozessor 1 auf Prozessor 0 wechselt, wie es auch in Abbildung 6.18 zu sehen ist. Wesentlich ist vielmehr, dass der Prozessor 1 und der Prozessor 2 relativ störungsfrei sind. Dagegen werden Prozessor 0 und Prozessor 3 recht deutlich gestört. Zusätzlich scheint noch eine weitere Störung auf Prozessor 0 zu existieren.

In Abbildung 6.20 auf Seite 64 wurde wieder der Abstand zwischen zwei Messpunkten gegen die Zeit aufgetragen. Für den Prozessor 3 ist die künstlich in das System eingebrachte Störung gut sichtbar. Alle 10 Sekunden taucht ein Prozess auf, der immer nach demselben Muster abläuft. Interessant ist das linke Bild für den Prozessor 0. Betrachtet

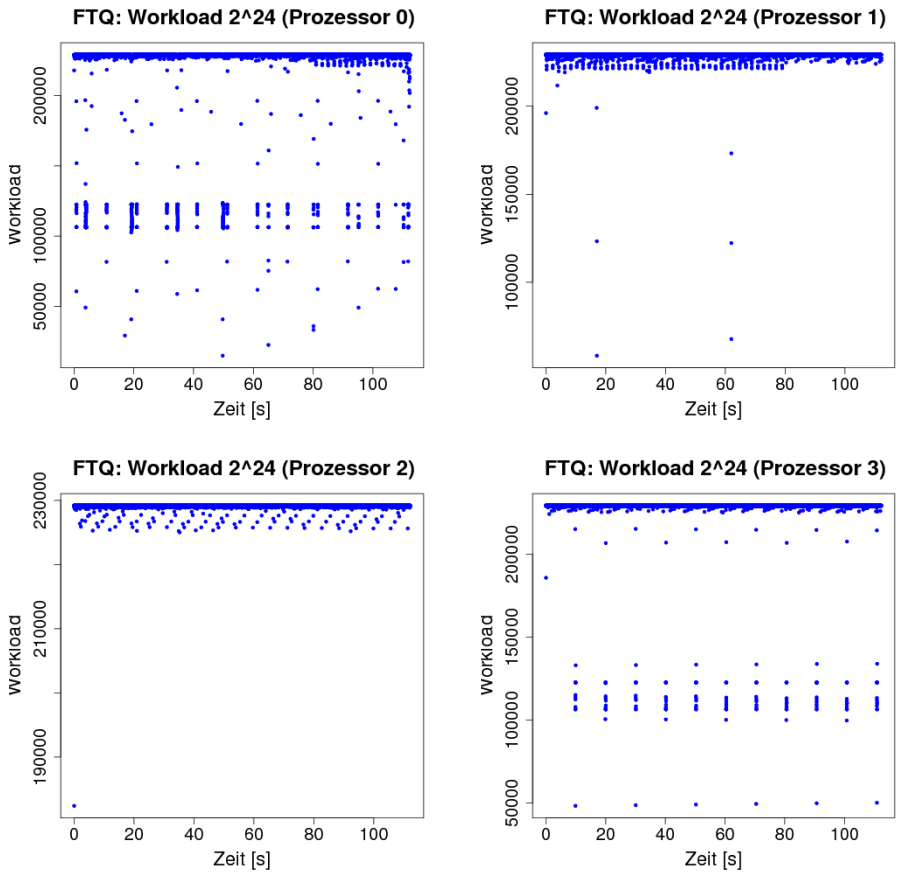


Abbildung 6.19: Messung einer künstlichen Störung: Auf dem 1. und 4. Prozessor eines Knoten wurde eine künstliche Störung erzeugt

man dieses genau, erkennt man zum einen das gleiche Störmuster wie es auch bei Prozessor 3 erscheint. Um es hervorzuheben, ist es mit Kreisen und Kästen markiert. Mit Pfeilen ist eine weitere Störung gekennzeichnet, die ca. alle 15 Sekunden auftaucht.

Der Ablauf der unbekannten Störung hat große Ähnlichkeit mit der, die in Abbildung 6.17 auf Seite 61 zu sehen ist. Am Anfang entsteht in diesem Fall eine Unterbrechung von $21ms$, danach teilen sich die beiden Prozesse den Prozessor: Eine Millisekunde rechnet der Benchmark, bis er für eine Millisekunde unterbrochen wird.

6.2.4 Overhead des Benchmarks

Bei der Beschreibung des *FTQ* Benchmarks in Abschnitt 4.2.2 wurde dargelegt, dass der Algorithmus vor jeder Messung die aktuelle Zeit bestimmt. Per Bitoperationen wird daraus der Endzeitpunkt errechnet. Dabei wird durch Löschen der niederwertigen Bits – abhängig von der Länge des gewählten Zeitintervalls – dafür gesorgt, dass die Abbruchbedingung für die Messschleifen nahezu immer exakt den vorgegebenen Abstand voneinander haben. Das ist nur dann nicht der Fall, wenn eine Störung auftritt, die dafür sorgt, dass ein ganzes Messintervall lang der Benchmark nicht rechnen durfte. In

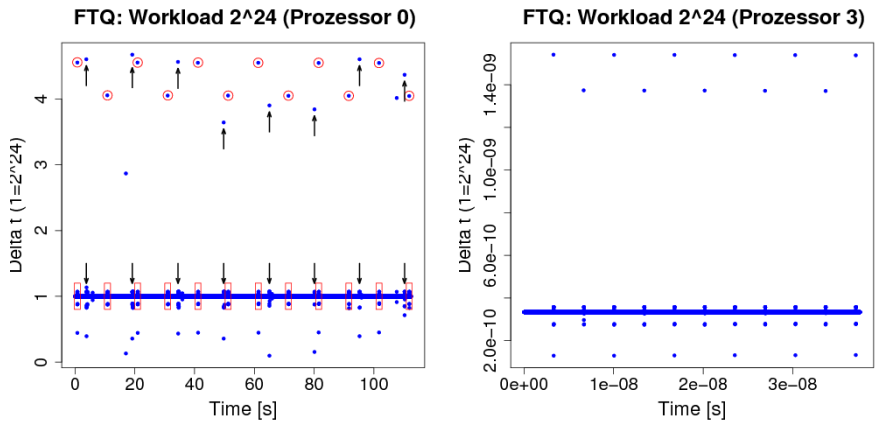


Abbildung 6.20: Analyse der künstlichen Störung: Links ist die künstliche Störung rot umrahmt, und die unbekannte Störung mit Pfeilen markiert. Rechts sieht man nur die künstliche Störung

der Ausgabedatei `ftq_times.dat` stehen aber nicht diese Zeiten, sondern die vor der Messung bestimmten Zeiten. Der *Overhead*, der zwischen den einzelnen Messungen entsteht, ist somit der Abstand zwischen dem Endzeitpunkt der letzten Messung und der erfassten Startzeit direkt vor der Messung.

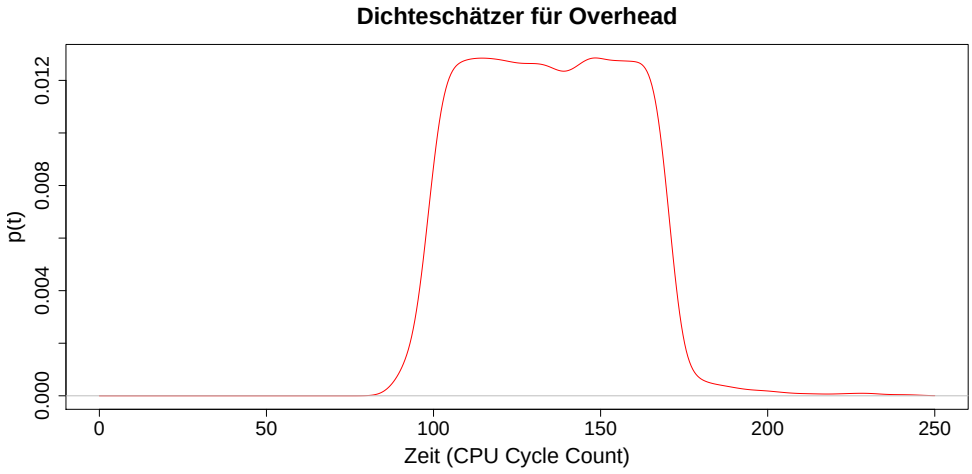


Abbildung 6.21: Verteilung des Overheads beim *FTQ* Benchmark

Diese Zeitunterschiede wurden auf dem *openSuSE* und auf dem *SuSE Linux Enterprise Real Time* System sowohl mit kleinen (2^{18}) als auch mit großen (2^{24}) Zeitintervallen untersucht. Der *Overhead* hat sich immer gleich verhalten. Er dauerte mindestens 90 Zyklen und war in der Regel spätestens nach 160 Zyklen fertig. Nur wenn während dieser Zeit eine Störung auftrat, waren die Zeiten wesentlich größer. In Abbildung 6.21 ist die Verteilung für den *Overhead* dargestellt. In diesem Fall wurde eine *FTQ* Messung mit einem *Workload* von 2^{24} Zyklen auf dem *Real Time* System durchgeführt.

Interessanterweise ist kein eindeutiger Peak zu sehen, sondern in dem Bereich von 90 bis ca. 150 Zyklen scheint die Dauer des *Overhead*s gleichverteilt zu sein. Vielleicht trifft man hier an die Grenzen der Genauigkeit für die Zeitmessung mittels des *Time Stamp Counter*.

Da es sich beim *Overhead* um eine recht kleine und konstante Zeitspanne handelt, kann er bei den Messungen vernachlässigt werden.

6.2.5 Fazit

Es hat sich gezeigt, dass mit dem *Fix Time Quantum* Benchmark Störungen im System sehr gut sichtbar gemacht werden können. Bei Durchführung weiterer Analysen können Frequenzen der Störungen gefunden werden. Per Frequenzanalyse mit der *Fourier-Transformation* kann man das Verhalten von Prozessen vergleichen. Der Nachteil ist, dass oft nur eine Frequenz dominant ist und die anderen überdeckt.

Bei der zweiten Variante, wo die Messdaten in horizontale Bänder unterteilt werden und dort jeweils die Frequenzen untersucht werden, erhält man sehr gute Ergebnisse, wenn es sich um eine Störung von nur einem Typ handelt.

Die zusätzlichen Informationen, die der *KTAU* Patch liefert, lassen sich gut mit den Messdaten des *FTQ* Benchmark kombinieren. Auf diese Weise können für die großen Störungen die Verursacher im System zugeordnet werden. Es fällt aber auch auf, dass nicht alle Unterbrechungen über die Scheduler-Schnittstellen geschehen sondern auf einer unteren Ebene abgearbeitet werden.

Der Nachteil des *KTAU* ist die erforderliche Anpassung des Linux Kernels, die – wie der *SLERT* Kernel gezeigt hat – nicht immer möglich ist.

6.3 MPI-Jitter Messungen

Nachdem der *FTQ* Benchmark gezeigt hat, dass auf dem *JuRoPA*-Testcluster lokaler *OS-Jitter* existiert, werden mit dem *MPI-Jitter* Benchmark die Auswirkung der Störungen auf parallele Programme untersucht.

Jeder Benchmarklauf besteht aus 20.000 Iterationen mit einer *Compute*-Phase von jeweils 100, 1.000, 10.000 bzw. 100.000 Operationen. Es werden jeweils 4 Prozessoren auf allen 16 Knoten genutzt. So nutzt *MPI-Jitter* jedesmal 64 Prozessen.

6.3.1 openSuSE 10.2 Kernel mit KTAU Patch

Im ersten Schritt wird das *JuRoPA*-Testcluster mit einem *openSuSE 10.2* Betriebssystem versehen, bei dem der Kernel um den *KTAU* Patch erweitert worden ist.

Es werden die Ergebnisse des 100-er und des 100.000-er Benchmarkslauf diskutiert. Für die anderen Messungen sind vergleichbare Abbildung im Anhang zu finden.

MPI-Jitter Messungen mit einem 100-er Workload

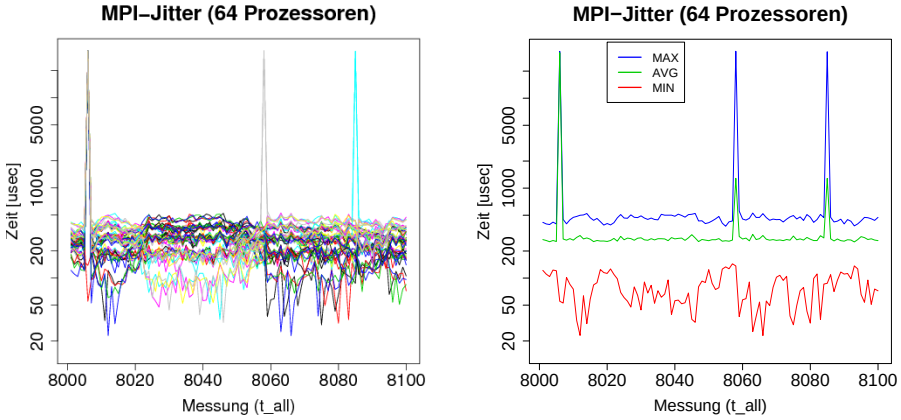


Abbildung 6.22: *MPI-Jitter* Messung mit einem *Workload* von 100

Ein Ausschnitt aus der Messung mit einer 100-er *Compute*-Phase ist in Abbildung 6.22 zu sehen. Um die Störungen und die nicht gestörten Bereiche gleichzeitig betrachten zu können, wurde die Zeit auf der Y-Achse logarithmisch aufgetragen. In der linken Graphik ist das $t\_all_{i,j} = t\_wait_{i,j} - t\_start_{i,j}$ dargestellt. Wider Erwarten erhält man keine einheitliche Linie, sondern es entsteht ein relativ großer Bereich und alle Zeiten scheinen unterschiedlich zu sein. Das rechte Bild zeigt das Minimum (ca. 78,4μs), den Durchschnitt (ca. 268μs) und das Maximum (ca. 458,2μs) dieser Linien. Dabei beträgt das absolute Minimum knappe 16μs und das absolute Maximum 87,6ms. Bei genauer Betrachtung ist in dieser Streuung eine Struktur zu erkennen.

In Abbildung 6.23 auf der nächsten Seite ist der Ausschnitt von der 8025. bis zur 8045. Messung vergrößert worden. Es scheint 5 horizontale Bereiche – im Folgenden als Bänder bezeichnet – zu geben, in denen sich die Prozesse befinden. Betrachtet man die vier Prozesse, die zu den Prozessoren eines Knoten gehören, genauer, so erkennt man,

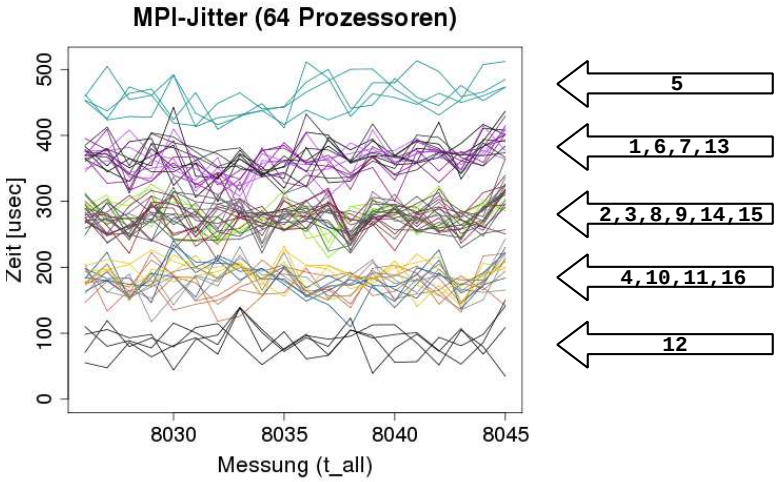


Abbildung 6.23: Bandstruktur der Messzeiten t_all (ohne Störung): Die Knoten gruppieren sich mit ihren Prozessoren in bestimmte Zeitbänder ein

dass diese vier Prozesse auch zum selben Band aus der Abbildung gehören. In dem Bild sind alle Prozesse auf demselben Knoten mit derselben Farbe versehen.

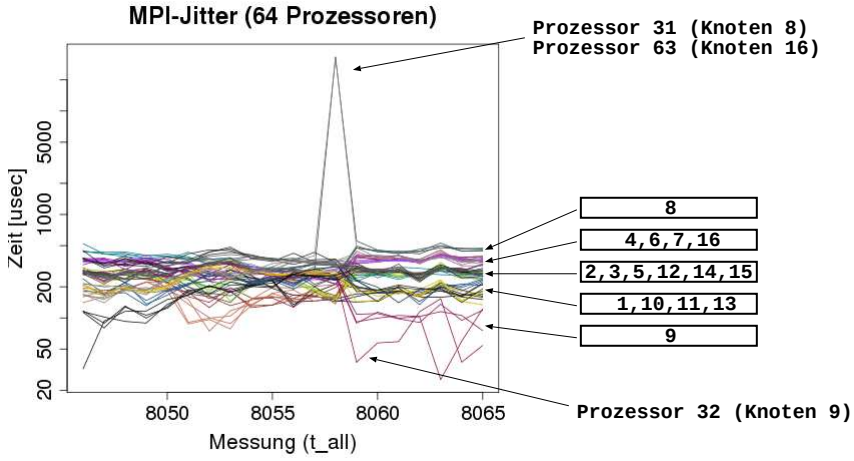


Abbildung 6.24: Bandstruktur der Messzeiten

Bis zur 8058. Messung ändert sich die Eingruppierung in die 5 Bänder nicht. In dieser Iteration tritt – wie in Abbildung 6.24 zu sehen ist – sowohl auf dem Prozessor 31 als auch auf dem Prozessor 63 eine Störung von ca. $33ms$ auf. Der zum Prozessor 63 gehörende $KTAU$ Log zeigt Folgendes:

```
...
45447352379586  <<< t_finished >>>
45447352381170  in   sys_poll
...   (alles KTAU Einträge)
45447354589644  out  sys_poll
45447354591696  in   __sched_text_start
45447453384147  in   ktau_trace_loss
45447453657405  <<< t_wait >>>
...
```

Die letzte Zeile zeigt den Endpunkt der 8058. Messung an – das Ende des *Barrier*. Die Zeile darüber (`in ktau_trace_loss`) ist eine Ausgabe des *KTAU* Protokolls und sagt aus, dass Beobachtungen verloren gegangen sind. Entscheidend ist die Meldung davor: `in __sched_text_start`. Diese zeigt, dass der Benchmark durch einen anderen Prozess unterbrochen worden ist. Zwischen diesen beiden Zeilen liegen fast genau $33ms$, also genau die Störung, die auch gemessen wird. Da die anderen parallelen Prozesse diese Störung nicht messen, war der *Barrier* hier schon in der zweiten Phase, so dass jeder Job weiterlaufen konnte.

Normalerweise wäre zu erwarten, dass die anderen Prozesse wesentlich schneller mit ihrer Arbeit fertig sind als die beiden gestörten Prozesse. Durch diesen Vorsprung sollten sie in eine Wartephase gelangen, die man im Benchmark sehen müsste. Dem ist aber nicht so. Das liegt an dem Aufbau des Benchmarks. Wie in Abbildung 4.7 auf Seite 34 beschrieben, wird zwischen jeder Messung eine zufällig Arbeit simuliert, bevor diese Prozesse wieder mittels eines *Barrier* synchronisiert werden. So eine Phase folgte der in diesem Fall gemessenen Störung, in der die anderen Prozesse auf die beiden warten konnten, ohne dass diese Wartezeit in $t\_all_{i,j}$ aufgetaucht ist.

Die hier aufgetretene Störung hat aber anscheinend Auswirkungen auf die Reihenfolge, wie die einzelnen Knoten in den *Barrier* gelaufen sind. In diesem Fall führt es zu einer neuen Aufteilung in den Bändern (siehe Abbildung 6.24 auf der vorherigen Seite). Entscheidend dafür ist, welcher Prozessor als erster fertig ist und den *Barrier* auslöst. In der 8059. Messung ist das der Prozess auf Prozessor 32, der sich auf dem 9. Knoten befindet. Wie zu sehen, bilden die Prozesse vom Knoten 9 das schnellste Band ab diesem Punkt. Welcher Knoten sich in welchem Band wiederfindet, entscheidet wahrscheinlich der Aufbau des verwendeten Netzwerks und Details der *MPI* Implementierung.

Eine andere Art der Störung findet man in Abbildung 6.25 auf der nächsten Seite. Dort handelt es sich um die 8006. Messung desselben Benchmarklaufs. In diesem Schritt benötigen fast alle Prozesse ca. $33ms$. Nur der Prozess auf Prozessor 25 braucht $138{,}2\mu s$ und der auf Prozessor 53 sogar nur $55{,}2\mu s$. Der *Barrier* sorgt dafür, dass sich alle Prozesse synchronisieren müssen. Schaut man sich die Messzeiten genau an, erkennt man bei dem Prozessor 25 und dem Prozessor 53 einen anderen Verlauf. Dort entsteht in der 8006. Messung die Wartezeit vor der Arbeit, während bei allen anderen die Zeit durch den anschließenden *Barrier* wieder ausgeglichen wird. Die Ursache liegt wahrscheinlich in einer Störung durch einen anderen Prozess, der den Prozessor 25 wie auch 53 genau in der *Barrier*-Freigabe vor der *Compute*-Phase trifft. Diese Vermutung wird durch den *KTAU* Log für den Prozessor 25 untermauert:

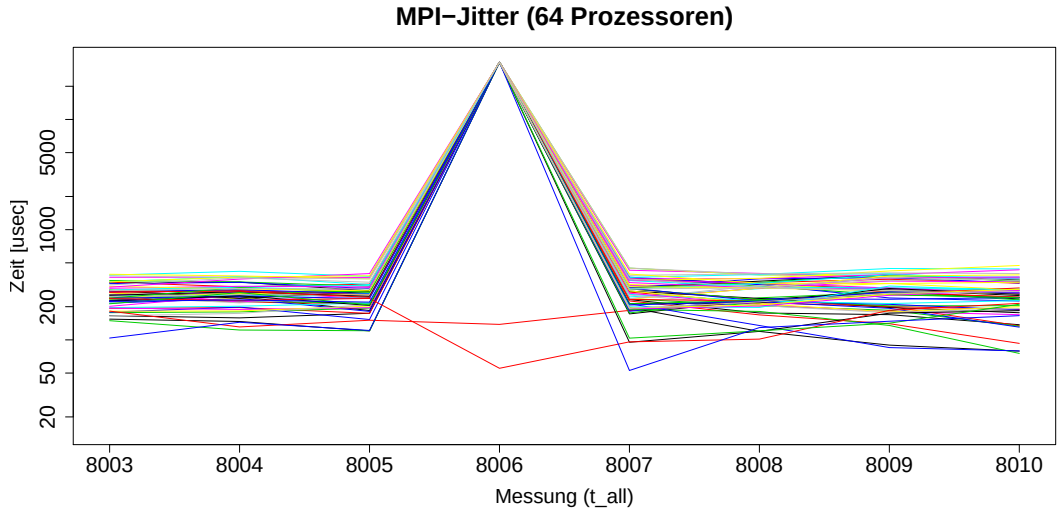


Abbildung 6.25: 100-er *Workload* Messung – Peak in der 8006. Messung für die Gesamtzeit (t_all_{8006})

```
...
249779662870419  <<< t_finished >>>
249779662875009  in   sys_poll
...   (alles KTAU Einträge)
249779663551836  out  sys_poll
249779663553618  in   __sched_text_start
249779762103978  in   ktau_trace_loss
249779762372484  <<< t_start >>>
...
```

Die letzte Zeile ist die Ausgabe des Benchmarks für den Start der 8006. *Compute*-Phase. Kurz vorher hat das *KTAU* zwar wieder die Datenerfassung für diesen Prozess unterbrochen (`ktau_trace_loss`), aber das passierte fast unmittelbar vor der nächsten Messung. Direkt davor wurde noch das Ereignis `__sched_text_start` erfasst, was auf eine Unterbrechung durch einen anderen Prozess hindeutet. Der *Barrier* war zu dieser Zeit schon fast komplett abgearbeitet. So konnten die anderen Prozesse schon rechnen, während Prozessor 25 noch ca. $33ms$ warten musste. Diese Wartezeit wurde durch den nächsten *Barrier* bei den anderen auch gemessen. Wahrscheinlich ist die lange Unterbrechung die Ursache für den `ktau_trace_loss` beim Benchmark. Die Störung wird den *Ring-Buffer* des *KTAU* füllen und dabei Daten für Ereignisse des *MPI-Jitter* Prozesses überschreiben.

Als Ergebnis kann man festhalten, dass der Knoten mit dem schnellste Prozess den Aufbau der Bandstruktur bestimmt. In Abbildung 6.26 auf der nächsten Seite sind zum einen alle Prozesse eines Knoten zusammengefasst, und dann nur die Messungen betrachtet worden, in denen der erste Knoten am schnellsten fertig war. Man sieht, dass die Einteilung der anderen Knoten in die Bänder immer gleich zu bleiben scheint. Das gilt für jeden Knoten. Unterteilt man die Messdaten in 16 Teile, in denen jeweils nur der i-te Knoten der schnellste ist, kann für die anderen Knoten mittels dieser Teilmenge

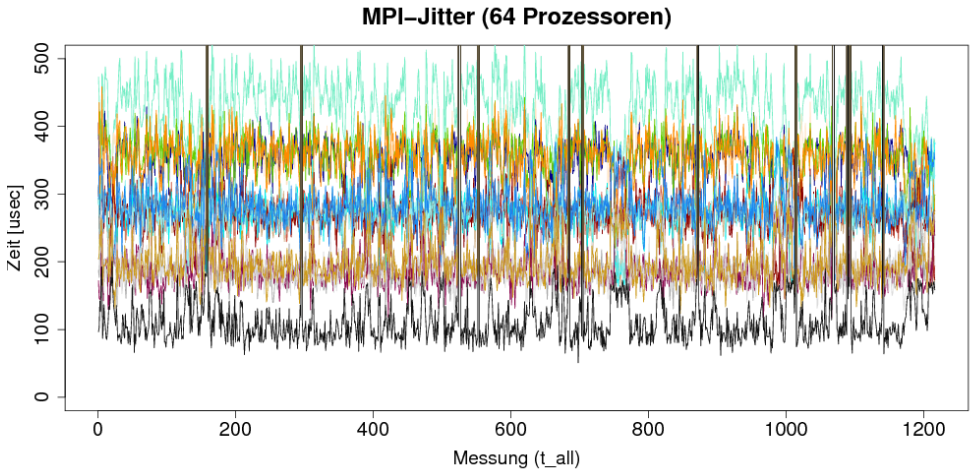


Abbildung 6.26: Messungen, in denen der 1. Knoten am schnellsten ist

eine Rangfolge erstellt werden. Das Ergebnis ist in Tabelle B.1 im Anhang auf Seite 112 zu sehen. Wenn man dort die Rangfolgen den 5 Bändern zuordnet, ist die Matrix sowohl zur Hauptdiagionalen als auch zur Nebendiagonalen symmetrisch.

Für die Messung des *OS-Jitter* bedeutet das, es sind nur die Ausreißer weit oberhalb von $600\mu s$ interessant. In Abbildung 6.27 wurde auf die $t\_all_{i,j}$ der Messung mit der 100-er *Compute*-Phase die Dichteschätzung angewendet. Die Daten sind in zwei Gruppen unterteilt worden. In der ersten sind alle Messungen ohne Störung ($t\_all_{i,j} < 600\mu s$, linkes Bild) und in der zweiten die restlichen ($t\_all_{i,j} > 600\mu s$, rechtes Bild) enthalten.

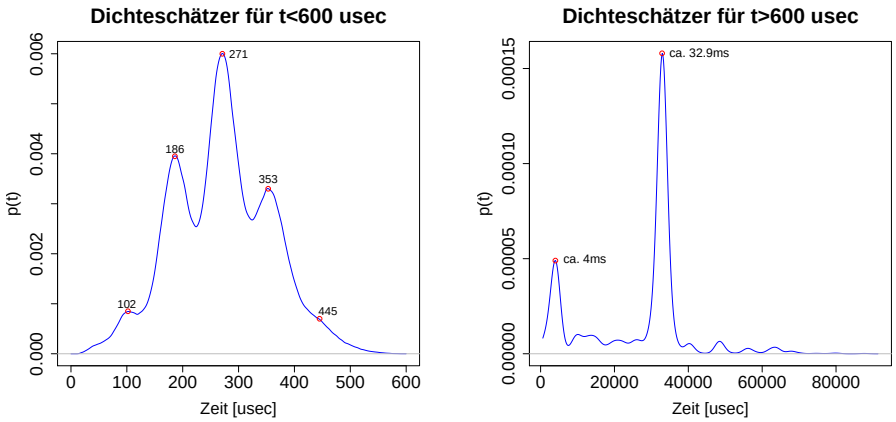


Abbildung 6.27: Dichteschätzer der Messdaten t_all

Im linken Bild sind wieder die 5 Bänder zu erkennen, die auch in Abbildung 6.23 auf Seite 67 zu sehen sind. Laut dem Dichteschätzer befinden sich die Bänder jeweils bei $102\mu s$, $186\mu s$, $271\mu s$, $354\mu s$ und $445\mu s$.

Der Dichteschätzer im rechten Bild zeigt zwei Spitzen. Die eine liegt bei $3{,}9ms$. und die andere bei $32{,}9ms$. Ein Beispiel für die Störung größerer Art ist auch in Abbildung

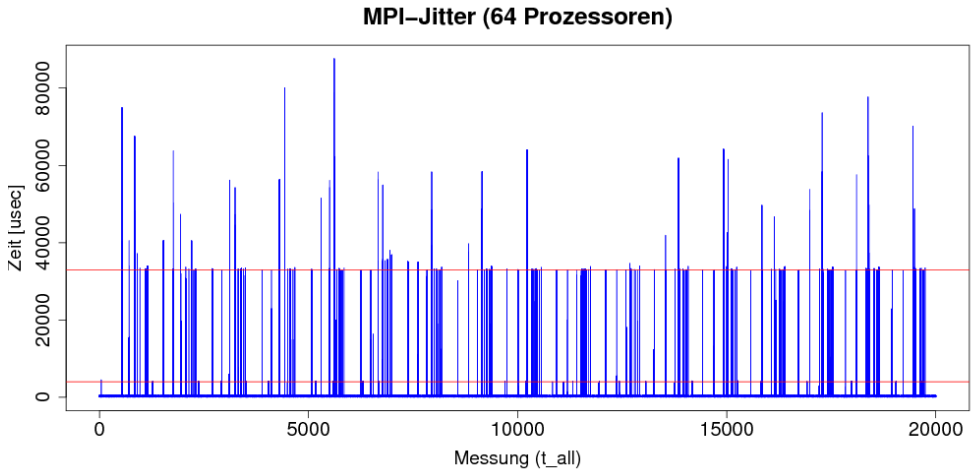


Abbildung 6.28: Markante Störungen in die 100-er Messung (t_all)

6.24 auf Seite 67 zu sehen.

Diese Störungszeiten findet man auch in Abbildung 6.28 wieder, wo die t_all für die gesamte Messung aller 64 Prozesse aufgetragen sind. Dieses Mal ist die Zeit auf der Y-Achse linear aufgetragen, weil nur die Störungen interessant sind. Es sind zwei horizontale Linien zu erkennen, eine bei ca. $4ms$ und bei ca. $33ms$.

Betrachtet man für diese Messung die *Compute*-Phase aller Jobs, dann erhält man eine interessante Dichteverteilung für die Laufzeiten < 7500 Taktzyklen (siehe Abbildung 6.29).

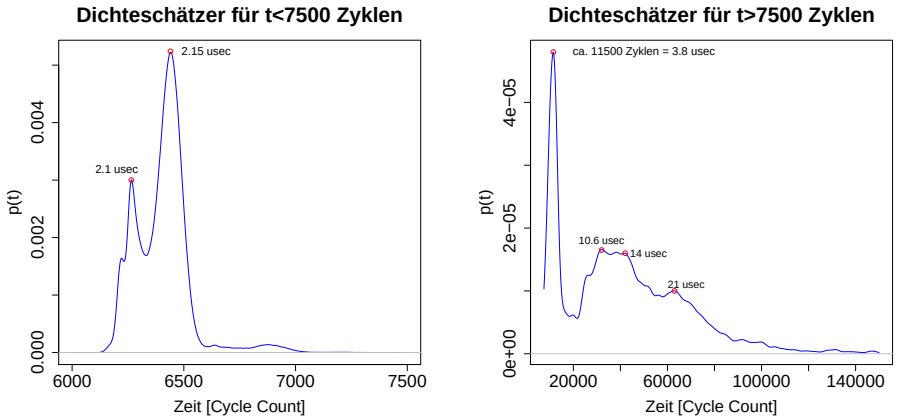


Abbildung 6.29: Verteilung der Laufzeiten für die 100-er *Compute*-Phase

Im linken Bild sind zwei Peaks zu erkennen, einer bei $6265\mu s$ und der andere bei $6440\mu s$. Die Standardabweichung für die Messdaten < 7500 Zyklen beträgt $126\mu s$, damit sind die beiden Häufungspunkte signifikant voneinander entfernt.

Obwohl es zu einer Verzögerung kommt, ist in diesem Bereich laut *KTAU* keine Störung aufgetreten. Ursache dafür wird wahrscheinlich derselbe Effekt sein, der schon

bei den *FTQ* Messungen zu sehen war (vergleiche dazu Abbildung 6.9 auf Seite 54). Auch dort brach die maximale Leistung immer wieder von 3590 auf 3235 ein, ohne dass dafür im *KTAU* LOG eine klare Ursache zu finden ist.

Erst Messungen von *Compute*-Zeiten über 10.000 Taktzyklen beruhen auf einem Ereignis durch einen fremden Prozess wie in diesem Beispiel:

```
...
239956932530160  <<< t_start >>>
239956932536901  in   __do_softirq
239956932538341  out  __do_softirq
239956932542058  <<< t_finished >>>
...
```

Das rechte Bild in Abbildung 6.29 zeigt typische Störungen, die auch vom *KTAU* erfasst worden sind. Es sind markante Zeiten zu erkennen, und zwar enden viele Messungen nach 3,8μs (das entspricht ca. 11500 Taktzyklen), einige gehen bis zu 10,6μs oder weiter. Es handelt sich um Hardware- (`softirq`) oder Netzwerkereignisse. Im gesamten Benchmarklauf treten nur zwei größere Störung während der *Compute*-Phase auf, die durch einen anderen Prozess ausgelöst werden. In beiden Fällen entsteht eine Wartezeit von fast 33ms.

MPI-Jitter Messungen mit einem 100.000-er Workload

Im nächsten Schritt werden die Messungen untersucht, bei denen die *Compute*-Phase 100.000 Iterationen beträgt. Zuerst werden die t_all betrachtet. In Abbildung 6.30 ist ein Ausschnitt der Messungen dargestellt, wobei die Y-Achse wieder logarithmisch aufgetragen worden ist.

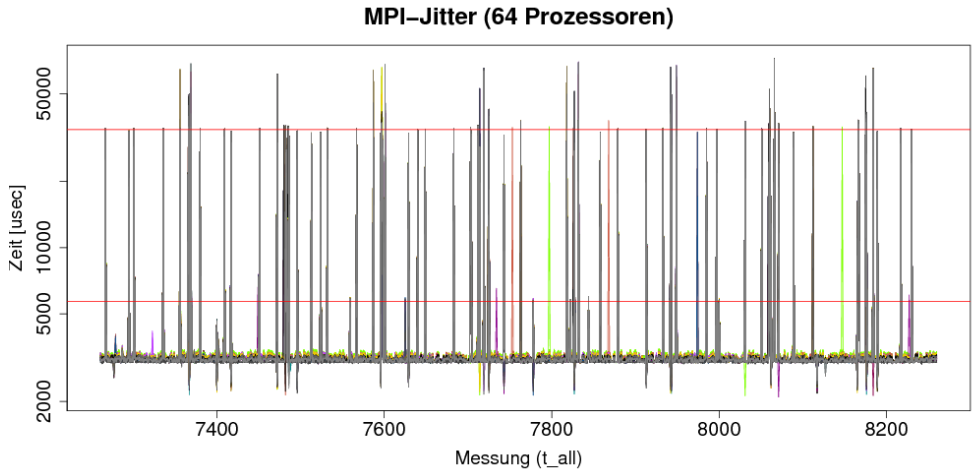


Abbildung 6.30: *MPI-Jitter* Messung: t_all-Zeiten für die 100.000-er *Compute*-Phase

Zum einen befindet sich der Großteil der Messwerte bei ca. 3100μs. Die Knoten scheinen sich wieder in Gruppen einzusortieren. Abbildung 6.31 auf der nächsten Seite zeigt alle Messungen, in denen $t\_all < 3500\mu s$ ist. Vergleicht man diese mit den Werten aus der 100-er Messung, wie sie in Abbildung 6.23 auf Seite 67 dargestellt sind, so haben sie

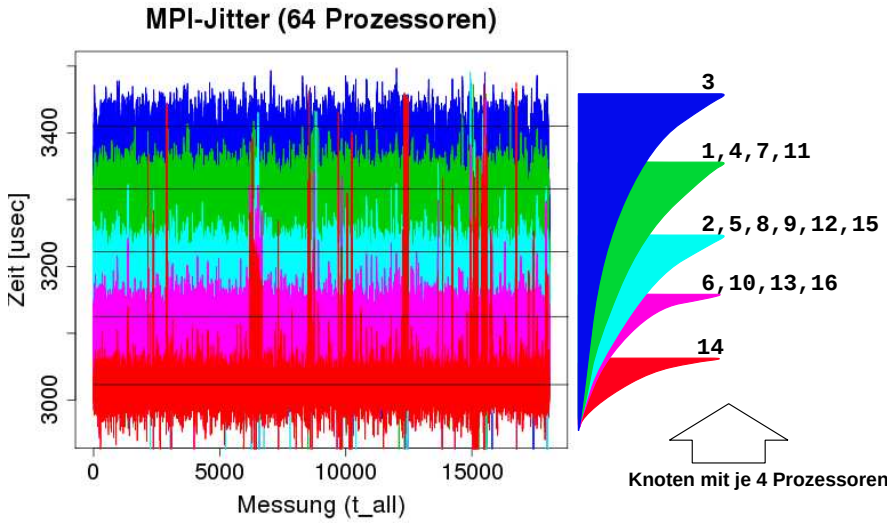


Abbildung 6.31: Messung der t_all-Zeiten für die 100.000-er *Compute*-Phase, bei denen keine Störung aufgetreten ist. Knoten 14 ist immer der schnellste

gemeinsam, dass die Prozesse auf demselben Knoten immer sehr ähnliche Messzeiten besitzen. Außerdem gibt es einen Knoten – in diesem Fall den Knoten 14, der im Mittel $3000\mu s$ benötigt. Das Gegenstück ist der Knoten 3, der im Mittel etwa $3200\mu s$ braucht. Die anderen gruppieren sich in die Bereiche ein, wie es im Bild von Abbildung 6.31 zu sehen ist. Anders als bei der 100-er Messung ist hier, dass über die gesamte Messung der Knoten 14 der Schnellste ist, selbst wenn zwischendurch der Prozess eines anderen schneller fertig werden sollte. Die Ursache bzw. eine mögliche Begründung wird später untersucht. Ein weiterer Unterschied ist, dass die Gruppen nicht mehr in separate Bänder unterteilt sind, sondern dass sie sich überlappen. Alle Prozesse erreichen die gleiche Mindestlaufzeit, haben aber einen anderen Mittelwert und infolge der Streuung eine größere Varianz bzw. Standardabweichung. So haben die Messwerte für t_all auf dem Knoten 14 eine Standardabweichung von $63{,}05\mu s$, während diese auf dem Knoten 3 $135{,}9\mu s$ beträgt.

Dieses Ergebnis spiegelt sich ebenfalls in den Dichteschätzungen aus Abbildung 6.32 auf der nächsten Seite wieder. Im linken Bild werden alle Messungen kleiner $3.500\mu s$ betrachtet, also die ohne Störung. Es entstehen 4-5 Peaks. Deren Werte entsprechen jeweils den Mittelwerten der 5 Bänder.

Das rechte Bild in Abbildung 6.32 zeigt die Verteilung der Störungen. Es sind zwei markante Stellen zu erkennen, eine bei $5.700\mu s$ und eine wesentlich stärkere bei $34.400\mu s$. Diese Werte sind als horizontale Linie in dem Ausschnitt der Messwerte in Abbildung 6.30 auf der vorherigen Seite eingetragen.

Aufschluss über die Ursache, dass immer der Knoten 14 am schnellsten ist, geben die Messdaten für $t\_compute$. In Abbildung 6.33 auf der nächsten Seite ist die komplette Messung aufgetragen. Auffallend sind zwei Messlinien (rot und blau), die nicht auf der gleichen Höhe liegen wie die anderen. Der entsprechende Dichteschätzer aus Abbildung 6.34 auf Seite 75 verdeutlicht das. Für $t\_compute < 3500\mu s$ ist der Mittelwert auf

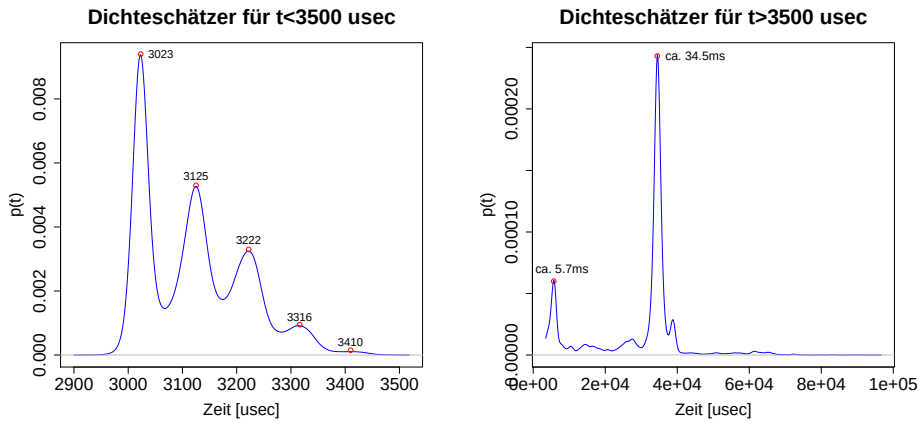


Abbildung 6.32: Verteilung der Messzeiten. Links: ohne Störung, rechts die Störungen

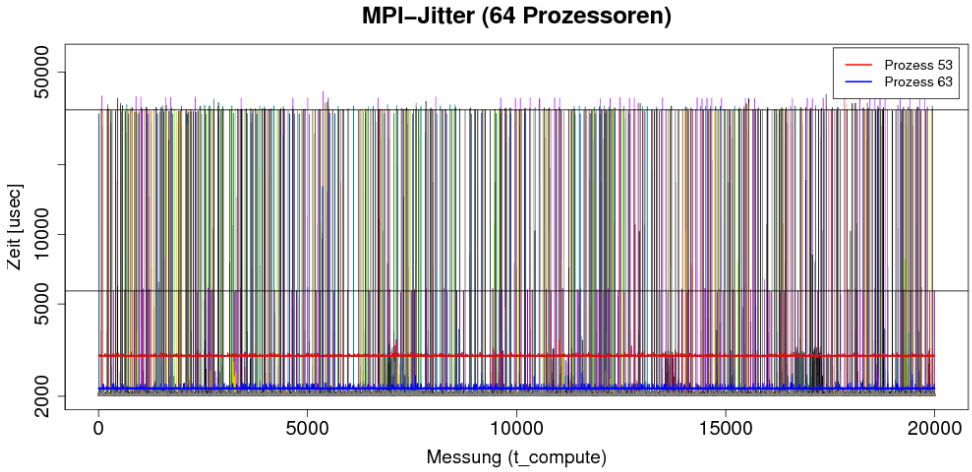


Abbildung 6.33: Messungen der *Compute*-Phase für den 100.000er Benchmark-Lauf

fast allen Prozessoren kleiner als 2050μs. Nur für Prozessor 63 (blau) beträgt er 2165μs und für Prozessor 53 (rot) sogar 2985μs.

Es scheint irgendetwas zu passieren, das diese beiden Prozesse ausbremst, und zwar ziemlich konstant. Das *KTAU* Protokoll gibt hierzu zumindest einen Anhaltspunkt:

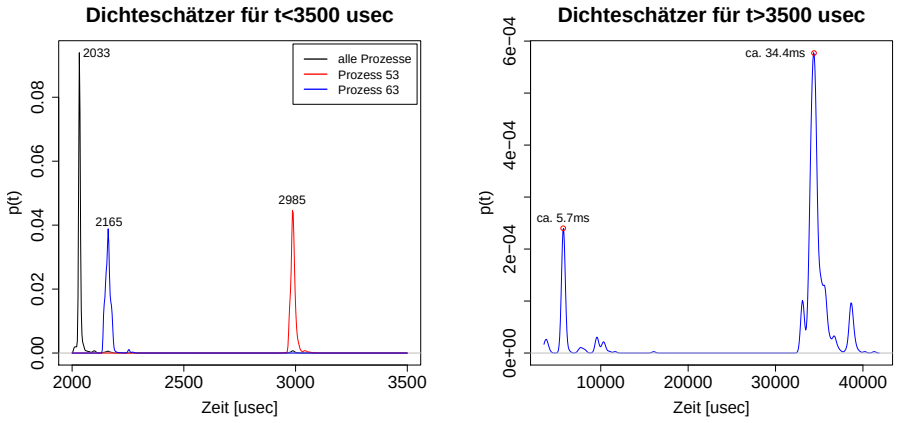


Abbildung 6.34: Verteilung der Messzeiten für die *Compute*-Phase. Links: ohne Störung, rechts die Störungen

Prozessor 53

```
...
379889723  <<< t_start >>>
384828671  in   timer_interrupt
384832856  out  timer_interrupt
385484600  in   smp_apic_timer_interrupt
385485680  in   __do_softirq
385486409  in   __run_timers
385487057  out  __run_timers
385487300  out  __do_softirq
385487516  out  smp_apic_timer_interrupt
386720174  in   __do_softirq
386731883  in   tcp_v4_rcv
386733125  in   tcp_v4_do_rcv
386733359  in   tcp_rcv_established
386738732  out  tcp_rcv_established
386738957  out  tcp_v4_do_rcv
386739353  out  tcp_v4_rcv
386741297  out  __do_softirq
388824608  <<< t_finished >>>
388831736  in   sys_sendmsg
...
```

Prozessor 54

```
...
429503438  <<< t_start >>>
433370144  in   smp_apic_timer_interrupt
433371377  in   __do_softirq
433372088  in   __run_timers
433372772  out  __run_timers
433375112  out  __do_softirq
433375328  out  smp_apic_timer_interrupt
435574568  <<< t_finished >>>
435577097  in   sys_poll
...
```

In beiden Ausschnitten sind die Messungen aus dem *MPI-Jitter* Benchmark mit dem *KTAU* Protokoll zusammengefügt worden. Es zeigt sowohl für den Prozessor 53 als auch für den Prozessor 54 einen für diesen Job typischen Verlauf der *Compute*-Phase. Der gravierende Unterschied ist, das auf dem Prozessor 53 immer auch ein Netzwerk-Ereignis erfasst wird, eingebettet durch ein `__do_softirq`. Dessen Abarbeitung benötigt in diesem Beispiel nur 21.123 Zyklen, was nicht die ca. 2,8 Millionen Zyklen erklärt, die von dem Prozessor 53 mehr benötigt werden.

Es führt aber dazu, dass immer derselbe Prozess als Letzter fertig wird, und damit auch als letzter in den *Barrier* läuft. Der Prozessor 53 gehört zum Knoten 14, und genau der hat im Mittel die niedrigste Zeit für t_all. Schon für den Benchmark Lauf mit der 100-er *Compute*-Phase haben die Messwerte gezeigt (Tabelle B.1 auf Seite 112), dass der Knoten 3 das Gegenstück zum Knoten 14 ist. Man findet hier dieselbe Struktur wieder.

Es scheint nur so, dass der langsamste Knoten den *Barrier* zuerst wieder verlässt und damit der Schnellste zu sein scheint.

Nun werden die größeren Störungen betrachtet, die im Dichteschätzer für die *Compute*-Phase mit Zeiten größer $3500\mu s$ zu sehen sind (rechtes Bild in Abbildung 6.34 auf der vorherigen Seite). Es sind zwei markante Bereiche zu erkennen, zum einen um die $5{,}7ms$ und zum anderen um die $34{,}4ms$.

Prozessor ⇓	Knoten 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	−	±					−				+			−		
2			−			+		−	±			+	+	+		
3			+	+	+			+		+	−			−	+	
4	+		−		−	−	+					−	−			±

Tabelle 6.1: Aufteilung der Störungen auf die Knoten: „+“ für die Störung von $34{,}4ms$, „-“ für die Störung von $5{,}7ms$

Die Tabelle 6.1 zeigt, wie sich die beiden Arten von Störungen auf dem gesamten System verteilen. Der Prozess mit den $34{,}4ms$ taucht auf jedem Knoten einmal auf. Die Störungen mit $5{,}7ms$ Dauer ist häufiger zu sehen, aber nicht auf jedem Knoten. So zeigt keine einzige Messung auf Knoten 15 diese Art der Störung, während auf anderen Knoten zum Teil mehrere zu sehen sind. Schaut man sich die Verteilungen der $5{,}7ms$ Störungen an (linkes Bild in Abbildung 6.35), unterteilen sich die Unterbrechungen in Typen von $4ms$, $6ms$, $8ms$ und $10ms$. Es sind oft 2-3 Peaks für denselben Prozess zu sehen. Ob es sich aber um verschiedene Ursachen handelt oder um dieselbe, ist diesen Daten nicht zu entnehmen.

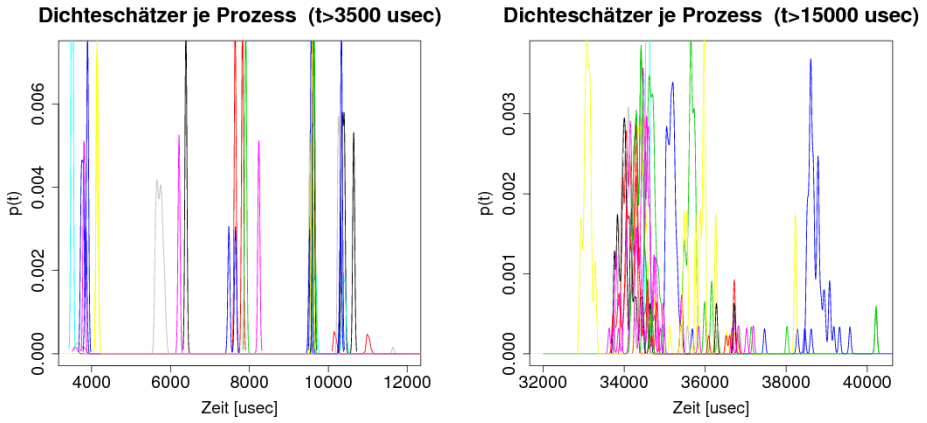


Abbildung 6.35: Verteilung der Störungen: Links um $5{,}7ms$, rechts um $34{,}4ms$

Im rechten Bild von Abbildung 6.35 ist der Dichteschätzer für die $34{,}4ms$ Störung zu sehen. Die Messungen der einzelnen Prozesse streuen relativ gleichmäßig um diese Zeit. Das ist ein Indiz dafür, dass es sich bei allen um die gleiche Art der Störung handelt. Bei einigen gibt es noch einen Peak zwischen $38ms$ und $40ms$. Das ist wahrscheinlich eine Kombination aus der $5{,}7ms$ Störungsklasse und der $34{,}4ms$ Störung.

6.3.2 SuSE Linux Enterprise Real Time 10 System (SLERT)

Ähnlich wie bei der *FTQ* Messreihe wird der *MPI-Jitter* Benchmark auch auf dem SLES System mit dem Echtzeit Patch durchgeführt.

Es werden wieder nur die 100-er und die 100.000-er Messungen genauer betrachtet. Für die anderen sind vergleichbare Abbildung im Anhang zu finden.

MPI-Jitter Messungen mit einem 100-er Workload

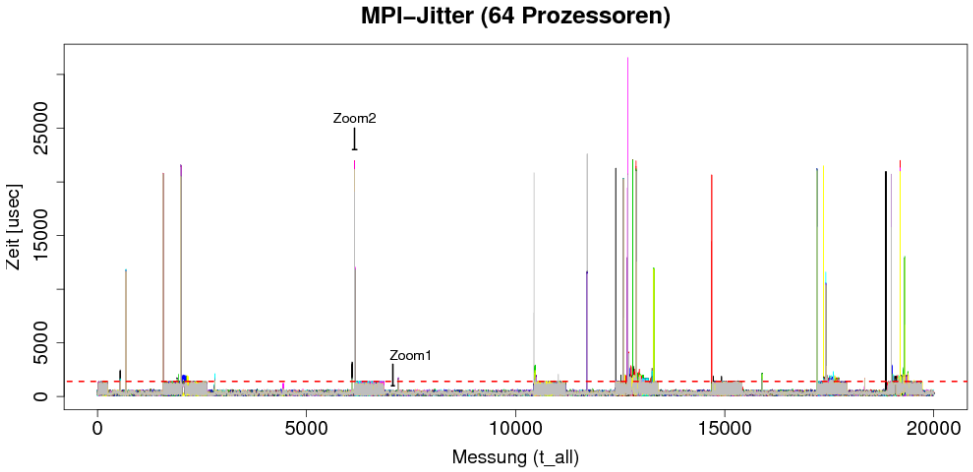


Abbildung 6.36: Messungen des 100-er *MPI-Jitter* Benchmarks (t_all).

In Abbildung 6.36 ist die Gesamtzeit t_all der 100-er Messung aufgetragen. Man sieht einen gravierenden Unterschied zu dem äquivalenten Lauf auf dem *openSuSE* System (Abbildung 6.30 auf Seite 72). Normalerweise beträgt die Laufzeit für t_all ca. $300\mu s$. Aber immer wieder kommt es zu längeren Störungen, die zu einer Messzeit von ca. $1400\mu s$ führen (gestrichelte Linie).

Der zweite Unterschied ist, dass zwar wieder eine Bandstruktur zu erkennen ist (siehe linkes Bild in Abbildung 6.37 auf der nächsten Seite), aber wesentlich häufiger die Reihenfolge gewechselt wird. Das liegt wahrscheinlich an dem *System Tick*, der bei dem *Real Time* Kernel auf eine Frequenz von $1000Hz$ eingestellt ist. Gleichgeblieben ist dafür die Eingruppierung in die Bänder, wie man es in der Tabelle B.2 auf Seite 113 im Anhang erkennen kann. Das war auch zu erwarten, denn die Kommunikationswege und die *MPI* Implementierung haben sich nicht geändert.

Der Dichteschätzer für $t\_all < 2000\mu s$ in Abbildung 6.37 auf der nächsten Seite bestätigen noch einmal die Zeiten. Im rechten Bild sind zwei Peaks zu erkennen, einer bei $300\mu s$ und ein zweiter bei $1170\mu s$. Es ist nicht die Bandstruktur zu erkennen, wie sie bei dem entsprechenden Dichteschätzer für Messungen auf dem *openSuSE*-System (Abbildung 6.27 auf Seite 70) zu sehen waren. Die Zeiten streuen nur in einem Bereich von ca. $100\mu s$ bis $500\mu s$.

Einen Grund für den zweiten Peak findet man, wenn man eine Störung genauer betrachtet. Im linken Bild in Abbildung 6.38 auf der nächsten Seite ist ein entsprechender

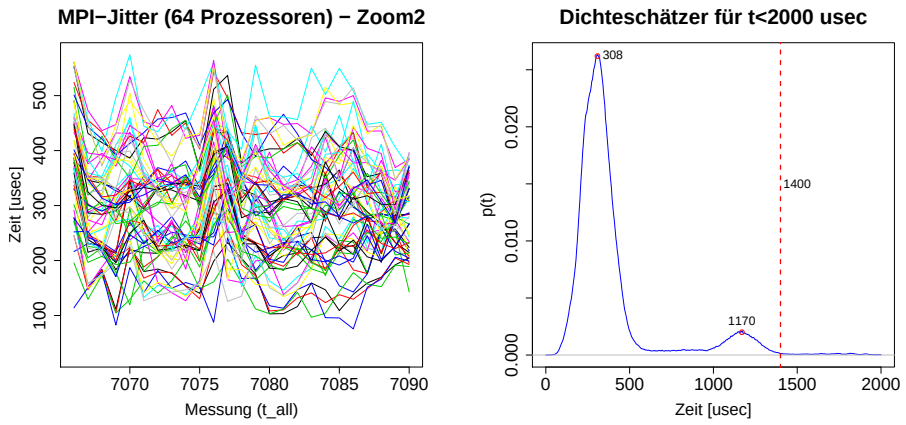


Abbildung 6.37: schwache Bandstruktur zu erkennen (linkes Bild). Das rechte Bild zeigt den Dichteschätzer für die Messungen ohne Störungen (t_all)

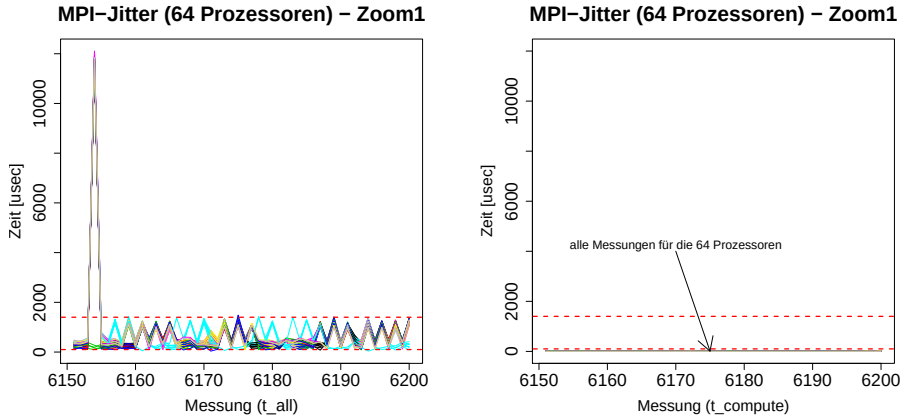


Abbildung 6.38: Aufbau einer Störung: links die Zeiten für t_all, rechts die Zeiten für $t\_compute$

Abschnitt vergrößert worden. Die Zeiten für t_all befinden sich anfangs auf dem niedrigen Niveau von $300\mu s$. Dann tritt eine Störung von $15ms$ auf, die von einer Phase gefolgt wird, in der die Zeiten zwischen $1400\mu s$ und $300\mu s$ variieren. Man sieht dasselbe Verhalten, wie es auch bei der *FTQ* Messung in Abbildung 6.17 auf Seite 61 zu beobachten ist. Es tritt eine Störung auf, der Scheduler wird aktiv, muss Resourcen anfordern und neu verteilen, und dann teilen sich die beiden Prozesse den Prozessor. Jeweils $1ms$ rechnet der Benchmark und $1ms$ der störende Prozess.

Die *Compute*-Phase selbst wird nicht gestört, wie das rechte Bild in Abbildung 6.38 zeigt. Das liegt daran, dass die im Mittel $2{,}3\mu s$ lange Berechnung im Verhältnis zu den t_all Zeiten sehr klein ist. Dementsprechend unwahrscheinlich ist es, dass eine Störung genau in der *Compute*-Phase auftritt.

Die Störungen selbst unterteilen sich in zwei Arten laut der Dichteschätzung, die in Abbildung 6.39 auf der nächsten Seite zu sehen ist. In der ersten Gruppe entstehen

Wartezeiten von ca. $11ms$, in der zweiten von ca. $21ms$.

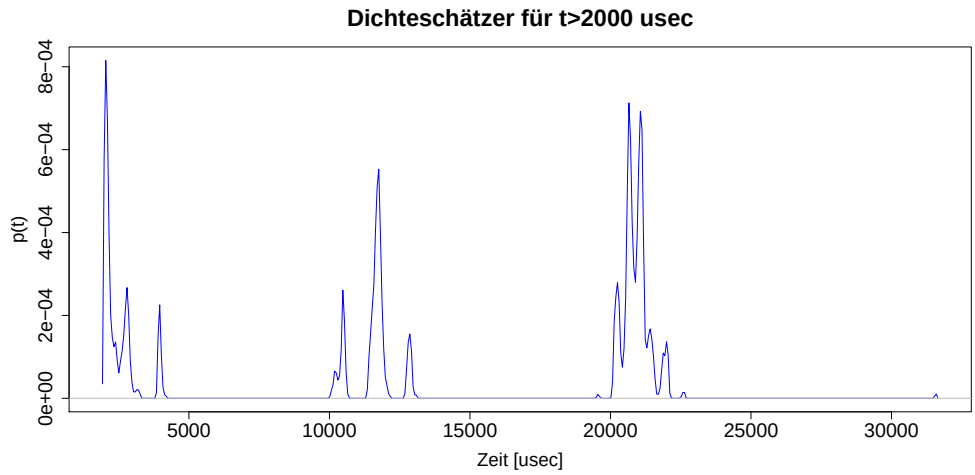


Abbildung 6.39: Dichteschätzer für die Störungen bei der 100-er *MPI-Jitter* Messung (t_all)

Vergleicht man die Ergebnisse der 100-er Messung mit denen der 1000-er Messung (siehe im Anhang Abbildung B.3 auf Seite 110), sieht man ein äquivalentes Verhalten. Es fällt nur auf, dass sich bei den 1000-er Messung Störungen auch häufiger auf die *Compute*-Phase auswirken.

An dieser Stelle ist wichtig zu erkennen, dass eine lokale Störung bei der Synchronisation durch blockierende Kommunikation alle Beteiligten ausbremst.

MPI-Jitter Messungen mit einem 100.000er Workload

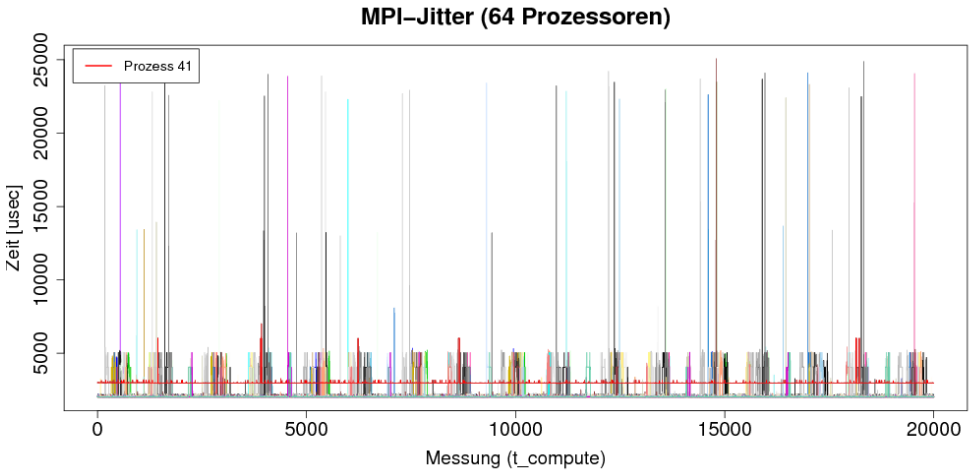


Abbildung 6.40: *Compute*-Phase der 100.000-er Messung

Als Letztes soll noch der 100.000-er Benchmark genauer betrachtet werden. Das Verhalten ist wieder sehr ähnlich den bisher gemessenen Zeiten. Die *Compute*-Phasen beanspruchen laut Dichteschätzer $2ms$ (siehe Abbildung B.4 auf Seite 111 im Anhang). In Abbildung 6.40 fällt auf, dass ein Prozess – hier der auf Prozessor 41 (rot) – mindesten $3ms$ benötigt. Dieser Effekt war auch auf dem *openSuSE*-System bei dem 100.000-er *MPI-Jitter* Benchmark zu sehen. Es entsteht auch derselbe Effekt, dass der Knoten 11, der den Prozessor 41 enthält, im Vergleich mit den anderen Knoten häufiger als erster den *Barrier* beendet.

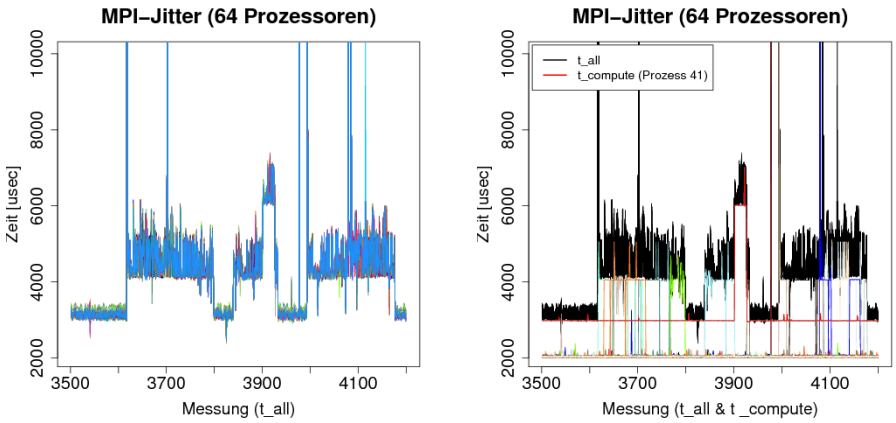


Abbildung 6.41: Vergrößerung einer Störung des 100.000-er Benchmarks: links nur t_all, rechts mit $t\_compute$

In Abbildung 6.41 ist links ein kleiner Ausschnitt von t_all dargestellt. Besonders auffallend ist die treppenartige Funktion der Messwerte. Die Sprünge treten in $1ms$

Schritten auf. Das entspricht genau einer Zeitscheibe bei dem *System Tick* von $1000Hz$. Das rechte Bild zeigt zusätzlich die Zeiten von $t\_compute$ aller Prozesse. Die rote Linie, die sich meistens bei $3000\mu s$ befindet, repräsentiert den Wert $t\_compute$ des Prozessors 41. Dieser langsame Prozess verursacht bei allen anderen Prozessen, eine Wartezeit von $1ms$. Störungen auf den anderen Prozessoren wirken sich erst dann aus, wenn sie mehr als eine Zeitscheibe benötigen und damit länger dauern als Prozessor 41.

Betrachtet man den Verlust, so beträgt er, wenn man sich nur auf $t\_compute$ bezieht, fast 72%. Aber allein schon durch den Prozess 41 entstehen 50% Anteil an dem Verlust. Berechnet man dasselbe für t_all, so verliert man ca. 6,5% allein im *Barrier*.

6.3.3 Fazit

Der *MPI-Jitter* Benchmark hat die Fähigkeit, einige interessante Verhaltensweisen des Systems aufzudecken. In diesen Tests wurden Bandstrukturen bei der Kommunikation festgestellt, deren Ursache in der Struktur des Netzwerkes bzw. in der *MPI* Implementierung zu suchen sind. Es wurden auch unterschiedliche Verhaltensweisen zwischen *openSuSE* und *SuSE Linux Enterprise Real Time* festgestellt.

Wie erwartet, haben sich lokale Störungen auf den gesamten Job ausgewirkt. Sie führen zu Wartezeiten bei allen anderen beteiligten Prozessen.

Bei den 100.000-er Messungen sind jeweils 1-2 Prozesse durch ihre ca. $1ms$ lange Verzögerung aufgefallen. Es scheint sich um ein Software Problem zu handeln, dessen Ursache letztendlich aber nicht geklärt werden konnte. Die zugehörigen $KTAU$ Protokolle zeigten zwar mehr Einträge auf den betroffenen Prozessoren, aber diese Informationen reichten nicht aus.

Insgesamt muss festgestellt werden, dass das $KTAU$ in den entscheidenden Situationen bei den Unterbrechungen leider häufig die Protokollierung unterbrochen hat. Vielleicht würde eine optimierte Konfiguration des $KTAU$ im Kernel (z.B. eine Vergrößerung des *Ring Buffer*) zu weniger Verlusten führen.

Ein Problem kann noch die Datenmenge verursachen. Je mehr beteiligte Prozessoren, desto mehr Daten produziert der *MPI-Jitter* Benchmark. Pro Iteration werden auf jedem Prozessor drei Zeiten gemessen. Nimmt man an, dass pro Zeit 20 Zeichen ausgegeben werden, ergibt das 60 Bytes. Stellt man den Benchmark auf 20.000 Messungen ein, wird je Prozessor eine Datei der Größe $1{,}2MB$ angelegt. Wenn jetzt z.B. eine Messung auf 1024 Prozessoren durchgeführt wird, ergibt das in der Summe eine Datenmenge über $1{,}2GB$.

Kapitel 7

Konzipierung und Implementierung eines *OS-Jitter* Werkzeugs

Im letzten Kapitel wurde gezeigt, wie man den *OS-Jitter* in einem System sichtbar machen kann.

Im Folgenden wird ein Konzept vorgestellt, mit dem man den *OS-Jitter* auf einem beliebigen System untersuchen kann. Dafür wird ein Paket aus drei Komponenten benötigt:

FTQ: Der Benchmark zeigt die Störungen auf einem Prozessor unabhängig von den anderen Prozessoren.

MPI-Jitter: Der Benchmark zeigt, wie sich lokale Störungen auf die globale, blockierende Kommunikation auswirken.

Statistik Software *R*: Ein mathematisches Software Paket, mit dem die Daten aus den Benchmarks ausgewertet werden können. Die Auswertung von den Messdaten der beiden Benchmarks wird mithilfe dieser Software durchgeführt.[15]

Im Anschluss wird deren Anwendung auf den beiden derzeit verfügbaren *Supercomputern* des Forschungszentrum Jülich demonstriert, dem *JuMP* und der *JuGene*.

7.1 FTQ-Benchmark

Es wird empfohlen, den *Fix Time Quantum* Benchmark leicht zu verändern. Die Messungen auf verschiedenen Prozessoren sollen gleichzeitig durchgeführt werden, damit z.B. wandernde Prozesse sichtbar werden. Das Programm wird als normaler Job mit dem vorhandenen Batch-System gestartet.

Source `ftq.c` anpassen:

Vorspann (Zeile 5) Einfügen der MPI Header-Datei

```
#include <mpi.h>
```

main (Zeile 84) Variablen definieren

```
int rank,size;
```

main (Zeile 98) *MPI* starten, Zahl der Prozesse und eigene Nummer bestimmen

```
MPI_Init(&argc, &argv);
```

```
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

main (Zeile 99) Ausgabe-Dateinamen erweitern

```
sprintf(fname_times,"ftq_times-%d.dat",rank);
sprintf(fname_counts,"ftq_counts-%d.dat",rank);
```

main (Zeile 128) (optionale) Ausgabe-Dateinamen erweitern

```
sprintf(fname_times,"%s_times-%d.dat",optarg,rank);
sprintf(fname_counts,"%s_counts-%d.dat",optarg,rank);
```

main (Zeile 242) MPI beenden

```
MPI_Finalize();
```

Nach diesen Änderungen kann das Programm mithilfe des mitgelieferten `Makefile` übersetzt werden.

Per Batch-System kann das Programm auf dem Cluster System unter Verwendung beliebig vieler Prozessoren gestartet werden. Auf jedem beteiligten Prozessor wird ein eigener `ftq` gestartet, der jeweils eine Ausgabedateien `ftq_times-#.dat` für die Zeiten und `ftq_counts-#.dat` für den Workload erzeugt.

7.1.1 Auswertung der Daten mit *R*

Ziel ist es, Frequenzen in den *FTQ* Benchmark Messungen zu finden. Dazu müssen die Zeiten und der *Workload* eingelesen werden.

```
COUNTS=read.table("ftq_counts-42.dat")[,1]
TIMES=read.table("ftq_times-42.dat")[,1]
```

Für die Zeit gilt, dass sie in Prozesszyklen gemessen worden ist. Mithilfe der CPU Frequenz kann man diese in Sekunden umrechnen.

```
FREQ=2997
TIMES=TIMES/(FREQ * 10^6)
plot(TIMES,COUNTS,type="p")
```

Anstelle der `COUNTS` kann man für die Y-Koordinaten auch das ΔTIMES bestimmen. Es macht das automatische Justieren der Messintervalle durch den *FTQ* Algorithmus sichtbar. Treten nur kleine Störungen auf, werden die Zeitintervalle meistens eingehalten und müssen nicht angepasst werden. Bei größeren Unterbrechungen können je nach Größe der Messintervalle viele übersprungen werden. Das führt zu markanten Ausreißern bei den Zeitabständen.

```
DELTATIMES=vector(mode="numeric",length=length(TIMES)-1)
dt=2^24 # Intervallgroesse: Normierung auf 1
for(i in 1:(length(TIMES)-1)) {
  DELTATIMES[i]=(TIMES[i+1]-TIMES[i])/dt
}
plot(TIMES,DELTATIMES,type="p")
```

Geglättete *Fourier-Transformation*

Für die korrekte Darstellung der *FFT* Daten muss ein Vektor mit den passenden Frequenzen erzeugt werden. Dafür wird aus den Messdaten das Δt für die Zeitintervalle bestimmt und darüber werden die Frequenzen ausgerechnet. Alternativ kann man das Δt auch direkt angeben, wie man es beim Aufruf des Benchmark definiert hat (z.B. 2^{22}).

```
xdiff=vector(mode = "numeric", length = length(TIMES)-1)
for(i in 1:(length(TIMES)-1)) {
    xdiff[i]=TIMES[i+1]-TIMES[i]
}
dt=median(xdiff) # Delta t
Fs=1/dt          # Abtast-Rate
f=seq(Fs/(2*n),Fs/2, by=Fs/(2*n))
```

Um die geglättete *Fourier-Transformation* durchzuführen, kann ein vordefinierter Kern, wie es im Kapitel über die mathematischen Hilfsmittel auf Seite 37 beschrieben ist, benutzt werden. Hier wird ein Beispiel für einen selbst definierten Kern (k) vorgestellt.

```
k=kernel(coef=c(11,10,9,8,7,6,5,4,3,2,1)/121,m=10)
result=spec.pgram(COUNTS,kernel=k,plot=F)
plot(f,result[["spec"]],type="l",log="x")
```

Frequenzen bestimmen mit Dichteschätzer

Eine Störung ist in der *FTQ* Messung dadurch charakterisiert, dass sie immer zu demselben Leistungseinbruch führt (Klasse einer Störung). Deswegen ist eine weitere Möglichkeit, alle Messdaten zu extrahieren, die einer Störungsklasse angehören. Dazu legt man um den Leistungswert der Störung einen festen Bereich. Dann kopiert man alle Daten. Die Werte aus dem gewählten Bereich bleiben erhalten, alle anderen werden auf „0“ gesetzt (`cutoff`-Funktion). Man erhält ein Ergebnis, das man auf enthaltene Frequenzen untersuchen kann.

```
cutoff=function(dat=y,ymin=0,ymax=10^10){
  z=length(dat)
  dat.r=rep(0,z)
  for (i in 1:z){
    if ( (dat[i] >= ymin) && (dat[i] <= ymax) ){
        dat.r[i]=dat[i]
    } else {
        dat.r[i]=0
    }
  }
  return(dat.r)
}
BCOUNT=cutoff(COUNT,3000,34000)
HTIME=TIME[BCOUNT>0]
```

Bestimmt man den Zeitabstand Δt zwischen den Störungen, kann man darauf einen Dichteschätzer anwenden.

```
deltaT=vector(mode="numeric",length=length(HTIME)-1)
for(i in 1:(length(HTIME)-1)) {
  deltaT[i]=HTIME[i+1]-HTIME[i]
}
plot(density(deltaT))
```

7.2 MPI-Jitter Benchmark

Das *MPI-Jitter* ist ein im Rahmen dieser Arbeit entwickelter Benchmark. Es misst die Auswirkung von Störungen bei der globale Kommunikation, die auf den einzelnen Prozessoren auftreten. Es handelt sich um ein *MPI*-Programm, das per Batch-System auf dem Cluster verteilt wird. Auf jedem Prozessor wird eine Ausgabedatei `mpi-jitter.#1.#2.#3.out` erzeugt. Die `#1` steht für die Anzahl der Messungen, `#2` wird durch die Gesamtzahl der verwendeten Prozessoren ersetzt, und `#3` ist die Nummer des Prozessors, von dem diese Datei erzeugt wird.

Die Datei besteht aus drei Spalten:

t_start: Zeitpunkt vor der *Compute*-Phase

$t\_finished$: Zeitpunkt nach der *Compute*-Phase

t_wait: Zeitpunkt nach der blockierenden Kommunikation (*Barrier*)

Die Zeiten sind die Werte, die aus dem *Time Stamp Counter* Register des Prozessors ausgelesen worden sind. Sollte auf dem System die Funktion `rdtsc()` nicht anwendbar sein, weil die Prozessoren dieses Register nicht besitzen, dann kann beim Compilieren die Option `-D GTOD` verwendet werden. In dem Fall wird statt des *Time Stamp Counters* die Funktion `gettimeofday()` verwendet, und die Zeiten in Mikrosekunden (μs) ausgegeben.

7.2.1 Auswertung der Daten mit *R*

Die Messdaten liegen in einer dreispaltigen Tabelle vor. Um die Zeiten für die *Compute*-Phase einzulesen, kann z.B. folgende Funktion verwendet werden:

```
readdata_compute=function(knots=32,workload=100000){
  daten=matrix(0,ncol=knots,nrow=20000)
  for(i in 0:(knots-1)){
    datei=paste("mpi-jitter",workload,knots,i,"out", sep=".")
    hg=read.table(datei)
    daten[,i+1]=hg[,2]-hg[,1]      # 2-1 Spalte
  }
  return(daten)
}
```

Man erhält einen guten Eindruck von der Messung, wenn man sie alle in eine Abbildung einzeichnet. Über `OFFSET` und `LANG` kann man sich auch nur einen speziellen Abschnitt anzeigen lassen und mit `ylimits` lassen sich die Y-Werte beeinflussen.

```
CPU=128
WORK=100000
daten=readdata_compute(CPU,WORK)
OFFSET=0
LANG=length(daten[,1])
ylimits=vector(mode = "numeric", length = 2)
ylimits[1]=min(daten)
ylimits[2]=max(daten[(OFFSET+1):(OFFSET+LANG),])
x=seq(OFFSET+1,OFFSET+LANG, length=LANG)
plot(x,daten[(OFFSET+1):(OFFSET+LANG),1],ylim=ylimits,type="l",col=1)
for(i in 2:CPU){
        lines(x,daten[(OFFSET+1):(OFFSET+LANG),i],type="l",col=i)
}
```

Tauchen Auffälligkeiten auf – z.B. immer wieder sichtbare Störungen von einer bestimmten Größe, kann man über den Dichteschätzer die dominanten Messzeiten herausfiltern. Da aber die Anzahl der Messungen mit Störung oft sehr gering ist gegenüber den „normalen“ Messwerten, muss man den Dichteschätzer auf bestimmte Bereiche der Daten ausführen.

```
start=1700
stop=1800
plot(density(daten[daten>start],from=start,to=stop))
```

7.3 Anwendung des Konzepts exemplarisch auf den HPC-Systemen *JuMP* und *JuGene*

Um die Anwendung des Konzeptes zu veranschaulichen, wurden Messungen auf den beiden derzeit verfügbaren Supercomputern des Forschungszentrum Jülich durchgeführt. Es handelt sich zum einen um das *Jülich Multi Processor (JuMP)* System basierend auf der IBM Power 6 Technologie. Es besteht aus 14 Knoten mit jeweils 32 Prozessoren, die mit einer Taktfrequenz von 4,7GHz arbeiten. Die Prozessoren sind *SMT* fähig, d.h. sie können zwei Threads gleichzeitig verarbeiten. Das System besitzt eine *Peak Performance* von 8,5$TFlop/s$.

Für hochskalierende Projekte steht der *JuGene* Supercomputer zur Verfügung. Er besteht aus 16 *Racks* mit je 32 *Node Cards*, von denen jede 32 *Compute Cards* mit jeweils einem PowerPC 450 Chip enthält. Jeder Chip besitzt 4 Prozessoren, die mit einer Frequenz von 850MHz betrieben werden. Insgesamt stehen maximal 65536 Prozessoren zur Verfügung, die in der Summe eine *Peak Performance* von 223$TFlop/s$ besitzen.

7.3.1 FTQ Messungen auf dem *JuMP* System

Die Darstellungen der Messwerte, die auf dem *JuMP* System gemessen worden sind, zeigen, dass Störungen existieren. In Abbildung 7.1 sind die Daten des *FTQ* Benchmarklaufs mit einem Messintervall von 2^{18} Zyklen zu sehen. Bei der ersten Vergrößerung sind schon mehrere Klassen zu erkennen. Im rechten Bild sieht man, dass die zweite Linie von oben eine sehr regelmäßig auftretende Störung ist. Es handelt sich wahrscheinlich um den *System Tick* des Systems.

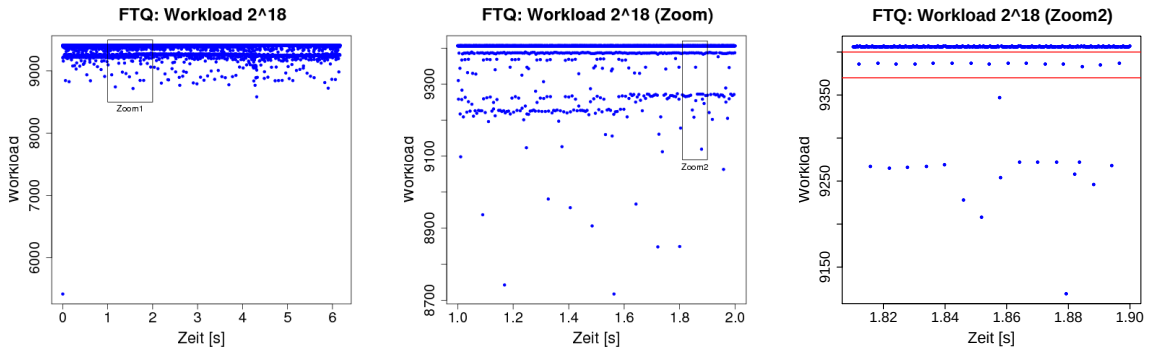


Abbildung 7.1: FTQ Messung auf dem *JuMP* System mit einem Messintervall von 2^{18} Zyklen

Schneidet man alle Messdaten heraus, die zu dieser Störung gehören (Anwendung der `cutoff`-Funktion), ergibt der Dichteschätzer für die Zeitabstände zwischen den Messungen einen Wert von 6ms (siehe Abb. 7.2 auf der nächsten Seite).

Die Frequenzanalyse der Daten mithilfe der *Fast Fourier-Transformation* bestätigt das Ergebnis (mittleres Bild). Sie zeigt eine dominante Frequenz von 166Hz und deren Vielfache. Insgesamt tauchen in dieser Messung keine gravierenden Störungen auf, denn die Zeitintervalle werden während des Benchmarks ziemlich genau eingehalten (rechtes

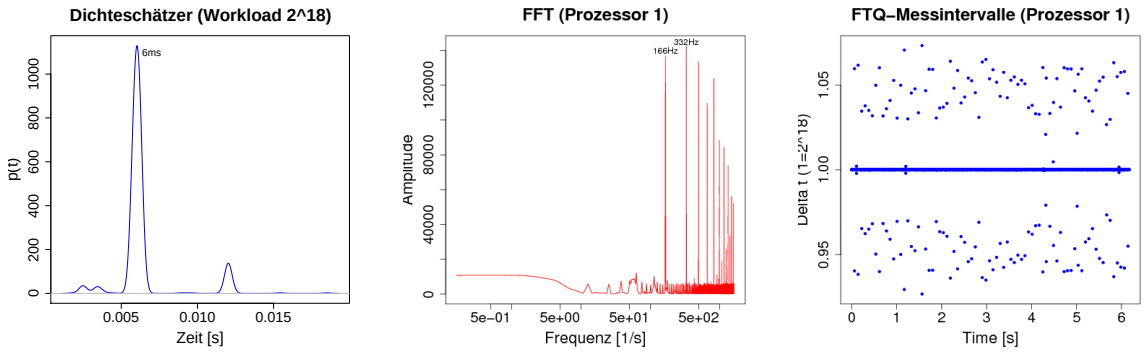


Abbildung 7.2: FTQ-Analyse der Messungen auf dem *JuMP* System: Links der Dichteschätzer auf die Abstände der ersten Störungen, in der Mitte die Anwendung der geglätteten *Fast Fourier-Transformation* und rechts die automatische Justierung der Zeitintervalle

Bild). Grund dafür ist die *SMT* Fähigkeit der Prozessoren. Der Benchmarklauf wurde auf 2×32 Prozessoren ausgeführt, ohne den zweiten *Thread* zu nutzen. Damit standen dem Betriebssystem auf jedem Knoten 32 *Threads* zur Verfügung, um Störungen abzuarbeiten.

7.3.2 FTQ Messungen auf dem *JuGene* System

Bei dem *JuGene* System fällt kaum ein *OS-Jitter* auf. Das zeigt, dass IBM mit dem *Compute Node Kernel* keine Störungen hat. In Abbildung 7.3 sieht man zwar eine Störung, die sehr regelmäßig für eine Messung auftaucht, aber sie wirkt sich seltsamerweise nach oben aus.

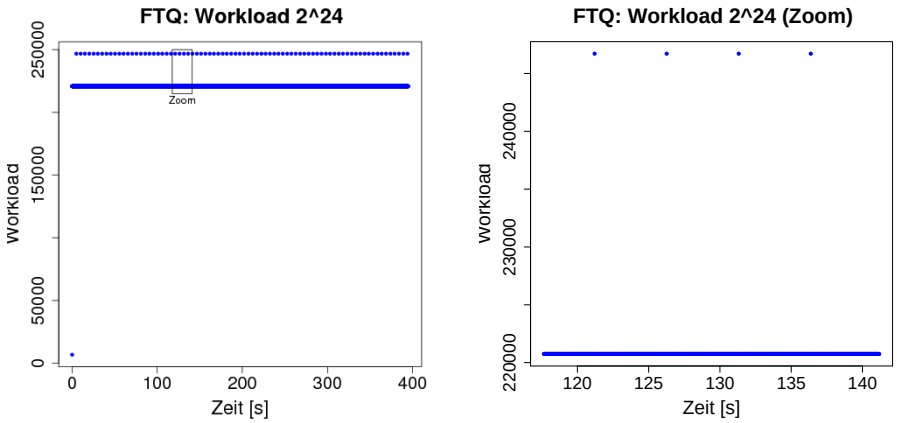


Abbildung 7.3: FTQ Messung auf dem *JuGene* System mit einem Messintervall von 2^{24} Zyklen

Wendet man auf die Zeitabstände der Störungen den Dichteschätzer an, so erhält man ziemlich genau eine Zeit von 5,0529s (Abb. 7.4 auf der nächsten Seite, linkes Bild). Die

Analyse der Daten mithilfe der *Fast Fourier-Transformation* bestätigt das Ergebnis sehr gut. Es wird eine Frequenz von ca. 0,2Hz und deren Vielfache sichtbar. Das entspricht einem Abstand von 5s. Als Ursache ist ein interessanter Effekt bei den ausgelesenen Werten des *Time Stamp Counter* Register zu sehen. Nach genau 5,0525s wird dessen Wert auf „0“ zurückgesetzt. Wahrscheinlich handelt es sich um eine Art *System Tick* bzw. einem Äquivalent dazu.

Das Zurücksetzen des Registers würde erklären, dass keine Störung an den Stellen vorliegt, sondern nur ein Messfehler des Benchmarks.

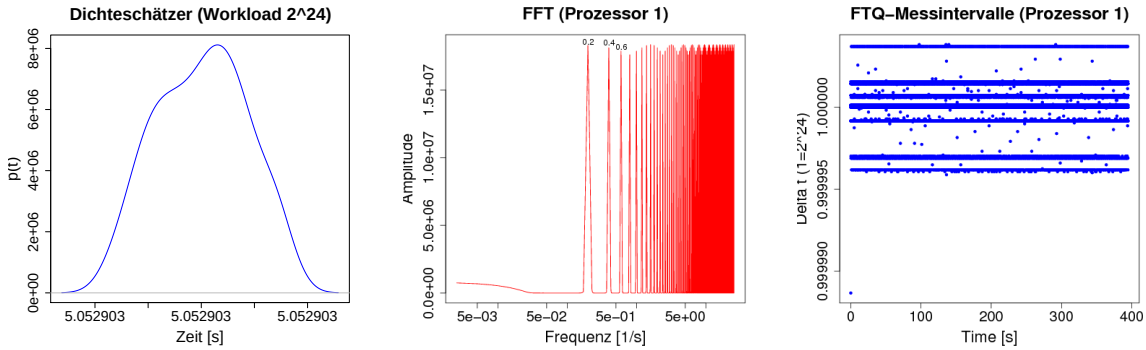


Abbildung 7.4: *FTQ* Analyse der Messungen auf dem *JuGene* System: Links der Dichteschätzer auf die Abstände der Störungen, in der Mitte die Anwendung der geglätteten *Fast Fourier-Transformation* und rechts die automatische Justierung der Zeitintervalle

7.3.3 MPI-Jitter Messungen auf dem *JuMP* System

Auf dem *JuMP* System wurde eine Messung mit 64 Prozessoren und einem Workload von 100 (*Compute Phase*) durchgeführt. Die Betrachtung von den Messzeiten für t_all in Abbildung 7.5 auf der nächsten Seite zeigt, dass die Prozesse alle nahezu gleichzeitig fertig werden. Selbst bei der starken Vergrößerung im mittleren Bild sieht man, dass sich alle sehr ähnlich verhalten. Der Dichteschätzer ergibt eine Zeit von 1,9μs.

Bei den Messdaten für $t\_compute$ sieht man mehr Unterschiede. In den beiden linken Diagrammen von Abbildung 7.6 auf der nächsten Seite sind verschiedene horizontale Linien zu erkennen. Der Dichteschätzer für $t\_compute > 0.4$ (rechtes Bild) liefert die Information für die Zeiten 0,44μs, 0,49μs, 1,28μs und 1,36μs.

7.3.4 MPI-Jitter Messungen auf dem *JuGene* System

Auf dem *JuGene* wurde der *MPI-Jitter* Benchmark mit einem Workload von 100 auf 1024 Prozessoren gestartet. In Abbildung 7.7 auf Seite 92 ist ein Ausschnitt der Messzeiten t_all aller Prozessoren zu sehen. Über die gesamte Messung beträgt das absolute Minimum 8,591ms und das absolute Maximum 8,593ms.

Bei den Messungen für die $t\_compute$-Zeiten sind unterschiedliche Werte zu erkennen, wie es in Abbildung 7.8 auf Seite 92 im linken Bild zu sehen ist. Der Dichteschätzer bestätigt dieses Ergebnis. Die Masse der Zeiten befindet sich knapp über 8,656μs, aber

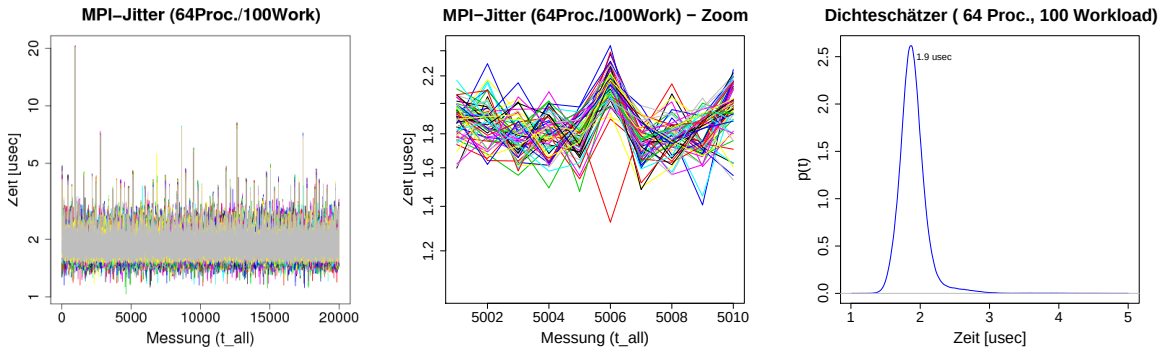


Abbildung 7.5: *MPI-Jitter* Messung für *t_all* auf dem *JuMP* System mit 64 Prozessoren und einem *Workload* von 100. Rechts der Dichteschätzer

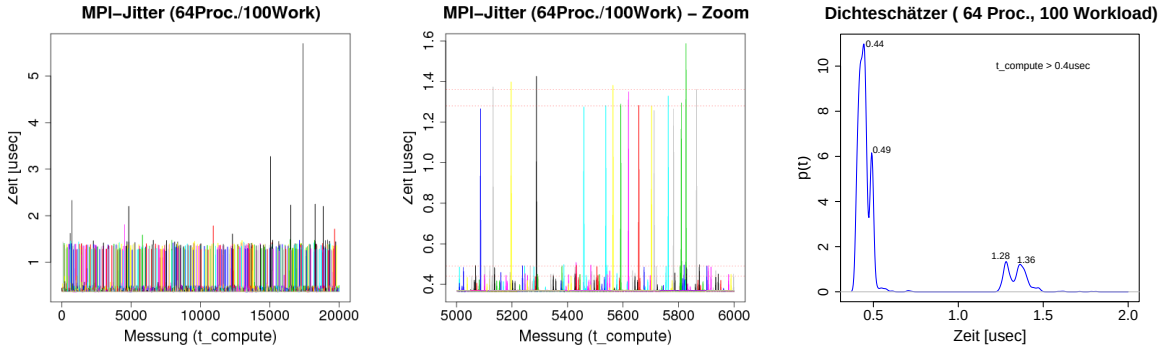


Abbildung 7.6: *MPI-Jitter* Messung für *t_compute* auf dem *JuMP* System mit 64 Prozessoren und einem Workload von 100. Rechts der Dichteschätzer

es gibt noch zwei weitere Häufungspunkte bei 8,672μs und bei 8,686μs. Für die gesamte Messung gilt, dass sich die Werte im Intervall $[8{,}655\mu s, 8{,}928\mu s]$ befinden. In diesem Fall tauchen keine großen Schwankungen auf.

Es hat sich gezeigt, dass die Datenmenge des *MPI-Jitter* für größere Prozessorzahlen sehr groß werden. Die Datenauswertung benötigt dementsprechend mehr Hauptspeicher auf dem Rechner, auf dem sie durchgeführt wird.

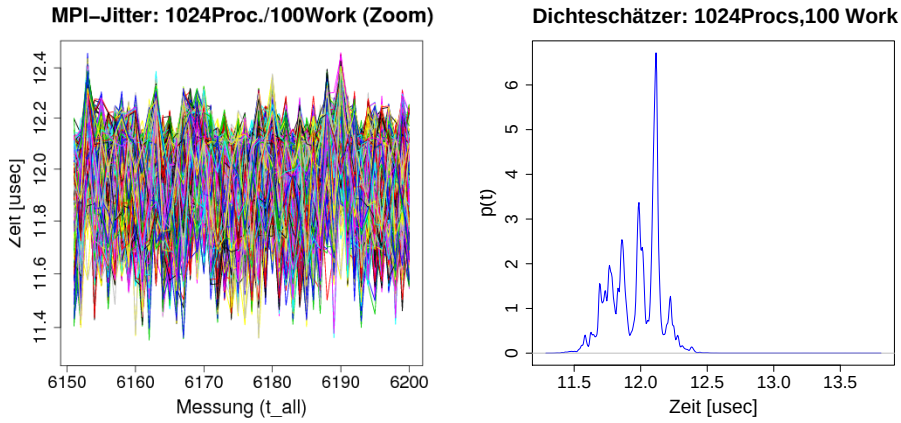


Abbildung 7.7: *MPI-Jitter* Benchmark auf dem *JuGene* (*t_all*): 1024 Prozessoren, 100er *Workload*

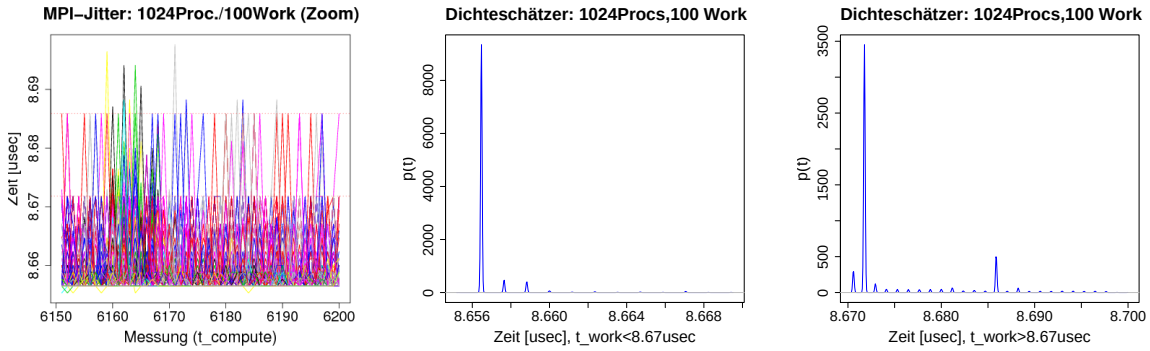


Abbildung 7.8: *MPI-Jitter* Benchmark auf dem *JuGene* (*t_compute*): 1024 Prozessoren, 100er Workload

Kapitel 8

Ausblick

Im Laufe des Jahres 2009 wird im Forschungszentrum Jülich im Rahmen des *JuRoPA*[1] Projektes ein neues *General Purpose HPC*-System aufgebaut.

Das System wird von der Firma *Bull GmbH* als Generalunternehmer geliefert. Weitere Teilkomponenten werden von *SUN Microsystems* zugeliefert. Darüberhinaus werden Infiniband-Komponenten von *Mellanox Technologies* sowie Prozessoren von *Intel* verwendet. *ParTec Cluster Competence Center GmbH* liefert die *Parastation* Software als *Cluster Middleware* und ist für die Softwareintegration verantwortlich. Das *Forschungszentrum Jülich GmbH* ist Käufer und Betreiber des Rechners. Alle genannten Firmen sind auch am *JuRoPA* Projekt beteiligt.

Das *JuRoPA*-Cluster wird aus insgesamt 3288 *Compute Nodes* bestehen. Jeder Knoten enthält 2 *Nehalem* CPUs mit einer Taktfrequenz von ca. $2,93GHz$. Eine CPU besteht aus 4 Prozessoren, die jeweils als zwei virtuelle Prozessoren vom Betriebssystem erkannt werden.

Damit stehen in der Summe über 26.000 physikalische Prozessoren (virtuell 52.000) zur Verfügung. Als Betriebssystem ist ein *SuSE Linux Enterprice Real Time (SLERT)* vorgesehen.

Bei den Auswertungen hat sich gezeigt, dass auf einem Cluster mit *SLERT OS-Jitter* Effekte gut zu sehen sind. Die Arbeiten durch Betriebssystem, Netzwerk und Dienste führten immer wieder zu Wartezeiten der Prozesse. Besonders, wenn man die Ergebnisse mit den Messungen auf dem *JuGene* System vergleicht, bei denen keine Störung zu sehen war.

Das bedeutet, dass es auf dem zukünftigen *JuRoPA* System *OS-Jitter* geben wird. Um die Leistung problematischer Anwendungen zu steigern, muss das Betriebssystem entsprechend angepasst werden. Im *JuRoPA* Vertrag wurde unter anderem ein Projekt zur Entwicklung eines entsprechend optimierten Betriebssystem vereinbart. Zu diesem Zweck wird eine bestimmte Anzahl an Knoten auf dem System reserviert.

Die Erfahrungen, welche im Rahmen dieser Arbeit gesammelt wurden, bilden die Basis für dieses Entwicklungsprojekt. Um das Betriebssystem zu optimieren, müssen die folgenden Schritte umgesetzt werden:

1. Das Betriebssystem muss minimiert werden, d.h. alle unnötigen Dienste können ausgeschaltet werden.

2. Der Linux Kernel kann an die Hardware angepasst werden. Alle nicht benötigten Eigenschaften können ausgeschaltet werden.

[1] *Jülich's Research on Peta-flop Architectures*

3. Die Echtzeitfähigkeit des Betriebssystems bietet die Möglichkeit, Prozesse zu priorisieren und sie gegen Unterbrechungen zu schützen.

Der *OS-Jitter* kann mit dem *Fix Time Quantum* Benchmark gut sichtbar gemacht werden. Damit kann man Änderungen am System bei Benchmarkläufen direkt auf ihren positiven Effekt untersuchen. Der *MPI-Jitter* Benchmark über mehrere Knoten bietet im Anschluss die Möglichkeit, eine generelle Leistungsänderung festzustellen. Zu diesen Zweck wird dieser Benchmark um statistische Auswertungmethoden erweitert werden müssen, da mit steigender Zahl an Prozessoren das Datenvolumen wächst und die Auswertung der Messungen aufwendiger wird.

Anhang A

Messungen mit dem FTQ-Benchmark

A.1 openSuSE 10.2 mit KTAU Patch

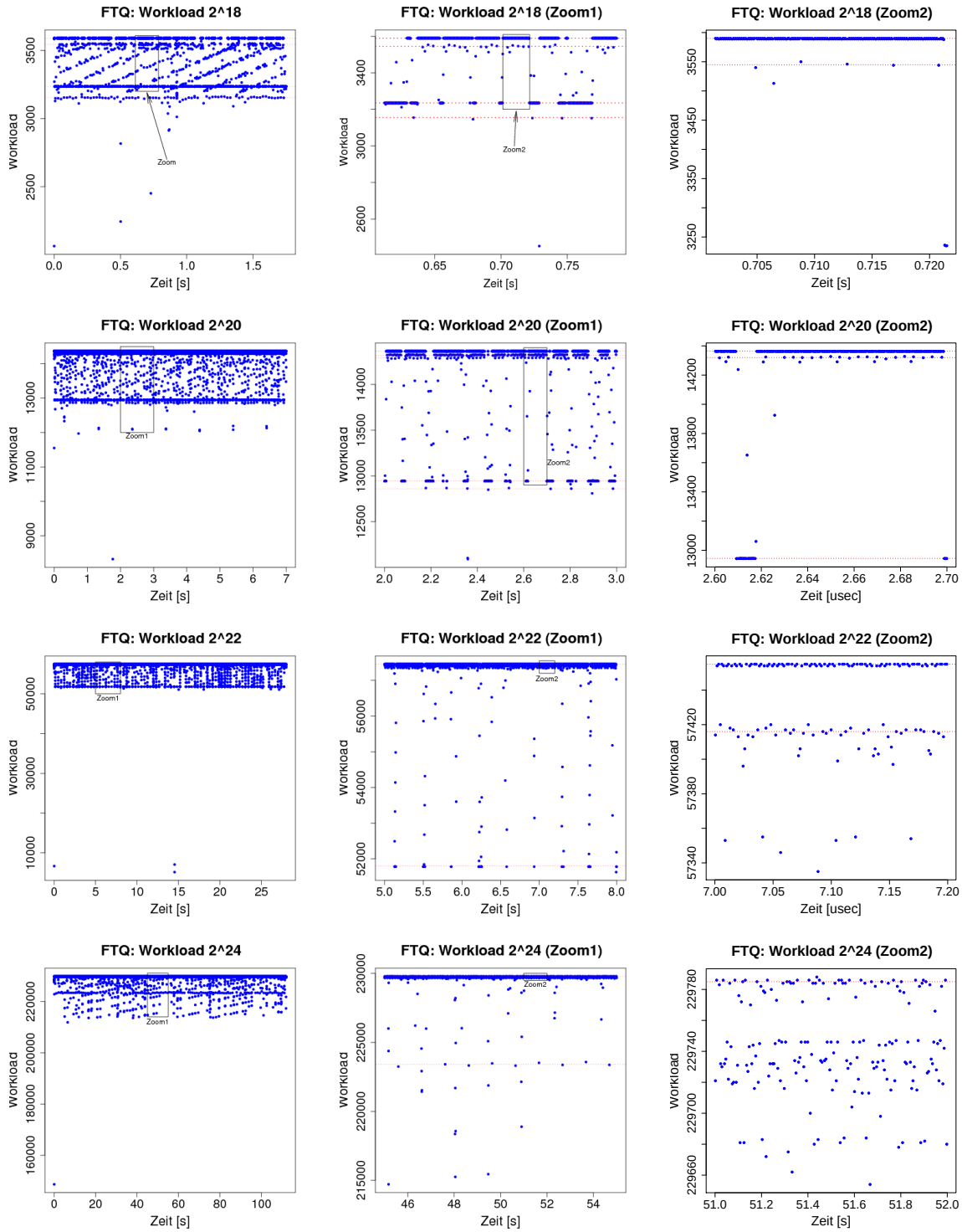


Abbildung A.1: *FTQ* Messwerte für 2^{18}, 2^{20}, 2^{22} und 2^{24} Zyklen Messintervalle (*openSuSE* Betriebssystem)

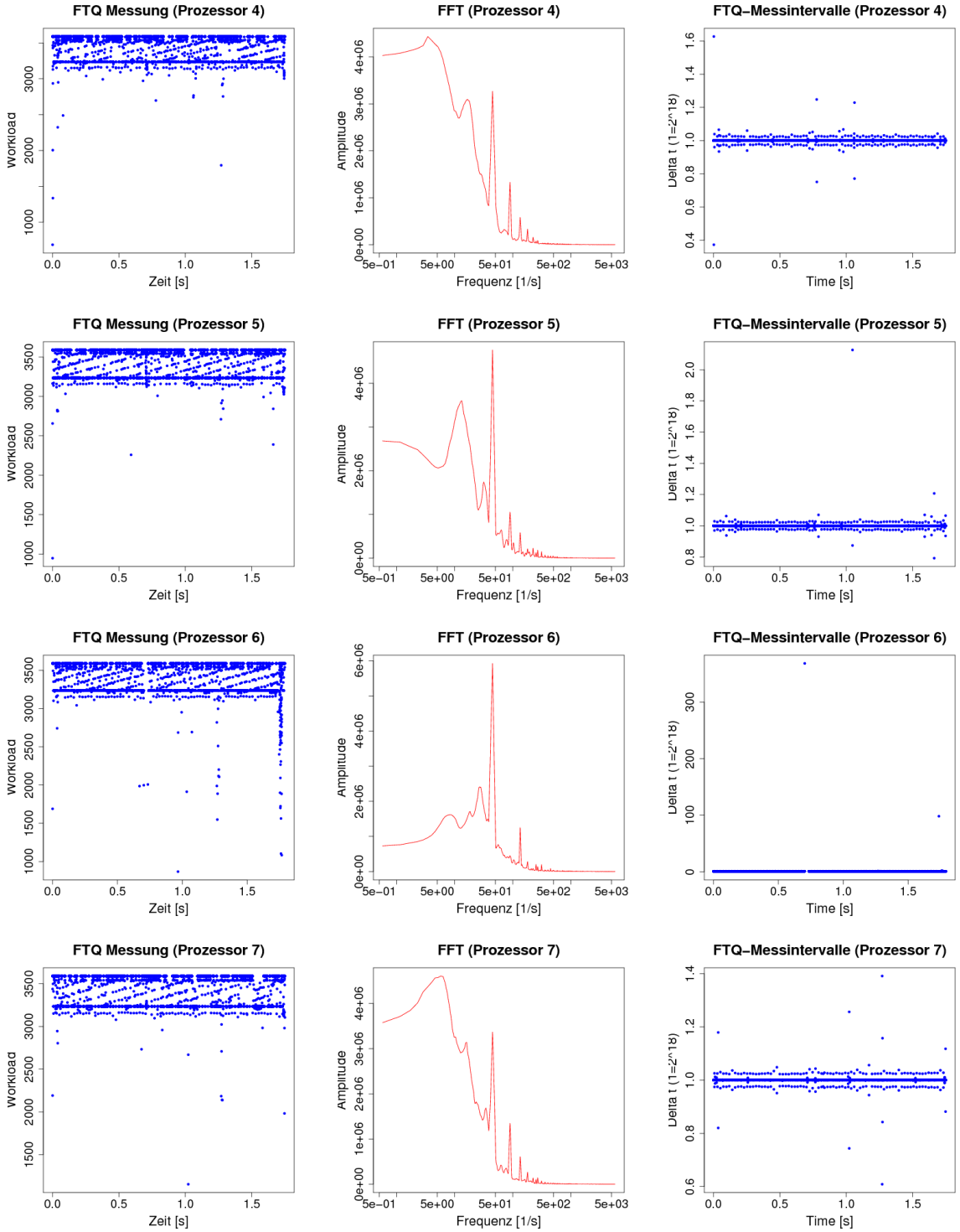


Abbildung A.2: *FTQ* Messung auf vier Prozessoren eines Knoten mit dem *openSuSE* Betriebssystem (2^{18} Zyklen Messintervalle)

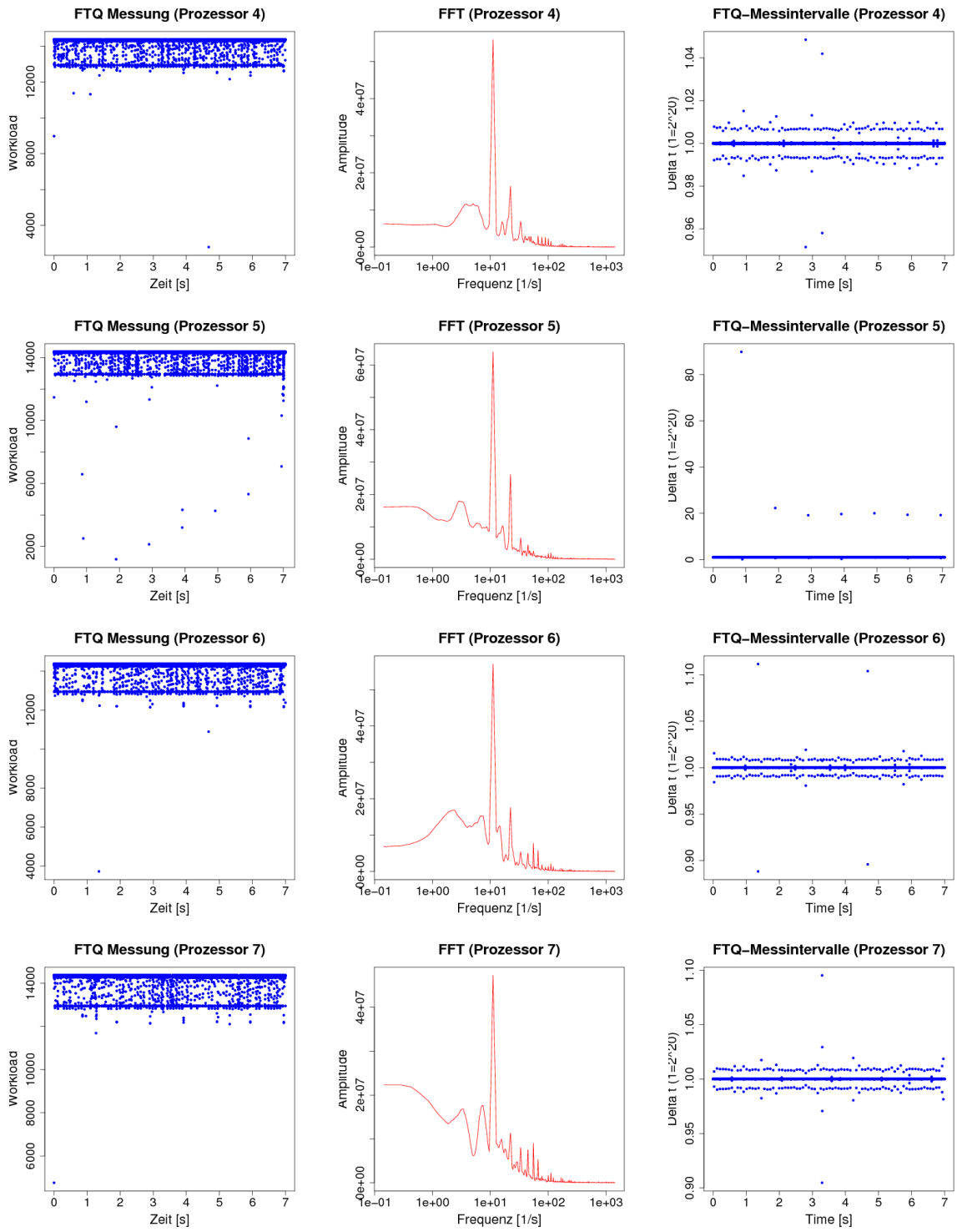


Abbildung A.3: *FTQ* Messung auf vier Prozessoren eines Knoten mit dem *openSuSE* Betriebssystem (2^{20} Zyklen Messintervalle)

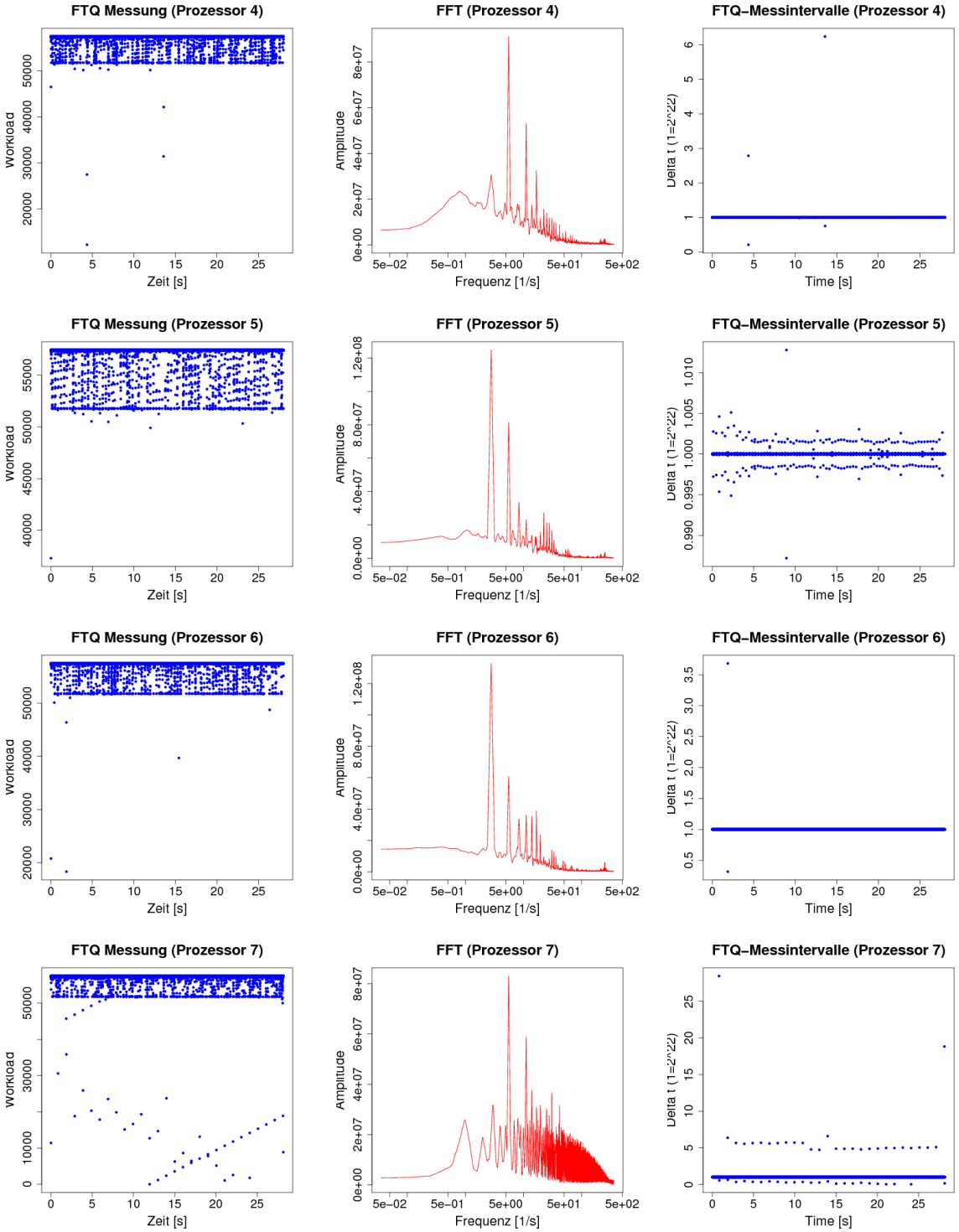


Abbildung A.4: *FTQ* Messung auf vier Prozessoren eines Knoten mit dem *openSuSE* Betriebssystem (2^{22} Zyklen Messintervalle)

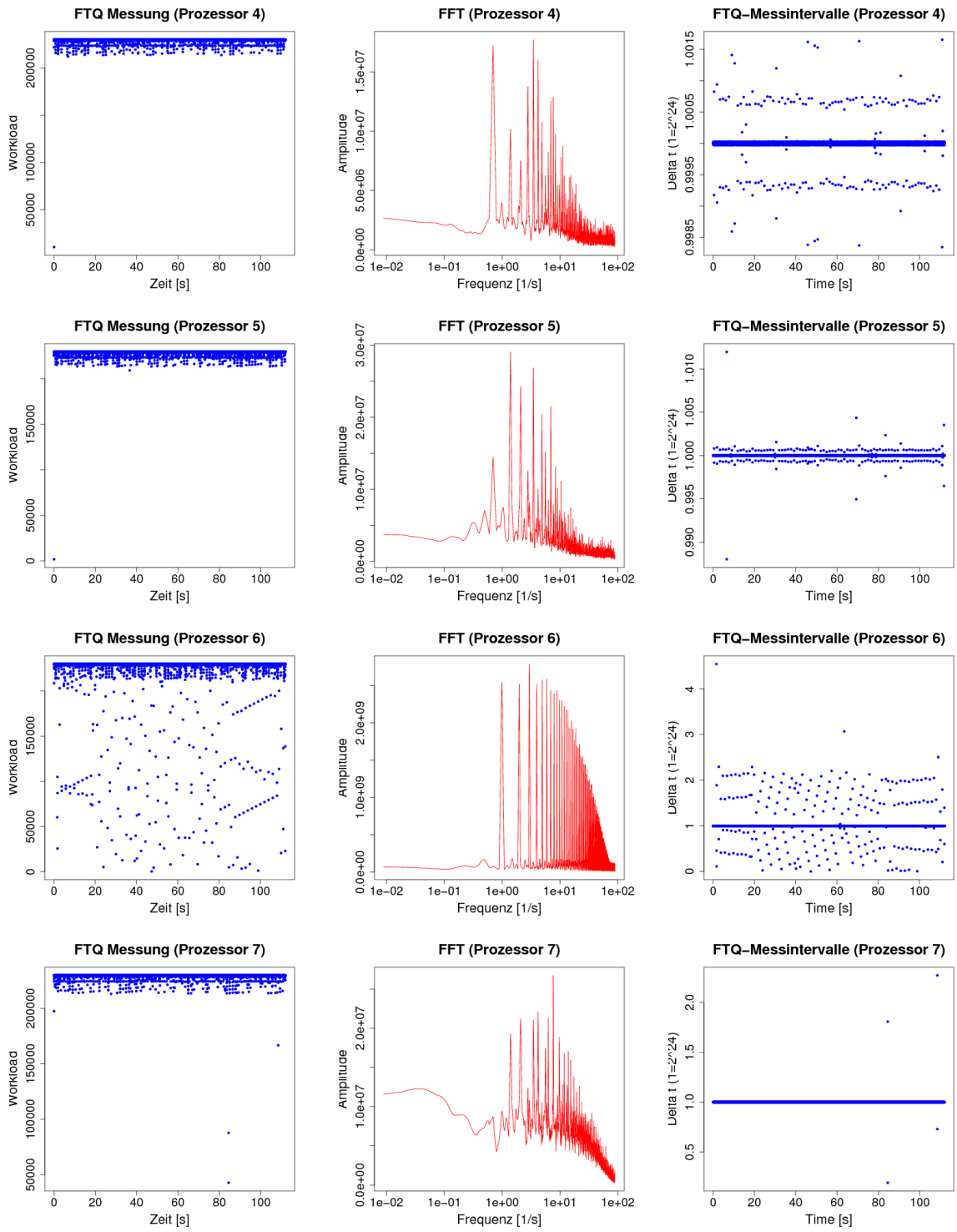


Abbildung A.5: *FTQ* Messung auf vier Prozessoren eines Knoten mit dem *openSuSE* Betriebssystem (2^{24} Zyklen Messintervalle)

A.2 SuSE Linux Enterprise Realtime

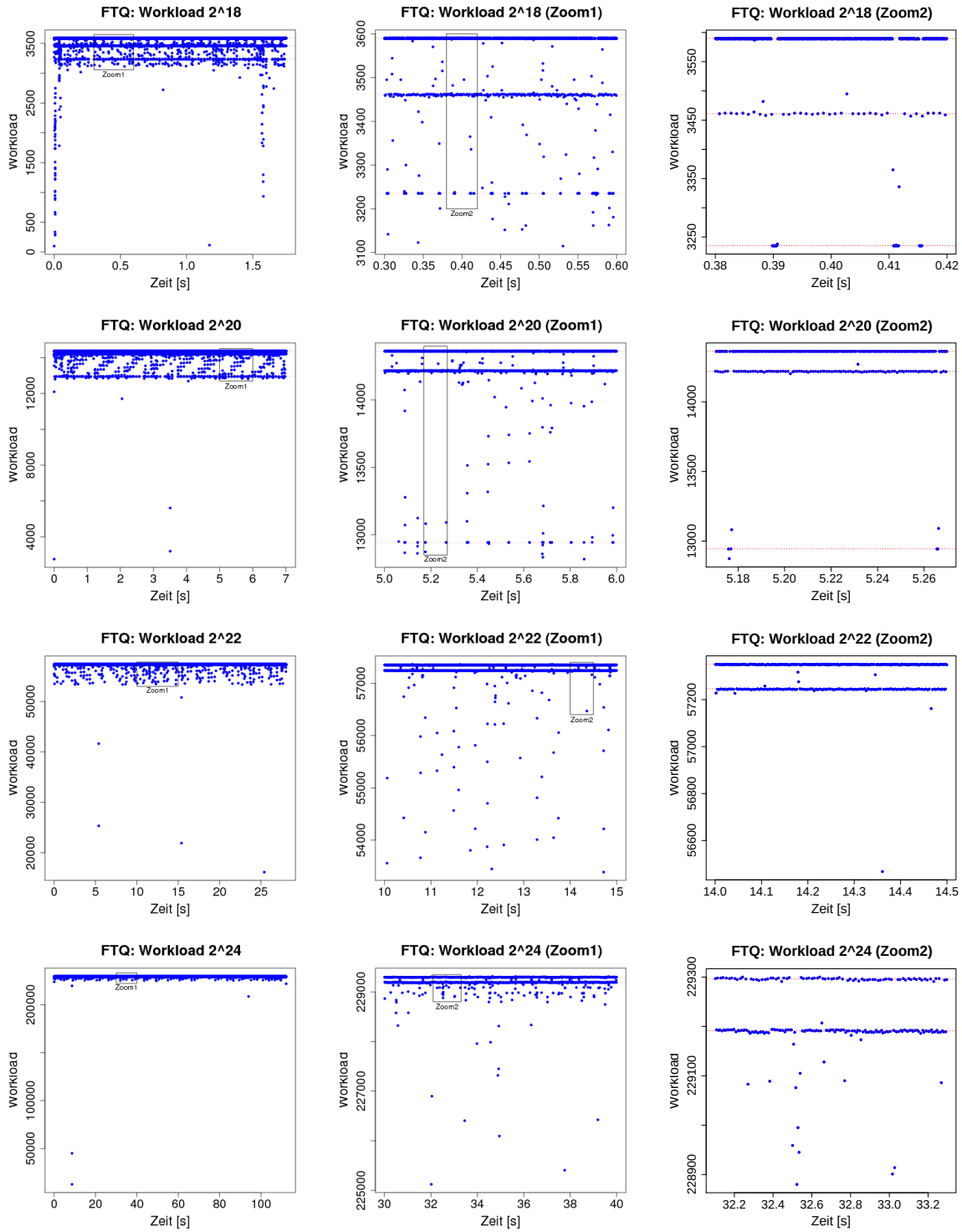


Abbildung A.6: *FTQ*-Messwerte für 2^{18}, 2^{20}, 2^{22} und 2^{24} Zyklen Messintervalle (*SLERT* Betriebssystem)

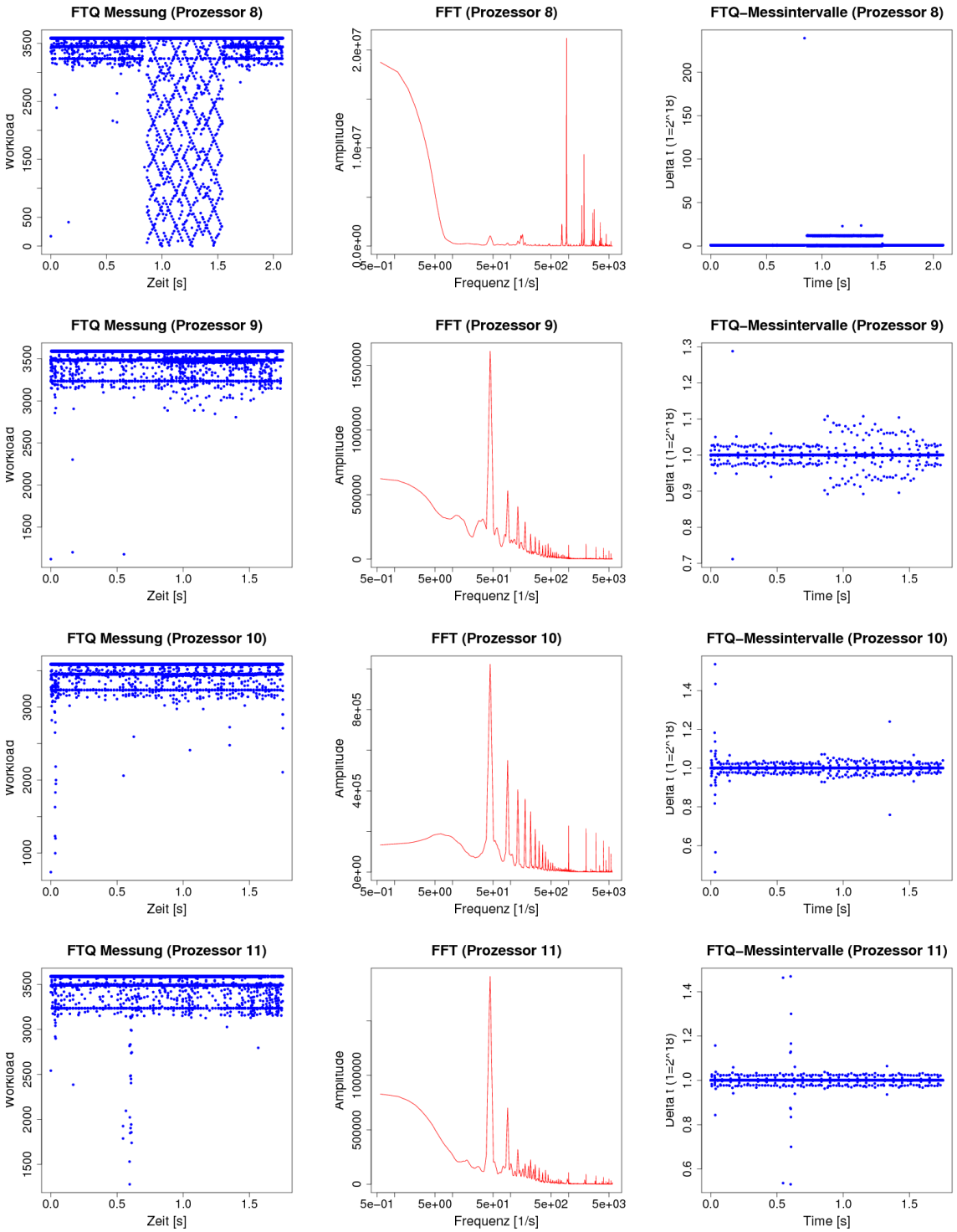


Abbildung A.7: *FTQ* Messung auf vier Prozessoren eines Knoten mit dem *SLERT* Betriebssystem (2^{18} Zyklen Messintervalle)

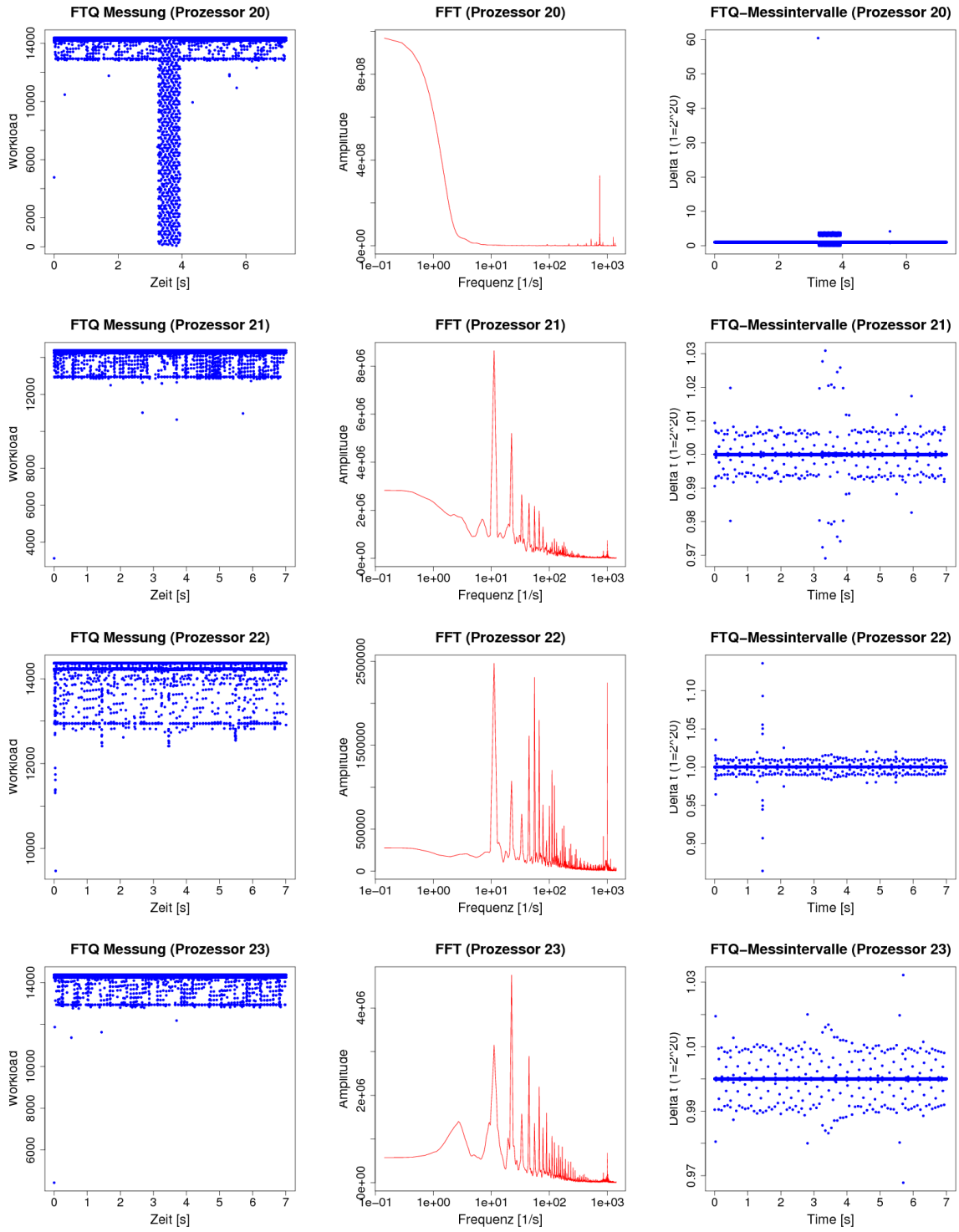


Abbildung A.8: *FTQ* Messung auf vier Prozessoren eines Knoten mit dem *SLERT* Betriebssystem (2^{20} Zyklen Messintervalle)

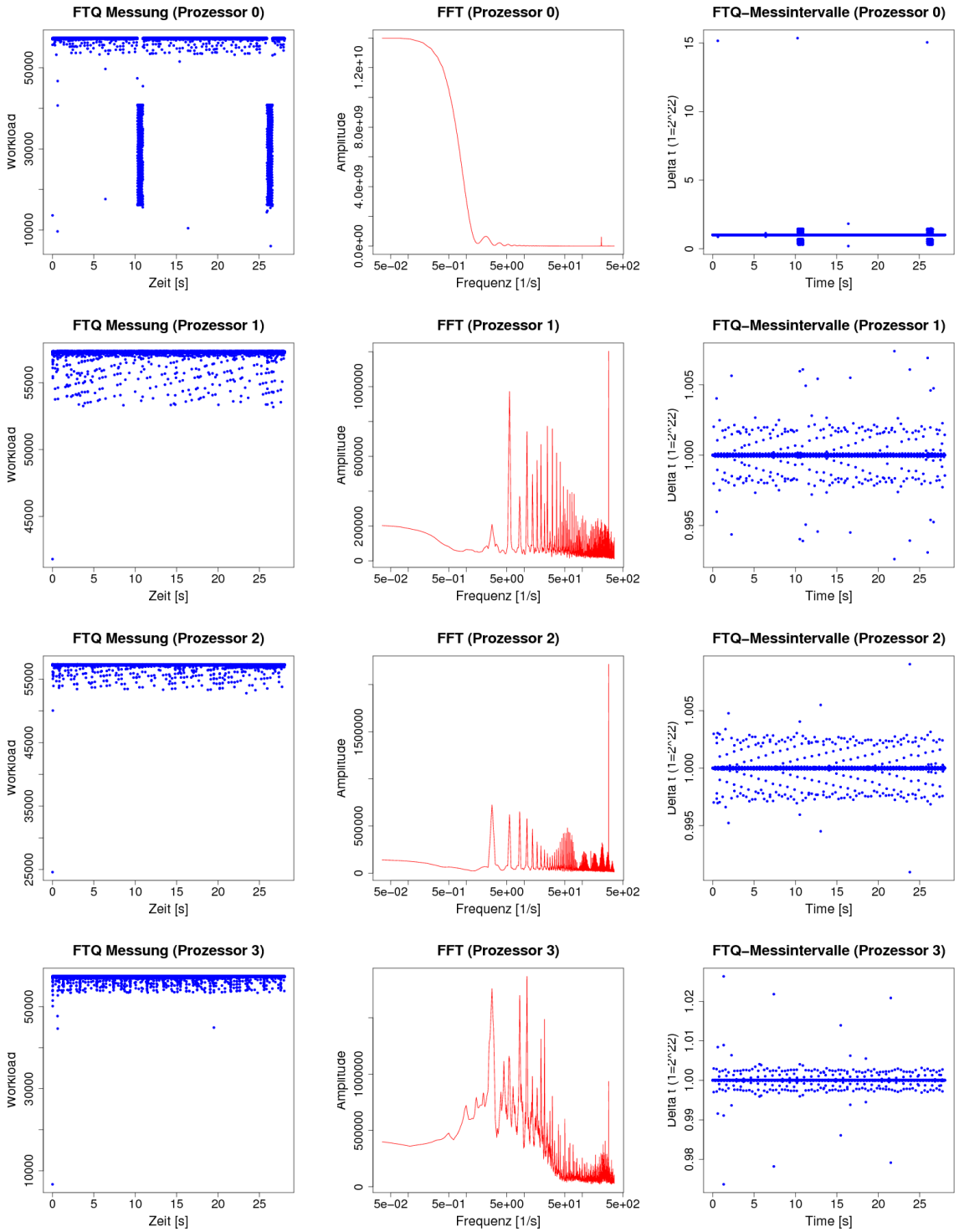


Abbildung A.9: *FTQ* Messung auf vier Prozessoren eines Knoten mit dem *SLERT* Betriebssystem (2^{22} Zyklen Messintervalle)

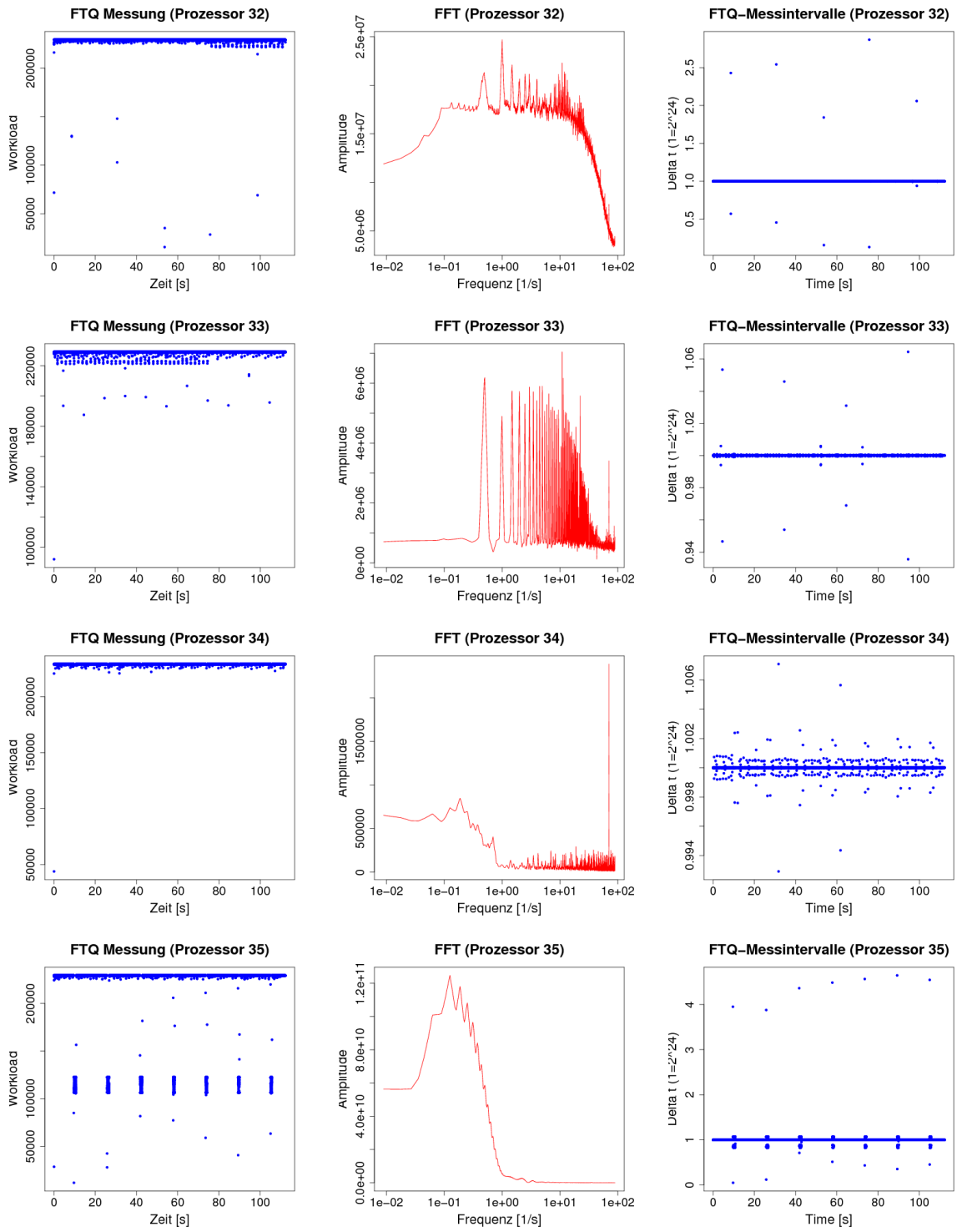


Abbildung A.10: *FTQ* Messung auf vier Prozessoren eines Knoten mit dem *SLERT* Betriebssystem (2^{24} Zyklen Messintervalle)

Anhang B

Messungen mit dem *MPI-Jitter* Benchmark

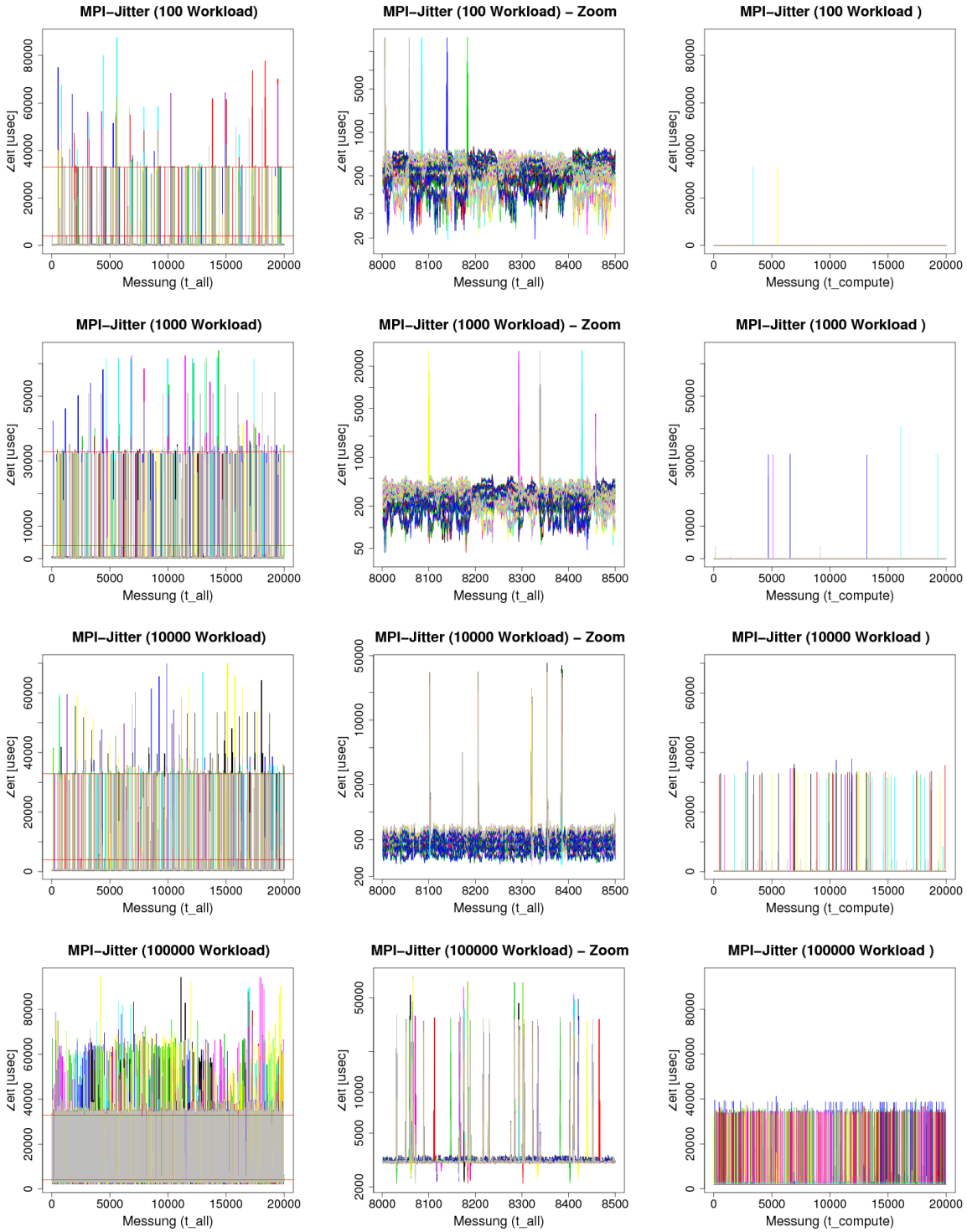


Abbildung B.1: *MPI-Jitter* Messung auf dem *JuRoPA* Testsystem mit 64 Prozessoren und dem *openSuSE* Betriebssystem

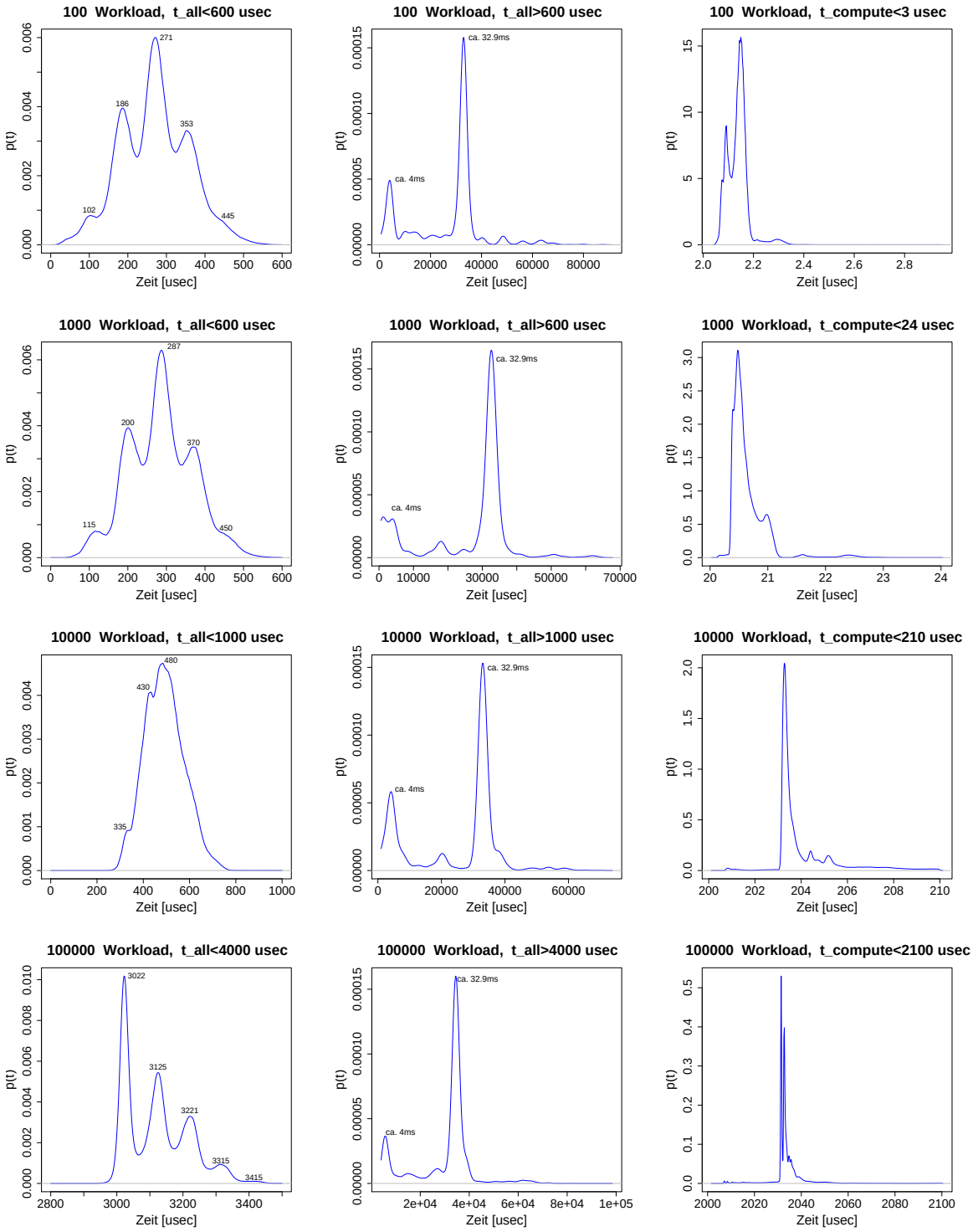


Abbildung B.2: Dichteschätzer für die *MPI-Jitter* Messung auf dem *JuRoPA* Testsystem mit 64 Prozessoren und dem *openSuSE* Betriebssystem

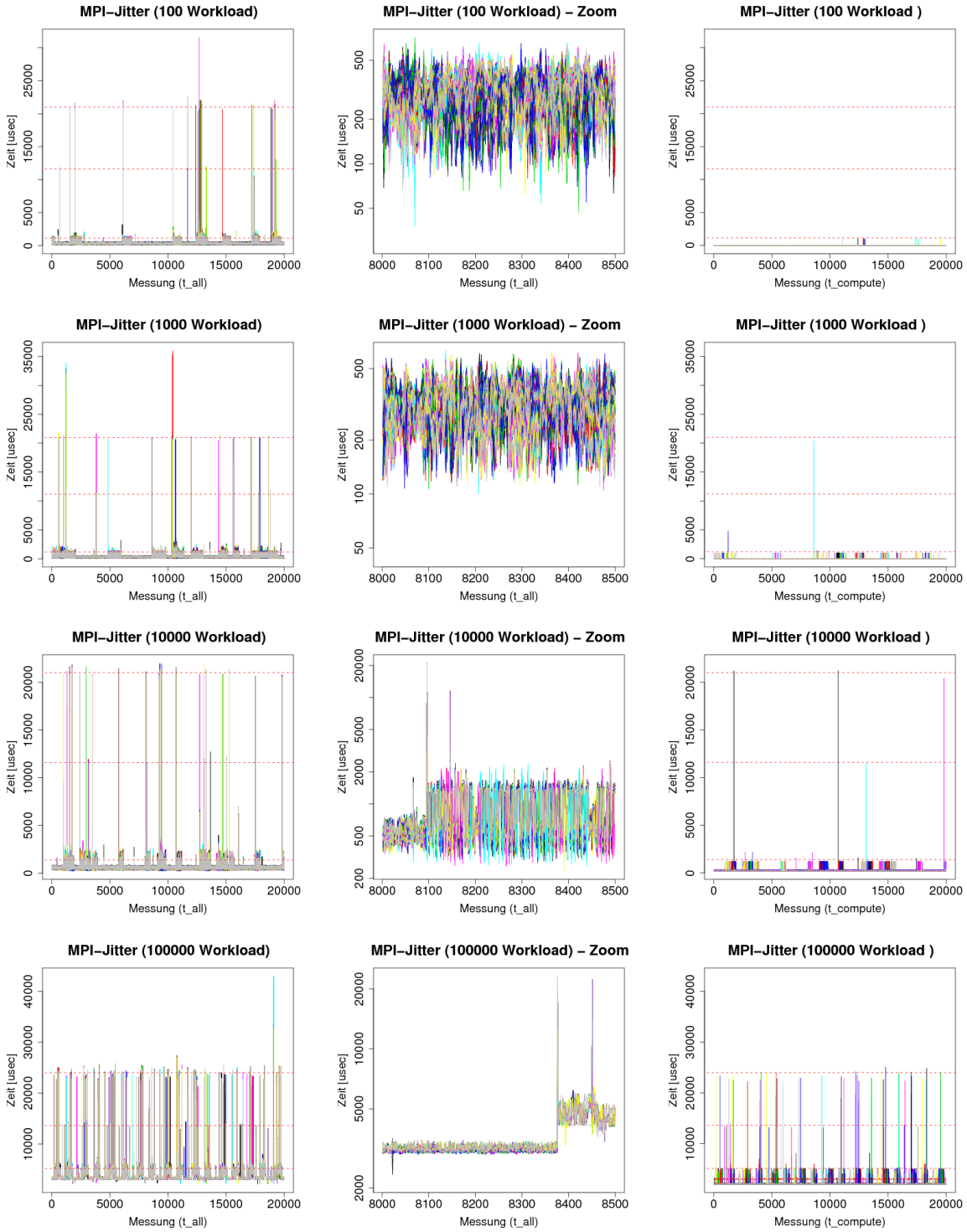


Abbildung B.3: *MPI-Jitter* Messung auf dem *JuRoPA* Testsystem mit 64 Prozessoren und dem *SuSE Linux Enterprise Real Time* Betriebssystem

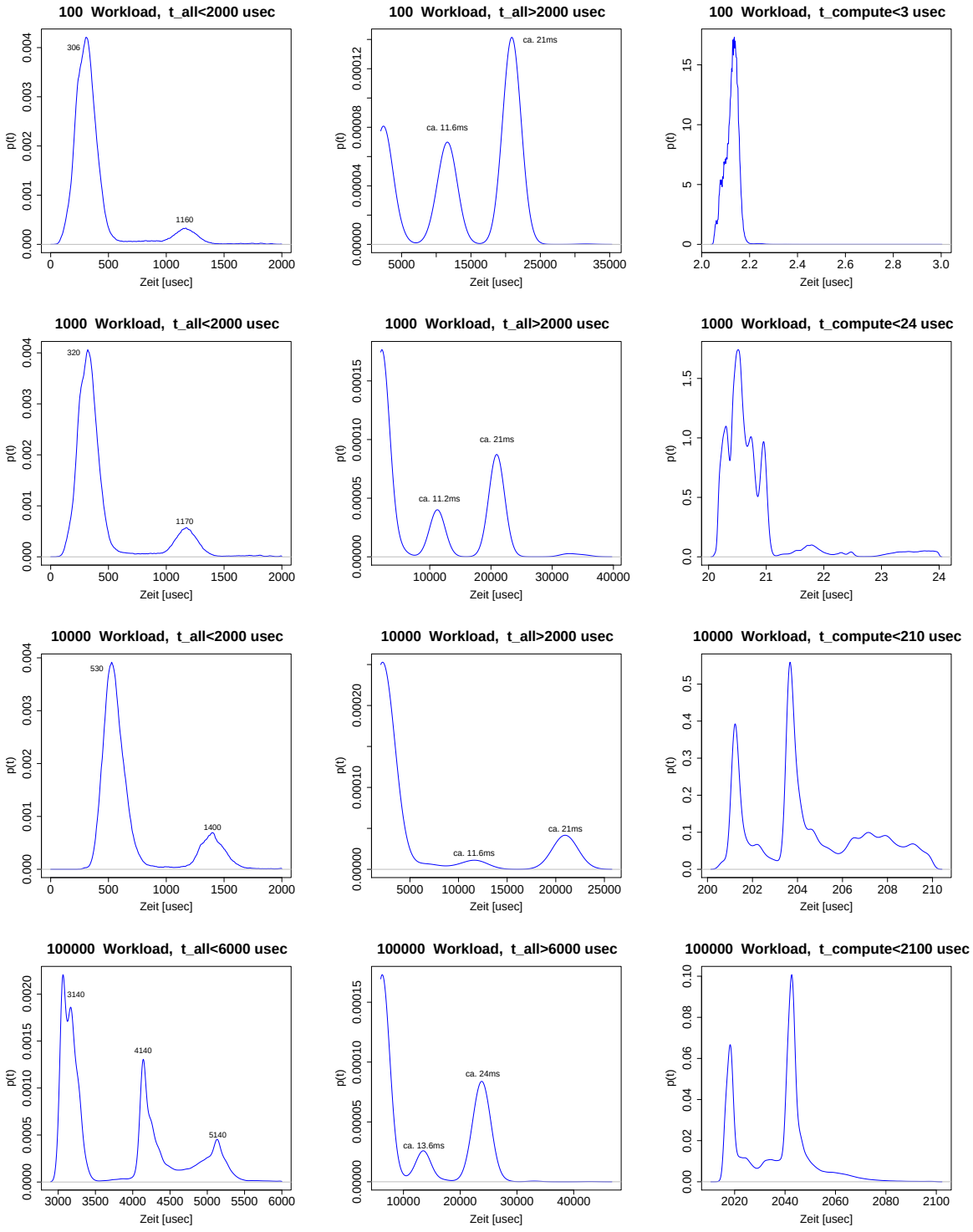


Abbildung B.4: Dichteschätzer für die *MPI-Jitter* Messung auf dem *JuRoPA* Testsystem mit 64 Prozessoren und dem *SuSE Linux Enterprise Real Time* Betriebssystem

schnellster Knoten ⇓	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15	#16
#1 (5.0%)	1.0	4.3	4.2	9.6	3.5	8.5	8.5	13.4	3.6	8.6	8.5	13.5	7.5	12.8	12.7	15.9
#2 (4.9%)	4.3	1.0	9.7	4.1	8.5	3.5	13.4	8.3	8.7	3.7	13.5	8.4	12.9	7.4	15.9	12.7
#3 (6.6%)	4.3	9.7	1.0	4.3	8.2	13.4	3.4	8.5	8.6	13.7	3.6	8.8	12.7	15.9	7.3	12.7
#4 (6.2%)	9.7	4.2	4.4	1.0	13.3	8.0	8.6	3.4	13.6	8.5	9.0	3.8	15.9	12.6	12.8	7.2
#5 (5.2%)	3.6	8.8	8.3	13.3	1.0	4.4	4.3	9.8	7.3	12.8	12.6	15.8	3.7	8.7	8.1	13.4
#6 (5.2%)	8.3	3.4	13.1	8.0	4.4	1.0	9.6	4.2	12.8	7.6	15.8	12.6	9.0	3.8	13.6	8.5
#7 (6.2%)	8.3	13.4	3.4	8.3	4.2	9.5	1.0	4.3	12.7	15.9	7.7	12.9	8.6	13.6	3.5	8.7
#8 (6.1%)	13.2	8.2	8.5	3.4	9.4	4.1	4.2	1.0	15.9	12.8	13.0	7.7	13.6	8.5	8.8	3.8
#9 (7.6%)	3.8	9.0	8.5	13.5	7.6	12.9	12.6	15.9	1.0	4.4	4.0	9.4	3.5	8.4	8.1	13.2
#10 (8.4%)	8.7	3.6	13.6	8.5	12.9	7.6	15.9	12.8	4.2	1.0	9.4	4.1	8.5	3.5	13.4	8.4
#11 (7.3%)	8.5	13.7	3.7	9.0	12.7	15.9	7.5	13.0	3.9	9.4	1.0	4.3	8.2	13.3	3.3	8.5
#12 (7.1%)	13.6	8.6	8.8	3.7	15.9	12.9	13.0	7.7	9.2	4.1	4.2	1.0	13.3	8.3	8.4	3.4
#13 (5.5%)	7.4	12.9	12.7	15.9	3.8	8.9	8.4	13.7	3.3	8.6	8.1	13.2	1.0	4.3	4.1	9.7
#14 (6.1%)	12.9	7.4	15.9	12.6	8.9	3.7	13.6	8.5	8.4	3.4	13.3	8.1	4.4	1.0	9.8	4.2
#15 (6.6%)	12.5	15.8	7.5	12.8	8.4	13.5	3.8	9.0	8.1	13.2	3.6	8.5	4.2	9.6	1.0	4.4
#16 (6.1%)	15.8	12.6	12.8	7.3	13.5	8.5	8.8	3.7	13.2	8.2	8.6	3.6	9.8	4.3	4.4	1.0

Tabelle B.1: Rangfolge der Knoten (abhängig vom schnellsten Knoten) beim *openSuSE* System mit einem 100-er *Compute*-Phase

schnellster Knoten ⇓	#1	#2	#3	#4	#5	#6	#7	#8	#9	#10	#11	#12	#13	#14	#15	#16
#1 (6.9%)	1.0	3.7	4.8	8.5	4.3	8.1	9.4	13.0	4.1	7.7	9.1	12.6	8.7	12.2	13.4	15.6
#2 (6.6%)	3.8	1.0	8.3	5.0	8.2	4.6	12.8	9.6	7.6	4.1	12.4	9.0	12.1	8.6	15.5	13.4
#3 (5.7%)	4.9	8.4	1.0	3.9	9.4	12.7	4.4	8.1	9.2	12.6	4.2	7.8	13.3	15.5	8.7	12.1
#4 (5.1%)	8.4	5.0	4.1	1.0	12.7	9.5	8.2	4.5	12.4	9.0	7.8	4.2	15.4	13.1	12.0	8.6
#5 (5.9%)	4.2	8.0	9.3	12.7	1.0	3.8	4.7	8.5	8.7	12.3	13.4	15.6	4.1	7.8	9.2	12.7
#6 (5.8%)	8.0	4.5	12.8	9.7	3.7	1.0	8.6	5.1	11.9	8.5	15.5	13.5	7.6	4.1	12.4	9.2
#7 (6.4%)	9.4	13.0	4.3	8.1	4.7	8.4	1.0	3.7	13.5	15.6	8.7	12.3	9.1	12.7	3.9	7.7
#8 (7.1%)	13.0	9.6	8.3	4.3	8.6	4.9	3.8	1.0	15.6	13.3	12.0	8.4	12.7	9.0	7.5	3.9
#9 (7.1%)	4.0	7.3	8.9	12.5	8.7	12.1	13.6	15.6	1.0	3.6	4.9	8.4	4.5	8.1	9.8	13.1
#10 (8.0%)	7.9	4.0	12.5	8.9	12.3	8.7	15.6	13.2	3.9	1.0	8.4	4.6	8.3	4.4	13.0	9.3
#11 (6.2%)	9.3	12.4	4.0	7.4	13.5	15.5	8.6	11.9	5.0	8.3	1.0	3.7	9.8	13.0	4.5	8.0
#12 (6.8%)	12.6	9.1	7.9	4.2	15.5	13.3	12.2	8.6	8.5	4.7	3.9	1.0	12.8	9.3	8.2	4.2
#13 (4.5%)	8.4	11.9	13.3	15.4	4.1	7.6	9.1	12.6	4.3	8.0	9.7	12.9	1.0	3.9	5.1	8.7
#14 (5.5%)	12.1	8.4	15.5	13.2	8.0	4.1	12.6	9.1	8.0	4.4	12.9	9.3	4.0	1.0	8.6	5.0
#15 (5.9%)	13.4	15.5	8.7	11.9	9.1	12.4	4.1	7.6	9.5	12.9	4.4	8.1	5.0	8.4	1.0	3.8
#16 (6.6%)	15.5	13.3	12.1	8.4	12.7	9.2	7.6	4.0	12.8	9.7	7.9	4.3	8.7	5.0	3.8	1.0

Tabelle B.2: Rangfolge der Knoten (abhängig vom schnellsten Knoten) beim *Real Time* System mit einem 100-er *Compute*-Phase

Literaturverzeichnis

[1] P. Beckman and Co. ZeptoOS: The Small Linux for Big Computers.

[2] P. Beckman, K. Iskra, K. Yoshii, and S. Coghlan. The Influence of Operating Systems on the Performance of Collective Operations at Extreme Scale. In *CLUSTER*, 2006.

[3] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[4] P. De, R. Kothari, and V. Mann. Identifying Sources of Operating System Jitter Through Fine-Grained Kernel Instrumentation. In *Proceedings of the 2007 IEEE International Conference on Cluster Computing (Cluster 2007)*, Austin, Texas, USA, September 17-20 2007. IEEE Computer Society. CD-ROM/Abstracts Proceedings.

[5] D. Deeths and G. Brunette. Using NTP to Control and Synchronize System Clocks - Part I: Introduction to NTP. *Sun Blue Prints*, 2001.

[6] D. Deeths and G. Brunette. Using NTP to Control and Synchronize System Clocks - Part II: Basic NTP Administration and Architecture. *Sun Blue Prints*, 2001.

[7] N. Doss and A. Skjellum. MPICH2 Model MPI Implementation – Reference Manual. Technical report, Argonne National Laboratory, 2003.

[8] J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 147–163, New York, NY, USA, 2002. ACM.

[9] Y. Etsion, T. Ben-Nun, D. Tsafrir, and P. D. Feitelson. KLogger - A Linux Kernel Logging Framework , 2007. [Online, letztes Update Oktober 2007].

[10] Y. Etsion and D. Feitelson. Time Stamp Counters Library - Measurements with Nano Seconds Resolution, 2000.

[11] Y. Etsion, D. Tsafrir, S. Kirkpatrick, and D. G. Feitelson. Fine grained kernel logging with KLogger: experience and insights. *SIGOPS Oper. Syst. Rev.*, 41(3):259–272, 2007.

[12] J. Q. Eva-Katharina Kunst. Kernel- und treiberprogrammierung mit dem kernel 2.6 - Folge 34. *Linux Magazin*, page 3 pages, 2007.

[13] M. P. I. Forum. MPI: A Message-Passing Interface Standard, 2008.

[14] R. Garg and P. De. Impact of Noise on Scaling of Collectives: An Empirical Evaluation. In Y. Robert, M. Parashar, R. Badrinath, and V. K. Prasanna, editors, *HiPC*, volume 4297 of *Lecture Notes in Computer Science*, pages 460–471. Springer, 2006.

[15] R. Gentleman and R. Ihaka. The R Project for Statistical Computing, 2008.

[16] C. Inc. The Cray XD1 High Performance Computer: Closing the Gap Between Peak and Achievable Performance in High Performance Computing. *Cray White Paper*, 2004.

[17] Intel. Intel MPI Benchmarks 3.2, 2008.

[18] H. B. (Intel). private communication, August, 2008.

[19] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 37–37, New York, NY, USA, 2001. ACM.

[20] A. Kumar. Multiprocessing with the Complete Fair Scheduler. Technical report, IBM, 2008.

[21] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. Top 500 Supercumputer Sites, 2008.

[22] A. Nataraj. Kernel Tuning and Analysis Utilities (KTAU), 2008. [Online].

[23] A. Nataraj, A. Morris, A. D. Malony, M. Sottile, and P. Beckman. The Ghost in the Machine: Observing the Effects of Kernel Operation on Parallel Application Performance. In *SuperComputing (SC07)*, November 2007.

[24] F. Petrini, D. J. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 55, Washington, DC, USA, 2003. IEEE Computer Society.

[25] P. D. W. F. T. P. RESH). Rechnerbündel als Plattform für Hochleistungsrechnen und Hochleistungsdatenbanken (RESH), 2008. [Online; letzte Änderung Mai 2004].

[26] J. Reuter and W. F. Tichy. Logging kernel events on clusters. *Future Gener. Comput. Syst.*, 22(3):313–323, 2006.

[27] R. Segewickh. *Algorithmen in C.* Addison-Wesley, 1992.

[28] E. Shmueli, G. Almasi, J. Brunheroto, J. Castanos, G. Dozsa, S. Kumar, and D. Lieber. Evaluating the Effect of Replacing CNK with Linux on the Compute-Nodes of Blue Gene/L. In *ICS '08: Proceedings of the 22th annual international conference on Supercomputing*, pages 165–174, Island of Kos, GR, Greece, 2008. ACM.

[29] B. W. Silverman. *Density estimation: for statistics and data analysis*. Chapman and Hall, London, 1986.

[30] M. J. Sottile and R. Minnich. Analysis of microbenchmarks for performance tuning of clusters. In *CLUSTER*, pages 371–377, 2004.

[31] P. Terry, A. Shan, and P. Huttunen. Improving application performance on HPC systems with process synchronization. *Linux J.*, 2004(127):3, 2004.

[32] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick. System noise, OS clock ticks, and fine-grained parallel applications. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312, New York, NY, USA, 2005. ACM.

[33] University of Oregon, Los Alamos National Laboratory, and Research Centre Jülich, JSC, Germany. Tuning and Analysis Utilities (TAU), 2005. [Online].

[34] D. Wallace. Compute Node Linux: New Frontiers in Compute Node Operating Systems, 2007.

[35] D. Wallace. Compute Node Linux: Overview, progress to date, and roadmap. In *CUG 2007 New Frontiers: 49th Cray User Group meeting*, Seattle, Washington, USA, 2007. Cray Inc.

[36] Wikipedia. Computercluster — Wikipedia, Die freie Enzyklopädie, 2008. [Online; letzte Änderung 21. November 2008].

[37] Wikipedia. LINPACK — Wikipedia, Die freie Enzyklopädie, 2008. [Online; letzte Änderung 9. Oktober 2008].

[38] Wikipedia. Page fault — Wikipedia, The Free Encyclopedia, 2008. [Online; last modified on 27 December 2008].

[39] Wikipedia. Simultaneous Multithreading — Wikipedia, Die freie Enzyklopädie, 2008. [Online; letzte Änderung 4. November 2008].

[40] P. Wurmsdobler. Real Time Linux Foundation, Inc., 2007. [Online].

Jül-4302
Juli 2009
ISSN 0944-2952