Einflüsse des PCI–Busses auf das Laufzeitverhalten von Realzeitsoftware

The Impact of the PCI Local Bus on the Timing Behaviour of Real–Time Software Technical Report

Jürgen Stohr, Alexander von Bülow, Matthias Goebl

Die Einflüsse des PCI–Busses auf die maximale Laufzeit von Software werden in den heute verwendeten Realzeitsystemen auf PC–Basis nicht berücksichtigt. Dieser Beitrag bestimmt die Größenordnung dieser Beeinflussungen und zeigt Methoden, wie mit diesen in einem Realzeitsystem umgegangen werden kann. Hierzu wird eine Softwarearchitektur für Multiprozessorsysteme vorgestellt die es erlaubt, Standard– und Realzeitsoftware im Parallelbetrieb auszuführen. Weiterhin wird eine Messmethodik zur Bestimmung maximaler Laufzeiten von Software beschrieben, mit deren Hilfe der Einfluss des PCI–Busses auf das Laufzeitverhalten eines Realzeitsystems untersucht wird.

The impact of the PCI Local Bus on the worst case execution time of software is ignored in nowadays real–time systems. This article determines the dimension of these impacts and presents methods to deal with them in real–time systems. For this a software architecture for multiprocessor systems based on PC hardware is presented. It enables the use of standard applications and real–time tasks in parallel. Further on a method to determine the worst case execution time of software is described which is used to evaluate the influence of the PCI Local Bus on real–time software.

1 Einleitung

PC–Standardkomponenten erfahren eine stetige und ungleich höhere Steigerung ihrer Leistungsfähigkeit im Vergleich zu Speziallösungen für Realzeitsysteme. Da sie im Regelfall preislich deutlich attraktiver sind, wird deren Einsatz in harten Realzeitsystemen immer interessanter. Die Kompatibilität der PC–Komponenten untereinander gewährleistet, dass vorhandene Software meist mit nur geringem Aufwand auf modernerer Hardware ausgeführt werden kann.

Ein Problem beim Einsatz von PC–Komponenten für harte Realzeitaufgaben ist, dass diese nicht für die Verwendung in Realzeitsystemen entworfen werden. So ist das Optimierungsziel bei der Entwicklung von Systemen auf Basis von Standardkomponenten eine möglichst hohe, *durchschnittliche* Rechenleistung zu erzielen. Die Minimierung einzelner, sporadisch auftretender maximaler Laufzeiten bleibt unberücksichtigt. Gerade diese maximalen Laufzeiten (WCET: *worst case execution time*) sind es jedoch, die in Realzeitsystemen eine zentrale Rolle spielen, wobei die Charakteristik der WCETs maßgeblich durch die Architektur des Systems bestimmt ist.

1.1 PC–Architektur

In aktuellen PC–Standardsystemen kommen Prozessoren zum Einsatz, die der Intel IA–32 Architektur angehören. Die wichtigsten Vertreter dieser Prozessorfamilie sind der Intel Celeron, Pentium III, Pentium IV sowie die Xeon Prozessoren. Von AMD kommen noch die Prozessorfamilien Duron und Athlon hinzu. Die hohe Verarbeitungsgeschwindigkeit dieser Prozessoren beruht vor allem auf dem Einsatz von Caches und TLBs (*Translation Lookaside Buffers*), der Verwendung von mehreren, parallel arbeitenden Pipelines und der Möglichkeit, Befehle in einer anderen als vom Programmfluss vorgesehenen Reihenfolge auszuführen (*out–of–order execution*). Unterstützt werden diese Mechanismen durch die Sprungvorhersage (*branch prediction*), die eine große Rolle vor allem für Prozessoren mit langen Pipelines spielt.

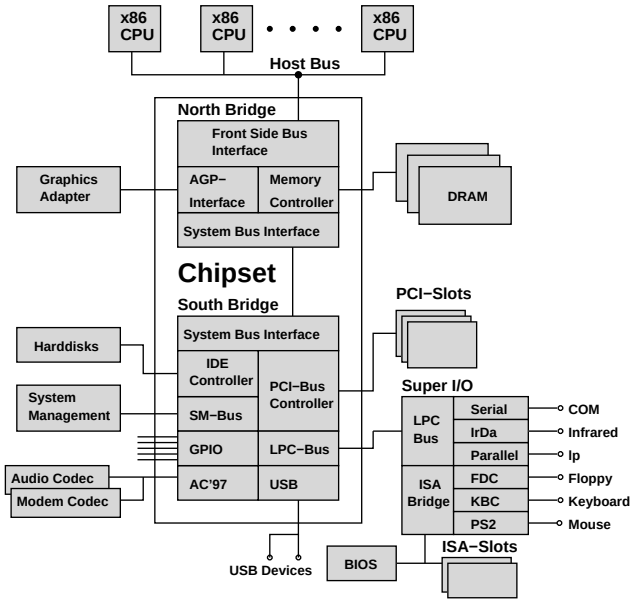

Bild 1: Chipsatz und angeschlossene Hardwarekomponenten

Multiprozessorsysteme auf Basis der Intel SMP Architektur (*symmetric multi–processing*, [10]) sind symmetrische Systeme. Dies bedeutet, dass in einem SMP System nur Prozessoren gleichen Typs vorhanden und alle CPUs gleichberechtigt sind. Jeder Prozessor kann mit jedem anderen kommunizieren. Zwei Aspekte stehen hierbei besonders im Vordergrund:

Symmetrie bzgl. des Speichers: Alle Prozessoren im System benutzen den Speicher gemeinsam und adressieren diesen mit den gleichen physikalischen Adressen.

Symmetrie bzgl. Ein–/Ausgabe: Alle CPUs können auf die gleichen Portadressen zugreifen und Interrupts gleichberechtigt und gleichzeitig von jeder möglichen Interruptquelle erhalten.

Für das Laufzeitverhalten eines modernen PC–Systems ist nicht nur der Prozessor verantwortlich, die ihn umgebende Hardware spielt ebenfalls eine bedeutsame Rolle. Wesentlicher Bestandteil hierbei ist der Chipsatz, der für die Anbindung weiterer Hardware an den Prozessor zuständig ist. Bild 1 zeigt den typischen Aufbau des Chipsatzes und der daran angeschlossenen Hardware.

1.2 Motivation

Grundlage für die Untersuchung der Einflüsse des PCI–Busses auf das Laufzeitverhalten von Realzeitsoftware sind Multiprozessorsysteme auf PC–Basis. Für den Realzeiteinsatz bieten diese die Möglichkeit, Standard– und Realzeittasks auf unterschiedlichen CPUs auszuführen: Ein Prozessor kann exklusiv für Standardaufgaben verwendet werden, alle weiteren Prozessoren stehen für den Realzeitbetrieb zur Verfügung. So wird vermieden, dass die Standardtasks sich auf das Laufzeitverhalten der Realzeittasks bezüglich der Prozessorarchitektur auswirken. Es kommt zu keinen Verdrängungen in den Caches und TLBs der Realzeit–Prozessoren, die durch Standardtasks verursacht werden. Bei geeigneter Anordnung von Code und Daten der Realzeittasks im Hauptspeicher können zudem mögliche Verdrängungen vorherbestimmt oder ganz vermieden und somit Teile des Realzeitsystems im Cache einer CPU fixiert werden.

Nachteilig wirkt sich jedoch insbesondere bei Multiprozessorsystemen der Zugriff auf außerhalb der Prozessoren liegende Hardware auf das Laufzeitverhalten der Realzeitsoftware aus: Über den PCI–Bus werden in den heute verwendeten PC–Systemen Komponenten, wie beispielsweise Prozess–Signaladapter, Netzwerkkarten und Massenspeicher an den Chipsatz und damit an die Prozessoren angebunden. Beim Zugriff kommt es hierbei zu Verzögerungen, die teilweise nicht unerheblich sind [9].

Die Einflüsse des PCI–Busses in PC–Systemen auf das Laufzeitverhalten von Realzeitsoftware wurden noch von keiner Forschungsgruppe behandelt. Deshalb wird in dieser Veröffentlichung die Größenordnung dieser Beeinflussungen untersucht. Es werden Methoden vorgeschlagen, wie mit diesen in einem Realzeitsystem umgegangen werden kann. Zunächst wird in Abschnitt 3 eine Softwarearchitektur für Multiprozessorsysteme vorgestellt, die es erlaubt, Standard– und Realzeitapplikationen echt parallel auszuführen und zudem die gegenseitigen Beeinflussungen beim Zugriff auf PCI–Komponenten minimiert. Um Aussagen bezüglich der Größenordnung der auftretenden Latenzzeiten zu ermöglichen, wird in Abschnitt 4 die verwendete Messmethodik beschrieben. Mit der prinzipiellen Funktionsweise des PCI–Busses befasst sich anschließend Abschnitt 5. In Abschnitt 6 werden die Messergebnisse zur Ermittlung der durch den PCI–Bus hervorgerufenen Latenzzeiten vorgestellt und diskutiert. Eine Zusammenfassung der Ergebnisse und der Ausblick auf weitere Arbeiten erfolgt abschließend in Kapitel 7.

2 Verwandte Arbeiten

Die hier vorgestellten Ergebnisse basieren auf Arbeiten, die aus der im Rahmen des DFG–Schwerpunktprogramms *Rapid Prototyping für integrierte Steuerungssysteme mit harten Zeitbedingungen* an unserem Lehrstuhl entwickelten Architektur REAR [6] [16] gewonnen wurden.

Im Projekt RTCPU [9] wird ein Dual–Prozessor–PC für Realzeitaufgaben eingesetzt. Dabei wird auf dem einen Prozessor das Standard–Betriebssystem Linux ausgeführt; auf der anderen CPU wird eine Realzeitapplikation zur Ausführung gebracht. Bei dieser Architektur zeigte sich, dass sich die einzelnen Komponenten eines Dual–PCs in nicht zu vernachlässigender Weise beeinflussen.

Der PCI–Bus ist in heutigen PC–Systemen der Standardbus, der die meisten Hardwarekomponenten mit

dem Prozessor und dem Hauptspeicher verbindet. Für das Verständis des Laufzeitverhaltens von PC–Systemen ist es daher von zentraler Bedeutung, die Arbeitsweise und das Zusammenspiel des PCI–Busses mit den Peripheriekomponenten zu analysieren. Leider gibt es keine Veröffentlichungen, die sich mit dem Einsatz des PCI–Busses für Realzeitsysteme beschäftigen. Fundierte Betrachtungen zur prinzipiellen Eignung von PC–Hardware für Realzeitsysteme finden sich in [4] oder [20], die Realzeiteigenschaften moderner Standard–Betriebssysteme werden in [7] untersucht.

Ein zentraler Punkt bei der Konstruktion und Validierung von Realzeitsystemen ist die Bestimmung der WCET der eingesetzten Software. Verschiedene Komponenten haben dabei einen Einfluss auf die WCET: Die Prozessorarchitektur, die eingesetzten Bussysteme, welche einzelne Hardwarekomponenten des Realzeitsystems miteinander verbinden und natürlich die Eigenschaften der Komponenten selbst. Viele Projekte beschäftigen sich mit der Bestimmung der WCET, wobei im Wesentlichen zwei verschiedene Ansätze verfolgt werden: Die exakte Berechnung der WCET und die Abschätzung der WCET durch Laufzeitmessungen auf dem Zielsystem.

Die Berechnung von Laufzeiten erfolgt mit Hilfe von statischen Sourcecode–Analysen, Zusatzinformationen über das dynamische Programmverhalten in Form von Annotationen [12] [17] und Modellen der zugrundeliegenden Hardware [11] [18]. Die Sourcecode–Analyse der Software liefert alle denkbaren Ausführungspfade und ermittelt aus diesen den Pfad, der die WCET des gesamten Programms liefert. Voraussetzung dafür ist die genaue Kenntnis über das Laufzeitverhalten der Hardware, wie beispielsweise die Zeiten, die der Prozessor zur Ausführung der jeweiligen Befehle im Worst Case benötigt. Darüber hinaus spielen die Einflüsse weiterer Komponenten des Realzeitsystems, insbesondere der Bussysteme, eine entscheidende Rolle für das Laufzeitverhalten, da sie die effektive Arbeitsgeschwindigkeit des Prozessors erheblich beeinflussen können. Eine Möglichkeit, die Eigenschaften der Hardware zu berücksichtigen, ist diese zu modellieren und aus dem Modell heraus die Worst Case Situation zu ermitteln. Dafür ist aber eine hinreichend genaue Kenntnis über die Funktionsweise der Hardware nötig, was bei PC–Komponenten oft nicht der Fall ist.

Eine andere Möglichkeit ist die Ermittlung von Ausführungszeiten durch Messungen auf dem Zielsystem. Um Laufzeiten im Worst Case messen zu können, muss man diesen vor der Messung erzwingen. Für Prozessoren der IA–32 Architektur, wie sie in heutigen PC–Systemen anzutreffen sind, werden in [13], [14] und [15] Verfahren und Methoden vorgestellt, Laufzeitabschätzungen vorzunehmen, in [5] wird ein Ansatz mit statistischen Methoden verfolgt. Viele Arbeitsgruppen beschäftigen sich mit den Einflüssen einzelner Komponenten wie beispielsweise Caches [11] [19] [21] oder Pipelines [18] von Mikroprozessoren, wobei dabei oft Architekturmodelle verwendet werden, die der Komplexität der modernen IA–32 Prozessoren nicht gerecht werden.

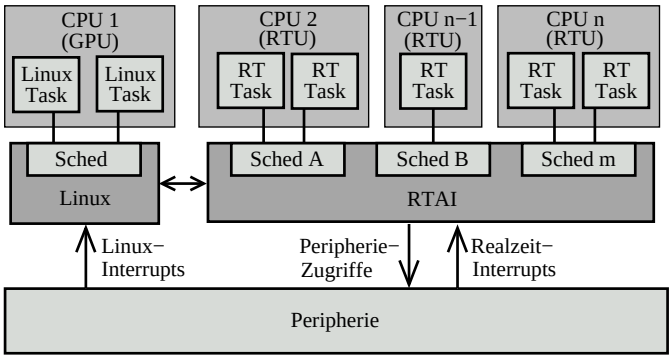

Bild 2: Softwarearchitektur

3 Softwarearchitektur

Auf dem Realzeitsystem sollen sowohl harte Realzeittasks als auch Standardapplikationen parallel zur Ausführung gebracht werden können. Dies bietet die Möglichkeit, das Softwaredesign in einen Realzeit– und einen Nicht–Realzeitanteil aufzuteilen, sodass nur die zeitkritischen Aufgaben unter Echtzeitbedingungen ausgeführt werden müssen. Andere Aufgaben, wie beispielsweise die Visualisierung von Daten, können parallel dazu vom Nicht–Realzeitteil bearbeitet werden. Der Vorteil hiervon ist, dass man für Standardaufgaben auf Standardsoftware zurückgreifen kann und die Möglichkeit hat, Realzeitsoftware klein und kompakt zu realisieren.

Als Realzeitbetriebssystem kommt ein modifiziertes RTAI [3] zum Einsatz, das verwandt zu dem von den FSMLabs entwickelten RT–Linux [22] ist. RTAI ist eine Realzeiterweiterung für das Betriebssystem Linux, welche bereits die Möglichkeit der Ausführung harter Realzeittasks parallel zu Standardanwendungen bietet. Dabei wird ein eigener RT–Kernel implementiert, welcher die Realzeittasks ausführt und den Linux–Kernel und dessen Tasks als die niederpriorste Task des Realzeitsystems betrachtet. Problematiken des Speichermanagements, sowie mögliche Blockierungen beim Zugriff auf gemeinsam genutzte Hardwarekomponenten werden sowohl von RT–Linux als auch von RTAI nicht berücksichtigt.

Deshalb wurde auf Basis von RTAI die in Bild 2 dargestellte Softwarearchitektur entwickelt: Für die Realzeituntersuchungen findet ein Multiprozessorsystem Verwendung. Linux und seine Standardanwendungen werden exklusiv von der GPU (*General Purpose Unit*) ausgeführt. Alle weiteren Prozessoren des Multiprozessorsystems — die RTUs (*Real–Time Units*) — stehen ausschließlich für die Realzeitapplikationen zur Verfügung. Interrupts, die von Realzeittasks bearbeitet werden müssen, werden direkt an die entsprechenden RTUs geleitet; Systeminterrupts, wie beispielsweise die der Ta-

statur und der Maus, die nur für die GPU relevant sind, werden ausschließlich an diese geliefert. Damit wird eine strikte Trennung zwischen Standard– und Realzeitapplikationen erreicht; die RTUs beschäftigen sich ausschließlich mit Realzeitaufgaben und müssen nicht — wie im Falle eines Uni–Prozessorsystems — die Tasks des Standardbetriebssystems sowie dessen Interrupts zusätzlich bearbeiten. Die Reaktionszeit der Realzeittasks ist somit deutlich geringer, da auf den RTUs keine Kontextwechsel durchgeführt werden müssen.

Für die GPU und die RTUs ist — im Gegensatz zu RTAI — jeweils ein eigener physikalischer Bereich des Hauptspeichers vorgesehen. Damit werden gegenseitige Beeinflussungen, wie Verdrängungen in den Caches und den TLBs, vermieden. Durch geeignete Anordnung von Code und Daten der Realzeittasks im Hauptspeicher können mögliche Verdrängungen in den Caches vorherbestimmt und Teile des Realzeitcodes in den Caches der RTUs fixiert werden.

Falls mehrere Prozessoren gleichzeitig auf die Peripherie zugreifen — dies sind Zugriffe auf bzw. über an den PCI–Bus angeschlossene Geräte — kann es dabei zu Latenzzeiten kommen, da aufgrund der SMP–Architektur nicht festgelegt ist, welche CPU zuerst den entsprechenden Zugriff ausführen darf. Ohne geeignete Maßnahmen bedeutet dies für Realzeitsysteme, dass die Zugriffszeit auf ein PCI–Gerät Schwankungen unterworfen ist, die durch die Multi–Master–Fähigkeit des PCI–Busses noch verstärkt werden (siehe hierzu auch Abschnitt 5). Deshalb sorgt die hier vorgestellte Architektur dafür, dass für die Dauer der Peripheriezugriffe einer Realzeittask die GPU keine derartigen Zugriffe ausführt. Damit wird erreicht, dass bei einem PCI–Zugriff seitens einer RTU keine Beeinflussungen durch die GPU auftreten. Falls mehrere RTUs vorhanden sind, dürfen diese nicht gleichzeitig auf die Hardware zugreifen. Dies kann mit geeigneten Synchronisierungsmaßnahmen wie beispielsweise Semaphore erreicht werden.

4 Messung von Ausführungszeiten

Um gemessene Ausführungszeiten miteinander vergleichen zu können — im vorliegenden Fall sollen die Verzögerungen durch PCI–Zugriffe abgeschätzt werden — benötigt man eine Messmethodik, die vor jeder Messung das System in einen definierten Ausgangszustand versetzt. Nur so können Laufzeitschwankungen aufgrund unterschiedlicher, undefinierter Ausgangszustände zu Messbeginn vermieden werden. Hier ist nur die WCET von Interesse, daher muss für jede Messung sichergestellt werden, dass alle Komponenten, die einen Einfluss auf die Laufzeit haben, sich so verhalten, dass sie die Ausführungszeiten jeweils um ihr Maximum verlängern. Zu diesen Komponenten gehören unter anderem der Prozessor selbst, der Chipsatz und auch alle Geräte, die über den PCI–Bus angeschlossen sind.

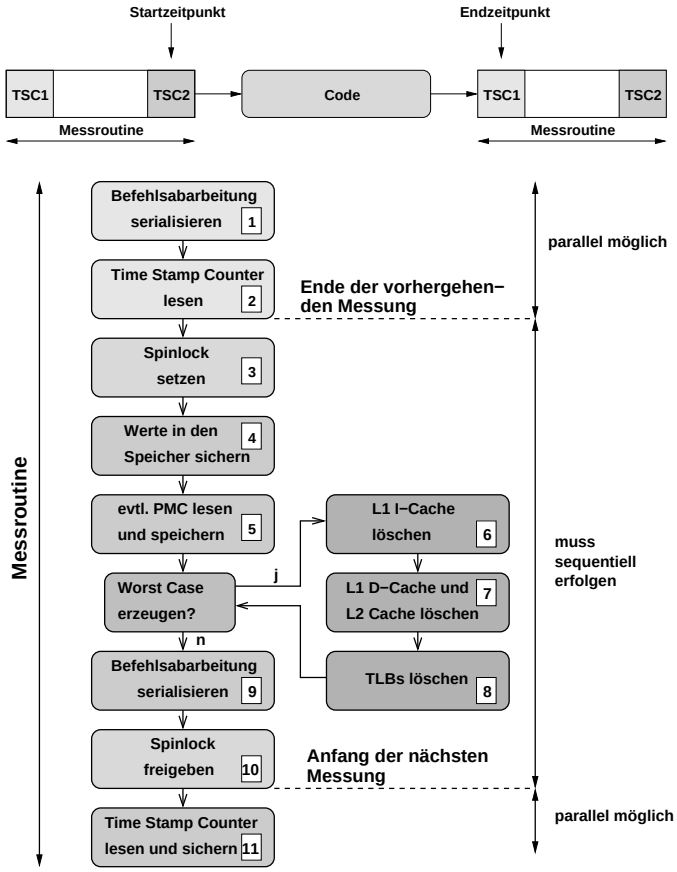

Bild 3: Die Messroutine

Bei der Ermittlung der WCETs sind auch mögliche Abhängigkeiten dieser Komponenten untereinander zu berücksichtigen, um keine Überschätzung der WCET zu bekommen. Die Messmethodik, die zur Messung der Zeiten in Kapitel 6 verwendet wurde, wird im Folgenden erläutert.

Die Vorgehensweise basiert auf Methoden, die bereits in [13] [14] veröffentlicht wurden. Ziel der Messungen dort ist es, den Worst Case bezüglich der Prozessorarchitektur zu erzwingen, um dann die WCET des betrachteten Codestücks abschätzen zu können. Dies ist auch für die Abschätzung der Laufzeitverlängerungen durch PCI–Zugriffe unabdingbar. Die Messroutine wurde im Hinblick auf Multiprozessorfähigkeiten erweitert, so ist es möglich, parallel auf mehreren Prozessoren zu messen und die Messdaten getrennt zu behandeln. Der Overhead der Messroutine wurde minimiert und die Zwischenspeicherung der Daten hat keinen Einfluss auf den Datencache mehr.

4.1 Messroutine

Die Prozessoren mit IA–32 Architektur verfügen über mehrere Mechanismen, welche die Ausführungszeiten von Code auf dem Prozessor teilweise erheblich beeinflussen und für die Messung dieser Ausführungszeiten berücksichtigt werden müssen. An erster Stelle sind hier die Caches zu nennen, welche die Differenz zwischen der

Arbeitsgeschwindigkeit des Prozessors und der Zugriffsgeschwindigkeit des Hauptspeichers ausgleichen sollen. Weiterhin arbeiten diese Prozessoren mit mehreren, auch verschiedenartigen Pipelines parallel und verfügen über sehr leistungsfähige Techniken zur Sprungvorhersage.

Soll die Ausführungszeit eines vorliegenden Codestücks gemessen werden, so müssen vor und nach der Messung spezielle Maßnahmen getroffen werden, die in Bild 3 schematisch dargestellt sind: Zunächst wird die Befehlsabarbeitung serialisiert [1], d. h. alle Befehle, die bereits dekodiert wurden, werden noch ausgeführt, ohne neue Befehle nachzuladen. Anschließend wird ein Zeitstempel mit Hilfe des TSC (*Time Stamp Counter*) genommen [2]. Das TSC–Register wird in jedem Taktzyklus um eins erhöht und ist auf allen moderneren Prozessoren vorhanden. Danach folgt ein Spinlock [3], der verhindert, dass eine andere CPU gleichzeitig den folgenden Code ausführt. Dies ist nötig, damit die Messpunkte, die von mehreren CPUs gleichzeitig genommen werden, korrekt abgespeichert werden [4]. Anhand der lokalen APIC–ID (*Advanced Programmable Interrupt Controller*) kann die Messroutine feststellen, von welchem Prozessor sie gerade ausgeführt wird. Die Spezifikation für SMP–Systeme fordert, dass diese IDs systemweit eindeutig sein müssen. Die Messung selbst wird dadurch nicht beeinträchtigt, da der Zeitstempel, der die vorhergehende Messung beendet, bereits genommen wurde. Danach werden die PMC–Register (*Performance Monitoring Counter*) ausgelesen und neu programmiert [5]. Mit Hilfe dieser Register können verschiedene Ereignisse, die innerhalb des Prozessors auftreten, wie beispielsweise die Anzahl der Cache-Misses, analysiert werden. Anschließend werden Maßnahmen getroffen um das Worst Case Szenario bzgl. Cache und TLBs herzustellen [6] [7] : Der L2–Cache wird mit nutzlosen Daten gefüllt, die zusätzlich verändert werden, um ein Zurückschreiben aller Cachelines vor der Ausführung des auszumessenden Codes in den Speicher zu erzwingen. Die TLBs können durch eine einfache Befehlsabfolge als ungültig markiert werden [8]. Abschließend wird der Befehlsablauf nochmals serialisiert [9] und der Spinlock wieder freigegeben [10] und ein weiterer Zeitstempel genommen, der die nächste Messung startet [11].

4.2 Ablauf einer Messung

Die Messroutine ist in Assembler realisiert und läuft, wie das Realzeitbetriebssystem, auf der höchsten Privilegierungsstufe des Prozessors. Der Messvorgang (vgl. Bild 4) wird von der GPU aus gesteuert. Zu Beginn einer Messreihe wird die Messung konfiguriert (Anzahl der Messungen, Konfiguration der PMCs), der auszumessende Code geladen und die Messung gestartet. Wenn der Zwischenspeicher, in den die Messroutine ihre Daten ablegt, voll ist, signalisiert sie das der Steuersoftware mit Hilfe einer entsprechenden Nachricht und stoppt

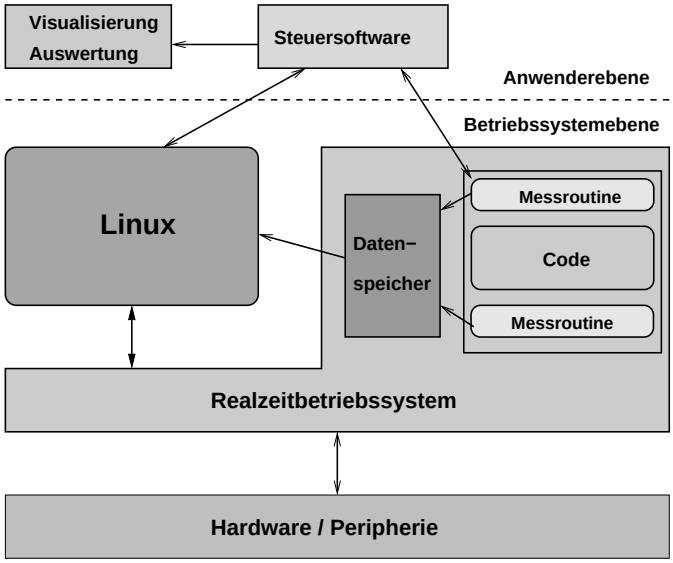

Bild 4: Architektur der Messumgebung

die Aufnahme weiterer Messwerte — das Realzeitsystem bleibt dabei aktiv. Die Messdaten werden aus dem Speicher gelesen und in einer Datei abgespeichert. Anschließend kann mit der Messung fortgefahren werden. Somit ist es möglich, nahezu beliebig viele Messwerte aufzunehmen.

Eine Zeitmessung wird durch das Auslesen des TSCs am Ende der Messroutine (vgl. Bild. 3) gestartet und mit dem Zeitstempel am Anfang der Routine gestoppt. Somit ist die Zeit, die für das Speichern der Daten und die Manipulation der Caches benötigt wird, von der Messung ausgenommen. Auch die Blockierungszeit, die durch den Spinlock bei parallelen Messungen auf einem Mehrprozessorsystem auftreten kann, ist nicht in den gemessenen Zeiten enthalten.

Mit dieser Messanordnung ist es möglich, auf mehreren Prozessoren parallel Ausführungszeiten zu messen. Die Laufzeitverlängerung, welche durch die Routine selbst erzeugt wird, ist sehr gering, da der Overhead der Messroutine nur aus drei Stack–Operationen, einem *call*–Befehl und dem Abspeichern des Zeitstempels des Messstarts besteht. Die Messroutine sorgt dafür, dass dieses Abspeichern ein Daten–Cache Hit ist, damit die gemessene Ausführungszeit nicht durch einen Cache–Miss verlängert wird, der nicht auf den zu messenden Code zurückzuführen ist; das selbe gilt für die TLBs.

5 PCI Local Bus

Der PCI–Bus ist der in PC–Standardsystemen am häufigsten anzutreffende Peripherie–Bus. Seine weite Verbreitung verdankt er maßgeblich seiner standardisierten Konfigurationsschnittstelle und seiner Prozessorunabhängigkeit. Der PCI–Bus wird im allgemeinen über die Host–Bridge an die Prozessoren und den Hauptspeicher angebunden.

5.1 Arbeitsweise

Der PCI–Bus ist ein Multi–Master–Bus. Jedes Gerät kann Bus–Master (*initiator*) werden und selbständig einen Datentransfer zu einem anderen Gerät (*target*) durchführen. Dies geschieht autonom, ohne die Unterstützung des Prozessors. Damit es zu keinen Kollisionen kommt, muss ein Bus–Master vor jedem Zugriff den Bus erfolgreich arbitrieren.

Alle Bus–Zugriffe werden vom Bus–Arbiter koordiniert, der sich in der Regel im Chipsatz befindet. Jedes Gerät besitzt dorthin eine gesonderte Anforderungs– und eine Zuteilungsleitung. Benötigt ein Gerät Zugriff auf den Bus, aktiviert es seine Anforderungsleitung. Der Arbiter entscheidet, welches Gerät als nächstes Zugriff auf den Bus bekommt und aktiviert dessen Zuteilungsleitung. Dies geschieht noch während des vorhergehenden Transfers (*hidden arbitration*). Der nächste Master darf erst aktiv werden, nachdem der vorhergehende den Bus wieder in den Ruhezustand gebracht hat.

Welche Strategie und welche Prioritäten der Arbiter bei der Busvergabe anwendet, ist chipsatzabhängig. Einer der betrachteten Chipsätze, der Intel 440FX, vergibt die Zugriffserlaubnis reihum zwischen den PCI–Geräten und der CPU.

Um den Durchsatzanforderungen gerecht zu werden, besitzt jedes Master–fähige Gerät einen konfigurierbaren *Latency Timer*, der besagt, wieviele Datenphasen es am Stück durchführen darf. Ein Bus–Master, dessen Zuteilungsleitung deaktiviert wurde und der noch nicht die durch den *Latency Timer* spezifizierte Anzahl an Datenphasen durchgeführt hat, darf bis zu deren Erreichen weitere Datenphasen durchführen.

Aus Sicht des PCI–Busses ist insbesondere die Host–Bridge auch ein PCI–Gerät und muss daher vor jedem Zugriff den Bus arbitrieren, um Master zu werden. Die Host–Bridge arbeitet auch als Target für andere PCI–Geräte und ermöglicht diesen den Zugriff auf den Hauptspeicher.

5.2 PCI–Zugriffe

Der PCI–Bus arbeitet mit kombinierten Adress– und Datenleitungen und zusätzlichen Kommando– bzw. Byteauswahlleitungen. Ein PCI–Zugriff besteht aus einer bzw. zwei Adressphasen und einer oder mehreren Datenphasen. In der Adressphase legt der Master die gewünschte Zieladresse und die beabsichtigte Zugriffsart auf den Bus. Alle darauffolgenden Phasen sind Datenphasen; dabei zählt nach jeder Datenphase das Target die Adresse selbständig hoch. Diese Art der Datenübertragung bezeichnet man als *Burst–Transfer*. Lange Bursts minimieren den Adressierungs–Overhead und führen zu einem höheren Datendurchsatz.

Die wichtigsten Zugriffsarten sollen im Folgenden kurz vorgestellt werden.

Speicherzugriffe: Ein Gerät greift auf den in einem anderen Gerät enthaltenen Speicher oder den Hauptspeicher zu.

I/O–Zugriffe: Damit wird auf einzelne Kommando– oder Status–Register von Geräten zugegriffen. Sie werden von der Host–Bridge durch I/O–Port–Zugriffe des Prozessors erzeugt. I/O–Zugriffe können nicht gepuffert werden.

Konfigurationszugriffe: Die von einem PCI–Gerät belegten Ressourcen, wie Speicherbereiche, I/O–Bereiche und Interrupts, werden in dessen Konfigurationsregistern festgelegt.

Speicherzugriffe des Prozessors, deren Adressen außerhalb des Hauptspeichers liegen, und I/O–Zugriffe werden von der Host–Bridge automatisch in die entsprechenden PCI–Zugriffe umgesetzt.

6 Einfluss des PCI–Busses

Für das Zeitverhalten des Realzeitsystems sind die Zugriffszeiten auf den Hauptspeicher und auf PCI–Geräte von Bedeutung. Die Messungen wurden auf den folgenden beiden Dual–Rechnern durchgeführt:

1. Ein Dual–Intel–Pentium II System mit Intel 440FX–Chipsatz. Die CPUs sind mit 233 MHz getaktet und über einen 66 MHz Front-Side Bus angeschlossen. Der Rechner ist mit 192 MB EDO RAM bestückt. Am PCI–Bus befinden sich zwei Intel Ethernet Pro 100–Netzwerkkarten mit einem i82557–Chip, eine S3 86C775 Trio64V2/DX–Grafikkarte und ein Adaptec AIC–7880U SCSI–Festplatten–Controller.
2. Ein Dual–AMD–Athlon MP 1800+ System mit AMD–762/768–Chipsatz, mit 1533 MHz CPU–Takt und einem 133 MHz Front-Side Bus. Der Rechner ist mit 512 MB DDR-R ECC RAM bestückt. Am PCI–Bus befinden sich zwei Intel Ethernet Pro 100–Netzwerkkarten mit einem i82557–Chip, eine NEC–USB und eine FireWire–Karte, sowie eine ASUS V7100 AGP Grafikkarte. Die IDE–Festplatte ist am in die Southbridge AMD–768 integrierten IDE–Controller angeschlossen.

Mit der in Abschnitt 4 vorgestellten Messmethodik wird die Größenordnung möglicher Latenzzeiten aufgrund von PCI–Aktivität bestimmt. Dazu werden die auftretenden Zeiten mit und ohne PCI–Aktivität bei Hauptspeicherzugriffen gemessen. Anschließend werden die auftretenden Latenzzeiten beim Zugriff auf ein PCI–Gerät bestimmt — ebenfalls mit und ohne paralleler PCI–Aktivität. Die dritte Messreihe beschäftigt sich mit der benötigten Zeit zur Deaktivierung der PCI–Master. Die Deaktivierung ist für die Messungen ohne PCI–Aktivität Voraussetzung.

Als Laufzeitumgebung wird die in Abschnitt 3 beschriebene Architektur verwendet. Auf der GPU läuft Linux, auf der RTU werden die für die Messungen notwendigen Realzeitprozesse gestartet.

Linux: Aufgabe der GPU ist es, die Messroutine aus Kapitel 4 zu initialisieren, und RTAI auf die RTU zu laden. Zu Beginn jeder Messung programmiert die GPU die PCI–Geräte, sodass diese autonom PCI–Buslast erzeugen. Anschließend löst die GPU einen Messzyklus aus, indem sie der RTU ein Signal sendet, dass die Messumgebung initialisiert ist. Um nur die Auswirkungen des PCI–Busses zu messen und Beeinflussungen durch die GPU zu vermeiden, wird diese für die Dauer jeder Messung in einer Warteschleife gehalten. Damit werden durch die GPU verursachte Einflüsse auf die Messergebnisse vermieden.

RTAI: Der Realzeitcode der RTU ist in einen Interrupt–Handler eingebettet. Durch die Verwendung eines Interruptgates wird das Interrupt–Enable–Flag gelöscht, sodass keine weiteren Interrupts die Messung beeinflussen können. Der Interrupt–Handler wird durch einen Inter–Prozessor–Interrupt (*IPI*) angestoßen. Um Messungen mit und ohne PCI–Aktivität durchführen zu können, befindet sich zu Beginn des Interrupt–Handlers eine Routine, die alle PCI–Geräte vollständig deaktivieren kann.

6.1 Hauptspeicherzugriffe

Die Zugriffszeiten des Prozessors auf den Hauptspeicher werden mit und ohne gleichzeitigem PCI–Betrieb gemessen. Ein Realzeitprozess lässt den Prozessor 2048 Lese– und Schreibzugriffe (32 Bit) auf den Hauptspeicher durchführen. Mit der Wahl eines Adress–Abstandes von 64 Bytes liegen diese Zugriffe sowohl für den Athlon (64 Bytes pro Cacheline) als auch für den Pentium II (32 Bytes pro Cacheline) auf immer neuen Cachelines. Durch die in Kapitel 4 beschriebene Messroutine ist sichergestellt, dass alle Cachelines ungesicherte Daten enthalten. Vor einem Lesezugriff muss die jeweils verdrängte Cacheline zurückgeschrieben werden. Ein Zugriff löst somit ein Schreiben und ein Lesen einer kompletten Cacheline aus. Dies entspricht dem gewünschten Worst Case Szenario. Gleichzeitig werden Bus–Master–fähige PCI–Geräte so programmiert, dass sie eigenständig Daten vom und zum Hauptspeicher übertragen.

Bild 5 und Bild 6 zeigen die Messergebnisse mit und ohne PCI–Aktivität bei Hauptspeicherzugriffen für den Pentium II Rechner. Mit PCI–Aktivität schwanken die Laufzeiten um 72%, ohne PCI–Aktivität beträgt die Verlängerung der Laufzeit gegenüber dem besten Fall nur noch 2%.

Selbige Messung wurde auch mit dem Athlon–System durchgeführt. Hier war der Einfluss durch PCI messbar, die Auswirkungen sind jedoch aufgrund der moderneren Systemarchitektur weniger gravierend als beim Pentium II Rechner: mit PCI–Aktivität betrug die Verlängerung der Laufzeit gegenüber dem besten Fall 15%, ohne PCI–Aktivität 7%. In absoluten Zahlen ist ohne PCI–Aktivität die Differenz der Laufzeiten beim Athlon–

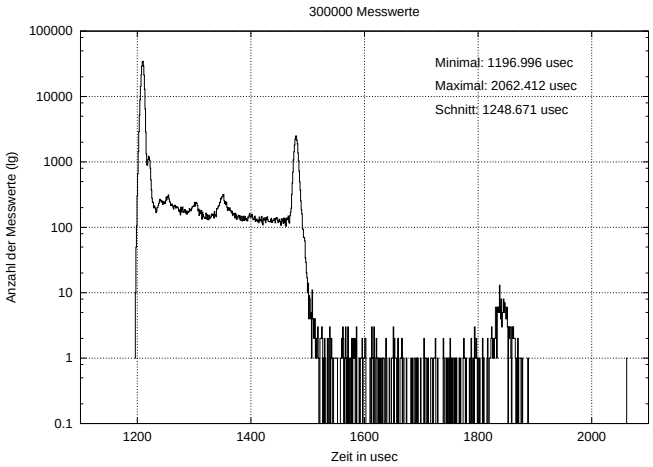

Bild 5: Speicherzugriffe mit PCI–Aktivität, PII Rechner

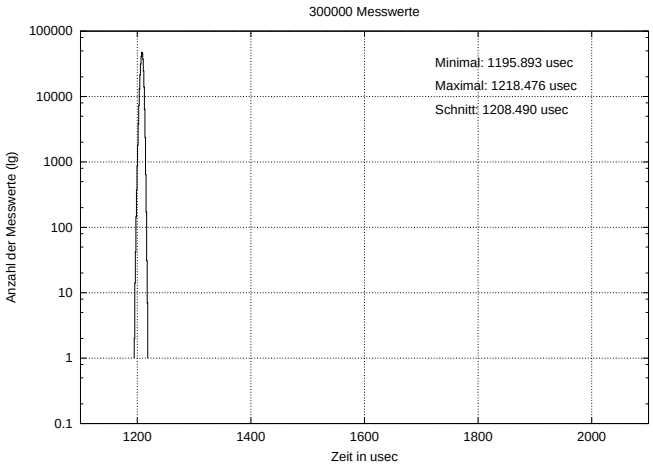

Bild 6: Speicherzugriffe ohne PCI–Aktivität, PII Rechner

System im Vergleich zum Pentium II Rechner geringer.

Die hier auftretenden Verzögerungen sind insbesondere auf die Konkurrenzsituation an der Host–Bridge zurückzuführen. Diese muss zwischen den Zugriffen des Prozessors auch die Speicherzugriffe der PCI–Geräte bedienen. Somit kommt es zu Latenzen beim Zugriff der RTU auf den Hauptspeicher, da die RTU nun bis zur Beendigung des Speichertansfers des jeweiligen PCI–Gerätes warten muss.

Die durchgeführten Messungen zeigen, dass sich die Beeinflussungen aufgrund paralleler PCI–Aktivität bei der Bestimmung der WCET für Realzeitsoftware nicht vernachlässigen lassen. Obwohl die Messwerte jeweils die Laufzeit von 2048 Speicherzugriffen widergeben, zeigt sich eine sehr starke Streuung. Dies bedeutet, dass die durch den PCI–Bus hervorgerufenen Störungen inhomogen sind und sich in nicht deterministischer Weise auf das Laufzeitverhalten von Software auswirken.

Für die Umsetzung in Realzeitsystemen bedeutet dies, dass für zeitkritische Tasks parallele PCI–Aktivität in größerem Umfang vermieden werden sollte. Eine an-

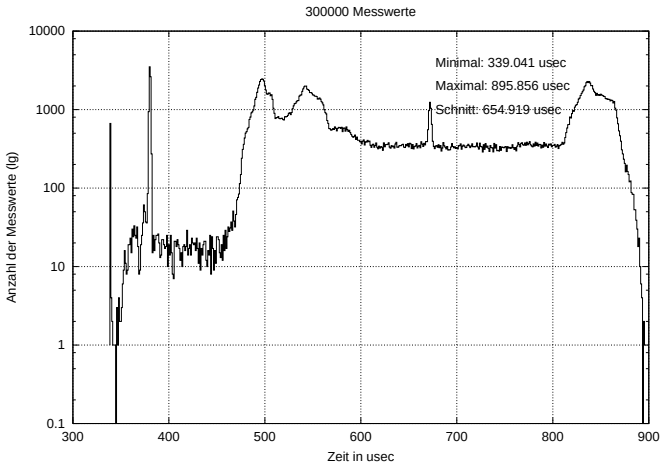

Bild 7: PCI–Zugriffe mit PCI–Aktivität, Athlon System

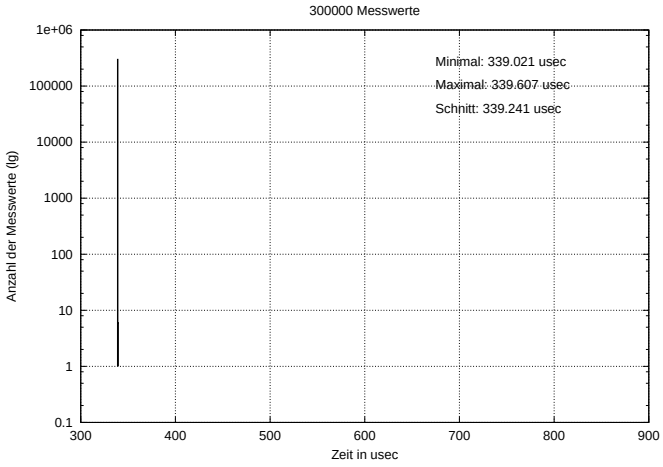

Bild 8: PCI–Zugriffe ohne PCI–Aktivität, Athlon System

dere Möglichkeit ist, die Tasks vollständig im Cache des jeweiligen Prozessors zu halten, sodass diese keine Hauptspeicherzugriffe mehr durchführen müssen. Abhilfe könnte auch eine NUMA–Architektur bieten, da hier die Prozessoren jeweils auf ihrem eigenen physikalischen Speicherbereich arbeiten können.

6.2 PCI–Zugriffe

In einer weiteren Messreihe wird die Zugriffszeit des Prozessors auf ein PCI–Gerät untersucht, ebenfalls mit und ohne gleichzeitigen Betrieb anderer PCI–Geräte. Dazu wird über den PCI–Bus auf den Speicherbereich einer Netzwerkkarte zugegriffen, indem dort eine Speicheradresse 1024 mal gelesen wird. Da beim Zugriff auf die Netzwerkkarte immer die gleiche Adresse verwendet wird, werden Bursttransfers vermieden. Damit der Prozessor jeden Speicherzugriff erneut an die Host–Bridge weitergibt, wird der Speicherbereich der Netzwerkkarte in den Memory–Type–Range–Registern des Prozessors als „uncachable“ eingetragen.

Das Ergebnis der Messung für die 1024 PCI–Zugriffe auf die Netzwerkkarte sind in Bild 7 und Bild 8 für das Athlon System dargestellt. Mit paralleler PCI–Aktivität beträgt die Schwankung der Laufzeit 164%, ohne PCI–Aktivität sind kaum mehr messbare Laufzeitschwankungen vorhanden. Ähnliche Ergebnisse erhält man für das Pentium II System: Mit PCI–Aktivität beträgt die Laufzeitverlängerung maximal 53%, ohne PCI–Aktivität 5%.

Ursache für die auftretenden Laufzeitschwankungen ist, dass für jeden Zugriff des Prozessors auf die Netzwerkkarte die Host–Bridge den PCI–Bus arbitrieren muss. Wenn ein anderes Gerät gerade einen PCI–Transfer durchführt, muss sie warten, bis dieses Gerät den Bus wieder freigibt.

Falls von einer Realzeittask auf ein PCI–Gerät zugegriffen werden soll, kann es sinnvoll sein, vor den Zugriffen andere PCI–Geräte zu deaktivieren. So kommt es bei jedem Zugriff mit paralleler PCI–Aktivität zu Laufzeitschwankungen; wobei eine Realzeittask in Abhängigkeit der Anzahl ihrer PCI–Zugriffe ebenfalls diesen Schwankungen unterworfen ist. Hierbei ist allerdings zu berücksichtigen, dass die Deaktivierung aller PCI–Geräte auch eine gewisse Zeit beansprucht. Dies wird im folgenden Abschnitt gezeigt.

6.3 Deaktivierung der PCI–Master

In den vorgestellten Messungen wurden die PCI–Geräte deaktiviert, um so Einflüsse durch parallele PCI–Aktivität zu verhindern. Bild 9 zeigt die Ausführungszeit von acht PCI–Konfigurationszugriffen, um die im Pentium II Rechner enthaltenen PCI–Geräte abzuschalten. Es wurden Zeiten zwischen 32μsec und 67μsec gemessen. Die Host–Bridge muss hier vor jedem Konfigurationszugriff warten, bis der PCI–Bus frei ist. Nachdem alle Geräte abgeschaltet wurden gibt es auf dem PCI–Bus keine Aktivität mehr.

Das Athlon–System benötigt mit zwölf PCI–Geräten 31μsec bis 119μsec und im Mittel 39μsec zur Deaktivierung aller möglichen PCI–Master.

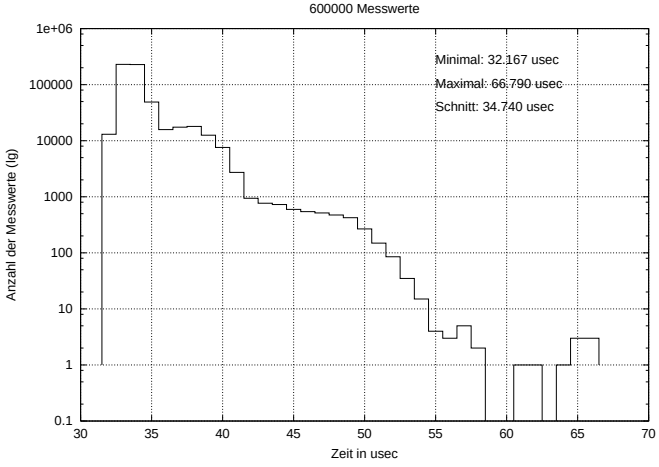

Bild 9: Zeit des PCI–Master–Deaktivierens, PII Rechner

7 Zusammenfassung und Ausblick

In diesem Artikel wurde eine Softwarearchitektur für Realzeitsysteme auf Multiprozessorbasis für PC–Standard–Hardware vorgestellt, die die Grundlage der Untersuchung der Einflüsse des PCI–Busses auf das Laufzeitverhalten von Realzeitsoftware ist. Diese Architektur basiert auf Linux mit der Realzeiterweiterung RTAI. Ein Prozessor (*GPU*) wird exklusiv für Linux und seine Applikationen reserviert, alle weiteren Prozessoren (*RTUs*) werden ausschließlich für Realzeitapplikationen verwendet. Die Interrupts werden nur an die für deren Bearbeitung vorgesehenen Prozessoren geleitet. Zugriffe auf die Hardware werden von dem Realzeitbetriebssystem kontrolliert.

Um Ausführungszeiten von Software für diese Architektur im Worst Case auf dedizierter Hardware zu bestimmen, wurde eine Methodik zur Messung der Laufzeiten vorgestellt. Hierbei wird der Prozessor vor jeder Messung in einen Zustand gebracht, der zu maximalen Laufzeiten von Software bezüglich der Prozessorarchitektur führt. Mit diesem Verfahren wird sichergestellt, dass der Einfluss der Prozessorarchitektur auf jede Messung identisch ist. Damit werden Einflüsse durch die Prozessorarchitektur auf die Messergebnisse vermieden — es werden ausschließlich die durch den PCI–Bus hervorgerufenen Laufzeitschwankungen bestimmt.

Basierend auf dieser Softwarearchitektur in Verbindung mit dem Messverfahren wurde der Einfluss des PCI–Busses auf die Laufzeiten von Realzeitsoftware untersucht. Dabei wurden Hauptspeicherzugriffe seitens einer RTU und Zugriffe auf ein PCI–Gerät betrachtet. Es zeigen sich deutliche Unterschiede bei den Messungen in Abhängigkeit der vorhandenen parallelen PCI–Aktivität. Weiterhin wurde die Zeit zu Deaktivierung aller möglichen PCI–Master bestimmt.

Die Messungen haben gezeigt, dass die Laufzeitverlängerungen von Software durch den Einfluss von PCI–Geräten erheblich sein können. Da nach der PCI–Spezifikation das Arbitrierungsverfahren lediglich fair sein muss und die Hersteller das gewählte Verfahren oft nicht dokumentieren, kann nicht garantiert werden, dass die Messungen mit PCI–Aktivität wirklich den Worst Case beinhalten. Das Arbitrierungsverfahren des PCI–Busses spielt nur dann eine Rolle, wenn mehrere Geräte gleichzeitig auf den Bus zugreifen möchten. Das Realzeitbetriebssystem muss deshalb sicherstellen, dass nie mehrere Geräte gleichzeitig den Bus anfordern. Somit hat der Arbitrierungsalgorithmus keinen Einfluss mehr auf das Laufzeitverhalten.

Die Umsetzung der hier gewonnenen Erkenntnisse zur Konstruktion eines Realzeitsystems auf Basis von PC–Multiprozessorsystemen ist Thema weiterer Forschungsarbeiten. Insbesondere die Umstände, die eine Deaktivierung aller oder einiger PCI–Geräte beim Aufruf einer Realzeittask rechtfertigen, bedürfen noch eingehenderen Betrachtungen. Falls jedoch eine Realzeittask in größerem Umfang PCI–Zugriffe durchführen soll ist eine Deaktivierung der PCI–Master mitunter sinnvoll, um vorhersagbare Laufzeiten zu erhalten. Parallel hierzu sollte die GPU keine Peripheriezugriffe durchführen.

Zugriffe auf den Hauptspeicher seitens einer Realzeittask werden durch parallele PCI–Aktivität ebenfalls verzögert. Bei der Bestimmung der WCET einer Realzeittask müssen diese Einflüsse berücksichtigt werden. Aktuelle Forschungsarbeiten beschäftigen sich mit der Thematik, die Realzeittask komplett im Cache einer RTU zu halten um somit Zugriffe auf den Hauptspeicher zu vermeiden. Damit könnte die Laufzeit von Realzeitsoftware innerhalb bestimmter Schranken vorherbestimmt werden — Einflüsse durch parallel arbeitende Prozessoren und PCI–Aktivitäten würden damit zu keinen Latenzzeiten mehr führen.

Weiterhin wird der Einfluss verschiedener, für ein Realzeitsystem unverzichtbarer Standard–PCI–Geräte näher untersucht; insbesondere soll hier auf die Anbindung von Festplatten und Prozess–Signaladaptern eingegangen werden. Zusätzlich sollen die Untersuchungen auf Systeme mit vier Prozessoren ausgedehnt werden. Einflüsse der Prozessorarchitektur sind ebenfalls Gegenstand weiterer Untersuchungen.

Danksagung

Der vorliegende Beitrag entstand im Rahmen des Projekts "RECOMS", das von der DFG unter dem Förderzeichen FA 109/15-1 gefördert wird.

Literatur

[1] Barabanov, M.: *A Linux-based RealTime Operating System*, Juni 1997.

[2] Bianchi, E. and L. Dozio: *Some Experiences in fast hard realtime control in user space with RTAI-LXRT.* In *Realtime Linux Workshop*, Orlando, 2000.

[3] Bianchi, E., L. Dozio, G.L. Ghiringhelli, and P. Mantegazza: *Complex Control Systems, Applications of DIAPM–RTAI at DIAPM.* In *Realtime Linux Workshop*, Vienna, 1999.

[4] Burmberger, Gregor: *PC-basierte Systemarchitekturen für zeitkritische technische Prozesse.* Doktorarbeit, Technische Universität München, 2002.

[5] Burns, Alan and Stewart Edgar: *Predicting computation time for advanced processor architectures.* In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, Stockholm, Sweden, June 19–21 2000.

[6] Fischer, Franz, Thomas Kolloch, Annette Muth, and Georg Färber: *A configurable target architecture for rapid prototyping high performance control systems.* In Arabnia, Hamid R. and others (editors): *Proc. of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, volume 3, pages 1382–1390, Las Vegas, Nevada, USA, June 30 – July 3 1997.

[7] Gopalan, K.: *Real–Time Support in General Purpose Operating Systems*, January 2001. Research Proficiency Exam Report.

[8] G. Quaranta, P. Mantegazza: *Using MATLAB-Simulink RTW to Build Real Time Control Applications in User Space with RTAI-LXRT.* In *Realtime Linux Workshop*, Milano, 2001.

[9] Hopfner, Thomas, Jürgen Stohr, Wolfram Faul und Georg Färber: *RTCPU – Realzeitanwendungen auf Dual-Prozessor PC Architekturen.* it+ti — Informationstechnik und Technische Informatik, 43(6):291, Dezember 2001.

[10] Intel Corporation: *MultiProcessor Specification Version 1.4.* Intel Corporation, 1997.

[11] Müller, Frank: *Timing analysis for instruction caches.* Journal of Realtime Systems, 18:217–247, 2000.

[12] Niehaus, Douglas: *Program representation ans translation for predictable real–time systems*, October 1991.

[13] Petters, Stefan M.: *Worst Case Execution Time Estimation for Advanced Processor Architectures.* PhD thesis, Institute for Real–Time Computer Systems, Technische Universität München, September 2002.

[14] Petters, Stefan M. und Alexander von Buelow: *Laufzeitbestimmung von Realzeitsoftware.* it+ti — Informationstechnik und Technische Informatik, 43(4):206–214, August 2001.

[15] Petters, Stefan M. and Georg Färber: *Making worst case execution time analysis for hard real–time tasks on state of the art processors feasible.* In *6th Int. Conf. on Real–Time Computing Systems and Applications (RTCSA'99)*, Hongkong, ROC, December 13–15 1999. IEEE, IEEE Computer Society Press.

[16] Petters, Stefan M., Annette Muth, Thomas Kolloch, Thomas Hopfner, Franz Fischer, and Georg Färber: *The REAR framework for emulation and analysis of embedded hard real–time systems.* In *Proc. of the 10th IEEE Int. Workshop on Rapid Systems Prototyping (RSP'99)*, pages 100–107, Clearwater, Florida, June 16–18 1999. IEEE Computer Society Press.

[17] Puschner, P. and A. v. Schedl: *Computing maximum task execution times — a graph–based approach.* Journal of Realtime Systems, pages 67–91, July 1995.

[18] Schneider, J., Ch. Ferdinand, and R. Wilhelm: *Pipeline bahavior prediction for superscalar processors.* Technical report, Universität des Saarlandes, February 99.

[19] Sebek, Filip: *Cache memories and real–time systems.* Technical report, Department of Computer Engineering, Märlardalen University Västerås, October 2001.

[20] Srinivasan, B., S. Pather, R. Hill, F. Ansari, and D. Niehaus: *A Firm Real–Time System Implementation Using Commercial Off–The–Shelf Hardware and Free Software.* In *Proceedings of Real–Time Technology and Applications Symposium, Denver*, June 1998.

[21] White, R., F. Mueller, C. Healy, D. Whalley, and M. G. Harmon: *Timing analysis of data caches and set–associative caches.* In *3rd IEEE Real–Time Technology and Applications Symposium*, Montreal Canada, June 9–11 1997.

[22] Yodaiken, V.: *The RTLinux Manifesto.* In *Proceedings of The 5th Linux Expo*, Raleigh North Carolina, March 1999.