Ein generisches Treiber-Framework zur HW/SW-Kommunikation mittels OSSS-RMI

Philipp Ittershagen
Carl v. Ossietzky Universität Oldenburg
philipp.ittershagen@uni-oldenburg.de

Philipp A. Hartmann, Kim Grüttner
OFFIS – Institut für Informatik
Oldenburg
{hartmann,gruettner}@offis.de

Achim Rettberg
Carl v. Ossietzky Universität
Oldenburg
achim.rettberg@iess.org

Kurzfassung

Die Realisierung der Kommunikation zwischen verschiedenen Tasks ist eine entscheidende Aufgabe bei der Entwicklung paralleler, eingebetteter Systeme. Wird im Laufe der Verfeinerung eines Systems die Zuordnung einzelner Komponenten auf unterschiedliche Berechnungseinheiten (Hardware, Software) verändert, werden oftmals aufwändige Designänderungen nötig. In dieser Arbeit wird ein generisches Framework vorgestellt, um die Kommunikation von (Software-) Tasks mit sogenannten *Shared Objects* durchgängig zu ermöglichen. Die Kommunikation wird dabei über eine methodenbasierte Schnittstelle mit Hilfe des *Remote Method Invocation* (RMI) Protokolls realisiert. Die Objekte können dabei als dedizierte Hardware, in gemeinsam genutztem Speicher oder lokal zugreifbar vorliegen – durch das hier vorgestellte Framework wird der Zugriff vereinheitlicht. Die Effektivität des Ansatzes wird anhand eines Beispiels evaluiert.

1. Einleitung

Der Entwurf eines komplexen eingebetteten Systems umfasst unter anderem die Partitionierung in Hardware- und Software-Blöcke. Durch die Verwendung von Hardware-Blöcken kann die Leistungsfähigkeit gesteigert werden, allerdings nimmt man dadurch vor allem in Kauf, dass das System aufgrund der geringen Flexibilität nur noch marginal wartbar ist und die Entwicklungskosten des Gesamtsystems entsprechend hoch ausfallen. Werden im Gegensatz dazu überwiegend Softwarekomponenten eingesetzt, so ist das resultierende System zwar in der Entwicklung und Wartung aufgrund der Flexibilität von Software einfacher, doch können dann unter Umständen nicht die benötigten Ausführungszeiten oder Datendurchsätze erreicht werden.

Eine vereinfachte Hardware/Software-Partitionierung besitzt daher einen hohen Stellenwert bei der Entwicklung heutiger Systeme, da sie es dem Entwickler ermöglichen kann, eine angemessene Architektur auszuwählen. Durch geeignete Designmethoden wird deshalb versucht, den Partitionierungsschritt während der Entwicklung so flexibel wie möglich zu gestalten, um aktuellen Produkteinführungszeiten gerecht zu werden [2]. Hierzu gehören vor allem aussagekräftige Simulationsergebnisse, die den Entwickler bei der Partitionierungsentscheidung unterstützen. Eine

homogene Entwicklungsumgebung, in der sowohl Hardware- als auch Softwareblöcke in der gleichen Beschreibungssprache repräsentiert werden können, lässt diese Abschätzungen weiter vereinfachen. Sie bietet dem Entwickler die Möglichkeit, kurzfristig Änderungen an der Partitionierung des Systems durchzuführen.

Bei der Aufteilung eines Systems in Hardware/Software-Blöcke muss neben den Ausführungszeiten der einzelnen Komponenten auch die Kommunikation der Blöcke untereinander beachtet werden. Unterschiedliche Bindungen einzelner Teilsysteme auf Hard- oder Softwarekomponenten können dabei aufwändige Designänderungen erfordern. Wird jedoch die Kommunikation auch über HW/SW-Grenzen hinweg bereits werkzeugseitig transparent unterstützt, kann dies vermieden werden.

Die Verfeinerung und Untersuchung von Interprozesskommunikation ist ein zentraler Bestandteil der OSSS-Designmethodik. Um von einer ausführbaren Spezifikation in OSSS zu einer Implementation auf einer speziellen Plattform zu gelangen, ist jedoch auch die Betrachtung der HW/SW-Schnittstelle wesentlich. In dieser Arbeit wird das Treiber-Framework *rmi4linux* präsentiert, welches eine Laufzeitumgebung für OSSS auf Linux-basierten, eingebetteten Plattformen ermöglicht. Dazu wird zunächst in Abschnitt 2 die Designmethodik OSSS kurz eingeführt. In Abschnitt 4 werden die grundsätzlichen Anforderungen an das Treiber-Framework dargestellt. Getrieben durch diese ermittelten Anforderungen wird in Abschnitt 5 die Umsetzung des Frameworks sowie die HW/SW-Schnittstelle zwischen *rmi4linux* und einem OSSS *Shared Object* und die Unterstützung von *Software Shared Objects* beschrieben. Anschließend wird in Abschnitt 6 eine Evaluation des Frameworks durchgeführt.

2. Vorausgegangene Arbeiten – OSSS

Die vorliegende Arbeit basiert auf der OSSS-Designmethodik[1][10]. OSSS unterstützt den Entwickler bei der Modellierung komplexer, eingebetteter Systeme, indem es eine homogene Beschreibungs- und Verfeinerungsmethode bietet, in der sowohl Hardware- als auch Softwarekomponenten dargestellt werden können.

Als Erweiterung der Modellierungs- und Simulationssprache SystemC [13], bietet OSSS dedizierte Primitive zur Darstellung von Hardwaremodulen, *Software-Tasks* zur Beschreibung von Softwarekomponenten, und *Shared Objects* als zentralen Kommunikationsmechanismus. *Shared Objects* ermöglichen eine objektorientierte Beschreibung gemeinsam genutzter Ressourcen, welche eine methodenbasierte Schnittstelle bereitstellen. Parallel zugreifende Tasks (*Clients*) werden dabei für den Entwickler transparent synchronisiert.

Das initiale Modell eines Systems wird in OSSS auf dem so genannten *Application Layer* modelliert (Abb. 1). Dabei besteht das System aus parallelen Tasks, welche mittels *Methoden*aufrufen auf *Shared Objects* miteinander kommunizieren. Geschätzte Ausführungszeiten (EET) können dabei an die Funktionalität annotiert werden, so dass initiale Entwurfsentscheidungen getroffen werden können.

Zusätzlich zum garantierten, wechselseitigen Ausschluss von *Shared Object*-Methodenaufrufen können logische Vorbedingungen mit diesen Methoden verknüpft werden, so genannte *Guards*. Basierend auf dem internen Zustand des Objekts können hiermit Anfragen solange blockiert werden, bis die entsprechende Bedingung (wieder) erfüllt ist. Ist ein Guard nicht erfüllt, so muss zunächst

[1] Oldenburg System Synthesis Subset

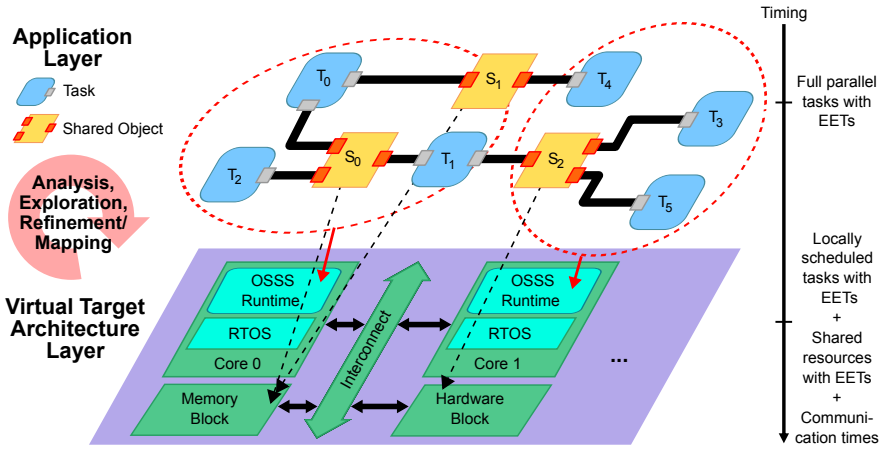


Abb. 1: Übersicht über die OSSS-Entwurfsmethodik [5].

der interne Zustand des Objekts von anderen Clients geändert werden, bis die durch den Guard blockierte Anfrage ausgeführt werden kann.

Um verschiedene Architektur-Alternativen zu explorieren, bietet OSSS eine weitere Verfeinerungsebene, den so genannten *Virtual Target Architecture Layer* (VTA). Auf dieser Ebene werden die Komponenten der Zielplattform abstrakt dargestellt (Prozessoren, Speicher, dedizierte Hardware-Blöcke, Kommunikationsinfrastruktur). Die Tasks und Objekte des Application Layer werden dann spezifischen Komponenten auf dem VTA Layer zugeordnet. Dies ermöglicht eine genauere Simulation, welche Ressourcenkonflikte durch gemeinsam genutzte Komponenten sichtbar macht. Tasks und insbesondere Shared Objects können dabei sowohl dedizierten Hardware-Blöcken als auch Prozessoren/Speichern zugewiesen werden [5].

Im Falle einer Zuordnung von mehreren Software-Tasks auf einen Prozessorkern wird in der Simulation bereits eine Abstraktion des Betriebssystems inklusive der resultierenden Scheduling-Artefakte berücksichtigt [6]. Zur Kommunikation mit entfernten (Hardware) Shared Objects wird die Technik der *Remote Method Invocation* (RMI) angewandt, um die abstrakten Methodenaufrufe auf ein konkretes, plattform- und systemspezifisches Protokoll abzubilden [4]. Im Falle einer Hardware-Implementation bieten Shared Objects einen eigenen Scheduler, um Anfragen zu serialisieren. Dieser Scheduler ist für eine Arbitrierung nötig, damit die durch Guards blockierten Methodenaufrufe durch den Aufruf einer anderen Methode wieder deblockiert werden können.

Um nun von dem VTA Layer-Modell zu einer Implementation auf der gewünschten Zielplattform zu gelangen, werden die in SystemC/OSSS beschriebenen Hardware-Komponenten mittels des Synthese-Werkzeugs *Fossy*[2] automatisch in RT-Level VHDL-Code übersetzt. Für die Software-Tasks ist auf der Zielplattform eine Laufzeitumgebung nötig, welche die Semantik der *Software Tasks* und insbesondere der *Shared Objects* auf Primitive des zugrunde liegenden Betriebssystems abbildet. In der hier vorgestellten Arbeit wird eine generische Implementation des RMI-Ansatzes unter Linux vorgestellt, welche zur Realisierung eben dieser Laufzeitumgebung verwendet werden soll.

Für die verschiedenen Alternativen des Zugriffs auf gemeinsam genutzte Ressourcen, sind im allgemeinen Fall nun auch verschiedene Szenarien der Kommunikation von Tasks mit *Shared Objects* zu betrachten [5]. Dies beinhaltet die Kommunikation mit dedizierten Hardware-Objekten,

[2]Functional Oldenburg System SYnthesiser, `http://system-synthesis.org`

Software-Objekten in gemeinsam genutztem Speicher zwischen verschiedenen Prozessoren, als auch herkömmliche kritische Abschnitte mit Hilfe der existierenden Betriebssystemprimitive im Falle lokaler Objekte.

3. Verwandte Arbeiten

In der Literatur finden sich zahlreiche Ansätze zur automatischen Generierung von Softwaretreibern für die HW/SW-Kommunikation. Bereits in [11] wird eine automatisierte Lösung für das Erstellen von applikationsspezifischen heterogenen Multiprozessor-Architekturen vorgestellt, um die fehleranfällige manuelle Entwicklung von Kommunikationskanälen zwischen Hard- und Software zu automatisieren. Eine Möglichkeit zur automatischen Generierung von Linux-Treibern wurde in [8] untersucht. Dabei wurde nicht nur ein generischer Ansatz verfolgt, sondern eine generelle Abstraktion der Treiber-Entwicklung. Wir möchten uns hier allerdings auf SystemC-basierte Ansätze, mit dem Ziel eine Implementierung der HW/SW Kommunikation für ein eingebettetes HW/SW System abzuleiten, beschränken.

In [12] wird ein Verfahren zur automatischen HW/SW-Generierung basierend auf der SpeC-Verfeinerungsmethodik für Software vorgestellt. Dieser Ansatz ist vergleichbar zu unserem. Im Gegensatz zu Shared Objects werden hier einfachere Channel-Modelle benutzt, die zwar einen wechselseitigen Ausschluss per Mutex erlauben, aber kein zusätzliches Scheduling wie bei HW Shared Objects ermöglichen.

In [7] wird ein Ansatz zur Software-Modellierung basierend auf SystemC mittels einer POSIX-kompatiblen Simulationsbibliothek vorgestellt. Diese erlaubt dem Designer, SW-Simulationen und Timings durchzuführen und dabei die Einflüsse eines POSIX-kompatiblen RTOS zu beachten. Dabei wird automatisch C++-Code für die unterstützten Betriebssysteme erzeugt. Dieser Ansatz unterstützt allerdings weder die Kommunikation zwischen Tasks auf unterschiedlichen Betriebssystemen, noch die mit dedizierten HW Ressources.

Die Arbeit in [9] stellt einen Ansatz zur systematischen Verfeinerung eines verteilten eingebetteten Systems dar. Mit Hilfe von SystemC wird die Beschreibung des Gesamtsystems schrittweise in Form eines virtuellen Prototypen bis hinunter auf ein taktzyklusgenaues Modell verfeinert. Aus diesem Verfeinerungsprozess lässt sich SW-Code mit RTOS Primitiven und Kommunikationstreibern für eine paketbasierte Netzwerkkommunikation (CAN) ableiten. Dieser Ansatz unterscheidet sich wesentlich in diesem Punkt, da er keine Shared Memory Architektur inkl. der daraus entstehenden Zugriffskonflike berücksichtigt.

4. Remote Method Invocation – Anforderungen

Um die Implementation spezieller Applikationen auf einer vorgegebenen Plattform zu erleichtern, muss für die Software Tasks eine geeignete Plattform-, oder Hardwareabstraktionsschicht (HAL) geschaffen werden um den Tasks eine einheitliche Schnittstelle, unabhängig von der Bindung der benötigten Objekte (HW, Speicher, oder lokal) zu ermöglichen. In der vorliegenden Arbeit wird dies insbesondere für die *Remote Method Invocation* betrachtet.

Im RMI-Protokoll ist im Falle von Software-Clients und Hardware-*Shared Objects* die Kommunikation zwischen Client und Objekt durch vier Zustände gekennzeichnet, die bei der Durchführung eines Methodenaufrufs durchlaufen werden [4]:

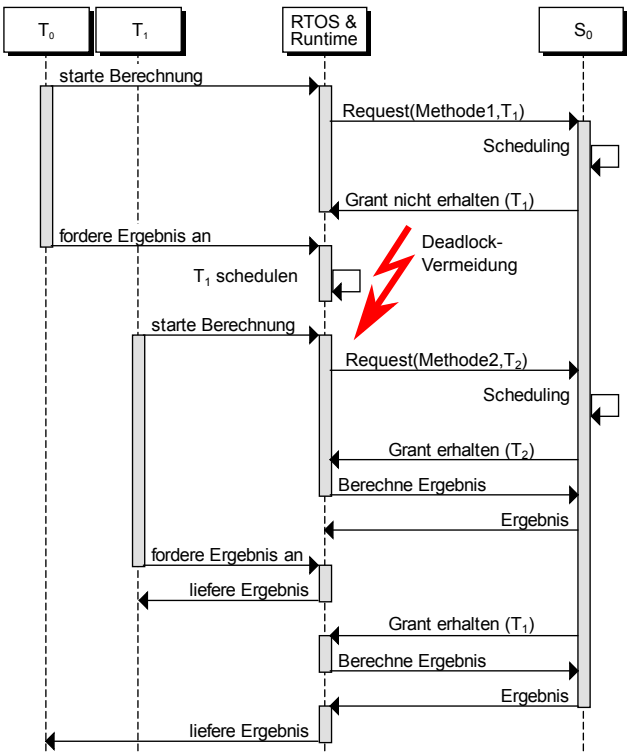


Abb. 2: Beispielhafter Ablauf zweier Software-Tasks, angelehnt an die Konfiguration von Shared Object und Tasks aus Abb. 1.

1. Die Argumente der aufzurufenden Methode werden serialisiert, um sie als Folge geeignet angeordneter Worte über den Kommunikationskanal (Bus) senden zu können.
2. Das Shared Object wird über den angeforderten Aufruf notifiziert. Dabei sendet der Client seine eindeutige Client-ID zusammen mit der Methoden-ID der aufzurufenden Methode an das Shared Object. Dessen Scheduler kann nun aufgrund der Client ID die Arbitrierung vornehmen und den Aufruf gestatten.
3. Die Argumente werden vom Shared Object empfangen und deserialisiert, um im Anschluss daran den Methodenaufruf durchführen zu können.
4. Ein etwaiger Rückgabewert durchläuft diese Schritte in umgekehrter Reihenfolge.

Da nun mehrere Clients des gleichen Objekts auf der gleichen CPU ausgeführt werden können, ist bei der Umsetzung des RMI-Protokolls zusätzliche Unterstützung seitens des Betriebssystems, bzw. der Laufzeitumgebung nötig, wie in Abb. 2 verdeutlicht. Die Ursache für die erweiterten Anforderungen an die Synchronisation entstehen durch die mehrstufige Blockierung durch ein Shared Object, die zusätzlich zum wechselseitigen Ausschluss zur Serialisierung konkurrierender Anfragen durch die möglichen Guard-Bedingungen entstehen kann.

Ist eine lokale Task durch eine solche Guard-Bedingung blockiert, würde aber den (lokalen) Lock auf das Interface des Shared Objects nicht freigeben, kann es zu einem Deadlock führen, falls die Task, die den Guard durch separate Methodenaufrufe wieder freigeben kann, auf der gleichen CPU ausgeführt wird.

Neben dieser komplexeren Synchronisation ist die Vereinheitlichung des Zugriffs auf *Software Shared Objects* wünschenswert. Die Methoden solcher Software-Objekte werden auf der CPU des Clients ausgeführt, der Zustandsspeicher kann, neben lokalem Speicher jedoch auch gemeinsam, von unterschiedlichen CPU/OS-Instanzen aus genutzter Speicher sein. Im letzteren Fall kann nicht auf Betriebssystemprimitive zur Realisierung des wechselseitigen Ausschlusses zurückgegriffen werden, dies muss explizit implementiert werden. Zusammenfassend können folgende Anforderungen an das Treiber-Framework formuliert werden:

- Der Zugriff von mehreren Clients auf mehrere Shared Objects unterschiedlicher Art soll möglich und transparent sein.
- Eine gemeinsame Abstraktion von der Bindung des Shared Objects ($\rightarrow$ Hardware, Shared Memory, lokaler Speicher) aus Sicht des Clients soll bereitgestellt werden.
- Zur Erhöhung der erreichbaren Parallelität soll die Interferenz zwischen Client und (Hardware) Shared Object durch ein asynchrones Protokoll reduziert werden.
- Zur leichteren Portierung zwischen verschiedenen Plattformen soll die Implementation möglichst getrennt von den plattformspezifischen Informationen erfolgen.

Im Rahmen einer prototypischen Implementierung haben wir uns für die Nutzung des für eine Vielzahl von Plattformen verfügbaren Betriebssystems Linux entschieden. Im nachfolgenden Abschnitt wird das generische Treiber-Framework *rmi4linux* für die Kommunikation mit Shared Objects beschrieben.

5. Umsetzung basierend auf Linux

Das *rmi4linux* Treiber-Framework besteht aus einem Kernel-Modul und einer Software-Bibliothek zur vereinfachten Verwendung des Kernel-Moduls für die unter Linux als Prozesse abgebildeten Clients. Die plattformspezifischen Informationen wie z.B. definierte Methoden oder physikalische Adressbereiche der Shared Objects werden statisch außerhalb des eigentlichen Frameworks in einem sog. *Device Tree* gespeichert [3]. Dadurch ist der eigentliche Code frei von externen Abhängigkeiten und kann flexibel wiederverwendet werden.

Ein Client kommuniziert mit dem Treiber mittels *ioctl*-Befehlen zur Signalisierung von Start und Ende der RMI. Die Übermittlung der Argumente und des Ergebniswertes geschieht über einen gemappten Adressbereich. Dabei erhält jeder Client einen eigenen Speicherbereich, der bei der Initialisierung des Clients vom Treiber-Framework vorbereitet und anschließend in den Adressbereich des Clients eingeblendet wird. Somit können Kopiervorgänge zwischen Kernel- und Userspace vermieden werden, da der Client direkt auf den Speicherbereich zugreifen kann.

RMI erfordert eine Synchronisation über die Hardware/Software-Grenze hinweg. Erst wenn der Guard einer Methode erfüllt ist, kann diese ausgeführt werden. Dies bedeutet, dass das Framework unter Umständen auf einen Grant warten muss. Erst nach dessen Gewährung können die Argumente an das Shared Object übergeben und der Methodenaufruf durchgeführt werden. Danach muss das Framework auf die Beendigung der Methode warten, um den Ergebniswert auslesen zu können. Diese Umstände erfordern einerseits die Überwachung des Grants für den Zugriff auf das Shared Object und andererseits das Warten auf das Ergebnis eines Methodenaufrufs.

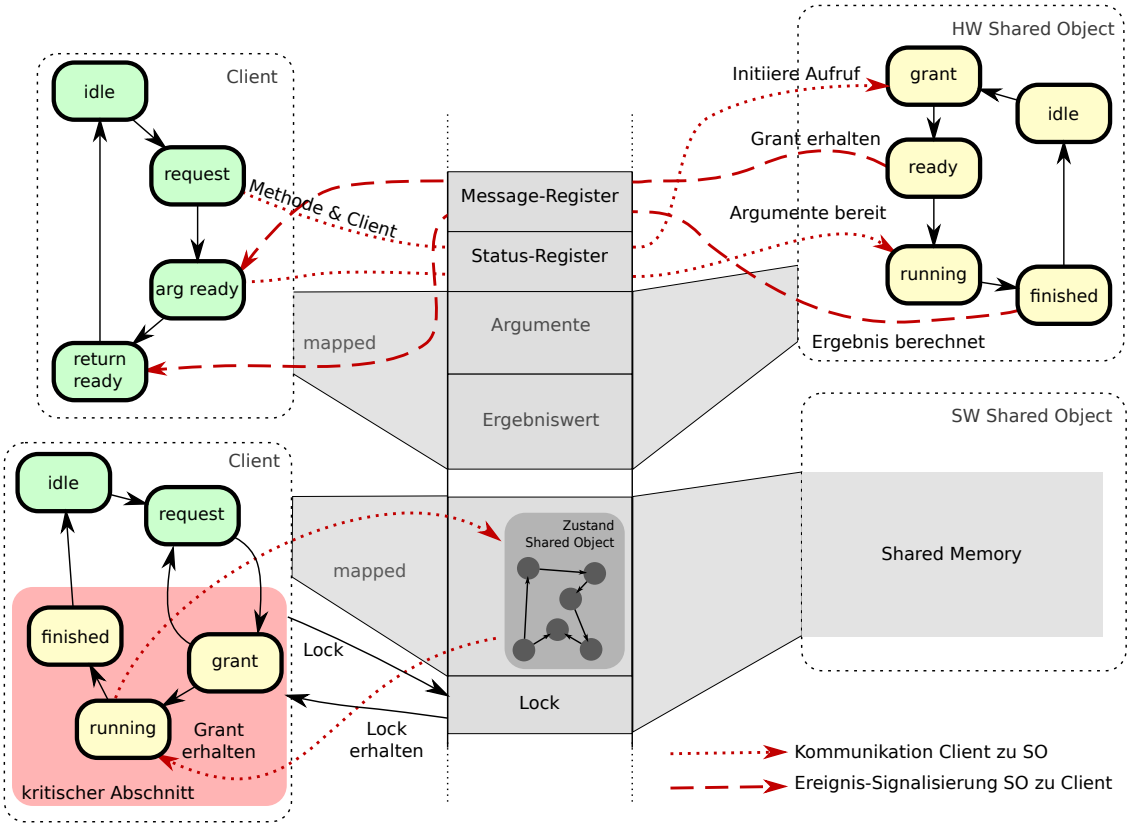


Abb. 3: Zustände der Clients (links) und der Shared Objects (rechts), sowie das Speicherlayout (Mitte).

Um den Aufruf einer Methode zu ermöglichen, muss das Treiber-Framework Informationen über den Zustand des Shared Objects besitzen. Es muss eine geeignete Kommunikation stattfinden, die signalisiert, dass ein Methodenaufruf starten kann oder Ergebnisse gelesen werden können. Zu jeder RMI muss bekannt sein, welcher Client den Aufruf initiiert hat, damit das Framework bei eintretenden Ereignissen mehrere gleichzeitig laufende Aufrufe unterscheiden kann. Es ist dabei festzustellen, dass ein Client nur maximal einen Methodenaufruf gleichzeitig durchführen kann. Deshalb enthält die Schnittstelle für jeden Client ein Status-Register, in dem die aktuell auszuführende Methoden-ID gespeichert wird.

Über das Status-Register wird zusätzlich der Zustand (idle, grant, ready, running, finished) der gerade ausgeführten Methode festgelegt, damit ein Client signalisieren kann, wann anstehende Argument/Ergebnis-Übertragungen beendet wurden.

Über das einmal pro Shared Object existierende Message-Register können Ereignisse an einen Client gesendet werden. Hat eine Methode einen Grant erhalten oder wurde eine Methode beendet, besteht Handlungsbedarf seitens des Clients und das Shared Object signalisiert dies über das Message-Register. Daher wird das Message-Register bei der Behandlung eines Interrupts bzw. beim Polling gelesen, um den Status des Shared Objects zu überprüfen und entsprechend reagieren zu können.

Während die Kommunikation mit einem Hardware Shared Object über Register verläuft, verdeutlicht Abb. 3 auch die Einbettung eines Software Shared Objects im Ausführungskontext eines Clients. Lediglich der innere Zustand eines Software Shared Objects, und damit auch die Guard-Bedingungen und methodenübergreifenden Variablen werden in einem Shared Memory gespei-

chert. Der Zugriff auf diesen Speicherbereich ist exklusiv (kritischer Abschnitt).

Hat der Client exklusiven Zugriff erhalten, kann der Guard evaluiert werden, da nun der innere Zustand des Software Shared Objects verfügbar ist. Erhält der Client auch den Grant, führt er die Methode mit dem inneren Zustand des Software Shared Objects aus. Nach Beendigung des Aufrufs wird der Speicherbereich wieder für andere Clients freigegeben. Der mit Hilfe eines Locks implementierte CPU-übergreifende Synchronisationsmechanismus ermöglicht es, mehreren auf unterschiedlichen CPUs laufende Clients den Zugriff auf ein Software Shared Object zu ermöglichen.

6. Evaluation

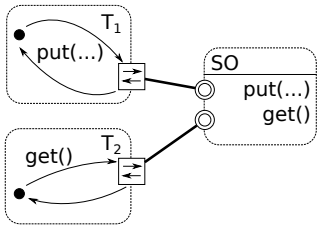
Die Evaluation des Treiber-Frameworks erfolgte auf einem Xilinx ML410-Board. Das Board besitzt einen Virtex-4-FPGA mit zwei eingebauten PowerPC-Prozessoren, die mit einer Taktung von 100MHz konfiguriert wurden. Das Plattform-Design beinhaltet zusätzlich zu diesen beiden Prozessoren auch zwei Interrupt-Controller, um auf Anfragen des Shared-Objects reagieren zu können.

Insgesamt wurden vier Experimente mit einem Shared Object und zwei Tasks durchgeführt. Die ersten zwei Experimente wurden mit einem Hardware Shared Object durchgeführt, die anderen zwei mit einem Software Shared Object. Um auch die Multiprozessor-Fähigkeit des Frameworks zu evaluieren, wurde das Hard- bzw. Software Shared Object einmal mit zwei auf der selben CPU befindlichen Tasks benutzt, das andere Mal befanden sich die Tasks auf unterschiedlichen CPUs. Tabelle 4(b) listet zur Übersicht die vier verschiedenen Konfigurationen auf.

Gemessen wurden die Zeiten mit dem im Linux-Kernel eingebauten *function tracer* [1]. Es ist festzustellen, dass das Tracing der Funktionsaufrufe diese in ihrer Ausführungszeit beeinflusst, weshalb das Tracing nur für die Funktionen der asynchronen Initiierung der RMI (Start) und dem Warten auf das Ergebnis (Rückgabe) aktiviert wurde.

Das Shared Object stellt dabei in allen vier Fällen einen FIFO-Buffer dar, der maximal ein Element speichern kann. Einer der beiden Tasks fügt Elemente in den Buffer, der andere entfernt diese wieder. Die Software-Tasks rufen die entsprechende Methode des Shared Objects auf, sobald sie dazu in der Lage sind und dies in einer Endlosschleife wiederholen (vgl. dazu Abb. 4(a)). Die geringe Kapazität des Buffers erlaubt es, das durch einen vollen bzw. leeren Buffer erzwungene Blockierungsverhalten des Frameworks und dadurch die Wartezeiten der Tasks zu beobachten.

Das Zeitverhalten eines Methodenaufrufs der ersten beiden Experimente ist im Diagramm in Abb. 5(a) dargestellt. Auffällig ist, dass beim zweiten Experiment lediglich ca. die Hälfte der Zeit

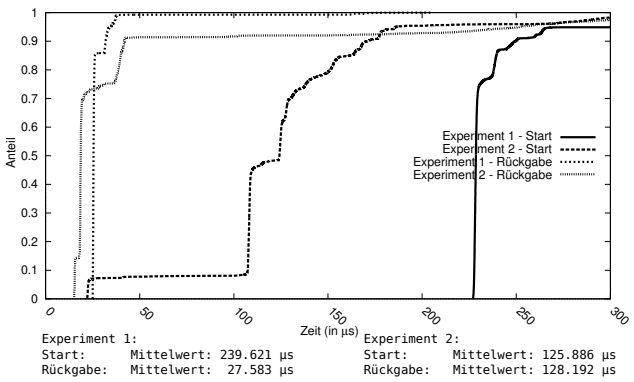


(a) Zusammenspiel zwischen Tasks und Shared Object

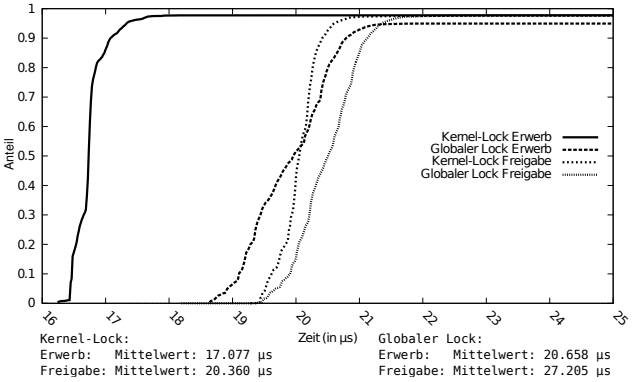
	CPUs		Shared Object	
	CPU_1	CPU_2	Hardware	Speicher
Experiment 1	T_1,T_2	-	SO	
Experiment 2	T_1	T_2	SO	
Experiment 3	T_1,T_2	-		SO
Experiment 4	T_1	T_2		SO

(b) Konfiguration der einzelnen Experimente

Abb. 4: Visualisierung und Konfiguration der Evaluation



(a) Kumulative Distribution des Zeitaufwands eines Methodenaufrufs und der Anfrage auf den Rückgabewert eines Hardware Shared Objects (Experiment 1 und 2)



(b) Vergleich zwischen globalem Lock und kernel-internem Locking von Experiment 3 und 4.

Abb. 5: Ergebnisse der Evaluation

benötigt wird. Dies lässt sich dadurch erklären, dass bei der Konfiguration mit zwei CPUs jede CPU auch nur die Hälfte der Interrupts tatsächlich behandeln muss. Die Interrupts trafen direkt nach Initiierung der Methode ein, weshalb sie die Laufzeit des Aufrufs einer Methode beeinflussen. Durch die frühe Behandlung der Interrupts konnte beobachtet werden, dass Tasks nicht mehr auf die Durchführung der Shared Object Methode warten mussten, weshalb die in Abb. 5(a) dargestellten Zeiten für die Anfrage auf den Rückgabewert entsprechend gering ausfallen.

Die Experimente 3 und 4 zeigen für die Wartezeiten auf ein Lock nahezu identische Ergebnisse. Der Erwerb des exklusiven Zugriffs benötigt also bei zwei CPUs genauso viel Zeit wie bei einer. Aus diesem Grund wurden die Ergebnisse beider Experimente mit Software Shared Objects in Abb. 5(b) zusammengefasst. Da beim dritten Experiment nur eine CPU eingesetzt wird, wurde zum Vergleich zwischen global und lokal verfügbarem Shared Object das vom Betriebssystem angebotene Locking-Verfahren eingesetzt. Abb. 5(b) zeigt, dass bei Einsatz des lokalen Lockings eine etwas geringere Latenz auftritt. Besonders der Erwerb des Locks ist schneller als das globale Locking des Treiber-Frameworks.

Die Ausführung von Methoden eines Software Shared Objects unterliegt momentan noch stark der Konfiguration der eingesetzten Laufzeitumgebung und ein Vergleich mit Hardware Shared Objects ist daher nur eingeschränkt möglich. Während der Durchführung der Experimente 3 und 4 stellte sich heraus, dass die Ausführung einer Methode ungefähr $4ms$ benötigt, da die Zeitscheiben des Linux-Schedulers nicht die nötige Auflösung boten, um eine mit den Aufrufen der Experimente 1 und 2 vergleichbare Reaktionsgeschwindigkeit zu erreichen.

7. Fazit

In der vorliegenden Arbeit wurde das Treiber-Framework *rmi4linux* vorgestellt, welches eine einheitliche Schnittstelle für OSSS *Software Tasks* zum Zugriff auf *Shared Objects* unter Linux bietet. Dabei werden die möglichen, verschiedenen Ausprägungen der *Shared Objects*, welche als Hardware, in gemeinsam genutztem Speicher, oder lokal zugreifbar vorliegen können, durch eine gemeinsame Abstraktion bereitgestellt.

In Abschnitt 6 wurden Experimente bezüglich der unterschiedlichen Kosten beim Zugriff auf solch gemeinsam genutzte Ressourcen dargestellt. Es lässt sich beobachten, dass der Overhead des

Zugriffs durchaus abhängig von der gewählten Variante und dem Zugriffsmuster ist.

In zukünftigen Arbeiten in diesem Zusammenhang ist geplant, die Ausführungszeiten der Software Shared Objects näher zu betrachten, um einen Vergleich zwischen Hard- und Software Shared Object und damit entsprechende Design-Entscheidungen zu unterstützen. Diese Entscheidungen können durch Rückannotation der Kommunikationszeiten an das Simulationsmodell die Exploration von HW/SW-Partitionierungsmöglichkeiten weiter verbessern.

Literatur

[1] Bird, T.: *Measuring Function Duration with Ftrace*. In: *Proceedings of the Linux Symposium*, S. 47–54, Montreal, Quebec, Canada, Juli 2009. The Linux Symposium.

[2] Gajski, D. D., J. Zhu und R. Dömer: *Hardware-Software Co-Synthesis – Principles and Practice: Essential Issues in Codesign*. Kluwer Academic Publishers, Dordrecht, Niederlande, 1997.

[3] Gibson, D. und B. Herrenschmidt: *Device trees everywhere*. Dunedin, New Zealand, Feb. 2006. OzLabs, IBM Linux Technology Center, linux.conf.au. `http://ozlabs.org/~dgibson/home/papers/dtc-paper.pdf`.

[4] Grüttner, K., C. Grabbe, F. Oppenheimer und W. Nebel: *Object Oriented Design and Synthesis of Communication in Hardware-/Software Systems with OSSS*. In: *Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI), Hokkaido, Japan*, Okt. 2007.

[5] Hartmann, P. A., K. Grüttner, A. Rettberg und I. Podolski: *Distributed Resource-Aware Scheduling for Multi-Core Architectures with SystemC*. In: *DIPES'10: IFIP Conference on Distributed and Parallel Embedded Systems*, Brisbane, Australia, Sep. 2010.

[6] Hartmann, P. A., P. Reinkemeier, H. Kleen und W. Nebel: *Modeling of Embedded Software Multitasking in SystemC/OSSS*. In: Radetzki, M. (Hrsg.): *Languages for Embedded Systems and their Applications*, Bd. 36 d. Reihe *Lecture Notes in Electrical Engineering*, S. 213–226. Springer Niederlande, 2009.

[7] Herrera, F., H. Posadas, P. Sánchez und E. Villar: *Systematic Embedded Software Generation from SystemC*. In: *in Proc. of DATE*, 2003.

[8] Katayama, T., K. Saisho und A. Fukuda: *Prototype of the device driver generation system for UNIX-like operating systems*. In: *Intl. Symposium on Principles of Software Evolution (ISPSE'2000)*, S. 302–310, 2000.

[9] M. Krause, O. Bringmann, W. R.: *Communication Refinement and Target Software Generation using SystemC*. In: *GI/ITG/GMM Workshop für Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, Dresden, 2006.

[10] OFFIS Institut für Informatik: *OSSS – A Library for Synthesisable System Level Models in SystemC*. Arbeitsgruppe Hardware/Software Design Methodik, Oldenburg, 2008.

[11] Vercauteren, S., B. Lin und H. De Man: *Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications*. In: *Custom HW/SW Applications. Design Automation Conference*, S. 521–526, 1996.

[12] Yu, H., R. Dömer und D. D. Gajski: *Software and Driver Synthesis from Transaction Level Models*. In: Rettberg, A., M. C. Zanella und F. J. Rammig (Hrsg.): *From Specification to Embedded Systems Application*, Bd. 184 d. Reihe *IFIP International Federation for Information Processing*, S. 65–76. Springer Boston, 2005.

[13] IEEE P1666™-2005: *Standard SystemC Language Reference Manual*, 2005.