Intercore-Kommunikation für Multicore-Mikrocontroller
Die Folgen durch effizientes Speichermanagement minimieren

Philipp Jungklaß, Ingenieurgesellschaft Auto und Verkehr GmbH
Prof. Dr.-Ing. Mladen Berekovic, Universität zu Lübeck –
Institut für Technische Informatik

Der Einsatz von embedded Multicore-Mikrocontrollern in modernen Steuergeräten mit harter Echtzeitanforderung stellt Entwickler immer wieder vor größere Herausforderungen, da die Separierung der Software häufig nicht in dem Maße möglich ist, wie es die Anzahl der Prozessorkerne vorgibt. Dadurch ist es notwendig, dass zwischen den Prozessorkernen ein Datenaustausch stattfindet. Diese Intercore-Kommunikation erfolgt derzeit über geteilte Speicher, auf welche die Prozessorkerne konkurrierend zugreifen. Bedingt durch diese parallelen Zugriffe entstehen Wartezyklen, welche für ein System mit harter Echtzeitanforderung aufwendig zu kalkulieren sind. Aus diesem Grund wird in diesem Artikel ein Prioritätsbasiertes Verfahren zur Intercore-Kommunikation vorgestellt, dass die Wartezyklen durch eine effektive Nutzung der vorhandenen Speicherhierarchie minimieren. Als Nachweis der Funktionsfähigkeit wird das Verfahren auf zwei embedded Multicore-Mikrocontroller der AURIX-Familie portiert und mit dem bisherigen Ansatz verglichen.

1 Einführung
Seit der Einführung der ersten Steuergeräte mit embedded Multicore-Mikrocontroller ist die Anzahl der verfügbaren Prozessorkerne stetig gestiegen. Diese Entwicklung stellt besondere Anforderungen an die Software der Steuergeräte, welche auf die vorhandenen Prozessorkerne aufgeteilt werden muss. Dabei ist jedoch zu beachten, dass zwischen den Software-Komponenten auf den unterschiedlichen Kernen Abhängigkeiten bestehen, welche einen kernübergreifenden Datenaustausch erfordern. Aktuell erfolgt der Datenaustausch zwischen den Prozessorkernen über einen geteilten globalen Speicher, auf welchen die Kerne konkurrierend zugreifen. Die dadurch entstehenden Wartezeiten sind schwer vorhersagbar und können die Echtzeitfähigkeit des Systems gefährden. Daher wird in diesem Artikel ein Verfahren vorgestellt werden, welches durch eine intelligente Nutzung der vorhandenen Speicherhierarchie des embedded Multicore-Mikrocontrollers die Anzahl der Wartezyklen minimiert, wodurch die Vorhersagbarkeit erhöht wird [10][11].

Der Aufbau dieses Artikels gliedert sich in sechs Bereiche. Nach der Einführung wird der aktuelle Stand der Technik präsentiert. Im Anschluss wird das entwickelte Konzept vorgestellt und detailliert beschrieben. Als nächstes wird der genutzte Versuchsaufbau erläutert, bevor im Anschluss die ermittelten Resultate der zwei Testplattformen präsentiert werden. Den Abschluss dieses Artikels bildet die Diskussion der erreichten Resultate.

2 Stand der Technik

2.1 Aufbau der embedded Multicore-Mikrocontroller
Grundlegend besitzen die derzeit verfügbaren Multicore-Mikrocontroller der verschiedenen Hersteller einen ähnlichen Aufbau. Dazu gehört, je nach Prozessorfamilie und Derivat, eine unterschiedliche Anzahl an Prozessorkernen, welche mittels einer

Crossbar mit den globalen Speichern des Systems sowie untereinander verbunden sind. Der Vorteil einer Crossbar liegt in der Realisierung von parallelen Verbindungen zwischen verschiedenen Teilnehmern. Zusätzlich bieten die Kerne jeweils einen lokalen Speicher, auf welchen diese ohne Wartezyklen direkt zugreifen können. Abbildung 1 zeigt den schematischen Aufbau [1][2][3][7].

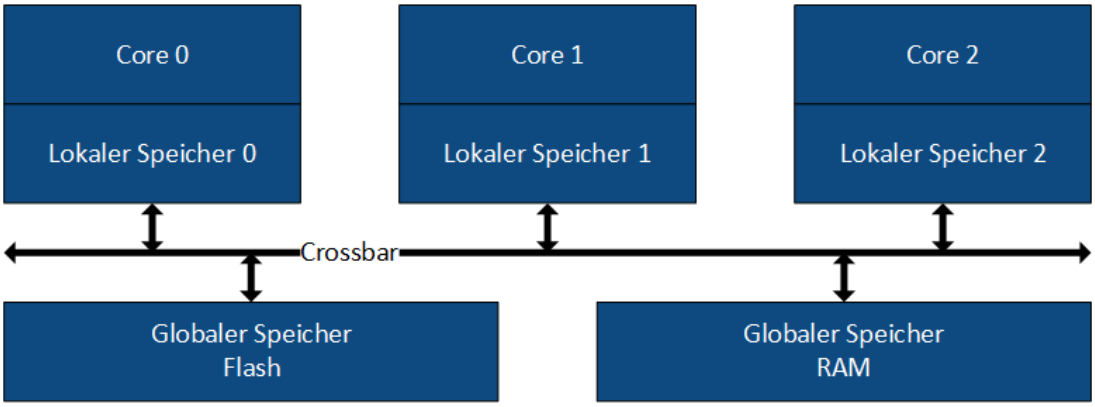

Abbildung 1: Schematischer Aufbau eines embedded Multicore-Mikrocontrollers mit drei Kernen [1][2][3]

2.2 Intercore-Kommunikation

In modernen Steuergeräten mit embedded Multicore-Mikrocontrollern erfolgt der Datenaustausch zwischen den Prozessorkernen über einen globalen Speicher, auf welchen alle Kommunikationsteilnehmer Zugriff haben. Zur Vermeidung von inkonsistenten Daten, welche bei konkurrierenden Zugriffen mehrerer Kommunikationsteilnehmer entstehen können, werden drei grundlegende Mechanismen zur Absicherung genutzt [5][6].

Mutex

Ein Mutex ist ein Verfahren, welches für einen wechselseitigen Ausschluss von verschiedenen Kommunikationspartnern genutzt wird. Wichtig ist hierbei, dass nur der Partner, welcher den exklusiven Zugriff auf die geteilten Ressourcen reserviert hat, diese auch wieder freigeben kann [6].

Semaphore

Mittels einer Semaphore wird ebenfalls der exklusive Zugriff auf geteilte Ressourcen geregelt. Im Gegensatz zu einem Mutex muss jedoch nicht zwingend der reservierende Kommunikationspartner auch der Freigebende sein [6].

Spinlock

Der Begriff Spinlock beschreibt gleichermaßen ein Verfahren, welches den exklusiven Zugriff auf eine geteilte Ressource regelt. Im Gegensatz zu einem Mutex oder einer Semaphore warten die Kommunikationspartner jedoch aktiv, falls die benötigte Ressource derzeit nicht exklusiv verfügbar ist [6].

Alle drei Verfahren zur Regelung des Zugriffs auf geteilte Speicher würden von einer beschleunigten Intercore-Kommunikation profitieren. Dies ist darauf zurückzuführen, dass bei jedem exklusiven Zugriff ein kritischer Abschnitt im Programmcode des Steuergeräts ausgeführt wird, welcher nicht unterbrochen werden darf. Je länger

solch ein Abschnitt dauert, desto schwieriger ist die Aufrechterhaltung der harten Echtzeitfähigkeit.

3 Konzept

Wie bereits in Abschnitt 2 beschrieben, erfolgt der Zugriff der Prozessorkerne auf ihre lokalen Speicher deutlich schneller als auf die globalen Speicher, welche mittels einer Crossbar angebunden sind. Daher wird in dem hier vorgestellten Konzept die Intercore-Kommunikation mittels der lokalen Speicher realisiert. Hierbei ist zu beachten, dass der Zugriff eines Kerns auf den lokalen Speicher eines anderen Kerns deutlich langsamer als der Zugriff auf den eigenen lokalen Speicher erfolgt, jedoch immer noch schneller als auf den globalen Speicher. Im Gegensatz zu dem bisher genutzten Verfahren wird in dem hier vorgestellten Konzept die Priorität der Prozessorkerne in einem Steuergerät berücksichtigt. Die Vergabe der Priorität kann dabei anhand der zugewiesenen Aufgabe, zum Beispiel dem Sicherheitslevel, oder der Auslastung erfolgen.

Zur besseren Beschreibung des Konzepts wird das Beispiel in der Abbildung 2 genutzt. In diesem System existieren drei Prozessorkerne, welche jeweils über einen lokalen Speicher verfügen und über eine Crossbar mit den globalen Speichern und untereinander verbunden sind. Jeder der drei Kerne besitzt eine separate Priorität. Zur Darstellung der Intercore-Kommunikation in den sich anschließenden Beispielen wird die Farbe Rot für einen Schreibzugriff und Grün für einen Lesezugriff verwendet. Mit der Farbe Gelb wird eine Operation eines Kerns mit seinem lokalen Speicher aufgezeigt, welche nicht Teil einer Intercore-Kommunikation ist.

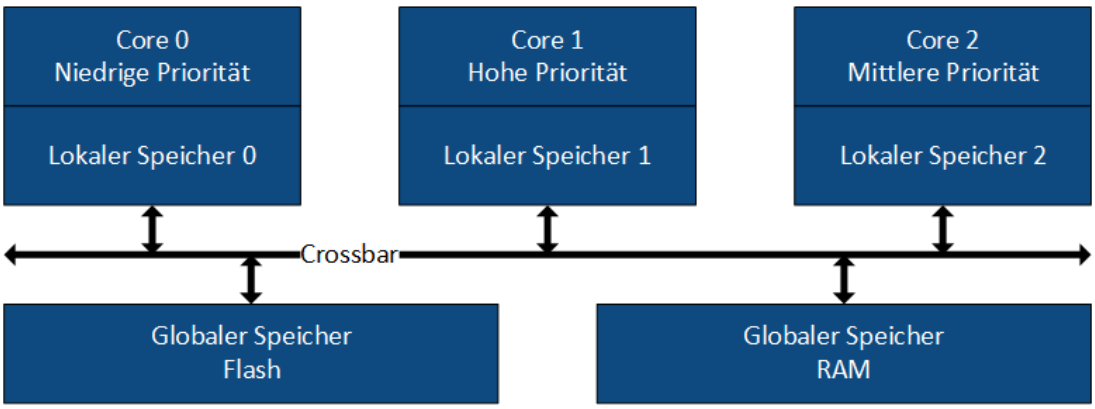

Abbildung 2: Schematischer Aufbau eines embedded Multicore-Mikrocontrollers mit drei Kernen und unterschiedlichen Prioritäten

In dem ersten Szenario, welches in Abbildung 3 dargestellt ist, stellt Core 0 Informationen bereit, welche Core 1 und Core 2 benötigen. Da Core 0 die niedrigste Priorität in diesem System besitzt, schreibt er die Daten in die lokalen Speicher von Core 1 und Core 2. Dadurch können diese bei einem lesenden Zugriff auf die Daten deutlich schneller zugreifen, wodurch Wartezyklen bei den höher priorisierten Kernen vermieden werden.

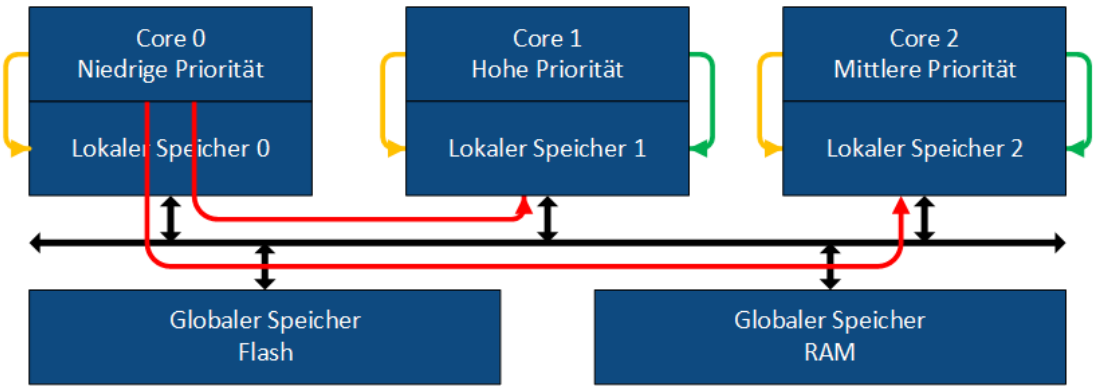

Abbildung 3: Intercore-Kommunikation durch Core 0

Bei dem zweiten Anwendungsfall in Abbildung 4 stellt Core 1 Daten für die anderen beiden Kerne zur Verfügung. Da Core 1 die höchste Priorität in dem hier gezeigten System besitzt, schreibt dieser seine Werte in seinen eigenen lokalen Speicher. Durch diese Konfiguration benötigen Core 0 und Core 2 zwar mehr Zeit bei dem lesenden Zugriff, jedoch kann Core 1 entlastet werden.

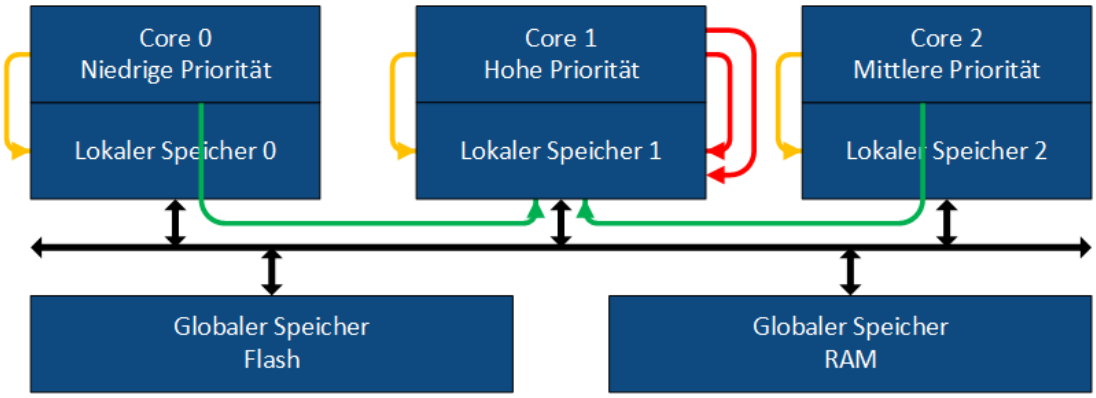

Abbildung 4: Intercore-Kommunikation durch Core 1

In dem dritten Szenario, welches in Abbildung 5 vorgestellt wird, werden Daten von Core 2 berechnet, welche Core 0 und Core 1 für die weitere Verarbeitung benötigen. Auf Grund der Tatsache, dass Core 2 die mittlere Priorität in diesem System besitzt, werden die Werte für Core 1 direkt in dessen Speicher geschrieben, da dieser eine höhere Priorität besitzt. Die Werte für Core 0 legt Core 2 in seinem eigenen lokalen Speicher ab. Durch diese Verteilung benötigt Core 0 zwar mehr Zeit für den lesenden Zugriff, jedoch wird Core 1 durch den schnelleren Zugriff auf seinen eigenen Speicher entlastet.

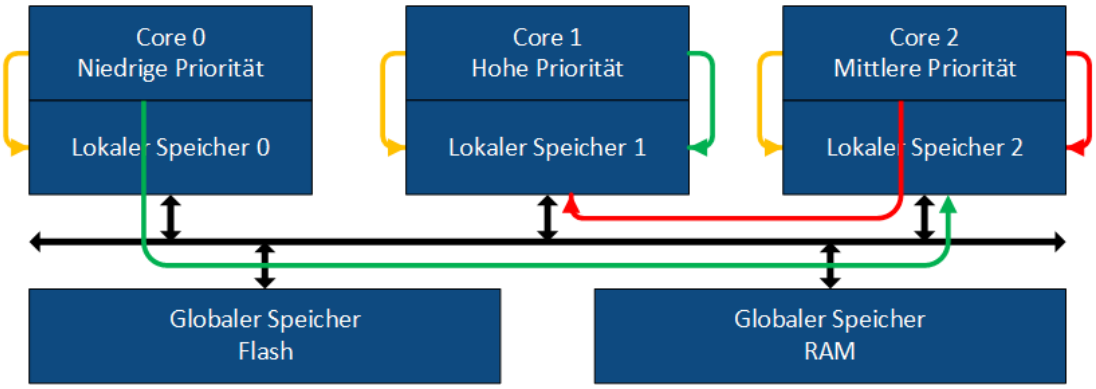

Abbildung 5: Intercore-Kommunikation durch Core 2

Wie in den aufgezeigten Anwendungsfällen in Abbildung 3 bis 5 zu sehen ist, nutzen die Prozessorkerne ihre lokalen Speicher sowohl zur Intercore-Kommunikation als auch für Berechnungen, welche exklusiv auf einem Kern durchgeführt werden. Durch diesen Umstand entsteht ein konkurrierender Zugriff, wodurch Wartezyklen entstehen, welche die Echtzeitfähigkeit beeinflussen können. Aus diesem Grund haben einige Prozessorhersteller begonnen, zwei lokale Speicher pro Kern zu integrieren, wodurch die Intercore-Kommunikation über einen separaten Speicher erfolgen kann. Die Abbildung 6 zeigt die schematische Darstellung der erweiterten Speicherhierarchie [4].

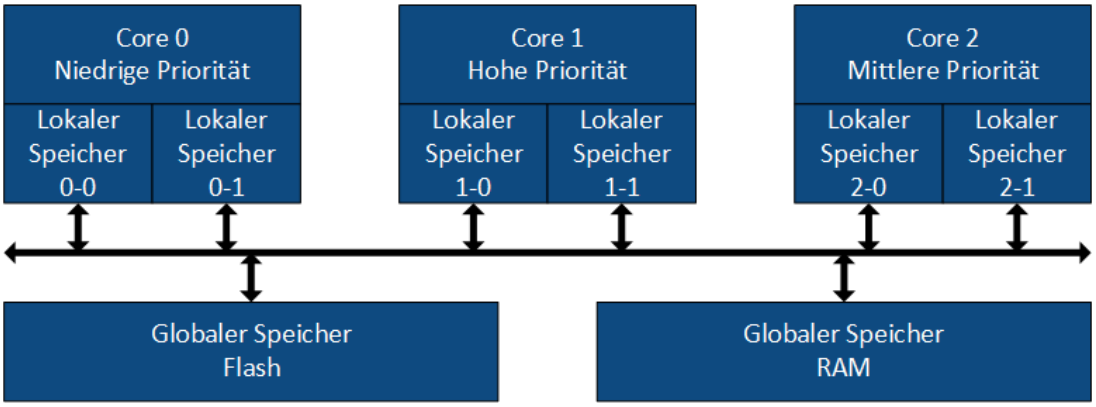

Abbildung 6: Schematischer Aufbau eines embedded Multicore-Mikrocontrollers mit drei Kernen und unterschiedlichen Prioritäten sowie jeweils zwei lokalen Speichern [4]

In dem folgenden Szenario, welches in Abbildung 7 dargestellt wird, stellt Core 0 Werte zur Verfügung, welche von Core 1 und Core 2 benötigt werden. Auch in dieser Konfiguration besitzt Core 0 die niedrigste Priorität und schreibt daher seine Werte in die Speicher von Core 1 und Core 2. Der Unterschied zu dem ersten Anwendungsfall besteht darin, dass die lokalen Speicher X-0 exklusiv von den Prozessorkernen und die Speicher X-1 für die Intercore-Kommunikation genutzt werden. Dadurch können konkurrierende Zugriffe bei der Bearbeitung der exklusiven Aufgaben effektiv verhindert werden.

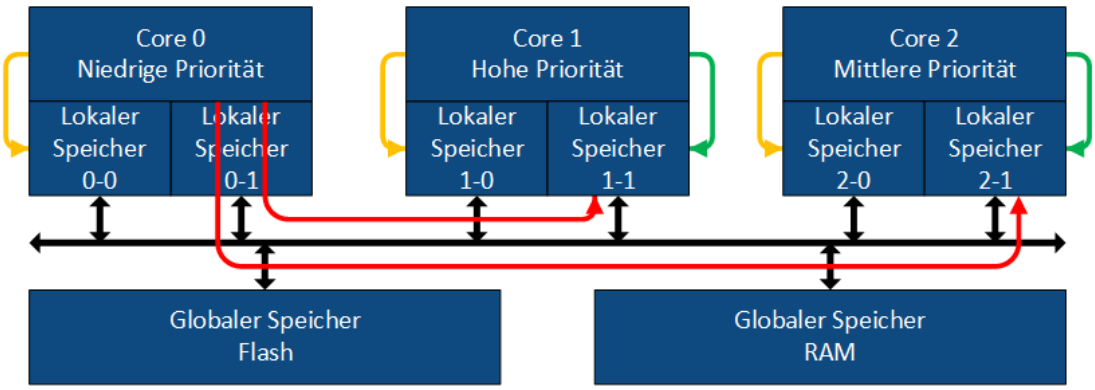

Abbildung 7: Intercore-Kommunikation durch Core 0 mit zwei lokalen Speichern

Da Core 1 in dem Beispiel die höchste Priorität besitzt, schreibt dieser seine Werte für Core 0 und Core 2 in den lokalen Speicher 1-1. Dies erhöht zwar die Dauer des Lesezugriffs für Core 0 und Core 2, reduziert jedoch die Dauer des Schreibvorgangs für Core 1. Bedingt durch die zwei vorhandenen Speicher erfolgt auch hier eine Aufteilung für exklusive Berechnungen und Intercore-Kommunikation zur Reduzierung von konkurrierenden Zugriffen.

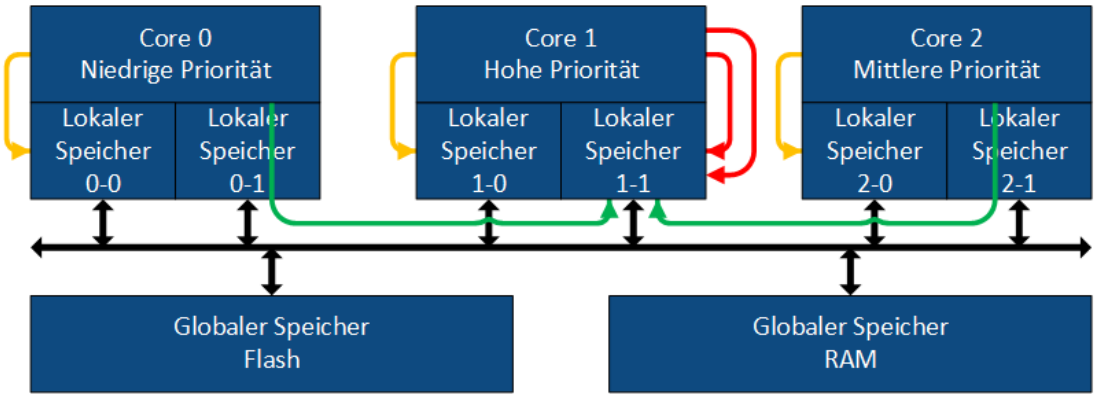

Abbildung 8: Intercore-Kommunikation durch Core 1 mit zwei lokalen Speichern

In dem nächsten Szenario aus Abbildung 9 werden die Daten von Core 2 bereitgestellt. Da Core 2 die mittlere Priorität in diesem System besitzt, schreibt dieser die Werte für Core 1 in dessen lokalen Speicher auf Grund der höheren Priorität. Die Werte für Core 0 schreibt Core 2 in seinen eigenen lokalen Speicher. Bedingt durch die erweiterte Speicherhierarchie kann auch hier eine Aufteilung der Speicher in ex-

klusive und geteilte Variablen erfolgen, wodurch konkurrierende Zugriffe bedingt durch die Intercore-Kommunikation verringert werden können.

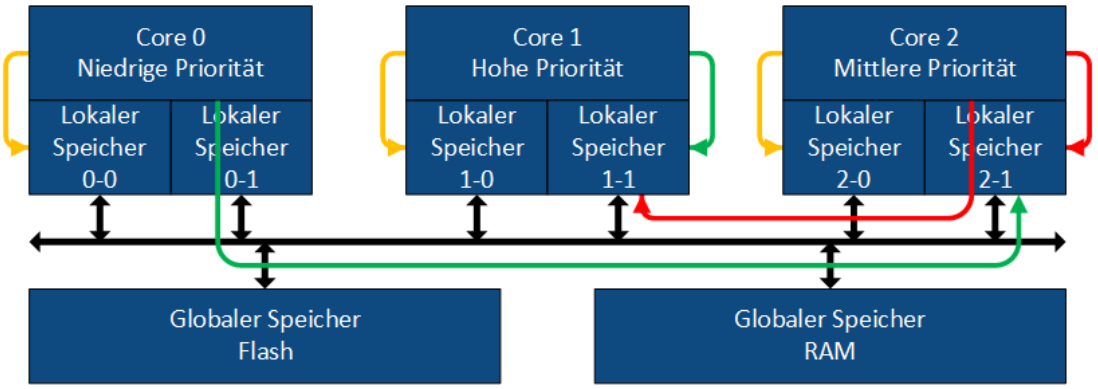

Abbildung 9: Intercore-Kommunikation durch Core 2 mit zwei lokalen Speichern

Eine Erweiterung dieses Systems kann durch den DMA-Controller erfolgen, welcher in den meisten modernen embedded Multicore-Mikrocontrollern integriert ist. Dadurch kann jeder Prozessorkern seinen eigenen lokalen Speicher nutzen und der DMA-Controller übernimmt den Kopiervorgang in den lokalen Speicher eines anderen Kerns. Abbildung 10 verdeutlicht die Separierung, wobei hier nur die Speicherbereiche für die Intercore-Kommunikation dargestellt sind.

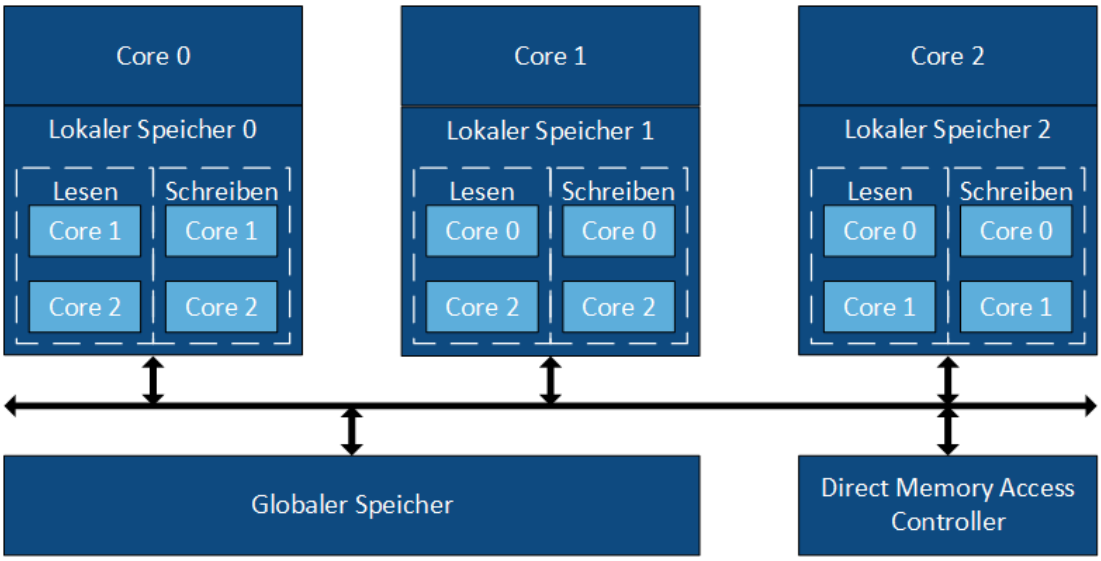

Abbildung 10: Schematischer Aufbau eines embedded Multicore-Mikrocontrollers mit drei Kernen und separaten Speicherbereichen zur Intercore-Kommunikation für jeden Kommunikationsteilnehmer

In dem folgenden Anwendungsbeispiel stellt Core 1 Daten für Core 2 bereit. Core 1 schreibt die aktualisierten Werte in seinen lokalen Speicher mit der maximalen Geschwindigkeit. In Anschluss wird der DMA-Controller aktiviert, welcher die neuen Daten in den lokalen Speicher von Core 2 schreibt. Durch die Nutzung des DMA-

Controllers kann auch Core 2 mit voller Geschwindigkeit auf seinen eigenen lokalen Speicher zugreifen.

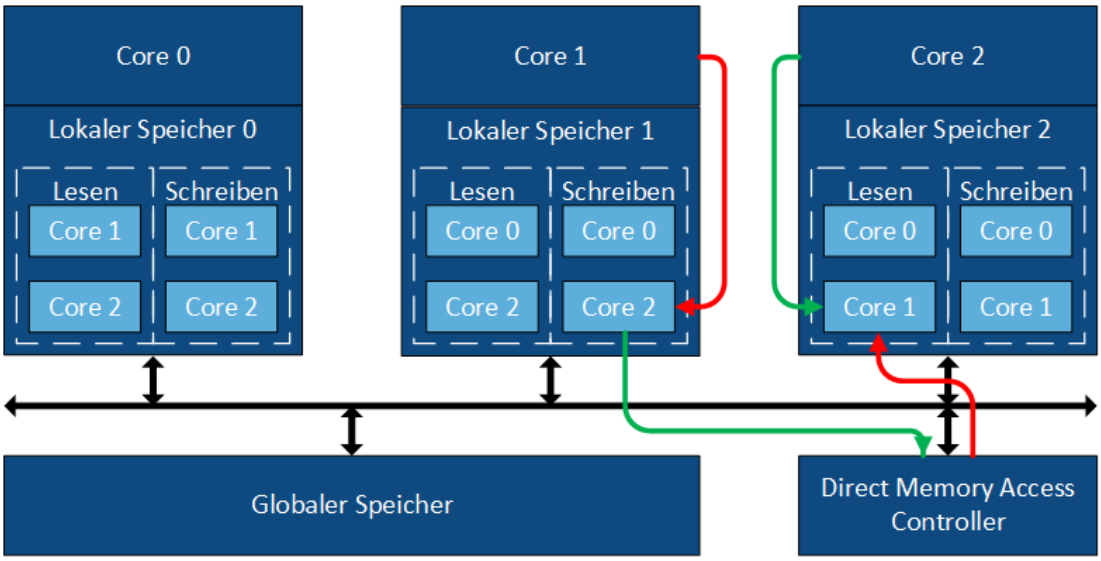

Abbildung 11: Intercore-Kommunikation durch den DMA-Controller

Auch bei diesem Ansatz bieten sich embedded Multicore-Mikrocontroller mit zwei lokalen Speichern pro Prozessorkern an, da sonst auch in diesem Fall ein konkurrierender Zugriff durch den DMA-Controller und dem jeweiligen Prozessorkern bei einer Operation auf den lokalen Speicher erfolgen kann.

4 Versuchsaufbau

Die Überprüfung des in diesem Artikel vorgestellten Konzepts erfolgt mit zwei Evaluierungsboards der Firma hitex. Eines nutzt den Infineon AURIX TC277 der ersten Generation und das andere ist mit dem AURIX TC397 der zweiten Generation bestückt. Für das Flashen und Debuggen sowie zum Auslesen der Messwerte wird ein Debugger der Firma Lauterbach genutzt. Die genaue Bezeichnung aller Versuchswerkzeuge kann der Tabelle 1 entnommen werden.

Tabelle 1: Versuchswerkzeuge

Mikrocontroller	**Evaluierungsboard**	**Taktfrequenz**
SAK-TC277TF-64F200S CA ES	TriBoard TC2X7 V1.0	200 MHz
SAK-TC397XE-256F300S AA EES	TriBoard TC3X7 TH V1.0	300 MHz
Bezeichnung	**Verwendung**	
Lauterbach Power Debug Interface / USB3; TRACE32 R2016	Flash-Adapter, Debugger	

In der folgenden Tabelle 2 wird die genutzte Software mit den dazugehörigen Versionsnummern sowie die Verwendung detailliert aufgeschlüsselt.

Tabelle 2: Verwendete Versuchs-Software

Bezeichnung	**Verwendung**
TASKING VX-toolset for TriCore v6r2	Compiler, Linker
Infineon Software Framework	Entwicklungs-Framework
Infineon Low Level Driver 1.0.0.12.0	Mikrocontrollertreiber

5 Resultate

Zur Validierung des oben beschriebenen Konzeptes wird in jeder Messung ein 4KB großes Array mit 32-Bit Werten zwischen den Prozessorkernen ausgetauscht. Die Zeitmessung erfolgt mit dem internen Performance-Countern der Prozessorkerne, welche taktgenau die benötigte Zeit sowie die ausgeführten Instruktionen ermitteln. Weiterhin, zur gezielten Provokation von konkurrierenden Zugriffen, werden die Kopiervorgänge synchron mittels eines Broadcast-Interrupts ausgelöst.

5.1 Infineon AURIX TC277

Um das Konzept zu überprüfen wurde das Verfahren auf einen embedded Multicore-Mikrocontroller der ersten AURIX-Generation portiert. Das Derivat vom Typ TC277 nutzt drei Prozessorkerne, welche jeweils einen lokalen Speicher für besonders schnelle Zugriffe zur Verfügung stellen. Zusätzlich ist in dem TC277 ein globaler RAM Speicher integriert, auf welchen alle Kerne mit derselben Geschwindigkeit zugreifen können [13].
In der ersten Messreihe wird untersucht, inwieweit sich die Speichernutzung auf die Kopierdauer auswirkt. Dazu kopiert Core 1 in jeder Messung ein 4KB großes Array wechselseitig zwischen den unterschiedlichen Speichern des TC277. Um in dieser Messreihe konkurrierende Zugriff zu vermeiden, sind Core 0, Core 2 und der DMA-Controller deaktiviert.

Tabelle 3: Speicherperformance bei einem Kopiervorgang, Core 1 aktiv

Quelle:	**Ziel:**	**Ticks:**	**Instruktionen:**
Lokaler Speicher 1	Lokaler Speicher 1	2076	3082
Lokaler Speicher 1	Globaler Speicher	12281	3082
Globaler Speicher	Lokaler Speicher 1	11287	3082
Globaler Speicher	Globaler Speicher	22529	3082
Lokaler Speicher 1	Lokaler Speicher 2	8197	3082
Lokaler Speicher 2	Lokaler Speicher 1	719	3082
Lokaler Speicher 2	Lokaler Speicher 2	14345	3082

Tabelle 3 zeigt, dass der Zugriff von Core 1 auf seinen eigenen lokalen Speicher deutlich schneller ist als der Zugriff auf die globalen Speicher. Des Weiteren zeigt die Messung, dass der Zugriff auf die lokalen Speicher eines anderen Prozessorkerns

ebenfalls deutlich schneller ist als eine Operation auf dem globalen Speicher. Die Anzahl der benötigten Instruktionen ist unabhängig von Quelle und Ziel des Kopiervorgangs. Die Dauer hängt von dem Speicher selbst und dessen Anbindung an den Prozessorkern ab.
Für die zweite Messreihe wird ein konkurrierender Zugriff auf den lokalen Speicher von Core 1 simuliert. Zu diesem Zweck kopieren Core 0 und Core 2 ein 4KB großes Array aus dem lokalen Speicher von Core 1 in ihren eigenen. Diese Konfiguration entspricht dem Szenario aus Abbildung 8.

Tabelle 4: Speicherperformance Core 1 bei konkurrierendem Zugriff durch Core 0 / 2

Quelle:	**Ziel:**	**Ticks:**	**Instruktionen:**
Lokaler Speicher 1	Lokaler Speicher 1	4121	3082
Lokaler Speicher 1	Globaler Speicher	12283	3082
Globaler Speicher	Lokaler Speicher 1	11288	3082
Globaler Speicher	Globaler Speicher	22530	3082
Lokaler Speicher 1	Lokaler Speicher 2	8199	3082
Lokaler Speicher 2	Lokaler Speicher 1	7197	3082
Lokaler Speicher 2	Lokaler Speicher 2	14353	3082

Wie in der Messreihe in Tabelle 4 zu sehen ist, ist eine signifikante Änderung der Kopierdauer im Vergleich zur Tabelle 3 lediglich bei der ersten Messung von dem lokalen Speicher 1 in den lokalen Speicher 1 zu beobachten. Die anderen Messungen sind von dem konkurrierenden Zugriff nicht betroffen. Jedoch zeigt dies deutlich, dass durch die Intercore-Kommunikation auch Berechnungen betroffen sind, welche auf Core 1 exklusiv ausgeführt werden. Dies sollte bei einer Bewertung der Echtzeitfähigkeit eines Systems zwingend berücksichtigt werden.

In der dritten Messreihe, welche in der Tabelle 5 zu sehen ist und der Abbildung 11 entspricht, wird der DMA-Controller zur Intercore-Kommunikation genutzt. Hierbei zeigt sich ebenfalls, dass der Zugriff auf die globalen Speicher langsamer ist, jedoch ist die Differenz zu den lokalen Speichern deutlich geringer im Vergleich zur Verwendung der Prozessorkerne. Die Anzahl der Instruktionen ergeben sich aus dem Aktivieren des DMA-Transfers durch Core 1.

Tabelle 5: Speicherperformance DMA-Controller

Quelle:	**Ziel:**	**Ticks:**	**Instruktionen:**
Lokaler Speicher 1	Lokaler Speicher 1	2971	17
Lokaler Speicher 1	Globaler Speicher	3483	17
Globaler Speicher	Lokaler Speicher 1	3483	17
Globaler Speicher	Globaler Speicher	3995	17
Lokaler Speicher 1	Lokaler Speicher 2	2971	17
Lokaler Speicher 2	Lokaler Speicher 1	2971	17
Lokaler Speicher 2	Lokaler Speicher 2	2971	17

5.2 Infineon AURIX TC397

Mit der zweiten Generation der AURIX Mikrocontrollerfamilie hat Infineon die Anzahl der Prozessorkerne sowie die Speicherhierarchie im Vergleich zur ersten Generation deutlich überarbeitet. Dazu gehört, dass die maximale Anzahl an Prozessorkernen auf sechs verdoppelt wurde und das jedem Core jetzt zwei lokale Speicher zur Verfügung stehen, wodurch konkurrierende Zugriffe bei der Intercore-Kommunikation minimiert werden können. Für die Messung in diesem Artikel wird der Infineon AURIX TC397 genutzt, welcher sechs Kerne mit jeweils zwei lokalen Speichern bietet. Zusätzlich hat Infineon die Anzahl der globalen RAM-Speicher insgesamt auf vier erhöht. Da diese globalen Speicher jedoch eine untergeordnete Rolle in dem hier vorgestellten Konzept einnehmen, dienen diese lediglich als Referenz [12].

Die Messreihe in Tabelle 6 zeigt, dass der Zugriff auf den lokalen Speicher 1-0 identisch zur ersten AURIX-Generation ist. Auch der Zugriff auf den zweiten lokalen Speicher 1-1 verhält sich äquivalent zu den Messungen für den lokalen Speicher 1-0. Eine Ausnahme stellt ausschließlich der Kopiervorgang dar, bei welchem sowohl die Quelle als auch das Ziel der lokale Speicher 1-1 ist. Dieser Speicher scheint lediglich eine Anbindung an den Core 1 zu besitzen, wodurch parallele Zugriffe deutlich ausgebremst werden. Beim Vergleich von Tabelle 3 und Tabelle 6 ist zu erkennen, dass der Kopiervorgang beim AURIX 2G in Bezug auf die globalen Speicher weniger Ticks benötigt. Die Erklärung steckt in der Verbesserung der Anbindung der globalen Speicher durch Infineon. Bei der zweiten AURIX Generation entspricht die Zugriffsgeschwindigkeit eines Kerns auf die globalen Speicher somit der Operationsgeschwindigkeit des Zugriffs auf die lokalen Speicher der anderen Kerne.

Tabelle 6: Speicherperformance Core 1

Quelle:	**Ziel:**	**Ticks:**	**Instruktionen:**
Lokaler Speicher 1-0	Lokaler Speicher 1-0	2076	3082
Lokaler Speicher 1-0	Lokaler Speicher 1-1	2076	3082
Lokaler Speicher 1-1	Lokaler Speicher 1-0	2076	3082
Lokaler Speicher 1-1	Lokaler Speicher 1-1	4113	3082
Lokaler Speicher 1-0	Globaler Speicher	2076	3082
Globaler Speicher	Lokaler Speicher 1-0	9260	3082
Globaler Speicher	Globaler Speicher	9239	3082
Lokaler Speicher 1-0	Lokaler Speicher 2-0	2076	3082
Lokaler Speicher 2-0	Lokaler Speicher 1-0	9239	3082
Lokaler Speicher 2-0	Lokaler Speicher 2-0	9239	3082

6 Diskussion

Seit der Einführung der ersten embedded Multicore-Mikrocontroller für Steuergeräte mit harter Echtzeitanforderung wurde eine Leistungssteigerung hauptsächlich über die Erhöhung der vorhandenen Prozessorkerne realisiert. Leider können die bestehenden Algorithmen häufig nicht so separiert werden, dass diese völlig autark auf verschiedenen Prozessorkernen gerechnet werden. Aus diesem Grund wird Intercore-Kommunikation in den nächsten Jahren und bei steigender Anzahl von Prozessorkernen zunehmend zur Herausforderung werden. Dabei stellt sowohl die Zugriffsgeschwindigkeit auf die geteilten Speicher als auch die Effekte der konkurrierenden Zugriffe ein Problem dar.

Das in diesem Artikel präsentierte Konzept zur effektiven Nutzung der vorhandenen Speicherhierarchie stellt dabei einen ersten Schritt dar. Durch die Priorisierung der Prozessorkerne können die Wartezyklen bei der Intercore-Kommunikation so verteilt werden, dass höher priorisierte Kerne gezielt entlastet werden. Des Weiteren werden durch die Verteilung der auszutauschenden Daten auf verschiedene Speicher die Effekte von konkurrierenden Zugriffen deutlich verringert.

Die zukünftige Entwicklung sieht vor, dass das Konzept in ein Framework überführt wird, welches die Zuordnung der geteilten Daten auf die vorhandenen Speicher automatisch vornimmt. Zu diesem Zweck sollen die Trace-Daten von Steuergeräten analysiert und die geteilten Werte sowie die dazugehörigen Kommunikationsteilnehmer selbstständig extrahiert werden. Des Weiteren wird dem Framework eine allgemeine Beschreibung des embedded Multicore-Mikrocontrollers mit seiner Speicherhierarchie zur Verfügung gestellt. Mit diesen Informationen kann dann eine optimale Verteilung errechnet werden. Ein weiteres Ziel ist die Integration des LET-Paradigmas für die jeweiligen Speicher. Dadurch können die Effekte von konkurrierenden Zugriffen besser geplant und ihre Auswirkungen auf die Echtzeitfähigkeit minimiert werden, wodurch die Worst-Case-Execution Time verringert wird [8][9][14][15].

Literatur- und Quellenverzeichnis

[1] Abbi Ashok and Jens Harnisch. 2017. AURIX - Programming close to hardware for best performance. In Embedded Multi-Core Conference. Infineon Technologies AG, 81726 Munich, Germany.

[2] Infineon Technologies AG 2014. AURIX TC27x C-Step User's Manual V2.2. Infineon Technologies AG, 81726 Munich, Germany.

[3] Infineon Technologies AG 2014. AURIX TC29x B-Step User's Manual V1.3. Infineon Technologies AG, 81726 Munich, Germany.

[4] Infineon Technologies AG 2016. AURIX TC3xx Target Specification V2.0.1. Infineon Technologies AG, 81726 Munich, Germany.

[5] Thomas Barth and Peter Fromm. 2016. Warp 3 zwischen allen Kernen - Entwicklung einer schnellen und sicheren Multicore-RTE. In Tagungsband Embedded Software Engineering Kongress 2016.

[6] Günther Bengel, Christian Baun, Marcel Kunze, and Karl-Uwe Stucky. 2015. Masterkurs Parallele und Verteilte Systeme. Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-8348-2151-5

[7] Hartmut Ernst, Jochen Schmidt, and Gerd Beneken. 2016. Grundkurs Informatik. Springer Fachmedien Wiesbaden. https://doi.org/10.1007/978-3-658-14634-4

[8] Christoph M Kirsch and Ana Sokolova. 2012. The Logical Execution Time Paradigm. In Advances in Real-Time Systems. Springer, 103–120.

[9] Florian Kluge, Martin Schoeberl, and Theo Ungerer. 2016. Support for the Logical Execution Time Model on a Time-predictable Multicore Processor. SIGBED Rev. 13, 4 (Nov. 2016), 61–66. https://doi.org/10.1145/3015037.3015047

[10] Philipp Jungklass and Mladen Berekovic. 2018. Effects of concurrent access to embedded multicore microcontrollers with hard real-time demands. 13th International Symposium on Industrial Embedded Systems (2018).

[11] Philipp Jungklass and Mladen Berekovic. 2018. Performance-Oriented Memory Management for Embedded Multicore Microcontrollers. In 26th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing.

[12] Infineon Technologies AG 2015. TriCore TC1.6.2 Core Architecture. Infineon Technologies AG, 81726 Munich, Germany.

[13] Infineon Technologies AG 2012. TriCore TC1.6P & TC1.6E Core Architecture. Infineon Technologies AG, 81726 Munich, Germany.

[14] Florian Kluge, Mike Gerdes, and Theo Ungerer. 2014. An Operating System for Safety-Critical Applications on Manycore Processors. 2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (2014), 238–245.

[15] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. 2012. Memory-centric Scheduling for Multicore Hard Real-time Systems. Real- Time Syst. 48, 6 (Nov. 2012), 681–715. https://doi.org/10.1007/s11241-012-9158-9

Autor

M.Sc. Philipp Jungklass studierte an der Hochschule Stralsund Informatik und arbeitet seit vielen Jahren als Entwicklungsingenieur im automobilen Sektor. Seine berufliche Tätigkeit begann mit der Treiberprogrammierung für Kommunikationssysteme im Fahrzeug. Aktuell beschäftigt er sich mit Multicore-Mikrocontrollern in sicherheitskritischen Anwendungen und ist für die ausbildungsrelevante Betreuung zuständig. Zudem berichtet er regelmäßig über sein Arbeitsfeld bei Symposien im embedded Bereich.

Kontakt

Email: philipp.jungklass@iav.de