HIERARCHISCHES ASYNCHRONES MULTICORE-SCHEDULING IN HOCHINTEGRIERTEN SOFTWARE-SYSTEMEN

M. Ernst, A. Frey
Technische Hochschule Ingolstadt, Esplanade 10, 85049 Ingolstadt, Germany

Zusammenfassung

In der Vergangenheit wurde für jede neue Funktion in Flugzeugen sowie Fahrzeugen ein eigenes, neues Steuergerät entworfen und entsprechend verbaut. Dies führt zu einer rapide ansteigenden Zahl von Funktionen und Steuergeräten. Damit die Anzahl der Steuergeräte überschaubar gehalten werden kann, gibt es zwei Schritte der Effizienzsteigerung und Kostenreduktion. Der erste Schritt ist, diese Steuergeräte zu einem einzigen Steuergerät mit einem Multicore Prozessor zusammen zu fassen. Der zweite Schritt ist die Integration der Funktionen auf diesem System. Um den Nutzungsgrad eines Multicore-Prozessors weiter zu verbessern ist eine Alternative zu den bislang statisch konfigurierten Softwaresystemen erforderlich. Im folgenden Paper soll ein neues Konzept, eines zur Laufzeit rekonfigurierbaren Software-Systems für Embedded-Echtzeit-Betriebssysteme von Multicore-Prozessoren vorgestellt werden. Dieses Software-System kann durch die Berücksichtigung der phasenabhängigen Rechenzeit der Funktionen den Nutzungsgrad der Rechenleistung von hochintegrierten Steuergeräten weiter steigern.

1. EINLEITUNG

Motivation In Flugzeugen und Fahrzeugen werden dem Piloten bzw. dem Fahrer immer mehr softwarebasierte Hilfs- und Zusatz-Funktionen zur Verfügung gestellt. Dies führt zu einer rapide ansteigenden Zahl von Funktionen und Steuergeräten. Um Kosten zu reduzieren musste der Anstieg der Anzahl der Steuergeräte gestoppt bzw. die Anzahl der Steuergeräte verringert werden. Ein wichtiger Schritt war es Steuergeräte zu einem einzigen Steuergerät mit einem Multicore-Prozessor zusammen zu fassen und die konsequente Integration der einzelnen Funktionen auf diesen Systemen (Hochintegration).

Ein Ansatz für die Hochintegration der Funktionen, ist es ein gemeinsames Betriebssystem für das komplette Multicore-System zu verwenden. Im Vorfeld wird ein statischer Verteilungsplan der Anwendungen offline bestimmt und auf die einzelnen Prozessorkerne verteilt. Dadurch bleibt die Ausführungszeit der Funktionen welche in einem vorherbestimmten Timeslot auf dem Kern verankert sind, wie in einem statisch geplantem Singlecore-System vorher-berechenbar und erspart Rechenzeit. In der Luft und Raumfahrt macht sich der *ARINC653* –Standard [1] diese Trennung in Partitionen zunutze und beschreibt die Trennung von Software und Hardware-Ressourcen (Rechenzeit, Hauptspeicher, …) für einzelne Tasks auf demselben Prozessor. Dabei wird den Tasks statisch Rechenzeit zugesprochen obwohl diese gerade keine Rechenzeit benötigen. Zum Beispiel werden in einer statischen Konfiguration zum Beispiel durch den Parkassistenten während der Phase "Autobahn" oder durch das Spurhaltesystem während der Phase "Parken" unnötig Ressourcen als auch Rechenleistung belastet oder blockiert. Diese Phasen schließen sich gegenseitig aus (BILD 1).

Ein weiterer Ansatz für die Hochintegration von Funktionen auf Multicore-Systemen ist es dieses System in seine einzelnen Kerne aufzuteilen und auf jedem dieser Kerne ein eigenes und völlig unabhängiges Betriebssystem zu betreiben. Dadurch geht allerdings die zeitliche Synchronisation der virtualisierten Systeme zueinander verloren. Eine Migration von Tasks zwischen den Systemen ist nicht möglich. Kommunikation zwischen den einzelnen Tasks kann auch nur noch bedingt über einen virtuellen Netzwerktreiber oder speziell allokierten geteilten Speicher stattfinden. [2]

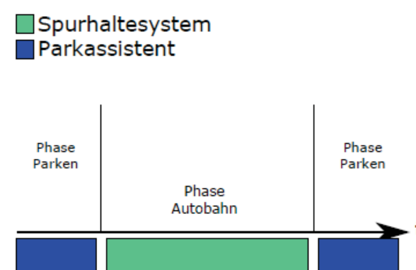

BILD 1. gegenseitiger Ausschluss von Phasen

Ziel. Eine Möglichkeit die Kosten im Flugzeug oder Automobil weiter zu reduzieren, ist es die Rechenzeit des vorhandenen Multicore-Prozessors effizienter zu nutzen. Um dieses Ziel zu erreichen, wird in diesem Paper das Konzept des Hierarchical Asynchronous Multicore Scheduler (HAMS) vorgestellt. Kern des Konzeptes ist die Betrachtung von sich gegenseitigen ausschließenden Phasen (BILD 1). Der HAMS ist ein in hierarchischen Schichten unterteilter Scheduler der das Softwaresystem phasenabhängig dynamisch rekonfigurieren kann. Hierbei bleibt das Softwaresystem zu jeder Zeit statisch vorher-berechenbar und kann somit auch in echtzeitfähigen Betriebssystemen eingesetzt werden.

2. SCHEDULING GRUNDLAGEN

2.1. Multicore Scheduling

Multicore Scheduling kann in zwei Bereiche unterteilt werden, *global scheduling* und p*artitioned scheduling* [3].

Global Scheduling beschreibt den Ansatz der in allen heutigen Desktop PCs zu finden ist. Es gibt eine systemweite Liste von Tasks die sich im ready-Status befinden. Diese Tasks können auf jedem beliebigen Kern des Multicore Systems verplant werden. Eine Migration der Tasks zwischen den Kernen kann jederzeit stattfinden um die Systemlast auszubalancieren [4].

Partitioned Scheduling beschreibt einen statischen Ansatz in welchem die einzelnen Tasks als sog. *scheduling objects* behandelt werden und als solche den zugewiesen Kern niemals verlassen dürfen. Der partitioned Scheduling-Ansatz teilt das Multicore-Gesamtsystem in viele separate Singlecore-Systeme auf. Sind dem System die laufenden Tasks bereits im Vorfeld bekannt, kann die Aufteilung offline vorberechnet werden. Danach wird jeder Kern separat geplant, sofern es keine Abhängigkeiten zwischen den Tasks gibt.

Hierarchical Scheduling bezeichnet einen in hierarchische Ebenen unterteilte Vermischung des partitioned- und global-Scheduling Ansatzes. Dafür wird eine Scheduler-Hierarchie erstellt, welche einem Baum ähnelt. Sie besitzt einen *global scheduler*, oft auch als Wurzelknoten bezeichnet. Je nach Anzahl der Prozessorkerne gibt es *client scheduler*, die Blätter des Baumes. Das Konzept besagt weiter, dass durch den globalen Scheduler alle Applikationen auf seine Clients verteilt werden. Danach wird deren Ausführungszeit durch die einzelnen Clients separat auf der Hardware geplant.

2.2. Related Work

Für hierarchisches Scheduleing für Echtzeit-Multicore-Systeme gibt es bislang nur wenige wissenschaftliche Veröffentlichungen.

Checconi et al. sowie das AQuoSA Framework verwenden sogenannte Constant Bandwidth Servers. [5] Diese Server sind einem Prozessorkern zugewiesen und können selbst nicht migriert werden. In diesen Server werden die einzelnen Tasks geplant. Diese können zwischen den Servern und somit auch auf einen anderen Prozessorkern migrieren. Dadurch geht allerdings die Berechenbarkeit der Tasks verloren.

Åsberg et al. verwendet ebenfalls einen hierarchischen Ansatz und erweitert den oben genannten Serveransatz um einen hierarchischen globalen Scheduler [3]. Allerdings sind seine Server und zusätzlich auch die Tasks fix einem Kern zugewiesen und können während der Laufzeit nicht migriert werden.

Der nächste logische Schritt für eine Erweiterung dieser Ansätze ergibt sich aus der genaueren Betrachtung der Tasks und dessen Phasen.

3. DYNAMISCHE PHASENABHÄNGIGE REKONFIGURATION

Um das Konzept des HAMS besser verstehen zu können, müssen als erstes Systemphasen, Phasenübergänge und der gegenseitige Ausschluss von Phasen betrachtet werden.

3.1. Systemphasen und Phasenübergänge

Jedes softwaregetriebene System kann über ein Phasenmodel mit x Systemphasen abgebildet werden. Wir definieren eine Systemphase als einen zeitlichen Abschnitt in dem sich eine Funktion zum Zeitpunkt *t* in einem fest definierten Zustand befindet. Eine einzelne Funktion kann ähnlich einem gerichteten Zustandsautomaten zwischen beliebig vielen Phasen wechseln, sich jedoch zu einer bestimmten Zeit *t* in nur einer Phase befinden. Beliebig viele Funktionen können sich gleichzeitig in einer bestimmten Phase befinden.
Ein Phasenübergang kann nach einem oder mehreren bestimmten Ereignissen *e* stattfinden und ist der dafür vorhergesehenen Funktion bekannt. Eine Funktion kann durch Analyse der Ereignisse einen Phasenübergang erkennen und in eine andere Phase wechseln. Haben 2 oder mehrere Funktionen exakt die gleichen Phasenübergänge und gehören exakt denselben Phasen an, können sie zu *Funktionsets* zusammengefasst werden.

3.2. Beispiel

Die dynamische phasenabhängige Rekonfiguration soll anhand eines Beispiels aufgezeigt werden.
Gegeben seien 3 Tasks T_A, T_B und T_C welche in der folgenden Tabelle definiert sind und ein DualCore System. Aus Übersichtlichkeitsgründen wird eine zyklische Wiederholung von 30 *ms* angenommen, welche nicht weiter betrachtet werden muss. Für jede Phase (P_1 und P_2) werden verschiedene Worst Case Execution Times (WCET_P_1 & 2) angenommen. Ferner benötigt der Beispiel-Task T_A in Phase 2 mehr Rechenzeit während Task T_B in Phase 2 sich komplett abschalten lässt, bzw. nicht benötigt wird. Zusätzlich schließen die Phasen 1 und 2 sich gegenseitig aus, dh. sie können nicht zeitgleich aktiv sein. (Beispiel: „Tag“ und „Nacht“).

Taskname	WCET_P_1	WCET_P_2
T_A	5	20
T_B	10	0
T_C	5	10

In einem statisch geplanten System (*partitioned scheduling*) müsste die WCET der längsten Phase jedes Tasks verwendet werden. Daraus ergibt sich eine aufaddierte WECT von:

$$WCET_{ges} = max_{T_A}(5,20) + max_{T_B}(10,0) + max_{T_C}(5,10) \quad (1)$$

$$WCET_{ges} = 20ms + 10ms + 10ms = 40ms \quad (2)$$

Da die errechneten 40ms die gegebene Zykluszeit von 30ms überschreitet, muss auf den 2ten Kern des DualCores ausgewichen werden.

In einem phasenabhängigen System würde abhängig von der jeweiligen Systemphase eine eigene Konfiguration erstellt:

$$P1 : 5ms + 10ms + 5ms = 25ms \quad (3)$$

$$P2 : 20s + 0ms + 10ms = 30ms \quad (4)$$

Dadurch wird die gegebene Zykluszeit von 30 ms nicht überschritten. Alle Tasks können somit auf nur einem Kern geplant werden. So könnten die Hardwarekosten mithilfe eines preiswerteren Prozessors verringert werden oder alternativ die neu gewonnene Kapazitäten anderen Funktionen zur Verfügung gestellt werden.

3.3. Beweis

Die Gesamtanzahl der Tasks (*NumTasks$_{multiOS}$*) in einem statisch geplanten Multicore Software System, welches für jeden Prozessorkern ein eigenes separates Betriebssystem (OS) verwendet, kann wie folgt vereinfacht abgebildet werden. Hierbei steht T_n für alle benutzerspezifizierten Tasks.

$$NumTasks_{multiOS} = T_{OS} * Num_{Cores} + T_n \quad (5)$$

In einem hierarchisch geplantem Betriebssystem welches alle zur Verfügung gestellten Kerne verwaltet, müssen die OS-spezifischen Tasks T_{OS} nur einmal ausgeführt werden. Allerdings kommt ein weiterer Task für den globalen Scheduler (T_{gSched}) hinzu. Die Gesamtanzahl der laufenden OS-Tasks (*NumTasks$_{hierOS}$*) lässt sich wie folgt berechnen:

$$NumTasks_{hierOS} = T_{OS} * 1 + T_{gSched} + T_n \quad (6)$$

Lassen sich nun eine Anzahl Tasks T_m identifizieren die sich in Abhängigkeit ihrer Phasen gegenseitig ausschließen, so können diese Tasks zu einem Funktionsset zusammengefasst werden. Ein solches System, kann mit einem zusätzlichen globalen Scheduler Task T_{gSched}, die Anzahl der identischen OS Tasks T_{OS} weiter minimieren:

$$NumTasks_{HAMS} = T_{OS} * 1 + T_{gSched} + (T_n - \frac{T_m}{2}) \quad (7)$$

Da bei einem Vergleich zwischen statischem Scheduling und HAMS die verwendeten OS-Tasks zwingend erforderlich sind, können diese aus der Gleichung für eine bessere Lesbarkeit gegenseitig gekürzt werden.

$$T_n \geq T_{gSched} + (T_n - \frac{T_m}{2}) \quad (8)$$

Da der T_{gSched} nur dann aktiv ist, wenn ein Phasenübergang stattfindet kann dieser während der eigentlichen Tasklaufzeit vernachlässigt werden. So ergibt sich im statischen Betrieb bei einem direkten Vergleich:

$$T_n \geq (T_n - \frac{T_m}{2}) \quad (9)$$

Lassen sich nun keine gegenseitig ausschließenden Phasen finden verhält sich das System wie ein herkömmliches statisch konfiguriertes Softwaresystem.

Für

$$T_m \geq 2 \quad \&\& \quad T_n \geq 2 \quad (10)$$

gilt allerdings dass mehr Tasks im System untergebracht werden können als es in einem statisch geplanten System möglich wäre. Diese Überlegung bildet die Grundlage für den im nächsten Kapitel vorgestellten Scheduler.

4. HAMS

4.1. Übersicht

Im Folgenden soll nun das Konzept eines von Systemphasen abhängigen rekonfigurierbaren Software-Systems vorgestellt werden. Kern dieses Software-Systems ist der Hierarchische Asynchrone Multicore Scheduler (HAMS), der es ermöglicht das Echtzeitsystem während der Laufzeit phasenabhängig zu rekonfigurieren um damit die Rechenzeit effektiver nutzen zu können. Zuvor müssen einzelne Funktionen zu Funktionssets zusammengefasst und den Systemphasen zugeordnet werden, und zusätzlich für die spätere Rekonfiguration, sich gegenseitig ausschließende Phasen und Funktionssets bestimmt werden.

Um die bereits vorhandene logische Aufteilung und Unabhängigkeit von Multicore Prozessorkernen auf der Softwareebene zu nutzen, wird jedem Prozessor-Kern eine eigene unabhängige asynchrone Managementeinheit zugewiesen, der sogenannte First Level Scheduler (FLS). Jeder FLS plant durch die Verwendung von bereits etablierten Singlecore-Schedulingklassen (RMS, EDF, …) autonom seine ihm zugewiesenen Funktionen. Zusätzlich sammelt er permanent Leistungsparameter (LP) seines zugewiesenen Kerns. Alle im System befindenden FLS arbeiten asynchron zueinander. (BILD 2)

Über eine gepufferte asynchrone Kommunikationsschnittstelle (KS) überträgt jeder einzelne FLS unabhängig Leistungsparameter sowie Phasenübergänge der Funktionen. Phasenübergänge welche durch die laufenden Tasks selbst bestimmt.

Anschließend werden alle Daten an den globalen Scheduler, den Second Level Scheduler (SLS) übermittelt.

Ist es aufgrund eines Phasenwechsels durch den neu gegebenen Taskverteilungsplan erforderlich einen Task von einem Kern zu einem anderen zu migrieren, wird eine gemeinsame Zeitbasis benötigt um eine zeitlich korrekte Verschiebung zu gewährleisten. Diese ergibt sich durch eine gemeinsame globale verwendete zeitlich synchronisierte Betriebssystemzeit.

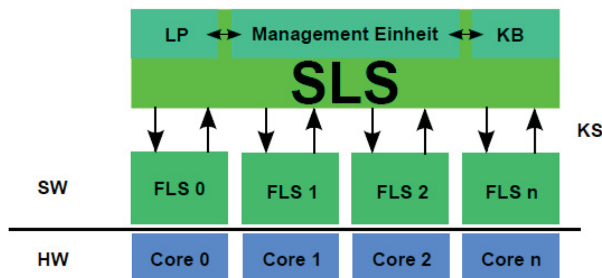

BILD 2. Logischer Aufbau von HAMS

4.2. Second Level Scheduler

Der Second Level Scheduler (SLS) überwacht und administriert als hierarchisch höchste Schedulinginstanz das Softwaresystem. Der SLS selbst besteht aus 3 Teilen:

Die Knowledgebase (KB) ist eine offline vorberechnete statische Datenbank. Sie beinhaltet alle auftretenden Phasen und die Verteilungspläne von Funktionen auf den einzelnen Prozessorkernen für alle möglichen Kombinationen von Phasen unter Berücksichtigung der Systemkonfiguration [6].

Die dynamische Datenkomponente speichert die aktuellen periodisch übertragenen Leistungsparameter (LP) der FLS und die aktuelle Phasenkonfiguration der Funktionen und stellt diese Daten zur Verfügung.

Durch die gezielte Verwendung der aktuellen Phasenkonfiguration und der in der Knowledgebase gespeicherten phasenabhängigen statischen Verteilungspläne kann nun das System durch die dritte Komponente, der *Management-Einheit*, dynamisch rekonfiguriert werden.

Der SLS-Task selbst wird periodisch, mit maximaler Priorität, auf einem, durch den aktuell geladenen statischen Verteilungsplan bestimmten Core (FLS) ausgeführt.
Im Folgenden sollen die drei Kernkomponenten des SLS genauer beschrieben werden.

4.2.1. Dynamische Datenkomponente

Der SLS überwacht alle ihm unterstellten FLS durch periodisch angeforderte Nachrichten über die Kommunikationsschnittstelle und speichert sie ab. Diese Nachrichten enthalten Leistungsparameter der einzelnen Prozessorkerne, wie dessen aktuelle Auslastung, die durchschnittlichen als auch die grenzwertigen WCETs von Tasks oder dessen unerlaubte Überschreitung, nicht durchführbare Konfigurationen und Hardwareausfälle. Zusätzlich werden durch die FLS Phasenänderungen des Softwaresystems übertragen und in der dynamischen Datenkomponente für eine spätere Auswertung zwischengespeichert.

4.2.2. Knowledgebase

Um die durch den SLS benötigte Rechenzeit für die Taskverteilung auf die einzelnen Kerne des Prozessors zu minimieren wurde diese Berechnung ausgelagert. Durch den Umstand, dass dem Entwickler des Gesamtsystems alle Tasks, Phasen und auftretende Ereignisse im Voraus bekannt sind, ist es möglich offline auf einem leistungsfähigen System alle auftretenden Funktions-Verteilungen im Voraus zu berechnen *[3]*.
Dafür wurde ein Knowledgebase-Tool entwickelt, welches Funktionen auf ihre logische Abhängigkeit hin überprüft und in unabhängige, komplementär rechnende, logische Funktionssets zusammengefasst. Diese Überprüfung findet in folgenden vier Schritten statt:

Als erstes werden die phasenabhängigen Echtzeitparameter der Funktionen betrachtet, z.B. WCETs, Abhängigkeiten innerhalb der Funktionssets, deren Ressourcenbedarf und deren Schedulingklasse.
Als zweites werden die phasenabhängigen Echtzeitparameter für das zugrundeliegende System evaluiert. Hierbei wird betrachtet welche Ressourcen zu welcher Zeit benötigt werden und ob Deadlines bei einem Rekonfigurationsvorgang eingehalten werden können.
Als drittes wird in einem Multicore-System die Funktionsverteilung auf die Cores untersucht, d.h. wann und wo welche Tasks auf dem Multi-Core System allokiert werden, wann der Wechsel von Funktionsset angestoßen werden kann und wie der Software-Fehlerfall behandelt werden muss.
Als letztes werden die Multicore-Systemparameter hinzugezogen, wie z.B. die Zeiten für die Inter-Core Kommunikation und sowie Migrationszeiten der Funktionen.

Danach werden die erstellten phasenabhängigen Verteilungspläne durch das offline-System in ein Konfigurationsfile exportiert, damit sie später auf dem Laufzeitsystem während des Startvorgangs importiert werden können. Der SLS muss dadurch die Funktions-Verteilung zur Laufzeit nicht selbst berechnen, er liest sie stattdessen aus dieser Datenbank.

4.2.3. Management-Einheit

Die Managementeinheit ist die zentrale logische Einheit des SLS und zuständig für die dynamische (Re)Konfiguration der Softwarefunktionen.
Im Falle eines Phasenüberganges einer Funktion wird der zugeordnete Verteilungsplan geladen. Dies geschieht in mehreren Schritten.
Der erste Schritt ist die Abarbeitung des Phasenüberganges. Dafür muss der SLS seine gespeicherte Phasenkonfiguration unter Betrachtung des neuen Phasenüberganges modifizieren.
Danach findet ein Abgleich der neuen Phasenkonfiguration mit allen Phasenkonfigurationen der statischen

Knowledgebase statt.
Als nächstes wird der an die gefundene Phasenkonfiguration gebundene statische Verteilungsplan aus der Knowledgebase in den SLS geladen.
Anschließend wird der gefundene Verteilungsplan durch die im SLS integrierte Management-Einheit unter zusätzlicher Beachtung der aktuellen gesammelten Leistungsparameter evaluiert. Das geschieht durch die aus der Knowledgebase bekannten WCETs der Funktionen und dessen gemittelten Funktionslaufzeiten aus den gesammelten dynamischen Leistungsparametern.
Ist der Verteilungsplan gültig, werden Befehle für die unterstellten FLS bestimmt und an die FLS asynchron über die Kommunikationsschnittstelle weitergeleitet. Dadurch kann der FLS die entsprechenden Funktionen laden bzw. entladen oder falls nötig komplette Funktionen auf einen anderen Prozessorkern migrieren können.
Für den Fall eines nicht durchführbaren Verteilungsplanes aufgrund unvorhergesehener Ereignisse oder von transienter Systemüberlast durch z.B. fehlerhafte Anwendungen, kann der SLS mithilfe von fest vordefinierten Notfall-Konfigurationen reagieren und diese laden.

4.3. First Level Scheduler

Bei HAMS sind jedem Kern ein eigener Sub-Scheduler sowie eine Schedulingklasse zugeteilt. Die Unterbrechungen des Systems durch den Scheduler müssen nicht mehr zwingend periodisch sein, sondern bestimmen sich durch die Schedulingklassen selbst. Die Unterbrechungen beschränken sich damit auf einen einzelnen Kern anstatt auf den gesamten Prozessor.

Der FLS selbst besitzt keine Kenntnisse über phasenübergang-auslösende Ereignisse. Das dafür notwendige logische Wissen oder die Sensordaten sind ausschließlich im dafür vorgesehenen Task verfügbar. Um dieses Problem zu lösen musste eine Möglichkeit gefunden werden, dem FLS diese Information mitteilen zu können. Dafür besitzt jeder Task in einem HAMS-System das Recht mit dem FLS unidirektional zu kommunizieren.
Die übermittelte Phasenänderung des Tasks wird vom FLS an den SLS weitergeleitet, um die dynamische Rekonfiguration des Komplett-Systems zu veranlassen. Zusätzlich zu der erkannten Phasenänderung sendet der FLS durch das Kommunikationssystem periodisch seine aktuellen Laufzeit-Daten und Statistiken (siehe oben).

4.4. Kommunikationsschicht

Dadurch dass in diesem zueinander asynchron laufenden System keine gemeinsamen Unterbrechungen der Kerne des Prozessors stattfinden, muss die Kommunikation zwischen SLS und FLS gepuffert und asynchron ablaufen. Bedingt durch das hierarchische Konzept dürfen die einzelnen FLS nicht miteinander kommunizieren. Sie gelten als autonome eigenständige Systeme welche, durch den SLS generierte Steuerbefehle, entgegen nehmen und verarbeiten (BILD 3).
Eine synchrone Kommunikation zwischen Task und FLS ist nicht möglich, da beide immer auf demselben, sequenziell arbeitenden, Prozessorkern ausgeführt werden. Für die Realisierung dieser Kommunikation ergeben sich damit zwei Möglichkeiten.
Die erste Möglichkeit ist die Verwendung der bereits beschriebenen asynchronen gepufferten Kommunikation und dessen Abarbeitung sobald sich ein Zeitfenster im Zeitplan ergibt. Diese Art des Phasenwechsels wäre Unterbrechungsfrei und würde die Ausführung des Echtzeitsystems nicht beeinträchtigen.
Alternativ kann ein schneller Phasenwechsel durch eine aktive Unterbrechung des laufenden Tasks durch den FLS ausgeführt werden. Die Verarbeitung des Phasenwechsels geschieht in diesem Fall sofort.

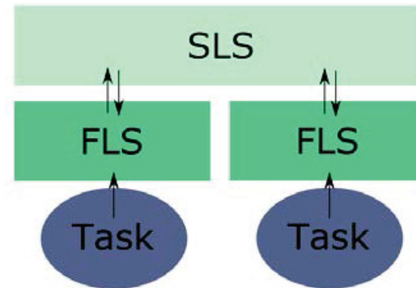

BILD 3. Kommunikation zwischen Task, FLS und SLS

4.5. Logging und Debugging

Durch die gepufferte Kommunikation kann in der Kommunikationsschicht selbst ein Logger oder Debugger dazwischen geschaltet werden, ohne das System zu verzögern.

5. TECHNISCHE REALISIERUNG

5.1. Related Work

Framework Paper wie Litmus^{RT} [7], AQuoSA [5] oder ExSched [8] beschäftigen sich damit, dem Entwickler Werkzeuge für Linux zur Verfügung zu stellen, um die Entwicklung neuer Scheduling-Algorithmen zu vereinfachen.
Dabei wird, am Beispiel von ExSched ein Kernelmodul geladen, welches im Userspace verwendete API-Befehle entgegen nimmt und damit aktiv in den aktuellen Scheduleablauf eingreifen kann. Diese API-Befehle werden durch ein ExSched-Modul im Kernel ausgeführt. Dadurch können Entwickler neue Scheduling-Algorithmen testen ohne dabei aktiv den Kernel verändern zu müssen.
Dieses Konzept der Trennung von User und Kernelspace (Abb. BILD 4) bildet einen wichtigen Baustein für die technische Realisierung von HAMS.

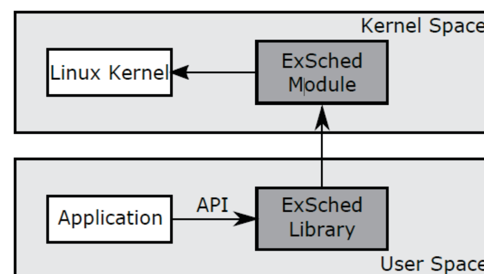

BILD 4. Aufbau ExSched nach [8]

5.2. HAMS in Linux

Für die technische Realisierung haben wir uns für das quelloffene Linux entschieden.
Der erste Schritt, der für moderne Betriebssysteme wie Linux betrachtet werden muss, ist die Taskunterteilung in Kernel und Userspace-Tasks. Diese Unterteilung dient hauptsächlich der Sicherheit, um das Betriebssystem vor Angriffen zu schützen. Es erschwert allerdings für HAMS die Kommunikation zwischen FLS (Kernelspace) und SLS (Userspace) oder zwischen FLS und dem eigentlichen User-Task (Userspace).
Der SLS muss innerhalb des Userspace ausgeführt werden, da die Management-Einheit des SLS Zugriff auf das Filesystem haben muss, um die Knowledgebase erfolgreich laden zu können.
Um über diese Betriebssystemgrenzen zu kommunizieren, muss der FLS ähnlich dem ExSched-Projekt-Prinzip (siehe BILD 5) in zwei Komponenten aufgeteilt werden. Einer Userspace-Komponente und einer Kernelspace-Komponente.
Mit Hilfe einer durch HAMS zur Verfügung gestellten API kann ein beliebiger, im Userspace laufender Task, unidirektional Nachrichten mit dem UserSpace-Anteil des FLS austauschen.
Diese Nachrichten können anschließend intern von der FLS-Userspace-Komponente an die Kernelspace-Komponente des FLS weitergeleitet werden.
Die FLS-Kernelspace-Komponente sendet diese Nachrichten durch den asynchron gepufferten Kommunikationskanal zu dem übergeordneten SLS.
In der bereits beim Systemstart in den Speicher geladenen Knowledgebase kann anhand des gemeldeten Phasenübergangs der richtige Softwareverteilungsplan gefunden werden und die erforderlichen Steuerbefehle über den gleichen Weg zurück an den zugehörigen FLS übertragen werden. Nach der erfolgreichen Abarbeitung der Befehle befindet sich das System erneut in einem offline-vorberechneten statischen Zustand. (BILD 5)

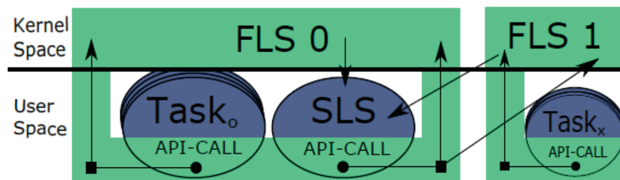

BILD 5. Logischer Aufbau von HAMS

6. ZUSAMMENFASSUNG

Um die effiziente Verteilung von Rechenzeit weiter zu verbessern wurde in diesem Paper das Konzept eines hierarchischen asynchronen Multicore Scheduler vorgestellt. Dies geschieht durch gezieltes Ausnutzen von sich gegenseitig ausschließenden Phasen des Systems. Dafür wurde das System in hierarchische Ebenen unterteilt und jeder Ebene eine eigene Schedulingeinheit zugewiesen. An dieser Stelle setzt der Hierarchical Asynchronous Multicore Scheduler (HAMS) an, um die Vorteile eines statisch geplanten Systems mit den Vorteilen eines dynamischen rekonfigurierbaren hierarchischem Systems zu verbinden. Des Weiteren bleibt das System nach außen weiterhin statisch und deterministisch, solange sich das System in keinem Phasenwechsel befindet. Das Konzept kann auf jedes echtzeitfähige Betriebssystem übertragen werden.

Verweise

[1] „653 Avionics Application Software Standard Interface," [Online]. Available: aviation-ia.com.

[2] R. T. Systems, „RTS Real-Time Embedded Hypervisor," Real Time Systems, [Online]. Available: {http://www.real-time-systems.com/de/real-time_hypervisor/index.php.

[3] M. Åsberg, T. Nolte und S. Kato, „Towards Partitioned Hierarchical Real-Time Scheduling on Multi-core Processors," 06.2014.

[4] W. Systems, VxWorks Kernel Programmers Guide, 2015.

[5] F. Checconi, T. Cucinotta, D. Faggioli und G. Lipari, „Hierarchical Multiprocessor CPU Reservations for the Linux Kernel," June 2009.

[6] M. Ernst, T. Hanti und A. Frey, „Higher Utilization of Multi-Core Processors in Dynamic Real-Time Software Systems," Dez 2013.

[7] J. Calandrino, H. Leontyev, A. Block, U. Devi und a. J. Anderson, „LITMUSRT: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers," December 2006.

[8] M. Asberg und T. Nolte, „ExSched: An External CPU Scheduler Framework for Real-Time Systems," 2012.