Parallel-Design für Echtzeit-Software
Anwendbarkeit im AUTOSAR-Umfeld

Ralph Mader, Continental Automotive GmbH

Pattern für das Design von parallel ausführbaren Algorithmen sind bislang noch nicht systematisch in Embedded Realzeit-Systeme vorgedrungen. Bei Continental wird derzeit die Anwendbarkeit von Pattern in AUTOSAR basierten Anwendungen untersucht. Der Vortrag wird auf die als in diesem Anwendungsbereich anwendbaren Pattern abzielen und die Vorgehensweise bei der Analyse darstellen. Des Weiteren wird die Anwendung eines Patterns an einem konkreten Beispiel aus dem Motorsteuerungsumfeld dargestellt. Es werden mögliche Lösungswege zur Scheduling-Unterstützung, z.B. über Kerngrenzen synchronisierte Zustandswechsel oder die Anwendung von logischer Ausführungszeit, in diesem Kontext aufgezeigt.

1. Vorstellen der Anwendungsdomäne

Eingebettete Systeme im Antriebsstrang eines Kraftfahrzeugs sind seit mehr als 25 Jahren in Verwendung. Sie dienen dazu, die Verbrennung, die Schaltvorgänge und das Batteriemanagement zu kontrollieren sowie auch vermehrt übergeordnete Algorithmen in Domänen-Steuergeräten zu darzustellen. Seit etwa fünf Jahren haben Mehrkernrechner in dieses Anwendungsfeld Einzug gehalten. Es werden dort Mikrokontroller mit bis zu drei nahezu homogenen Kernen und einer Taktfrequenz von bis zu 300 MHz eingesetzt. Die darauf zum Einsatz kommende Software ist im Laufe der Jahrzehnte gewachsen; sie umfasst in einer Motorsteuerung etwa 2 Millionen Lines of Code und benötigt etwa 6 MB Programmspeicher.

Die verwendete Software-Architektur basiert auf dem AUTOSAR-Standard 4.0.[1]. Bei der Umstellung auf Mehrkernrechner wurde die Applikationssoftware im Hause Continental im Wesentlichen „Multicore Ready“ überarbeitet, mit dem Ziel, Software über Kerne hinweg zu verteilen und die Datenkonsistenz weiterhin sicherzustellen [2] Bei der Verteilung wurde darauf geachtet, Wirkketten zu erhalten und im Wesentlichen den inhärent vorhandenen Task-Parallelismus auszunutzen. Bei derzeit etwa 60 Tasks (siehe Abbildung 1) in einem Motorsteuerungssystem war bis dato der erzielte Speedup ausreichend, um die Leistungsanforderungen der aktuellen Steuergerätegeneration zu befriedigen.

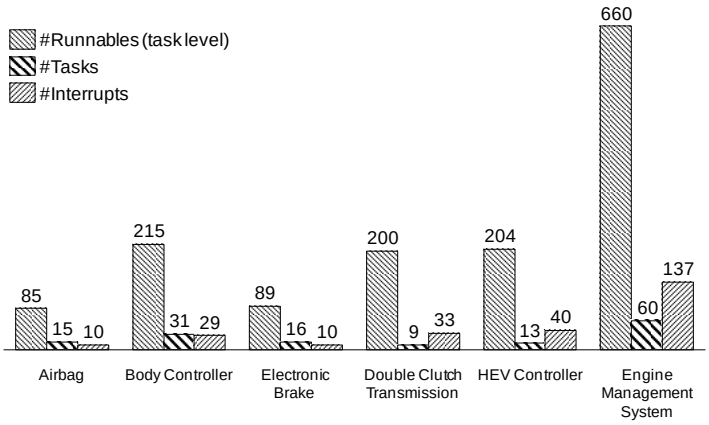

Abb.1: Anzahl von ausführbaren Softwareteilen-Tasks in verschiedenen Anwendungsbereichen des Automobils

2. Herausforderungen der Anwendung

Eine wesentliche Herausforderung der im Abschnitt 1 genannten Anwendungen ist die Echtzeitfähigkeit. So finden dort Berechnungsvorgänge in sich wiederholenden Tasks mit einer minimalen Wiederholrate von 1 Millisekunde und Deadlines von minimal 200 Mikrosekunden statt. Des Weiteren gibt es an die Motordrehzahl und das Ventilspiel der Verbrennungsmotoren gebundene, sich wiederholende Tasks. Die darin stattfindenden Berechnungen sind ein wesentlicher Faktor für die Qualität der Verbrennung und damit der Abgaszusammensetzung. Aufgrund steigender Anforderungen an die Abgasreinheit und den Kraftstoffverbrauch ist zu erwarten, dass diese Berechnungen in Zukunft noch mehr Rechenzeit benötigen. Deshalb ist es notwendig, diese Algorithmen in einer Weise zu entwerfen, die eine parallele Ausführung zulässt, um auch weiterhin das Echtzeitverhalten gewährleisten zu können.

3. Wie kann Parallelität gefunden werden?

Es stellt sich nun die Frage, welche Strategie zur Findung von Parallelität zielführend ist. Bei Analyse des Codes stellt man in dieser Art Anwendungen sehr häufig fest, dass es längere Berechnungsketten gibt, in denen man Cluster bilden kann, die sich auf Kerne verteilen lassen. Dazu kann man noch den Sourcecode durchsuchen und zum Beispiel hinsichtlich Schleifen-Parallelität Analysen durchführen. Dabei lässt sich häufig feststellen, dass sehr häufig kurze Schleifen zum Einsatz kommen, meist nur über die Anzahl der Zylinder. In den gegebenen Rechnerarchitekturen ist es allerdings aus Effizienzgründen wenig sinnvoll, diese Schleifen über Kerne zu parallelisieren.
Aus diesem Grunde wurde zu einer anderen Methode umgeschwenkt. In Interviews wurden die Funktionsentwickler, meist Maschinenbauingenieure, bezüglich des zu lösenden funktionalen Problems befragt, und es wurde anhand der Problemstellung versucht, den Softwareentwurf komplett neu zu gestalten, um Parallelität in der Softwareausführung zu ermöglichen. Dabei wurde den Entwicklungsteams einige Entwurfsmuster vorgestellt, um die Ideenfindung zu unterstützen. In den folgenden Kapiteln werden einige der Entwurfsmuster vorgestellt, die sich dabei als umsetzbar herausgestellt haben.

4. Design Pattern für paralleles Programmieren

4.1 Pipe and Filter Design Pattern

Das “Pipe and Filter” Entwurfsmuster hilft auf struktureller Ebene, Parallelität zu identifizieren, auch wenn eine Reihe von Tasks voneinander abhängt. Bei einem nach vorwärts gerichteten Datenfluss und im Fall, dass die Taskausführung nicht vom Ergebnis der vorausgegangenen Stufe abhängt, lassen sich unterschiedliche Tasks in Stufen gruppieren, ähnlich dem „Pipeline“-Verfahren in Mikroprozessoren (siehe Abbildung 2).
Die Granularität der Stufen muss mit dem Synchronisierungsaufwand in einem ausgewogenen Verhältnis stehen. Ideal ist, wenn alle Stufen eine annähernd gleiche Ausführungszeit aufweisen. Ist dies nicht der Fall, bestimmt die längste Stufe die Taktung von Stufe zu Stufe.
In einer Steuerungsanwendung, wie z.B. einer Motorsteuerung, kann für die relevanten Daten einer Systemfunktion Vorwärtsrichtung im Datenfluss angenommen werden. Die verschiedenen Tasks können in Stufen eingeordnet werden. Ein Schema für die Einordnung stellt das bei Continental im Einsatz befindliche Phasenmodell [4] dar, welches sich in dieses Entwurfsmuster überführen lässt.

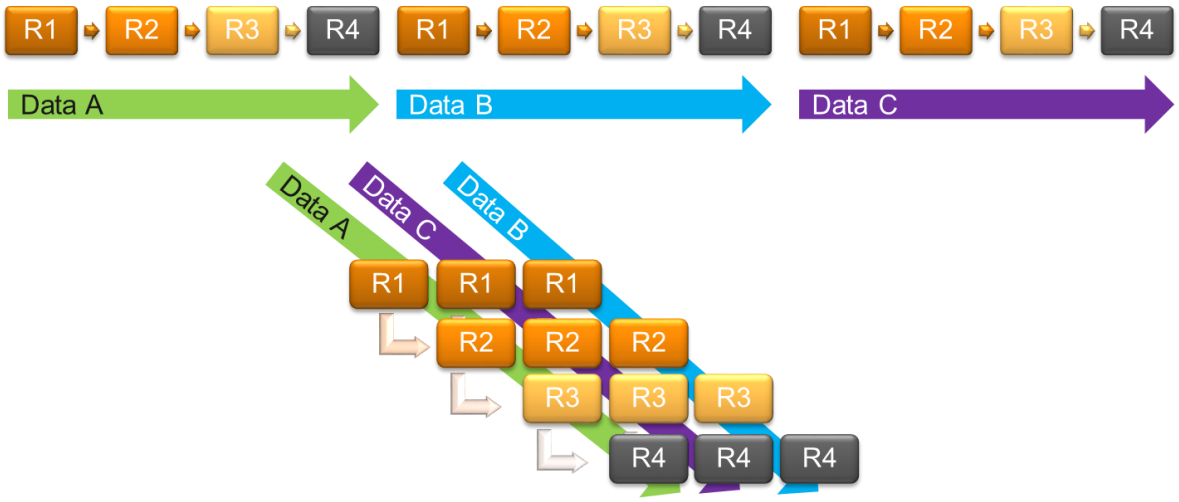

Abbildung2: Einordnung eines sequenziellen Vorgangs in das „Pipe and Filter“ Entwurfsmuster. Alle Runnables gleicher Funktionalität werden in einer Stufe gruppiert, die auf einem Kern ausgeführt wird.

4.2 Divide and Conquer (Algorithm Strategy Pattern)

Das „Divide and Conquer“ Pattern lässt sich auf viele Algorithmen anwenden. Dabei wird eine Aufgabenstellung in Unteraufgaben geteilt. Bei dem Entwurf ist darauf zu achten, dass diese Unteraufgaben unabhängig voneinander gelöst werden können. Dies erlaubt dann, die Unteraufgaben auf verschiedene Kerne auszulagern (siehe Abbildung 2). Am Ende müssen die Teilergebnisse wieder zusammengeführt werden, welches einen gewissen Synchronisationsaufwand bedeutet. In Steuerungsanwendungen existiert dafür eine Reihe von Anwendungsfällen. Von Vorteil ist, mehrere wiederum voneinander unabhängige Algorithmen dieser Art zusammenzufassen, um den Synchronisierungsaufwand gering zu halten.

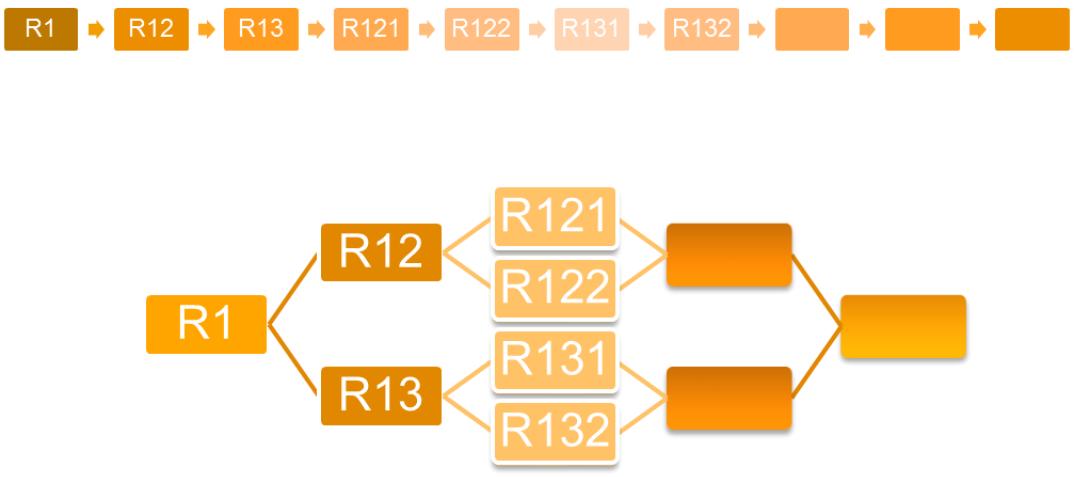

Abb. 2: Parallelisierung über das „Divide and Conquer“ Pattern, welches die Aufteilung eines Problems in unabhängig voneinander lösbare Unteraufgaben erfordert.

4.3 Bulk Synchronous Parallel

Das “Bulk Synchronous Parallel Pattern” erlaubt, Tasks, die auf unterschiedlichen Kernen zur Ausführung kommen, an einer Barriere zusammenzuführen und dadurch zu synchronisieren.

Zu Beginn einer parallelen Phase werden die Daten jedem Task in einer lokalen Kopie aus dem globalen Speicher zur Verfügung gestellt. Während der Ausführung wird auf der lokalen Kopie gearbeitet. Während der Taskausführung ist Kommunikation nur zwischen Runnables dieses Tasks erlaubt.
Nachdem alle parallel laufen Tasks beendet wurden, werden die Ergebnisse wieder im Global-Speicher zur Verfügung gestellt, bevor die nächste Phase startet.
Diese Art von Barrieren kann zur Implementierung des „Divide and Conquer“ Patterns als auch des “Pipe and Filter“ Patterns verwendet werden. Aufeinanderfolgende Phasen können die verschiedenen Stufen der Pipeline darstellen.

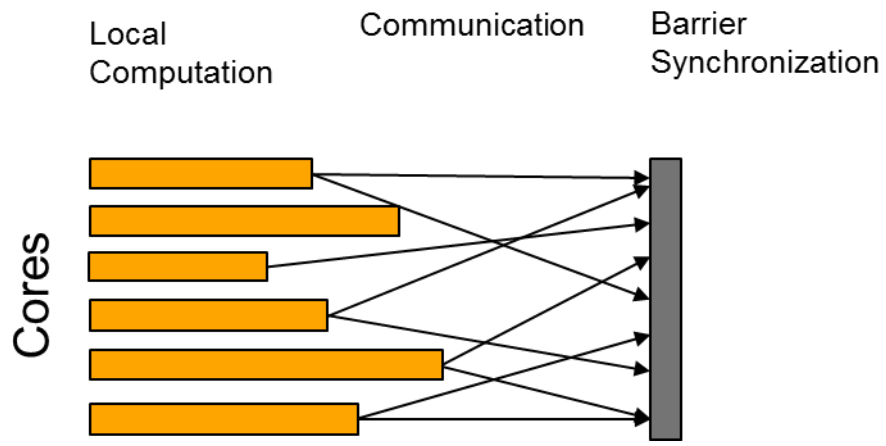

Abb. 3: Prinzip-Darstellung des „Bulk Synchronous Parallel“ Pattern mit der Barriere zur Synchronisation der Tasks am Ende.

5. Implementierungsbeispiele

5.1 Über Kerne synchronisierter Zustandswechsel

In Steuergeräten im Fahrzeug finden sehr häufig Zustandswechsel statt. Einige davon haben Auswirkungen auf das Gesamtsystem, was zur Konsequenz hat, dass bei einer auf die Kerne eines Controllers verteilten Anwendung diese Zustandswechsel zu synchronisieren sind. Um das Echtzeitverhalten der zyklischen Tasks nicht zu beeinträchtigen, werden durch einen sog. „Event-Koordinator“ nach einer erkannten Anforderung zum Wechsel eines Systemzustandes die zyklisch laufenden Tasks beendet und aktivierte Tasks mit weniger kritischen Deadlines verzögert; dadurch wird im Gesamtsystem ein freier Zeitslot provoziert. Der „Event-Koordinator“ feuert dann zeitgleich auf allen Kernen jeweils einen Zustandswechsel-Task. Darin lassen sich sehr gut Initialisierungsfunktionen parallel ausführen, da diese typischerweise nicht oder nur geringfügig gekoppelt sind. Nachdem auf allen Kernen die Zustandswechsel-Tasks beendet sind, wird die Abarbeitung der zyklischen Tasks vom „Event Koordinator“ wieder freigegeben.
Durch dieses Verfahren lässt sich die benötigte Rechenzeit bei derlei Zustandswechseln signifikant verkürzen. Die Initialisierung eines Fehlerspeichers mit etwa 1000 Fehlerorten liefert hier die eindrucksvollsten Ergebnisse, da dies eine der wenigen Möglichkeiten ist, in einer Steuerungsanwendung von einer Schleifenoptimierung zu profitieren. Der Parallelisierungsgrad dabei ist nahezu 100%, was gemäß „Amdahls Law“ einem Speedup gleich der Anzahl der Kerne entspricht.

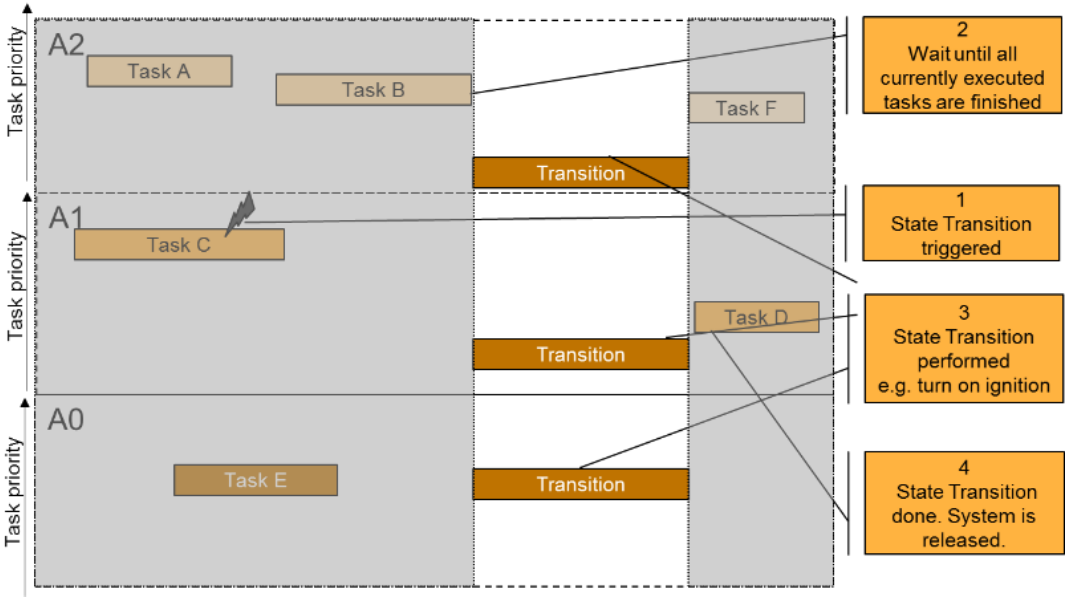

Abb. 4: Anwendungsbeispiel des „Bulk Synchronous Parallel“ Patterns für einen über Kerne synchronisierten Zustandswechsel

5.2 Logical Execution Time (LET)

Das Zeitverhalten von Mehrkernrechner-Systemen mit Hilfe des logischen Ausführungszeitmodells zu beschreiben und zu implementieren, erlaubt die Implementierung des im Abschnitt 4 beschriebenen „Pipe and Filter“ als auch des „Divide and Conquer“ Pattern.

Durch das Einteilen einer Zeit-Periode in Abschnitte mit logischen Ausführungszeit lassen sich Berechnungsketten darstellen, die zwar in einer Sequenz abgearbeitet werden, aber streckenweise Parallelisierungsmöglichkeiten beinhalten. Analysiert man zum Beispiel, wie in Abbildung fünf gezeigt, die Sequenz eines Tasks, kann man dieses Muster erkennen. Die so gefundenen sequenziellen und parallelen Cluster können in einem wie in Abbildung 6 dargestellten logischen Ausführungszeitrahmen zur Abarbeitung [5] gebracht werden. Die Kommunikation von sequenziell aufeinanderfolgenden LETs erfolgt ohne zusätzliche Datenkonsistenz-Mechanismen, wobei die Kommunikation zwischen parallelen Tasks über Konsistenzpuffer geschützt wird [6].

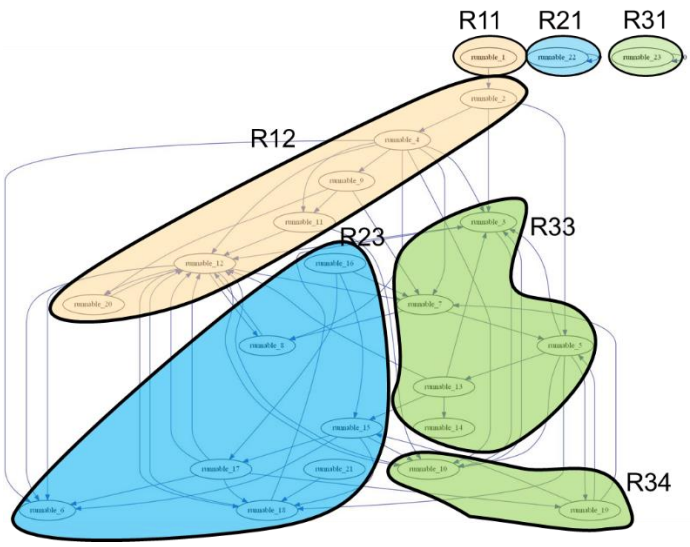

Abb. 5: Gerichteter Graph einer 10ms-Sequenz mit der Einteilung in schwach gekoppelte Cluster

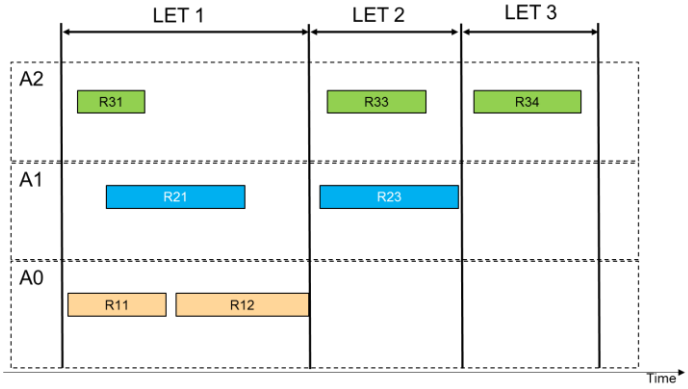

Abb. 6: Logischer Ausführungszeitrahmen für die 10ms-Sequenz aus Abbildung 5

6. Zusammenfassung und Ausblick

Wie gezeigt, wurde die Identifikation von Parallelität über den bereits gefundenen Taskparallelismus hinaus in Kontrollanwendungen eingeschränkt. Trotzdem lassen sich einige Pattern identifizieren und durchaus mit Erfolg anwenden. Ein wesentlicher Faktor für den Erfolg ist Parallelität bereits beim Funktionsdesign mit zu berücksichtigen. Ein architektureller Rahmen, wie ihn die Anwendung des logischen Ausführungszeitsmodells bietet, kann in abstrahierte Form ähnlich dem Phasenkonzept [4] den Entwicklern an die Hand gegeben werden. Dies würde die Anwendbarkeit dieses Ansatzes deutlich vereinfachen. Hier ist in Zukunft noch eine bessere Unterstützung durch Designwerkzeuge wünschenswert.

Quellenverzeichnis

[1] AUTOSAR-Consortium, in http://www.autosar.org/index.php?p=3&up=2&uup=0, Rev 4.0.3.

[2] D. Claraz, F. Grimal, T. Ledier, R. Mader, and G. Wirrer, “Introducing multi-core at automotive engine systems,” in ERTS2, 2014.

[3] M. Negrean, R. Ernst, and S. Schliecker, “Mastering timing challenges for the design of multi-mode applications on multicore real-time embedded systems,” in 6th International Congress of Embedded Real-Time Software and Systems (ERTS), 2012.

[4] D. Claraz, S. Kuntz, U. Margull, M. Niemetz, and G. Wirrer, “Deterministic Execution Sequence in Component Based Multi-Contributor Powertrain Control Systems,” in ERTSS 2012, 2012.

[5] J. Hennig, H. von Hasseln, H. Mohammad, S. Resmerita, S. Lukesch, and A.Naderlinger. 2016. Towards parallelizing legacy embedded control software using the LET programming paradigm. In Proc. of WiP Papers of the 22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS ’16).

[6] Stefan Resmerita, Andreas Naderlinger, Stefan Lukesch, “Efficient Realization of Logical Execution Times in Legacy Embedded Software” MEMOCODE’17, Vienna, Austria, 2017

Autor

Ralph Mader studied Electrical Engineering at the University of Applied Sciences in Regensburg. He worked on the design and selection of microcontrollers for the Powertrain area and the efficient use of microcontroller resources. Since 2010, he is also leading development of the Multi Core software architecture for engine management systems. Mr. Mader represents Continental Automotive GmbH in several research projects in this area.