Migration von AUTOSAR-basierten Echtzeitanwendungen auf Multicore-Systeme *

Michael Bohn, Jörn Schneider, Christian Eltges, Robert Rößger
Fachhochschule Trier
Schneidershof, 54293 Trier
Email: {m.bohn, j.schneider, c.eltges, r.roessger}@fh-trier.de

***Kurzfassung*—Selbst sicherheitskritische Echtzeitanwendungen im Automobilbereich werden in naher Zukunft auf Multicore-Systeme migriert werden, um den steigenden Innovationshunger nach neuen Produktmerkmalen decken zu können. Eine manuelle Migration der über Jahre gewachsenen Softwarebasis ist, schon aufgrund des Umfangs der zu migrierenden Software, kaum zu leisten. Hinzu kommen die kaum zu durchschauenden Abhängigkeiten der Softwarebestandteile untereinander, die mühevoll analysiert und bewertet werden müssen, um eine korrekte Migration durchzuführen. Dieser Beitrag stellt einen neuartigen Ansatz für das automatische Auffinden und Bewerten von Abhängigkeiten mit Hilfe statischer Programmanalysen vor. Die ebenfalls vorgestellten ersten Ergebnisse stützen die These, dass das Konzept zur teilautomatisierten Migration von Echtzeitsoftware auf Multicore-Systeme einsetzbar ist.**

***Schlüsselwörter*-Statische Programmanalyse, Echtzeitsysteme, Migration, Multicore-Systeme**

I. Einführung

In Zukunft werden Multicore-Prozessoren verstärkt Einzug in Kfz-Steuergeräte finden. Die Gründe dafür sind u.a. der stetig ansteigende Leistungshunger neuer Funktionen gepaart mit dem Erreichen technologischer Grenzen, bzgl. Verlustwärme, elektromagnetischer Verträglichkeit und Sensitivität gegenüber kosmischer Strahlung bei sicherheitsrelevanten Anwendungen, aufgrund immer höherer Taktfrequenzen von Singlecore-Prozessoren. Zwar gibt es hierfür auch andere Lösungsansätze, beispielsweise aktive Kühlung und Abschirmung, aber diese sind in der Regel wesentlich teurer oder nicht sinnvoll durchführbar.

Der anstehende Wechsel zu Multicore-Prozessoren stellt für die Automobilindustrie einen disruptiven Technologiewechsel bezüglich der Softwareentwicklung dar. Über Jahrzehnte hinweg mussten technische Aufgaben, die inhärent paralleler Natur sind, in das Korsett sequentieller, bestenfalls pseudo-paralleler Programme gezwungen werden. Die effizientesten Programme waren hierbei diejenigen, die die erzwungene Sequentialisierung bestens ausnutzten. Wozu sollte man sich beispielsweise die Mühe machen, einen Zugriff auf eine geteilte Ressource mittels semaphorartiger Mechanismen zu schützen, wenn eine simple Interruptsperre wesentlich schneller geht?

Somit ist es nur natürlich, dass eine Vielzahl von Nebenläufigkeitseinschränkungen in heutiger Automobilsoftware vorherrschen. Für die Migration der Systeme auf Multicore-Prozessoren stellt dies eine Herausforderung dar. Würde man alle diese Einschränkungen mittels Multicore-geeigneter Mechanismen erhalten, wäre das Ergebnis de facto ein zum Singlecore-System ausgebremstes Multicore-System. Glücklicherweise sind viele dieser Nebenläufigkeitseinschränkungen nicht erforderlich. Allerdings ist es extrem aufwändig und fehleranfällig für den Entwickler, zu erkennen, welche das sind.

Hier setzt das Forschungsprojekt ProSyMig[1] an, in dem statische Programmanalysewerkzeuge entwickelt, bzw. eingesetzt [1] werden, die einen ersten Schritt in Richtung Automatisierung der Migration auf Multicore-Systeme darstellen. Als Ausgangspunkt für die analyseunterstütze Migration sind sowohl OSEK-basierte [2] als auch AUTOSAR Anwendungen [3] inklusive Basissoftware geeignet. Im Fokus stehen insbesondere sicherheitsrelevante Echtzeitanwendungen. Daher basieren die Analysen auf der Methodik der Abstrakten Interpretation [4], damit bleiben keine im Code begründeten Einschränkungen unentdeckt und die Korrektheit der Ergebnisse ist formal beweisbar.

Im praktischen Einsatz von Werkzeugen, die auf Abstrakter Interpretation beruhen, ist eine besondere Herausforderung, die Anzahl der *false positives*, d.h. der lediglich vermeintlichen Problemstellen, einzuschränken. Um dies von vornherein bestmöglich zu adressieren, wurde das hier vorgestellte zweistufige Analysekonzept entwickelt.

II. Verwandte Arbeiten

In [5] wird beschrieben, wie eine automatisierte Partitionierung von Runnables AUTOSAR-basierter Echtzeitsoftware vorgenommen werden kann. Software auf OSEK-Basis wird hier nicht berücksichtigt. Abhängigkeiten in Form von Zugriffen auf globale Variable werden mit Hilfe statischer Programmanalysen ermittelt. Eine detaillierte Beschreibung der Analyse wird nicht gegeben, daher ist ein Vergleich der Ansätze an dieser Stelle nicht seriös möglich.

Weiterhin werden verschiedene Ansätze zur Parallelisierung vorgestellt. Aus den ermittelten Werten der Prozessorauslastungen des migrierten Systems lässt sich erkennen,

*Diese Arbeit wurde teilweise gefördert durch das Bundesministerium für Bildung und Forschung, Förderkennzeichen 17N1309

[1] ***Pro**gramm- und **Sy**stemanalysewerkzeuge zur **Mig**ration eingebetteter Software auf Multicore-Systeme*

dass Abhängigkeiten und Synchronisationskosten einen hohen Einfluss auf die Effizienz des parallelisierten Systems haben.

[6] beschreibt die Partitionierung modellbasierter Anwendungen auf Basis von Simulink. Aufgrund der Betrachtung von Simulink-Modellen können alle Abhängigkeiten aus dem Modellcode extrahiert und betrachtet werden. Im Vergleich dazu, ist der hier vorgeschlagene Ansatz unabhängig von der Art der Erzeugung des Quellcodes. Anders als in [6] kann der propagierte Ansatz neben generiertem Quellcode auch manuell erstellten Quellcode und Basissoftware korrekt analysieren.

III. Gesamtkonzept Migrationsablauf

Ein bestehendes System ist in Tasks, Runnables, d.h. zusammengehörende Quelltextabschnitte innerhalb eines Tasks, und Interrupt-Service-Routinen (ISR) organisiert. Die Aufgabe bei der Migration besteht darin, jedes bestehende, bzw. neu hinzugekommene Laufzeitobjekt (Task, Runnable oder ISR) der Anwendungs- und Basissoftware genau einem Kern zuzuweisen. Über diese Partitionierung wird die gewünschte Parallelität erreicht. Zu beachten ist hier, dass die kleinste Einheit für die Zuordnung auf Prozessorkerne ein Task, ein Runnable oder eine ISR ist. Bei der hier vorgeschlagenen automatisierten Migration wird nicht versucht eine eventuell vorhandene Intra-Runnable oder Intra-Task Parallelität auszunutzen. Die Einheiten Task, Runnable und ISR werden daher zusammenfassend als *Atomare Migrationseinheiten* (AME) bezeichnet und das Zuweisen der AMEs an Kerne im Multicore-System als AME-Partitionierung.

Als Randbedingung der AME-Partitionierung darf kein Kern überlastet werden, wodurch das Problem zu einer Variante des Bin-Packing-Problems wird. Durch die NP-schwere dieses Problems ist es je nach Problemgröße unwahrscheinlich, dass eine optimale Lösung in effizienter Laufzeit automatisch gefunden werden kann. Sind alle AMEs unabhängig voneinander, reduziert sich das Migrationsproblem auf die Bin-Packing-Problematik.

Die Unabhängigkeit der AMEs trifft in der Realität allerdings nur in den seltensten Fällen zu. Im Allgemeinen wird die beliebig nebenläufige Ausführung der AMEs durch *Einschränkungen* begrenzt. Diese Einschränkungen lassen sich in drei Klassen unterteilen: *wechselseitiger Ausschluss*, *Reihenfolge-* und *zeitliche Einschränkung*. Für jede dieser Einschränkungsklassen existieren mehrere unterschiedliche Mechanismen, beziehungsweise Kombinationen von Mechanismen, zur konkreten Realisierung in einem Singlecore-System. Ein wechselseitiger Ausschluss kann beispielsweise über den Ressourcen-Mechanismus[2], das Sperren von Interrupts[3] oder über gleiche Taskprioritäten absichtlich oder unabsichtlich realisiert werden. Für die Einschränkungsklassen Reihenfolge und Zeit existieren ähnliche Mengen unterschiedlicher Mechanismen [7].

Bei den Mechanismen lässt sich zwischen *expliziten* und *impliziten* Mechanismen unterscheiden. Ein expliziter Mechanismus wurde für die Realisierung der jeweiligen Abhängigkeit entworfen. Im Beispiel ist der Ressourcen-Mechanismus ein expliziter Mechanismus. Ein impliziter Mechanismus realisiert eine bestimmte Abhängigkeit hingegen nur als Seiteneffekt oder nur in Kombination mit anderen Mechanismen. Im Beispiel werden das Sperren von Interrupts und das Ausnutzen der Taskpriorität zu den impliziten Mechanismen gezählt. Das Sperren von Interrupts ist ein impliziter Mechanismus, da ein Sperren von Interrupts nur einen wechselseitigen Ausschluss zwischen dem sperrenden Task und Tasks mit höherer Priorität (bzw. mit ISRs) herstellt. Das Sperren von Interrupts wirkt somit nur in Kombination mit der Taskpriorität als wechselseitiger Ausschluss zwischen Tasks. Für eine detailliertere Einteilung der Einschränkungsklassen siehe [7].

A. Migrationsphasen

Für die Migration eines Singlecore-Systems wird ein vierphasiger Ansatz verfolgt:

- **Phase 1:** Extraktion von Einschränkungen
- **Phase 2:** Reduktion von Einschränkungen
- **Phase 3:** Exploration von AME-Partitionierungen
- **Phase 4:** Codegenerierung

In der Extraktionsphase werden alle im Singlecore-System realisierten Einschränkungen zwischen AMEs ermittelt. Ein großer Teil der Analysen in dieser Phase kann über statische Programmanalysen, bzw. das Auslesen der Systembeschreibung, automatisiert werden. In dieser Phase wird eine Vielzahl von Einschränkungen erkannt, die im Singlecore-System potentiell realisiert sind. Die erkannten Einschränkungen dieser Phase lassen sich dabei in drei Mengen einteilen:

1) Vermeintliche Einschränkungen
2) Für die Korrektheit des Systems notwendige Einschränkungen
3) Für die Korrektheit nicht bedeutsame Einschränkungen

Die Einschränkungen aus der ersten Menge entstehen durch die Überapproximation der Programmanalysen. Dies lässt sich im Allgemeinen nicht verhindern, da Programmanalysen aufgrund der Unentscheidbarkeit des zugrundeliegenden Problems nur eine approximierte Annäherung liefern können, wenn keine tatsächliche Einschränkung unentdeckt bleiben darf.

Alle Einschränkungen der zweiten Menge müssen zwingend auch im Multicore-System erhalten bleiben, um eine korrekte Arbeitsweise zu gewährleisten. Im Gegensatz dazu sind die Einschränkungen aus der dritten Menge im System zwar realisiert, sie werden aber nicht für die Korrektheit

[2] *Get-* und *ReleaseResource* in OSEK/AUTOSAR
[3] *Disable-* und *EnableAllInterrupts* in OSEK/AUTOSAR

benötigt. Hauptquelle für die Einschränkungen in dieser Menge sind insbesondere die impliziten Mechanismen. Wird ein expliziter Mechanismus verwendet, kann im Allgemeinen davon ausgegangen werden, dass dieser Mechanismus eine erforderliche Einschränkung realisiert. Bei der Verwendung eines impliziten Mechanismus entstehen eine Vielzahl unnötiger Einschränkungen.

Nach Abschluss der ersten Phase werden alle Einschränkungen vorläufig als erforderlich klassifiziert. Ein möglicher Ansatz wäre, alle erkannten Einschränkungen ebenfalls im Multicore-System zu realisieren. Dies würde letztlich zu einer Simulation des Singlecore-Verhaltens auf einem Multicore-System führen. Die Parallelität würde unnötig eingeschränkt werden und das resultierende System wäre sehr ineffizient.

In der zweiten Phase (Reduktion von Einschränkungen) wird daher versucht, Einschränkungen als *wahrscheinlich notwendig* bzw. *vermutlich unnötig* zu klassifizieren. Um diese Klassifizierung automatisiert vorzunehmen, werden weitere statische Programmanalysen verwendet. Ein möglicher Grund für die Einschränkung *wechselseitiger Ausschluss* könnte der Schutz des nebenläufigen Zugriffs auf geteilten Speicher sein, was durch eine Analyse des Speicherzugriffsverhaltens ermittelt werden kann. Wird der Zugriff auf geteilten Speicher erkannt, kann eine Einschränkung als *wahrscheinlich notwendig* klassifiziert werden. Wird kein Zugriff auf geteilten Speicher erkannt, kann eine Einschränkung als *vermutlich unnötig* klassifiziert werden.

Wie an den Bezeichnungen der Klassifizierungen zu erkennen ist, lassen sich hier keine absoluten Einteilungen vornehmen. Wird ein Zugriff auf geteilten Speicher festgestellt, kann, bedingt durch den approximativen Charakter der Speicheranalyse, nur ein möglicher Grund angenommen werden. Es ist möglich, dass in der tatsächlichen Ausführung des Programms nicht auf geteilten Speicher zugegriffen wird, die Analyse dies jedoch meldet.

Solche gemeldeten Speicherzugriffe gehören zur Gruppe der *false positive* Meldungen. Die Genauigkeit der Klassifikation ist hier sehr stark abhängig von der Genauigkeit der verwendeten Speicheranalyse. Wird von der Speicheranalyse kein gemeinsamer Speicherzugriff erkannt, kann *definitiv* ausgeschlossen werden, dass die Einschränkung für den Schutz des nebenläufigen Zugriffs auf geteilten Speicher existiert. Hier ist eine definitive Aussage möglich, da die von der Speicheranalyse erkannten Zugriffe garantiert alle tatsächlichen Speicherzugriffe enthalten. Es kann allerdings nicht ausgeschlossen werden, dass andere Gründe als der Schutz des Zugriffs auf geteilte Speicherbereiche existieren. Beispielsweise könnte die Anforderung existieren, dass zwei Aktoren niemals zur gleichen Zeit angesteuert werden, was anhand der Programmsemantik nicht feststellbar ist. Eine definitive Klassifikation in unnötige Einschränkung bzw. erforderliche Einschränkung muss letztendlich der Entwickler treffen.

Im Optimalfall bleiben nach Ende der Phase 2 nur die Einschränkungen übrig, die für die Korrektheit des Systems notwendig sind. In Phase 3 nimmt der Entwickler nun eine AME-Partitionierung vor und lässt diese bewerten. Bei der Bewertung einer Partitionierung wird, mit Hilfe der vorliegenden Analyseergebnisse der 2. Phase, für jede Einschränkung ermittelt, ob der jeweilige Mechanismus auch im so partitionierten Multicore-System die Einschränkung garantiert. Für die Einschränkungen, die im Multicore-System nicht realisiert sind, muss dann ein Multicore-sicherer Mechanismus als Ersatz gefunden werden.

In der vierten Phase wird der Quelltext des Systems angepasst, indem ungeeignete Mechanismen durch Multicore-fähige Mechanismen ersetzt werden. Diese Phase kann ebenfalls weitestgehend automatisiert werden.

IV. Analyse wechselseitigen Ausschlusses und Zugriffe auf geteilten Speicher

Von der allgemeinen Beschreibung der Migration aus dem vorherigen Abschnitt sind im Projekt bisher Phase 1 und 2 für die Einschränkungsklasse wechselseitiger Ausschluss realisiert worden. Diese beiden Phasen bestehen aus folgende Analysen:

- Extraktionsanalyse: API-Aufrufpaar-Analyse (hier *Disable*- und *EnableAllInterrupts*-Aufrufe)
- Reduktionsanalyse: Zugriffe auf geteilten Speicher.

Die API-Aufrufpaar-Analyse wurde eigens zur Analyse kritischer Abschnitte im Projekt entwickelt. Diese Analyse arbeitet auf der C-Quelltextebene und basiert auf Abstrakter Interpretation. Kritische Abschnitte werden i.d.R. über API-Aufrufpaare gebildet. Eine Variante dieser Analyse kann auch mit *Get*- und *ReleaseResource*-Aufrufpaaren umgehen. Die C-Ebene wurde hier gewählt, da beliebige API-Aufrufe so unmittelbar erkennbar sind.

Für die zweite Analyse wird das kommerzielle Analysewerkzeug *ValueAnalyzer*[4] der Firma AbsInt [1] verwendet. Dieses basiert ebenfalls auf Abstrakter Interpretation und analysiert Programme auf Binärebene. Für jeden Maschinencodebefehl wird ermittelt, auf welche Speicherstellen potentiell zugegriffen wird.

Das Zusammenspiel und die Arbeitsweise der beiden Analysen sowie die konkrete Realisierung der beiden Phasen wird im folgenden Abschnitt anhand eines Anwendungsbeispiels genauer erläutert.

A. Anwendungsbeispiel

Die in Listing 1 dargestellte Singlecore-Beispielanwendung besteht aus zwei Tasks, *T1* und *T2*, sowie einer ISR. Die ISR kopiert Daten aus dem Array *driverData* in das globale Array *buffer*. Das Array *driverData* steht hierbei für einen Speicherbereich, der von einer Hardwarekomponente beschrieben wird. Die beiden Tasks lesen die von der ISR

[4] http://www.absint.com/valueanalyzer/

kopierten Daten. Um sicherzustellen, dass die Werte im Array nicht verändert werden können, während sie von den Tasks gelesen werden, schützen beide Tasks den Zugriff auf das Array durch das Sperren von Interrupts. Als Betriebssystemplattform für die Beispielanwendung wurde das RTA-OSEK Betriebssystem der ETAS GmbH verwendet.

Listing 1. Szenario-Implementierung

```
/* Datei: T1.c */
extern int buffer[100];
TASK(T1) {
    int i,out,sum=0;
    DisableAllInterrupts();
    for(i=0; i<100; i++)
        sum += buffer[i];
    EnableAllInterrupts();
    out = sum * 100;
    TerminateTask();
}

/*Datei: T2.c*/
extern int buffer[100];
TASK(T2) {
    int i,out,product=1;
    DisableAllInterrupts();
    for(i=0; i<100; i++)
        product *= buffer[i];
    EnableAllInterrupts();
    out = product * 100;
    TerminateTask();
}

/*Datei: ISR.c*/
volatile int driverData[100];
int buffer[100];
ISR(ISR1) {
    int i;
    for(i=0; i<100; i++)
        buffer[i] = driverData[i];
}
```

B. Anwendung der Analysewerkzeuge

Phase 1: Über die API-Aufrufpaar-Analyse wird für jede C-Anweisung ermittelt, ob sie bei gesperrten oder nicht gesperrten Interrupts ausgeführt wird. Für das aufgeführte Beispiel-Szenario kann diese Analyse exakt entscheiden, ob eine Anweisung innerhalb oder außerhalb eines für Interrupts gesperrten Bereichs liegt.

Diese Exaktheit des Analyseergebnisses liegt jedoch in der Einfachheit des Beispiel-Szenarios begründet. Es werden keine Interrupts bedingt oder über Schleifengrenzen hinweg gesperrt. Die Analyse kann jedoch auch auf korrekte Weise mit Verzweigungen und Schleifen umgehen. Korrektheit für diese Art der Analyse bedeutet dabei, dass für eine C-Anweisung, die bei einer tatsächlichen Programmausführung bei gesperrten Interrupts ausgeführt wird, von der Analyse niemals die Aussage getroffen wird, dass diese Anweisung außerhalb gesperrter Interrupts ausgeführt werden kann. Die Analyse muss die Größe der kritischen Abschnitte somit überapproximieren.

Zusätzlich zu dem Ergebnis der Analyse für Interrupts, muss nun für jeden Task die Priorität ermittelt werden. Diese lässt sich durch Auslesen der Konfigurationsdaten im OIL- (*OSEK Implementation Language*) bzw. XML-Format ermitteln.

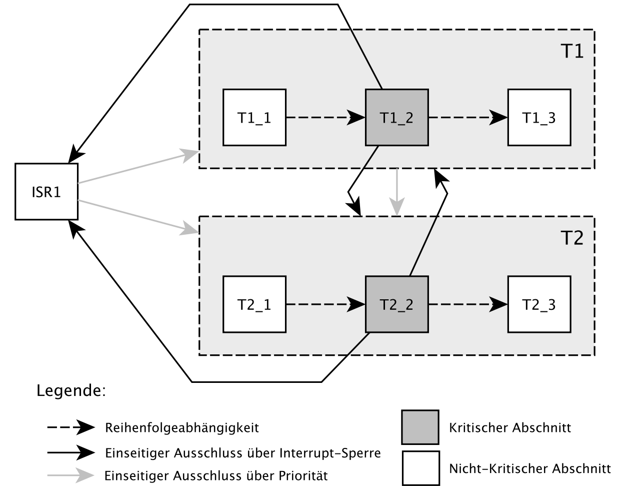

Abbildung 1. Im System realisierte Ausschlüsse

Aus diesen beiden Informationen kann nun der *Einseitige-Ausschluss-Graph* (EAG) erstellt werden. Zur Erstellung des Graphen werden die Tasks zunächst in Abschnitte unterteilt. Jeder Abschnitt besteht dabei aus einer Folge von C-Anweisungen, die innerhalb, beziehungsweise außerhalb, eines für Interrupts gesperrten Bereichs liegen. Für die beiden Tasks *T1* und *T2* ergeben sich jeweils drei Abschnitte. Die Trennstellen der Abschnitte sind dabei die Aufrufe von *Disable-* und *EnableAllInterrupts*. Im EAG entspricht jeder Knoten genau einem solchen Abschnitt. Der EAG für das Beispielszenario ist in Abbildung 1 aufgeführt.

Jede Kante im gerichteten EAG entspricht, in Richtung der Pfeilspitze, einem einseitigen Ausschluss. Um die Darstellung im Graph zu vereinfachen, werden zusammengehörige Knoten zu Superknoten zusammengefasst. Eine ausgehende Kante von einem Superknoten ist eine vereinfachte Darstellung für ausgehende Kanten von allen Kindknoten des Superknotens. Dies gilt analog für eingehende Kanten.

Zu beachten ist hier, dass die Kanten durch zwei unterschiedliche Mechanismen zustande kommen. In der Abbildung werden über Task-Prioritäten entstehende einseitige Ausschlüsse dunkelgrau dargestellt. Schwarze Kanten stellen einseitige Ausschlüsse durch das Sperren von Interrupts dar. Zwei Knoten können mehrfach über Kanten verbunden sein, wobei jede Kante dann einem anderen Mechanismus entspricht. Beispielsweise schließt Knoten *T1_2* alle Knoten aus *T2* über seine höhere Priorität, sowie über den Interrupt-Sperrmechanismus aus.

Ein wechselseitiger Ausschluss zwischen Taskabschnitten kommt durch das Zusammenwirken zweier einseitiger Ausschlüsse zustande. Beispielsweise existiert ein wechselseitiger Ausschluss zwischen den Knoten *T1_2* und *T2_2*. Diese beiden Knoten stehen jeweils auch im wechselseitigen

Ausschluss mit *ISR1*. Letzteres ist zwingend für den Schutz des geteilten Puffers notwendig.

Der resultierende EAG aus Phase 1 beschreibt die potentielle Menge an wechselseitigen Ausschlüssen. Alle diese im Graph beschriebenen Einschränkungen sind im Singlecore-System vorhanden, werden aber nicht notwendigerweise für die Korrektheit benötigt.

Phase 2: Ziel in dieser Phase ist es, Kanten aus dem EAG aus Phase 1 entfernen zu können. Wie beschrieben, werden dazu die Speicherzugriffe der einzelnen Tasks analysiert.

Zur Analyse des Beispielprogramms wurde es für die MPC5533-Plattform kompiliert und die Binärdatei analysiert. Das Ergebnis der Speicherzugriffsanalyse ist in Tabelle I aufgeführt. Für die Speicherzugriffe werden jeweils die Anfangsadressen angegeben. Alle Zugriffe haben eine Breite von vier Bytes. In der Tabelle ist für jeden Taskabschnitt und jede aufgerufene Betriebssystemfunktion der entsprechende Zugriff auf den globalen Speicher aufgeführt. Der globale Speicher wird in der Hardware auf die Adressbereiche [0x40006000, 0x40006fff] abgebildet. Da sich alle Adressen in diesem Bereich befinden, werden die ersten vier Stellen (4000) durch das Kürzel p ersetzt.

Tabelle I
ERGEBNISSE DER SPEICHERZUGRIFFSANALYSE

Taskabschnitt	**Lesen**	**Schreiben**
T1_1		
T1 DisableAllInterrupts		p6060, p60d4
T1_2	[p60e0,p626c]	
T1 EnableAllInterrupts	p60d4	p6060
T1_3		
T1 TerminateTask	$\top$	$\top$
T2_1		
T2 DisableAllInterrupts		p6060, p60d4
T2_2	[p60e0,p626c]	
T2 EnableAllInterrupts	p60d4	p6060
T2_3		
T2 TerminateTask	$\top$	$\top$
ISR	[p6270,p63fc]	[p60e0,p626c]

Für das verwendete Beispiel liefert der *ValueAnalyzer* für die Tasks und die ISR jeweils Zugriffe auf den Speicherbereich von 0x400060e0 bis 0x4000626c. In diesem Adressbereich befindet sich das globale Array *buffer*. Innerhalb der *TerminateTask* Funktion wurden Speicherzugriffe ermittelt. Da *TerminateTask*, ebenso wie *DisableAllInterrupts* und *EnableAllInterrupts*, zum Betriebssystem gehört, betreffen diese Speicherstellen nicht den Anwendungscode. Es wird hier davon ausgegangen, dass das Betriebssystems auf der Zielplattform durch eine Multicore-fähige Version ausgetauscht wird.

Die erkannten Zugriffe auf den globalen Speicher sind exakt und besagen, dass *T1* und *T2* nur lesend auf den Puffer zugreifen, und dass *ISR1* schreibend auf den Puffer zugreift. Für die Bereiche der Tasks außerhalb der kritischen Abschnitte, werden keine Zugriffe auf globalen Speicher

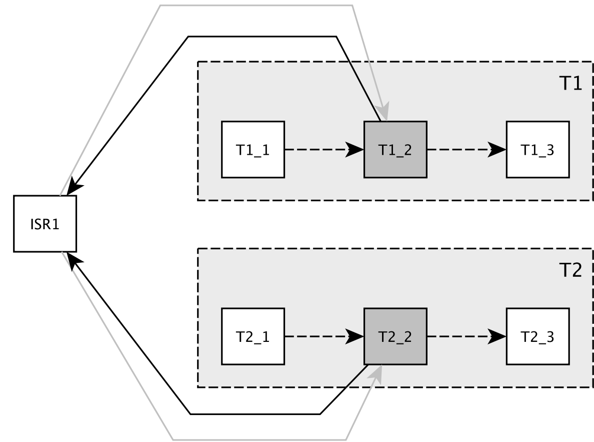

Abbildung 2. Durch Speicherzugriffe gerechtfertigte Ausschlüsse

gemeldet. Da auch diese Analyse auf Abstrakter Interpretation basiert, ist sichergestellt, dass jeder tatsächliche Speicherzugriff, der bei der konkreten Ausführung des Programms stattfindet, auch von der Analyse gemeldet wird. Es kann daher mit Sicherheit ausgeschlossen werden, dass in den beiden Tasks außerhalb der kritischen Abschnitte auf globalen Speicher zugegriffen wird. Die wechselseitigen Ausschlüsse zwischen Abschnitt $T2_2$ und den Abschnitten $T1_1$ und $T1_3$ kann daher als *vermutlich nicht erforderlich* klassifiziert werden.

Über die ermittelten Speicherzugriffe kann weiterhin auch der wechselseitige Ausschluss zwischen den Abschnitten $T1_2$ und $T2_2$ als *vermutlich nicht erforderlich* klassifiziert werden. In beiden Abschnitten wird zwar auf den selben Speicherbereich zugegriffen, allerdings nur lesend. Das Lesen der Daten aus *buffer* kann nebenläufig, respektive parallel, erfolgen und muss nicht durch einen wechselseitigen Ausschluss geschützt werden. Dieser wechselseitige Ausschluss schränkt die Parallelisierung unnötig ein.

Werden die wechselseitigen Ausschlüsse im Singlecore-System ausschließlich für den Schutz nebenläufiger Speicherzugriffe verwendet, bleiben nur die in Abbildung 2 gezeigten wechselseitigen Ausschlüsse nach Phase zwei übrig. Im Vergleich zum Ausgangsgraph aus Phase eins ist eine deutliche Reduktion von Einschränkungen erkennbar.

Phase 3: In dieser Phase kann nun eine vorgegebene AME-Partition bewertet werden. Bei der Migration auf ein Zweikernsystem könnte eine mögliche Partitionierung darin bestehen $ISR1$ und Task $T1$ an Kern eins und Task $T2$ an Kern zwei zu binden. Hier kann nun für jede Einschränkung entschieden werden, ob sie unter der gegebenen Partitionierung auch noch im Multicore-System vorhanden ist. Der wechselseitige Ausschluss zwischen $ISR1$ und $T1_2$ ist hierbei auch im Multicore-System vorhanden, da beide AMEs an den selben Kern gebunden wurden. Der wech-

selseitige Ausschluss zwischen $ISR1$ und $T2_2$ ist jedoch nicht mehr gegeben. Die höhere Priorität der ISR schließt nun nicht mehr Abschnitt $T2_2$ aus, da Prioritäten nicht über Kerngrenzen hinweg wirken. Ebenso hat das Sperren der Interrupts in $T2$ keine Wirkung auf $ISR1$. Soll diese Partitionierung realisiert werden, muss der wechselseitige Ausschluss zwischen $ISR1$ und $T2_2$ durch einen geeigneten Multicore-fähigen Mechanismus realisiert werden.

AUTOSAR Release 4.0 bietet lediglich Spin-Locks als expliziten Mechanismus für wechselseitigen Ausschluss über Kerngrenzen hinweg. Diese sind für die Verwendung in Echtzeitanwendungen nicht ohne weiteres geeignet, wegen der Gefahr von Verklemmungen (Deadlocks), Prioritätsumkehr (priority inversion) und potentiell unbegrenzter Blockadedauer über Kerngrenzen hinweg (unbounded remote blocking). Bessere Alternativen bieten hier Multicore-fähige Protokolle zum Sperren globaler Ressourcen. Diese müssen allerdings auf ihre Eignung für den Automobilbereich zunächst untersucht werden, wie in [8] und [9] dargestellt.

Im Vergleich zu einer manuellen Migration ist der hier vorgeschlagene automatisierte Ansatz vielversprechend. Eine naheliegende aber naive Strategie, die bei der manuellen Migration verfolgt werden könnte, wäre es alle Mechanismen durch Multicore-fähige Mechanismen zu ersetzen. Im Beispiel könnte das Sperren von Interrupts durch das Sperren einer einzigen globalen Ressource ersetzt werden. Alle ISRs sperren bei Eintritt diese Ressource und geben sie bei Austritt wieder frei. Alle *Disable-* und *EnableAllInterrupts*-Aufrufe werden durch Sperren und Freigeben dieser globalen Ressource ersetzt. Dieser einfache Ansatz ist zwar korrekt, würde die Parallelität aber massiv und unnötig einschränken. Der für die Korrektheit des Programms nicht notwendige wechselseitige Ausschluss zwischen den Abschnitten $T1_2$ und $T2_2$ würde beispielsweise bei dieser Ersetzung realisiert werden. Eine solche Ersetzungsstrategie dürfte letztlich die Performanz des Multicore-Systems auf die eines Singlecore-Systems reduzieren.

V. Zusammenfassung und Ausblick

In diesem Artikel wurde der Ansatz des Projektes ProSyMig zur automatisierten Migration von Echtzeitsoftware auf Multicore-Prozessoren beschrieben. Dabei liegt der Fokus zunächst auf der Einschränkungsklasse *wechselseitiger Ausschluss*. Für diese Einschränkungsklasse wurde das Zusammenspiel zweier statischer Programmanalysen vorgestellt, um einen möglichst hohen Grad an Parallelisierung zu ermöglichen. Der vorgestellte Ansatz geht hierbei konservativ vor. D.h. um die Korrektheit des Programms nicht zu gefährden, wird nur dann Parallelität erlaubt, wenn nachgewiesen werden kann, dass dies die Korrektheit des Programms nicht gefährdet.

Beispielhaft wurde hier der Ansatz an einer einfachen Anwendung erläutert, die wechselseitigen Ausschluss über das Sperren von Interrupts realisiert. Dieser Mechanismus ist nur einer von vielen, der zur Realisierung wechselseitiger Ausschlüsse in Singlecore-Systemen verwendet werden könnte.

Die nächsten Herausforderungen im Projekt bestehen darin, auch die Einschränkungsklassen *Reihenfolge-* und *zeitliche Einschränkung* durch geeignete Analysen abzudecken. Für diese Einschränkungssklassen wird es ungleich schwieriger werden geeignete Begründungen für oder gegen ihre Notwendigkeit zu finden.

Ein weiterer wichtiger Punkt ist die geeignete Visualisierung der Einschränkungen und Abhängigkeiten. Wie erläutert ist eine vollständig automatisierte Migration ohne Benutzereingriff nicht zielführend. Die Analyseergebnisse müssen dem Benutzer in geeigneter Form dargestellt werden, damit eine Bewertung möglich ist.

Literatur

[1] R. Wilhelm, J. Engblom *et al.*, “The worst-case execution-time problem–overview of methods and survey of tools,” *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, pp. 1–53, 2008.

[2] *OSEK/VDX Specification 2.2.3*, Std. [Online]. Available: http://www.osek-vdx.org/

[3] *AUTOSAR 4.0 Specification*, AUTOSAR Std., 2009. [Online]. Available: http://www.autosar.org

[4] P. Cousot and R. Cousot, “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, California: ACM Press, New York, NY, 1977, pp. 238–252.

[5] K. D. Scheidenmann, M. Knapp, and C. Stellwag, “Load Balancing in AUTOSAR-Multicore-Systemen,” *Elektroniknet*, 2010. [Online]. Available: http://www.elektroniknet.de/automotive/technik-know-how/test-entwicklungstools/article/26546/0/Load_Balancing_in_AUTOSAR-Multicore-Systemen_Teil_1/

[6] T. Kuhn, D. Barkowski, and R. Kalmar, “Software-Parallelisierung für Multicore-Hardware,” *ATZ online*, 2011. [Online]. Available: http://www.atzonline.de/index.php;do=show/alloc=3/id=12652

[7] J. Schneider, M. Bohn, and R. Rößger, “Migration of automotive real-time software to multicore systems: First steps towards an automated solution,” in *Proceedings Work-In-Progress Session of the 22th ECRTS*, July 6–9 2010, pp. 37–40.

[8] J. Schneider and C. Eltges, “Towards an evaluation infrastructure for automotive multicore real-time operating systems,” in *Proceedings of the 4th ISoLA - Volume Part II*, ser. ISoLA’10. Springer-Verlag, October 18–20 2010, pp. 483–486.

[9] J. Schneider, M. Bohn, and C. Eltges, “SimTrOS: A heterogenous abstraction level simulator for multicore synchronization in real-time systems,” in *WATERS 2011 in conjunction with ECRTS 2011 (to appear)*, July 5–8 2011.