# Extraktion von Interthread-Kommunikation in eingebetteten Systemen

Martin Wittiger Steffen Keul

Universität Stuttgart, Institut für Softwaretechnologie
Universitätsstr. 38, 70569 Stuttgart

{martin.wittiger, steffen.keul}@informatik.uni-stuttgart.de

**Abstract:** Mit der zunehmenden Verbreitung von Multicore-Rechnern werden Multicore-Architekturen auch in eingebetteten Systemen mehr und mehr Einzug halten. Zusätzlich zu den Schwierigkeiten der Softwareentwicklung für Singlecore-Plattformen müssen Software-Ingenieure somit die Herausforderungen bewältigen, bestehende Systeme zuverlässig und fehlerfrei auf Multicores zu portieren und dabei dennoch das Parallelisierungspotential möglichst effektiv zu nutzen. Bislang existiert kaum Werkzeugunterstützung, um diese Portierung in der Praxis durchzuführen. Unsere Arbeit verfolgt das Ziel, Algorithmen und Werkzeuge zu entwickeln, die existierende Steuersoftware im Automotive-Bereich semi-automatisiert auf Multicore-Plattformen portieren können. In diesem Beitrag wird eine statische Analysetechnik vorgestellt, mit der aus dem Quelltext eines eingebetteten Systems Kommunikationsgraphen extrahiert werden können. Diese können verwendet werden, um Modifikationsbedarf in bestehender Software zu identifizieren, und eignen sich als Grundlage für die spätere Partitionierung. Die vorgestellten Algorithmen wurden prototypisch in unserer Programmanalyse-Toolsuite Bauhaus implementiert und ihre prinzipielle Tauglichkeit wurde durch Anwendung auf bestehende industrielle Softwaresysteme bestätigt.

## 1 Einleitung

Moderne Anwendungen in eingebetteten Systemen, wie sie unter anderem im Automotive-Bereich eingesetzt werden, benötigen erhebliche Rechenleistung. Sie stehen auf der einen Seite dem beständigen Anwenderwunsch nach neuer und noch komfortablerer Funktionalität gegenüber, müssen jedoch auf der anderen Seite harte Anforderungen an Antwortzeiten erfüllen. Aufgrund der zunehmenden Verfügbarkeit von Multicore- und der zu erwartenden abnehmenden Verfügbarkeit leistungsfähiger Singlecore-Plattformen besteht der Wunsch, existierende Systeme auf Multicore-Plattformen lauffähig zu machen.

Daher soll nun Regelungssoftware, die für Singlecore-Systeme entwickelt wurde, auf Multicores portiert werden. Eine konservative Strategie mit geringem Portierungsaufwand besteht offensichtlich darin, die Software nur auf einem Kern auszuführen und weitere Kerne

Diese Arbeit wurde im Rahmen des Projektes ARAMiS durch das Bundesministerium für Bildung und Forschung (BMBF) unter dem Förderkennzeichen 01IS11035 gefördert. Die Verantwortung für den Inhalt liegt bei den Autoren.

ungenutzt zu lassen. Um aber rechenintensivere Probleme zu bearbeiten, sollte natürlich eine echte Parallelisierung angestrebt werden. Die Parallelisierung von vormals sequentiellem Programmcode birgt Risiken für die Zuverlässigkeit des Systems. In nebenläufigem Code können Race-Conditions auftreten, die fatales Fehlverhalten der Software bewirken können. Für eine Portierung müssen zu parallelisierende Einheiten identifiziert werden und es muss, z. B. durch statische Programmanalyse, nachgewiesen werden, dass diese Einheiten frei von Race-Conditions parallelisierbar sind [BSER11], oder es müssen entsprechende Modifikationen vorgenommen werden.

Ist eine Entscheidung getroffen, welche Programmteile parallel ausgeführt werden können, so muss im nächsten Schritt eine möglichst optimale Partitionierung, also die Verteilung von Threads auf Kerne, gefunden werden. Wir gehen davon aus, dass eine statische Zuweisung von Threads an Kerne der CPU erfolgen soll. Eine Partitionierung der Software weist also jedem Thread einen Kern zu, auf dem ihm Laufzeit zur Verfügung gestellt wird. Die Migration von Threads, wie sie bei dynamischem Scheduling erfolgt, ist ausgeschlossen, da ein solcher Systemaufbau nicht dem Stand der Technik zur Entwicklung sicherheitskritischer eingebetteter Systeme entspricht [RJFG10].

Die Qualität einer Partitionierung lässt sich anhand verschiedener Eigenschaften bemessen. Eine Partitionierung muss die Kerne der CPU möglichst gleichmäßig belasten – was auch beinhaltet, keinen Kern überzubelegen –, um Tasks die Möglichkeit zu geben, ihre Deadlines einzuhalten, und um kurze Antwortzeiten zu ermöglichen. Eine gleichmäßige Belastung vereinfacht auch eine spätere Erweiterung des Systems.

Wir gehen davon aus, dass Kern-interne Kommunikation Kern-übergreifender Kommunikation vorzuziehen ist. Als Vorteile Kern-interner Kommunikation konnten wir identifizieren:

- Ist bekannt, dass bestimmte Tasks auf dem gleichen Kern laufen werden, so hat dies bei der Synchronisation zwischen ihnen verlaufender Kommunikation Vorteile. Solche Kern-interne Kommunikation kann möglicherweise durch sonst nur Singlecore-Systemen vorbehaltene Synchronisationsmuster (zum Beispiel Deaktivierung von Interrupts) abgesichert werden, oder es kann sogar ganz auf explizite Synchronisation verzichtet werden, weil gegenseitiger Ausschluss schon aus dem Nebenläufigkeitsmodell hergeleitet werden kann. Dabei sinkt der Portierungsaufwand, und Leistungseinbußen durch Synchronisation werden vermieden [SKS10].
- In AUTOSAR 4.0 ist der `SetResource`-/`GetResource`-Mechanismus nur für die Synchronisation auf einem einzigen Kern einsetzbar, sodass zur Synchronisation von gemeinsamen Speicherbereichen zwischen verschiedenen Kernen nur Spin-Locks eingesetzt werden können. Diese bedingen jedoch ein andauerndes wiederholtes Prüfen der Verfügbarkeit des Spin-Locks (Busy-Waiting) und können somit eine geringere Auslastung der einzelnen Kerne erzwingen.
- Auch vermuten wir, dass Software, die vorwiegend Kern-interne Kommunikation verwendet, besser von Caches profitieren kann – insbesondere bei Schreibzugriffen.

Aufgrund dieser Annahme ist in unserem Modell Kommunikation zwischen Kernen generell unerwünscht. Anders ausgedrückt: Greifen unterschiedliche Threads auf dieselben

globalen Variablen zu, ist es wünschenswert, dass diese Threads auf dem gleichen Kern laufen.

Darüber hinaus kann es weitere Einschränkungen geben, denen eine Partitionierung entsprechen soll. So kann der Entwickler aufgrund (lizenz-)rechtlicher, technischer oder architektonischer Gegebenheiten die Anforderung haben, dass bestimmte Threads einem spezifischen Kern oder zumindest dem gleichen Kern zugeordnet werden. Denkbar sind auch gegensätzliche Einschränkungen: Tasks von unterschiedlichen Zulieferern sollen auf keinen Fall auf dem gleichen Kern ausgeführt werden.

## 2 Analyse

Wir gehen im Folgenden davon aus, dass der Benutzer die gewünschte Parallelität in einem eingebetteten System planen und potentiell parallele Einheiten identifizieren kann. Die Spezifikation der gewünschten Parallelität kann durch geeignete Mittel direkt im Quelltext des zu analysierenden Systems oder durch eine separate Konfigurationsdatei erfolgen. Hierzu müssen die Startfunktionen der einzelnen Threads angegeben werden.

Als konkreter Thread sollen alle Programmteile gelten, die parallel zueinander ausgeführt werden können. Zur Extraktion von Abhängigkeiten zwischen nebenläufigen Programmeinheiten führen wir den Begriff eines abstrakten Threads ein. Ein abstrakter Thread ist die Repräsentation von konkreten Threads in der statischen Programmanalyse. Die Unterscheidung zwischen konkreten und abstrakten Threads kann von Bedeutung sein, falls in einem Programm zur Laufzeit neue Threads erzeugt werden können und möglicherweise die Anzahl der konkreten Threads nicht präzise abgeschätzt werden kann. Als abstrakter Thread gilt also:

- der Boot-Loader, d. h. der Programmteil, der zum Start des eingebetteten Systems mit der Ausführung beginnt und evtl. das Betriebssystem startet,
- Interrupt-Handler, also Funktionen des Programms, die als Reaktion auf externe Ereignisse aufgerufen werden, inkl. aller davon aufgerufenen Funktionen,
- durch das Betriebssystem verwaltete Programmeinheiten (in OSEK/VDX OS „Task“ genannt),
- evtl. durch API-Funktionen wie z. B. der POSIX-Funktion `pthread_create` dynamisch gestartete Threads. In unserem Modell gelten alle auf diese Weise aus einer Programmstelle erzeugten Laufzeit-Threads als ein abstrakter Thread. Diese Variante wird in eingebetteten Systemen typischerweise nicht verwendet.

Unser Programmanalyse-Modell identifiziert die abstrakten Threads in der Software und erzeugt für jeden einen Aufrufgraphen. Im Aufrufgraphen existiert jede Funktion als Knoten und jeder Funktionsaufruf als Kante von der aufrufenden zur aufgerufenen Funktion.

Um den Aufrufgraphen aufzubauen, müssen daher Informationen über die möglichen Ziele von Zeigern in dem Programm vorliegen, um indirekte Funktionsaufrufe durch Funkti-

onszeiger aufzulösen. Hierbei verlangen wir, dass die eingesetzten Analysen konservativ sind. Die Zeigeranalyse ist konservativ, falls sie garantiert alle möglichen Zeigerziele berechnet. Dabei dürfen aber die Zielmengen überschätzt und Zeigerziele annotiert werden, die in keiner realen Ausführung des Programms tatsächliche Ziele sein können. Diese Konservativität ist notwendig, da fast alle Fragestellungen der Programmanalyse im Allgemeinen unentscheidbar sind. Unsere Zeigeranalyse orientiert sich an der fluss- und kontextinsensitiven Analyse nach Pearce et al. [PKH04].

Unsere Analysewerkzeuge sind als Teil der Programmanalyse-Suite Bauhaus [RVP06] implementiert. Unsere Analysealgorithmen werden auf eine abstrakte Programmzwischendarstellung angewendet, sodass eine beliebige Programmiersprache als Quelle für die Analyse dienen kann. Aufgrund der weiten Verbreitung der Programmiersprache C im Bereich eingebetteter Systeme beschränken wir unsere Beschreibungen auf diese Quellsprache.

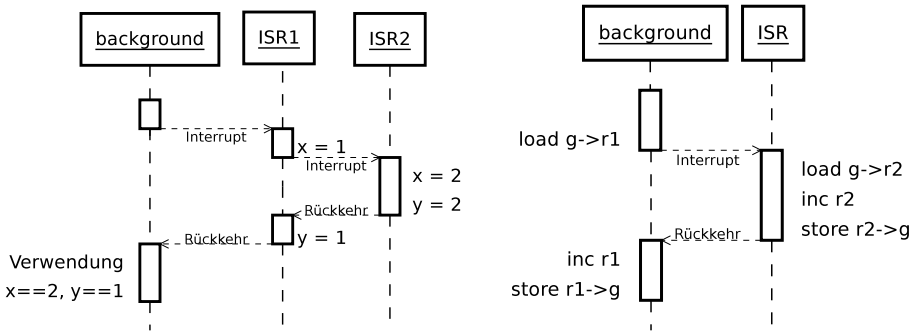
Nachdem die gewünschte Parallelisierung festgelegt ist und die Zwischendarstellung für das Programm erzeugt wurde, sollen nun die Analysen wie folgt angewendet werden. Zunächst wird das nebenläufige Programm mit einer Race-Condition-Analyse auf gravierende Kommunikationsfehler hin untersucht, wie in Abschnitt 2.1 beschrieben. Sind keine Fehler (mehr) vorhanden, erfolgt der Aufbau des in Abschnitt 2.2 beschriebenen Zugriffsgraphen. Der dazu verwendete Extraktionsalgorithmus wird in Abschnitt 2.3 präsentiert.

## 2.1 Data-Race-Analyse

Die grundlegenden Probleme, die durch Race-Conditions entstehen, sind verlorene Aktualisierung und das Lesen inkonsistenter Datenstrukturen. In Abbildung 1(a) werden inkonsistente Verwendungen illustriert. Zwei Interrupt-Handler, `ISR1` und `ISR2`, aktualisieren zwei Variablen, `x`, `y`. Durch die Verzahnung der einzelnen Zuweisungen werden im Hintergrund-Task jeweils die Aktualisierung aus `ISR1` an `y` und aus `ISR2` an `x` sichtbar. Existiert zwischen den beiden Variablen eine funktionale Abhängigkeit, so liegt eine Inkonsistenz vor, die zu einer Fehlfunktion des Systems führen kann. In Abbildung 1(b) hingegen wird parallel in dem Interrupt-Handler `ISR` und in dem Hintergrund-Task `background` eine Inkrementierung der globalen Variable `g` ausgeführt. Dazu wird der gerade aktuelle Wert von `g` jeweils in ein Register (`r1` oder `r2`) geladen und inkrementiert. Dann wird das Resultat wieder in `g` gespeichert. Durch die Verzahnung der einzelnen Anweisungen geht die Aktualisierung durch den `ISR` verloren und nur eine Inkrementierung der Variablen hat einen Effekt, was in vielen Fällen nicht die beabsichtigte Semantik darstellt.

Ein Data-Race ist eine Situation, in der mehrere Threads ohne Synchronisation auf eine Speicherstelle zugreifen und mindestens einer dieser Threads die Speicherstelle verändert. Die Data-Races sind eine Teilmenge der Race-Conditions. Es existieren einige Programmanalysatoren, die in der Lage sind, Data-Races zu identifizieren [PFH06, NAW06, VJL07]. Es existieren auch Forschungsarbeiten, die gezielt Data-Race-Analysatoren für den Automotive-Bereich bei Einsatz eines OSEK/VDX OS untersuchen [Keu11, SSV+11].

Während Data-Races als einzelne unzureichend synchronisierte Speicherzugriffe definiert sind, können bei einer Portierung auch komplexere Race-Conditions auftreten, deren Feh-




(a) inkonsistente Verwendung (b) verlorene Aktualisierung

Abbildung 1: Beispiele für Auswirkungen von Race Conditions

lerpotential nicht einfach erkennbar ist. Es existieren jedoch zumindest Ansätze zur Erkennung von Higher-Level Data-Races [AHB03, RFP11] und zur Identifikation von unerwünschtem Datenfluss über Thread-Grenzen hinweg [Keu10].

Auf der Zwischendarstellung des Programms kann eine Race-Condition-Analyse durchgeführt werden, um die Korrektheit der gewünschten Parallelisierung zu prüfen oder gegebenenfalls Korrekturen am Quelltext vorzunehmen, um die Korrektheit herzustellen.

## 2.2 Zugriffsgraphen

Um die Aufgabe der Partitionierung eines Systems zu beschreiben, führen wir als neue Datenstruktur den Zugriffsgraphen ein. Dieser Graph soll die Kopplung zwischen den Threads beschreiben. Er kann automatisiert in den Kommunikationsgraphen (siehe Abschnitt 3) transformiert werden.

Der Zugriffsgraph enthält zwei verschiedene Arten von Knoten, abstrakte Threads und Shared Variables. Als Shared Variable bezeichnen wir alle Speicherbereiche, die potentiell von mehr als einem konkreten Thread verwendet werden. Als Verwendung gilt das Lesen oder das Schreiben des Werts.

Der Zugriffsgraph ist bipartit. Für jedes Paar eines abstrakten Threads und einer Shared Variable wird eine Kante erzeugt, falls der abstrakte Thread auf diese Shared Variable zugreift. Weitere Kanten existieren nicht.

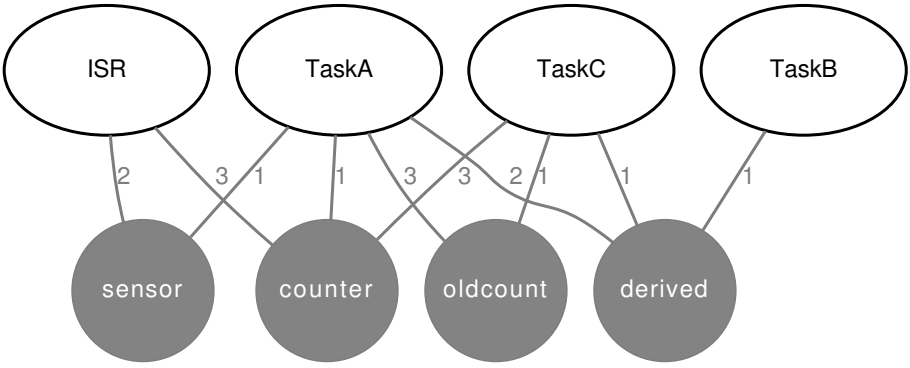
Kanten werden mit einem Gewicht annotiert, das die Kopplung einer Shared Variable an einen Thread beschreibt. Ein großes Gewicht soll die Threads, die auf eine gemeinsame Shared Variable zugreifen, in einer nachfolgenden Partitionierung tendenziell auf einen gemeinsamen Kern zusammenziehen.

Als mögliche Bewertungen für das Gewicht einer Kante $(T, V)$ zwischen einem abstrakten Thread $T$ und einer Shared Variable $V$ haben wir identifiziert:

```
1  int counter = 0;
2  int oldcount = -1;
3  int sensor = 0;
4  int derived = 0;
5
6  void ISR()
7  {  sensor = read();
8     counter++;
9  }
10
11 void TaskA()
12 {  if (oldcount < counter) {
13        derived = work(sensor);
14        oldcount = counter;
15 }  }
```

```
1  void TaskB()
2  {
3     log(derived);
4  }
5
6  void TaskC()
7  {
8     int reset;
9     if (oldcount == counter) {
10       reset = process(derived);
11       if (reset) {
12          counter = 0;
13       }
14    }
15 }
```

(a) Beispielquelltext




(b) Zugriffsgraph

Abbildung 2: Beispiel für die Extraktion eines Zugriffsgraphen

```
1  DeclareResource(M);
2  int global;
3
4  void Task_A()
5  {  GetResource(M);
6     if (global > 7) {
7        ...
8     }
9     ReleaseResource(M);
10 }
```

```
11 void TaskB()
12 {
13    while (...) {
14       GetResource(M);
15       current = global;
16       ...
17       global = current;
18       ReleaseResource(M);
19 }  }
```

Abbildung 3: Beispiel für Zugriffe mit konsistenter Sperrung

1. Ein konstantes Gewicht $c_r$, falls $T$ mindestens einen lesenden Zugriff auf $V$ enthält, ein konstantes Gewicht $c_w$, falls $T$ einen schreibenden Zugriff auf $V$ enthält, und $c_r + c_w$, falls beide Arten in $T$ enthalten sind.

2. Die Anzahl der Zugriffsstellen im Programmcode, an denen $T$ auf $V$ zugreift.

3. Um die tatsächliche Häufigkeit eines Zugriffs stärker zu gewichten, kann zusätzlich die Tiefe der Schachtelung in Schleifen berücksichtigt werden. Liegt in diesem Fall ein Zugriff auf eine Shared Variable innerhalb einer Funktion, die aus einer Schleife heraus aufgerufen wird, so wird dieser Zugriff stärker bewertet als ein Zugriff aus rein sequentiellem Code.

4. Aus ähnlicher Motivation heraus kann auch die Zugriffshäufigkeit eines konkreten Threads $T$ auf $V$ in bestimmten Testausführungen der Software durch dynamische Analyse ermittelt und als eine Metrik verwendet werden.

5. Eine Metrik, die stärker auf Wartezeiten der Threads fokussiert, gewichtet Zugriffe auf Semaphoren $V$ besonders, indem der Kante $(T, V)$ das Gewicht der Zugriffe auf alle Shared Variables innerhalb der kritischen Region zugewiesen wird. Diese Metrik zielt darauf ab, kritische Regionen möglichst demselben Kern zuzuteilen. In diesem Fall kann, je nach Nebenläufigkeits-Modell, dadurch die kritische Region sogar überflüssig werden und in sequentiellen Code ohne Synchronisation übersetzt werden.

6. Sind alle Zugriffe auf eine oder mehrere Shared Variables stets durch Sperrung einer Semaphore konsistent in allen Threads geschützt, so kann die Gewichtung für die Shared Variables ausschließlich der Kante zu der Semaphore zugeschlagen werden. Dies kann in unserer Analysekette einfach durch Anwendung der klassischen Lockset-Analyse [Keu11] erreicht werden. In Abbildung 3 werden Zugriffe auf die Semaphore `M` durch Kanten mit besonders hohem Gewicht im Zugriffsgraphen repräsentiert, und die Zugriffe auf die Shared Variable `global` verstärken das Gewicht dieser Kanten.

Ein Beispiel für einen Zugriffsgraphen wird in Abbildung 2 gezeigt. In dem Beispielquelltext sind vier Funktionen definiert, `ISR`, `TaskA`, `TaskB` und `TaskC`, die separate

parallelisierbare Prozesse einer Steuersoftware darstellen sollen. Sie greifen auf gemeinsame globale Variablen zu. In dem Zugriffsgraphen werden die Threads als helle, ovale Knoten angezeigt und die Shared Variables als dunkle, kreisförmige Knoten. Durch die Kanten wird angezeigt, welche Zugriffe tatsächlich durchgeführt werden. Für die Kantengewichtung haben wir eine sehr einfache Strategie gewählt: Zu Beginn ist das Gewicht der Kante $0$. Gibt es mindestens einen lesenden Zugriff, wird das Gewicht der Kante um eins erhöht. Analog dazu wird, wenn es mindestens einen schreibenden Zugriff gibt, das Gewicht um $2$ erhöht. Es werden nur Kanten mit Gewicht $> 0$ angelegt. Somit ist z. B. erkennbar, dass im Beispiel der Thread `TaskB` lesend auf die Shared Variable `derived` zugreift (in Zeile 10), oder dass der Thread `TaskA` sowohl lesend als auch schreibend auf die Shared Variable `oldcount` zugreift (in den Zeilen 12 und 14).

### 2.3 Extraktionsalgorithmus

Zum Aufbau der Zugriffsgraphen werden die Aufrufgraphen der abstrakten Threads nacheinander durchlaufen. Innerhalb einer Funktion werden die Zugriffsstellen mit Zugriffen auf Shared Variables einzeln betrachtet und gegebenenfalls eine Kante in den Zugriffsgraphen eingefügt. Das Resultat der Zeiger-Analyse wird verwendet, um die Ziel-Variablen an Dereferenzierungen zu bestimmen.

Zur Ermittlung des Kantengewichts werden entsprechende Annotationen an den Kanten vorgenommen. Die Gewichtungsfunktion ist parametrierbar ausgeprägt, um verschiedene Varianten evaluieren zu können.

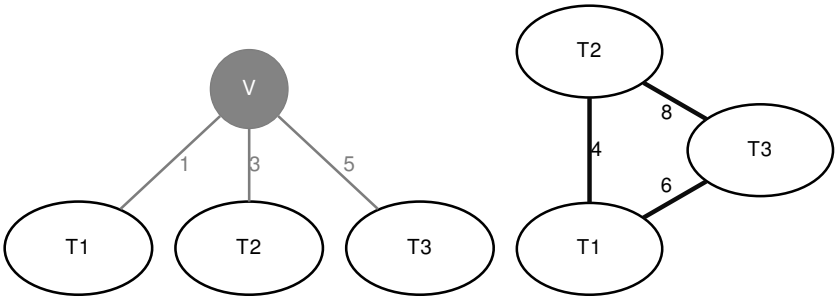
In unserer prototypischen Implementierung wird Strategie 1 aus Abschnitt 2.2 umgesetzt: Gewicht $1$ wird für die Existenz eines lesenden Zugriffs und Gewicht $2$ für die Existenz eines schreibenden Zugriffs annotiert.

## 3 Kommunikationsgraphen

Als Datenstruktur zur Partitionierung eines Systems definieren wir den Kommunikationsgraphen. Er kann aus dem in Abschnitt 2.2 beschriebenen Zugriffsgraphen abgeleitet werden.

Um den Zugriffsgraphen in den Kommunikationsgraphen umzuwandeln, müssen die Shared Variables aufgelöst und durch Kommunikationskanten zwischen Threads ersetzt werden.

Eine Shared Variable, die über $n$ Kanten mit den Gewichten $w_1 \ \ldots \ w_n$ mit $n$ Threads verbunden ist, wird zu $m = n(n-1)/2$ Kommunikationskanten zwischen den $n$ Threads. Die Gewichte dieser $m$ Kanten muss der Algorithmus berechnen. Hier sind verschiedene Berechnungswege möglich. Eine einfache Umsetzung weist der Kommunikationskante zwischen zwei Threads die Summe der Zugriffskantengewichte zu. Diese Variante lässt sich wie folgt beschreiben:




(a) Ausgangssituation (Zugriffsgraph) (b) Resultat (Kommunikationsgraph)

Abbildung 4: Auflösung einer Variablen mit Zugriffen aus drei Threads

Hat eine Shared Variable $V$ Zugriffskanten zu den Threads $T_1, \ldots, T_n$, wobei $\forall i.\, 1 \leq i \leq n$ die Kante $(V,\ T_i)$ das Gewicht $w_i$ hat, fügt der Algorithmus dem Graphen $\forall i, j.\, 1 \leq i < j \leq n$ eine Kommunikationskante $(T_i,\ T_j)$ mit Gewicht $w_i + w_j$ hinzu. Ist eine Kommunikationskante, die hinzugefügt werden soll, bereits vorhanden, wird das Gewicht der bestehenden Kante einfach entsprechend erhöht.

Abbildung 4 zeigt die Auflösung einer Variablen nach diesem Schema, auf die von $n = 3$ Threads aus zugegriffen wird.

Verschiedene Varianten dieser Heuristik sind anwendbar. Statt der Summe könnte auch das Maximum der beiden Werte verwendet werden. Auch können die Gewichte der Zugriffskanten, die die Variable mit anderen Threads verbinden, in die Berechnung mit eingehen. So könnte beispielsweise die Summe aller Zugriffskantengewichte multipliziert mit einem geeigneten Faktor in die Berechnung mit einfließen, um zu berücksichtigen, dass auch wenige Zugriffe auf eine hoch frequentierte Variable erheblichen Einfluss haben können.

Kommunizieren zwei Threads über mehr als eine Variable miteinander, so existiert im Grundalgorithmus trotzdem nur eine Kommunikationskante zwischen den Threads. Ihr Gewicht entspricht dann der Summe der Gewichte, die sich aus der Auflösung aller im Zugriffsgraphen benachbarten Variablen ergeben.

Der in Abbildung 5 dargestellte Kommunikationsgraph entsteht aus dem in Abbildung 2(b) dargestellten Zugriffsgraphen durch Anwendung des im obigen Beispiel beschriebenen Algorithmus. So ergibt sich beispielsweise das Gewicht der Kante zwischen den Threads `ISR` und `TaskA` aus der Summe der Kommunikation über die Variablen `sensor` ($2+1 = 3$) und `counter` ($3 + 1 = 4$) – zusammen also 7.

Zusätzlich wurde in der Abbildung eine Abschätzung der CPU-Utilisation der einzelnen Tasks annotiert. Diese muss vom Benutzer eingegeben werden und kann beispielsweise aus WCET-Werkzeugen übernommen werden oder aus empirischen Tests der Software stammen.

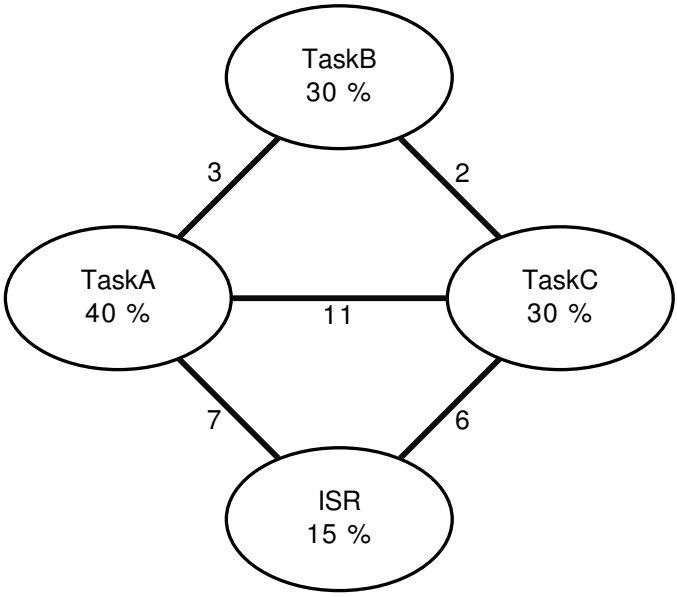



Abbildung 5: Kommunikationsgraph, aus dem Zugriffsgraphen in Abbildung 2(b) hergeleitet

# 4 Anwendungen des Kommunikationsgraphen

Der Kommunikationsgraph eines eingebetteten Systems stellt eine abstrakte Beschreibung des Datenaustauschs zwischen den einzelnen Threads dar.

## 4.1 Architekturdokumentation

Der Kommunikationsgraph kann eingesetzt werden, um die Kopplung zwischen Threads explizit zu dokumentieren und die Einhaltung von Architektur-Vorgaben zu prüfen. Ist vorgegeben, dass zwei Threads in einem System stets vollständig unabhängig voneinander agieren sollen, so kann die Einhaltung dieser Vorgabe durch das Fehlen einer Kante zwischen diesen Threads verifiziert werden.

## 4.2 Ermittlung von Synchronisationsbedarf

Steht nur eine Hypothese für eine möglicherweise geeignete Parallelisierung eines Systems zur Verfügung, so kann von diesem hypothetischen System ein Kommunikationsgraph erzeugt werden. Anhand des Graphen kann evaluiert werden, ob in dem hypothetischen System alle Kommunikationswege berücksichtigt wurden und insbesondere, ob die

Änderungen an der Singlecore-Implementierung vollständig durchgeführt wurden.

Zusätzlich kann die Möglichkeit geschaffen werden, die Kanten des Kommunikationsgraphen mit den Quelltext-Positionen zu annotieren, die diese Kanten hervorrufen. Dadurch wird es möglich, iterativ Hypothesen für geeignete Parallelisierungen zu testen und somit die Kosten einer Änderung des Systems gegenüber dem erwarteten Gewinn durch eine Parallelisierung abzuwägen.

### 4.3 Partitionierung

Eine weitere Verwendungsmöglichkeit für Kommunikationsgraphen ergibt sich bei der Partitionierung. Wir haben ein Werkzeug entwickelt, das basierend auf dem Kommunikationsgraphen die Software partitioniert. Hierbei beschränken wir uns auf die beiden am zentralsten erscheinenden Kriterien. Neben dem Kommunikationsgraphen wird eine Abschätzung der CPU-Utilisation der einzelnen Threads in die Software eingegeben. Das Partitionierungsproblem besteht für uns demnach darin, einen guten Kompromiss zwischen Gleichbelastung der Kerne und Minimierung der Interkern-Kommunikation zu finden. Im Idealfall werden alle Kerne gleich belastet und es findet keine Interkern-Kommunikation statt.

Für unsere prototypische Implementierung haben wir eine Bewertungsfunktion gefunden, die diese beiden Kriterien berücksichtigt. Sie lässt sich durch Parameter beeinflussen. So kann beispielsweise vom Nutzer vorgegeben werden, wie wichtig die Gleichbelastung der Kerne im Vergleich zur Minimierung der Kommunikation ist. Die Bewertungsfunktion weist jeder möglichen Partitionierung einen Wert zu, der umso niedriger ist, je besser die Partitionierung den Bewertungskriterien entspricht.

Es bleibt nun eine, gemessen an der Bewertungsfunktion näherungsweise optimale, Partitionierung zu finden. Hierbei handelt sich um ein numerisches Optimierungsproblem. Dieses lässt sich näherungsweise mit Simulated Annealing [KGV83, SK06] lösen. Unsere Implementierung kann damit Graphen, die aus industriell eingesetzter, eingebetteter Steuerungssoftware resultieren, auf üblichen Entwicklungssystemen in realistischer Zeit bearbeiten. Die berechnete Lösung wird dem Entwickler als Partitionierung vorgeschlagen.

## 5 Zusammenfassung und Ausblick

Wir haben in dem vorliegenden Beitrag eine Strategie skizziert, wie eine Migration sicherheitskritischer Software von Singlecore- zu Multicore-Plattformen durchgeführt werden kann. Nach einer Auswahl parallel ausführbarer Threads werden die Entwickler durch Programmanalyse bei ihrer Portierungsaufgabe unterstützt. Mittels Race-Condition-Analyse kann die Abwesenheit von Synchronisationsfehlern in Software validiert werden. Durch Extraktion des Kommunikationsgraphen wird eine automatisierte Generierung geeigneter Partitionierungen ermöglicht. Darüber hinaus haben wir mit dem Kommunikationsgraphen

eine Datenstruktur entworfen, die weitere wichtige Funktionen, besonders in der Architektur-Dokumentation und -Prüfung, übernehmen kann.

Als Fortsetzung dieser Arbeit wollen wir uns mit der Frage befassen, wie eine optimale Partitionierung erreicht werden kann. Wir vermuten, dass durch eine geeignete Wahl der Heuristiken zur Berechnung der Kantengewichte im Zugriffsgraphen bereits sehr gute Partitionierungen möglich werden.

Wir erwarten auch, dass unsere Technik durch iteriertes Prüfen von Parallelisierungs-Hypothesen wertvolle Hinweise zur Verteilung der einzelnen Tasks und Interrupt-Handler auf Threads geben kann.

## Literatur


[AHB03] Cyrille Artho, Klaus Havelund und Armin Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, November 2003.

[BSER11] Michael Bohn, Jörn Schneider, Christian Eltges und Robert Röger. Migration von AUTOSAR-basierten Echtzeitanwendungen auf Multicore-Systeme. In Fachgruppen des Fachbereichs Softwaretechnik der Gesellschaft für Informatik e.V., Hrsg., *Softwaretechnik-Trends*, Band 31 / 3, Seiten 9–14. Köllen Druck & Verlag GmbH, Bonn, 2011.

[Keu10] Steffen Keul. Static Versioning of Global State for Race Condition Detection. In *Ada-Europe '10*, Band 6106 der *LNCS*, Seiten 111–124. Springer, 2010.

[Keu11] Steffen Keul. Tuning Static Data Race Analysis for Automotive Control Software. In *SCAM '11*, Seiten 45–54. IEEE Computer Society, 2011.

[KGV83] Scott Kirkpatrick, C. Daniel Gelatt und Mario P. Vecchi. Optimization by simulated annealing. *Science*, (220):671–680, 1983.

[NAW06] Mayur Naik, Alex Aiken und John Whaley. Effective Static Race Detection for Java. In *PLDI '06*, Seiten 308–319, New York, NY, USA, 2006. ACM.

[PFH06] Polyvios Pratikakis, Jeffrey S. Foster und Michael Hicks. LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In *PLDI '06*, Seiten 320–331, New York, NY, USA, 2006. ACM.

[PKH04] David J. Pearce, Paul H. J. Kelly und Chris Hankin. Efficient field-sensitive pointer analysis for C. In *PASTE '04*, Seiten 37–42, New York, NY, USA, 2004. ACM.

[RFP11] Aoun Raza, Stefan Franke und Erhard Plödereder. Detecting High-Level Synchronization Errors in Parallel Programs. In Alexander Romanovsky und Tullio Vardanega, Hrsg., *Reliable Software Technologies – Ada-Europe 2011*, Band 6652 der *LNCS*, Seiten 17–30, Heidelberg, 2011. Springer.

[RJFG10] Kai Richter, Marek Jersak, Christian Ferdinand und Peter Gliwa. Eine ganzheitliche Methodik für den automatisierten Echtzeit-Nachweis zur Absicherung hoch integrierter, sicherheitskritischer Software-Systeme. In Hubert B. Keller, Erhard Plödereder, Peter Dencker und Herbert Klenk, Hrsg., *Automotive - Safety & Security 2010*, Seiten 27–41, Aachen, 2010. Shaker.

[RVP06] Aoun Raza, Gunther Vogel und Erhard Plödereder. Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In *Ada-Europe 2006*, Band 4006 der *LNCS*, Seiten 71–82. Springer, 2006.

[SK06] Johannes Josef Schneider und Scott Kirkpatrick. *Stochastic optimization*. Springer, Berlin, 2006.

[SKS10] Kathrin D. Scheidenmann, Michael Knapp und Claus Stellwag. Load Balancing in AUTOSAR-Multicore-Systemen. *Elektroniknet*, (3/2010):22–25, 2010.

[SSV+11] Martin D. Schwarz, Helmut Seidl, Vesal Vojdani, Peter Lammich und Markus Müller-Olm. Static analysis of interrupt-driven programs synchronized via the priority ceiling protocol. In *POPL '11*, Seiten 93–104, New York, NY, USA, 2011. ACM.

[VJL07] Jan Wen Voung, Ranjit Jhala und Sorin Lerner. RELAY: Static Race Detection on Millions of Lines of Code. In *ESEC-FSE '07*, Seiten 205–214, New York, NY, USA, 2007. ACM.