TEC: Time-Enhanced C

Erweiterung einer imperativen Programmiersprache um Zeitvorgaben

René Fritzsche, Günter Kemnitz, Christian Siemers
Institut für Informatik - Technische Universität Clausthal
{rene.fritzsche, guenter.kemnitz, christian.siemers}@tu-clausthal.de

ABSTRACT:
Zeitangaben in Programmen für eingebettete Systeme sind zur Analyse und Umsetzung von Echtzeitvorgaben unvermeidbar. Derzeitige Implementationen stellen entweder eine komplett neue Programmiersprache vor oder untersuchen den fertig-compilierten Code auf seine WCET. Der vorgestellte Ansatz erweitert die Sprache C um Zeitangaben und ermöglicht eine Codegeneration in Anhängigkeit dieser Vorgaben, wobei dieses quasi-statische Scheduling durch zur Laufzeit veränderliche Variablen noch gesteuert werden und somit auf äußere Einflüsse reagiert werden kann.

EINLEITUNG:
Eingebettete Systeme interagieren mit ihrer Umwelt durch Aufnahme von Kommunikations- und Sensorwerten sowie der Ausgabe von Ergebnissen auf Anzeigen, Schnittstellen oder Steuer- & Regelungsanweisungen.
Die Software eines eingebetteten Systems ist typischerweise durch nebenläufige, konkurrierende Aufgaben charakterisiert, welche über Bedingungen zeitlich gekoppelt sind. Reaktionen auf externe Ereignisse haben zudem bei Echtzeitsystemen eine vorgegebene maximale Reaktionszeit, die zwischen den anderen Aufgaben im Hinblick auf die Sicherheit des Systems und ausgehender Gefahren eingehalten werden muss.
Ein übergeordnetes Betriebssystem (z.B. RTOS) für diese Prozessverwaltung wird häufig aufgrund der begrenzten Ressourcen eingespart und die Abläufe sowie das Scheduling der Tasks durch den Kontrollfluss im einzigen Programm realisiert. Aus diesem Grund liegt die Herausforderung beim Programmierer, mithilfe von Interrupts und allgemeinen Kontroll-strukturen dieses Scheduling zu ermöglichen. Hierfür wird neben der Beschreibung der Schnittstellen zwischen den Funktionen ebenfalls deren zeitlicher Aufwand benötigt.
Eine Alternative stellt das Scheduling zur Compile-Zeit dar, bei der der Precompiler die Tasks ineinander verschränkt und so statisch Code erzeugt, der die festen zeitlichen Vorgaben erfüllt. Im Folgenden wird ein möglicher Algorithmus für diese Codegenerierung der Sprache C vorgestellt – eine Abschätzung der Laufzeit einzelner Funktionen und Anweisungen ist hier ebenfalls notwendig.

VORARBEITEN:
Für die Laufzeit-Abschätzungen wird gefordert, dass die Programme einer gewissen Struktur unterliegen und komplizierte, fehleranfällige Konstrukte der Sprache C vermeiden – denn diese sind mit automatischen Parsern und Analysetools im Bezug auf die Umsetzung in Assemblercode nach der Compilierung schwer zu überprüfen. Coding-Standards wie *CleanC*[2] und *MISRA*[4] versuchen hier, die Randeffekte unsauberer C-Programmierung vermeiden zu helfen und dabei der Software ein erhöhtes Maß an Verlässlichkeit, Wartbarkeit und Portabilität zu ermöglichen.
Eine theoretische Erweiterung der Sprache C wurde durch *TimeC* [1] eingeführt, mit der sich zeitliche Abstände zwischen zwei, durch spezielle Kommentarzeilen symbolisierte, Markern beschreiben lassen. Auf diesem Ansatz baut das Konzept zu *TEC* auf, wobei die zusätzlichen Timing-Statements sowohl zur Analyse der Zeitvorgaben und ebenfalls als Basis zur Codegeneration verwendet werden. Im Gegensatz zu TimeC werden die Zeitvorgaben direkt

hinter dem jeweiligen Befehl eingefügt, was einer schnellen Erfassung durch einen einzigen Durchlauf mittels Parser ermöglicht.

TIME-ENHANCED C:
Die Generation von Code aus einer abstrakten Sprache heraus ist nichts Neues - die Erstellung der Strukturen zur Einhaltung von Zeitvorgaben hingegen schon. Im Konzept steht zunächst die Anordnung und Strukturierung der zu planenden Funktionen im Vordergrund. Es handelt sich hierbei um Scheduling-Angaben auf C-Codebasis [*Bild 1*].

```
StoreData(Value) /* @_TEC_@ triggered IO-Value */
DFTready = ComputeDFT(Num, Data) /* @_TEC_@ periodic 300ms */
AnalyseDFT(Coeff) /* @_TEC_@ triggered DFTready */
ShortAnalysis(Data) /* @_TEC_@ periodic 10ms */
Ready = DND(Value) /* @ TEC @ uninterruptible */
```

Bild 1 – TEC-Kommentare hinter Funktionsaufrufen.

Für diese Aufgabe werden alle Funktionen des Programms voneinander gelöst und nur durch Rückgabeparameter und Nutzung von festen Austauschspeicherbereichen gekoppelt. Zwischenergebnisse und Variablen sind nur lokal sichtbar und statisch angelegt. So wird ermöglicht, dass jede Funktion ihren eigenen Kontext besitzt und diesen auch so wieder vorfindet, wie er bei Aussprung hinterlassen wurde. Dies löst das generelle Problem der Kontextwechselzeit, die sonst für das Sichern der Zwischenergebnisse benötigt wird.
Des Weiteren werden die Abläufe innerhalb der Struktur in Blöcke eingeteilt, welche sequenziell nacheinander abgearbeitet werden sollen. So entstehen automatisch Zustände fixen Kontextes innerhalb einer Funktion nach jedem Block. Welchen Umfang ein Block hat, bestimmt gleichzeitig die Granularität der möglichen Aussprünge aus der Funktion für die Abarbeitung einer höher priorisierten Aufgabe.
Eine besondere Behandlung benötigen Schleifenkonstrukte für diesen Ansatz; entweder es wird versucht statische Schleifen aufzurollen und nach jedem Schleifenkorpus einen Block zu setzen oder die Schleifenstruktur wird aus dem Block heraus gezogen und in die Übergangsfunktion des Automaten abgebildet [*Bild 2*]. Ein Block wird so zu einem Zustand.

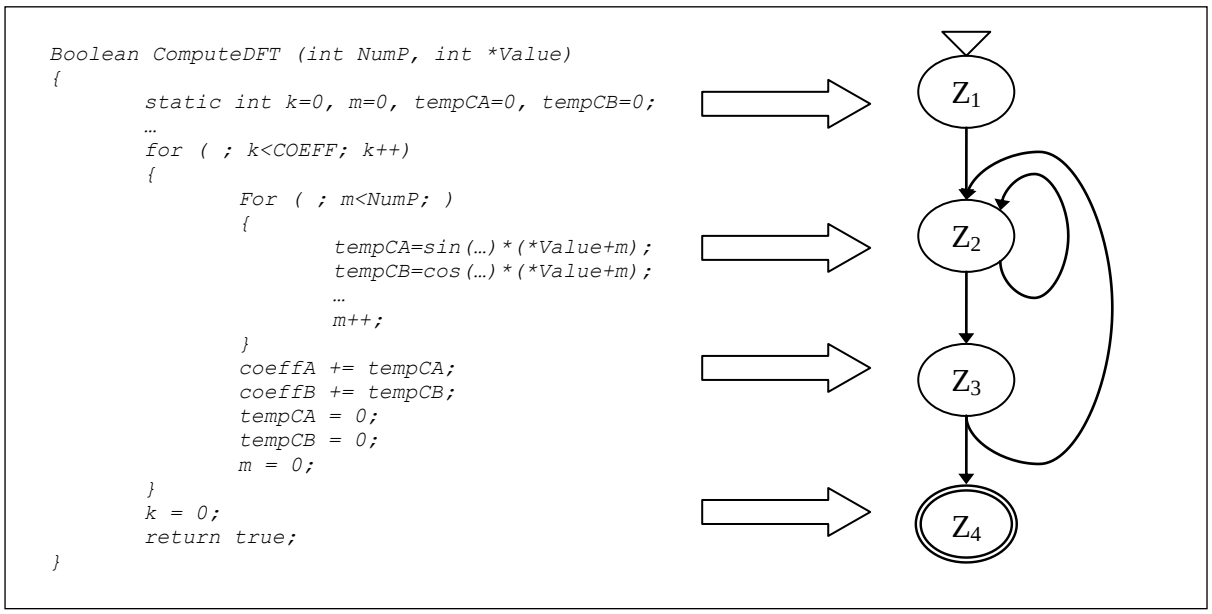

Bild 2: Transformation einer Funktion in einen Automaten

Nach jedem Zustand kann nun aus der Funktion gesprungen und ein Task höherer Priorität bearbeitet werden. Danach wird zur Funktion zurückgekehrt und im nächsten Zustand

fortgefahren. Je kleiner diese Blöcke sind, desto geringer wird die Blockierzeit einer kurzen, wichtigen Funktion durch einen Langläufer. Im Extremfall besteht ein Block aus einem Befehl und stellt in diesem Spezialfall ein Cycle-by-Cycle-Interleaving dar.

INTERRUPTSTRUKTUR:
Die zeitlichen Vorgaben werden mittels interruptgesteuerter Zustandsflags für jede einzelne Funktion umgesetzt. Durch die Serviceroutine eines Timer-Interrupts wird hierzu lediglich der Zustandszähler einer Funktion auf den Wert 1 gesetzt und somit beim nächsten Umlauf der Hauptschleife die entsprechende Funktion im ersten Zustand gestartet. Prioritäten bei der Interruptbehandlung ermöglichen eine direkte Selektion von wichtigen und Aufgaben geringerer Priorität – auch das kann bereits in den zusätzlichen Kommentaren für den Funktionsaufruf angegeben werden. Die ISR können so sehr kurz gehalten werden, was einer Blockierung entgegenwirkt. Lediglich I/O-Serviceroutinen müssen weiterhin direkt im Interrupt die Daten zumindest zwischenpuffern bevor dann wiederum durch ein Flag der Transfer signalisiert und durch entsprechende, getriggerte Funktionen reagiert werden kann.

HAUPTPROGAMM:
Das generierte Hauptprogramm besteht somit nur aus der Endlosschleife, welche periodisch jedes einzelne Flag abfragt und die entsprechenden Funktionen aufruft.
Durch die Verlagerung der Flussstruktur der Funktionen in die Hauptschleife des Codes können Aussprungbefehle nach jedem Block ersetzt werden. Das kontinuierliche Überprüfen von gesetzten Ereignisflags nach jedem Zustand und damit eine geringe Reaktionszeit werden so ermöglicht.
Der Zustandszähler einer Funktion wird zur Selektion des nächsten Zustands jeweils nach einem Zustand um eins erhöht. Im Anschluss zum letzten Zustand geht er auf den Wert 0 zurück was keinem Zustand innerhalb der Funktionsblöcke entspricht und symbolisiert somit, dass die Funktion erneut durch ein Ereignis (Interrupt) aktiviert werden muss [*Bild 3*].

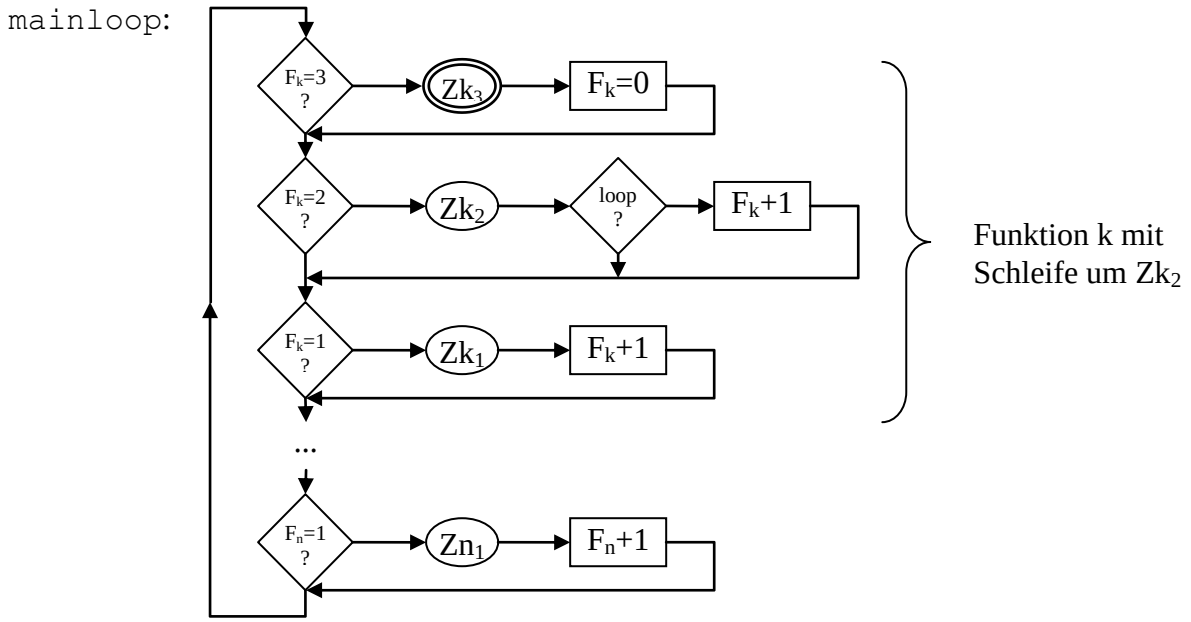

Bild 3: Struktur der Hauptschleife mit Funktionszuständen

Die Einbettung der einzelnen Zustände aus den Funktionen in die Hauptschleife benötigt eine detaillierte Betrachtung. Werden diese sequentiell durch Flagabfragen hintereinander angeordnet, so besteht die Gefahr, dass zwar eine Funktion nacheinander alle ihre Zustände abarbeitet, zwischendurch kurz in das Hauptprogramm wechselt, jedoch keine fremde Flags beachtet werden können. Aus diesem Grund müssen die einzelnen Zustandsabfolgen

rückwärts in die Schleife integriert werden, so dass nach jedem Zustand einer Funktion erst alle Flagabfragen anderer durchlaufen werden müssen bevor der Nachfolgezustand angesprungen wird.

Ein Vorteil dieser Schedulingstruktur liegt in der dynamischen Reaktionsmöglichkeit durch einfaches Anpassen der Zustandszähler. Im Gegensatz zu statischem Scheduling durch Codeinterleaving kann hier noch zur Ausführzeit auf Änderungen in den Daten oder den Timinganforderungen reagiert werden.
In dem beschriebenen Ansatz ist bereits jetzt ein deterministisches Verhalten im Bezug auf Blockierungen durch zu lange Berechnungen implementiert. Funktionen deren Algorithmen, unterteilt in mehrere Zustandsblöcke, mehr Zeit in Anspruch nehmen als ihnen durch eine periodische Zeitangabe zugesichert wurde, werden durch das Rücksetzen des Zustandszählers auf 1 automatisch erneut gestartet. Zwischenergebnisse können dabei allerdings jederzeit von abhängigen Tasks übernommen werden – so dass sich keine Blockierung aus diesen Abbrüchen ergibt.
Die Gefahr besteht nur, wenn die Periode einer Funktion definitiv für jede Eingabe nicht lang genug ist. Für diese Abschätzung ist dann eine Worst-Case-Execution-Time-Berechnung unumgänglich.

FAZIT:
Der vorgestellte Ansatz zur Implementierung von Zeitangaben in C-Code ermöglicht es zum einen Code zu generieren, dessen Struktur die geforderten Vorgaben erfüllen kann. Zum anderen lassen sich mithilfe dieser Informationen und einer WCET-Abschätzung Timingabweichungen bereits zur Compilezeit analysieren und erkennen.
Der Vorteil liegt hierbei in der Möglichkeit, dass diese Angaben optional in den Code einzupflegen sind und nicht jedes Statement mit einer Zeitvorgabe versehen werden muss.

LITERATUR:
[1] A. Leung, K. V. Palem, A. Pnueli. *TimeC: A time specification language for ILP processor compilation.* In K. A. Hawick and H. A. James, editors, Proceedings 12 of PART'98: The 5th Australasian Conference on Parallel And Real-Time Systems, pages 57-71. Springer-Verlag, 1998.
[2] M. van Bavel, M. Tilman, S. Vernalde, IMEC, CleanC analysis tools, Webpage: http://www.imec.be/CleanC, 2008.
[3] T. A. Henzinger, B. Horowitz, C. M. Kirsch, *Giotto: A time-triggered language for embedded programming.* In Proceedings of the First International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2211, pages 166-184, Springer-Verlag, 2001.
[4] Motor Industry Software Reliability Association (MISRA). *Guidelines for the Use of the C Language in Critical Systems*, ISBN 0-9524156-2-3 (paperback), October 2004.