Lara — Compilertechnologie für harte Echtzeit

Carsten Kuckuk*

1. Einleitung und Überblick

In der Echtzeitdatenverarbeitung werden andere Forderungen an Programmiersprachen und die zugehörigen Übersetzer gestellt, als in der herkömmlichen Datenverarbeitung [1]. Dazu gehören die Forderungen nach einer garantierbaren Ausführungszeit und nach einem begrenzten, im Voraus bekannten Speicherbedarf des übersetzten Programmes.

Mit Lara (LAnguage for Real-time Applications) wird nun eine Sprache vorgestellt, für die der Übersetzer in der Lage ist, diese beiden Forderungen zu erfüllen.

Zuerst wird informell die Sprache Lara zusammen mit der Struktur des übersetzten Codes vorgestellt. Daran anschließend wird gezeigt, wie man mit Hilfe von attributierten Grammatiken (ATGs) in der Lage ist, einen Übersetzer für Lara-Programme in die Maschinensprache des Digitalen Signalprozessors TMS320C25 von Texas Instruments [2] zu erstellen.

2. Lara

Lara ist eine Pascal-ähnliche, imperative, maschinennahe Programmiersprache ohne Multitasking mit Sprachmitteln für "harte" Echtzeit.

Ein Lara-Programm besteht aus einer Ansammlung von Konstanten-, Typen-, Variablen-, Prozedur- und Funktionsvereinbarungen und einem Konfigurationsteil, der zu jeder möglichen Unterbrechung (einschließlich Reset) eine abzuarbeitende Prozedur angibt. In diesem Konfigurationsteil können zeitliche Einschränkungen für diese Prozeduren gemacht werden. Zur Laufzeit wird beim Auftreten einer Unterbrechung die in der Konfiguration angegebene Prozedur abgearbeitet, in einen Wartezustand übergegangen und auf eine erneute Unterbrechung gewartet. Eine einmal gestartete Prozedurausführung ist nicht unterbrechbar (keine Vorrangunterbrechung).

2.1 "Harte" Echtzeit in Lara

In Lara wird Zeit in Maschinenzyklen des benutzten Prozessors gemessen. Beim TMS320C25 besteht ein Maschinenzyklus aus vier Taktzyklen. Der Lara-Übersetzer kennt intern zwei Arten der Zeitangabe:

1. GenaueZeit: Eine Zeitangabe besteht aus genau einer Zahl.
2. IntervallZeit: Eine Zeitangabe besteht aus einer Ober- und einer Untergrenze. Das durch eine solche Angabe beschriebene Konstrukt benötigt eine Ausführungszeit, die zwischen diesen beiden Werten oder genau einer dieser beiden Werte ist. Bei der Berechnung von Intervallzeiten kann, etwa bei Schleifen, auch $+\infty$ entstehen.

Der Lara-Programmierer kommt mit diesen beiden Zeitarten nicht direkt in Berührung, sondern gibt zeitliche Einschränkungen in Form von `Every`- oder `in`-Klauseln an. Der Übersetzer bestimmt dann im Laufe der Übersetzung für jedes übersetzte Konstrukt eine Ausführungszeit. Dann wird geprüft, ob die so ermittelte Ausführungszeit mit der im Programm angegebenen Zeitforderung in Einklang zu bringen ist. Ist diese Prüfung erfolgreich, so kann der Übersetzer — nach eventuellem Einfügen von `NOPs` — mit der Übersetzung fortfahren. Im erfolglosen Fall wird eine Fehlermeldung ausgegeben und der Übersetzer bricht ab.

Der Zeitmodus, in dem ein Befehl übersetzt wird, wird durch die syntaktisch übergeordnete Einheit, in der er sich befindet, bestimmt. Auf der Prozedur-Ebene wird durch das Schlüsselwort `timed` festgelegt, daß eine Prozedur im GenaueZeitModus übersetzt werden soll. Ohne dieses Schlüsselwort erfolgt die Übersetzung im IntervallZeitModus. Durch besondere Befehle kann der Wechsel vom IntervallZeitModus in den GenaueZeitModus erzwungen werden. So erzwingt eine `every`-Klausel in einer Schleife, daß das Innere der Schleife im GenaueZeitModus übersetzt werden muß. Konstrukte, die im GenaueZeitModus übersetzt sind,

* *Ruhr-Universität Bochum, Fakultät für Mathematik, Lehrstuhl Praktische Informatik, Postfach 10 21 48, D-4630 Bochum.*

können als Teile von Konstrukten benutzt werden, die im IntervallZeitModus übersetzt werden; umgekehrt ist dies nicht möglich, weil in diesem Falle ein übersetztes Programmstück, von dem nur eine IntervallZeit als Ausführungszeit angegeben werden kann, so in ein größeres Programmstück integriert werden müßte, daß dieses neu entstehende Programmstück eine GenaueZeit als Ausführungszeit hat.

2.2 Laufzeitmodell

Speziell für den Lara-Übersetzer für den TMS320C25 werden folgende Vereinbarungen getroffen: Das übersetzte Programm verwaltet einen Stapel, auf dem lokale Variablen, Rücksprungadressen, Argumente bei Prozeduraufrufen und Zwischenergebnisse zwischengespeichert werden können. Ergebnisse eines arithmetischen Ausdruckes werden im Akku gehalten, arithmetische Operationen werden immer auf den Akku und wenn nötig auf das oberste Element des Stapels angewandt.

Das AR1-Register dient als Stapelzeiger und zeigt auf das nächste freie Element des Stapels. Der Stapel wächst mit steigenden Adressen. Das AR2-Register zeigt auf den Beginn des Speicherbereichs der lokalen Variablen, und das AR3-Register zeigt auf den Beginn des Speicherbereiches der Argumente. Das DP-Register ist mit 4 geladen, so daß die Möglichkeit besteht, den internen RAM des DSPs als schnellen Zwischenspeicher zu benutzen. (Die direkte Adressierung erfolgt beim TMS320C25 segmentiert und der Offset-Wert ist Teil des auszuführenden Befehlswortes. Für weitergehende Informationen siehe [2].) Die ersten 6 Worte dieses RAMs sind für spezielle Aufgaben fest eingeplant.

2.3 Sprachmittel

1. **Typen**: In Lara hat jeder Ausdruck und Teilausdruck genau einen Typ, der sich aus den Typen der Teilausdrücke und der darauf angewandten Operation ergibt. Eine automatische Typanpassung findet nicht statt. In Lara gibt es die Typen `Boolean` und `Bitset` (entspricht dem Pascal-Typ `Set of 0..15`). Festkommatypen können in der Form `"Fixed" "(" "Sign" "=" ("True" | "False")";" "Mantissa""="` Expression `"Bits"";" "Fraction""="` Expression `"Bits" ")"` vereinbart werden. Vordefinierte Festkommatypen sind `Integer`, `Fractional` und `Cardinal`. Mit `Array` können Felder gebildet werden.

2. **Variablen**: In Lara gibt es, wie in C, zwei Arten von Variablen: statische und dynamische. Der Platz für dynamische Variablen wird auf dem Stapel angelegt. Dynamische Variablen verlieren ihren Wert zwischen zwei Aufrufen einer Prozedur. Statische Variablen haben eine feste Adresse und verlieren ihre Inhalte zwischen zwei Aufrufen einer Prozedur nicht. Statische Variablen sind im Adressraum des Prozessors beliebig plazierbar. Unter Verwendung des Typen `Bitset` kann man so Hardwarebausteine direkt mit Lara manipulieren.

3. **Konstanten**: In Lara können skalare Konstanten und konstante Felder vereinbart werden. Skalare Konstanten können zur Übersetzungszeit vom Übersetzer in Rechnungen ausgewertet werden. Arrays werden nur wie initialisierte statische Variablen behandelt, die nicht verändert werden können.

4. **Zuweisung**: Statement ⟶Ident [`"["` Expression `"]"`] `":" "="` Expression

5. **Bedingung**: Wenn das `If`-Statement im IntervallZeitModus übersetzt wird, dann werden auch der `Then`- und der `Else`-Fall im IntervallZeitModus übersetzt und die sich ergebende Ausführungszeit ist eine IntervallZeit. Wird das `If`-Statement im GenaueZeitModus übersetzt, so werden auch der `Then`- und der `Else`-Fall im GenaueZeitModus übersetzt. Wenn dabei zwei verschiedene Zeiten entstehen, so wird der kürzere Fall soweit mit `NOP`s aufgefüllt, bis die Ausführungszeiten für beide möglichen Fälle gleich sind.
   Syntax:
   Statement ⟶`"If"` Expression `"Then"` StatementSequence [`"Else"` StatementSequence] `"End" "If"`

6. **Prozeduraufruf**: Syntax: Statement ⟶Ident [`"("` Expression { `","` Expression } `")"`]

7. **Schleifen**: In Lara gibt es die `While`-, `Repeat`-, `Loop`- und die `For`-Schleife. Bis auf einen Sonderfall der `For`-Schleife können alle Schleifen nur im IntervallZeitModus übersetzt werden, weil die Anzahl der Schleifendurchläufe nicht im Voraus bekannt ist. Wenn die Schleife eine `Every`-Klausel besitzt, so wird das Innere der Schleife in GenaueZeitModus übersetzt, und zwischen zwei Schleifendurchläufen vergeht genau die angegebene Zeit. Wenn bei einer `For`-Schleife eine Konstante als Unter- und Obergrenze angegeben ist, dann kann diese Schleife auch im GenaueZeitModus übersetzt werden. Syntax:

Statement ⟶ "Loop" ["Every" Expression "Cycles"]
StatementSequence "End" "Loop"

⟶ "While" ["Every" Expression "Cycles"] Expression "do"
StatementSequence "End" "While"

⟶ "Repeat" ["Every" Expression "Cycles"]
StatementSequence "Until" Expression

⟶ "For" ["Every" Expression "Cycles"]
Ident ":" "=" Expression ["to" | "downto"] Expression "do"
StatementSequence "End" "For"

8. **In**: Mit `In` kann für eine Befehlsfolge eine GenaueZeit als Ausführungszeit gefordert werden. Dabei werden, falls nötig, `NOPs` angefügt. Syntax:

   Statement ⟶"In" Expression "Cycles" "do" StatementSequence "End" "In"

9. **Prozedur- und Funktionsvereinbarungen**: Prozeduren und Funktionen dürfen nur textuell vorangehende Prozeduren und Funktionen aufrufen. Parameter werden mit einem der Modi `in`, `out` oder `inout`, ähnlich wie in ADA, vereinbart. Prozeduren und Funktionen dürfen sich nicht rekursiv selbst aufrufen und dürfen nicht geschachtelt werden. Diese Einschränkungen machen es erst möglich, für jeden Prozedur- oder Funktionsaufruf einen maximalen Platzbedarf anzugeben und so sicherzustellen, daß das übersetzte Programm nicht zur Laufzeit aus Speicherplatzmangel in einen undefinierten Zustand übergeht.

2.4 Beispielprogramm

Um einen kleinen Eindruck von einem Lara-Programm zu bekommen, wird unten ein FIR-Filter abgedruckt [3].

```
Module FIR;

-- Es implementiert ein Lowpass-Filter in FIR-Technik mit einer
-- Sampling-Frequenz von 1Khz und einer Cutoff-Frequenz von 100Hz.
-- Es wurde ein Prozessor-Takt von 40 MHz zugrunde gelegt. Nach:
-- Roberts/Mullis 1987 : Digital Signal Processing, pp. 178-79.

Type fractional      = Fixed(Sign = True;
                             Mantissa = 15 Bits;
                             Fraction = 15 Bits);

     integer         = Fixed(Sign = True;
                             Mantissa = 15 Bits;
                             Fraction = 0 Bits);

     fractional_feld = Array[0 .. 12] of fractional;

Const h = fractional_feld'(-0.0311829, 0.0, 0.0467744, 0.1009102,
                           0.1513653, 0.1870978, 0.2,
                           0.1870978, 0.1513653, 0.1009102,
                           0.0467744, 0.0, -0.0311829);
--  Filterkoeffizienten h(k)=(1/5)sinc(k*pi/5), k=-6..6, sinc(x)=sin(x)/x.

Var e   : static fractional_feld;       -- 13 Eingangssamples
    adc : static fractional at [io,5]; -- AD-Wandler
    dac : static fractional at [io,6]; -- DA-Wandler
```

```
Procedure init;  -- Samples mit 0.0 vorbelegen
Var i : integer;
Begin init;
  For i:=0 to 12 do
      e[i]:=0.0
  End For
End init;

Procedure main;
Var i   : integer;
Begin main;
  init;                          -- Samples mit 0.0 vorbelegen

  Loop every 10000 Cycles        -- alle 1000 microSec.
       e[0]:=adc;                -- Neuen Wert samplen
       dac:=Convert((e|h),Fractional);  -- Skalarprodukt bilden
       For i:=12 downto 1 do
           e[i]:=e[i-1]          -- Delay-Operation ausführen
       End For
  End Loop                       -- und wieder von vorne
End main;

Configuration FIR;
  Reset = main;--Bei einem Reset wird main abgearbeitet
End Configuration FIR.
```

3. Attributierte Lara-Grammatik

Zur Beschreibung der Syntax von Programmiersprachen werden im Allgemeinen kontextfreie Grammatiken (kfG) verwendet [4]. Für die manuelle Implementation eines Übersetzers besonders geeignet sind LL(1)-Grammatiken. Zu jedem herleitbaren Wort läßt sich ein Ableitungsbaum angeben, dessen innere Knoten den nichtterminalen und dessen Blätter den terminalen Symbolen bei der Herleitung des Wortes entsprechen.

Jedem Knoten aus dem Ableitungsbaum läßt sich eine Menge von Attributen zuordnen, die Eigenschaften dieses Knotens darstellen. Eine attributierte Grammatik (ATG) [5] ist eine erweiterte kfG, die angibt, zu welchen Symbolen der kfG welche Attributmengen gehören und weiterhin Beziehungen zwischen den einzelnen Attributen innerhalb der einzelnen Produktionen festlegt. Mit Hilfe dieser Regeln ist man dann in der Lage, die Werte der Attribute in einem Ableitungsbaum zu bestimmen. In Abhängigkeit vom Aufbau der Regeln muß man den Ableitungsbaum unter Umständen mehrfach durchlaufen, um alle Attribute zu berechnen. Bei einer L-attributierten Grammatik (LATG) genügt ein inorder-Durchlauf des Ableitungsbaumes.

Beispiel:

$$\boxed{\text{Summe}}\uparrow Wert_1$$
$$\longrightarrow \boxed{\text{Zahl}}\uparrow Wert_2\ \boxed{\text{"+"}}\ \boxed{\text{Zahl}}\uparrow Wert_3\ \{Wert_1 := Wert_2 + Wert_3\}$$
$$\boxed{\text{Zahl}}\uparrow Wert$$
$$\longrightarrow \boxed{\text{"1"}}\{Wert := 1\}$$
$$\longrightarrow \boxed{\text{"2"}}\{Wert := 2\}$$

Zu 1 + 2 gehört dann der (attributierte) Ableitungsbaum

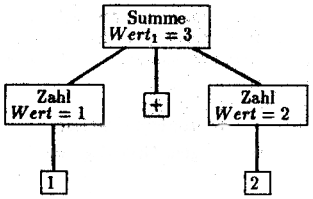

Abbildung 1: Attributierter Ableitungsbaum zu 1 + 2

Die für Lara aufgestellte ATG ist eine LATG mit LL(1)-Eigenschaft. In dieser LATG sind die wichtigsten Attribute:

- *Umgebung:* Setzt sich aus Listen zusammen, die Informationen über definierte Typen, Konstanten, Variablen und Prozeduren enthält
- *Modus:* Gibt an, in welchem Zeitmodus übersetzt werden soll
- *Code:* Das Ergebnis des Übersetzungsvorganges
- *Zeit:* Die benötigte Zeit für die Ausführung eines *Codes*
- *Stapelgröße:* Anzahl der von einem *Code* auf dem Stapel benötigten Worte.

Die Regel für die `If`-Anweisung aus der Lara-ATG verdeutlicht die Verwendung dieser Attribute:

Statement$\downarrow Umgebung_1 \downarrow Tiefe_1 \downarrow Modus_1 \uparrow Umgebung_4 \uparrow Code_1 \uparrow Stapelgröße_1 \uparrow Zeit_1$

$\longrightarrow$ "If" Expression$\downarrow Umgebung_1 \downarrow Tiefe_1 \downarrow Modus_1 \uparrow Umgebung_2 \uparrow Ergebnis_1$
$\uparrow Stapelgröße_2 \uparrow Zeit_2 \uparrow TypeID_1$
$\{Assert(Typgleich(TypeID_1, Boolean\_ID))\}$ "Then"
StatementSequence$\downarrow Umgebung_2 \downarrow Tiefe_1 \downarrow Modus_1 \uparrow Umgebung_3 \uparrow Code_3 \uparrow Stapelgröße_3 \uparrow Zeit_3$
ElseFall$\downarrow Umgebung_3 \downarrow Tiefe_1 \downarrow Modus_1 \uparrow Umgebung_4 \uparrow Code_4 \uparrow Stapelgröße_4 \uparrow Zeit_4$
"End" "If"
$\{(ErgStapel, ErgCode, ErgZeit) := MachCode(Ergebnis_1, Stapelgröße_2, Zeit_2, Modus_1)$
$ElseLabel := NewIdent; ExitLabel := NewIdent$
Wenn $Modus_1$ =GenaueZeit: $MaxZeit := Max(Zeit_3, Zeit_4)$
$ThenCode := Code_3@WaitCode(MaxZeit - Zeit_3)$
$ElseCode := Code_4@WaitCode(MaxZeit - Zeit_4)$
$Code_1 := ErgCode@[\texttt{BZ}ElseLabel]@ThenCode@[\texttt{B}ExitLabel]$
$@[Elselabel :]@ElseCode@WaitCode(4p)@[ExitLabel :]$
$Zeit_1 := ErgZeit \oplus (5 + 5p) \oplus MaxZeit$
Wenn $Modus_1$ =IntervallZeit: $Code_1 := ErgCode@[\texttt{BZ}ElseLabel]@Code_3@[\texttt{B}ExitLabel]$
$@[Elselabel :]@Code_4@[ExitLabel :]$
$ThenZeit := Zeit_3 \oplus (5 + 5p, 5 + 5p)$
$ElseZeit := Zeit_4 \oplus (3 + 3p, 3 + 3p)$
$Zeit_1 := ErgZeit \oplus MinMaxZeit(ThenZeit, ElseZeit)$
$Stapelgröße_1 := Max(ErgStapelgröße, Stapelgröße_3, Stapelgröße_4)$
$\}$

ElseFall$\downarrow Umgebung_1 \downarrow Tiefe_1 \downarrow Modus_1 \uparrow Umgebung_2 \uparrow Code_1 \uparrow Stapelgröße_1 \uparrow Zeit_1$

$\longrightarrow$ "Else" StatementSequence$\downarrow Umgebung_1 \downarrow Tiefe_1 \downarrow Modus_1 \uparrow Umgebung_2 \uparrow Code_1$
$\uparrow Stapelgröße_1 \uparrow Zeit_1$

$\longrightarrow$ $\varepsilon\{Umgebung_2 := Umgebung_1; Code_1 := nil; Stapelgröße_1 := 0; Zeit_1 := 0_{Modus_1}\}$

Hier bewirkt der Operator @ die Verkettung der Codelisten und der Operator $\oplus$ die Addition von Zeiten. $\uparrow$ und $\downarrow$ geben an, ob es sich um ein berechnetes oder ererbtes Attribut handelt.

Voraussetzung für diese Art der Übersetzung und insbesondere der Zeitermittlung ist eine genaue Dokumentation des benutzten Prozessors. Beim Entwurf der Hardware muß darauf geachtet werden, daß das Zeitverhalten aller beteiligten Komponenten vorhersagbar ist. Unter Anderem bedingt dies einen Verzicht auf DMA und Caches.

4. Implementation

Ein Übersetzersystem auf der oben gezeigten Grundlage wurde auf einem IBM-PC-AT-kompatiblen Rechner unter MS-DOS in Turbo-Pascal 5.5 in Zusammenarbeit mit der Aston GmbH, Oberhausen, realisiert. Zielsystem ist eine DSP-Karte mit dem Prozessor TMS320C25.

Das Entwicklungssystem besteht aus einem Übersetzer, Makro-Assembler, Linker und Debugger.

Die Übersetzer-Implementation erfolgte wie folgt: Zuerst wurde zu der LL(1)-Grammatik ein rekursiv absteigender Parser entworfen. Die einzelnen Prozeduren wurden dann entsprechend den Attributen der Symbole in der ATG mit Parametern versehen und die einzelnen Prozeduren entsprechend der ATG um Befehle zur Berechnung der Attribute erweitert. In dieses Design wurde dann noch die Ausgabe von Informationen für den Debugger und die Ausgabe aller anfallenden Zeit- und Stapelinformationen in ein Log-File eingearbeitet. Dies dient dazu, daß der Anwender im Detail sehen kann, wie effizient seine einzelnen Lara-Befehle übersetzt worden sind und wo er eventuell sein Programm überarbeiten soll.

5. Zusammenfassung und Ausblick

Die oben skizzierte Technik erlaubt es, die Ausführungszeit von übersetzten Programmen auf der Grundlage der vom Hersteller des Prozessors gelieferten Dokumentation zu ermitteln und zu garantieren. Dabei gibt es ähnliche Probleme, wie bei den Korrektheitsbeweisen von Programmen.

Dieser Ansatz ist für eine breite Klasse von Prozessoren geeignet. Viele Microcontroller und digitale Signalprozessoren haben feste, garantierte Ausführungszeiten und sind einer Berechnung der Ausführungszeiten in der oben gezeigten Art zugänglich.

Die exakte Berechnung der Ausführungszeiten von Programmfragmenten bildet die Grundlage für den Einsatz spezieller Task-Einplanungsalgorithmen [6], die in der Lage sind, die rechtzeitige Ausführung von akzeptierten Task-Einplanungen zu garantieren. Somit ist der oben beschriebene Ansatz also nicht nur für single-Task Aufgaben brauchbar, sondern kann in Zukunft dazu beitragen, Multitasking für den "harten" Echtzeitbereich nutzbar zu machen.

[1] W. A. Halang: Real-Time Programming Languages. In: Michael Schiebe, Saskia Pferrer (hrsg.): Echtzeitverarbeitung, Theoretische Grundlagen und Anwendungen. 10.-12. Oktober 1989. 10. Tagungsband. Arbeitskreis Medizinische Informatik, Universität Ulm, Ulm, 1990, pp.111-142.

[2] Second Generation TMS320 User's Guide, Texas Instruments, 1988.

[3] Richard A. Roberts, Clifford T. Mullis: Digital Signal Processing. Addison-Wesley, Reading, Massachusetts, USA, reprinted with corrections, May, 1987.

[4] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: Compilers, Principles, Techniques and Tools. Addison-Wesley, Addison Wesley, Reading, Massachusetts, USA. Reprinted with corrections 1987.

[5] Donald E. Knuth : Semantics of Context-Free Languages. Mathematical Systems Theory, Vol.2, No.2, 1968, pp.127-145, Springer-Verlag. (Korrektur in Math. Syst. Theory, Vol.5, No.1, pp.95-96.)

[6] Shem-Tov Levi, Satish K.Tripathi, Scott D. Carson, Ashok K. Agrawala: The MARUTI Hard Real-Time Operating System. Operating Systems Review, Vol. 23, Nr. 3, July 1989, New York, pp.90-105.

Echtzeit'91

Kongreß & Ausstellung

für angewandte Echtzeit-Datenverarbeitung
in Automation, Meßtechnik, Simulation
und anderen technischen Bereichen

Kongreß-Vorträge
Conference Proceedings

11. bis 13. Juni 1991 - Messehalle in Sindelfingen bei Stuttgart