# Eine rechnerunabhängige Übersetzungstechnik für die Realzeitsprache PEARL, angewandt für eine SIEMENS 310

P. Holleczek

Physikalisches Institut der
Universität Erlangen-Nürnberg

31.7.1980

## 1. Einleitung

Das Auseinanderklaffen von Hardware- und Software-Preisen macht die kostengünstige Programmierung immer neuer Generationen von Klein- und Kleinstrechnern zu einem Problem. Das Problem wird verschärft durch eine immer schnellere Generationsfolge der Rechner, die die Entwicklung von Systemprogrammen kaum schritthalten läßt. Ein Weg, diesem Problem zu begegnen, ist der Einsatz von portablen Systemsoftware-Komponenten.

Während hiervon in konventionellen Kleinrechensystemen bereits Gebrauch gemacht wird [1], soll hier von einem Projekt, gefördert über BMFT/PDV, berichtet werden, bei dem dieser Weg auch bei Realzeitanwendungen mit Erfolg beschritten wurde.

Es ist Absicht dieses Beitrags, nicht so sehr auf die theoretischen Grundlagen einzugehen, sondern über Erfahrungen aus dem Einsatz der portablen Systemsoftwarekomponenten zu berichten. Bezüglich einer detaillierteren Darstellung portabler Techniken sei auf die Beiträge von Lindstedt, Pelz, Prester und Rössler in [2] verwiesen.

Das Projekt war in verschiedene Stufen unterteilt. Ausgehend von der Realzeitprogrammiersprache PEARL wurden in der ersten Phase, einer Pilotimplementation, schnellstmöglich Compiliersysteme erstellt, mit deren Hilfe neuartige Sprachelemente erprobt werden konnten. Die zweite Phase des Projektes bestand darin, durch Entwicklung portabler Systemsoftware den Implementationsaufwand für ein Compiliersystem zu senken. Die erste Projektphase wurde durch ein PEARL-Compiliersystem u.a. für den Prozeßrechner S 306 abgeschlossen [3]; in der zweiten Projektphase wurde durch ein PEARL-Compiliersystem für den Prozeßrechner S 310 eine Zwischenbilanz gezogen [4]. Beide Compiliersysteme stehen im Einsatz [5,6].

Festgehalten werden muß, daß die erste Projektphase, die Pilotimplementation, vom Phys. Inst. der Univ. Erlangen-Nürnberg im Rahmen der aus Firmen und Instituten bestehenden Arbeitsgemeinschaft ASME [7] verfolgt wurde. Die zweite Projektphase, die Entwicklung portabler System-Software-Komponenten und die 310-Implementation, wurde am Phys. Inst. selbst abgewickelt.

## 2. Konzept und Grundlagen

Die Problematik bei der Entwicklung portabler Systemsoftware-Komponenten und das verfolgte Konzept läßt sich am besten erklären, wenn zunächst kurz auf die Pilotimplantation im Rahmen der ASME [7] eingegangen wird.

### 2.1 Grobkonzept

Das technische Konzept [8] der Pilotimplementation (schnellstmögliche) Implementation einer Reihe von Compiliersystemen) sah vor, den Übersetzungsablauf so zu strukturieren, daß leicht verschiedene Zielmaschinen bedient werden konnten.

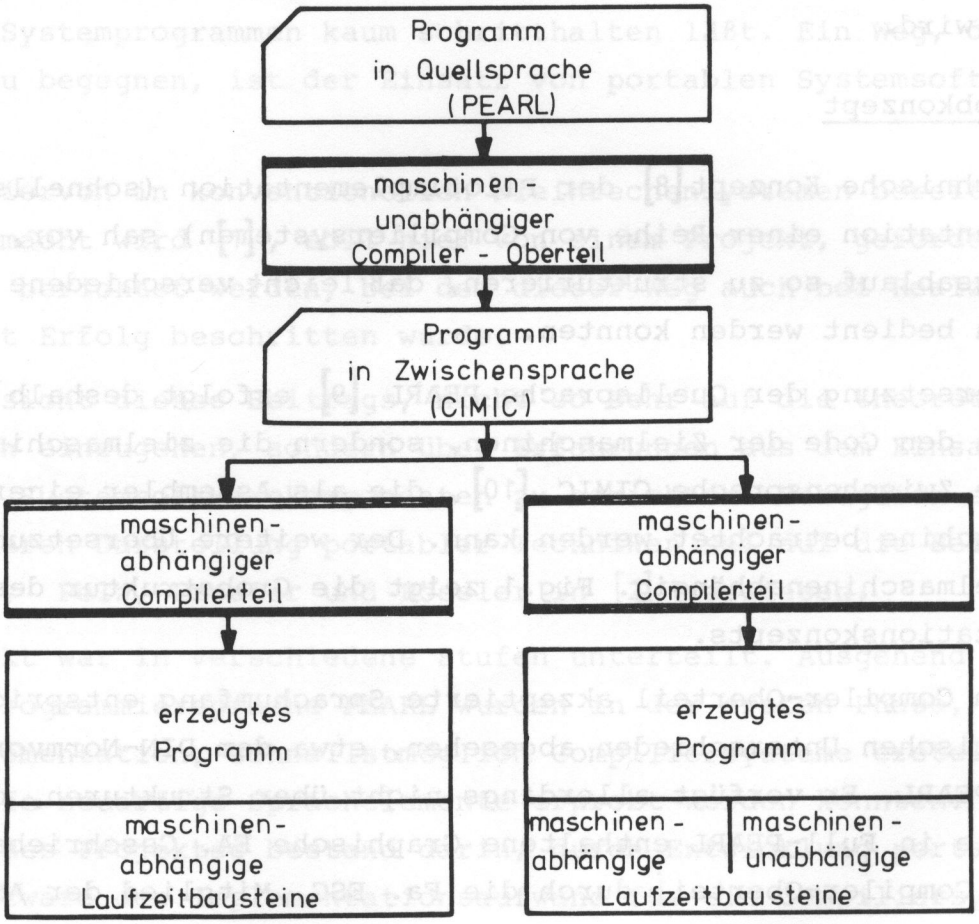
Die Übersetzung der Quellsprache PEARL [9] erfolgt deshalb nicht direkt in den Code der Zielmaschinen, sondern die zielmaschinenunabhängige Zwischensprache CIMIC [10], die als Assembler einer abstrakten Maschine betrachtet werden kann. Der weitere Übersetzungsvorgang ist zielmaschinenabhängig. Fig.1 zeigt die Grobstruktur des ASME-Implementationskonzepts.

Der vom Compiler-Oberteil akzeptierte Sprachumfang entspricht, von syntaktischen Unterschieden abgesehen, etwa dem DIN-Normvorschlag [11] BASIC-PEARL. Er verfügt allerdings nicht über Strukturen, dafür aber über die in Full-PEARL enthaltene Graphische EA. Geschrieben wurde dieser Compiler-Oberteil durch die Fa. ESG, Mitglied der ASME [7], in Fortran. Dies ist zwar keine Systemimplementierungssprache, dafür aber auf den meisten Maschinen vorhanden. Dank seines modularen Aufbaus beansprucht der Compiler-Oberteil nach geeignetem (Overlay-) Binden einen residenten Arbeitsspeicherbereich von ca. 20 KW.

Von CIMIC aus wurden die Zielmaschinen mit Code versorgt. Um den erzeugten Code ablaufen lassen zu können, mußte für PEARL-geeignete Laufzeitbausteine wie Laufzeitfunktionen, Betriebssystem und Gerätetreiber gesorgt werden.

Die Laufzeitfunktionen gliedern sich in EA-Funktionen und algorithmische Funktionen. Die algorithmischen Laufzeitfunktionen dienen zur Behandlung von Operationen auf nicht-primitive Datentypen, die vom Codegenerator nicht aufgelöst werden können. Die EA-Laufzeitfunktionen stellen das Bindeglied zwischen den vom Codegenerator meist aus dem Quellprogramm nur weitergereichten EA-Aufrufen und dem Leistungsangebot des Betriebssystems (Datentransfer zwischen einem Gerät und einem

Programm-Speicherbereich) dar. Sie umfassen Funktionen für die Standard-EA, die graphische EA, die Dateiverwaltung und die Prozess-EA.




SIEMENS 306

(Pilotimplementation
wie AEG 6050, 6010
und SIEMENS 404)

SIEMENS 310

Fig.1 Stark vereinfachte Darstellung des Implementationskonzepts.

## 2.2 Die 306-Implementation

Bei der 306 handelt es sich um eine 24 bit Akku-Maschine mit Standardperipherie, graphischem Sichtgerät und CAMAC-Prozeßperipherie. Ihr Speicherausbau beträgt 32 k Worte, von denen ca. 10 kW durch den residenten Teil des Standard-Betriebssystems belegt werden. Um die Implementation so schnell wie möglich vorantreiben zu können, wurde möglichst viel von der Hersteller-Betriebssoftware übernommen. Alle weiteren Komponenten wurden zielmaschinenspezifisch erstellt.

Der Codegenerator wurde mit dem Makrogenerator Stage II erstellt. Erzeugt wurde Assembler-Code, der vom Hersteller-Assembler weiterverarbeitet wurde. Der Binder wurde dem FORTRAN-Compiliersystem entnommen.

Auch bei den Laufzeitbausteinen wurde Rücksicht auf vorhandene Komponenten genommen. Da die Leistungen des Standard-Betriebssystems für PEARL nicht ausreichten, wurde es um eine Schicht für die fehlenden Tasking- und Timing-Funktionen erweitert [12]. Für die CAMAC-Prozeßperipherie [13] und das graphische Sichtgerät GSG1 [14] gab es keine Software-Unterstützung, so daß entsprechende Treiber erstellt und in das Betriebssystem integriert werden mußten.
Ein Teil der Laufzeitfunktionen für die Standard-E/A [15] (Formatweiterschaltung, Formatierung der herkömmlichen Datentypen) konnte aus dem FORTRAN-Laufzeitpaket übernommen werden. Neuerstellt wurden die Funktionen für die realzeitspezifischen Datentypen (Formatierung von Zeiten, Bit-, Zeichenketten) sowie für die graphische EA. Ebenfalls neu implementiert wurden die algorithmischen Laufzeitfunktionen und die Dateiverwaltungsfunktionen [16]. Die Implementation wurde abgerundet durch ein Test- und Bediensystem, das dem Operateur die Möglichkeit gibt, Größen abzufragen und in den Ablauf einzugreifen.

Trotz der Verwendung mancher Komponenten der Hersteller-Software belief sich die Implementationszeit auf ca. 2,5 Jahre. Der für die einzelnen Komponenten nötige Aufwand ist aus Tabelle 1 ersichtlich. Nicht enthalten sind dabei Inbetriebnahme, Arbeiten an der Hardware etc.

| | MJ |
|---|---|
| Codegenerator | 1 1/2 |
| Laufzeitfunktionen | |
| Standard EA | 1 1/2 |
| Filehandling | 1 |
| Graphische EA | 1 |
| Prozeß-EA | 1 1/2 |
| Algorithmik | 1/2 |
| Betriebssystem (-Anpassung) | 1 1/2 |
| Test- und Bediensystem | 1/2 |
| | 9 MJ |

Tab. 1 S 306-Implementationsaufwand (Spezialanfertigungen)

Diesem Aufwand steht ein einmaliger Erstellungsaufwand von ca. 10 Mannjahren für den maschinenunabhängigen Compiler-Oberteil gegenüber.

## 2.3 Entwicklung portabler Komponenten

Aufgrund des im Vergleich zum portablen Compileroberteil hohen Implementationsaufwands für die maschinenspezifischen Komponenten (wie Codegenerator und Laufzeitbausteine) schien es dringend geraten, auch hierfür portable Lösungen zu entwickeln. Das läßt sich prinzipiell auf verschiedenen Wegen erreichen (eine Übersicht über verschiedene Methoden ist in [17] enthalten). Stellt man fest, daß in den üblichen Systemprogrammen immer wiederkehrende Aufgaben zu lösen sind, kann man die hierzu nötigen Algorithmen repräsentativ, z.B. in einer höheren Programmiersprache, formulieren. An Stellen, wo auf feste Randbedingungen Rücksicht zu nehmen ist (z.B. auf Eigenschaften der Maschine) und keine allgemein gültige Lösung angegeben werden kann, muß eine Schnittstelle eingeführt werden. Dieser Weg wurde bei den E-/A-Funktionen und bei dem Betriebssystem gewählt. Wenn z.B. die zu erreichenden Maschinen zu verschieden sind, lassen sich oft keine repräsentativen, sondern allenfalls "ähnliche" Algorithmen isolieren. Hier kann man versuchen, diese Unterschiede so zu konzentrieren, daß abhängig von Maschinen-Parametern bestimmte Programmteile nur gegen andere auszutauschen sind. Ein solcher Austausch kann dann über einen "Generator" erfolgen. Dieser Weg wurde bei der graphischen E/A, der

Algorithmik und dem Codegenerator gewählt.

Als Beispiel für die Entwicklungen der portablen Komponenten soll hier auf die Standard-E/A-Laufzeitfunktionen eingegangen werden [18], die von gewissem Einfluß auf die Entwicklung der anderen Komponenten war. Diese Funktionen dienen dazu, die in der Datenliste einer EA-Anweisung vorkommenden Größen entsprechend den Elementen der Formatliste zwischen interner und externer Darstellung zu wandeln. Es bot sich an, dabei zwischen einer Formatweiterschaltung (Verwalten der Zeiger auf Daten- und Formatliste) und der Formatierung (Wandeln eines Datenelements entsprechend eines Formats) zu unterscheiden. Da die Formatweiterschaltung, wie sie in CIMIC verwendet wird, sicher nicht allgemein gültig ist, wurde nur die Umformatierung, die eine immer wiederkehrende Aufgabe darstellt, portabel formuliert. Es wurden also die Formatierungsfunktionen für alle PEARL-Datentypen erstellt.

Eine geeignete Untermenge von PEARL fällt aber bei einer Implementation ohnehin an. Es muß also kein besonderes Erstellungs-"Werkzeug" installiert werden.

Speziell zu programmieren waren die Funktionen zur Format-Weiterschaltung ("obere" Schnittstelle zum Anwenderprogramm) und die Aufrufe der Geräte ("untere" Schnittstelle zum Betriebssystem).
Vorteil dieses Vorgehens war, daß die Programm- und Datenstrukturen der übersetzten Anwenderprogramme mit den Laufzeitfunktionen verträglich waren. Als Nachteil stellte sich heraus, daß PEARL als Erstellungswerkzeug einerseits zu mächtig und andererseits nicht effektiv genug war (und als Anwendersprache nicht sein konnte).

Aufbauend auf den gewonnenen Erfahrungen wurden weitere Komponenten wie Algorithmik, Betriebssystem etc. entwickelt und auf verschiedenen Maschinen erprobt. Die algorithmischen Laufzeitfunktionen [19] werden in einer maschinenunabhängigen Notation geschrieben und in eine maschinennahe Form umgesetzt. Der hierzu nötige Umsetzer wertet Maschinen-Parameter aus und führt Optimierungen durch.
Das portable Betriebssystem wurde in einer niedrigen abstrakten Assemblersprache formuliert und sah Schnittstellen für maschinenabhängige Primitivfunktionen (Treiber) vor [20]. Die Anpassung dieser Komponenten an eine Zielmaschine besteht aus der Erstellung eines Umsetzers und ggf. der maschinenabhängigen Treiber.

## 3. Die Implentation des Compiliersystems für die S 310

Hinter der 310-Implementation stand die Absicht, im Sinn einer Zwischenbilanz [21], möglichst viele der inzwischen entwickelten portablen Systemsoftware-Komponenten zur Erstellung eines neuen vollständigen Compiliersystems zu benutzen und Erfahrungen mit ihrem Einsatz zu gewinnen. Das Vorgehen bei der Implementation sah vor, an Stellen, wo (noch) keine portabel erstellten Komponenten vorlagen, maschinenabhängige Speziallösungen zu entwickeln. Eventuell vorhandene Hersteller-Systemsoftware war, wenn problemlos möglich, mitzuverwenden. Auf möglichst klare Schnittstellen war zu achten.

Vorteil dieses pragmatischen Vorgehens war, die portablen Komponenten in realistischer Umgebung auf eine Zielmaschine zu bringen.

Die S 310 schien dabei im Rahmen der Zielsetzung als Zielmaschine hinreichend typisch für die derzeit auf dem Markt befindlichen Kleinrechner zu sein. Als 16-bit-Registermaschine verfügt sie über wenig Hardware-Unterstützung bei der Arithmetik. Ihre Unterbrechungsstruktur macht sie für kleinere Prozeßanwendungen geeignet. An Geräten steht neben einer einfachen Standardperipherie ein modulares Prozeßperipheriesystem (z.B. mit Interruptregister, Digital-E/A, Analogeingabe) zur Verfügung. Sie ist mit einem Standard-Betriebssystem ausgerüstet, das auch einen Realzeitbetrieb zuläßt (Interruptverarbeitung, Semaphoroperationen etc.). Darüber hinaus ist ein (Cross-) Assembler und ein Ladebinder verfügbar.

### 3.1 Installation der einzelnen Komponenten

Um Vergleichsmöglichkeiten mit der 306-Implementation zu haben, wurde der schon auf der 306 laufende Compiler-Oberteil beibehalten. Zunächst nicht implementiert wurde die Dateiverwaltung und (wegen fehlender Hardware) die Graphische EA. Für die Anwender hatte dieses Verfahren den Vorteil, daß in der gleichen Sprache zwei verschiedene Rechner programmiert werden konnten. Der Codegenerator wurde wie an der 306 maschinenabhängig erstellt. Als Werkzeug diente allerdings nicht mehr der Makrogenerator Stage II, sondern eine Systemsprache, was die Codegenerator-Laufzeiten merklich verkürzte. Die verwendete Programmstruktur erlaubte reentrantfähige (Laufzeit-) Prozeduren). Zur Versorgung der 310 mit erzeugtem Code wurde eine Rechnerkopplung installiert.

Codegenerator, Programmstruktur und Schnittstellen zu den Laufzeitfunktionen sind genauer in [22] beschrieben.

Die algorithmischen Laufzeitfunktionen wurden teils portabel, teils maschinenspezifisch erstellt. Mit diesen Laufzeitfunktionen, dem Codegenerator und dem Compiler konnte ein algorithmischer PEARL-Subset übersetzt werden. Dies wurde ausgenutzt, um die Standard-EA-Laufzeitfunktionen zu übersetzen. Nach Erstellen der Funktionen zur Formatweiterschaltung war die Standard-EA ablauffähig [18].

Als Betriebssystem stand das portable PEARL-Betriebssystem zur Verfügung. Um es in Betrieb zu nehmen, mußte es von dem abstrakten Assembler-Code in den 310-Code umgesetzt und durch maschinenabhängige Primitiv-Funktionen erweitert werden [23]. Im Gegensatz zu früheren Testeinsätzen wurde es aber nicht auf die "nackte" Maschine aufgesetzt, sondern auf das vorhandene Standard-Betriebssystem. Dadurch waren zwar wiederum Effektivitätseinbußen unausweichlich, doch für den praktischen Betrieb von ausschlaggebender Bedeutung war, daß sämtliche Dienstprogramme mit benutzt werden konnten. Das erleichterte die Handhabung im Testbetrieb, das Binden, Laden und Starten einzelner Programme ungemein: Für das Binden und Laden wurde der Ladebinder des Herstellers benutzt. Fig. 2 zeigt die Aufrufhierarchie der 310-Laufzeitbausteine.

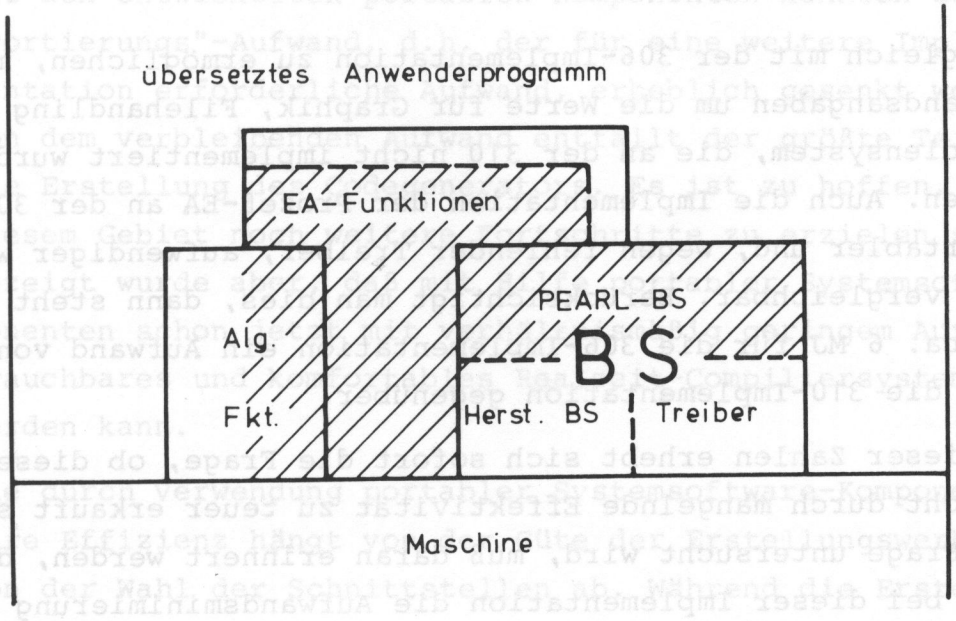



Fig.2 Aufrufhierarchie der 310-Laufzeitbausteine
(Portabel erstellte Lösungen sind gestrichelt angedeutet)

## 3.2 Erfahrungen

Die Implementation des 310-Compiliersystems bestand also im wesentlichen aus Spezialanfertigungen für Codegenerator und Prozeß-EA-Laufzeitfunktionen, sowie aus der Anpassung portabler Laufzeitbausteine für die Standard-EA, die Algorithmik und das Betriebssystem. Der hierzu benötigte Erstellungsaufwand ist aus Tabelle 2 ersichtlich.

Auffällig ist, daß etwa die Hälfte des Gesamtaufwandes auf die Implementation des Codegenerators entfällt.

| | MJ |
|---|---|
| Codegenerator | 1 |
| Laufzeitfunktionen | |
| Standard EA | 1/4 |
| Prozeß-EA | 1/4 |
| Algorithmik | 1/4 |
| Betriebssystem(-Anpassung) | 1/2 |
| | ca. 2 MJ |

Tab.2 S 310-Implementationsaufwand
(Spezialanfertigungen und Anpassung portabler Komponenten)

Um einen Vergleich mit der 306-Implementation zu ermöglichen, müssen die 306-Aufwandsangaben um die Werte für Graphik, Filehandling sowie Test- und Bediensystem, die an der 310 nicht implementiert wurden, reduziert werden. Auch die Implementation der Prozeß-EA an der 306 ist, da sie komfortabler und, wegen fehlender Treiber, aufwendiger war, nicht direkt vergleichbar. Berücksichtigt man dies, dann steht einem Aufwand von ca. 6 MJ für die 306-Implementation ein Aufwand von ca. 2 MJ für die 310-Implementation gegenüber.

Angesichts dieser Zahlen erhebt sich sofort die Frage, ob diese Einsparungen nicht durch mangelnde Effektivität zu teuer erkauft sind. Bevor diese Frage untersucht wird, muß daran erinnert werden, daß die Prämisse bei dieser Implementation die Aufwandsminimierung war, und deswegen keine Optimierungen berücksichtigt wurden, es sei denn, sie waren konzeptionell verankert.
Von der Bearbeitung der Standard-E/A-Laufzeitfunktionen war bekannt, daß sich der Effektivitätsverlust hinsichtlich einer Laufzeitver-

größerung kritischer auswirkt als hinsichtlich einer Speicherplatzerhöhung. Ein für Realzeitanwendungen typischer Test besteht z.B. darin, die Frequenz zu ermitteln, mit der ein Anwenderprogramm im Grenzfall durch Interrupts aus der Prozeßumwelt ("Interrupt-Reaktions-Programm") zyklisch ausgelöst werden kann. Diese Frequenz hängt natürlich von der Dauer des Interrupt-Reaktions-Programms ab. Legt man ein Programm zugrunde, das auch Reaktionsmöglichkeiten erlaubt und einen Betriebssystemaufruf sowie einige weniger zeitkritische Operationen enthält und ca. 4 ms dauert, dann erreicht man damit eine Frequenz von ca. 100 Hz. Die Zeit vom Eintreffen eines Interrupts bis zum Ablauf des ersten PEARL-Statements im Anwenderprogramm beträgt also ca. 6 ms. Der entsprechende Wert bei der 306-Implementation liegt bei ca. 2.5 ms.

Die niedrigen Geschwindigkeiten kommen durch das "mitlaufende" Hersteller-Betriebssystem sowie die (verglichen mit der 306) relativ hohen Operationszeiten zustande und liegen bei einem Betriebssystem, das auf einen "nackten" Mikroprozessor aufgesetzt ist [19] viel niedriger nämlich bei ca. 0.5 ms.

## 4. Zusammenfassung

Überdenkt man die im Rahmen der verschiedenen Implementationen gewonnenen Erfahrungen, zeichnen sich folgende Ergebnisse ab:

- Mit den entwickelten portablen Komponenten konnten der "Portierungs"-Aufwand, d.h. der für eine weitere Implementation erforderliche Aufwand, erheblich gesenkt werden. Von dem verbleibenden Aufwand entfällt der größte Teil auf die Erstellung des Codegenerators. Es ist zu hoffen, daß auf diesem Gebiet noch weitere Fortschritte zu erzielen sind. Gezeigt wurde aber, daß mit Hilfe portabler Systemsoftwarekomponenten schon jetzt mit verhältnismäßig geringem Aufwand ein brauchbares und komfortables Realzeit-Compiliersystem installiert werden kann.

- Die durch Verwendung portabler Systemsoftware-Komponenten erreichbare Effizienz hängt von der Güte der Erstellungswerkzeuge wie von der Wahl der Schnittstellen ab. Während die Erstellungswerkzeuge im allgemeinen vorgegeben sind, hat man die Wahl der Schnittstellen im Rahmen von Design-Entscheidungen z.T. in der Hand: Die hohen Laufzeiten des in Fortran geschriebenen PEARL-Compileroberteils waren durch den ineffizienten Code des verwendeten Fortran-Compilers bedingt. Die Laufzeiteffizienz des übersetzten

PEARL-Programmen hängt von mehreren Faktoren ab: Entscheidet man sich (wie bei der S 310) dafür, das PEARL-Betriebssystem auf das Herstellerbetriebssystem aufzusetzen, kann man Hersteller-SW mitverwenden,muß aber höhere Laufzeiten in Kauf nehmen. Setzt man das PEARL-Betriebssystem auf die nackte Maschine auf, erhält man zwar wesentlich bessere Reaktionszeiten, ist aber mit sonstiger SW inkompatibel. Entscheidet man sich aus Aufwandsgründen dafür, auch die primitivsten maschinennahen Funktionen portabel zu erstellen, erreicht man keine optimalen Laufzeiten. Ersetzt man gewisse kritische Funktionen durch maschinenspezifische Lösungen, erhält man wiederum verbesserte Laufzeitbedingungen.

Unausbleiblich ist bei solchen Betrachtungen natürlich die Frage nach dem Stellenwert der Portabilität selbst. Eine verbindliche Antwort hierauf kann nicht gegeben werden. Es hängt entscheidend vom Anwendungsfall ab, ob man sich teuere, aber effiziente Speziallösungen wirtschaftlich, oder weniger effiziente, dafür aber billigere portable Lösungen technologisch leisten kann. Einen Einfluß auf die Entscheidung dürfte auch die kürzer werdende Generationsfolge neuer Hardware haben, die schnell nach brauchbaren Compiliersystemen verlangt. Es ist Absicht dieses Beitrags, hierzu konkrete Erfahrungen beizusteuern.

12 R. Rössler: Programmstruktur und Laufzeitverwaltung für den PEARL-Subset (Stufe 1) der ASME in der Zielmaschine S 306. PDV-Entwicklungsnotiz E 75, Ges. f. Kernforschung mbH, Karlsruhe, 1976

13 W. Lindstedt: Bedienung von CAMAC-Peripherie mit PEARL-Stufe 1 an der SIEMENS 306; PDV-Entwicklungsnotiz E 78; Ges. f. Kernforschung mbH, Karlsruhe, 1976

14 F.J. Prester: Die graphische Ein-/Ausgabe des Erlanger ASME-PEARL-Subsets; PDV-Entwicklungsnotiz E 104, Ges. f. Kernforschung mbH; Karlsruhe, 1977

15 F.J. Prester: Die Standard EA für den Erlanger ASME-PEARL-Stufe-1-Subset; PDV-Entwicklungsnotiz E 76; Ges. f. Kernforschung mbH, Karlsruhe, 1976

16 P. Holleczek: Das Filehandling für den Erlanger-ASME-PEARL-Subset; PDV-Entwicklungsnotiz E 83, Ges. f. Kernforschung mbH; Karlsruhe, 1976

17 H.B. Berner et al.: VEPAS - Verfahren zur Erstellung von Prozessautomatisierungssoftware - Eine Studie,PDV-Entwicklungsnotiz E 101, Ges. f. Kernforschung mbH, Karlsruhe, 1977

18 K. Lampe, F.J. Prester: Die rechnerunabhängigen Laufzeitroutinen der PEARL-Standard-E/A und ihre Anpassung für die Siemens 310 und 306, PDV-Entwicklungsnotiz, Kernforschungszentrum Karlsruhe GmbH; wird veröffentlicht

19 W. Lindstedt in: Portable Software / hrsg. von H.J. Schneider - Stuttgart: Teubner 1980 (Berichte des German Chapter of the ACM, Bd. 4)

20 R. Rössler: PEARL-Betriebssystem für den Z80, PDV-Entwicklungsnotiz E 119, Kernforschungszentrum Karlsruhe, 1978

21 P. Holleczek in: Portable Software / hrsg. von H.J. Schneider - Stuttgart: Teubner 1980 (Berichte des German Chapter of the ACM, Bd. 4)

22 M. Trautner: Codegenerator - und Systembeschreibung des Siemens 310 - PEARL-Compiliersystems; PDV-Entwicklungsnotiz, Kernforschungszentrum Karlsruhe GmbH; wird veröffentlicht

23 A. Fleischmann, F.J. Prester: Ein PEARL-Betriebssystem für die 310. PDV-Entwicklungsnotiz, Kernforschungszentrum Karlsruhe GmbH, wird veröffentlicht.