Pascalli-P

Markus F. Frisch, Katja Jucht

Report No. 68 4. August 1993

1 Modellbeschreibung

Unser Projekt bestand aus drei Teilpunkten. Der erste Punkt war die Definition einer höheren Programmiersprache Pascalli-P, die, angelehnt an die bekannte Sprache Pascalli, zusätzlich einfache parallele Programmierung ermöglichen soll.
Für diese Sprache wurde dann im zweiten Teil ein Compiler geschrieben, der das Pascalli-P Programm in ein für RAMs (Random Access Machines) verständliches Programm übersetzt, den P-Code. Dazu war es nötig, zusätzlich zu den bisherigen RAM-Befehlen, neue Synchronisationsbefehle zu definieren.
Der dritte Teil bestand aus der Entwicklung eines Simulationsprogrammes für sequentielle Rechner, das mehrere RAMs parallel den P-Code berechnen läßt. Am Ende der Simulation wird die benötigte Zeit ausgegeben, wodurch sich Zeitkomplexitätsbetrachtungen für parallele Berechnungen erstellen lassen.
Bei der Bearbeitung der drei Punkte wurden mehrere Entwurfsprinzipien diskutiert. Für welche Prinzipien wir uns entschieden haben, wird in den folgenden Seiten genauer erläutert.

1.1 Grammatik

Die Grammatik, die im Anhang zu finden ist, unterscheidet sich in mehren Punkten von der Grammatik für Pascalli.
Jedem Prozessor ist eine Prozessornummer PID (Prozessor-ID) beginnend bei Null zugewiesen. Diese PID kann auch, wie aus der Grammatik ersichtlich, im Programm als Term verwendet werden, anstatt konkreter Zahlen oder Variablen. Jeder Prozessor wertet den Wert der Zahl PID für sich unterschiedlich aus. Der Prozessor mit der Prozessor-ID Null würde z. B. PID in seinem Programm immer durch Null ersetzen.
Das Lesen und Schreiben von Variablen anderer Prozessoren erfolgt mittels einer erweiterten Definition der Variablen: Variable @ [Expression]. Damit wird die Variable desjenigen Prozessors angesprochen, dessen PID gleich der ausgewerteten Expression ist.
Um die Synchronisation der Prozessoren zu ermöglichen, wurden zwei verschiedene Modelle implementiert. Diese werden durch eine wait-Anweisung bzw. eine P- und eine V-Anweisung repräsentiert. Die Funktionsweise und die Notwendigkeit einer Synchronisation werden in einem späteren Abschnitt noch genauer erläutert.
Das sind die Änderungen, die nötig waren, um ein paralleles Konzept zu verwirklichen, und die Programmierung paralleler Berechnungen möglichst bequem zu machen.

1.2 P-Code

Aus dem vom Nutzer geschriebenen Programm erzeugt unser Compiler einen Zwischencode, hier P-Code genannt. Dieser Code stimmt weitgehend mit einem RAM-Programm überein. Die einzelnen Instruktionen sind von der Form:

ri := Konstant	(lade Konstante in Register)
ri := rj	(Register kopieren)
ri := rj op rk	(binäre Operation)
ri := op rj	(unäre Operation)
ri := rrj	(indirektes Laden)

rri := rj	(indirektes Speichern)
halt	(beende Berechnung)
if ri goto m else goto n	(bedingte Sprünge)
if ri goto m	(bedingter Sprung)
goto m	(unbedingter Sprung)
in ri	(einlesen von der Tastatur)
out ri	(schreiben auf den Bildschirm)

Zusätzlich kommen für die Parallelverarbeitung noch folgende P-Code Befehle dazu:

ri := PID	(lade PID in Register)
ri := rrj of rk	(Lesen eines Registers eines anderen Prozessors)
rri of rj := rk	(Schreiben eines Registers eines anderen Prozessors)
wait until ri of rk is not zero	(wait-Statement, siehe 1.3.1)
inc ri of rk	(P-Statement, siehe 1.3.1)
Vwait until ri of rk is zero	(V-Statement, siehe 1.3.1)

Jede dieser Anweisungen wird von den Prozessoren als ein Befehl gewertet.

1.3 Simulation

Die Simulation eines Pascalli-P Programmes auf RAMs läuft prinzipiell folgendermaßen ab. Die RAMs, deren Anzahl vom Anwender vorher festgelegt werden muß, erhalten als Eingabeprogramm den P-Code. Jede RAM arbeitet für sich Befehl für Befehl des Programms ab, bis sie auf den Halt-Befehl stößt. Wenn alle RAMs gestoppt haben, wird die Berechnung abgebrochen, und die Anzahl der benötigten Takte ausgegeben.
Bei der Simulation wurden mehrere Prinzipien berücksichtigt, die im folgenden näher erläutert werden.

1.3.1 Synchronisation der Prozessoren

Wir haben in der Simulation eine Abwandlung des SIMD-Modells (Single Instruction-stream, Multiple Data-stream) implementiert. Jede RAM bekommt zu Anfang der Berechnung das gleiche Programm zugewiesen, beginnt also zur gleichen Zeit mit dem gleichen Befehl. Im Gegensatz zum SIMD-Modell hat bei unserem Modell jede RAM einen eigenen Befehlszähler, den sie selbst verwaltet. Damit sind Verzweigungen in der Berechnung möglich.
In manchen Fällen ist es allerdings notwendig, daß bestimmte Prozessoren zur gleichen Zeit ein Konstrukt (z.B. if- oder while-Anweisungen) betreten oder verlassen. In diesen Fällen ist eine Synchronisation der Prozesse notwendig. Wie im Kapitel 1 schon beschrieben, wurde dies mit Hilfe von zwei Modellen gelöst.
Das erste Modell besteht aus einer wait-Anweisung. Trifft ein Prozessor auf einen solchen Programmbefehl, so wartet er, bis die angegebene Variable einen Wert ungleich Null erhält. Dazu ist es notwendig, eine Art „Steuerprozessor“ zu definieren, der den Wert dieser Variable zu einer bestimmten Zeit verändert. Dies ist zum Beispiel sinnvoll, wenn die

Prozessoren warten sollen, bis ein anderer Prozessor einen Wert von der Tastatur eingelesen hat.
Das zweite Modell ist an das Konzept der Semaphore angeknüpft, das in manchen Programmiersprachen implementiert ist. Es gibt zwei Funktionen, die P-Funktion und die V-Funktion. Trifft ein Prozessor auf die P-Funktion, so erhöht er den Wert der angegeben Variable um eins. Trifft er auf eine V-Funktion, so erniedrigt er den Wert der angegeben Variable um eins und wartet bis diese Variable den Wert Null erhält. Dieses Konzept ist vor allem dann sinnvoll, wenn zu Beginn eines Konstruktes noch nicht feststeht, wie viele Prozessoren dieses Konstrukt durchlaufen, diese aber synchronisiert werden sollen. Natürlich ist es auch möglich, die V-Funktion ohne die P-Funktion zu verwenden.

1.3.2 Prozessorkommunikation

Um die Prozessorkommunikation leiten zu können haben wir ein Steuerwerk implementiert. Dieses Steuerwerk lädt das Programm und aktiviert in jedem Takt eine RAM in der Reihenfolge ihrer PIDs, sofern diese noch nicht gehalten haben, indem es der RAM den als nächstes zu bearbeitenden Programmbefehl übergibt. Es überprüft, welche der RAMs schon gehalten haben, und beendet gegebenenfalls die Berechnung.
Außerdem ist das Steuerwerk für die korrekte Bearbeitung von Lese- und Schreibzugriffen auf Register (=Variablen) anderer RAMs zuständig. Da eine parallele Verarbeitung simuliert werden soll, ist es notwendig, daß gleichzeitige Lese- und Schreibzugriffe auf das gleiche Register vermieden werden. Unser Protokoll sieht vor, daß gleichzeitige Lesezugriffe erlaubt sind, bei gleichzeitigen Schreibzugriffen schreibt der Prozessor mit der niedrigsten PID (PRIORITY-CRCW). Alle anderen Schreibzugriffe werden ignoriert. Kommen im gleichen Takt sowohl Lesezugriffe als auch ein Schreibzugriff vor, so wird der Wert gelesen, den die Variable am Anfang des Taktes hatte. Die einzige Ausnahme von dieser Regelung bilden die P- und V-Funktion, da es bei diesem Modell notwendig ist, daß wirklich jeder Prozess den Variablenwert verändert.
Wenn Lesezugriffe auf Register erfolgen, die bisher nicht gebraucht und initialisiert wurden, so wird eine Null gelesen, d.h. alle Register sind zu Beginn der Berechnung mit Null initialisiert.

1.3.3 Komplexitätsberechnungen

Ein Takt ist der Zeitabschnitt, den eine RAM braucht, um genau einen Befehl des P-Codes auszuführen. Ein Takt des Netzwerkes ist zu Ende, wenn alle noch nicht gehaltenen RAMs genau einen Takt gerechnet haben, d.h. wenn alle RAMs genau einen Befehl abgearbeitet haben. Die Anzahl der benötigten Takte der Berechnung wird bestimmt als die Anzahl der Takte der RAM, deren Berechnung am längsten gedauert hat. Anhand dieser Taktzahl kann die Zeitkomplexität der Berechnung in Abhängigkeit von der Eingabe bestimmt werden.

2 Benutzer-Manual

Die Simulation besteht aus mehreren voneinander unabhängigen Programmen.

Das erste Programm **Pcode [<Programmname>]** erzeugt aus dem Pascalli-P File mit dem angegebenen Namen und der Extension .twp den P-Code des Programmes und gibt die Symboltabelle auf die Standardausgabe aus. (Für Neugierige: twp steht für Turbo Wagner parallel.)
Im Anhang findet sich ein einfaches Beispiel für die Umwandlung eines Pascalli-P Programmes in P-Code.

Pcode erzeugt zwei Files: Programmname.stb und Programmname.ram. In Programmname.stb ist eine lesbare Version der Symboltabelle gespeichert, die für die Simulation auch noch von Bedeutung ist.
In Programmname.ram ist der P-Code gespeichert, allerdings nicht lesbar. Um sich den P-Code ansehen zu können, muß das Programm
PrgView [<Programmname>] aufgerufen werden. PrgView hängt an den Programmnamen die Extension .ram an.
Der P-Code wird durch PrgView ausgegeben, wie auf den Seiten 1 und 2 beschrieben. Möchte man statt der Registernummer wieder den ursprünglichen Variablennamen erhalten, so muß man folgenden Befehl eingeben:
PrgView [<Programmname>] | Filter [<Programmname>]. Das Programm Filter führt dann die Ersetzung anhand der Symboltabelle Programmname.stb durch.

Das Programm **Sim -n [<Programmname>]** führt die Simulation des P-Codes durch, und zwar mit so vielen RAMs, wie durch die Option -n angegeben wurde. Der default-Wert für n ist Eins, d.h. eine sequentielle Berechnung einer RAM.
Beim Starten von Sim werden Sie als erstes gefragt, ob sie die volle Berechnung sehen möchten. Wenn sie hier „nein“ wählen, so erhalten sie nur Ausgaben, wenn eine RAM bei ihrer Berechnung auf in oder out stößt. Außerdem wird am Ende die gebrauchte Taktzahl ausgegeben. Die restliche Berechnung bleibt für Sie unsichtbar. Wählen Sie hier „ja“, so gibt jede RAM, die gerade rechnet, zuerst ihre PID, dann den Befehl, den sie gerade bearbeitet, und zuletzt die Berechnung aus. Dies hat den Vorteil, daß Sie den Programmverlauf besser verfolgen können, und den Nachteil, daß die Ausgabe unübersichtlich wird. Auch hier sorgt **Sim -n [<Programmname>] | Filter [<Programmname>]** dafür, daß in der Ausgabe die Register durch ihre ursprünglichen Variablennamen ersetzt werden. Bei der „kleinen“ Ausgabe erfolgt diese Umwandlung automatisch. Ein Beispiel dafür können Sie im Anhang finden.

Wird beim Aufruf eines unserer Programme kein Programmname mit angegeben, so wird als Default-Name „my“ angenommen, d.h. das Pascalli-Programm muß in einem File namens „my.twp“ stehen.
Die Standard Ein- und Ausgabe kann natürlich wie in Unix üblich umgelenkt werden.

Anhang A: Grammatik für Pascalli-P

Kommentare können an jeder Stelle des Programmes innerhalb geschweifter Klammern eingefügt werden.

program	::	**program id ;** declare subprogrdeclare compound-statement **.**
declare	::	**var** declarelist | ε
declarelist	::	declarelist identlist : type ; | ε
identlist	::	**id** | identlist, **id**
type	::	**integer** | **array [num] of integer**
subprogrdeclare	::	subprogrdeclare subdeclare | ε
subdeclare	::	head declare compound-statement ;
head	::	**function id** arguments : integer ; | **procedure id** arguments ;
arguments	::	(parameterlist) | (ε) | ε
parameterlist	::	identlist : type | parameterlist ; identlist : type
compound-statement	::	**begin** optstatements **end**
optstatements	::	statementlist | ε
statementlist	::	statement ; | statementlist statement ;
statement	::	xvariable := expres | procedurestatement | compound-statement | **if** expres **then** statement **else** statement | **if** expres **then** statement | **while** expres **do** statement | **read** (variable) | **write** (variable) | **wait** (variable , expres)

		| **P** (variable , expres)
		| **V** (variable , expres)
		| ε
xvariable	::	variable | variable @ [expres]
variable	::	**id** | **id** [expres]
procedurestatement	::	**id** (expreslist)
expreslist	::	expres | expreslist , expres | ε
expres	::	expres2 | expres2 **relop** expres2
expres2	::	term | expres2 **addop** term
term	::	factor | term **mulop** factor
factor	::	xvariable
		| **id** (expreslist)
		| **num**
		| (expres)
		| **not** factor
		| **PID**

relop :: $= \mid < \mid > \mid <= \mid >= \mid <>$
addop :: $+ \mid - \mid$**or**
mulop :: $* \mid / \mid$**and**|**mod**

id :: letter (letter | digit)*
letter :: [a–z A–Z]
digit :: [0–9]
optvorz :: $(+ \mid - \mid \varepsilon)$
num :: optvorz digit (digit)*

Anhang B: Beispielprogramm 1

Dieses Beispiel soll die Arbeitsweise des Übersetzers in Zwischencode verdeutlichen.

```
program Summe;
var a,m,n,s,t: integer;
begin
    if PID = 0 then begin
        read(n);
        s := 1;
        s := 0;
        wait(s,0);
        t := a@[1];
        write(t);
    end
    else begin
        wait(s,0);
        if PID <= n@[0] then begin
            read(a);
            m := 1;
            while m <= n@[0] do begin
                a := a + a@[PID+m];
                m := 2+m;
            end;
            write(a);
            if PID=1 then s@[0] := 1;
        end;
    end;
end.
```

Dieses Programm wird in den P-Code übersetzt, der auf der folgenden Seite zu finden ist.

P-Code für das Programm Summe, erzeugt durch „Pcode Summe“, aufgerufen durch „PrgView Summe | Filter Summe“:

```
(0) r2 = 43
(1) r3 = 44
(2) goto 3
(3) r15 = PID
(4) r16 = 0
(5) r17 = r15 == r16
(6) if r17 goto7 else goto 19
(7) in n
(8) r18 = 1
(9) s = r18
(10) r19 = 0
(11) s = r19
(12) r20 = 0
(13) Wait until s of r20 is not zero
(14) r21 = 1
(15) r22 = a of r21
(16) t = r22
(17) out t
(18) goto 50
(19) r23 = 0
(20) Wait until s of r23 is not zero
(21) r24 = PID
(22) r25 = 0
(23) r26 = n of r25
(24) r27 = r24 <= r26
(25) if r27 goto 26 else goto 50
(26) in a
(27) r28 = 1
(28) m = r28
(29) r29 = 0
(30) r30 = n of r29
(31) r31 = m <= r30
(32) if r31 goto 33 else goto 42
(33) r32 = PID
(34) r33 = r32 + m
(35) r34 = a of r33
(36) r35 = a + 34
(37) a = r35
(38) r36 = 2
(39) r37 = r36 * m
(40) m = r37
(41) goto 29
(42) out a
(43) r38 = PID
(44) r39 = 1
(45) r40 = r38 == r39
(46) if r40 goto 47 else goto 50
(47) r41 = 0
(48) r42 = 1
(49) s of r41 = r42
(50) halt
```

Aufruf von „Sim -5 Summe“ mit beschränkter Ausgabe:

```
Wollen Sie die volle Ausgabe ? (y/n) n
Bitte geben sie fuer Prozessor 0 die Variable n ein: 2
Bitte geben sie fuer Prozessor 1 die Variable a ein: 15
Bitte geben sie fuer Prozessor 2 die Variable a ein: 23
Die Variable a von Prozessor 1 hat den Wert 38
Die Variable a von Prozessor 2 hat den Wert 23
Die Variable t von Prozessor 0 hat den Wert 38
Die Berechnung hat bei dieser Eingabe 64 Takte gebraucht.
```

Auszug aus der Ausgabe bei Aufruf von „Sim -5 Summe“ mit voller Ausgabe:

```
Wollen Sie die volle Ausgabe ? (y/n) y
pid = 0  | (0) r2 = 43       Berechnung: r2 := 43
pid = 1  | (0) r2 = 43       Berechnung: r2 := 43
pid = 2  | (0) r2 = 43       Berechnung: r2 := 43
pid = 3  | (0) r2 = 43       Berechnung: r2 := 43
pid = 4  | (0) r2 = 43       Berechnung: r2 := 43
pid = 0  | (1) r3 = 44       Berechnung: r3 := 44
pid = 1  | (1) r3 = 44       Berechnung: r3 := 44
pid = 2  | (1) r3 = 44       Berechnung: r3 := 44
pid = 3  | (1) r3 = 44       Berechnung: r3 := 44
pid = 4  | (1) r3 = 44       Berechnung: r3 := 44
pid = 0  | (2) goto 3
pid = 1  | (2) goto 3
pid = 2  | (2) goto 3
pid = 3  | (2) goto 3
pid = 4  | (2) goto 3
pid = 0  | (3) r15 = PID     Die Pid dieses Prozessors lautet 0
pid = 1  | (3) r15 = PID     Die Pid dieses Prozessors lautet 1
pid = 2  | (3) r15 = PID     Die Pid dieses Prozessors lautet 2
pid = 3  | (3) r15 = PID     Die Pid dieses Prozessors lautet 3
pid = 4  | (3) r15 = PID     Die Pid dieses Prozessors lautet 4
```

usw.

Komplexitätsbetrachtung des Programms Summe:

Um die Zeitkomplexität des Programms zu berechnen, wurde die gebrauchte Taktzahl für verschiedene Eingabelängen betrachtet:

Eingabelänge	gebrauchte Taktzahl	1. Differenz
1	57	
		15
2	72	
		15
4	87	
		15
8	102	
		15
16	117	

Da die Taktzahl nach der ersten Differenz schon konstant bleibt, läßt sich die Zeitkomplexität des Programms Summe bestimmen als: $t(n) = \log(n)$.

Anhang B: Beispielprogramm 2

```
{ Sortiernetzwerk nach Batcher  }
{ Sortieren durch Mischen bitoner Folgen }

program BatcherSort;

var n,a,b,s,t,offset,blocklen: integer;

function min(x,y: integer): integer;
begin
  if x <= y then min := x else min := y;
end;

function max(x,y: integer): integer;
begin
  if y >= x then max := y else max := x;
end;


procedure SORT();
begin
  blocklen := 1;              { Laenge der zu sortierenden Folgen    }
  while blocklen < n do       { laeuft von 2..n                      }
    begin
      offset := blocklen;     { offset fuer Vergleich                }
      blocklen := blocklen*2; { ist halbe Folgenlaenge               }
      while offset > 0 do
        begin
          if PID mod (2*blocklen) < blocklen    { PID/blocklen gerade? }
          then { aufsteigend sortieren}

 { zum Sortieren einer Folge benoetigen wir blocklen/2 Komparatoren    }
 { jeder Komparator wird simuliert durch 2 Prozessoren:                }
 { die ersten blocklen/2 Prozessoren berechnen oberen Komp.-Ausgang    }
 { die anderen berechnen unteren Komparator-Ausgang                    }

               if PID mod (2*offset) < offset   { PID/offset gerade?   }
               then                             { oberer Ausgang       }
                 begin
                   b := a@[PID+offset];
                   V(s,0); s := n;     { a erst schreiben, nachdem alle}
                   a := min(a,b);      { gelesen haben                 }
                 end
               else                    { unterer Ausgang               }
                 begin
```

```
                  b := a@[PID-offset];
                  V(s,0); s := n;
                  a := max(b,a);
                end
          else { absteigend sortieren}
            if PID mod (2*offset) < offset
            then
              begin
                b := a@[PID+offset];
                V(s,0); s := n;
                a := max(a,b);
              end
            else
              begin
                b := a@[PID-offset];
                V(s,0); s := n;
                a := min(b,a);
              end;
          offset := offset/2;
        end;
    end;
end;



{ main }

begin
  if PID = 0 then  { Prozessor 0 liest Anzahl der Elemente               }
    begin
      read(n);     { Anzahl Elemente; immer Zweierpotenz bitte!          }
      s := n;      { Semaphor; n Prozessoren sollen synchronisiert werden}
      t  := 1;
    end
  else wait(t,0);  { die anderen warten auf Prozessor 0                  }
  n := n@[0];
  if PID < n then
    begin
      V(s,0);      { jetzt sind Prozessoren 0..n-1 synchron              }
      s := n;
      read(a);     { jeder Prozessor liest einen Wert                    }
      SORT();
      V(s,0);      { jetzt sind Prozessoren 0..n-1 synchron              }
      write(a);
    end;
end.
```

Auch hier wurde die Zeitkomplexität durch verschiedene Eingabelängen berechnet:

Eingabelänge	gebrauchte Taktzahl	1. Differenz	2. Differenz
2	789		
		242	
4	1031		116
		358	
8	1389		116
		474	
16	1863		116
		590	
32	2453		116
		706	
64	3159		

Da die Taktzahl nach der zweiten Differenz konstant bleibt, läßt sich die Zeitkomplexität des Programms Batcher bestimmen als: $t(n) = \log^2(n)$.

Aufruf von „Sim -16 Batcher“ mit beschränkter Ausgabe:

```
Wollen Sie die volle Ausgabe ? (y/n) n
Bitte geben sie fuer Prozessor 0 die Variable n ein: 16
Bitte geben sie fuer Prozessor 0 die Variable a ein: 4
Bitte geben sie fuer Prozessor 1 die Variable a ein: 11
Bitte geben sie fuer Prozessor 2 die Variable a ein: -1
Bitte geben sie fuer Prozessor 3 die Variable a ein: 0
Bitte geben sie fuer Prozessor 4 die Variable a ein: 14
Bitte geben sie fuer Prozessor 5 die Variable a ein: 3
Bitte geben sie fuer Prozessor 6 die Variable a ein: 7
Bitte geben sie fuer Prozessor 7 die Variable a ein: 9
Bitte geben sie fuer Prozessor 8 die Variable a ein: 1
Bitte geben sie fuer Prozessor 9 die Variable a ein: 8
Bitte geben sie fuer Prozessor 10 die Variable a ein: 10
Bitte geben sie fuer Prozessor 11 die Variable a ein: 2
Bitte geben sie fuer Prozessor 12 die Variable a ein: 12
Bitte geben sie fuer Prozessor 13 die Variable a ein: 5
Bitte geben sie fuer Prozessor 14 die Variable a ein: 13
Bitte geben sie fuer Prozessor 15 die Variable a ein: 6
Die Variable a von Prozessor 0 hat den Wert -1
Die Variable a von Prozessor 1 hat den Wert 0
Die Variable t von Prozessor 2 hat den Wert 1
Die Variable a von Prozessor 3 hat den Wert 2
Die Variable a von Prozessor 4 hat den Wert 3
Die Variable a von Prozessor 5 hat den Wert 4
Die Variable a von Prozessor 6 hat den Wert 5
Die Variable a von Prozessor 7 hat den Wert 6
Die Variable a von Prozessor 8 hat den Wert 7
Die Variable a von Prozessor 9 hat den Wert 8
Die Variable a von Prozessor 10 hat den Wert 9
Die Variable a von Prozessor 11 hat den Wert 10
Die Variable a von Prozessor 12 hat den Wert 11
Die Variable a von Prozessor 13 hat den Wert 12
Die Variable a von Prozessor 14 hat den Wert 13
Die Variable a von Prozessor 15 hat den Wert 14
Die Berechnung hat bei dieser Eingabe 1863 Takte gebraucht.
```

Auf den folgenden Seiten finden Sie den P-Code des Programmes Batcher, aufgerufen durch „PrgView Batcher | Filter Batcher“:

```
#3 | (0) r2 = 60
#3 | (1) r3 = 61
#3 | (2) goto 263
#0 | (3) r5 = 1
#0 | (4) r1 = r2
#0 | (5) r2 = r2 + r5
#0 | (6) r9 = 17
#0 | (7) r8 = 2
#0 | (8) if r8 goto 9 else goto 17
#0 | (9) r6 = rr9
#0 | (10) r7 = rr2
#0 | (11) rr2 = r6
#0 | (12) rr9 = r7
#0 | (13) r2 = r2 + r5
#0 | (14) r9 = r9 + r5
#0 | (15) r8 = r8 - r5
#0 | (16) goto 8
#0 | (17) r6 = 3
#0 | (18) r7 = 2
#0 | (19) r8 = r6 - r7
#0 | (20) if r8 goto 21 else goto 27
#0 | (21) r6 = rr9
#0 | (22) rr2 = r6
#0 | (23) r2 = r2 + r5
#0 | (24) r9 = r9 + r5
#0 | (25) r8 = r8 - r5
#0 | (26) goto 20
#0 | (27) rr2 = r0
#0 | (28) r3 = r2 + r5
#0 | (29) r19 = x <= y
#0 | (30) if r19 goto 31 else goto 33
#0 | (31) r1 = x
#0 | (32) goto 34
#0 | (33) r1 = y
#0 | (34) r5 = 1
#0 | (35) r0 = rr2
#0 | (36) r2 = r2 - r5
#0 | (37) r9 = 17
#0 | (38) r8 = 3
#0 | (39) r9 = r9 + r8
#0 | (40) r9 = r9 - r5
#0 | (41) if r8 goto 42 else goto 48
#0 | (42) r7 = rr2
```

```
#0 | (43) rr9 = r7
#0 | (44) r2 = r2 - r5
#0 | (45) r9 = r9 - r5
#0 | (46) r8 = r8 - r5
#0 | (47) goto 41
#0 | (48) r8 = 3
#0 | (49) r3 = r2 + r5
#0 | (50) return
#1 | (51) r5 = 1
#1 | (52) r1 = r2
#1 | (53) r2 = r2 + r5
#1 | (54) r9 = 17
#1 | (55) r8 = 2
#1 | (56) if r8 goto 57 else goto 65
#1 | (57) r6 = rr9
#1 | (58) r7 = rr2
#1 | (59) rr2 = r6
#1 | (60) rr9 = r7
#1 | (61) r2 = r2 + r5
#1 | (62) r9 = r9 + r5
#1 | (63) r8 = r8 - r5
#1 | (64) goto 56
#1 | (65) r6 = 3
#1 | (66) r7 = 2
#1 | (67) r8 = r6 - r7
#1 | (68) if r8 goto 69 else goto 75
#1 | (69) r6 = rr9
#1 | (70) rr2 = r6
#1 | (71) r2 = r2 + r5
#1 | (72) r9 = r9 + r5
#1 | (73) r8 = r8 - r5
#1 | (74) goto 68
#1 | (75) rr2 = r0
#1 | (76) r3 = r2 + r5
#1 | (77) r19 = y (null) x
#1 | (78) if r19 goto 79 else goto 81
#1 | (79) r1 = y
#1 | (80) goto 82
#1 | (81) r1 = x
#1 | (82) r5 = 1
#1 | (83) r0 = rr2
#1 | (84) r2 = r2 - r5
#1 | (85) r9 = 17
#1 | (86) r8 = 3
#1 | (87) r9 = r9 + r8
#1 | (88) r9 = r9 - r5
```

```
#1 | (89) if r8 goto 90 else goto 96
#1 | (90) r7 = rr2
#1 | (91) rr9 = r7
#1 | (92) r2 = r2 - r5
#1 | (93) r9 = r9 - r5
#1 | (94) r8 = r8 - r5
#1 | (95) goto 89
#1 | (96) r8 = 3
#1 | (97) r3 = r2 + r5
#1 | (98) return
#2 | (99) r5 = 1
#2 | (100) r1 = r2
#2 | (101) r2 = r2 + r5
#2 | (102) r9 = 17
#2 | (103) r8 = 0
#2 | (104) if r8 goto 105 else goto 113
#2 | (105) r6 = rr9
#2 | (106) r7 = rr2
#2 | (107) rr2 = r6
#2 | (108) rr9 = r7
#2 | (109) r2 = r2 + r5
#2 | (110) r9 = r9 + r5
#2 | (111) r8 = r8 - r5
#2 | (112) goto 104
#2 | (113) r6 = 43
#2 | (114) r7 = 0
#2 | (115) r8 = r6 - r7
#2 | (116) if r8 goto 117 else goto 123
#2 | (117) r6 = rr9
#2 | (118) rr2 = r6
#2 | (119) r2 = r2 + r5
#2 | (120) r9 = r9 + r5
#2 | (121) r8 = r8 - r5
#2 | (122) goto 116
#2 | (123) rr2 = r0
#2 | (124) r3 = r2 + r5
#2 | (125) r17 = 1
#2 | (126) blocklen = r17
#2 | (127) r18 = blocklen <  n
#2 | (128) if r18 goto 129 else goto 246
#2 | (129) offset = blocklen
#2 | (130) r19 = 2
#2 | (131) r20 = blocklen * r19
#2 | (132) blocklen = r20
#2 | (133) r21 = 0
#2 | (134) r22 = offset >  r21
```

```
#2 | (135) if r22 goto 136 else goto 245
#2 | (136) r23 = PID
#2 | (137) r24 = 2
#2 | (138) r25 = r24 * blocklen
#2 | (139) r26 = r23 % r25
#2 | (140) r27 = r26 <  blocklen
#2 | (141) if r27 goto 142 else goto 192
#2 | (142) r28 = PID
#2 | (143) r29 = 2
#2 | (144) r30 = r29 * offset
#2 | (145) r31 = r28 % r30
#2 | (146) r32 = r31 <  offset
#2 | (147) if r32 goto 148 else goto 170
#2 | (148) r33 = PID
#2 | (149) r34 = r33 + offset
#2 | (150) r35 = 11
#2 | (151) r35 = r35 of r34
#2 | (152) b = r35
#2 | (153) r36 = 0
#2 | (154) Vwait until s of r36 is zero
#2 | (155) s = n
#2 | (156) rr3 = a
#2 | (157) r5 = 1
#2 | (158) r3 = r3 + r5
#2 | (159) rr3 = b
#2 | (160) r5 = 1
#2 | (161) r3 = r3 + r5
#2 | (162) r6 = 4
#2 | (163) r5 = 163
#2 | (164) r5 = r5 + r6
#2 | (165) r0 = r5
#2 | (166) goto min
#2 | (167) r37 = r1
#2 | (168) a = r37
#2 | (169) goto 191
#2 | (170) r38 = PID
#2 | (171) r39 = r38 - offset
#2 | (172) r40 = 11
#2 | (173) r40 = r40 of r39
#2 | (174) b = r40
#2 | (175) r41 = 0
#2 | (176) Vwait until s of r41 is zero
#2 | (177) s = n
#2 | (178) rr3 = b
#2 | (179) r5 = 1
#2 | (180) r3 = r3 + r5
```

```
#2 | (181) rr3 = a
#2 | (182) r5 = 1
#2 | (183) r3 = r3 + r5
#2 | (184) r6 = 4
#2 | (185) r5 = 185
#2 | (186) r5 = r5 + r6
#2 | (187) r0 = r5
#2 | (188) goto max
#2 | (189) r42 = r1
#2 | (190) a = r42
#2 | (191) goto 241
#2 | (192) r43 = PID
#2 | (193) r44 = 2
#2 | (194) r45 = r44 * offset
#2 | (195) r46 = r43 % r45
#2 | (196) r47 = r46 <  offset
#2 | (197) if r47 goto 198 else goto 220
#2 | (198) r48 = PID
#2 | (199) r49 = r48 + offset
#2 | (200) r50 = 11
#2 | (201) r50 = r50 of r49
#2 | (202) b = r50
#2 | (203) r51 = 0
#2 | (204) Vwait until s of r51 is zero
#2 | (205) s = n
#2 | (206) rr3 = a
#2 | (207) r5 = 1
#2 | (208) r3 = r3 + r5
#2 | (209) rr3 = b
#2 | (210) r5 = 1
#2 | (211) r3 = r3 + r5
#2 | (212) r6 = 4
#2 | (213) r5 = 213
#2 | (214) r5 = r5 + r6
#2 | (215) r0 = r5
#2 | (216) goto max
#2 | (217) r52 = r1
#2 | (218) a = r52
#2 | (219) goto 241
#2 | (220) r53 = PID
#2 | (221) r54 = r53 - offset
#2 | (222) r55 = 11
#2 | (223) r55 = r55 of r54
#2 | (224) b = r55
#2 | (225) r56 = 0
#2 | (226) Vwait until s of r56 is zero
```

```
#2 | (227) s = n
#2 | (228) rr3 = b
#2 | (229) r5 = 1
#2 | (230) r3 = r3 + r5
#2 | (231) rr3 = a
#2 | (232) r5 = 1
#2 | (233) r3 = r3 + r5
#2 | (234) r6 = 4
#2 | (235) r5 = 235
#2 | (236) r5 = r5 + r6
#2 | (237) r0 = r5
#2 | (238) goto min
#2 | (239) r57 = r1
#2 | (240) a = r57
#2 | (241) r58 = 2
#2 | (242) r59 = offset / r58
#2 | (243) offset = r59
#2 | (244) goto 133
#2 | (245) goto 127
#2 | (246) r5 = 1
#2 | (247) r0 = rr2
#2 | (248) r2 = r2 - r5
#2 | (249) r9 = 17
#2 | (250) r8 = 43
#2 | (251) r9 = r9 + r8
#2 | (252) r9 = r9 - r5
#2 | (253) if r8 goto 254 else goto 260
#2 | (254) r7 = rr2
#2 | (255) rr9 = r7
#2 | (256) r2 = r2 - r5
#2 | (257) r9 = r9 - r5
#2 | (258) r8 = r8 - r5
#2 | (259) goto 253
#2 | (260) r8 = 43
#2 | (261) r3 = r2 + r5
#2 | (262) return
#3 | (263) r17 = PID
#3 | (264) r18 = 0
#3 | (265) r19 = r17 == r18
#3 | (266) if r19 goto 267 else goto 272
#3 | (267) in n
#3 | (268) s = n
#3 | (269) r20 = 1
#3 | (270) t = r20
#3 | (271) goto 274
#3 | (272) r21 = 0
```

```
#3 | (273) Wait until t of r21 is zero
#3 | (274) r22 = 0
#3 | (275) r23 = 10
#3 | (276) r23 = r23 of r22
#3 | (277) n = r23
#3 | (278) r24 = PID
#3 | (279) r25 = r24 <  n
#3 | (280) if r25 goto 281 else goto 293
#3 | (281) r26 = 0
#3 | (282) Vwait until s of r26 is zero
#3 | (283) s = n
#3 | (284) in a
#3 | (285) r6 = 4
#3 | (286) r5 = 286
#3 | (287) r5 = r5 + r6
#3 | (288) r0 = r5
#3 | (289) goto SORT
#3 | (290) r27 = 0
#3 | (291) Vwait until s of r27 is zero
#3 | (292) out a
#3 | (293) halt
```

Preprint-Reihe
Institut für Informatik
Universität Würzburg

Verantwortlich: Die Vorstände des Institutes für Informatik.

[1] K. Wagner. *Bounded query classes.* Februar 1989.

[2] P. Tran-Gia. *Application of the discrete transforms in performance modeling and analysis.* Februar 1989.

[3] U. Hertrampf. *Relations among mod-classes.* Februar 1989.

[4] K. W. Wagner. *Number-of-query hierarchies.* Februar 1989.

[5] E. W. Allender. *A note on the power of threshold circuits.* Juli 1989.

[6] P. Tran-Gia und Th. Stock. *Approximate performance analysis of the DQDB access protocol.* August 1989.

[7] M. Kowaluk und K. W. Wagner. *Die Vektor-Sprache: Einfachste Mittel zur kompakten Beschreibung endlicher Objekte.* August 1989.

[8] M. Kowaluk und K. W. Wagner. *Vektor-Reduzierbarkeit.* August 1989.

[9] K. W. Wagner (Herausgeber). *9. Workshop über Komplexitätstheorie, effiziente Algorithmen und Datenstrukturen.* November 1989.

[10] R. Gutbrod. *A transformation system for chain code picture languages: Properties and algorithms.* Januar 1990.

[11] Th. Stock und P. Tran-Gia. *A discrete-time analysis of the DQDB access protocol with general input traffic.* Februar 1990.

[12] E. W. Allender und U. Hertrampf. *On the power of uniform families of constant depth threshold circuits.* Februar 1990.

[13] G. Buntrock, L. A. Hemachandra und D. Siefkes. *Using inductive counting to simulate nondeterministic computation.* April 1990.

[14] F. Hübner. *Analysis of a finite capacity a synchronous multiplexer with periodic sources.* Juli 1990.

[15] G. Buntrock, C. Damm, U. Hertrampf und C. Meinel. *Structure and importance of logspace-MOD-classes.* Juli 1990.

[16] H. Gold und P. Tran-Gia. *Performance analysis of a batch service queue arising out of manufacturing systems modeling.* Juli 1990.

[17] F. Hübner und P. Tran-Gia. *Quasi-stationary analysis of a finite capacity asynchronous multiplexer with modulated deterministic input.* Juli 1990.

[18] U. Huckenbeck. *Complexity and approximation theoretical properties of rational functions which map two intervals into two other ones.* August 1990.

[19] P. Tran-Gia. *Analysis of polling systems with general input process and finite capacity.* August 1990.

[20] C. Friedewald, A. Hieronymus und B. Menzel. *WUMPS Würzburger message passing system.* Oktober 1990.

[21] R. V. Book. *On random oracle separations.* November 1990.

[22] Th. Stock. *Influences of multiple priorities on DQDB protocol performance.* November 1990.

[23] P. Tran-Gia und R. Dittmann. *Performance analysis of the CRM a-protocol in high-speed networks.* Dezember 1990.

[24] C. Wrathall. *Confluence of one-rule Thue systems.*

[25] O. Gihr und P. Tran-Gia. *A layered description of ATM cell traffic streams and correlation analysis.* Januar 1991.

[26] H. Gold und F. Hübner. *Multi server batch service systems in push and pull operating mode — a performance comparison.* Juni 1991.

[27] H. Gold und H. Grob. *Performance analysis of a batch service system operating in pull mode.* Juli 1991.

[28] U. Hertrampf. *Locally definable acceptance types—the three valued case.* Juli 1991.

[29] U. Hertrampf. *Locally definable acceptance types for polynomial time machines.* Juli 1991.

[30] Th. Fritsch und W. Mandel. *Communication network routing using neural nets – numerical aspects and alternative approaches.* Juli 1991.

[31] H. Vollmer und K. W. Wagner. *Classes of counting functions and complexity theoretic operators.* August 1991.

[32] R. V. Book, J. H. Lutz und K. W. Wagner. *On complexity classes and algorithmically random languages.* August 1991.

[33] F. Hübner. *Queueing analysis of resource dispatching and scheduling in multi-media systems.* September 1991.

[34] H. Gold und G. Bleckert. *Analysis of a batch service system with two heterogeneous servers.* September 1991.

[35] H. Vollmer und K. W. Wagner. *Complexity of functions versus complexity of sets.* Oktober 1991.

[36] F. Hübner. *Discrete-time analysis of the output process of an atm multiplexer with periodic input.* November 1991.

[37] P. Tran-Gia und O. Gropp. *Structure and performance of neural nets in broadband system admission control.* November 1991.

[38] G. Buntrock und K. Loryś. *On growing context-sensitive languages.* Januar 1992.

[39] K. W. Wagner. *Alternating machines using partially defined “AND” and “OR”.* Januar 1992.

[40] F. Hübner und P. Tran-Gia. *An analysis of multi-service systems with trunk reservation mechanisms.* April 1992.

[41] U. Huckenbeck. *On a generalization of the bellman-ford-algorithm for acyclic graphs.* Mai 1992.

[42] U. Huckenbeck. *Cost-bounded paths in networks of pipes with valves.* Mai 1992.

[43] F. Hübner. *Autocorrelation and power density spectrum of atm multiplexer output processes.* September 1992.

[44] F. Hübner und M. Ritter. *Multi-service broadband systems with CBR and VBR input traffic.* Oktober 1992.

[45] M. Mittler und P. Tran-Gia. *Performance of a neural net scheduler used in packet switching interconnection networks.* Oktober 1992.

[46] M. Kowaluk und K. W. Wagner. *Vector language: Simple description of hard instances.* Oktober 1992.

[47] B. Menzel und J. Wolff von Gudenberg. *Kommentierte Syntaxdiagramme für C++.* November 1992.

[48] D. Emme. *A kernel for funtions definable classes and its relations to lowness.* November 1992.

[49] S. Öhring. *On dynamic and modular embeddings into hyper de Bruijn networks.* November 1992.

[50] K. Poeck und M. Tins. *An intelligent tutoring system for classification problem solving.* November 1992.

[51] K. Poeck und F. Puppe. *COKE: Efficient solving of complex assignment problems with the propose-and-exchange method.* November 1992.

[52] Th. Fritsch, M. Mittler und P. Tran-Gia. *Artificial neural net applications in telecommunication systems.* Dezember 1992.

[53] H. Vollmer und K. W. Wagner. *The complexity of finding middle elements.* Januar 1993.

[54] O. Gihr, H. Gold und S. Heilmann. *Analysis of machine breakdown models.* Januar 1993.

[55] S. Öhring. *Optimal dynamic embeddings of arbitrary trees in de Bruijn networks.* Februar 1993.

[56] M. Mittler. *Analysis of two finite queues coupled by a triggering scheduler.* März 1993.

[57] J. Albert, F. Duckstein, M. Lautner und B. Menzel. *Message-passing auf transputer-systemen.* März 1993.

[58] Th. Stock und P. Tran-Gia. *Basic concepts and performance of high-speed protocols.* März 1993.

[59] F. Hübner. *Dimensioning of a peak cell rate monitor algorithm using discrete-time analysis.* März 1993.

[60] G. Buntrock und K. Loryś. *The variable membership problem: Succinctness versus complexity.* April 1993.

[61] H. Gold und B. Frötschl. *Performance analysis of a batch service system working with a combined push/pull control.* April 1993.

[62] H. Vollmer. *On different reducibility notions for function classes.* April 1993.

[63] S. Öhring und S. K. Das. *Folded Petersen Cube Networks: New Competitors for the Hyepercubes.* Mai 1993.

[64] S. Öhring und S. K. Das. *Incomplete Hypercubes: Embeddings of Tree-Related Networks.* Mai 1993.

[65] S. Öhring und S. K. Das. *Mapping Dynamic Data and Algorithm Structures on Product Networks.* Mai 1993.

[66] F. Hübner und P. Tran-Gia. *A Discrete-Time Analysis of Cell Spacing in ATM Systems.* Juni 1993.

[67] R. Dittmann und F. Hübner. *Discrete-Time Analysis of a Cyclic Service System with Gated Limited Service.* Juni 1993.

[68] M. Frisch und K. Jucht. *Pascalli-P.* August 1993.

[69] G. Buntrock. *Growing Context-Sensitive Languages and Automata.* September 1993.