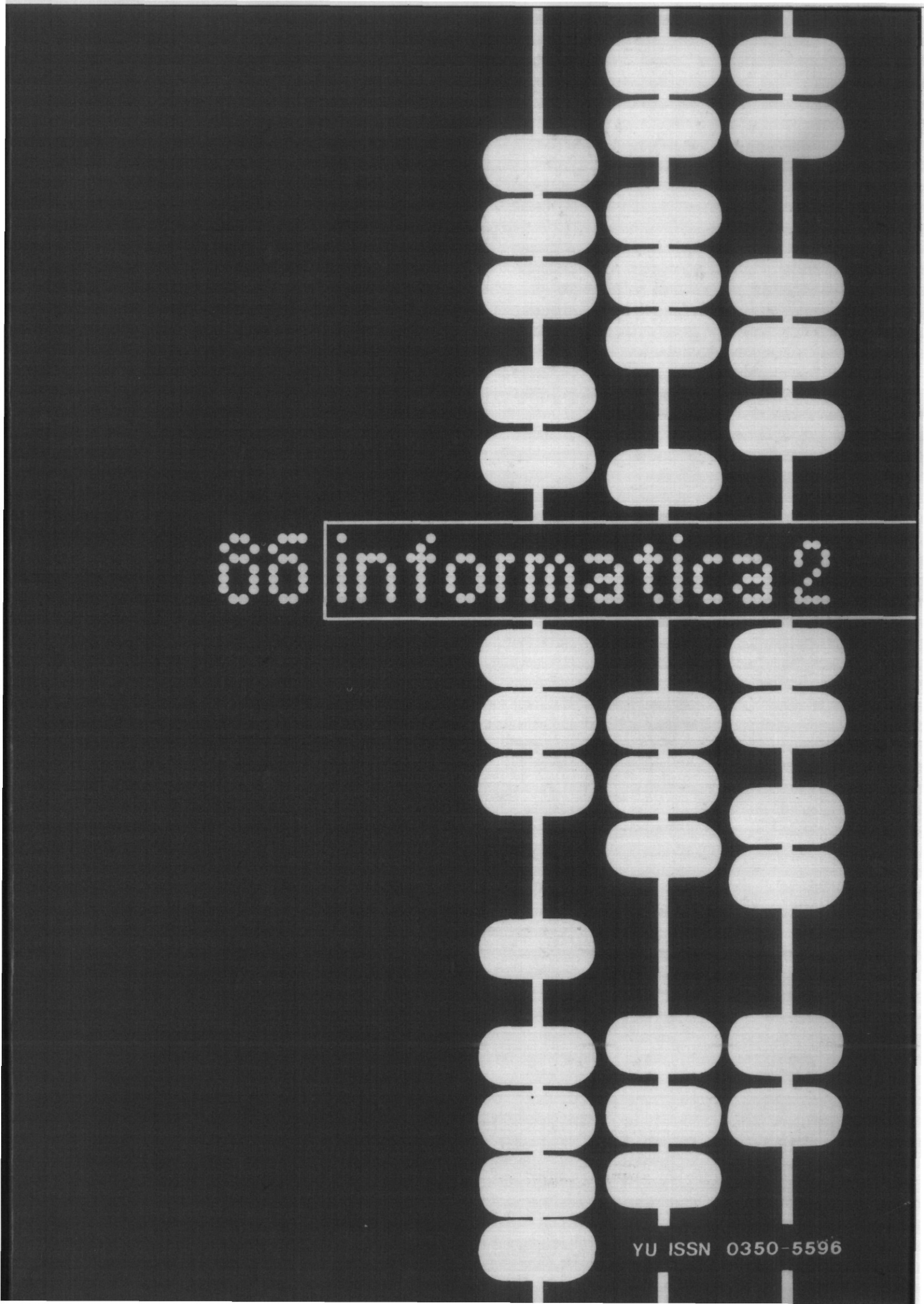
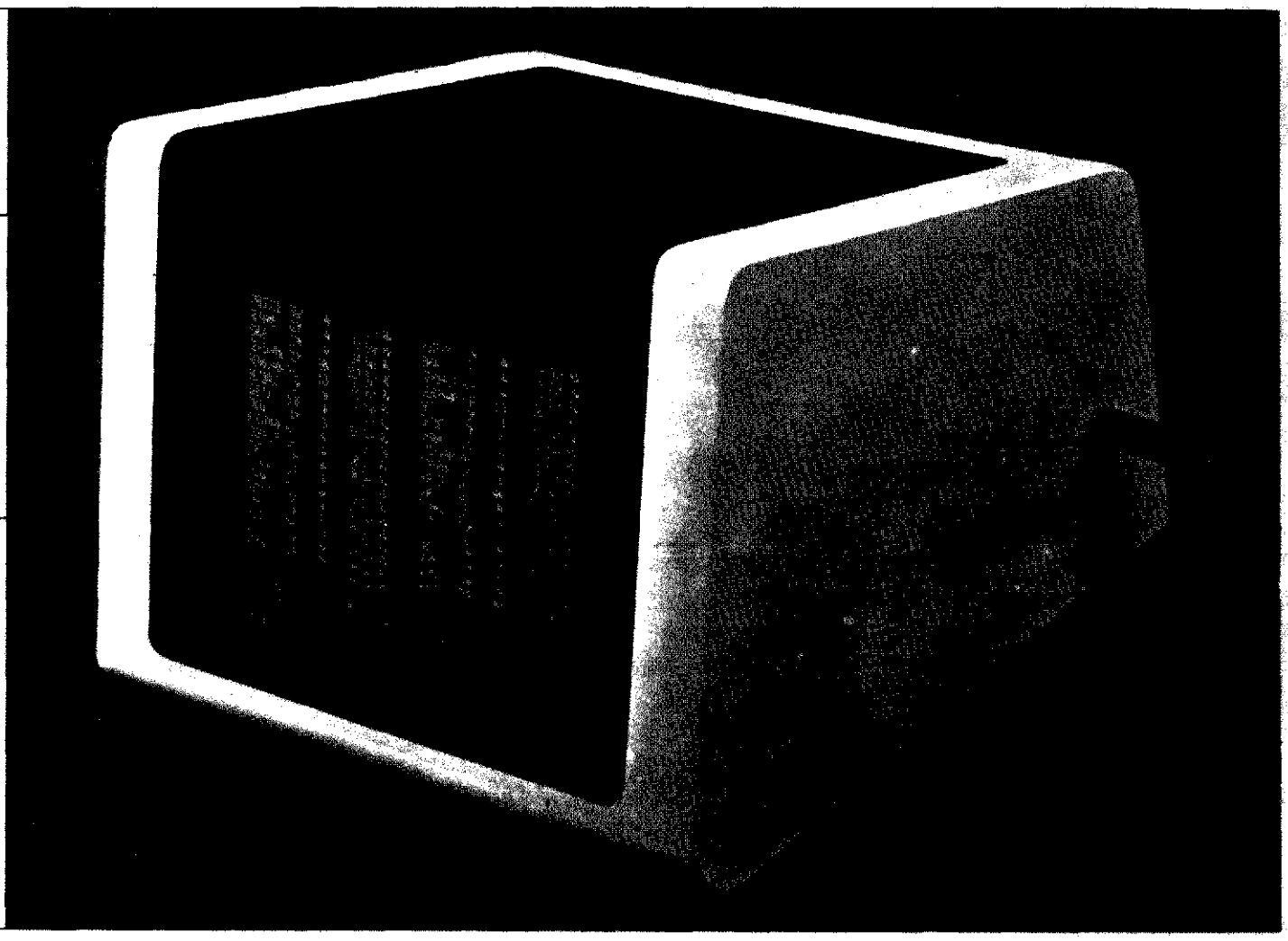


YU ISSN 0350-5596

SISTEM ZA ŠALTERSKO POSLOVANJE V BANKAH IN NA POŠTAH



računalniški sistemi delta

Sistem za šaltersko poslovanje je sodobna računalniška oprema za delo v bankah in na poštah, opremljen z ustrezno programsko opremo.
Sistem omogoča samostojno ažurno poslovanje – od posameznih operativnih del na šalterjih do zajema podatkov za nadaljnjo obdelavo. Deluje lahko povsem samostojno ali v povezavi z glavnim računalnikom (prenos informacij je mogoč prek stalno najetih ali navadnih telefonskih linij). Delovanje sistema tudi ni odvisno od razpoložljivosti računalniških kapacitet glavnega računalnika.
Sistem nadomešča raznovrstno opremo, ki se uporablja pri šalterskem poslovanju – od klasičnih mehanografskih strojev, pisalnih strojev do kalkulatorjev in deloma mikročitalnikov.

Sistem za šaltersko poslovanje je savremena računarska oprema za rad u bankama i poštama, opremljen sa odgovarajućom programskom opremom.
Sistem omogućava samostalno ažurno poslovanje – od pojedinih operativnih poslova na šalterima do zahvata podataka za dalju obradu. Može da radi sasvim samostalno, ili da komunicira sa glavnim računarom (prenos informacija je moguć preko stalno iznajmljenih ili običnih telefonskih linija). Rad sistema je takođe nezavisan od raspoloživosti računarskih kapaciteta glavnog računara.
Sistem zamenjuje raznovrstnu opremu, koja se upotrebljava u šalterskom poslovanju – od klasičnih mehanografskih mašina, pisaćih mašina do kalkulatora i delimično čitača mikrofiševa.

informatica

Časopis izdaja Slovensko društvo Informatika, 61000 Ljubljana, Parmova 41, Jugoslavija

LETNIK 10, 1986 – št. 2

Uredništvo in uprava:

Informatica, Parmova 41, 61000 Ljubljana, telefon (061) 312 988; teleks 31366 YU Delta.

Letna naročnina za delovne organizacije znaša 5900 din, za zasebne naročnike 1590 din, za študente 490 din; posamezna številka 2000 din.

Številka žiro računa: 50101-678-51841

Pri financiranju časopisa sodeluje Raziskovalna skupnost Slovenije

Na podlagi mnenja Republiškega komiteja za informiranje št. 23-85, z dne 29. 1. 1986, je časopis oproščen temeljnega davka od prometa proizvodov

Tisk: Tiskarna Kresija, Ljubljana

Grafična oprema: Rasto Kirn

V S E B I N A

informatica

Published by Informatika, Slovene Society for Informatics, Parmova 41, 61000 Ljubljana, Yugoslavia

VOLUME 10, 1986 — No. 2

Headquarters:

Informatica, Parmova 41, 61000 Ljubljana, Yugoslavia. Phone: 61 31 29 88. Telex: 31366 yu delta

Annual Subscription Rate: US$ 22 for companies, and US$ 10 for individuals

Opinions expressed in the contributions are not necessarily shared by the Editorial Board

Printed by: Tiskarna Kresija, Ljubljana

Design: Rasto Kirn

CONTENTS

OVERLAPPING: A PARADIGM OF PARALLEL AND SEQUENTIAL PROCESSING (*)

INFORMATICA 2/86

Anton P. Železnikar
Iskra Delta, Ljubljana

UDK: 681.3:519.712

Abstract. In this article a philosophy of the so-called overlapping approach basically derived from the notion of overlapping algorithm [1, 2] is presented. In the framework of this approach two basic constructs are defined: the overlapping (OAM) and the metaoverlapping abstract machine (MOAM). These constructs represent parallel-sequential problem solvers. Several formal definitions concerning the OAM and MOAM are given (23 definitions) and from these theorems are deduced (13 theorems). It is shown how an OAM implements a sequence of parallel actions (processes) and how by an MOAM the parallelism can be "increased." OAM's and MOAM's can be easily modeled by some well-known parallel architectures (hypercube, bus, switching system) and also by a parallel programming language.

Keywords. Abstract machine, action pattern, action pattern transition, action set, arbiter, grid processor, left state pattern, library of logical schemes, logical scheme, metaaction, metaoverlapping abstact machine, metaoverlapping rule, metastate, parallel processing, pattern domain, pattern shape, problem solver, process diagram, processor grid (multidimensional), processor space, processor subspace, overlapping, overlapping rule, overlapping rule base, overlapping signal, overlapping step, overlapping subrule, right state pattern, state set, state pattern, state pattern transition, subrule set, wait state.

1. Introduction

This article deals with the notion termed overlapping as a systematic (algorithmic) approach how to *master problems in a processor array* (grid) environment. Problems can probably be decomposed in sequences of parallel processes. Overlapping approach solves the problem decomposition in a specific, particular way and can put the capabilities of emerging VLSI technology into effective function. Overlapping approach can be easily adapted to be performed on various parallel architectures like hypercube, bus, systolic, shuttle network, four nearest neighbors, cross-bar switch, etc.

The overlapping approach basicly derived from overlapping algorithms [1, 2] shows some promise *for realizing massive parallelism* and it can be viewed as a method for designing special processor arrays. This approach can lead to an *algorithm design* and also to *programming methodology*. Parallel logic programming (parallel Prolog, guarded Horn clauses) can be used to program overlapping models efficiently. Under limited conditions overlapping models can be simulated by von-Neumann computers using high-level programming languages.

This article proposes a framework for realizing the potential, termed overlapping. Overlapping needs a *machine*, a *processor array where node* processors have some memories which can be down-loaded for particular functions. Several appropriate *commercially avaible* and *non-commercial* (universities, institutes) multiprocessor systems exist by which the overlapping approach could be *realized* [3].

In this phase of investigation overlapping is viewed as a hardware-oriented methodology for constructing special-purpose processors. It can also comprise an abstract machine, algorithm development methodology etc.

2. Philosophy of the Overlapping Approach

(1) There exists a grid (array, space) of processors which constitute an abstract (unlimited) processor space (multidimensional, in general).

(2) In the *processor space there is* a subspace (mechanism) dedicated for control purposes.

(3) There exists a problem (user, input problem) which will be solved by the activity in the processor space. Constructively, there is not clear yet in which way the problem will be solved by means of processor space.

(*) The manuscript of this article was prepared as author's lecture at the Hokkaido University, Sapporo, Japan, held on November 6, 1985.

(4) For the given problem, some kind of initial state pattern over the processor space is needed. This pattern will be generated for a given problem by a special, initial operator (see later, in the logical scheme).

(5) The term pattern has to be explained. Generally, a pattern is a non-complete array of entities which is selfgoverning, but it can be mapped also onto processor space. In some way, indices of array entities and indices of processors in a processor space do correspond. Entities of a pattern array can be directly mapped into processors (registers, memories).

(6) There will be several kinds of pattern entities as states and actions. In the processor space there will be state and action patterns, in overlapping subrules only state patterns. The so-called arbiter function will generate actions as consequences of state subsets causing action pattern in the processor space. Pattern entities will be symbols representing specific (problem solving) states and actions.

(7) A problem (3) will be solved constructively and the way of constructive solution will be called the overlapping approach. For this approach additional constructs are needed.

(8) In (5) state and action patterns were explained and in (1) the processor space was introduced. In (5) and (6) the correspondence of patterns to processor space was explained. By means of state and action patterns the overlapping process in the processor space will be conducted step by step.

(9) For a stepwise conduction of the overlapping process the notion of a particular rule (similar to rules in expert systems) is necessary; such a rule is called the overlapping rule. In the next paragraphs the mechanism of the overlapping rule will be developed.

(10) The overlapping rule (OR for short) will be defined part by part. An OR will consist of two parts: a non-empty subrule set and an arbiter.

(11) Subrule is a rule of the form

```
IF state_pattern THEN
 OVERLAP_BY state_pattern
```

Subrule is an overlapping rule; it is a preaction rule, where actions will be performed later after using the arbiter. The execution of a subrule represents an overlapping operation, described in the next paragraph.

(12) A subrule has a left state and a right state pattern (noncomplete array of states). By a subrule execution (usage) all of left state patterns included in a processor space will be searched and each of the found left state patterns of the subrule will be overlapped by the right state pattern of the subrule in the processor space (e.g. in processor memory). Here, overlapping is not a replacement. The states of the right state pattern will be stored (accumulated) in the memories of pattern corresponding processors. The result of a subrule execution (application) over the processor space is accumulation of some states in some processor memories of the processor space.

(13) In the same systematic way as application of a subrule was defined, an application of a subrule set over the processor space can be defined. For each subrule in a subrule set the procedure described in (12) has to be executed. By this, additional states are accumulated (stored) in some memories of the processor space.

(14) By overlapping (applying subrules) some processors in the grid have accumulated more than one state (including the original processor state before overlapping) and in these processors after terminating overlapping (13) an arbitration has to be performed to decide which action in the processors with more than one accumulated state will occur. For this purpose, to each subrule set an arbitration function is defined, called the arbiter. An arbiter will produce a single action in the processor having more than one accumulated state. This action (its symbol) will be stored and executed to produce a problem function, reset the accumulated states and generate a problem dependent new processor state. Actions being performed in a parallel way in different processors can communicate among each other; these are the well-known communication problems among parallel processes.

(15) Arbiter is a function being defined over a set of state subsets producing an action symbol belonging to the action set.

(16) A subrule set and an arbiter associated to this subrule set constitute a pair called the overlapping rule. Procedures described in (11) to (14) represent the so-called overlapping rule application.

(17) Overlapping rule application will be termed also overlapping step. A problem (3) will be solved by several overlapping steps.

(18) A problem solver has a base of overlapping rules (e.g. rules of knowledge etc.) and can use the processor grid for solving problems. In this case, the definition of a program for overlapping rules application (logical scheme) will be necessary. All overlapping rules needed to solve a problem by a particular sequence of overlapping rules are elements of the so-called rule base or briefly base.

(19) Let us define the so-called logical scheme for applying overlapping rules when solving a problem. Operators in the logical scheme are symbols (names) representing overlapping rules in the rule base. Rule symbol in the logical scheme represents the rule call, i.e., the application (execution) of the overlapping rule. In the logical scheme there is an initialization operator at its beginning, there are rule calls, if statements and labels (as described later);

(20) All left state patterns of subrules, belonging to a subrule set of an overlapping rule are travelling (moving) through the processor space. The travelling process is systematic and starts at some "beginning edge" of the processor space. The processors of the space are signalling when a pattern state is becoming equal to the processor original state. These signals are attributed by the position (coordinates, indices) of the signalling proces-

sor in the space. A monitor processor in the processor subspace signals the inclusion of a left state pattern in the processor state pattern and in this case the corresponding right state pattern is stored (accumulated) in the corresponding processors.

(21) Each subrule in the overlapping subrule set can have its own monitoring processor in the processor subspace, so, the overlapping through the processor space is carried out as fast as possible. When the overlapping process has reached the so-called "ending edge" of the processor space the rule arbiter function in the processors having more than one stored (accumulated) state is performed (executed), generating actions. By this, the overlapping step (applying of an overlapping rule) is terminated.

(22) A particular processor in the processor subspace (the so-called application register) is signalling if an overlapping rule was really applied. Application of an overlapping rule means that at least one overlapping of a left state pattern in the processor state pattern by right state pattern was performed. The notion of the rule application register is important at the further construction of the logical scheme.

(23) A logical scheme is a sequence of overlapping rule call statements, of if statements having the shape

```
IF overlapping_signal THEN
  GOTO label1;
ELSE
  GOTO label2;
ENDIF;
```

and labels. At the beginning of the scheme there is a problem dependent state initializing operator. The overlapping signal represents the state of the application register described in (22). By this, a logical scheme is a non-trivial sequence of elements with the syntax

```
[label] overlapping_rule_call
  to_rule_attributed_if_statement .
```

(24) Logical scheme is also a program for processor allocations to parallel actions in the processor space. A rule call in the scheme represents several inherent and problem solving operations being the following:

(a) searching for all left state patterns of the subrule set in the instantaneous processor state pattern;
(b) overlapping of each in the instantaneous processor state pattern occuring left state patterns of subrule set by the corresponding right state pattern;
(c) parallel arbitrating in processors having accumulated more than one state after overlapping by transforming these states to actions;
(d) parallel execution of problem solving actions from (c), reseting the accumulated states and determinig the new processor state.

(25) Actions resulting by arbitration in a rule call represent parallel problem solving processes or parallel subprograms and have to be problem (user) defined.

(26) Some remarks to the initial state pattern generation in the processor space are necessary. Before the abstact machine begins to apply overlapping rules according to logical scheme, the processor initial state pattern has to be generated otherwise the rules can not be applied. This pattern is put into the processor space by a special operator at the beginning of the logical scheme. This first operator can also clear the processor (state) space puting all the processors in the wait state 'w' and then initializing the proper processors with corresponding states.

(27) The state initialization operator is not an overlapping rule and in the logical scheme it is recognized as such. The purpose of the operator is to map a state pattern into processor space. The initialization operators will be denoted by init.

(28) The overlapping approach described in (1) to (27) is based on the following constructs:

(a) processor space (grid, array);
(b) processor subspace (control purposes);
(c) a user problem (state and action patterns are deduced for appropriate rules);
(d) overlapping rule (subrule set, arbiter);
(e) base of overlapping rules;
(f) logical scheme for overlapping rule application and initialization of processor space (logical scheme is a processor subspace driven program).

We say that a problem will be solved by the overlapping approach.

(29) Let us explain the overlapping approach for solving a problem as a serial-parallel processing. For this purpose, the processors in the processor space are named (numbered, indexed) and a process in a processor is denoted by a[i]; here, 'a' denotes a problem solving action as described previously and 'i' is the processor index. Actions a[i] can communicate among each other in the framework of an overlapping step. A logical scheme represents a sequence of parallel processes groups where parallel processes occur within the overlapping rule call. So, for example, we obtain the process diagram shown in Fig. 1 (numbers in the scheme

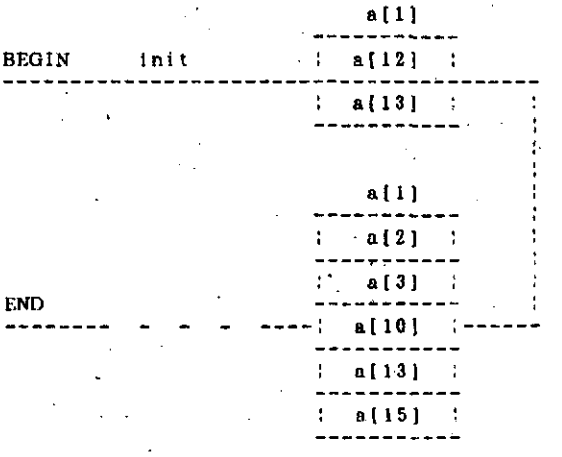


Fig. 1. Process (action) diagram with processor indices [i] being allocated

represent the processor indices in the processor grid).

Here, actions a[1], a[12], and a[13] can communicate among each other in the first overlapping step; in the second overlapping step such communication is possible among actions a[2], a[3], a[10], a[13], and a[15].

(30) Example: Adding two binary numbers by overlapping approach. Let a processor grid represented by processor wait states be given:

```
w w w w w w
w w w w w w
w w w w w w
w w w w w w
```

Let a processor subspace for control purposes be given. The set of states has three elements: w, 0, 1 (where 0 and 1 represent binary ciphers). The set of actions has three elements: w, m, r (where w is wait action, m is action for replacing the initial wait states w by 0 and 1 states and r is the so-called replacement, for which r(x,y) = y, where x and y are elements of the set {0, 1, w}. Further, we have a subrule set SRS with four subrules

left_state_pattern ___ right_state_pattern,

where we simply denote them by

```
  0  ---  1        1  ---  0
  1       0     0  1     1  0

  w  ---  1        1  ---  0
  1  0    0  0     w  1     1  0
```

The arbiter is action 'r' being described previously. By this, overlapping addition rule is determined. The initial state pattern is

```
0 1 0 1
1 1 1 1
```

representing two binary numbers (number in a line). This pattern will be mapped by the init operator 'm' onto the processor grid. In this case, logical scheme is the following program:

```
        BEGIN
        init;
beg:    addition_rule;
        IF overlapping_signal THEN
          GOTO beg;
        ELSE GOTO end;
end:    ENDIF;
        END;
```

Let us denote the wait state w by a dot (.) for better lucidity. We get the following state transitions in the processor space:

```
. . . . . .        . . . . . .
. . . . . .   s    . 0 1 0 1 .   s
. . . . . .  |--   . 1 1 1 1 .  |--
. . . . . .        . . . . . .

. . . . . .        . . . . . .
. 1 1 1 1 .   s    . 1 0 1 0 .   s
. 0 1 0 1 .  |--   . 1 0 1 0 . -|--
. . . . . .        . . . . . .
```

```
. . . . . .        . . . . . .
. 0 0 0 0 .   s    1 0 1 0 0 .
1 0 1 0 0 .  |--   0 0 0 0 0 .  --
. . . . . .        . . . . . .
```

c

Here |-- denotes a state pattern transition corresponding to an overlapping step. If addition rule could be applied the overlapping signal is true, otherwise false. Sign --| denotes that termination of addition process is state pattern driven. At the end, the sum appears in the second line of the processor space. In a similar way we can get the action transitions in the processor space:

```
. . . . . .        . . . . . .
. . . . . .   a    . m m m m .   a
. . . . . .  |--   . m m m m .  |--
. . . . . .        . . . . . .

. . . . . .        . . . . . .
. r . r . .   a    . . r . r .   a
. r . r . .  |--   . r r r r .  |--
. . . . . .        . . . . . .

. . . . . .        . . . . . .
. r . r . .   a    r . r . . .
r r r r . .  |--   r r r . . .  --|
. . . . . .        . . . . . .
```

a

Here, |-- denotes an action pattern transition corresponding to an overlapping step. Actions appearing in an action pattern are parallel. Wait (dot) actions are actually nonactions, and m is the initial mapping action. One can draw the process diagram shown in Fig. 2.

3. Formal Definition of the Overlapping Abstract Machine

3.1. A Top-down Introduction

Now, we have to clear the construction and the properties of the overlapping abstract machine in a systematic and formal way. In the previous chapter we have intuitively described the overlapping approach and it became more and more clear what the construction and the properties or functions of an adequate abstract machine should be. Let us begin with top-down definition of the abstract machine. Let us stress that some algorithms realized by the abstract machine will remain on the intuitive level: namely, some basic optimized machine functions have not been examined yet (e.g. as programs expressed in an appropriate programming language).

D e f i n i t i o n 0. The overlapping abstract machine (OAM for short) is a quintuple

OAM = (P, S, A, B, L),

where:

P is a multidimensional (linear, quadratic, cubic etc.) processor grid (space, array) with a processor subgrid for control purposes (for executing logical scheme, signalling, communications etc.); this grid is potentially unlimited; each processor has a local

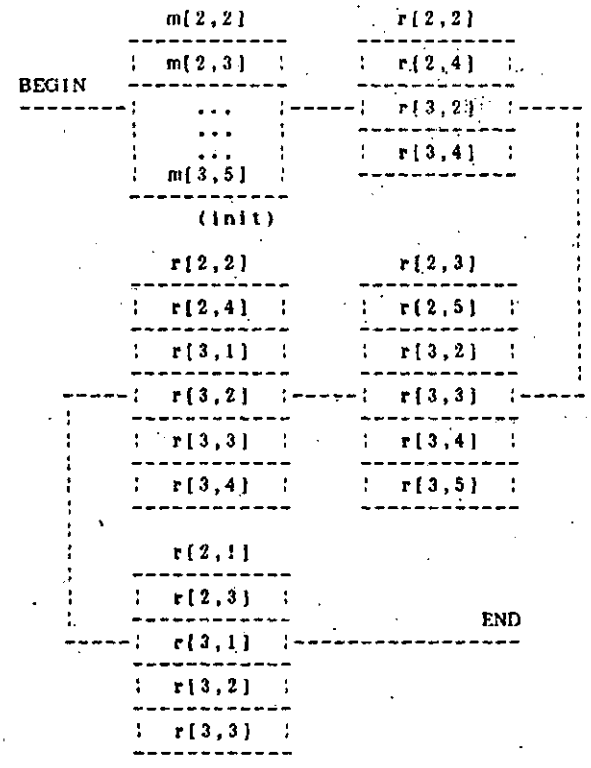


Fig. 2. Parallel-sequential process diagram resulting from the example of adding two binary numbers

memory and can execute actions (problem solving subprograms); processors in the grid are well-connected in some respect so they can ensure basic machine (inherent) operations; for instance, processors are able to search left state patterns of a rule throughout the processor space; they are able to perform overlapping of states in each particular processor; they are able to execute arbitration in each particular processor (i.e. determining actions in dependence of the overlapped states);

S is a state set; the set cardinality has to consider sufficient number of distinguished states to solve a problem; states are symbols of the so-called state type; this set is potentially unlimited;

A is an action set; this set represents an ordered library of action subprograms dedicated to perform partial functions needed in the processes of problem solving; set A is potentially unlimited;

B is an overlapping rule base; overlapping rules can be expressed as particular information type structures adequate for storing and usage; base B is potentially unlimited;

L is a library of logical schemes; a scheme calls overlapping rules when solving a particular problem; a logical scheme conducts a problem solution by rule calling; library L is potentially unlimited.

Sets S and A, base B, and library L are generative sets. During their life cycles these sets will be modified; the cardinality of these sets will be variable; set elements will be added and deprivated in dependence of problem environment requirements. OAM is obviously a problem solver with learning capabilities improving its solving power through its life cycle.

We say that an OAM is overlapping well-connected; it performs all necessary overlapping operations automatically. In general, an OAM is a serial-parallel inference machine (SPIM). In a particular case, when an adequate solution exists, an OAM can function as a (pure) parallel inference machine (PIM).

There is nothing more substantial to say about sets S and A. Subprograms in A can be expressed in a proper programming language.

Rules in the overlapping rule base can be formalized. Each overlapping rule consists of two elements: a subrule set and an arbiter. This construct can be written for an overlapping rule OR as

$$OR = (\{SR[1], SR[2], \ldots, SR[n]\}, arb)$$

and further, a subrule SR can be formalized as

$$SR = (left\_state\_pattern, right\_state\_pattern)$$

Here, patterns are adequate type structured, so correspondence between left_state_pattern and right_state_pattern is unambigiously defined.

A logical scheme begins always with the state initialization operator and continues with overlapping rule calls. A rule call executes some to processor grid inherent operations and finally also defined problem solving actions (as an action pattern). Processes (actions) of a rule call are parallel (they can certainly communicate among each other). Inherent operations executed by a rule call have been listed at the description of P (searching, overlapping, arbitrating). After this operations problem solving actions will be performed. The connectivness of B ensures that inherent operations are as efficient as possible (parallel). Optimized algorithms for efficient inherent operations have to be elaborated (developed).

3.2. Formal Definitions

3.2.1. State Patterns and Some Relations Among Them

Definition 1. Let M(k) be a finite non-empty subset of a Cartesian product I(k) of k factors, where I is the set of all integers, and let S be a finite non-empty set of states. A mapping sp(k) (sp denotes a state pattern) of a set M(k) into a set S, i.e.,

$$sp(k) = \{(x, y) \mid x \text{ EL } M(k) \;\&\; y = sp(k, x) \text{ EL } S\}, \quad k = 1, 2, \ldots,$$

where 'EL' denotes the relation 'being element of', is said to be the k-dimensional state pattern in S or briefly a pattern. Here, sp(k, x) represents sp(k)(x), i.e., the value of function sp(k) at its argument x.

By w EL S we denote an empty entity, and each pair (x, w) in sp(k) can be omitted. An empty pattern is a set

$$\{(x, w) \mid x \text{ EL } M(k)\}$$

Definition 2. If sp(k) is a pattern and

$$sp(a, k) = \{(x+a, y) \mid (x, y) \text{ EL } sp(k) \;\&\; x+a = (x(1)+a(1), \ldots, x(k)+a(k))\},$$

where a EL I(k) and k = 1, 2, ..., then sp(a, k) is the so-called a-displacement of the pattern sp(k).

Definition 3. Two state patterns sp(k) and sq(k) are said to be equivalent if

$$EX\ a(a\ EL\ I(k)\ \&\ sq(k) = sp(a, k)),$$

where 'EX' denotes 'there exists'. Let this case be denoted by sp(k) .= sp(a, k) where '.=' is the relation of state pattern equivalence.

Obviously, each a-displacement of a pattern is equivalent to this pattern.

Definition 4. A pattern sp(k) is said to be included in a pattern sq(k) if

$$EX\ a(a\ EL\ I(k)\ \&\ sp(a, k)\ INCL\ sq(k)),$$

where 'INCL' represents the relation of inclusion. Let this case be represented by sp(k) pattern inclusion. Evidently,

$$sp(k)\ .=\ sq(k) \iff EX\ a(sp(a, k)\ .INCL\ sq(k)) \ \&\ EX\ b(sp(k)\ .INCL\ sq(b, k))$$

where '<=>' is the logical equivalence symbol, is true.

3.2.2. Pattern Domains

Definition 5. Let * NOT_EL S, where 'NOT_EL' denotes 'being not element of'; then the pattern

$$r(*, k) = \{(x, z) : x\ EL\ M(k)\ \&\ z = *\},\quad k = 1, 2, \ldots,$$

will be called the k-dimensional pattern shape or briefly the shape. Thus, each shape is a pattern in {*} or in {w, *}. Let a pattern sp(k) in S and let the set

$$f(*) = \{(y, z) : (y\ EL\ S \setminus \{w\}\ \&\ z = *)\ V\ (y = w\ \&\ z = \ )\},$$

where 'V' represents the logical 'or' operation, be given; then the set composition f(*) COMP sp(k), i.e.,

$$sp(*, k) = \{(x, z) : EX\ y(sp(k, x) = y\ \&\ f(*, y) = z)\}$$

will be called the shape of a given pattern sp(k). A pattern sp(k) has the shape r(*, k) if and only if sp(*, k) .= r(*, k). Two patterns sp(k) and sq(k) have the same shape if and only if

$$sp(*, k)\ .=\ sq(*, k)$$

Definition 6. Let B(*, k) be a finite non-empty set of k-dimensional pattern shapes. The domain D(k, OAM) of an OAM will be called a set of k-dimensional patterns in S, i.e.,

$$D(k, OAM) = \{sp(k) : sp(k)\ IN\ S\ \&\ sp(*, a, k)\ EL\ B(*, k)\ \&\ a\ EL\ I(k)\},$$

where sp(*, a, k) is an a-displacement of the shape sp(*, k); here, 'IN' is the relation 'being in'. Thus the domain D(k, OAM) will be generated by a pair (S, B(*, k)).

3.2.3. Overlapping Rule

Definition 7. Overlapping subrule, or briefly, subrule SR, will be a pair (sp(k), sq(k)) of k-dimensional state patterns in S, where S is the set of states and where, generally, sp(k) and sq(k) do not have equivalent shapes. The pattern sp(k) is said to be the left and the pattern sq(k) the right state pattern of SR. Two subrules SR_1 = (sp_1(k), sq_1(k)) and SR_2 = (sp_2(k), sq_2(k)) are equivalent if and only if

$$EX\ a(a\ EL\ I(k)\ \&\ sp\_1(k) = sp\_2(a, k)\ \&\ sq\_1(k) = sq\_2(a, k))$$

Definition 8. A finite non-empty set SRS of pairwise non-equivalent subrules SR[i] (i = 1, 2, ... , m), i.e.,

$$SRS = \{SR[1], SR[2], \ldots, SR[m]\},$$

will be called the subrule set. The application of a given subrule set to a state pattern sp(k) is defined as follows. Let it be SR[i] EL SRS, where SR[i] = (left[i](k), right[i](k)) and let sp(k) be a state pattern. To each SR[i] and sp(k) a set

$$SP[i](k) = \{(x, y) : (x, y)\ EL\ right[i](a, k)\ \&\ left[i](a, k)\ INCL\ sp(k)\ \&\ a\ EL\ I(k)\}$$

can be constructed. Hence, SP[i](k) includes pairs (x, y) with equal elements x and can also be an empty set. Now we construct to a given SRS = {SR[1], ... , SR[m]} and to a state pattern sp(k) a set

$$SV = sp(k)\ \cup\ \bigcup_{i=1}^{i=m} SP[i](k),$$

where 'U' represents the union operation and introduce the equivalence relation 'EQ' defined in SV by

$$(x1, y1)\ EQ\ (x2, y2) \iff x1 = x2$$

To the equivalence class EQ((x, y)) now belong all the elements of SV which are in the relation EQ with (x, y), i.e.,

$$EQ[(x, y)] = \{(x1, y1) : (x1, y1)\ EL\ SV\ \&\ x1 = x\}$$

The corresponding factor set SV/EQ of SV connected with EQ is

$$SV/EQ = \{X : EX\ (x, y)((x, y)\ EL\ SV\ \&\ X = EQ[(x, y)])\}$$

Now the overlapping value of SV will be the set

$$OSV = \{(x, C) : (x, y)\ EL\ X\ \&\ C = \{y : (x, y)\ EL\ X\}\ \&\ X\ EL\ SV/EQ\}$$

Obviosly, this set is a mapping of the set M(k) into the power set of the set S. We say that a given state pattern is transformed by applying a subrule set SRS to the overlapping value OSV, and we denote it by

$$SRS(sp(k)) = OSV$$

From OSV, which is a state subset pattern (C EL power_set(S)),we get its shape

$$OSV(*) = \{(x, *) : EX\ C(OSV(x) = C\}$$

OSV(*) will be also the shape of the action pattern.

Definition 9. The set

$$arb = \{(C, (x, y)) : C\ EL\ power\_set(S)\ \&\ y\ EL\ A\ \&\ x \cdot y()\ EL\ S\ \&\ (x, y)\ \ arb(C)\},$$

where A is an action set, is called the overlapping arbiter or briefly arbiter. The arbiter is said to be generally defined in a state set S if arb(zero) = (w, w) (zero denotes an empty set), arb((z)) = (z, w) where y EL S, and

w EL C & arb = arb(C) <=> arb = arb(C\(w))

where C EL power_set(S) and w EL S. In the arbiter definition, it is clearly understood according to paragraph (14) in the previous section, that the new state x EL S of a processor in the processor grid is produced by action y, so, x = y(). We suppose that problem dependent action y is executed within the arbiter function. Arbiter function actually replaces the set C by the new state x.

R e m a r k 1. Def. 9 conforms only one of possibilities for the arbiter determination. The arbiter function can be defined in several ways. So, generally, one can have

arb = arb(x[1], x[2], ... , x[m]),
m = 2, 3, ...

by which the possibility of defining arb(x, x) etc. is achieved. Further, arb can be determined by some kind of automaton calculating the new state always when the second state by overlapping was accumulated. This approach reduces the number of memory state locations in local memories to 2.

D e f i n i t i o n 1 0. A pair (SRS, arb), where SRS is an overlapping subrule set and arb an overlapping arbiter, is called the overlapping rule OR or briefly the rule. The application of a rule OR = (SRS, arb) to a state pattern sp(k) is determined by the transformation SRS(sp(k)) = OSV and the composition arb COMP OSV = spl(k) when OR maps a state pattern sp(k) into a new state pattern sp[1](k). We define

OR: sp(k) :-- sp[1](k) <=>
sp[1](k) = arb COMP SRS(sp(k))

A procedure OP: sp(k) :-- sp[1](k) is called the overlaping step and is performed by an overlapping rule call 'or' in the logical scheme.

D e f i n i t i o n 1 1. Different overlapping rules conform the so-called base of overlapping rules, briefly B; this base is adequatly structured. So, B is a structured set of OR's.

3.2.4. Logical Scheme

According to paragraphs (18) to (27) in Chapter 2 we have the following:

D e f i n i t i o n 1 2. Logical scheme LS is a program (algorithm)

LS = LS(init, or[1], ... , or[r],
o_sig[1], ... , o_sig[r]),

where init is the state pattern initialization operator, or[i] (i = 1, ... , r) are overlapping rule calls for rules belonging to rule base B, and o_sig[i] are overlapping signals as described in paragraphs (22) and (23) in Section 2. Schematically, for an LS we have the following segments:

init; seg[1]; seg[1]; ... ; seg[r];

where for seg[i] (i = 1, ... , r) there is

```
possible_label[i]:
        or[i];
        IF o_sig[i] THEN
          (null ; (GOTO possible_label[p]));
        ELSE
          (null ; (GOTO possible_label[q]));
        ENDIF;
```

If a segment is at the end of LS or if for a segment seg[j] we have

```
possible_label[j]:
        or[j]
        IF o_sig[j] THEN
          GOTO end_of_LS;
        ELSE
          (null ; (GOTO possible_label[q]));
        ENDIF;
```

then this segment is called the terminal segment of an LS. In the upper two segments of LS '(' and ')' are metaparentheses, null is a null statement, ';' is the alternative sign, and p, q = 1, ... , r.

There exists a library L of logical schemes of type LS within the OAM; this library is adequately structured.

By Def. 1 to Def. 12 the construct of overlapping abstact machine in Def. 0 is completely determined.

4. Basic Theorems for the Overlapping Abstract Machine Application

D e f i n i t i o n 1 3. The application of an OAM = (P, S, A, B, L) is defined as follows. For the given processor grid P a logical scheme LS from library L is selected. This scheme represents processes needed for a problem solution. By initializing operator init of the scheme the processor grid is state initialized. Further, logical scheme calls the overlapping rules belonging to the base B for execution and controls by means of overlapping signals the further overlapping rule calls. By arbitration within the overlapping rules problem solving actions belonging to A are executed. At the end of the logical scheme the OAM problem solving process is stoped and OAM is ready to solve another problem.

D e f i n i t i o n 1 4. First, let us define the value of a logical scheme within an OAM application. The value of a logical scheme in an OAM application is the sequence of overlapping signals which in the problem solving process have became the true value. This sequence represents the actual application of corresponding overlapping rules when some left pattern of a subrule of an overlapping rule was included in the processor state pattern. Now, we have the following instances of logical scheme values:

(1) The value of an LS is a finite (possibly empty) sequence of true overlapping signals where the end of LS was attained.
(2) The value of an LS is an infinite sequence of true overlapping signals where the end of LS cannot be attained.
(3) The value of an LS is a finite (possibly empty) sequence of true overlapping signals where the end of LS cannot be attained.

We say that an application of an OAM is resultative in case (1), and non-resultative in cases (2) and (3).

D e f i n i t i o n 1 5. At the beginning of an OAM = (P, S, A, B, L) application all the

grid processors are in the wait state; this state pattern is called the wait pattern. After execution of the initializing operator in the selected logical scheme LS the initial pattern is mapped into the grid state pattern. If execution of the LS has attained its end and the value of LS is empty (no overlapping rule could be applied), we write

$$OAM: isp(k) \ \overline{\overline{\dagger}}$$

where isp(k) is the initial state pattern. In this case no problem solving action from action set A took place. The initial state pattern remained unchanged until the end of OAM application.

Let us apply OAM = (P, S, A, B, L) for a selected LS EL-L and let the LS value be

$$o\_sig[i[1]],\ o\_sig[i[2]],\ \dots\ ,\ o\_sig[i[j]],$$

when the end of LS was not attained yet. Obviously, in this case we have

$$OR[i[1]]: sp[0](k) \ \vdash\!\!- \ sp[1](k),$$
$$OR[i[2]]: sp[1](k) \ \vdash\!\!- \ sp[2](k),$$
$$\dots\ ,$$
$$OR[i[j]]: sp[j-1](k) \ \vdash\!\!- \ sp[j](k),$$

(here, sp(k) is equal to sp[0](k)) and in the sense of Def. 10,

$$sp[1](k) = arb[i[1]]\ COMP\ SRS[i[1]](sp[0](k)),$$
$$sp[2](k) = arb[i[2]]\ COMP\ SRS[i[2]](sp[1](k)),$$
$$\dots\ ,$$
$$sp[j](k) = arb[i[j]]\ COMP\ SRS[i[j]](sp[j-1](k)$$

Evidently,

$$sp[j](k) = arb[i[j]]\ COMP\ SRS[i[j]]\ (\dots\ (arb[i[2]]\ COMP\ SRS[i[2]](arb[i[1]]\ COMP\ SRS[i[1]](sp[0](k))\ \dots\ ).$$

Now, let $\vdash\!\!-[i[j]]$ denote the overlapping step corresponding to the overlapping rule OR[i[j]], and let $\vdash\!\!=[j]$ denote j single sequential overlapping steps. So, the overlapping chain, or briefly the chain, will be introduced by

$$OAM: sp[0](k) \vdash\!\!-[i[1]]\ sp[1](k) \vdash\!\!-[i[2]]\ \dots \vdash\!\!-[i[j]]\ sp[j](k)$$
$$\Leftrightarrow \quad \&_{m=1}^{m=j}\ [sp[m](k) = arb[i[m]]\ COMP\ SRS[i[m]](sp[m-1](k))],$$

where sp[0](k) = sp(k), and sp[j](k) is called the simple image of sp(k) mapped by OAM. Instead of the upper chain we denote briefly

$$OAM(LS): sp[0](k) \vdash\!\!= sp[j](k) \quad \text{or also}$$
$$OAM: \quad sp[0](k) \vdash\!\!=[j]\ sp[j](k)$$

We read it: For the selected LS the OAM translates the initial state pattern sp[0](k) by j overlapping steps freely into the state pattern sp[j](k).

Definition 16. Let over a processor grid state pattern sp[n-1](k) a terminal segment seg[i[n]] of LS (Def. 12) be applied, where o_sig[i[n]] is true. Here, overlapping rule OR[i[n]] is applied to sp[n-1](k), mapping it into state pattern sp[n](k) when the execution of LS was terminated. We define

$$OR[i[n]]: sp[n-1](k) \vdash\!\!- . sp[n](k) \ \Leftrightarrow$$
$$arb[i[n]]\ COMP\ SRS[i[n]](sp[n-1](k)\ \&$$
$$seg[i[n]]\ \text{is terminal}$$

Let OAM be applied to sp(k) and let the value of an LS be a sequence o_sig[i[1]], o_sig[i[2]], a terminal LS segment. In this case

$$OAM: sp(k) \vdash\!\!=[n-1]\ sp[n-1](k)\ \&$$
$$OR[i[n]]: sp[n-1](k) \vdash\!\!- . sp[n](k)$$

is valid, and this is denoted by

$$OAM: sp(k) \vdash\!\!=[n-1]\ sp[n-1](k) \vdash\!\!- . sp[n](k)$$

For brevity, we write this as

$$OAM: sp(k) \vdash\!\!= . sp[n](k) \quad \text{or}$$
$$OAM: sp(k) \vdash\!\!=[n] . sp[n](k);$$

sp[n](k) is called a terminal image of sp(k) mapped by OAM in n overlapping steps.

Definition 17. Let over a processor grid state pattern sp[n-1](k) a non-terminal segment seg[i[n]] of LS be applied, where o_sig[i[n]] is true. Now, OR[i[n]] will be applied to sp[n-1](k), mapping it into sp[n](k); afterward, let the end of LS be atained without applying an OR of LS to sp[n](k). In this case we write

$$OR[i[n]]: \ sp[n-1](k) \ \vdash\!\!- \ sp[n](k) \ \overline{\overline{\dagger}}.$$

If o_sig[i[1]], o_sig[i[2]], ... , o_sig[i[n]] is the LS value, and if OAM was applied to sp(k), then

$$OAM: sp(k) \vdash\!\!=[n-1]\ sp[n-1](k)\ \&$$
$$OR[i[n]]: sp[n-1](k) \vdash\!\!- sp[n](k) \ \overline{\overline{\dagger}}$$

This is denoted by

$$OAM: sp(k) \vdash\!\!=[n-1]\ sp[n-1](k) \vdash\!\!- sp[n](k) \ \overline{\overline{\dagger}},$$

or briefly by

$$OAM: sp(k) \vdash\!\!=[n]\ sp[n](k) \ \overline{\overline{\dagger}} \quad \text{or}$$
$$OAM: sp(k) \vdash\!\!= sp[n](k) \ \overline{\overline{\dagger}}$$

The resulting pattern sp[n](k) is called the exact image of sp(k) mapped by OAM in n overlapping steps.

Definition 18. If the application of an OAM to a processor grid pattern sp(k) is resultative and sp[n](k) is a result, we denote it by OAM(sp(k)) = sp[n](k). In this case, in the sense of Def. 14 (1), there exists only one of the possibilities:

$$OAM: sp(k) \ \dagger \ \text{with} \ \overline{OAM}(sp(k)) = sp(k),$$
$$OAM: sp(k) \vdash\!\!= . sp[n](k), \ \text{and}$$
$$OAM: sp(k) \vdash\!\!= sp[n](k) \ \overline{\overline{\dagger}},$$

where sp(k) is the initial state pattern. We introduce the following notation:

$$\text{if} \ \ OAM: sp(k) \vdash\!\!- sp[1](k), \quad \text{then} \ \ OAM: sp(k) \vdash\!\!= sp[1](k);$$
$$\text{if} \ \ OAM: sp(k) \vdash\!\!- . sp[1](k), \quad \text{then} \ \ OAM: sp(k) \vdash\!\!= . sp[1](k);$$
$$\text{if} \ \ OAM: sp(k) \vdash\!\!- sp[1](k) \ \overline{\overline{\dagger}}, \quad \text{then} \ \ OAM: sp(k) \vdash\!\!= sp[1](k) \ \overline{\overline{\dagger}}.$$

THEOREM 1. If OAM = (P, S, A, B, L) is an overlapping abstract machine and sp(k) EL D(k, OAM) an initial state pattern determined by the operator init of the selected LS EL L, then only one of the following possibilities is valid:

(1) $OAM: sp(k) \ \overline{\overline{\dagger}}$;

(2) OAM: sp(k) |-- sp[1](k);
(3) OAM: sp(k) |-- . sp[1](k);

(4) OAM: sp(k) |-- sp[1](k) ¯|;
(5) the value of LS EL L is empty and the end of LS cannot be attained

P r o o f. In the sense of Def. 14, the application of an OAM to a state pattern can be either resultative or not. If the application of an OAM to sp(k) is resultative, only one of the possibilities (1) - (4) is true. This means that the first overlapping step either exists (in cases (1) - (4)) or not (in case (1)). Evidently, in the resultative case no other possibility exists. If the application of OAM to sp(k) is non-resultative, only one of the possibilities (2) and (5) is valid. This means that the first overlapping step either exists (2) or not (5). Evidently, in the non-resultative case, there does not exist any other possibility. Thus Theorem 1 is proved.

T H E O R E M 2. If OAM = (P, S, A, B, L) and for initial state pattern sp(k) for selected LS there is sp(k) EL D(k, OAM), one of the following equivalences is valid:

```
OAM: sp(k) |==[j] sp[j](k)   &  j > 1  <=>
EX! sp[j-1](k)(OAM: sp(k) |==[j-1] sp[j-1](k)
                          |-- sp[j](k));

OAM: sp(k) d==[n] . sp[n](k) &  n > 1  <=>
EX! sp[n-1](k)(OAM: sp(k) |==[n-1] sp[n-1](k)
                          |-- . sp[n](k));


OAM: sp(k) |==[n] sp[n](k) ¯| & n > 1  <=>
EX! sp[n-1](k)(OAM: sp(k) |==[n-1] sp[n-1](k)

                          |-- sp[n](k) ¯|),
```

where 'EX!' means 'there exists exatly one'.

P r o o f. Obviosly, the validity of equivalences in Theorem 2 follows from Defs. 13, 15, 17, and 18.

T H E O R E M 3. If OAM = (P, S, A, B, L) and for initial state pattern sp(k) for selected LS there is sp(k) EL D(k, OAM), the following implications are valid:

```
OAM: sp(k) |== sp[i](k) |== sp[i+j](k)   ==>
                    OAM: sp(k) |== sp[i+j](k);

OAM: sp(k) |== sp[j](k) |== . sp[n](k)   ==>
                    OAM: sp(k) |== . sp[n](k);

OAM: sp(k) |== sp[j](k) |== sp[n](k) ¯|  ==>

                    OAM: sp(k) |== sp[n](k) ¯|.
```

P r o o f. These implications follow directly from Defs. 15, 16, and 17.

T H E O R E M 4. If OAM = (P, S, A, B, L) and for initial state pattern sp(k) for selected LS there is sp(k) EL D(k, OAM), the following implications are valid:

```
OAM: sp(k) |-- sp[i](k) &
OAM: sp(k) |==[j] sp[j](k) & j > 1  ==>
     OAM: sp(k) |-- sp[i](k) |==[j-1] sp[j](k);

OAM: sp(k) |-- sp[i](k) &
OAM: sp(k) |==[n] . sp[n](k)          ==>
     n > 1 & OAM: sp(k) |-- sp[1](k) |==[n-1]
     . sp[n](k);

OAM: sp(k) |-- sp[1](k) &

OAM: sp(k) |==[n] sp[n](k) ¯|         ==>
     n > 1 & OAM: sp(k) |-- sp[1](k) |==[n-1]
     sp[n](k) ¯|
```

P r o o f. These implications follow at once from Theorems 1 and 3.

T H E O R E M 5. If OAM = (P, S, A, B, L) and for initial state pattern sp(k) for selected LS there is sp(k) EL D(k, OAM), the following implication can be deduced from Def. 18, Theorem 2, and Defs. 15, 16, and 17:

```
OAM: sp(k) |== sp[j](k) &
OAM(sp(k)) = sp[n](k)      ==>
  (OAM: sp(k) |== sp[j](k) |== . sp[n](k) <=/=>

   OAM: sp(k) |== sp[j](k) |== sp[n](k) ¯|),
```

where '<=/=>' is the sign of logical non-equivalence.

5. Primitive Overlapping Abstract Machine (POAM)

In this section we shall deal with a particular case of OAM having a special type of the logical scheme. This LS is without an initialization operator and the initial state pattern can be mapped into the processor grid state pattern by a separate operator before the POAM application.

D e f i n i t i o n 19. The primitive overlapping abstract machine POAM = (P, S, A, B, PL) is an OAM, where P, S, A, and B are elements of OAM, and PL is a subset of L. Primitive library PL is an adequately structured set of the so-called primitive logical schemes (PLS). A PLS has the following properties:

(1) In PLS there is not an initial operator init (like in LS); operator init in PLS is an empty (null) operator.

(2) Segments seg[i] (i = 1, ... , r) of a primitive logical scheme have the following general form:

```
possible_label[i]:
    or[i];
    IF o_sig[i] THEN
      GOTO (beginning_of_NLS ¦ end_of_NLS);
    ELSE
      (null ¦ (GOTO possible_label[q]));
    ENDIF;
```

where '(' and ')' are metaparentheses, '¦' is an alternative sign, null is a null statement, and q = 1, ... , r.

(3) Obviosly, after each application of an overlapping rule or[i] the process is continued at the beginning of the PLS if it is not terminated (continued at the end of the PLS).

T H E O R E M 6. If POAM = (P, S, A, B, PL) and for an initial state pattern there is sp[0](k) EL D(k, POAM), the following equivalences hold:

```
POAM: sp[0](k) |--[i[1]] sp[1](k) |--[i[2]] ...
               |--[i[j]] sp[j](k)

      m=j
<=>    &  POAM: sp[m-1](k) |--[i[m]] sp[m](k);
      m=1

POAM: sp[0](k) |--[i[1]] sp[1](k) |--[i[2]] ...
               |--[i[n]] . sp[n](k)
```

```
                n-1
<=>              &  POAM: sp[m-1](k) |--[i[m]] sp[m](k)
                m=1
          &    POAM: sp[n-1](k) |--[i[n]] . sp[n](k);

POAM: sp[0](k) |--[i[1]] sp[1](k) |--[i[2]] ...

                    |--[i[n]] sp[n](k) ¯|

                n-1
<=>              &  POAM: sp[m-1](k) |--[i[m]] sp[m](k)
                m=1
          &    POAM: sp[n-1](k) |--[i[n]] sp[n](k) ¯|
```

P r o o f. The validity of Theorem 6 follows from Def. 19, and from the previous definitions and theorems.

T H E O R E M 7. If POAM = (P, S, A, B, PL) and sp(k) EL D(k, POAM), then

```
POAM: sp(k) |== sp[n](k) ¯|  ==>

POAM: sp[n](k) ¯|.
```

P r o o f. Evidently, the property of a POAM described in Theorem 7 can be deduced from Def. 19 and Theorem 6.

T H E O R E M 8. If POAM = (P, S, A, B, PL) and sp(k) EL D(k, POAM), then

```
POAM: sp(k) |==[j] sp[j](k)  &  j > 1        <=>
EX! sp[j-1](k)(POAM: sp(k) |==[j-1] sp[j-1](k)
          & POAM: sp[j-1](k) |-- sp[j](k));

POAM: sp(k) |==[n] . sp[n](k)  &  n > 1      <=>
EX! sp[n-1](k)(POAM: sp(k) |==[n-1] sp[n-1](k)
          & POAM: sp[n-1](k) |-- . sp[n](k));

POAM: sp(k) |==[n] sp[n](k) ¯|  &  n > 1     <=>
EX! sp[n-1](k)(POAM: sp(k) |==[n-1] sp[n]-1(k)

          & POAM: sp[n-1](k) |-- sp[n](k) ¯|).
```

The proof is similar to that of Theorem 2.

The methods of proof of Theorems 3, 4, and 5 yield, in order, the following three theorems.

T H E O R E M 9. If POAM = (P, S, A, B, PL) and sp(k) EL D(k, POAM), then

```
POAM: sp(k) |== sp[i](k) |== sp[i+j](k)    <=>
     POAM: sp(k) |== sp[i](k)  &
     POAM: sp[i](k) |== sp[i+j](k);

POAM: sp(k) |== sp[i](k) |== . sp[n](k)    <=>
     POAM: sp(k) |== sp[i](k)  &
     POAM: sp[i](k) |== . sp[n](k);

POAM: sp(k) |== sp[i](k) |== sp[n](k) ¯|   <=>
     POAM: sp(k) |== sp[i](k)  &

     POAM: sp[i](k) |== sp[n](k) ¯|.
```

T H E O R E M 10. If POAM = (P, S, A, B, PL) and sp(k) EL D(k, POAM), then

```
       POAM: sp(k) |-- sp[1](k)  &
       POAM: sp(k) |==[j] sp[j](k)  &  j > 1
==>    POAM: sp[1](k) |==[j-1] sp[j](k);

       POAM: sp(k) |==[m] sp[m](k)  &
       POAM: sp(k) |==[n] . sp[n](k)
==>    n > m  &
       POAM: sp[m](k) |==[n-m] . sp[n](k);

       POAM: sp(k) |==[m] sp[m](k)  &

       POAM: sp(k) |==[n] sp[n](k) ¯|
==>    n > m  &

       POAM: sp[m](k) |==[n-m] sp[n](k) ¯|
```

T H E O R E M 11. If POAM = (P, S, A, B, PL) and sp(k) EL D(k, POAM), then

```
              POAM: sp(k) |== sp[i](k)  &
              POAM(sp(k)) = sp[n](k)
     ==>      POAM(sp[i](k)) = sp[n](k)
```

D e f i n i t i o n 20. The normal overlapping abstract machine NOAM = (P, S, A, B, NL) is an POAM, where P, S, A, and B are elements of POAM, and NL is a subset of $\mathcal{L}$. Normal library NL is an adequately structured set of the so-called normal logical schemes (NLS). An NLS has the following properties:

(1) In NLS there is not an initial operator named init (like in LS); operator init in NLS is an empty (null) operator.

(2) Segments seg[i] (i = 1, ... , r) of a normal logical scheme have the following general form:

```
possible_label[i]:
    or[i];
    IF o_sig[i] THEN
      GOTO (beginning_of_NLS | end_of_NLS);
    ELSE
      null;
    ENDIF;
```

where '(' and ')' are metaparentheses, '|' is an alternative sign, null is a null statement.

(3) Obviosly, after each application of an overlapping rule or[i] the process is continued at the beginning of the PLS if it is not terminated (continued at the end of the PLS).

6. Problem Solving Compositions Using OAM

Overlapping abstract machine OAM = (P, S, A, B, L) is a general problem solver when using its actions (problem solving subprograms in A), overlapping rules (rule base B), and problem solving programs (logical scheme library L). In general, the processor grid P of an OAM is unlimited, and states belonging to S for solving different problems can be disjoint. Further, for practical (technical) reasons, the left state pattern searching when solving particular problem can be limited to some processor subspace (searching rationality). Evidently, the processor space P can be divided in functionally disjoint subspaces. In these subspaces logical schemes belonging to different problem solving tasks can be executed in parallel. This is the so-called second order parallelism of an overlapping abstract machine. The first order parallelism was achieved by the overlapping rule application.

6.1. The Metaoverlapping Abstract Machine

(1) The grid (array, space) of processors is devided into prosessor subspaces. Each processor subspace constitutes a processor grid in the sense of Def. 0. All the processor subspaces constitute the so-called processor metagrid (metaspace).

(2) In the processor metaspace there exists a subspace dedicated for control purposes of metaspace.

(3) There exists a complex serial-parallel problem (user, input problem) which will be solved by the activity over the adequate

information bases in the so-called metaoverlapping abstract machine.

(4) For a given complex serial-parallel problem some initial metastate pattern over processor metaspace is needed. This metapattern will be generated for a given input problem by a special, intial metaoperator (see later, in the metalogical scheme).

(5) Generally, a metapattern is a non-complete array of metastates or metaactions. Metapatterns are occuring in metaoverlapping rules and in metaspace, where metastate is a presentative of a processor subspace. In some way, indices of metaarray entities (states or actions) and indices of subspaces in a metaspace do correspond. Entities of a metapattern array can be directly mapped into subspaces (into registers or memories, which represent the corresponding subspaces).

(6) There will be two kinds of metapattern entities: metastates and metaactions. In the metaspace there will be metastate and metaaction patterns (briefly metapatterns), in metaoverlapping rules (briefly metarules) only metastate patterns. The so-called metaarbiter function will generate metaactions as consequences of metastate subsets causing metaaction pattern in (or on the level of) metaspace. Metapattern entities will be symbols representing specific (complex serial-parallel problem solving) metastates and metaactions.

(7) A complex serial-parallel problem (3) will be solved constructively on the level of a processor metaspace through the levels of its processor subspaces and the way of this constructive solution will be called the metaoverlapping approach. For this approach additional constructs to the existing ones are needed.

(8) In (5) metastate and metaaction patterns were explained and in (1) the processor metaspace was introduced. In (5) and (6) the correspondence of metapatterns to metaspace was explained. By means of metastate and metaaction patterns the metaoverlapping process in the metaspace will be conducted step by step.

(9) For the stepwise conduction of the metaoverlapping process the notion of metarule (metaoverlapping rule) is necessary.

(10) The metaoverlapping rule (MOR for short) consists of two parts: a non-empty metasubrule set and a metaarbiter.

(11) A metasubrule is a rule of the form

IF meta_state_pattern THEN
OVERLAP_BY meta_state_pattern

Metasubrule is an overlapping rule; it is a metapreaction rule, where metaactions will be performed later after using the metaarbiter. The execution of a metasubrule represents a metaoverlapping operation, described in the next paragraph.

(12) A metasubrule has a left metastate and a right metastate pattern (non-coplete array of metastates). By a metasubrule execution (usage, application) all of left metastate patterns included in metaspace (metastates in particular subspaces) will be searched (on the meta level) and each of the found left metastate patterns of the metasubrule will be overlapped by the right metastate pattern of the metasubrule in the metaspace (e.g. in memory on metaspace level). The metastates of the right metastate pattern will be stored (accumulated) in the memories of pattern corresponding subspaces. The result of a metasubrule execution (metaapplication) over the metaspace is accumulation of some metastates in some subspace memories of the metaspace.

(13) In the same systematic way as metaapplication of a metasubrule was defined, a metaapplication of a metasubrule set over the metaspace can be defined. For each metasubrule in a metasubrule set the procedure described in (12) has to be executed. By this, additional metastates are accumulated (stored) in some subspace memories.

(14) By metaoverlapping (applying metasubrules) some subspaces in the metagrid have accumulated more than one metastate (including the original subspace metastate before metaoverlapping) and in these subspaces after terminating metaoverlapping (13) a metaarbitration has to be performed to decide which metaaction in the subspaces with more than one accumulated metastate will occur. For this purpose, to each metasubrule set a metaarbitration function is defined, called the metaarbiter. A metaarbiter will produce a single metaaction in the subspace having more than one accumulated metastate. This metaaction (its symbol) will be stored and executed to produce a first level logical scheme function, reset the accumulated metastates and generate a metaproblem dependent new subspace metastate. Metaactions being performed in a parallel way in different subspaces can communicate among each other; these are the communication problems among parallel processes on the second level of parallelism.

(15) A metaarbiter is a function being defined over a set of metastate subsets producing a metaaction belonging to the metaaction set.

(16) A metasubrule set and a metaarbiter associated to this metasubrule set constitute a pair called the metaoverlapping rule. Procedures described in (11) to (14) represent the socalled metaoverlapping rule application (metaaplication).

(17) The metaoverlapping rule application will be also termed a metaoverlapping step (briefly metastep). A complex serial-parallel problem (3) will be solved by several metaoverlapping steps.

(18) A complex problem solver has a metabase of metaoverlapping rules (e.g. rules of problem solving co-ordination on the second level of parallelism) and can use the subspace grid for solving complex problems. In this case, the definition of a program for metaoverlapping rules application (metalogical scheme) will be necessary. All metaoverlapping rules needed to solve a complex problem by a particular sequence of metaoverlapping rules are elements of the so-called metarule base or briefly metabase.

(19) We can now define the so-called metalogical scheme for applying metaoverlapping rules when solving a complex problem. Operators of the metalogical scheme are symbols (metanames) representing metaoverlapping rules in the metarule base. Metarule symbol in the metalogical scheme represents the metarule call, i.e., the metaaplication (metaexecution) of the me-

taoverlapping rule. In the metalogical scheme there is an metainitialization operator meta_init at its beginning, there are metarule calls, metaif statements and metalabels (as described later).

(20) All left metastate patterns of metasubrules, belonging to a metasubrule set of an metaoverlapping rule are travelling (moving) through the metaspace (space of subspaces). The travelling process is systematic and starts at some "metabeginning edge" of the metaspace. The subspaces of the metaspace are signalling when a metapattern metastate is becoming equal to the subspace original metastate. These signals are attributed by the position (metacoordinates, metaindices) of the signalling subspace in the metaspace.

(21) Each metasubrule in the metasubrule set can have its own monitoring subspace, so, the metaoverlapping through the metaspace is carried out as fast as possible. When the metaoverlapping process has reached the so-called "metaending edge" of the metaspace the metarule metaarbiter function in the subspaces having accumulated more than one metastate is performed, generating metaactions. Here, metaactions represent also execution of particular logical schemes from library L. The metaaction pattern in the subspace metaspace represents a distribution of parallel logical scheme executions to corresponding subspaces. By this, the metaoverlapping step (metaaplying of a metaoverlapping rule) is terminated.

(22) A particular subspace in the metaspace is signalling if a metaoverlapping rule was really applied. Application of a metaoverlapping rule means that at least one overlapping of a left metastate pattern in the metaspace metastate pattern by right metastate pattern was performed. The notion of the metarule application register is important at the further contruction of the metalogical scheme.

(23) A metalogical scheme is a sequence of metaoverlapping rule call statements, metaif statements, and metalabels. Metalogical scheme MLS is a program (algorithm) of the form

MLS = MLS(m_init, m_or[1], ... , m_or[r], m_o_sig[1], ... , m_o_sig[r]),

where m_init is the metastate pattern initialization operator, m_or[i] (i = 1, ... , r) are metaoverlapping rule calls for metarules belonging to metarule base MB, and m_o_sig[i] are metaoverlapping signals as described in (22) above. Schematically, for an MLS we have the following segments:

m_init; m_seg[1]; m_seg[2]; ... ; m_seg[r];

where for m_seg[i] (i = 1, ... , r) there is

```
possible_meta_label[i]:
    m_or[i];
    IF m_o_sig[i] THEN
      (null) ! (GOTO possible_meta_label[p]) !
        (GOTO end_of_MLS));
    ELSE
      (null ! (GOTO possible_meta_label[q]));
    ENDIF;
```

Here, '(' and ')' are metaparentheses, null is a null statement, '!' is metaalternative sign, and p, q = 1, ... , r. Metasegments are metafree or metaterminally (alternatives in the upper THEN clause).

(24) A metalogical scheme is a program for subspace allocations to parallel metaactions in the subspace metaspace. A metarule call in the metascheme represents several inherent and complex problem solving metaoperations being the following:

(a) searching for all left metastate patterns of the metasubrule set in the instantaneuos subspace metastate pattern;
(b) overlapping of each in the instantaneous subspace metastate pattern occuring left metastate patterns of metasubrule set by the corresponding right metastate pattern;
(c) parallel metaarbitrating in subspaces having accumulated more than one metastate after overlapping by transforming these metastates to metaactions;
(d) parallel execution of metaactions from (c), reseting the accumulated metastates and determining the new subspace metastate.

(25) Metaactions, resulting by metaarbitration in a metarule call represent complex parallel problem solving processes or parallel usage of subspaces, i.e., overlapping abstract machines with selected logical schemes. Here, logical schemes of executing OAM's represent subproblem actions.

(26) Some remarks to the initial metastate pattern in the metaspace and initial operators of OAM logical schemes are necessary. OAM's in the problem solving must ignore their initial operators (first level parallelism) and act over the resulting state patterns from the previous OAM executions. We conclude, that in sequential problem compositions only the very first OAM's are using LS's with initialization operators; in all following LS applications by OAM's initialization operators are ignored.

D e f i n i t i o n 2 1. The metaoverlapping abstract machine (briefly MOAM) is a quintuple

MOAM = (MP, MS, MA, MB, ML),

where:

MP is a multidimensional processor subspace grid, called metagrid, with a subspace for metacontrol purposes; each grid subspace has the properties of the OAM (Def. 0); each OAM[position] in the metagrid is a quintuple

OAM[position] = (P[position], S, A, B, L),

where P[position] represents the processor subspace having the corresponding position index; elements S, A, B, L are common for all OAM's in the metagrid and their meaning is described in Def. 0;

MS is a metastate set similar to state set S; metastates are distiguishable from states and their role is to control complex problem solving in the metagrid (metaspace); S can be seen as a subset of the set MS; by this, metastates are elements of the set MS \ S;

MA is a metaaction set similar to action set A; but actions and metaactions are semantically different; actions are problem solving subprograms, generating a part of solution and a new state; metaactions are calls of existing logical schemes from library L for

execution in a subspace (OAM) ignoring or not the initial LS operator and producing a new metastate; set A can be seen as a subset of the set MA; by this, metaactions are elements of the set MA \ A;

MB is a metaoverlapping rule base and has the similarity of the overlapping rule base B; set B can be seen as a subset (subbase) of the base MB; so, metaoverlapping rules are elements of the set MB \ B;

ML is a metalogical scheme library; comparing to logical scheme, which is an overlapping approach for solving a problem, a metalogical scheme represents a metaoverlapping approach for solving a complex problem being composed of problems (subcomplex or particular problems); also set L can be seen as a subset (sublibrary) of library ML; so, metalogical schemes are elements of the library ML \ L.

The definition of MOAM (this definition) is transparent to the definition of OAM (Def. 0).

D e f i n i t i o n 2 2. The consequence of transparency between OAM and MOAM is the analogy of MOAM definitions to OAM definitions (Defs. 1 to 12); this analogy proceeds also from paragraphs (4) to (26) of this section.

6.2. Basic Theorems for MOAM Application

D e f i n i t i o n 2 3. The application (metaaplication) of an MOAM = (MP, MS, MA, MB, ML) is defined as follows. For the given metaspace (metagrid) MP a metalogical scheme MLS from the library ML is selected. This metascheme represents metaprocesses needed for a complex problem solution. By initial metaoperator meta_init of the metascheme the metagrid is subspace state initialized. Further, the metalogical scheme calls the metaoverlapping rules belonging to base MB for execution in terms of the values of metaoverlapping signals. By metaarbitration within the metaoverlapping rules complex problem solving metaactions belonging to MA are executed. At the end of the metalogical scheme the MOAM complex problem solving process is stoped and MOAM is ready to solve another complex problem. Obviously, Defs. 14 to 18 for OAM are transparent to MOAM, where OAM deduction symbols

$$\vdash, \quad \vdash=, \quad \vdash_{-}[m], \quad \vdash=[n]$$

can be replaced by metaoverlapping deduction symbols

$$\overset{M}{\vdash}, \quad \overset{M}{\vdash=}, \quad \overset{M}{\vdash_{-}}[m], \quad \overset{M}{\vdash=}[n],$$

where M denotes the metadeductive process.

T H E O R E M 1 2. Let the following parameters be given:

(a) a metaoverlapping step marked by 'i' representing the number of performed metaoverlapping steps from the beginning of MOAM application, e.g., on k-dimensional metastate pattern msp[i](k);
(b) a metaoverlapping rule marked by 'j', e.g., MOR[j];
(c) a number of parallel metaactions within the execution of a metarule is equal to the cardinality of the interval (first[j], last[j]);
(d) an index 'g' of the executing OAM[g] within metastep in a given subspace of metaspace with the selected logical scheme LS[g];
(e) a number 'n' of overlapping steps n[g] for subspace OAM[g].

By this, the following equivalence is valid:

$$\mathrm{MOR}[j]:\ msp[i-1](k) \overset{M}{\vdash_{-}}[j]\ msp[i](k) \quad <=>$$

$$g=last[j] \ \&\ (\dot{\vee}_{g=first[j]}\ (OAM[g]:\ sp[j,g,0](k) \overset{\cdot}{\vdash})) \quad <=/=>$$

$$(OAM[g]:\ sp[j,g,0](k) \vdash=[n[j,g]] \cdot sp[j,g,n[g]](k)) \quad <=/=>$$

$$(OAM[g]:\ sp[j,g,0](k) \vdash=[n[j,g]] \cdot sp[j,g,n[g]](k) \overset{\cdot}{\vdash}))$$

P r o o f. This equivalence shows how complex a single metaoverlapping step actually could be or which subspaces (OAM's) of the metaspace (MOAM) could be involved in a single metaoverlapping step. This equivalence is a consequence of the previous MOAM definitions and of the Theorem 1.

T H E O R E M 1 3. If MOAM = (MP, MS, MA, MB, ML) and for initial metastate pattern msp[0](k) for selected metalogical scheme MLS there is msp[0](k) EL D(k, MOAM), and OAM[(h, g)] (h = 1, ... , q; g = first[h], ... , last[h]), then the following equivalence is valid:

$$\mathrm{MOAM}:\ msp[0](k) \overset{M}{\vdash=}[q]\ msp[q](k) \ \&\ q > 0$$

$$<=>$$

$$h=q \ \&\ (\dot{\vee}_{g=last[h]}\ (\dot{\vee}_{h=1}\ (OAM[h,g]:\ sp[h,g,0](k) \overset{\cdot}{\vdash}))$$

$$<=/=>$$

$$(OAM[h,g]:\ sp[h,g,0](k) \vdash=[n[h,g]] \cdot sp[h,g,n[h,g]](k))$$

$$<=/=>$$

$$(OAM[h,g]:\ sp[h,g,0](k) \vdash=[n[h,g]] \cdot sp[h,g,n[h,g]](k) \overset{\cdot}{\vdash})))$$

P r o o f. This theorem shows the possibilities of the so-called free translation of the initial, k-dimensional metastate pattern msp[0](k) by q metaoverlapping steps into k-dimensional metastate pattern msp[q](k) and is the consequence of the previous theorem.

From Theorems 1, 2, 3, 4, and 5 the so-called metaequivalent theorems can be deduced. From Def. 19 a metaequivalent definition for primitive metaoverlaping abstract machine PMOAM can be constructed. By this metadefinition, metaequivalent theorems to Theorems 6, ... , 11 are possible.

R e m a r k 2. The question is, which kinds of parallel and sequential processes can be realized by a metaoverlapping abstact machine. MOAM represents only a special kind of OAM composition. Does this composition (metaoverlapping) approach enable all the possible parallel-sequential and sequential-parallel process configurations?

The extreme parallel case (only parallel processes) can be achieved by an MOAM having only one metaoverlapping step, where all of its OAM's act only in one overlapping step. In Theorem 13 this case would be determined by h = 1 and g = first[1]. In this case, parallel processes can communicate within each particular OAM and process communication should be possible also within MOAM (inter-OAM communication). The problem is how to decompose the

input problem into parallel subproblems, and how to decompose each subproblem into parallel basic actions.

The extremely sequential case (only sequential processes) can be achieved by an MOAM, where in each metaoverlapping step, there is active at most one OAM (there is always only one metaaction), and each acting OAM has overlapping steps with at most one action. In Theorem 13, this case is within limits h = 1, ... , q and g = first[h]. All other cases are parallel-sequential or sequential-parallel.

7. Conclusion

Computer design philosophy has entered the age of parallelism, and it will never go back. But current programming languages are totally inadequate for parallel processing. To use the overlapping approach efficiently an adequate programming language is necessary. There will be a long way from serial to parallel processing.

Parallel processing and the overlapping approach within it is a solution looking for a problem. Parallel processing is been developed for many years. Now the researchers are trying to split problems to fit them into parallel processing. For instance, payrolls are not problems to run on several hundreds of processors; they run linearly until the paychecks are done. The problem is to find problems to which parallel processing can be applied, because today we understand problems as being linear by nature.

Overlapping seems to be a natural operation on multidimensional arrays of objects as substitution (mathematics) and replacement (normal algorithms, grammars) are natural operations on linear arrays (strings, words). The overlapping approach covers, for example, the well-known overlapping of I/O with arithmetic and logic unit and also another important parallelism called pipelining.

References

[1] Zeleznikar, A. P., "Overlapping Algorithms," Mathematical Systems Theory, Vol. 1, No. 4, pp. 325-345, 1967 (Springer-Verlag, New York).

[2] Zeleznikar, A. P., "Some Algorithm Theory and Its Applicability," Periodicum Math.-Phys.-Astr., Vol 18, pp. 141-158, 1963 (Zagreb, Yugoslavia).

[3] Lineback, J. R., "Parallel Processing: Why a Shakeout Nears," Electronics, Oct 28, 1985, pp. 32-34.

[4] Serlin, O., "Parallel Processing: Fact or Fancy?" Datamation, Dec 1, 1985, pp. 93-105.

List of Used Symbols

Symbol:	Meaning in:
A	Def. 0: Action set
arb	Def. 9: Overlapping arbiter
B	Def. 0: Overlapping rule base
B(*,k)	Def. 6: Set of pattern shapes
COMP	Def. 5: Set composition operation
D(k,OAM)	Def. 6: Pattern domain of OAM
EL	Def. 1: Set relation 'being the element of'
EX	Def. 3: Quantor 'there exists'
IN	Def. 6: Relation 'being in'
INCL	Def. 4: Set inclusion
isp(k)	Def.15: Initial state pattern
I(k)	Def. 1: Integer Cartesian product of k factors
L	Def. 0: Logical scheme library
left[i](k)	Def. 8: Left subrule state pattern, index i
LS	Def.12: Logical scheme
NL	Def.20: NOAM logical scheme library
NLS	Def.20: Normal logical scheme
NOAM	Def.20: Normal overlapping abstract machine
MA	Def.21: Metaaction set
MB	Def.21: Metaoverlapping rule base
meta...	Concerns an MOAM entity
ML	Def.21: Metalogical scheme library
MOAM	Def.21: Metaoverlapping abstract machine
MOR	Theorem 12: Metaoverlapping rule
MP	Def.21: Multidimens. processor subspace grid
MS	Def.21: Metastate set
M(k)	Def. 1: Subset of Cartesian product I(k)
NOT_EL	Def. 5: Relation 'being not the element of'
OAM	Def. 0: Overlapping abstract machine
OR	Def.10: Overlapping rule
o_sig	Def.15: Overlapping signal
P	Def. 0: Multidimensional processor grid
PL	Def.19: POAM logical scheme library
PLS	Def.19: Primitive logical scheme
POAM	Def.19: Primitive overlapping abstract machine
right[i](k)	Def. 8: Right subrule state pattern, index i
r(*,k)	Def. 5: Pattern shape, k-dimensional
S	Def. 0: State set
sp(a,k)	Def. 2: a-displacement of pattern sp(k)
sp[j](k)	Def.15: State pattern, index j, k-dimensional
sp(k)	Def. 1: State pattern, k-dimensional
sp(k,x)	Def. 1: Function sp(k)(x)
sp(*,a,k)	Def. 6: a-displacement of sp(*,k)
sp(*,k)	Def. 5: Shape of the pattern sp(k)
sq(k)	Def. 4: State pattern, k-dimensional
SR	Def. 7: Overlapping subrule
SRS	Def. 8: Subrule set
U	Def. 8: Set union operation
V	Def. 5: Logical 'or' operation

'.INCL'	Def. 4: Pattern inclusion
'. sp[n](k)'	Terminal state pattern
'.='	Def. 3: State pattern equivalence
<=>	Logical equivalence
<=/=>	Logical non-equivalence
[i]	Index i of an entity
[i[j]]	Index i[j] of an entity
:--	Def.10: One overlapping step
:--[i]	Def.15: Overlapping step belonging to OR[i]
M :--	Def.23: One metaoverlapping step
:==	Def.15: At least one overlapping step
:==[j]	Def.15: j overlapping steps
M :==	Def.23: At least one metaoverlapping step
¯¯ :	Defs.15 and 17: State pattern before ¯¯ : is an exact translation by an OAM.

Prekrivanje: vzorec za paralelno in zaporedno procesiranje

Povzetek. V tem članku je prikazan koncept takoimenovanega prekrivanja, ki je bil pravzaprav izpeljan iz pojma prekrivnega algoritma (glej referenci 1 in 2). V okviru tega koncepta sta definirana dva osnovna konstrukta: prekrivni (OAM) in metaprekrivni abstraktni stroj (MOAM). Ta dva konstrukta sta v bistvu paralelno-zaporedna problemska reševalnika. V članku je navedenih več formalnih definicij, ki opredeljujejo prekrivni in metaprekrivni abstraktni stroj (23 definicij), iz teh definicij pa so izpeljani izreki (13 izrekov). Pokazano je, kako lahko prekrivni abstraktni stroj izvaja zaporedje paralelnih akcij (procesov) in kako je mogoče z uporabo metaprekrivnega abstraktnega stroja paralelizem še "povečevati." Prekrivni in metaprekrivni abstraktni stroji se lahko enostavno modelirajo z uporabo znanih paralelnih arhitektur, kot so npr. hiperkocka, vodilo in preklopni sistem in tudi z uporabo paralelnih programirnih jezikov, kot je npr. paralelni Prolog in uporaba varovalnih Hornovih klavzul.

A SURVEY OF MICROPROCESSOR ARCHITECTURES FOR MEMORY MANAGEMENT

INFORMATICA 2/86

B. Furht

Department of Electrical and Computer Engineering
University of Miami, Coral Gables, Florida 33124

V. Milutinović

School of Electrical Engineering
Purdue University, West Lafayette, Indiana 47907

UDK: 681.3.325.6.08

This paper presents an overview of current microprocessor architectures which support memory management. Basic requirements for a processor to support the memory management are defined, and the hierarchically organized memory is introduced. Several address translation schemes, such as paging, segmentation, and combined paging/segmentation are described, and their implementation in current microprocessors is discussed. A special emphasis is given to the application of the associative cache memory. Single-level and multi-level address mapping schemes are analyzed and compared. Furthermore, the paper discusses the capabilities of current microprocessors to support virtual memory, which includes abilities to recognize an address fault, to abort the execution of the current instruction and save necessary information, and the ability to restore the saved state and resume normal processing. Two methods to restart the interrupted instruction, instruction restart and instruction continuation, are evaluated, and their implementation in current microprocessors is discussed. Protection and security requirements are defined, and two protection schemes, hierarchical and non-hierarchical, are evaluated.

I. INTRODUCTION

New generation 16-bit and 32-bit microprocessors are extensively used in multiuser and multitasking environments. Therefore, there is an increased demand for the support of memory management. Furthermore, as shown in Figure 1, the capacity of primary and secondary memories in advanced microprocessors is increasing, which in turn requires an increased virtual memory space, as well as more sophisticated virtual memory management mechanisms.

In the 16-bit microprocessor arena, the techniques applied to solve memory management problems are relatively inadequate, and inefficient. At the 32-bit level, a more standardized approach can be found, and significantly more sophisticated architectures for memory management have been designed. The paper evaluates various architectures for memory management and virtual memory support, and their implementations in existing microprocessors. Several important issues are addressed, such as selection of a virtual memory organization, multi-level memory mapping schemes, associative cache memories applied to address translation, virtual memory support techniques, dynamic memory allocation algorithms, as well as protection and security techniques.

The implementation of these techniques in current 16-bit and 32-bit microprocessors, such as Intel 286, 386, and 432, Motorola 68010 and 68010, National 32032, and Zilog Z80,000, is discussed.

The paper is organized in eight sections. The Section 2 discusses the requirements for a processor to support memory management. Two main strategies applied in current microprocessors are presented: memory management unit (MMU) on-the-CPU chip versus off-the-CPU-chip. Two memory addressing schemes, linear and augmented, are evaluated. Section 3 deals with the various address translation techniques, such as paging, segmentation and combined paging/segmentation, and their implementations in current microprocessors. Both single-level and multi-level address mapping schemes are

evaluated. Techniques to support virtual address mechanism are presented in Section 4. The implementations of two methods, which resume operation after an address fault is detected and corrected, are discussed. Section 5 describes the security and protection techniques applied in current microprocessors.

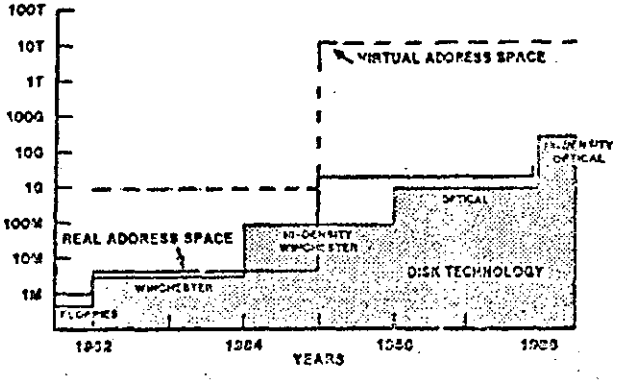


Figure 1. Addressing range needs [54]

2. MEMORY MANAGEMENT REQUIREMENTS

Advanced microprocessor system architecture, which is able to support memory management, uses the hierarchically structured memory system, as shown in Figure 2.

The memory system consists of three levels and involves the maintaining of a large address space based on a hierarchy of memory devices, which differ in memory capacity, speed, and cost. At the first level is the high-speed cache memory, which is the most expensive and, therefore of the lowest capacity. At the second level is the real (primary) memory, which is slower, but less expansive than the cache memory. The

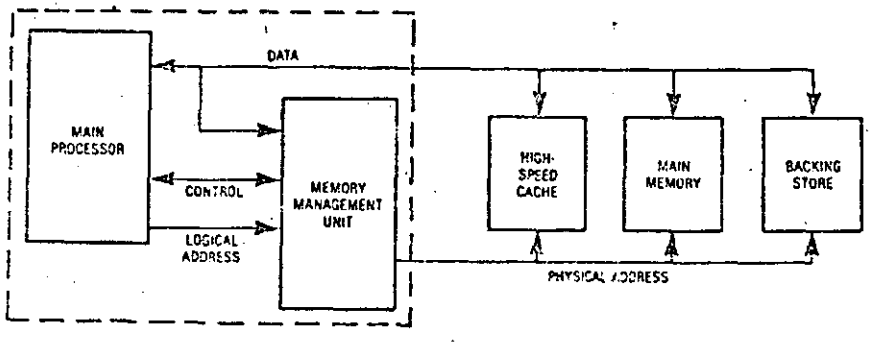


Figure 2. Microprocessor system architecture with three levels of hierarchically organized memory to support memory management [36]

third level consists of large capacity storage devices, such as disks, which hold the programs and data that cannot fit in the first two levels. When a process is to be run, its code and data are brought into primary or cache memory, where cache memory always holds the most recently used code or data.

In this hierarchical memory structure, the basic requirements of the memory management system can be specified as follows:

1. ability to translate addresses and support dynamic memory allocation,
2. ability to support virtual memory, and
3. ability to provide memory protection and security.

There are two basic strategies in creating the microprocessor system architecture for memory management:

1. memory management unit is on the CPU chip, and
2. memory management unit off the CPU chip.

Both strategies, as well as the list of microprocessor systems which apply them, are indicated in Figure 3.

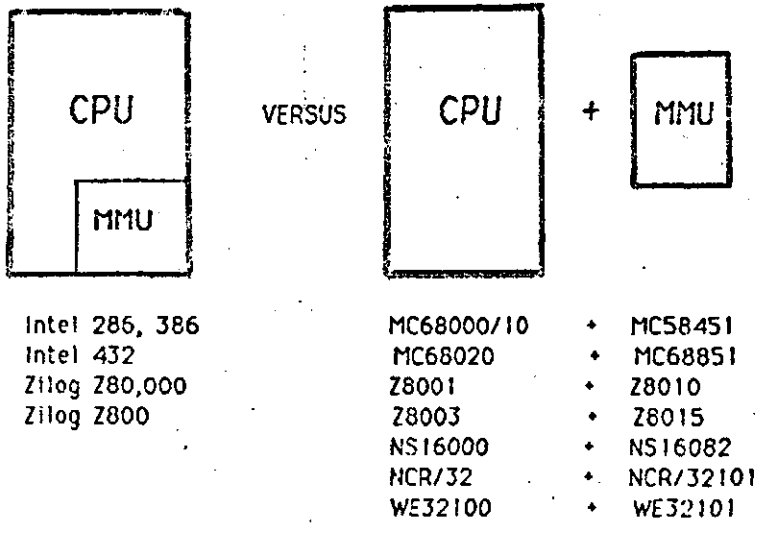


Figure 3. On-chip versus off-chip memory management unit

The main advantages of having the memory management on the CPU chip are:

1. access time improvement, because there is no off-chip MMU-related delays,
2. maximum portability of operating system and application programs, and
3. parts-count reduction.

On the other hand, the memory management on the CPU chip requires additional transistor count, which could be invested into other more frequently used resources. For example, the Motorola 68020, which applies memory management off the CPU chip, uses the saved transistor count to implement the instruction cache on the chip.

Another important issue related to memory management is selection of the memory organization scheme. Basically, there are two types of memory organization schemes: linear and segmented.

In the linear addressing schemes, addresses typically start from zero, and proceed linearly. The memory may later be structured, by software, at the level of address translation.

In the segmented addressing schemes, the programs are not written as a linear sequence of instructions and data, but rather as modules of code and data. The logical address space is broken into several linear address spaces, each of the specified length. An effective logical address is computed as a combination of the segment number, which is a pointer to a block in memory, and the segment offset, which defines the displacement within the segment.

Table 1 shows memory addressing schemes applied in various advanced microprocessors.

Note that Intel and Zilog offer both segmented and linear addressing on their 32-bit processors i80386 and Z80,000, respectively, as software programmable options.

In general, a linear addressing scheme is better suited for the applications that manipulate large data structures, while the segmented addressing scheme facilitates programming, enabling the programmer to structure software into segments. In addition, the segmented addressing scheme simplifies protection and relocation of objects in memory. As an example of the segmented addressing scheme, Intel's i8086 processor contains four 16-bit segment registers, which point to four objects in the memory: code, stack, data, and extra segment (alternate data), as shown in Figure 4a. The address calculation mechanism, which produces 20-bit physical address for the i8086, is shown in Fig. 4b.

3. ADDRESS TRANSLATION TECHNIQUES

Regardless of the memory organization scheme, the processor must have an address translation mechanism to handle virtual memory. The address translation mechanism also provides a method of protecting memory objects.

The address translation is a process of mapping logical to physical memory

TABLE 1
Memory addressing schemes in advanced microprocessors

PROCESSORS	ADDRESSING SCHEME	
	Linear	Segmented
Intel		
8086,80286,432		*
80386	*	*
Motorola		
68000, 68010, 68020	*	
National		
16032, 32032	*	
Zilog		
Z8000 family		*
Z80,000	*	*
AT&T		
WE32100	*	
NCR		
NCR/32	*	

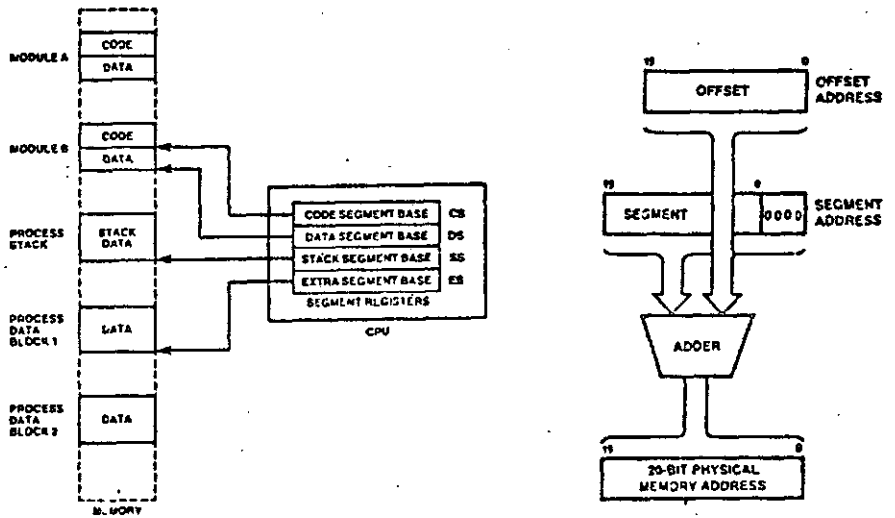


Figure 4 Segmentation and address calculation
a. Segmented addressing scheme of the i8086 [28]
b. Address calculation in the i8085 [28]

addresses. The address translation mechanism divides the memory into blocks, and then performs mapping of a block of logical addresses into a block of physical memory addresses. It allows programs to be relocated in the primary memory. It also provides the base for virtual memory system design, where the logical address space can be larger than the physical address space. The virtual memory mechanism allows programs to execute even when only few blocks of a program are in the primary memory, while the rest of the program is in the secondary memory (on the disk). The other important processor requirements for virtual memory support are discussed in Section 4. Three basic address translation schemes are:

1. paging
2. segmentation, and
3. combined paging/segmentation.

In the paging systems, the primary memory is divided into fixed-size blocks (pages), while in the segmentation systems, the blocks are of various size (segments), as shown in Figure 5.

Generally, the segments can overlap, while pages cannot, so pages are usually of a relatively small size, compared to total memory. Typical page size is between 256 and 2048 bytes, while segments can be 64K bytes or more.

The paging/segmentation systems combine the features of both paging and segmentation addressing schemes. The segmentation part of the scheme manages virtual space by dividing the programs into segments, while the paging part manages physical memory, which is divided into pages. Each segment consists of a number of pages, as shown in Figure 6.

Selection of the address translation mechanism has a crucial impact on the memory management techniques, which have to be implemented by the operating system, to handle page or segment fetching, placement, and replacement. For example, the paging address translation system is well suited for page placement and replacement, because all pages are of uniform size, while the segmentation system needs more complicated placement and replacement algorithms to match incoming segments with available memory space in the segmentation systems, a segment must reside entirely in physical (primary) memory in order to be executed, because the minimum unit that can be swapped is the segment itself. The available memory space becomes then fragmented into many small pieces, and there is not enough contiguous memory for storing one large segment. Because of the fragmentation problem associated with the segmentation systems, the paging systems are more efficient with respect to memory utilization. In the paging systems, all pages are of equal size, thus pages can be swapped without leaving unusable fragmented spaces. Also, it is not necessary to swap in all pages of a program at once, in order to execute it, but only the pages required ("demand paging"). This significantly reduces the swapping time.

For all these reasons, the demand paging address translation system seems to be

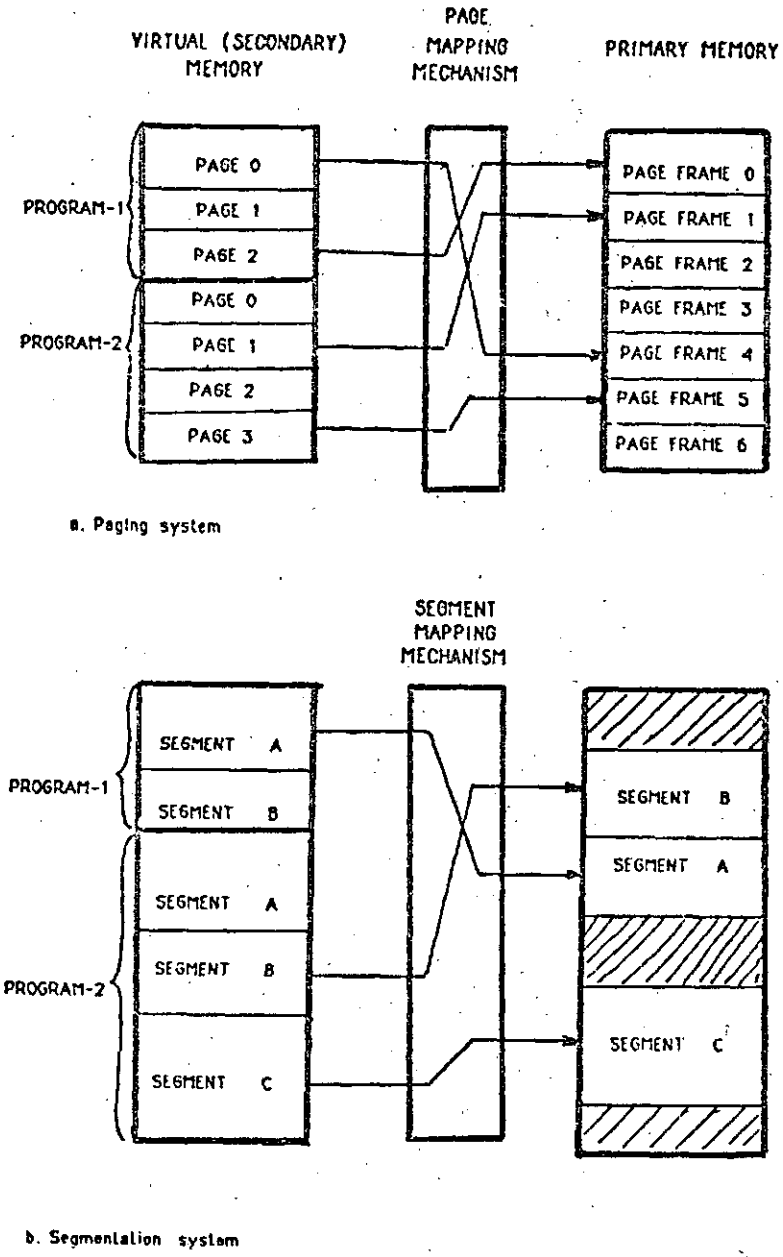


Figure 5. Address translation schemes
a. paging
b. segmentation

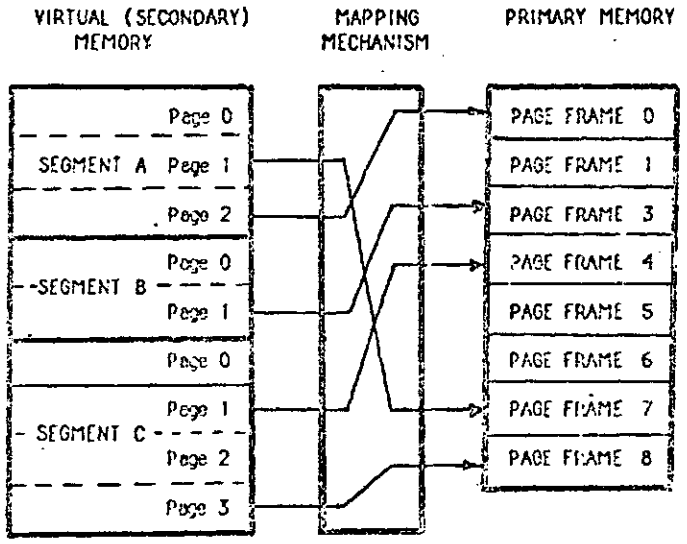


Figure 6 Address translation by combined paging and segmentation (paging/segmentation)

the way to go. As a matter of fact, all advanced 32-bit processors, as well as several 16-bit processors, fully support demand paging technique, which may become a standard address translation mechanism in future microprocessors.

Furthermore, when one selects the address translation scheme (paging, segmentation, or combined system), there are two additional issues which should be addressed:

1. implementation of the selected address translation mechanism, and
2. selection of the number of mapping levels

3.1 Implementation of the address translation schemes

Regardless of the address translation organization, the implementation method is always based on translation tables located in primary memory: page map tables (PMT) in the paging systems, and segment map tables (SMT) in the segmentation systems [10,11,14,16]. The table entries contain information to translate the logical into the physical address, as well as additional data for protection purposes, and to support placement and replacement algorithms. A typical format of a translation table entry is shown in Figure 7.

As an example of the address translation implementation, the virtual address of the i286 processor consists of a pair: segment selector and displacement v=(s,d). The

RESIDENCE BIT	ACCESS RIGHTS & PROTECTION	SUPPORT FOR REPLACEMENT	PHYSICAL ADDRESS

Figure 7. Typical format of a page or segment table entry

segment selector points to the segment descriptor in the segment map table, as shown in Figure 8. The segment descriptor contains the primary memory address s', at which the segment begins. The displacement d is added to s' forming the real physical address, r=d+s', corresponding to the virtual address v.

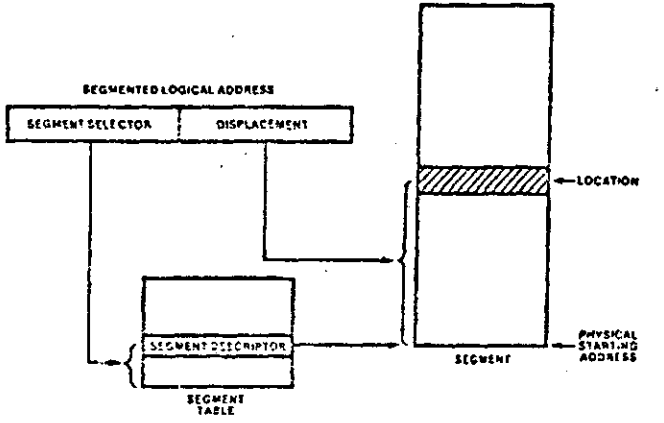


Figure 8. Address translation mechanism of the i286 [28]

The described address translation implementation method is known as direct mapping. Translating a logical address to a physical address, using direct mapping, requires an additional memory access operation to obtain segment (or page) base address, and therefore the use of direct mapping can cause the computer system to run programs at lower speed. There are several solutions applied in modern microprocessor architectures to overcome this problem. These solutions are discussed below.

In the Intel's i286 processor standard, four segment registers are extended with the corresponding four 48-bit segment descriptor cache registers, as shown in Figure 9 [26,28].

Segment registers are loaded by the program, while the CPU loads the explicit cache registers, which are invisible to programs. Explicit cache speeds up the operation by eliminating the need to refer to a descriptor table for every memory reference instruction. Loading the explicit cache is performed in four steps:

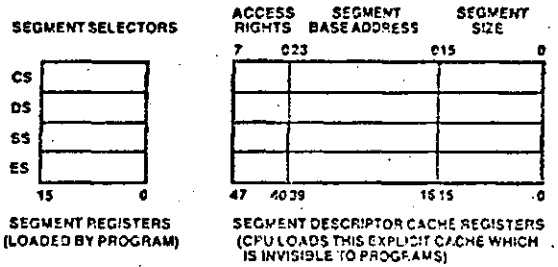


Figure 9. Descriptor data type in the i286 [28]

1. Program places a selector in the corresponding segment register.
2. Processor adds the selector index to the base address of the descriptor table, to select a descriptor.
3. After the processor verifies segment access rights, it copies the descriptor to the data segment register in cache.
4. The processor uses the descriptor information to check segment types and limits, as well as to form the effective address.

The described technique based on explicit cache registers speeds up the direct mapping, but still is not efficient enough, because it requires cache loading whenever control is transferred from one to another segment of the same type.

A much more sophisticated solution is based on a special associative cache (32 to 64 locations), which holds the most recently used set of translation values. Then, the translation process is performed in the following steps, as shown in Figure 10:

1. First, the virtual address received from the CPU is searched through the cache. If the address matches with one of the cache entries, then the corresponding physical address stored in the cache is used by the CPU to access the primary memory directly.
2. If the received virtual address does not match with cache entries, but the page or segment is in the primary memory, then the physical address will be fetched from the translation tables located in the primary memory, and then stored in the cache. Then, the CPU will access the required physical memory.
3. Finally, if the page or segment is not in the primary memory, it must be first swapped from the secondary memory into the primary memory. The translation tables must be updated, and the physical memory address will be fetched from the translation tables into the cache.

The associative cache memory is usually organized as a Translation Lookaside Buffer (TLB). When the address translation mechanism receives a logical address, every entry in the TLB is searched simultaneously for the logical address.

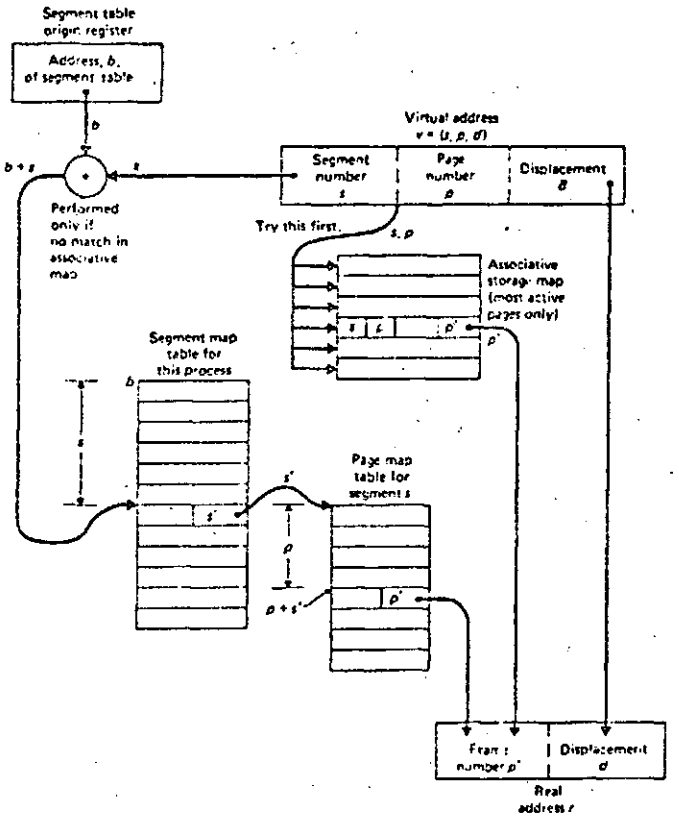


Figure 10. Address translation mechanism using the associative cache memory [10]

A number of simulation studies have proven that the small associative cache significantly speeds up the system operation, because the hit ratio of finding the address in the cache riches 98%. Many recent processors, (such as Motorola 68000 family with its MC58451 MMU, Intel 80386, Zilog Z8000 family and Z80,000, National NS16000 family with its NS16082 MMU, and others) have implemented the address translation scheme by using the TLB method [5,33,34,35,37,50,51,54]. Although translation mechanisms based on the TLB method vary in complexity, they can be classified in two basic groups: address-accessable TLBs, and content-addressable TLBs. In the address-accessable TLB approach, a logical address field identifies the register in the TLB that holds the physical base address. As an example of this technique, the Z800 with on-chip MMU is shown in Figure 11 [33].

The virtual address consists of a 4-bit TLB pointer, and a 12-bit offset. The TLB pointer selects one of the 16 translation registers of the TLB. Then, the 24-bit physical

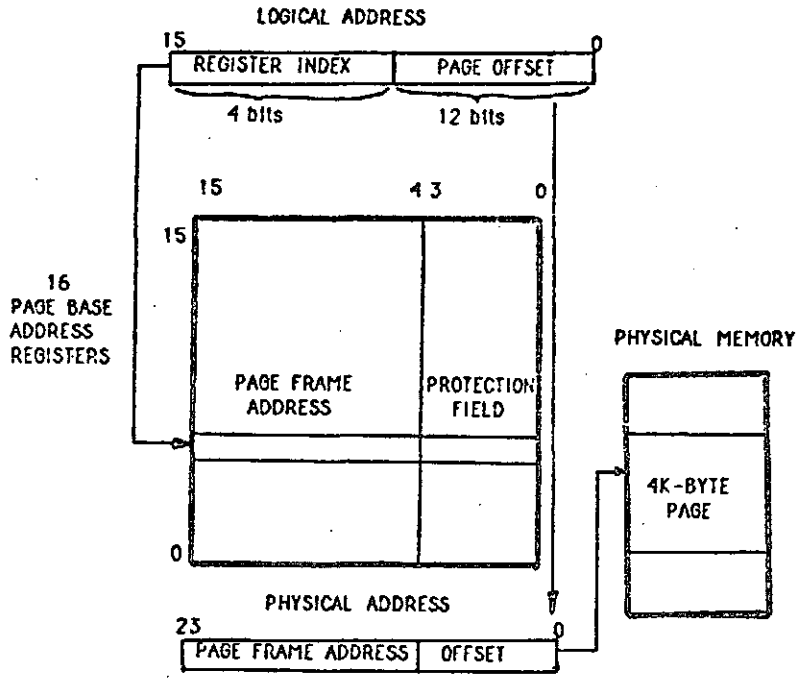


Figure 11. The Z800 address translation based on the address-accessable TLB [33]

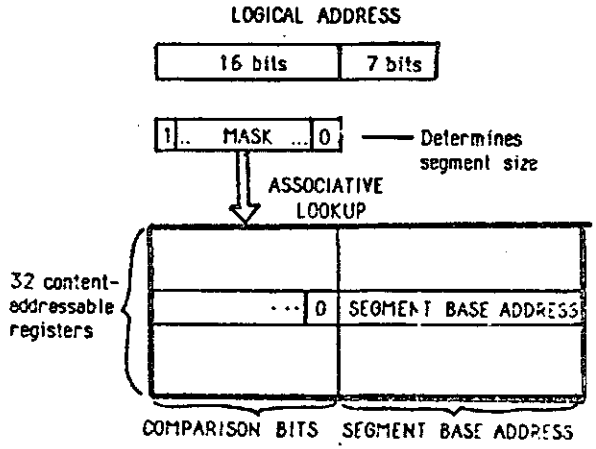


Figure 12. Content-addressable TLB in the MC58451 MMU [44]

address is formed, as a selected 12-bit page base address from the TLB, concatenated with the 12-bit offset.

The address-accessable TLB technique is not practical for large systems, because accessing the TLB by addresses requires a segment register for each logical segment or page that can be relocated.

The content-addressable TLB is more suitable for large systems. This method has been applied in several microprocessors, such as the MC68451 MMU, Z8015 PMMU, Intel 80386, and NS 16082 MMU. To illustrate this method, Figure 12 shows the content-addressable TLB applied in the MC68451 MMU.

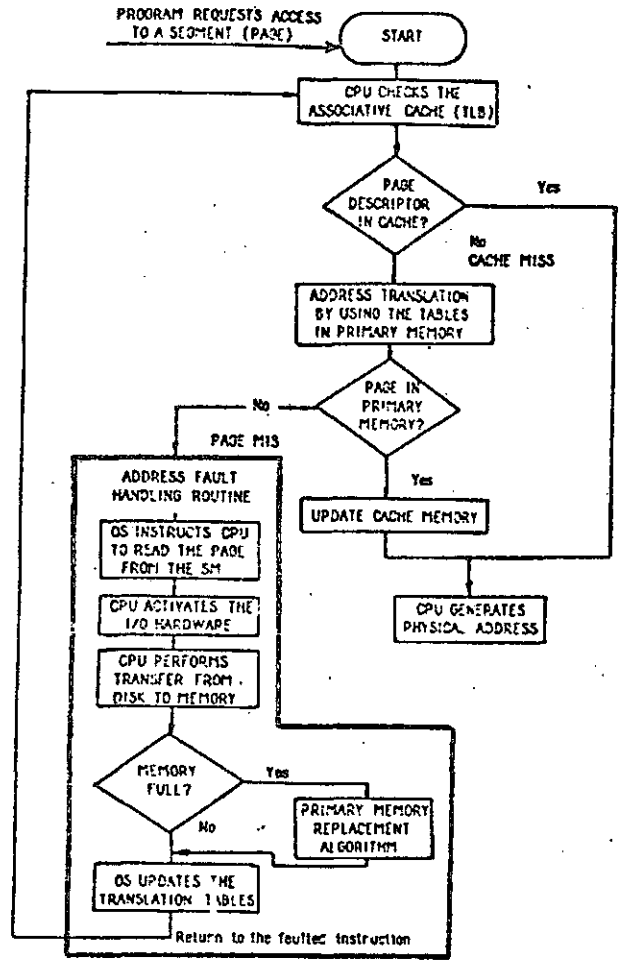


Figure 13. The flow-chart of a paging virtual memory system with an associative cache memory

The MMU receives a logical address (23 bits), and the mask register masks the low order bits to determine the segment size. Then, the MMU compares the rest of the most significant bits with the comparison field values of 32 content-addressable registers. If a match is found, the MMU performs address translation. If there is no match, the MMU generates a fault condition, and activates a trap routine. The trap routine will update the TLB from translation tables stored in primary memory.

The flow-chart in Figure 13 illustrates the necessary operations in a paging-based virtual memory system with an associative cache memory for recently used pages.

The virtual memory is activated whenever program requests an access to a page. The flow-chart in Figure 13 indicates three different control paths:

1. when the page descriptor is found in the associative cache,
2. when the page descriptor is not found in the cache ('cache miss'), but the page is in the primary memory, and
3. when the page descriptor is not found in the associative cache, and the page is not in the primary memory ('page miss'). Then, the address fault handling routine is activated.

In addition to address translation mechanism, the MC68451 MMU supports dynamic memory allocation. The dynamic memory allocation mechanism is able to allocate the memory to a process, while it is running. The Binary Buddy system, an algorithm for dynamic memory allocation, is implemented in the MC68451. The algorithm divides the entire physical address space into buffers, the size of which varies from 256 bytes to 256K bytes (in the MC68451). The algorithm maintains these buffers by using the buffer lists for all sets of buffers of the same size, as well as buffer descriptors for each buffer independently [52].

When a memory request is received, the algorithm searches through the list of available buffers in order to find the best fitted buffer. If the best-fitted buffer is not available, the search process is continued for the next larger size buffer. The flow-chart of the Binary Buddy algorithm is shown in Figure 14.

A detailed description of the algorithm, as well as additional issues related to it, are discussed in [45,48,52].

3.2 Single-level versus multi-level address mapping

The second issue closely related to address translation architectures is dealing with the number of mapping levels in address translation schemes. The conventional address mapping scheme consists of just one mapping level, such as in most of the 16-bit processors (i286, Z8010, and Z8015 MMUs). On the other hand, almost all 32-bit processors use multi-level mapping schemes, which brings some new features in the memory management.

The basic advantages of multi-level mapping schemes versus single-level mapping schemes can be summarized as follows:

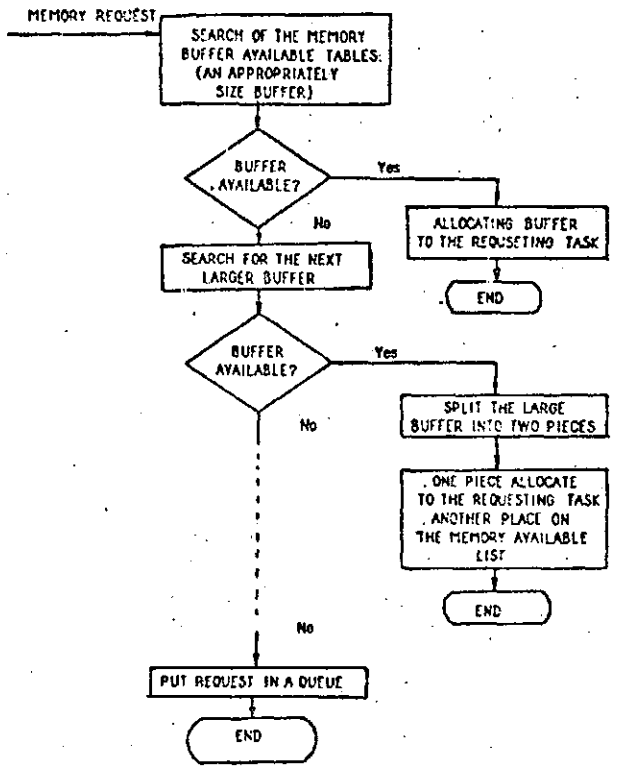


Figure 14. Binary Buddy algorithm for dynamic memory allocation

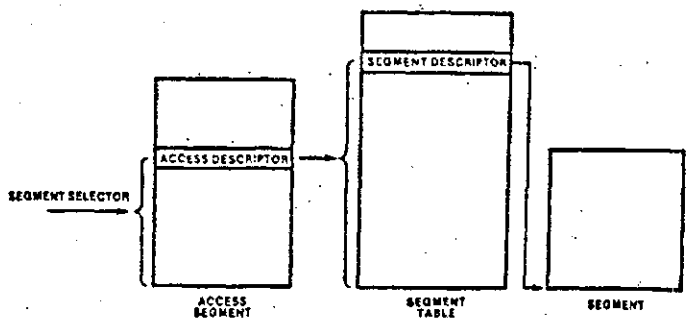


Figure 15. Two-level address mapping scheme in the i432 processor [27]

1. they provide more sophisticated protection mechanism,
2. they are able to accommodate larger address space, and
3. they provide page sharing.

Several multi-level mapping schemes are evaluated below.

Intel's i432 processor uses two-level mapping in order to provide more sophisticated protection mechanism, as shown in Figure 15 [27,40,47].

The segment selector register points to an entry of the access segment, where the access rights are stored and are thus associated with program modules. The access descriptor contains the pointer to the segment table, and finally the segment descriptor contains a pointer to the beginning of the selected segment in the primary memory. Because the access rights are stored independently of the segment descriptors, several modules can share the same segment, each with different access rights to it. In Figure 15, the module A can write and read the selected segment, while the module B can only read the segment.

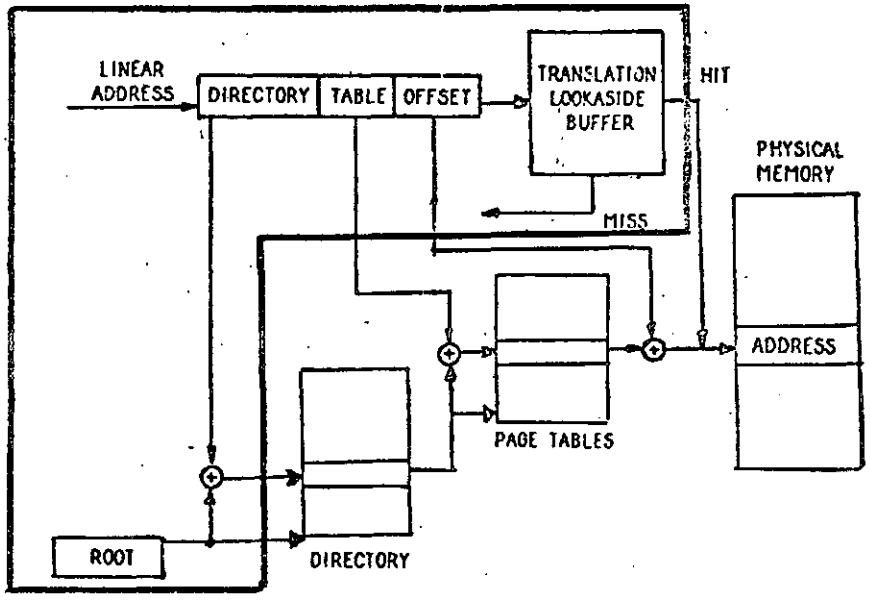


Figure 16. Paging system architecture in the i386 processor [54]

In addition, the two-level mapping scheme makes it possible to restrict the number of segments accessable by a given program. In single-level mapping systems, such as i286, any program may address any segment in memory, simply by pointing to it through the segment table.

The two-level scheme of the i432 also enables fewer address bits to point to a particular segment.

The Intel's i386 provides two options, which are user selectable: segmentation system (same as in the i286), or paging system. The paging system architecture uses two-level mapping scheme, along with a translation lookaside buffer, designed as a cache memory. The complete architecture is shown in Figure 16.

The linear virtual address consists of three fields (directory, table, offset), and address translation is performed in the following steps:

1. first, the address is searched through the TLB. If the address is found, the translation is performed in the TLB, and the primary memory is accessed directly.
2. if the address is not found in the TLB, the miss signal is generated, and the translation is performed through the two-level mapping built on the CPU chip, as shown in Figure 15.

The two-level on-chip mapping scheme enables fast address translation, and page tables can be shared and/or swapped.

A similar two-level mapping scheme has been implemented in the NS16082 MMU [6,25,38]. The total physical address space is divided into 32,768 fixed pages of 512 bytes each. The virtual address consist of 24 bits divided into three fields: index-1 and index-2 of the page selector, and the offset, as shown in Figure 17.

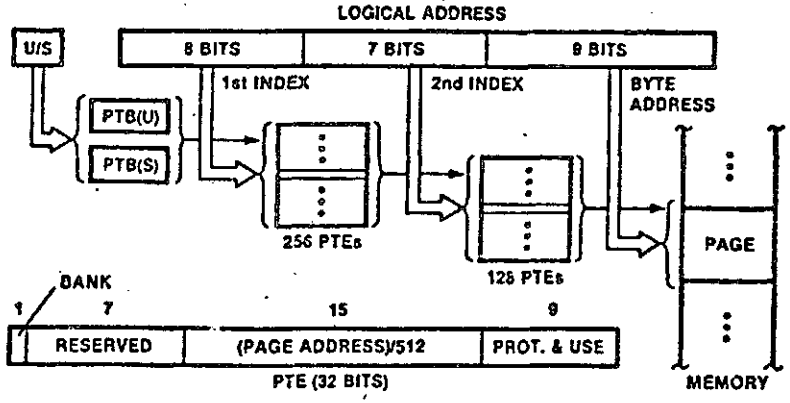


Figure 17. Two-level mapping scheme of the NS16082 MMU [38]

TABLE 2
Address translation characteristics of advanced microprocessors

PROCESSOR	ADDRESS SPACE REAL	ADDRESS SPACE VIRTUAL	ADDRESS TRANSLATION SCHEME	MAPPING LEVELS	ASSOCIATIVE CACHE
Intel 80286	16M	1G	SEGMENTATION	1	4 segment descr.reg.
Intel 432	16M	1T	SEGMENTATION	2	Assoc. cache TLB
Intel 80386	4G	64T	PAGING&SEGMENTATION	2	Assoc. cache TLB
MC68000	16M	4G	USER SELECTABLE	1	32 content-addr. reg.
MC68010 + 58451 MMU	16M	4G	USER SELECTABLE	1	32-content-addr. reg.
MC68020 + 68851 MMU	4G	?	USER SELECTABLE	1	32-content addr. reg.
Z8001 + Z8010 MMU	16M	4G	SEGMENTATION	1	64 segm. content-addr.reg.
Z8003 + Z8015 MMU	8M	4G	PAGING (page size=2K)	1	64 page descr. registers
Z800	16M	4G	PAGING (page size=4K)	1	16 addr.-accessable reg.
Z80,000	16M	4G	PAGING (page size=1K)	3	Assoc. cache TLB
NS16032	16M	4G	PAGING (page size=512)	2	32 content-addr. cache
NS32032 + 16082 MMU					
NCR/32 + NCR32101	16M	4G	PAGING (page size=1K)	1	16 associative memories
WE32100 + WE32101 MMU	4G	?	PAGING&SEGMENTATION	1	64-entry page descr. table 32-entry segm.descr.table

The index-1 (8 bits) of the page selector is used to locate one of the 256 entries of the page table. The contents of the page table PTE-1 points to the beginning of one of 256 pointer tables, each of which contains 128 entries. Then, the pointer to the pointer table is combined with the index-2 (7 bits) of the page selector, to locate one of the entries within the pointer table. The selected entry contains the actual page number in primary memory. The offset field is then used to locate data within the page. The NS 16082 MMU contains the associative cache to hold 32 recently used page address entries, as well.

The Z80,000 processor uses three-level mapping scheme based on the set of three translation tables located in primary memory [2,33,53]. It also contains an associative memory for the TLB, where 16 most recently referenced pages are stored. The CPU automatically loads the TLB from translation tables, when a logical address is missing.

The NCR/32 processor uses an address translation chip (ATC) for address translation based on paging system with one-level mapping [22]. The chip contains 16 associative memories for recently used pages.

The Z8010 MMU, which is used with the Z8001 processor, applies one-level segmentation system, based on 64 content-addressable segment descriptor registers. For more details see [33,56].

The Z8015 MMU differs from the Z8010 MMU in that the logical address is translated into page frames rather than segments. It applies one-level mapping scheme and uses 64 page descriptor registers, which are also content-addressable [33,56].

The WE32100 32-bit processor uses off-chip 32101 MMU, which supports both demand paging and segmentation systems, which are user selectable [15, 17, 18]. The MMU contains an on-chip cache memory: a 32-entry segment descriptor cache, and a 64-entry page descriptor cache, to hold recently used segment and page descriptors, respectively.

Table 2 summarizes address translation features of some 16- and 32-bit microprocessors.

4. VIRTUAL ADDRESS SUPPORT TECHNIQUES

A virtual memory system allows the user to execute programs on a very large memory of virtual address space, much larger than the actual physical memory. This is accomplished by the capability of a microprocessor to detect access to memory pages (or segments) which are not present in the physical memory. When the virtual memory system detects such a reference, it will fetch the required page from the secondary memory into the primary memory.

In order to support virtual memory capabilities, besides the address translation, a microprocessor must provide the following attributes:

1. to recognize a page or segment fault, if the page or segment is not present in the primary memory. The memory manager must then inform the processor so that the missing page or segment can be fetched from the secondary memory, and eventually one of the current pages or segments can be

replaced,

2. to abort execution of the current instruction *(instruction abort capability)*,
3. to save necessary information needed later to recover from the fault,
4. to call and execute the fault service routine in the operating system, which will swap the required page(s) or segments(s), from secondary memory to primary memory,
5. to provide necessary information for the operating system, in order to support page (or segment) placement and replacement algorithms *(indication of access activities)*, and
6. to restore the saved state and resume the normal processing *(instruction restart capabilities)*.

Although very different in complexity, all advanced microprocessors provide instruction abort and restart capabilities. Some solutions are presented below. Recognizing the access fault can be performed internally only, if the MMU is on the CPU chip, or both internally and externally, if the MMU is off the CPU chip.

When an access is made to an instruction or data which is not present in primary memory, an address error is internally detected, and it initiates the address error fault handling routine *(internally detected fault)*. If the off-chip MMU detects a fault situation, it will send a signal to the CPU, which will in turn activate the fault handling routine *(externally detected fault)*.

When the CPU recognizes an access fault, it saves the state information needed to recover from the fault. The information is usually saved on the stack. The typical information which must be saved in the program counter (starting address of the instruction), the status register, the fault address, the trap-specific parameters, the access type, the internal temporary registers, various internal statuses, etc. For illustration, the MC68010 processor which supports virtual memory, saves 26 words versus the MC68000 process, which saves only seven words, which is not enough to provide the user with the state of the machine after the fault has been occurred. Figure 18 shows the information saved on the stack for these two processors.

The MC 68010 address stack is divided into two parts: a user visible section, and a non-user visible section in which the internal status and the temporary data are saved.

The memory management unit also has to provide the information related to access activities needed by the operating system (placement and replacement algorithms). This information is usually stored in the transition table entries. There are three information bits which are present in typical systems:

1. *the valid bit* - which is controlled by the operating system, and specifies whether or not a block (page or segment) is in the primary memory.
2. *the references bit* - where the MMU typically sets this bit to indicate if access to the corresponding block in primary memory is on. The operating system may reset this bit to keep track of the access history.
3. *the modified bit* - which is set by any write operation to the corresponding

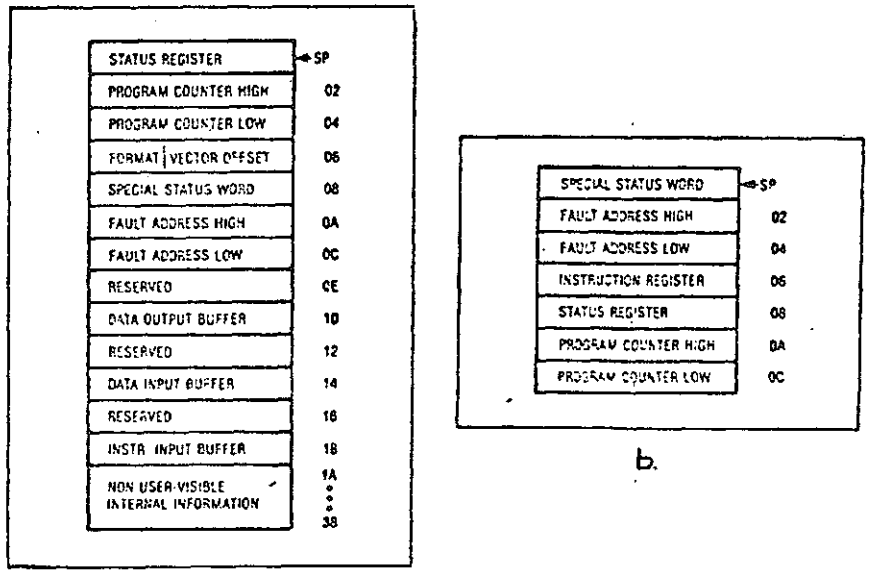


Figure 18. Address error stack [36]
a. MC68010
b. MC68000

block. This bit indicates whether the block must be written back to the secondary memory, before being replaced from the primary memory.

For illustration, the i432 processor contains four access activity bits in its segment descriptor, as shown in Figure 19.

The valid bit (V) indicates whether or not the segment is in the memory. The storage allocated bit (S) indicates whether any memory has been associated with this descriptor. The accessed bit (AC) indicates whether the segment has been accessed, while the altered bit (AL) indicates whether the information contained in the segment has been modified.

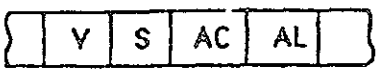


Figure 19. Access activity bits of the i432 processor contained in the segment descriptor [27]

The operating system uses V and S bits to detect when a physical segment is not present in memory, while the AC and AL bits are used by the replacement algorithm to decide which of the currently present segments should be swapped out by the new segment.

In addition, several fields in the segment descriptor can be used by the operating system to record other useful information about the segment (frequency of use, etc.).

The other advanced processors contain similar information on access activities used by the operating system. Commonly used page replacement techniques are Least Recently Used (LRU), Least Frequently Used (LFU), and First-In-First-Out (FIFO) [1,5,8,9,55]. The described information maintained by the CPU (referenced and modified bits), as well as some additional user-defined fields, can be used to design the page replacement algorithm in the operating system.

One of the popular schemes for the LRU algorithm classifies the pages into four groups:

Group 1: unreferenced (R=0) and unmodified (M=0)
Group 2: unreferenced (R=0) and modified (M=1)
Group 3: referenced (R=1) and unmodified (M=0)
Group 4: referenced (R=1) and modified (M=1)

The pages from the lowest groups are replaced first, and the pages from the highest groups are replaced last. The referenced bit is set by the CPU whenever the page is referenced. The operating system (OS) periodically clears the referenced bit. A sophisticated LRU algorithm, "software caching," has been implemented in the VAX/VMS operating system [31]. The LFU algorithm can also be incorporated into this scheme. Whenever the referenced bit is cleared, the OS can count the frequency with which the pages were used. The modified bit is set by the CPU whenever the page is written. When the page is swapped, the OS checks this bit to see if there is a need to update the copy of the page in the secondary memory.

The last attribute of a processor to support virtual memory is the most complex, and refers to reloading of the state of the program, and resuming the operation, after the address fault routine is completed. Two methods of implementing the resume operation on a processor are:

1. instruction restart method, and
2. instruction continuation method.

Advantages and drawbacks of these two methods are discussed in the following two subsections.

4.1 Instruction restart method

In this method, after the address fault error handling routine has completed all activities, the instruction in which fault occurred is restarted from the beginning. Figure 20 illustrates the execution of the microcode in the case when no address fault is present (Fig. 20a), and in the case when the restart method is applied, with an address fault occurred (Fig. 20b).

In Figure 20 it is assumed that a machine instruction consists of several microinstructions [m1, m2, m3, m4]. If there is no address fault, these instructions will execute sequentially, as shown in Fig. 20a. If the MMU detects an address fault in the microinstruction m2, the control will be transferred to the address error routine. The address error routine will first save the information state, and then the routine will handle the address error (the required page or segment will be fetched from the secondary memory). Finally, the saved information state will be restored, and the faulted instruction will be restarted from the beginning - at the machine instruction level. Therefore, the sequence [m1, m2, m3, m4] will be executed again.

The main problem in the instruction restart method is that the processor must reconstruct the state of the machine, as it was at the beginning of the machine instruction, while he faulted instruction was interrupted in the middle of its execution. There are some situations when this is very complex, such as when a resource is used both as input and output parameter in the same instruction. For example, in extended precision arithmetic operations, a carry (or borrow) bit from the previous operation is used in the instruction as an input parameter, but the instruction itself also sets the same bit as the result of the current operation. If the address fault is detected after this bit is updated, the original value must be restored before the instruction is restarted. A similar case is with autoincrement and autodecrement addressing modes.

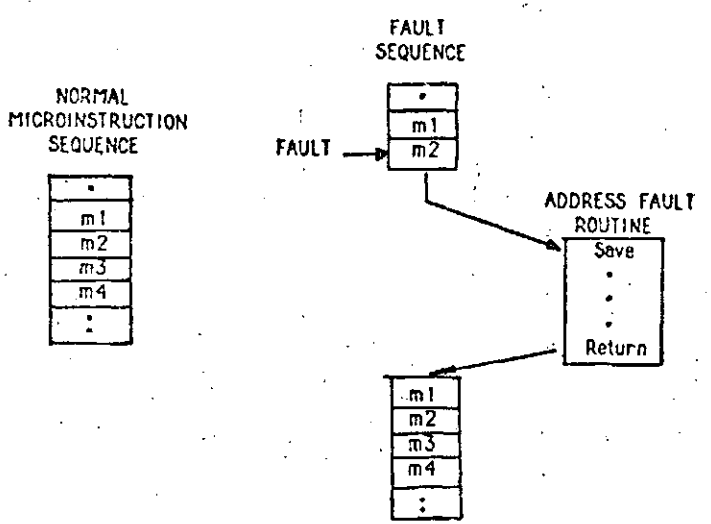


Figure 20. Microinstruction sequence [36]
a. No address fault
b. Instruction restart method

Several techniques have been proposed to solve this problem, and are discussed below:

1. The processor may postpone the modification of user-visible resources (such as carry bit), until the end of the instruction. Then, if the address fault has not occurred, the resources will be updated.
2. All modifications of the user-visible resources will be recorded by the processor if the address fault occurs. On the basis of this information, the processor will be able to restore the original values of the modified resources.
3. The processor maintains the copies of all user-visible resources, that are modified. Because the copy always contains the original value, if the address fault occurs, it will be easy to restore the original state.

4.2. Instruction continuation method

In the instruction continuation method, when the address error routine has been completed, the machine instruction will not be resumed from the beginning, but from the same location within the instruction at which the execution was suspended. The execution of the same sequence of microinstructions (m1, m2, m3, m4), in the case of the continuation method, is shown in Figure 21.

The address fault was detected in the microinstruction m2, and the control was transferred to the address error handling routine. After the routine has been completed, the processor will resume operation, by executing the microinstruction m3. The

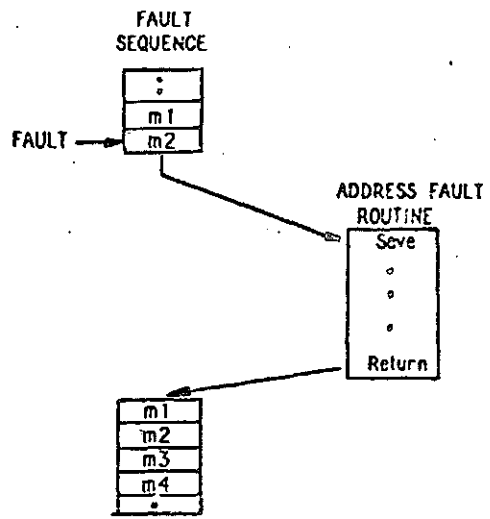


Figure 21. Microinstruction execution - the instruction continuation method [36]

continuation method is analogous to an interrupt operation at the microinstruction level.

In order to support the instruction continuation method, the processor must be able to save the entire state of the machine, when an address fault is detected. Therefore, the processors which apply this method usually have a large address error stack, to save all necessary information (e.g. MC68010). Regardless of this requirement, another problem with the continuation method is related to the instructions that require execution without interruption. In addition, this method requires the additional time and silicon resources for saving and restoring the complete state of the machine.

The instruction continuation method has been implemented in the MC68010 and the MC68020 processors only [35,36], while all other advanced processors use the instruction restart method.

The NS16082 MMU sends an abort signal to the CPU (NS 16032 or 32032), which will stop the execution and will return the CPU into the state before the aborted instruction. Then, all needed information (contained in program counter, machine status, stack pointer, and several other registers) is automatically saved. When the address fault routine is completed, a return-from-trap instruction is executed, which will resume the aborted instruction from the beginning [38].

Zilog processors also implement the instruction restart method. The Z8001/Z8015 system contains a special data count register which counts the number of successful data accesses before an address fault. This information is used to restore the machine state, which existed before the address fault.

The Z80,000 and Z800 processors, which have the MMU on the CPU chip, apply an improved instruction restart method compatible with their pipelining architecture. The Z80,000 executes instructions by using six-stage pipelining, and therefore the page fault can be detected before memory access. The address translation is performed in the third stage of the pipeline, and if an address fault is detected, the execution stage will be suspended, before any change of register contents is made [33,36]. The Z800 applies a similar technique, because it has a three stage pipeline allowing the instruction suspension, before any register is changed.

Intel processors i286 and i386 apply the instruction restart method, as well [26,28,54]. They are also able to detect an address fault before executing instruction, and thus faulted instruction restart becomes simple. After completing the execution of the address fault handling routine, the CPU places the address of the interrupted instruction into the instruction pointer, and resumes the program execution.

5. PROTECTION AND SECURITY TECHNIQUES

In multitasking and multiuser environments, it is required from processor architecture to support protection and security, in order to increase system performance and simplify system implementation. Basically, protection and security issues can be divided into the following topics:

1. memory protection,

2. program protection,
3. user protection, and
4. information security.

Memory protection mechanism should detect any addressing error before it caused damage. Each instruction should be checked to verify that it performs the intended operation. The MMU unit performs this check, and if there is an address error detected, it generates an address fault. The address fault handling routine is then activated, which analyzes the address error, eventually fixes it, and returns to the interrupted program. *Program protection mechanism* should prevent application program from making illegal modifications of the operating system. It also should control the transfer between system modules to achieve total reliability. *User protection mechanism* should protect users against each other. *Security mechanism* should provide limited access to information.

Two basic architectures that provide program and user protection are:

1. hierarchical protection system, or *ring protection system*, and
2. non-hierarchical protection system, or *capability-based protection system*.

These two systems are discussed in the following paragraphs.

Hierarchical protection system consists of a hierarchy of protection levels, or rings, starting from the most privileged to the least privileged. Basic principles of the ring system are:

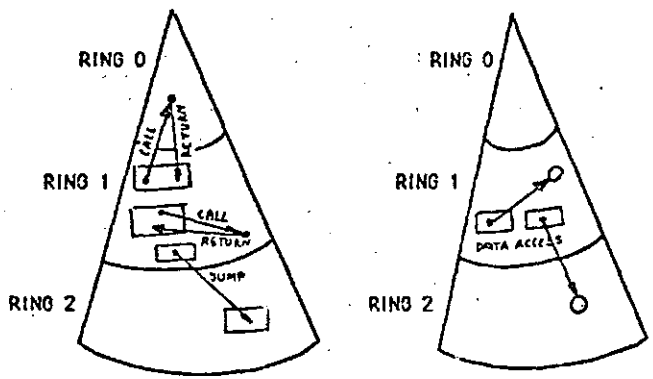


Figure 22. Principles of ring protection system [28]
a. control transfer between programs
b. data access

1. A program may access only data that reside on the same ring, or a less privileged ring,
2. A program may cal services that reside on the same, or a more privileged ring.

These two protection approaches are illustrated in Figure 22.

The ring system has been implemented in the i286 and the i386 processors [21,26,28,54]. Their ring protection system consists of four privilege rings, as shown in Figure 23.

Different priorities are assigned to different programs (segments) within the system. Greater privilege is assigned to more important programs. Typically, the operating system occupies the most-privileged ring, thus it is protected from the application programs. The programs may access the OS with a high-speed call instruction, rather than using the context switching technique, which is the traditional way to implement the call of OS services.

Second and third rings are typically used for system services and custom extensions, respectively, while the application programs are usually located at the least-privileged ring.

The i286/i386 protection model also provides task isolation, by having separate descriptor tables. The entire isolation between rings is provided by a separate stack for each ring.

In non-hierarchical protection systems (or capability-based protection systems), for each task a table of operations is defined. This table of operations specifies operations that may affect other tasks in the system. In order to perform an operation

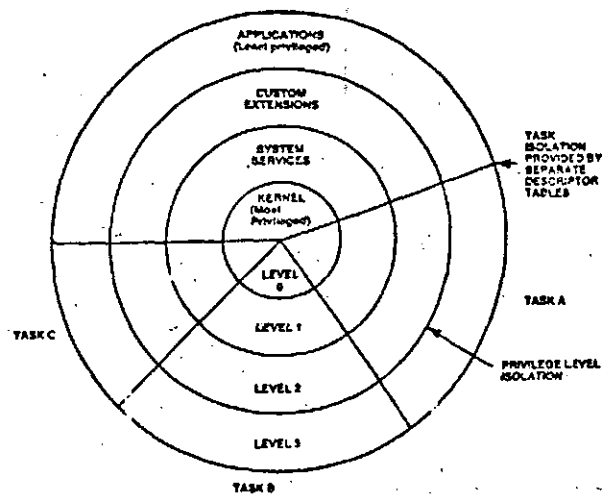


Figure 23. The ring protection system of the i286/i386 processors [28]

which could affect another task, a task must have the corresponding capability in its table of operations.

The capability-based protection system is more complex, and the current processors still do not implement it in the architecture, but in the operating system. The current processors provide some protection features, which can be used when designing a sophisticated protection system in software [3,4,5,25,35].

The MC68000, the Z8000, and the NS 16000 processors have two operating modes (or privilege levels) of the CPU: supervisor mode, and user mode. In the supervisor mode, the CPU can execute the complete set of instructions, while in the user mode, only a subset of instructions can be used. In Zilog processors, these two modes are called system and normal.

Typically, the operating system functions are placed at the supervisor level, while application programs execute at the user level, thus the operating system is protected from the application programs. The supervisor level typically has access to all of the processor resources, as well as to all external resources, such as memory and I/O. This enables the operating system to control both processor and external functions.

In addition, the NS16000 processors provide separate address spaces for each running process, thus protecting one user from another.

The MC68020 implements a concept of multiple access levels, which provides expansion on up to 256 hierarchical levels, which present a superset of ring architecture.

Security refers to the limited access to information. The basic principle is to allow a program to access only what it needs to know. For example, Linden suggests that "...almost every procedure should run in a protection domain that gives it an access to exactly what it needs to accomplish its function, and nothing more [32]." The security is provided by giving each process certain access rights to a page or a segment. The most commonly used access rights are:

1. *read access:* a process may obtain any information from the page or the segment.
2. *write access:* a process may modify the page or the segment, and may place additional information in it. The process may destroy all of the information in the page or the segment.
3. *execute access:* a process may run the page or the segment as a program. Execute access is given to pages or segments which are programs, and denied to data pages or segments.

Current processors typically store the access rights in page or segment descriptors. Before the processor accesses a page or a segment, it first checks its access rights, and if they are verified, it may access the selected page or segment. The diagram in Figure 24 illustrates the described mechanism, based on access rights stored in the page or segment descriptors. The character N indicates that the corresponding page or segment cannot be accessed at all.

The segmentation virtual memory system provides a more natural security system in a paging system. The logical address space is divided into pages, and the described mechanism cannot protect the program modules precisely. It either protects too little or too much. In the segmentation system, each segment is of specific length, and the way to protect segments by using access rights is more natural.

Regardless of the implemented virtual memory system, the drawback of the described security mechanism is that all users have the same access rights to common pages or segments, because the access rights are associated with the pages or the segments, and not with the users.

This problem can be solved by using two-level mapping scheme, as described in Section 3.2, for the case of the i432 processor [27]. In this two-level mapping scheme, the access rights are stored independently of the segment (or page) descriptors, and are associated with the users, and not with the segments (or pages).

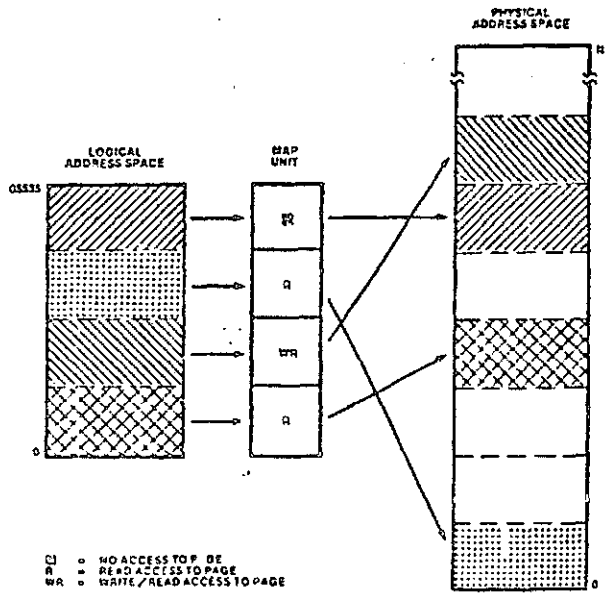


Figure 24. Security technique based on access rights stored in the page or segment descriptors [27]

6. DISCUSSION AND CONCLUSION

We have discussed in this paper several issues related to memory management in advanced microprocessors. All these concepts are not new; they are known for years from the operating system's theory and practice, however the approaches are sometimes modified, and implementation techniques may be different, in comparison with the minicomputer and mainframe environments.

The processor architect must make several crucial decisions related to the processor architecture, which have to support memory management and virtual memory. The main decisions to be made are listed in Figure 25.

The on-chip MMU versus off-chip MMU is one of the basic decisions which has to be made. Both concepts have advantages, as well as drawbacks. These have been discussed in the paper. In addition, the on-chip MMU has an advantage over the off-chip MMU which is related to cache memory design. An external MMU requires logical address caches to bypass the MMU delay, while the internal MMU implements the physical address cache. The logical address cache requires special address tag hardware, large operating system overhead on task switch, and flush cache when sharing data.

The issue related to virtual memory system: paging versus segmentation, is of crucial importance. Again, some microprocessors support paging, other processors support segmentation, while few microprocessors support both systems, in which case the mode is user selectable. Anyhow, it seems that the paging system has advantages over the segmentation system, and almost all 32-bit microprocessors support it.

The next two questions are related to the implementation of the address translation mechanism: levels of mapping and use of an associative cache memory. Both multi-level mapping and a small associative cache memory significantly improve system performance, and thus they should be built into an advanced microprocessor architecture. Practically, all 32-bit microprocessors have implemented these two concepts in their architecture.

MMU ON-CHIP	versus	MMU OFF-CHIP
PAGING	versus	SEGMENTATION
ONE-STAGE MAPPING	versus	MULTI-STAGE MAPPING
CACHE MEMORY-YES	versus	CACHE MEMORY-NO
INSTRUCTION RESTART	versus	INSTRUCTION CONTINUATION
PROTECTION BUILT-IN	versus	PROECTION IN SOFTWARE

Figure 25. A list of questions for the processor architect

Techniques to support the virtual memory system, especially the choice of the techniques to implement resume operation, after an address fault is detected and corrected, is also an important decision for the architect. The instruction restart method seems to be more efficient than the instruction continuation method, especially if the MMU is on the CPU chip. Then, due to pipelined nature of architectures in modern microprocessors, the address fault can be detected before a memory access. This significantly simplifies the restart of the faulted instruction.

Finally, the protection mechanism built in the architecture (such as the ring system in the i286/i386 processors) provides a powerful tool for an operating system designer, and reduces software overhead. On the other hand, because the protection system is already defined in the architecture, there is no choice for the OS designer, but to implement the available mechanism, whether he (she) likes it or not.

The other approach, in which the processor provides some basic protection elements, but not the whole protection system (such as supervisor/user modes and access concepts in the MC68020), requires from the OS designer to create the protection system in software, thus increasing the software overhead. However, this approach is more flexible.

We may conclude that the memory management architectures in current microprocessors are coming of age. However, one of the most challenging aspects of future processor design will be to provide more elegant solutions to all these problems, as well as to enable a more complete integration of memory management and virtual memory support.

7. ACKNOWLEDGEMENT

The authors are thankful to Jeff Pridmore and Walt Helbig, of RCA, for their comments.

8. REFERENCES

1. Aho, A.V., Denning, P.J., and Ullman, J.D., "Principles of Optimal Page Replacement," *JACM*, Vol. 18, No. 1, January 1971, pp. 80-93

2. Alpert, D., "Powerful 32-bit Micro Includes Memory Management," *Computer Design*, October 1983, pp. 213-220.

3. Alpert, D., Carberyy, D., Yamamura, M., Chow, Y., and Mak, P., "32-bit Processor Chip Integrates Major System Functions," *Electronics*, July 14, 1983, pp. 113-119.

4. *"An Architectural Contrast: The M68000 Microprocessor Family and the 8086/iAPX 286,"* Motorola Corp., November 1983.

5. Andrews, R., "The Z80,000 Processor Chip Integrates Major System Functions," *Proceedings of the IEEE Mini/Micro Southeast*, Orlando, Florida, January 1984, pp. 5.2.1-5.2.7.

6. Bal, S., et al, "The NS16000 Family - Advances in Architecture and Hardware," *IEEE Computer*, June 1982, pp. 58-67.

7. Beyers, J.W., et al, "A 32-bit VLSI CPU Chip," *IEEE Journal of Solid-State Circuits*, Vol. 16, October 1981, pp. 537-541.

8. Chamberlin, D.D., Fuller, S.H., and Lin, L., "An Analysis of Page Allocation Strategies for Virtual Memory Systems," *IBM Journal of R&D*, Vol. 17, 1973, pp. 404-412.

9. Chu, W.W., and Opderbeck, H., "Program Behaviour and the Page-Fault-Frequency Replacement Algorithm," *IEEE Computer*, November 1976, pp. 29-38.

10. Deitel, H.M., *"An Introduction to Operating Systems,"* Addison-Wesley Publishing Company, 1984.

11. Denning, P.J., "Virtual Memory," *ACM Computing Surveys*, Vol. 2, No. 3, September 1970, pp. 153-189.

Denning, P.J., "Working Sets Past and Present," *IEEE Transactions on Software Engineering*, Vol. 6, No. 1, January 1980, pp. 64-84.

13. Denning, P.J., and Schwartz, S., "Properties of the Working-Set Model," *Communications of the ACM*, Vol. 15, No. 3, March 19872, pp. 191-198.

14. Dennis, J.B., "Segmentation and the Design of Multiprogrammed Computer Systems," *JACM*, Vol. 12, No. 4, October 1965, pp. 589-602.

15. Diodato, P.W., et al, "CAD Construction of a VLSI Memory Management Unit," *Proceedings of the ICCAD*, 1983.

16. Doran, R.W., "Virtual Memory," *IEEE Computer*, October 1976, pp. 27-37.

17. Goksel, A.K., et al, "A Memory Management Unit for a Second Generation Microprocessor," *Proceedings of the Compcon*, 1984.

18. Goksel, A.K., et al, "A VLSI Memory Management Chip: Design Considerations and Experience," *IEEE Journal on Solid-State Circuits*, Vol. 19, No. 3, June 1984.

19. Gupta, A., and Toong, H.D., "An Architectural Comparison of 32-bit Microprocessors," *IEEE Micro*, Vol. 3, No. 1, February 1983, pp. 9-22.

20. Hansen, D.J., "Programming Motorola's 32-bit Microprocessor the 68010," *Proceedings of the IEEE Mini/Micro Southeast*, Orlando, Florida, January 1984, pp. 4.1.1-4.1.9.

21. Heller, P., "The Intel iAPX 286 Microprocessor," *Proceedings of the Wescon*, 1981, pp. 1.3.1-1.3.4.

22. Heller, P., Childs, R., and Slager, J., "Memory Protection Moves Onto 16-bit Microprocessor Chip," *Electronics*, Vol. 55, Feb. 24, 1982, pp. 133-137.

23. Hirschberg, S.D., "A Class of Dynamic Memory Allocation Algorithms," *Communications of the ACM*, Vol. 16, No. 10, October 1973, pp. 615-618.

24. Hoeschene, H.A., et al, "A Second Generation 32-bit CMOS Microprocessor," *Proceedings of the Compcon*, 1984.

25. Hunter, C.B., and Farquhar, E., "Introduction to the NS16000 Architecture," *IEEE Micro*, April 1984, pp. 26-47.

26. *"IAPX 286 Operating Systems Writer's Guide,"* Intel Corporation, Santa Clara, 1983.

27. *"Introduction to the iAPX 432 Architecture,"* Intel Corporation, Santa Clara, 1981.

28. *"Introduction to the iAPX 286,"* Intel Corporation. Santa Clara, 1982.

29. Kaminker, A., et al, "A 32-bit Microprocessor with Virtual Memory Support," *IEEE Journal of Solid-State Circuits*, October 1981, pp. 230-231.

30. Knowlton, K.C., "A Fast Storage Allocator," *Communications of the ACM*, Vol. 8, No. 10, October 1965, pp. 623-625.

31. Levy, H.M., and Lipman, P.H., "Virtual Memory Management in the VAX/VMS Operating System," *IEEE Computer*, March 1982, pp. 35-41.

32. Linden, T.A., "Operating System Structures to Support Security and Reliable Software," *ACM Computing Surveys*, Vol. 8, No. 4, Dec. 1976, pp. 419.

33. Look, H., "Virtual Memory for Zilog's 8-, 16-, and 32-bit Microprocessors," *Proceedings of the IEEE Mini/Micro Southeast*, Orlando, Florida, January 1984, paper 3.3.

34. MacGregor, D., "Hardware and Software Strategies for the MC68020," *EDN*, June 20, 1985, pp. 89-98.

35. MacGregor, D., Mothersole, D., and Moyer, B., "The Motorola MC68020," *IEEE Micro*, August 1984, pp. 101-118.

36. MacGregor, D., and Mothersole, D.S., "Virtual Memory and the MC68010," *IEEE Micro*, June 1983, pp. 24-38.

37. Martin, G., "Virtual Memory Management Expands Microprocessors," *Computer Design*, June 1983, pp. 169-178.

38. Mateosian, R., "Elegance is Everything in NS 16000 Memory Management," *Proceedings of the IEEE Mini/Micro Southeast*, Orlando, Florida, January 1984, paper 3.2.

39. Mateosian, R., "Operating System Support - the Z8000 Way," *Computer Design*, May 1982, pp. 255-261.

40. Mazor, S., Wharton, S., "Compact Code iAPX 432 Addressing Techniques," *Computer Design*, May 1982, pp. 249-253.

41. Mazor, S., Wharton, S., "Promote User Privacy Through Secure Memory Areas," *Computer Design*, October 1982, pp. 89-92.

42. *"MC68020 32-bit Microprocessor User's Manual,"* Prentice-Hall, 1984.

43. Myers, G.J., *"Advances in Computer Architecture,"* John Wiley & Sons, 1978.

44. Philips, D., "Memory-Management Strategies Suit Different Application Areas," *EDN*, September 1984, pp. 135-143.

45. Peterson, J.L., Theodore, N., "Buddy Systems," *Communications of the ACM*, June 1977, Vol. 20, No. 6, pp. 421-431.

46. Pohn, A.V., and Smay, T.A., "Computer Memory Systems," *IEEE Computer*, October 1981, pp. 93-110.

47. Pollack, F. J., et al, "Supporting ADA Memory Management in the iAPX-432," *ACM* 1982, pp. 117-130.

48. Purdom, P.W., and Stigler, S.M., "Statistical Properties of the Buddy System," *Journal of the ACM*, Vol. 17, No. 4, October 1970, pp. 683-697.

49. Saltzer, J.H., and Schroeder, M., "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, Vol. 63, No. 9, September 1975, pp. 1278.

50. Skoog, S.K., "Memory Management with the NCR/32 Scipset," *Proceedings of the IEEE Mini/Micro Southeast*, Orlando, Florida, January 1984, paper 3.1.

51. Stockton, J.F., "A Virtual Breakthrough for Micros, *Computer Design*, pp. 153-162.

52. Stockton, J.F., "The M68451 Memory Management Unit," *Electronic Engineering*, Vol. 54, May 1982, pp. 54-73.

53. Timms, B., "Z80,000 Mainframe Resources Optimize the Software Environment," *Proceedings of the IEEE, Mini/Micro Southeast*, Orlando, Florida, January 1984, pp. 4.4.1-4.4.13.

54. *"Touch the Future,"* Intel Design Seminar, Miami, Florida, 1985.

55. Turker, R., and Levy, H., "Segmented FIFO Page Replacement," *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems*, Las Vegas, Nevada, September 1981, pp. 48-51.

56. Walters, S., "Memory Management Made Easy with the Z8000," *Proceedings of the Wescon*, 1981, pp. 9.3.1-9.3.9.

9. ABOUT THE AUTHORS

Dr. B. P. Furht is on the faculty of the Department of Electrical and Computer Engineering, University of Miami, Coral Gables, Florida. He has published over 60 technical papers, and 2 books. He is the author of *Microprocessor Interfacing and Communication* (Reston 1985), and coeditor of the *Tutorial on Advanced Topics in Computer Architecture* (IEEE Press, 1985). His current research activities include high-level language computer architectures, multiprocessor systems, and architectures for virtual memory management. He presented over 30 invited lectures in Europe, North, and Latin America on various topics related to computer architecture. He has been involved in consulting activities for a number of companies such as NASA, RCA, Cordis, Honeywell, and others. He is a member of the IEEE, and a chief editor of the International Journal of Mini and Microcomputers.

Dr. V. M. Milutinović is on the faculty of the School of Electrical Engineering, Purdue University. He has published over 60 technical papers, 2 original books, and 4 edited books. His research papers have been published in IEEE Transactions, IEE Proceedings, IEEE Computer, and other refereed journals. One of his books has been republished (in various forms) in several languages. He is the editor of the IEEE Press *Tutorial on Advanced Microprocessors and High-Level Language Computer Architecture*, and the coeditor of the IEEE Press *Tutorial on Advanced Topics in Computer Architecture*. He is the editor and the contributing author for two multiauthor books on computer architecture. His pioneering paper on GaAs computer architecture for VLSI has been scheduled to appear in the September issue of IEEE Computer. He presented over 40 invited lectures in Europe, North, and Latin America. His current interests include VLSI computer architecture for GaAs, high-level language computer architecture, and microprocessor systems for AI. His current research support is equal to about $250k per year, predominantly in the area of VLSI computer architecture for GaAs. He has consulted for a number of high-tech companies, including Intel, Honeywell, NASA, RCA, and others. He is currently involved in the industrial implementation of a 32-bit VLSI microprocessor in the GaAs technology, with responsibilities in the microarchitecture domain. He is a member of the IEEE, and is on the EUROMICRO Board of Directors.

Janja Žabjek-Dolinšek
Iskra Avtomatika, Ljubljana

UDK: 519.852

Operativno načrtovanje gibanja vlakov zahteva v nekaj minutah izdelavo načrtov za dve do štiri ure vnaprej,po katerem naj bi se gibali na optimalno možen način.Kriterij optimalnosti je najmanjša izguba časa vlaka,pomnožena s koeficientom cene pri srečanju in prehitevanju vlakov.Naloga določanja optimalnih odločitev spada v večvariantno kombinatoriko.

THE OPERATIVE PLANNING OF TRAIN MOVEMENT

Plans for train movement have to be made in a few minutes for two to four hours ahead in the optimal possible way.The criterion of optimality is the smallest loss of train time multiplied by the coefficient of train price.The problem of determining optimal decisions belongs to multivariate combinatorics.

1.UVOD

V avtomatizaciji železniškega prometa igra pomembno vlogo daljinsko vodenje vlakov.Sistemi,ki se danes uporabljajo za daljinsko vodenje in nadzor železniškega prometa v realnem času ,so zelo veliki in kompleksni,saj morajo opravljati zahtevne funkcije zajemanja,prenosa,obdelave in predstavitve podatkov,kot tudi funkcije povezave operater-sistem.

V preteklosti so se tovrstni sistemi za manj zahtevne aplikacije praviloma realizirali samo s klasičnimi telemehanskimi napravami,ki so za zahtevnejše naloge vodenja neprimerni.

S prodorom sodobne tehnologije in cenenih mikroprocesorskih enot,pa se na tem področju vse bolj uveljavljajo mikroračunalniški teleinformacijski sistemi,ki v kombinaciji s sistemi vlakovnih številk in avtomatskim postavljanjem voznih poti omogočajo enostavno in racionalno reševanje vseh nalog daljinskega vodenja železniškega prometa.

Nadgradnja tega celotnega sistema daljinskega vodenja vlakov pa je avtomatsko razreševanje konfliktnih situacij,oziroma operativno planiranje gibanja vlakov,ki mora biti brezkonfliktno in optimalno.To delo zdaj opravlja človek-dispečer.

Dokler vlaki vozijo po voznem redu,nima dispečer nobenega dela;čim pa eden od vlakov zamuja,lahko pride do konfliktne situacije z ostalimi vlaki na odseku proge.Tedaj je potrebna pomoč dispečerja,da nastale konfliktne situacije reši.

Torej je delo dispečerja na železniški progi orientirano predvsem na vozni red.

V trenutku,ko vlaki odstopajo od voznega reda,je treba hitro najti rešitev za novo gibanje vseh vlakov v novih,spremenjenih pogojih.Vendar pa v primeru gostega prometa in pri sedanjih hitrostih vlakov človek ne more več dela v realnem času brez pomoči računalnika.Obstaja vrsta načinov in algoritmov za razreševanje takih konfliktnih situacij,ideja pa je v tem,da poiščemo najoptimalnejšo rešitev.

Uporaba modernih metod matematične optimizacije ni mogoča brez matematične opredelitve posameznih nalog,ki nastopajo pri gibanju vlakov.Za nastale konfliktne situacije torej konstruiramo matematični model.

2.KONFLIKTNE SITUACIJE so:

- -srečanje vlakov
- -prehitevanje vlakov
- -začasen izpad odsekov ali sprememba dvotirne proge v enotirno
- -prepoved postanka vlaka na postaji
- -če je na postaji več vlakov kot tirov
- -če sta na istem odseku proge dva ali več vlakov.(To pomeni ,če je pri prihodih in odhodih vlakov premajhen časovni presledek.)

Take konfliktne situacije lahko opisemo z matematičnim modelom.Pa začnimo.

3.METODE REŠEVANJA:

- -Globalno optimalno rešitev poiščemo z metodo DISJUNKTNEGA LINEARNEGA PROGRAMIRANJA.

-Lokalno rešitev pa dobimo s pomočjo empiričnih metod,ki hitreje poiščejo rezultat na račun odstopanja od optimalnosti.Ena takih metod je t.i. 'metoda razčlenjeno vezanih rešitev',ki smo jo razširili,tako da je uporabna tudi v primeru,ko gre za dvotirno magistralno progo z APB odseki in dodatnimi tehnološkimi omejitvami.

Članek opisuje prvo metodo.

4.KONSTRUKCIJA MATEMATIČNEGA MODELA

Opišimo najprej spremenljivke,ki nastopajo v matematičnem modelu:

NPOS ...število vseh postaj na odseku proge
NPOS-1 ...število odsekov med postajami
NVLAK ...število vlakov na odseku proge
PRIH(I,J)...čas prihoda vlaka J na postajo I
ODHOD(I,J)..čas odhoda vlaka J s postaje I
$t^{p}(i,I)$ ...prihod,ki vsebuje prisilne postanke
$t^{o}(I,I)$ ...odhod,ki vsebuje prisilne postanke
TV(I,J) ...čas,ki ga vlak J porabi za prevoz odseka I
TP(I,J) ...minimalni čas postanka vlaka J na postaji I
X(I,J) ...čas prisilnega postanka vlaka J na postaji I zaradi konflikta
K(J) ...koeficient relativne cene časa vlaka J
TEK(I,J) ...odpravni čas z začetne postaje I za vlak J
TEK(npos,Y).odpravni čas z začetne postaje NPOS za vlak Y
T ...časovni zaostanek za vlak,ki vozi zadaj

Vlak I vozi v smeri ----> :

a)čas prihoda vlaka I na postajo i

$$t^{p}(i,I)=\sum_{k=1}^{i-1}TV(k,I)+\sum_{k=1}^{i-1}TP(k,i)+\sum_{k=1}^{i-1}X(k,I)+TEK(1,I) \qquad (1)$$

b)čas odhoda vlaka I s postaje i

$$t^{o}(i,I)=\sum_{k=1}^{i-1}TV(k,I)+\sum_{k=1}^{i}TP(k,I)+\sum_{k=1}^{i}X(k,I)+TEK(1,I) \qquad (2)$$

Vlak Y vozi v smeri <-----:

a)čas prihoda vlaka Y na postajo i

$$t^{p}(i,I)=\sum_{k=i}^{NPOS-1}TV(k,Y)+\sum_{k=i+1}^{NPOS}TP(k,Y)+\sum_{k=i+1}^{NPOS}X(k,i)+TEK(NPOS,I) \qquad (3)$$

b)čas odhoda vlaka Y s postaje i

$$t^{o}(i,Y)=\sum_{k=i}^{NPOS-1}TV(k,Y)+\sum_{k=i}^{NPOS}TP(k,Y)+\sum_{k=i}^{NPOS}X(k,i)+TEK(NPOS,I) \qquad (4)$$

Torej se vlak I nahaja na postaji i v času $t^{po}(i,I)$:

$$t^{p}(i,I) <= t^{po}(i,I) <= t^{o}(i,I) \qquad (5)$$

Vlak I pa se nahaja na odseku i v času $t^{odsek}(i,I)$:

$$t^{o}(i,I) <= t^{odsek}(i,I) <= t^{p}(i+1,I) \qquad (6)$$

Podobno lahko sklepamo tudi za vlak Y.
Poglejmo,kako zgradimo matematični model za razrešitev najpreprostejših konfliktnih situacij!

a)NASPROTNI VLAKI SE NE SMEJO OB ISTEM ČASU NAHAJATI NA ENOTIRNIH ODSEKIH

1)če se je vlak I odpravil s postaje i in zavzema odsek i,je potrebno

-da se vlak Y v tem času nahaja na postaji I

ali

-da se vlak Y v tem času sploh ne sme odpraviti s postaje i+1.

To pomeni v neenačbah:

$$t^{o}(i,I) > t^{p}(i,Y) \qquad (7)$$

ali

$$t^{o}(i,I) < t^{o}(i+1,Y) \qquad (8)$$

Po drugi strani pa velja naslednje:

2)če se je vlak Y odpravil s postaje i+1 in zavzema odsek i,je potrebno:

-da se vlak I v tem času nahaja na postaji i+1

ali

-da se vlak I v tem času ne sme odpraviti s postaje i

To pomeni v neenačbah:

$$t^{o}(i+1,Y) >= t^{p}(i+1,I) \qquad (9)$$

ali

$$t^{o}(i+1,Y) < t^{o}(i,I) \qquad (10)$$

Če v neenačbah (7),...,(10) upoštevamo enačbe (1),...,(4) in jih preuredimo tako,da so neznane količine,to so prisilni postanki vlakov X(i,J), na levi strani neenačb,dobimo

$$\sum_{k=1}^{i}X(k,I)-\sum_{k=i+1}^{NPOS}X(k,Y) > PRIH(i,Y)-ODH(i,I) \qquad (11)$$

$$\sum_{k=1}^{i}X(k,I)-\sum_{k=i+1}^{NPOS}X(k,Y) < ODH(i+1,Y)-ODH(i,I) \qquad (12)$$

$$\sum_{k=i+1}^{NPOS}X(k,Y)-\sum_{k=1}^{i}X(k,I) > PRIH(i+1,I)-ODH(i+1,Y) \qquad (13)$$

$$\sum_{k=i+1}^{NPOS}X(k,Y)-\sum_{k=1}^{i}X(k,I) < ODH(i,I)-ODH(i+1,Y) \qquad (14)$$

pri čemer smo upoštevali naslednje

$$PRIH(i,Y) = t^{p}(i,Y) - \sum_{k=i+1}^{NPOS} X(k,Y) \qquad (15)$$

$$ODH(i+1,Y) = t^{o}(i+1,Y) - \sum_{k=i+1}^{NPOS} X(k,Y) \qquad (16)$$

in

$$PRIH(i,I) = t^{p}(i,I) - \sum_{k=1}^{i} X(k,I) \qquad (17)$$

$$ODH(i,I) = t^{o}(i,I) - \sum_{k=1}^{i} X(k,I) \qquad (18)$$

Ce analiziramo enačbe (11),...,(14)vidimo,da je pogoj (13) ostrejši od (12) in (11) ostrejši od (14),zato upoštevamo samo (11) in (13).Pogoj za nasprotnosmerne vlake je

$$\sum_{k=1}^{i} X(k,I) - \sum_{k=i+1}^{NPOS} X(k,Y) > PRIH(i,Y) - ODH(i,I) \qquad (19)$$

ali

$$-\sum_{k=i+1}^{NPOS} X(k,Y) + \sum_{k=1}^{i} X(k,I) <= PRIH(i+1,I) + ODH(i+1,Y) \qquad (20)$$

Pogoj za nasprotnosmerne vlake pa lahko zapišemo tudi takole:

$$t^{o}(i,I) <= t^{o}(i,J)+T \Rightarrow t^{p}(i+1,I) <= t^{p}(i+1,J)+T \qquad (21)$$

ali

$$t^{o}(i,J) <= t^{o}(i,I)+T \Rightarrow t^{p}(i+1,J) <= t^{p}(i+1,I)+T \qquad (22)$$

b)ISTOSMERNI VLAKI MORAJO NA ODSEKU UPOSTEVATI DOLOCENI PRESLEDEK V GIBANJU

1)pogoji za istosmerni prihod vlakov na postajo

$$t^{p}(i+1,J) - t^{p}(i+1,I) >= T(J) \qquad (23)$$

ali

$$t^{p}(i+1,I) - t^{p}(i+1,J) >= T(I) \qquad (24)$$

2)pogoji za istosmerni odhod vlakov s postaje

$$t^{o}(i,J) - t^{o}(i,I) >= T(J) \qquad (25)$$

ali

$$t^{o}(i,I) - t^{o}(i,J) >= T(I) \qquad (26)$$

Upostevamo enačbi (1) in (2) in dobimo:

$$\sum_{k=1}^{i} X(k,J) - \sum_{k=1}^{i} X(k,I) >= PRIH(i+1,I) - PRIH(i+1,J) + T(J) \qquad (27)$$

$$\sum_{k=1}^{i} X(k,I) - \sum_{k=1}^{i} X(k,J) >= PRIH(i+1,J) - PRIH(i+1,I) + T(I) \qquad (28)$$

$$\sum_{k=1}^{i} X(k,J) - \sum_{k=1}^{i} X(k,I) >= ODH(i,I) - ODH(i,J) + T(J) \qquad (29)$$

$$\sum_{k=1}^{i} X(k,I) - \sum_{k=1}^{i} X(k,J) >= ODH(i,J) - ODH(i,I) + T(I) \qquad (30)$$

Ce je vlak J na odseku i pred vlakom I,torej TV(i,I) > TV(i,J),sta močnejša pogoja (27) in (30). Ce pa je vlak I na odseku i pred vlakom J,torej TV(i,J) > TV(i,I),sta močnejša pogoja (28) in (29).
Torej

a)TV(i,J) > TV(i,J)

$$\sum_{k=1}^{i} X(k,I) - \sum_{k=1}^{i} X(k,J) \begin{cases} < -PRIH(i+1,I) + PRIH(i+1,J) - T(J) \\ \text{ali} \\ >= ODH(i,J) - ODH(i,I) + T(I) \end{cases} \qquad (31)$$

b)TV(i,J) > TV(i,I)

$$\sum_{k=1}^{i} X(k,I) - \sum_{k=1}^{i} X(k,J) \begin{cases} >= PRIH(i+1,J) - PRIH(i+1,I) + T(I) \\ \text{ali} \\ < ODH(i,I) - ODH(i,J) - T(J) \end{cases} \qquad (32)$$

Ce analiziramo zgornji neenačbi,vidimo:če prej odpravimo hitrejši vlak,je zato možno odpraviti počasnejši vlak,ne da bi čakali na iztek intervala T,v nasprotnem primeru pa tega ne moremo storiti,ker bi sicer prišlo do konfliktne situacije,zato počakamo.da se interval T izteče.

Pogoje lahko zapišemo tudi takole:

$$t^{o}(i,I) < t^{o}(i+1,Y) \Rightarrow t^{p}(i+1,I) < t^{o}(i+1,Y) \qquad (33)$$

ali

$$t^{o}(i+1,Y) < t^{o}(i,I) \Rightarrow t^{p}(i,Y) < t^{o}(i,I) \qquad (34)$$

Vemo,da se nekateri vlaki v danih situacijah lahko gibljejo po fiksnem voznem redu.Do take situacije lahko pride,če nekateri tovorni vlaki odstopajo od fiksnega voznega reda,torej zamujajo.V takem slučaju pa potniškim vlakom ni potrebno spreminjati voznega reda.

Naj ima vlak s fiksnim voznim redom indeks I,vlak,ki ne vozi po fiksnem voznem redu pa indeks J,oziroma Y.Potem neenačba (32) preide v

$$-\sum_{k=1}^{i} X(k,J) \begin{cases} >= T(I) + PRIH(i+1,J) - PRIH(i+1,I) \\ \text{ali} \\ < -T(J) + ODH(i,1) - ODH(i,J) \end{cases} \qquad (35)$$

Neenačbi (19) in (20) pa preideta v

$$-\sum_{k=i+1}^{NPOS} X(k,Y) \begin{cases} > PRIH(i,Y) - ODH(i,I) \\ \text{ali} \\ <= -PRIH(i+1,I) + ODH(i+1,Y) \end{cases} \qquad (36)$$

c)RAZRESITEV KONFLIKTNE SITUACIJE PRI SRECANJU VLAKOV

Vlaka I in Y se srečata na postaji i <=> ko veljata neenačbi:

$$t^{o}(i,I) >= t^{p}(i,Y) + T \qquad (37)$$

$$t^{p}(i,I) + T <= t^{o}(i,Y) \qquad (38)$$

Če upoštevamo enačbe (1),...,(4),dobimo

$$\sum_{k=1}^{I} X(k,I) - \sum_{k=I+1}^{NPOS} X(k,Y) >= ODH(i,I) PRIH(i,Y) + T \qquad (39)$$

$$\sum_{k=1}^{I-1} X(k,I) - \sum_{k=1}^{NPOS} X(k,Y) <= PRIH(i,I) - ODH(i,Y) - T \qquad (40)$$

To so neenačbe za srečanje vlakov.

d)RAZREŠITEV KONFLIKTNE SITUACIJE PRI PREHITEVANJU VLAKOV

Vlak J prehiti vlak I na postaji i <=> ko velja:

$$t^{p}(i,J) - t^{p}(i,I) >= T(J) \qquad (41)$$

ali

$$t^{o}(i,I) - t^{o}(i,J) >= T(I) \qquad (42)$$

Ločimo dve možnosti:

1)vlaka vozita v smeri ---->

2)vlaka vozita v smeri <----

V prvem primeru sta neenačbi

$$\sum_{k=1}^{i-1} X(k,I) + \sum_{k=1}^{i-1} X(k,J) >= T(J) - PRIH(i,J) + PRIH(i,I) \qquad (43)$$

ali

$$\sum_{k=1}^{I} X(k,I) - \sum_{k=1}^{I} X(k,J) >= T(I) - ODH(i,I) + ODH(i,J) \qquad (44)$$

kjer smo v neenačbah (37) in (38) upoštevali pogoje (1),...,(4).
V drugem primeru pa sta neenačbi

$$-\sum_{k=I+1}^{NPOS} X(k,I) + \sum_{k=I+1}^{NPOS} X(k,J) >= T(J) - PRIH(i,J) + PRIH(i,I) \qquad (45)$$

ali

$$\sum_{k=I}^{NPOS} X(k,I) - \sum_{k=I}^{NPOS} X(k,J) >= T(I) - ODH(i,I) + ODH(i,J) \qquad (46)$$

To so neenačbe za prehitevanje vlakov.

e)RAZREŠITEV KONFLIKTNE SITUACIJE,ČE JE EN ODSEK PROGE ZAPRT ZA PROMET

Odsek i je lahko v intervalu (t1,t2) zaprt za vlake.V tem primeru moramo upoštevati dodatne pogoje.

Čas odhoda vlaka I s postaje i mora biti večji od t2:

$$t^{o}(i,I) >= t2 \qquad (47)$$

ali pa mora biti čas prihoda vlaka I na postajo (i+1) manjši od t1:

$$t^{p}(i+1,I) <= t1 \qquad (48)$$

Če v neenačbah (43) in (44) upoštevamo enačbi (1) in (2),dobimo pogoj

$$\sum_{k=1}^{i} X(k,I) \begin{cases} <= & ODH(i,I) \\ & ali \\ <= & PRIH(i+1,I) \end{cases} \qquad (49)$$

Linearni program se potem glasi takole:

določiti je treba vrednosti spremenljivk X(I,J),I=1,...,NPOS,J=1,...,NVLAK,ki zadoščajo pogojem nenegativnosti X(I,J) >=0,I=1,...,NPOS, J=1,...,NVLAK in linearnim neenačbam:

-pogojem nasprotnosmernih vlakov (19,20)
-pogojem istosmernih vlakov (31,32)
-pogojem srečanja vlakov (35,36)
-pogojem prehitevanja vlakov (39,...,42)
-pogoju zaprtosti odseka (45)

tako,da ima namenska funkcija

$$\sum_{I=1}^{NPOS} \sum_{J=1}^{NVLAK} K(J)X(I,J) \qquad (50)$$

minimum.

Vendar se je izkazalo,da so pogojne neenačbe protislovne.Takih protislovnih pogojev pa linearni program že po definiciji ne rešuje,kajti taki pogoji ne tvorijo konveksnega poliedra,temveč poliedr,ki je odprt.Linearno programiranje z disjunktnimi pogoji se imenuje DISJUNKTNO PROGRAMIRANJE.Navedeni problem se ne da rešiti z navadnim linearnim programom,temveč z disjunktnim linearnim programom.

5.DISJUNKTNO PROGRAMIRANJE

Med linearne programe z logičnimi pogoji spadajo vsi nekonveksni programski problemi.Logični pogoji so take povezave med linearnimi neenačbami,ki vsebujejo operacije:

- in (konjunkcija)
- ali (disjunkcija)
- komplement (negacija).

Operacija če-potem (implikacija) je ekvivalentna disjunkciji,ker je (A => B) <=> ($\bar{A}$ $\vee$ B).V splošnem je disjunktni program (DP) problem oblike:

$$\{\min cx, Ax >= ao, x >= 0, x \in L\} \qquad (51)$$

kjer je A matrika m * n,ao vektor dimenzije m in L množica logičnih pogojev.

Obstaja več oblik disjunktnega linearnega programa.Najbolj znani sta dve:

i) DP je v "disjunktni normalni obliki",če so pogoji disjunkcije,ki ne vsebujejo naprej novih disjunkcij.

$$\bigvee_{h \in Q} \begin{pmatrix} h & h \\ A\,x & >= & ao \\ x & >= & 0 \end{pmatrix} \qquad (52)$$

ii) DP je v "konjunktivni normalni obliki",če so pogoji konjunkcije,ki ne vsebujejo naprej novih konjunkcij.

$$\begin{matrix} Ax >= ao \\ x >= 0 \end{matrix} \quad \bigvee_{l \in Q} \left(\bigvee_{j} (d^{i} x >= d^{i}_{o}), j \in S \right) \qquad (53)$$

ali

$$\begin{pmatrix} Ax >= ao \\ x >= 0 \end{pmatrix} \& \left(\bigvee_{i \in Q_j} (d^i x >= d^i_{io}) \right) \& \ldots \& \left(\bigvee_{i \in Q_{|S|}} (d^i x >= d^i_{io}) \right) \quad (54)$$

kjer je d^i vektor dimenzije n, d^i_{io} je skalar, Q_j, in S pa so lahko končne ali neskončne.

Zveza med obema oblikama je v tem, da ima vsaka disjunkcija (j) med (m+n) neenačbami sistema $Ax>=ao, x>=0$ natanko eno neenačbo

$$d^i x >= d^i_{io}, \quad i \in Q_j \quad (55)$$

iz vsake disjunkcije (ii) in da so vsi različni sistemi

$$A^h x >= ao^h, x >= 0 \quad (56)$$

s to lastnostjo, v mejah (i). Torej, če je Q_j (in hkrati vsi Q_j, $j \in S$) končen, potem je $|Q| = |\prod_j Q_j|$, $j \in S$, kjer je $\prod$ kartezični produkt. Ker sta operaciji in in ali distributivni, lahko vsak logični pogoj, ki vsebuje ti dve operaciji, zapišemo v prvi ali drugi značilni obliki DP. Splošna definicija disjunktnega linearnega programa se glasi: poiščimo vektor X, pri katerem linearna forma

$$L(X) = \sum_{j=1}^{n} c_j x_j \quad (57)$$

doseže minimalni minimum pri naslednjih omejitvah

$$\sum_{j=1}^{n} a_{ij} x_j \begin{cases} >= b_i \\ <= b_{ii} \end{cases} \quad (58)$$

kjer je $i=1,\ldots,m, j=1,\ldots,n, b_i < b_{ii}, c_j >= 0, x_j >= 0$.

Takoj lahko ugotovimo, da so pogoji za navedeni disjunktni linearni program v konjunktivni normalni obliki, torej se dajo zapisati kot konjunkcija samih disjunkcij.

Definicijsko območje C naloge v n-dimenzionalnem prostoru dobimo z unijo posameznih območij

$$C = \bigcup_{t=1}^{2^m} C^n_t$$

kjer je

$$C^n_t = \{ x, Ax >= B_t, x >= 0, t=1,2,\ldots,2^m \}$$

Problem rešujemo z metodo dihotomije (kar pomeni " razdeljenost na dvoje "), katere ideja je v zaporedni delitvi množice področij definiranosti naloge na dva dela, tako da izpustimo iz nadaljnega računanja tisto polovico, ki ne vsebuje ekstrema linearne forme. Vsako delno območje C^n_t, ki ga dobimo z vektorjem omejitev, ima v slučaju omejitve navzdol dano rešitev M_t. Naj bo minimalni tak minimum

$$M_m = \inf \{ M_t \}, \quad t=1,2,\ldots,2^m$$

Pri tem M_m je kriterijska funkcija najmanjša. Očitno je, da z naraščajočim m pridemo do prevelikega števila kombinacij.
Zapišimo pogoj omejitve tako, da uvedemo

$$\omega_i = \begin{cases} 1, & \text{pri } b_i \\ -1, & \text{pri } b_{ii} \end{cases} \quad (59)$$

Potem je pogoj omejitve

$$\omega_i \cdot \sum_{j=1}^{n} a_{ij} x_j >= b_i$$

$$\omega_i \cdot \sum_{j=1}^{n} a_{ij} x_j <= b_{ii} \quad (60)$$

Nalogo s tem pogojem omejitve označimo kot nalogo (1).

Definirajmo še dualno nalogo!

Iščemo minimalni maksimum linearnega programa

$$L(Y) = \sum_{i=1}^{m} b_i y_i \quad (61)$$

oziroma

$$L(Y) = \sum_{i=1}^{m} (-b_{ii}) y_i \quad (62)$$

pri naslednjih omejitvah

$$1) \sum_{i=1}^{m} \omega_i a_{ij} y_i <= c_j, \quad j=1,\ldots,n$$

$$2) \; y_i >= 0, \quad i=1,\ldots,m \quad (63)$$

Če uvedemo n dopolnilnih nenegativnih spremenljivk

$$1) \sum_{i=1}^{n} a_{ij} y_i + y_{n+j} = c_j, \quad j=1,\ldots,n$$

$$2) y_i >= 0, \quad i=1,\ldots,n+m \quad (64)$$

To nalogo pa imenujemo naloga 2.

Ker je $C^n_{t1} \cap C^n_{t2} = 0, t1 \neq t2$, ne obstaja splošen algoritem za obe področji definiranosti naloge 1. Nasprotno pa delna področja naloge 2 vzajemno vključujejo drugo drugega in za to nalogo obstaja splošen algoritem.

V (m+1) razsežnem prostoru Z razširjeni vektorski prostori Z^{m+1} presekajo množico konveksnih poliedrov :

$$C^{m+1}_t = \{ Z, AZ + B_t Z_{m+1} >= 0, Z >= 0 \}, \quad t=1,2,\ldots,2^m \quad (65)$$

Vektor omejitve v tem prostoru opišemo s premico

$$C = \{ Z, Z = C \} \quad (66)$$

definicijsko območje delne naloge v prostoru Z opišemo s presekom

$$C_t^{m+1} \cap C \qquad (67$$

Po drugi strani je ena mejna ploskev, to je ploskev, ki je definirana z enotskimi vektorji $A_{m+1}, \ldots, A_{m+n}$, skupna za vse poliedre. Ta ploskev ima pozitivno usmerjenost v prostoru dimenzije m in seka premico C, kjer je $c_j > 0$.

Od tod sledi, da se definicijska območja (67) pojavljajo kot urejene množice, ki vzajemno vključujejo druga drugo. Za nalogo 2 obstaja algoritem, ki temelji na zapovrstnem izboljševanju rešitve in je podoben algoritmu zapovrstnega izboljševanja rešitve pri klasični metodi linearnega programa. Postopek je iterativen.

Ker je bila v nalogo 2 uvedena enotska matrika, se v končni tabeli nahaja tudi rešitev začetne naloge 1.

6. ALGORITEM ZA DISJUNKTNO LINEARNO PROGRAMIRANJE

Algoritem za disjunktno programiranje podamo v obliki tabele, takšne, kot jo poznamo pri klasičnem linearnem programiranju. Označimo razliko

$$z_j - c_j = \Delta'_j \qquad (68)$$

pri b_i in

$$z_j - c_j = \Delta''_j \qquad (69)$$

pri b_{ii}.

Če je od Δ'_j in Δ''_j, $i=1,\ldots,m$ vsaj eden nenegativen, tedaj smo dobili minimalni maksimum. Naj bosta za vektor P_k oba vektorja Δ'_k in Δ''_k negativna, če uvedemo v bazo vektor P_k in odstranimo vektor P_p in v drugem primeru, če uvedemo vektor P_k in odstranimo vektor P_r. V prvem slučaju se vrednost linearne forme poveča za

$$\Delta L(P'_k) \qquad (70)$$

v drugem za

$$\Delta L(P''_k) \qquad (71)$$

V bazo uvedemo vektor, ki da pri dani iteraciji manjši prirast k linearni formi. Naj bo

$$\Delta L(P'_k) > \Delta L(P''_k) \qquad (72)$$

Potem je smiselno pri dani iteraciji neobhodno uvesti v bazo vektor P''_k. Vendar to vpliva tudi na Δ'_i in Δ''_i, zaradi česar moramo še dodatno zamenjati vektorje v bazi. Lahko pa nastopa tudi faktor Θ_i, ki povzroči dodatni prirast linearne forme. Enak premislek velja tudi za primer, ko uvedemo v bazo vektor P_k.

Pogoji, pri katerih je neobhodna zamenjava vektorjev, so naslednji:

1) če uvedemo v bazo P_k in odstranimo vektor P_p dobimo vrednosti

$$\Delta'^{(p)}_i = \Delta'_i - \Theta_p \cdot \Delta'_k \qquad (73)$$

$$\Delta''^{(p)}_i = \Delta''_i + \Theta_p \cdot \Delta'_k \qquad (74)$$

kjer je

$$\Theta_p = -(z'_{ip} / z''_{kp}) \qquad (75)$$

2) Če pa uvedemo v bazo vektor P_k in odstranimo vektor P_r, dobimo vrednosti.

$$\Delta'^{(r)}_i = \Delta'_i + \Theta''_r \cdot \Delta''_k \qquad (76)$$

$$\Delta''^{(r)}_i = \Delta''_i - \Theta''_r \cdot \Delta''_k \qquad (77)$$

kjer je

$$\Theta''^{(r)}_r = -(z''_{ir} / z''_{kr}) \qquad (78)$$

Naj bo iteracija, v kateri smo z uvedbo vektorja P_k, zadnja, pri kateri P'_k in P''_k na kakršenkoli način vplivata na prirast linearne forme. Torej od zdaj naprej uvedemo v bazo samo tak vektor, ki daje pri dani iteraciji manjši prirast. V tem primeru je splošni prirast v drugi in vseh naslednjih iteracijah

$$\sum \min_{k\,i} L(P'_k P'_i), \text{ pri uvedbi } P_p \qquad (79)$$

$$\Delta'^{(p)}_i < 0$$

in

$$\sum \min_{k\,i} L(P''_k P''_i), \text{ pri uvedbi } P_r \qquad (80)$$

$$\Delta''^{(r)}_i < 0$$

Označimo (79) z f1 in (80) z f2. Potem velja

Če $\Delta L(P'_k) + f1 < \Delta L(P''_k) + f2$, potem sledi:

je treba v bazo uvesti P'_k in odstraniti P_p.

Če $\Delta L(P'_k) + f1 > \Delta L(P''_k) + f2$, potem sledi:

je treba v bazo uvesti P''_k in odstraniti P_r.

Ta kriterij ostane v veljavi tudi v slučajih, ko vektorji, ki jih je treba uvesti, vplivajo tudi na prirast linearne forme v iteracijah, ki sledijo. Ker ima naloga 2 začetno umetno bazo, je

$$z_{ij} = a_{ij} \omega_i \qquad (81)$$

Rekurzivne enačbe za nalogo 2 pa so naslednje:

1)če zamenjamo vektor P_p z vektorjem P_k

$$z'_{ij}(p) = z_{ij} - (z_{ip} z_{kj})/z_{kp}, \quad j \neq k \qquad (82)$$

$$z'_{ij}(p) = \omega_i z'_{ij}(p), \quad j \neq k \qquad (83)$$

kjer je

$$z''_{ij} = \omega_i z_{ij} \qquad (84)$$

$$z'_{ik}(p) = z_{ik} / z_{kp} \qquad (85)$$

$$z'_{ik}(p) = \omega_i z'_{ik}(p) \qquad (86)$$

2)če zamenjamo vektor P_r z vektorjem P_k

$$z''_{ij}(r) = z'_{ij} - (z'_{ik} z'_{kj}) / z'_{kr}, \quad j \neq k \qquad (87)$$

$$z''_{ij}(r) = \omega_i z'_{ij}(r), \quad j \neq k \qquad (88)$$

$$z'_{ik}(r) = z''_{ik} / z''_{kr} = z''_{ik} / \omega_k z''_{kr} \qquad (89)$$

$$z''_{ik}(r) = \omega_i z'_{ik}(r) \qquad (90)$$

Iz formul vidimo,da ni potrebno računati obeh z'_{ij} in z''_{ij},ker se razlikujeta samo za faktor ω_i

Ocene Δ_j pa računamo po formulah (72), (74),(76) in (77).

Začetna tabela za prvo iteracijo je na sliki 1.

Prvi stolpec vsebuje vrednosti izbranih koeficientov linearne forme naloge 2.Ker je začetna baza iz umetnih vektorjev,ni zapolnjen.Drugi stolpec vsebuje oznake vektorjev baze.Tretji stolpec vsebuje vektorje omejitev po baznih vektorjih.Naslednjih m stolpcev vsebuje vektorje z'_{ij},za katere velja

$$z'_{ij} = a_{ij} \qquad (91)$$

Zgornji dve vrstici predstavljata dve možni vrednosti za koeficiente linearne forme.Predzadnji vrstici predstavljata ocene za Δ'_j in Δ''_j.Zadnja vrstica pa predstavlja velikost najmanjšega prirasta linearne forme,to je $\Delta_i L(P_i)$,če uvedemo v bazo vektor P_i.Levo spodaj pa je vsota vrednosti manjših prirastov in linearne forme pri dani bazi.V začetni tabeli je vrednost linearne forme

$$L(Y_{BI}) = 0 \qquad (92)$$

V poštev za uvedbo v bazo pridejo samo vektorji,pri katerih je

$$\Delta'_i < 0 \qquad (93)$$

in

$$\Delta''_j < 0 \qquad (94)$$

Vektor,ki ga uvedemo,izračunamo iz izraza

$$\min (z_{jo} / z_{kj}), \quad z_{kj} > 0 \qquad (95)$$

kjer je $z_{kj} = z'_{kj}$ pri uvedbi P_k in $z_{kk} = \omega_k z'_{kk}$ pri uvedbi P_k.Potem v skladu s prej navedenimi rekurzivnimi formulami sestavimo dve tabeli:

1)prvo,pri kateri v bazi zamenjamo vektor P_p z vektorjem P_k

2)drugo,pri kateri v bazi zamenjamo vektor P_r z vektorjem P_k

Ko v obeh tabelah zapolnimo zadnji vrstici,primerjamo najmanjše priraste in linearne forme pri novih bazah.Manjša vsota določa,kateri vektor je treba uvesti.Po določitvi nove baze iz nadaljnjega računanja izključimo eno tabelo,drugo tabelo pa kot začetno uporabimo pri naslednji iteraciji.

7.ZAKLJUCEK

Ta pristop za operativno načrtovanje daje globalno optimalno rešitev,vendar pa v praksi ni uporaben,ker pri večjem številu postaj in vlakov zahteva preveč računalniškega časa.Algoritem se zato uporablja samo za laboratorijsko proučevanje in načrtovanje gibanja vlakov na manjših primerih,ko je število vlakov majhno(npr.5) in število postaj tudi majhno(npr.4).Argumenti,ki kažejo na to,da tak algoritem ne ustreza praksi,so med drugim:

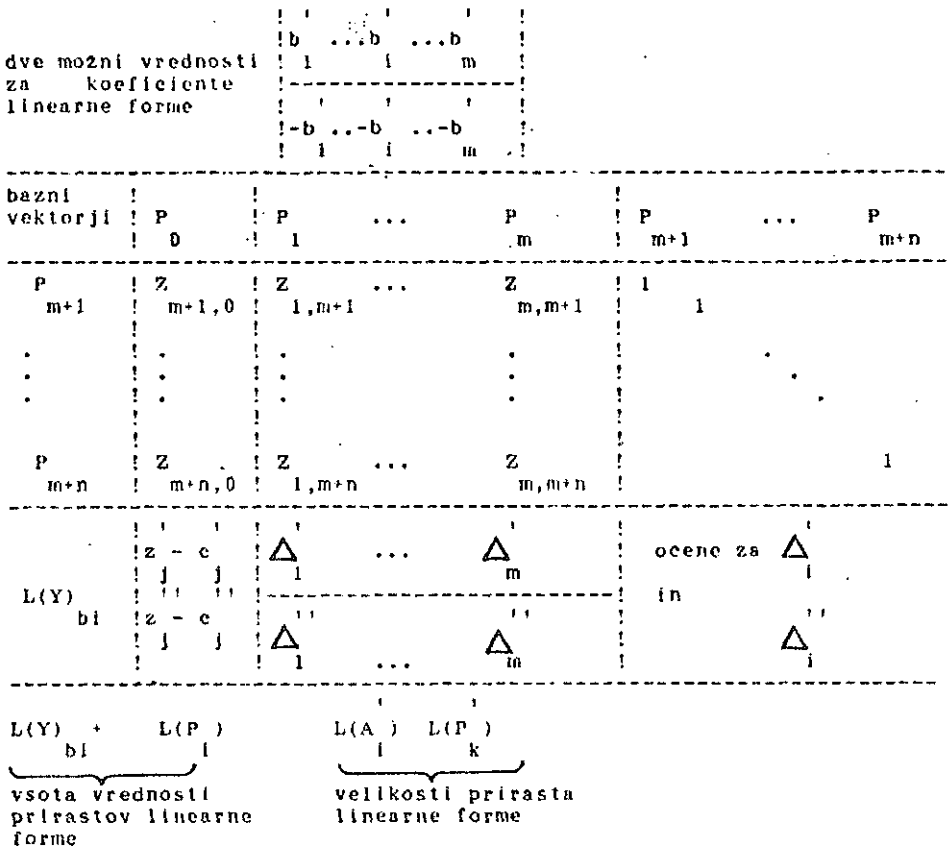
1)ponavadi podatki niso znani toliko vnaprej,da bi imel globalni optimum prednosti.Dogodki na progi se namreč stalno spreminjajo.

2)velik čas računanja zaradi že omenjenega prevelikega števila kombinacij

3)nekatere tehnološke omejitve je težko vključiti v model(npr.omejitev,da na postaji ne sme biti več vlakov kot tirov)

4)lokalni minimum je bližji realnemu gibanju vlakov na progi.

Zato so v praksi bolj uporabni algoritmi,ki problem rešujejo lokalno,kar pomeni,da rešujejo konflikte na manjšem omejenem območju nekaj postaj naprej in nekaj nazaj le med dvema vlakoma.



slika 1

8.LITERATURA

(1).P.L.Hammer,E.L.Johnson,B.H.Korte, Discrete optimization 2,Annals of discrete matematics,Disjunctive programming,North-Holland publishing company-Amsterdam,NewYork-Oxford,1979

(2).Bartkus A.T.,K voprosu o formalizaciji zadaci avtomatičeskogo regulirovanja dviženija poezdov,Zbornik trudov LIIZT,vlip.205, LIIZT, 1963

(3).Bartkus A.T.,Približenoe delenie nekotorih kombinatornih zadac linėinogo programirovania metodom dihotomii,Zurnal vičisnitelnoi matematiki i matematičeskoi fiziki,1964,N.4

(4).Zavjalov B.A.,Učastkovij avtodispečer, Izdajateljstvo TRANSPORT,Moskva,1967

(5).Vadnal Alojz,Linearno programiranje

(6).Korbut Anton,Sprotno prirejanje voznega reda,interna skripta v Iskri avtomatika

(7).Railway Safety Control and Automation towards the 21 th Century,London,1984

OPTIMAL CODE GENERATION FOR SOME SPECIAL CLASSES OF MACHINES

INFORMATICA 2/86

Dušan Petković
Siemens AG, München, F. R. Germany

UDK:681.3.325.6

ABSTRACT. Kameda and Abdel-Wahab [5] investigated the problem of generation of minimum-lengthcode for a wide class of machines. They establish an attainable lower bound on the number of "load"-instructions, as well as an attainable lower bound on the number of "store"-instructions. However, they did not succeed to reach the main goal: to minimize the total sum of "load"- and "store"- instructions, because of interaction between these two types of instructions. In this paper we show the optimal code generation for some special classes of machines given by Kameda and Abdel-Wahab.

OPTIMALNO GENERISANJE KODA ZA NEKE SPECIJALNE KLASE MAŠINA - Kameda i Abdel-Wahab [5] su ispitivali problem generisanja koda minimalne dužine za jednu širu klasu mašina. U svome radu oni su pokazali da broj "load"-naredbi i "store"- naredbi je minimalan za tu klasu mašina. Medjutim, nije im uspelo da dokažu da se suma "load"- i "store"- naredbi može minimizirati, zbog postojanja interakcije izmedju ova dva tipa naredbi za datu klasu mašina. U ovom radu ćemo pokazati optimalno generisanje koda za neke specijalne klase mašina koje su obuhvaćene klasom mašina datoj u radu Kamede i Abdel-Wahaba.

1. INTRODUCTION

Code generation is a part of compiler construction which has been getting increasing attention. In his paper Nakata [6] considered the problem of minimizing the number of general purpose registers needed to evaluate an arithmetic expression. Redziejowski [7] showed later that Nakata's algorithm gives the minimum number of registers when only commutative operators + and * are considered. Sethi and Ullman [8] developed an algorithm which generalized all previous results; their algorithm minimizes the number of instructions needed to evaluate an arithmetic expression when there is a fixed number of registers. All these algorithms are fast, because of a restriction that each subexpression will be used exactly once. Bruno and Sethi [2] showed that the problem of generating a shortest code for expressions containing common subexpressions is NP- complete even for a one-register machine. Carter [3] showed the best algorithm (until now) for generating code for one-register machine for expressions with common subexpressions.

In all these papers it is assumed that the result of an operation appears in a register and "store"- instruction is the only instruction that can change the content of memory locations. Kameda and Abdel-Wahab [5] relaxed this restriction and allow register-to-memory instructions other than "store", and also memory-to-memory instruction. Under such assumptions they defined a class of machines and tried to find an optimal solution for code generation for that class of machines. Unfortunately, they failed to reach that goal, because of interaction between "load"- and "store"- instructions .

In this paper we investigate some special classes of machines which belong to the wider class of machines defined by Kameda and Abdel-Wahab, and show optimal code generation for them.

In part 2 of this paper we give all definitions needed, and definition of the general class of machines. In the part 3 and 4 we show optimal code generation for two special classes of machines defined in part 2.

2. DEFINITIONS AND A GENERAL MODEL OF MACHINE

In this paper we adopt definitions given in [5] . An arithmetic expression is given as an expression tree whose leaves correspond to variables and/or constants, and whose interior nodes correspond to operations. If there is a downward path from a node v to another node w (we assume that a root is at the top), we say v is ancestor of w (similarly, we say w is descendent of v). A direct descendent will be called son. Thus, node representing a binary operation has two sons, left son and right son. A node and all its descendents comprise a subtree. The value of a node is defined to be the value of the subexpression represented by the subtree rooted at that node. The value of a tree is defined to be that of its root.

We call a downward path in an expression tree leftmost, iff each binary node in it, if any, which is not the last node of the path, is followed by its left son. Level-one node is a node both of whose sons are leaves. Level-n node is a node one of whose sons is level-(n-1) node and the level of the other node is $\leq$n-1. If the root of a tree is level-r node, we say that r is a depth of that tree.

We shall now give a general definition of a machine model for which program P has to be written (P evaluates a given expression tree).

The machine has $N \geq 1$ general registers and a countable sequence of memory locations. The instructions are

$r \leftarrow m$ "load"- instruction,
$m \leftarrow r$ "store"-instruction,

and following binary instructions:

$$\left.\begin{array}{l} m_1 \leftarrow m_1 \text{ op } m_2 \\ m \leftarrow m \text{ op } r \\ r_1 \leftarrow r_1 \text{ op } r_2 \\ r \leftarrow r \text{ op } m \end{array}\right\} \beta_1 \Big\} \beta_2$$

r denotes a register and m denotes a memory location. We assume that the type (β_1, β_2) of an instruction is determined by the operation it implements. If a binary operation requires its first operand to be in register, the corresponding instruction is said to be of a type β_1, and if a binary instruction requires its second operand to be in a register, the corresponding instruction is said to be of a type β_2. Analogous to that, a binary instruction that does not require its first (second) operand to be in a register is said to be of a type $\bar{\beta}_1$ ($\bar{\beta}_2$). A binary instruction may be of a type: $\beta_1 \wedge \beta_2$, $\beta_1 \wedge \bar{\beta}_2$, $\bar{\beta}_1 \wedge \beta_2$ and $\bar{\beta}_1 \wedge \bar{\beta}_2$. A particular machine may have only a subset of these types. For instance, the Sethi-Ullman machine model [8] is of the type $\beta_1 \wedge \bar{\beta}_2$.

Our goal is to find a program which evaluates a given expression tree for a special machine model in the minimum time.

3. $\bar{\beta}_1 \wedge \bar{\beta}_2$ - MACHINE MODEL

The set of instructions for this machine model is:

$m_1 \leftarrow m_1 \text{ op } m_2$,
$r \leftarrow m$
and $m \leftarrow r$.

The following algorithm is the basis for the optimal evaluation of a given expression tree.

Algorithm 1.

Step 1. Find each subtree of the given expression tree T, whose both sons are leaves. Evaluate all these subtrees (sequence of evaluation is irrelevant). If every subtree is evaluated, go to step 2.

Step 2. Replace every subtree from step 1 with a leaf and let the new subtree be T_1. If T_1 consists only of the root of T, terminate. If not, apply step 1 to the tree T_1.

Now, we prove some lemmas, which hold for this machine model.

Lemma 1. If every leaf of a given expression tree T is in memory, than "store"- and "load"-instructions are superflous in evaluating T.

Proof. If every leaf is in the memory, than the result of evaluating every internal node (with op-instruction) will also be in the memory.

Under conditions of lemma 1 this machine model has only one instruction:

$$m_1 \leftarrow m_1 \text{ op } m_2.$$

Lemma 2. Every level-one node I of a given expression tree can be immediately evaluated. Further, the result of that evaluation replaces the value of the left son of I in the memory.

Proof. Directly from the definition of the machine model.

Lemma 3. The number of instructions of the program which optimally generates code for a given expression tree is equal to the number of internal nodes of that tree.

Proof. We have to show how to evaluate every internal node of a given expression tree with only one instruction. Let I be the internal node with left son I_l and right son I_r, and let I_l and I_r be level-one nodes. Applying lemma 2 to I_l and I_r we can evaluate each of them with one instruction. Also, the values of I_l and I_r are now in memory. Hence, we can evaluate I also with one instruction and the value of I will be in memory. Proceeding that way for all internal nodes we can evaluate every internal node of a given expression tree with one instruction.

Theorem 1. Algorithm 1 produces program which optimally generates code for a given expression tree and the value of the root of the tree will replace the value of the left son of the leftmost downward path starting at the root. Further, algorithm 1 produces that program in linear time.

Proof. The first part of the theorem follows directly from lemma 2 and lemma 3. To show that algorithm can be constructed in linear time we traverse given expression tree and mark all subtrees from step 1. This can be done in linear time. Step 1 will be executed d times, where d is the depth of given expression tree. We need one instruction for evaluation of every subtree. Therefore, the algorithm can be performed in linear time.

4. $\bar{\beta}_1 \wedge \beta_2$ - MACHINE MODEL

The set of instructions for this machine model is:

$m \leftarrow m \text{ op } r$,
$r \leftarrow m$
and $m \leftarrow r$.

Algorithm 2.

Step 1. Find each subtree of the given expression tree T, whose both sons are leaves. For every such subtree S first "load" the right son of S and evaluate S immediately after that (the sequence of evaluation of subtrees is irrelevant). If every subtree is evaluated, go to step 2.

Step 2. Replace every subtree from step 1 with a leaf and let the new tree be T_1. If T_1 consists only of the root of T, terminate. If not, apply step 1 to the tree T_1.

We shall show that algorithm 2 generates optimal code for a given expression tree and the given machine model. For that purpose we proof some lemmas which hold for this machine model.

Lemma 4. If every leaf of a given expression tree T is in memory than "store"-instruction is superflous in evaluating T.

Proof. To evaluate any internal node, every left operand has to be in a memory and every right operand in a register. After evaluation, every internal node will be in memory. Therefore, "load"- instruction is only instruction we need, besides op- instruction.

Under condition of lemma 4 the machine model has than following two instructions:

$m \leftarrow m \text{ op } r$
and $r \leftarrow m$.

Lemma 5. The number of op-instructions required in an optimal code generation of a given expression tree T is equal to the number of internal nodes of T.

Proof. Trivial.

Lemma 6. The number of "load"-instructions required in an optimal code generation of a given expression tree T with the given machine model, is equal to the number of internal nodes of T.

Proof. To evaluate any level-one node of T we need one "load"-instruction for his right son. After evaluating all level-one nodes we can, in the same manner, evaluate all level-two nodes and so on.

Theorem 2. Algorithm 2 produces program P which optimally generates code for a given expression tree and the value of the root will replace the value of the left son of the leftmost downward path starting at the root. Further, the construction of program P is linear in time.

Proof. In lemma 4 we show that "load"- and op-instructions are only instructions needed to evaluate a given expression tree. Lemma 5 and lemma 6 give the number of op- and "load-instructions needed to generate optimal code for a tree. Algorithm 2 evaluates all expression trees exactly in 2*n steps, where n is a number of internal nodes of the tree. Therefore from lemma 5 and lemma 6 algorithm 2 produces program which optimally generates code for a given expression tree.

The proof of the second part of the theorem is similar to the proof of the second part of theorem 1.

5. CONCLUDING REMARKS.

Our final objective is to minimize the total number of "load"- and "store"-instructions for the class of machines defined in [5]. We still do not have a solution of that problem, but we are hopefull that one will be found.

6. REFERENCES

[1] Aho,A.V. and Johnson,S.C. - Optimal Code Generation for Expression Trees, Journal of ACM, Vol.23, No.3, (1976), p.488-501.

[2] Bruno,J.L. and Sethi,R. - Code Generation for One-Register Machine, Journal of ACM, Vol.23, No.3, (1976), p.502-510.

[3] Carter,L.R. - Further Analysis of Code Generation for a Single Register Machine, Acta Informatica, Vol.18, (1982), p.135-47

[4] Garey,M.R. and Johnson,D.S. - Computers and Intractability, Freeman and Co. 1979.

[5] Kameda,T. and Abdel-Wahab,H.M. - Optimal Code Generation for Machines with Different Types of Insturctions, in: Gilchrist, B.(ed.) - Information Processing, IFIP Congress, Proc. of 1977,North Holland.

[6] Nakata,I. - On Compiling Algorithms for Arithmetic Expressions, Comm. ACM, Vol.10, No.8, (1967), p.492-494.

[7] Redziejowski,R.R. - On Arithmetic Expressions and Trees, Comm. ACM, Vol.12, No.2, (1969), p.81-84.

[8] Sethi,R and Ullman,J.D. - The Generation of Optimal Code for Arithmetic Expressions Journal of ACM,Vol.17,No.4,(1970),p.715-28

[9] Ullman,J.D. - The Complexity of Code Generation, in: Traub,J.D.(ed.) - Algorithms and Complexity, Academic Press, 1976.

Anton Ružić, Aleš Klofutar
Inštitut Jožef Stefan, Ljubljana

UDK: 681.326

V članku podajamo pregled paralelnega programiranja. V uvodu opisujemo prednosti in potrebe po uporabi metod paralelnega programiranja ter načine doseganja paralelnega izvajanja. Na to opisujemo paralelno programiranje s prikazom razvoja tega področja. Ta je potekal skozi nezanesljivost začetne tovrstne programske opreme, postavljanje konceptualnih temeljev do razvoja programskih jezikov, ki vključujejo postavljene koncepte.

THE DEVELOPMENT OF CONCURRENT PROGRAMMING. In this article we describe concurrent programming. The methods and techniques are introduced through overviewing of the development of concurrent programming. This went through several stages: the initial development of complicated and unreliable software systems, the search for abstract concepts that simplified understanding and the incorporating of these concepts into new programming languages.

1 Uvod

Z nazivom paralelno programiranje zajamemo programske zapise in tehnike, ki izražajo možnost paralelnega izvajanja in ki omogočajo reševanje rezultirajočih sinhronizacijskih in komunikacijskih problemov. Posamezne programske enote, ki se paralelno izvajajo, imenujemo procesi. Imamo lahko navidezno in dejansko paralelnost. Navidezno paralelnost imamo, če tečejo vsi procesi na enem procesorju in je vsakemu procesu dodeljen določen čas. Dejansko paralelnost dosežemo, če uporabimo več procesorjev in vsak prevzame en ali več procesov.

Paralelni program omogoča, da računalnik izvaja več nalog istočasno. S paralelnim programiranjem zvečamo računalnikovo učinkovitost in enostavnejše obvladujemo okolja, v katerih se računalnik hkrati posveča različnim opravilom.

V prvem delu na kratko opisujemo prednosti, ki nam jih omogoča uporaba paralelnega programiranja pri izgradnji programskih sistemov. Podajamo možne načine za uvajanje paralelnosti in probleme, ki jih moramo pri tem rešiti.

V nadaljevanju širše opisujemo možne načine za predstavljanje paralelnih dejavnosti (procesov) in možne načine za komunikacijo in sinhronizacijo med procesi. Posamezne načine opisujemo skozi pregled razvoja paralelnega programiranja. Razvoj se je začel konec 60. let. Začetni motiv je bil izkoriščanje možnosti, ki jih je nudila nova materialna oprema.

Področja uporabe

Paralelno programiranje se uporablja predvsem pri operacijskih sistemih in v sistemih za delo v realnem času.

Pri operacijskih sistemih lahko izboljšamo izkoristek materialne opreme, če imamo v pomnilniku več programov hkrati. Medtem ko en program čaka na izvedbo vhodne ali izhodne operacije s počasno periferno enoto, procesor računalnika izvaja nek drug program, naložen v pomnilniku. V tem primeru se procesor preklaplja med različne programe, ki se izvajajo navidezno istočasno. Več uporabnikov lahko hkrati dela na računalniku in uporablja vse računalnikove resurse.

Uporaba paralelnega programiranja je posebej koristna, če ne skoraj nujna, pri vgnezdenih sistemih za delo v realnem času, kot so na primer procesni sistemi. Pri teh je delitev globalne naloge na več procesov motivirana predvsem z enostavnejšim reševanjem naloge. Računalnik mora pravočasno sprejemati signale z različnih neračunalniških ali računalniških naprav in jih ustrezno krmiliti. Zahteve po sprejemanju in krmiljenju se v splošnem lahko pojavijo v različnih časovnih trenutkih,

saj je to pogojeno z dinamiko sistemov. Računalnik mora hkrati obravnavati in obdelovati podatke s številnih vhodov in izhodov. V tem primeru upravljalsko nalogo enostavneje in naravneje rešimo, če jo razdelimo na več delov, ki se paralelno izvajajo.

Paralelno programiranje uporabimo tudi zato, da zvečamo zmogljivost računalnika in skrajšamo čas izvajanja neke naloge. Namesto enoprocesorskega vzamemo večprocesorski računalniški sistem. Celotno nalogo razdelimo na podnaloge, ki se paralelno izvajajo na posameznih procesorskih elementih.

Delitev naloge na dele, ki se lahko paralelno izvajajo

Pri zgrajevanju sistema, ki uporablja paralelno programiranje, moramo najprej ugotoviti, katere aktivnosti se lahko opravljajo paralelno. Celotna naloga, ki jo rešujemo, je lahko postavljena tako, da jo enostavno razdelimo na paralelne aktivnosti. Primer je krmiljenje določenega števila perifernih naprav. Pri nekaterih nalogah je ta problem bolj zapleten in se ne da zadovoljivo rešiti z uporabo "ad hoc" načinov. Takšen primer so nekateri matematični izračuni, kjer moramo identificirati delne izračune, ki se lahko istočasno izvajajo, preden se izmenjajo vmesni rezultati izračunov. Paralelno preračunavanje moramo optimizirati pri omejitvah zaradi serijsko-paralelnih prednostnih razmerij.

Pomembno merilo za uspešnost delitve je čim večja neodvisnost procesov. V idealnem primeru so procesi popolnoma samostojni in operirajo le nad svojimi podatki. Ponavadi pa posamezni procesi opravljajo neko skupno nalogo, potrebujejo iste resurse ali kako drugače sodelujejo. Zato je potrebno, da procesi med sabo komunicirajo ali pa se v določenih točkah sinhronizirajo. Potem, ko smo razdelili nalogo na logične enote, moramo izbirati način za izražanje paralelnega izvajanja posameznih procesov in izbrati orodja oziroma načine, ki bodo omogočili nadzorovano in pravilno interakcijo procesov. Te načine širše opisujemo v naslednjem poglavju.

Načini paralelnega izvajanja

Nazadnje se moramo pri takšnih sistemih odločiti za primeren način paralelnega izvajanja in sinhronizacije ter komunikacije med procesi.

Če nimamo nobenega primernega sistemskega orodja, napišemo večposlovni izvrševalnik. To je pravzaprav inačica jedra pri multiprogramskih operacijskih sistemih. Celotno nalogo razdelimo na procese in jih skupaj z izvrševalnikom naložimo v pomnilnik. Izvrševalnik skrbi za izmenjevanje izvajanja poslov na procesorju (procesom, ki so del enega uporabniškega programa, lahko rečemo posli, angl. "tasks"). V izvrševalnik vgradimo tudi semaforje ali druge mehanizme za komunikacijo in sinhronizacijo.

Pri multiprogramskem ali večposlovnem operacijskem sistemu uporabimo za doseganje paralelnega izvajanja programov sistemski izvrševalnik in druge funkcije, ki jih nudi operacijski sistem. V tem primeru posamezne procese zapišemo kot posle ali samostojne programe v primernem programskem jeziku. S sistemskimi ukazi sprožimo paralelno izvajanje posameznih programov. Takšni operacijski sistemi podpirajo tudi nek način medprocesne komunikacije. To so lahko semaforji, dogodkovne zastavice, poštni predali in podobno.

Nazadnje imamo možnost, da uporabimo programske jezike, ki podpirajo paralelno programiranje. Osnova teh jezikov je jedro, ki omogoča paraleno izvajanje modulov in interakcijo med procesi. Te možnosti uporabljamo s primernimi stavki jezika. V to kategorijo lahko postavimo jezike PEARL, Concurrent Pascal, Modula, Edison, Ada in druge. Vključitev paralelnosti z uporabo takšnih jezikov je zelo primerna, saj se v tem primeru bolj posvetimo reševanju naloge in ne implementaciji mehanizmov paralelnosti.

2 Pregled razvoja in opis načinov in orodij paralelnega programiranja

V prejšnjem poglavju smo pokazali, kako uporabljamo paralelno programiranje za učinkovito izkoriščanje računalnikovih zmogljivosti, zvečevanje računalnikove zmogljivosti in za uspešno obvladovanje okolij, v katerih se mora računalnik hkrati posvečati različnim dogodkom. Paralelni sistemi so pri takšnih sistemih koristni in potrebni, izdelava pravilnih in zanesljivih programov pa je zahtevna. Najmanjša napaka lahko povzroči, da se paralelni program izvaja nepravilno in neponovljivo, kar onemogoča testiranje programa. Opisali bomo, kako so programski inženirji postopoma reševali ta problem.

Paralelno programiranje so uvedli, da bi izkoristili dosežke na področju materialne opreme. Po začetnem poskušanju so programerji ob pomanjkljivem znanju izdelali zapletene sisteme. Ti sistemi so zato bili tako nezanesljivi, da so sami načrtovalci uporabili izraz "kriza programske opreme". Računalniški znanstveniki so

spoznali pomen problema in začeli iskati abstraktne koncepte, s katerimi so poenostavili razumevanje paralelnih programov. Ko so razumeli bistvo problema, so postavili zapis za osnovne koncepte in jih definirali tako precizno, da so jih lahko vključili v nove programske jezike. Jezikovni zapis je omogočil napredek pri formalnem razumevanju problema.

V tem poglavju bomo opisali razvoj paralelnega programiranja. Ob tem bomo obravnavali načine izražanja paralelnega izvajanja procesov in načine za komunikacijo in sinhronizacijo med procesi.

2.1 Razvoj materialne opreme

Paralelno programiranje se je začelo razvijati pri reševanju problemov pri operacijskih sistemih, ki so sledili razvoju materialne opreme.

Prvi, enoposlovni operacijski sistemi so bili zelo primitivni in so naenkrat izvajali le en posel (uporabnikovo nalogo, angl. "single job"). Tipična vhodna enota je bila čitalec kartic in tipična izhodna enota je bila vrstični tiskalnik. Obe napravi sta bili veliko počasnejši od procesorja. Zaradi čakanja na počasnejše naprave je bil procesor veliko časa neizkoriščen. Poleg tega so bili prvi sistemi občutljivi na programerjeve napake, ker ni imel zaščitnih mehanizmov.

Z razvojem hitrejših pomnilniških perifernih naprav so izdelali operacijske sisteme za paketno obdelavo (angl. "batch processing"). Na manjših računalnikih so večje število poslov posneli s počasnejših čitalcev kartic na hitrejši magnetni trak ali disk. Sistem je potem s teh hitrejših pomnilniških enot zaporedno bral in izvajal posle, rezultate pa je zapisoval na hitrejše pomnilniške enote. Ko je bila obdelava končana, so rezultate natisnili. Sistemi za paketno obdelavo so bili boljši od prejšnjih, so pa še vedno zaporedno izvajali posle. Rezultati posameznih poslov so bili dostopni, ko so bili izvedeni vsi posli. Zato so rezultati krajših in daljših poslov bili dostopni ob istem času.

Naslednjo izboljšavo operacijskih sistemov je omogočila uvedba oziroma uporaba prekinitev, s katerimi so presegli sekvenčno naravo računalnika in dosegli delitev procesorjevega časa med različne programe. Pripadajoče tehnike, ki podpirajo takšno paralelno izvajanje programov imenujemo "multiprogramiranje". Tu se je začel razvoj paralelnega programiranja.

Multiprogramiranje so uvedli zato, da bi bil procesor in vhodno izhodne enote čim bolj izkoriščeni. Pri osnovnem načinu imamo istočasno v pomnilniku več poslov s pripadajočimi podatki. Časovnik v enakomernih časovnih presledkih prekinja trenutno izvajani posel in sproži izvajanje naslednjega posla. V kratkem času se izmenjajo vsi posli, tako da se izvajajo navidezno istočasno. Trenutno aktivni posel se ustavi tudi takrat, ko sproži izvajanje vhodne ali izhodne operacije s periferno enoto. Medtem ko se vhodna ali izhodna operacija izvaja (na primer zapisovanje na disk), se na procesorju izmenjujejo drugi posli. Ko periferna naprava konča operacijo, s prekinitvijo javi da se pripadajoči posel lahko nadaljuje.

Pozneje so takšne multiprogramske sisteme za paketno obdelavo dopolnili tako, da je lahko več uporabnikov hkrati interaktivno delalo z računalnikom preko terminalov. Takšne sisteme imenujemo operacijski sistemi s časovnim prepletanjem (angl. "time sharing operating system").

2.2 Nezanesljivost programske opreme

Možnost istočasnega izvajanja večjega števila programov na enem računalniku, ki so jo omogočili prvi multiprogramski sistemi, je zelo zvečala zmogljivosti računalnikov, povečala pa se je zapletenost. Pri teh sistemih je namreč zelo pomembno, da se posamezni programi pri izvajanju ne motijo in da potekajo na pravilen način. Programske napake so povzročile, da se je paraleleni program obnašal nepravilno in časovno odvisno. Ker so bile posledice napak različne od primera do primera, celo pri enakih vhodnih podatkih, jih je bilo zelo težko odkriti.

Zaradi omenjenih težav paralelnosti je bilo pomembno, da se uporabniku omogoči enostaven, zaporeden pristop do stroja. Operacijske sisteme so zgradili zato, da bi bili računalniški sistemi učinkoviti in zanesljivi ter enostavni za uporabo.

Zgodnji operacijski sistemi za paketno obdelavo, kot Atlas (1961) in Exec II (1962) so bili učinkoviti in enostavni, niso pa bili popolnoma zanesljivi. Prvi sistemi s časovnim prepletanjem, kot CTSS (1962) in SDCQ-32 (1964) so bili sorazmerno majhni. Operacijski sistemi naslednje generacije pa so bili veliko bolj obsežni in zapleteni. Razvoj sistema Multics (1965) je tako zahteval 200 človek let, OS 360 (1966) pa celo 5000 človek let. Zaradi svoje velikosti je bil OS 360 precej nezanesljiv. V vsaki verziji je bilo okrog 1000 napak.

Veliki operacijski sistemi so se dnevno podirali in postalo je dvomljivo ali resnično omogočajo učinkovito in zanesljivo delo računalnika. Postalo je jasno, da

takšne obsežne programe ni mogoče izdelati brez določenih konceptualnih temeljev, ki bi omogočili boljše razumevanje. Računalniški znanstveniki, ki so pri svojem delu prav tako uporabljali računalnike, so ugotovili pomembnost operacijskih sistemov in začeli delo na tem področju.

2.3 Konceptualni temelji

Najpomembnejši dosežek v nadaljnjem razvoju je bila ideja o delitvi paralelnega programa v sekvenčne procese, ki se asinhrono izvajajo. Obnašanje programa mora biti neodvisno od relativnih hitrosti procesov.

Proces je programski modul, sestavljen iz podatkovne strukture in zaporedja ukazov, ki operirajo nad njo. Če proces operira le na svojih podatkih, se bo obnašal popolnoma enako (ponovljivo) vsakič, ko bo pognan z enakimi podatki.

Procesi, ki si delijo računalniške resurse ali pa delajo na skupnih nalogah morajo biti zmožni pravilne interakcije, ki jo imenujemo procesna komunikacija ali procesna sinhronizacija. Takšne procese delimo glede na medsebojno razmerje do nekega resursa na tekmovalne procese, ki tekmujejo za neke stalne resurse (tračna enota, tiskalnik, spremenljivke itd.) in takšne, ki so v razmerju proizvajalca in potrošnika začasnih ali potrošnih resursov (sporočila, signali itd.). Če si proizvajalci in potrošniki izmenujejo sporočila, jih imenujemo komunicirajoči procesi, če pa si izmenjujejo sinhronizacijske signale, jih imenujemo sodelujoči procesi.

Kritične sekcije

Dijkstra je ugotovil, da komunikacijo med procesi lahko prevedemo na izvrševanje operacij na podatkih, ki so skupni večim procesom. Pomembno je, da naenkrat le en proces izvaja določene operacije na nekem skupnem podatku, ker sicer lahko nastopijo nepredvidljive posledice. Vzrok je v tem, da nobeden od procesov ne ve, kakšne operacije izvajajo ostali procesi v istem času nad skupnimi podatki. Pripadajoče napake so časovno neodvisne in različne odvisno od tega, kako se procesi pri izvajanju prekrivajo.

Nazoren primer je, ko si dva procesa delita navadno spremenljivko, ki je števec dogodkov. Če se prekrije povečevanje števca dobimo:

cikli	prvi proces	drugi proces
	(n=3)	
t0	naloži n	
t1		naloži n
t2		n:=n+1
t3		shrani n
t4	n:=n+1	
t5	shrani n	
	(n=4)	

Po dvojnem povečanju vrednosti smo iz n=3 dobili n=4 namesto n=5.

Tiste dele procesov, ki vršijo operacije na skupnih spremenljivkah, je Dijkstra imenoval kritične sekcije. Potrebno je zagotoviti, da bo samo en proces naenkrat izvajal kritično sekcijo, kar imenujemo problem vzajemnega izključevanja izvajanja kritične sekcije.

Vzajemno izključevanje zagotovimo, če proces sestavimo tako:

```
neodvisni del
....
vstopni protokol
kritična sekcija
izstopni protokol
....
neodvisni del
```

Vsak proces pred kritično sekcijo izvede nek vstopni protokol. Ta bo dovolil nadaljevanje samo takrat, kadar noben drug proces ne izvaja kritične sekcije. Na koncu kritične sekcije proces izvede izstopni protokol, ki omogoči, da v kritično sekcijo vstopijo drugi procesi.

Najbolj znan je Dekkerjev algoritem za zagotavljanje vzajemnega izključevanja. Algoritem za dva procesa, ki ga predstavimo v višjem programskem jeziku poteka tako:

```
program Dekker;
var
  turn : integer;
  c1,c2: integer;

procedure p1;                    (*prvi proces*)
begin
  repeat
    .............;               (*neodvisni del*)
    c1:=0;                       (*vstopni protokol*)
    while c2=0 do
      if turn=2 then
```

```
      begin
        c1:=1;
        while turn=2 do;
        c1:=0
      end;
    crit1;                  (*kritična sekcija*)
    turn:=2;                (*izstopni protokol*)
    c1:=1;
    .....                   (*neodvisni del*)
    forever
end;

procedure p2;               (*drugi proces*)
begin
  repeat
    .....;                  (*neodvisni del*)
    c2:=0;                  (*vstopni protokol*)
    while c1=0 do
      if turn=1 then
        begin
          c2:=1;
          while turn=1 do;
          c2:=0
        end;
    crit2;                  (*kritična sekcija*)
    turn:=1;                (*izstopni protokol*)
    c2:=1;
    .....                   (*neodvisni del*)
    forever
end;

begin                       (*glavni program*)
  c1:=1;
  c2:=1;
  turn:=1;
  cobegin           (*paralelno izvajanje p1 in p2*)
    p1;p2
  coend
end.
```

V programu smo uporabili paralelni stavek cobegin - coend, ki ga bomo podrobneje opisali v naslednjem poglavju. Vstop v kritično sekcijo proces p1 (p2) naznani z nastavljanjem c1 (c2) na 0. Če p2 (p1) hkrati izvaja vstopni protokol, spremenljivka turn odloča, kateri proces bo vstopil v kritično sekcijo.

Algoritmi za vzajemno izključevanje več kot dveh procesov so precej zapleteni, zato so neprimerni za praktično uporabo. Druga slabost takšnih algoritmov je v zaporednem testiranju vstopnega pogoja, kadar želi več procesov hkrati izvajati isto kritično sekcijo. To vidimo iz stavka: while pogoj do (* nič *). Medtem ko en proces vstopi v kritično sekcijo, drugi procesi zapravljajo računalniški čas s preverjanjem, ali je sekcija prosta (angl. "busy waiting").

Semaforji

Dijkstra je uvedel podatkovni tip semafor, ki je primeren za prenašanje sinhronizacijskih signalov (1968). Semafor lahko uporabimo tudi pri reševanju ostalih problemov paralelnega programiranja. Na primer, z njim enostavno rešimo problem vzajemnega izključevanja.

Nad spremenljivko s tipa semafor sta definirani dve operaciji:

wait(s): če s>0 tedaj s:=s-1 sicer se izvajanje procesa, ki je klical wait(s) ustavi in proces se postavi v čakalno vrsto,

signal(s): če je bil nek drugi proces P z operacijo wait(s) nad tem semaforjem ustavljen in postavljen v vrsto ga zbudi in izvajaj, sicer s:=s+1

Operaciji "wait" in "signal" morata biti implementirani kot primitivni operaciji, torej se nedeljivo izvajata. Ko nek proces izvaja semaforsko instrukcijo, počakajo vsi ostali procesi, ki v tem času tudi zahtevajo izvajanje semaforske instrukcije. Semaforje, ki lahko zavzamejo poljubno pozitivno vrednost imenujemo splošni semaforji. Če dovolimo, da zavzamejo samo vrednosti 0 in 1, imenujemo takšne semaforje binarne semaforje.

Rešitev sinhronizacije dveh procesov s semaforji je preprosta:

```
program synchronisation;
var
  s: semaphore;
procedure p1;
begin
  .....
  wait(s);  (*cakanje na sinhronizacijo s p1*)
  .....
end;
procedure p2;
begin
  .....
  signal(s); (*sprozanje sinhronizacijskega signala*)
  .....
end;
begin       (*main program*)
  s:=0;
  cobegin   (*paralelno izvajanje p1 in p2*)
    p1;p2
  coend;
end.
```

Z operacijo "signal" proces preko semaforske spremenljivke odda sinhronizacijski signal drugemu procesu, ki oddani signal sprejme z "wait" operacijo. V paralelnem sistemu programer ne more predvideti relativne hitrosti asinhronih procesov. Ne moremo vedeti, če bo proces poslal signal preden ga bo drugi procesor pripravljen sprejeti. Semaforske operacije so definirane tako, da ni pomembno v kakšnem vrstnem redu se izvajajo. Proces, ki poskuša sprejeti sinhronizacijski signal še preden je ta oddan, se postavi v čakalno vrsto in zakasni, dokler nek drug proces ne bo oddal ustreznega signala. Če se signali oddajajo hitreje, kot se sprejemajo, se enostavno shranijo v semaforski spremenljivki, dokler ne bodo uporabljeni. Zaradi komutativnosti semaforskih operacij postane sinhronizacija procesov časovno neodvisna.

Rešitev navadne sinhronizacije s semaforji je enostavna, drugi problemi pa lahko zahtevajo bolj zapletene rešitve. Slabost semaforjev je tudi v tem, da se lahko sistem, zgrajen s semaforji, podre, če pozabimo na eno samo semaforsko operacijo.

Dijkstrin multiprogramirni sistem THE (1968) je uvedel večino konceptov na katerih temelji današnje razumevanje paralelnega programiranja. Njegov sistem je bil hierarhično zgrajen iz več programskih plasti, ki so fizični stroj postopoma pretvarjali v prijaznejši abstraktni stroj, ki je lahko izvajal številne procese. Ti so si delili obsežen homogen pomnilnik in številne virtualne naprave.

Pri sistemih, pri katerih paralelni procesi uporabljajo iste resurse so začeli raziskovati tudi načine urejanja zahtev po zaseganju resursov in prenašanja sporočil, da se preprečijo smrtni objemi (angl. "deadlock"). To je pojav, ko vsak izmed dveh ali več procesov zaseda nek resurs in čaka na resurs, ki ga zaseda drug proces. Napačna rešitev tega problema povzroči neskončno čakanje procesov.

2.4 Razvoj jezikov z elementi za paralelno programiranje

Okoli 1970. leta so raziskovalci začeli razvijati jezikovne zapise za opis novih konceptov.

Koncept programskega jezika mora predstavljati splošno idejo, ki se pogosto uporablja. Pomen in pravila koncepta programskega jezika morata biti natančno definirani. Predstavljen mora biti s kratkim in jedrnatim zapisom, ki omogoča enostavno spoznavanje elementov koncepta in njihove medsebojne odvisnosti. Pomembno je tudi, da je možna varna in učinkovita implementacija in da prevajalnik preverja, da so pravila gleda koncepta zadovoljena. Uporabljeni koncept mora omogočati programerju predvideti hitrost in velikost programa.

Paralelni stavek

Dijkstra je uvedel zapis paralelnega stavka, ki določi, da se več sekvenčnih stavkov izvaja paralelno. Paralelni stavek se zaključi, ko se zaključijo vsi sekvenčni stavki. Primer paralelnega stavka je:

```
var
  this, next: line;
....
cobegin
  consume(this); input(next)
coend
```

Medtem ko stavek consume porablja vrstico this, stavek input sprejema naslednjo vrstico next.

Paralelni stavek ima predvidljiv učinek samo v primeru, če posamezni pripadajoči stavki, ki predstavljajo paralelne procese operirajo nad različnimi spremenljivkami (v našem primeru this in next) . če več stavkov operira nad istimi spremenljivkami, bo učinek paralelnega stavka časovno odvisten. Da bi preprečili časovno odvisne programske napake mora prevajalnik razpoznavati privatne spremenljivke procesa, ki morajo biti nedostopne drugim procesom.

Kritične sekcije in pogojne kritične sekcije

čeprav je bistveno, da so nekatere spremenljivke dostopne samo enim procesom, je v primeru sodelovanja in komunikacije med procesi potrebno, da si procesi delijo nekatere spremenljivke.

Hoare in Brinch Hansen sta leta 1972 predlagala zapis za prirejanje deljene spremenljivke kritičnim sekcijam, ki operirajo z njo. Za primer lahko definiramo deljeno spremenljivko, ki jo uporabljamo kot uro:

```
var
  clock: shared integer
```

Procesi inkrementirajo in čitajo to spremenljivko s sledečimi stavki:

```
tick:    region clock do clock:=(clock+1) mod max
read(x): region clock do x:=clock
```

Deljena spremenljivka se lahko uporablja samo znotraj kritičnih sekcij (region), kar preverja prevajalnik. Kritične sekcije so tako implementirane, da je zagotovljeno, da se posamezne sekcije izvajajo brez prepletanja, ena za drugo.

Hoare in Brinch Hansen sta postavila tudi koncept in zapis pogojne kritične sekcije. Izvajanje sekcije se odlaša, dokler deljena spremenljivka ne izpolni nekega pogoja. Kot primer lahko prikažemo izravnalnik za sporočila, ki se sestoji iz vrstice "slot" in boolean vrednosti "full", ki kaže ali je vrstica polna:

```
var
  buffer: shared record
            slot:line;
            full:boolean
          end
```

V izravnalnik pišemo samo, ko je prazen in čitamo samo, ko je poln:

```
send(m):    region buffer when not full do
              begin slot:=m; full:=true end

receive(m): region buffer when full do
              begin m:=slot; full:=false end
```

Pri pogojnih kritičnih sekcijah nastopa problem učinkovite implementacije. Gre za omejevanje ponavljajočega testiranja boolean izraza, dokler pogoj ni izpolnjen. V ta namen so uvedli procesne vrste prirejene posameznim deljenim spremenljivkam. če proces testira v pogojni kritični sekciji vrednost izraza in pogoj ni izpolnjen, se postavi v procesno vrsto, prirejeno deljeni spremenljvki. Vsakič, ko neki proces konča izvajanje kritične sekcije, se testirajo izrazi procesov v vrsti. V primeru, da so izrazi nekaterih procesov izpolnjeni, se aktivira eden od teh procesov.

Monitorji

Dijkstra je predlagal, da v en programski modul zajamemo vse operacije nad deljenimi podatkovni strukturami, namesto da jih raztresemo po celotnem programu. S tem se poveča jasnost interakcij med procesi. Brinch Hansen je predlagal zapis jezika za ta koncept monitorja (1973) Hoare je opisal monitorski koncept in podal primere (1974).

Za ilustracijo vzemimo izravnalnik za sporočila z operacijami zapisovanja in čitanja, ki ga zgradimo z monitorjem.

```
monitor buffer;

var
  slot: line;              (*monitorjeve spremenljivke*)
  full: boolean;

procedure send(m: line);        (*proceduri monitorja*)
when not full do
  begin slot:=m; full:=true end;
procedure receive(var m: line);
when full do
  begin m:=slot; full:=false end;

begin                    (*telo monitorja-inicializacija*)
  full:=false
end;


procedure producer;
var
  m: line;
begin
  ....
  produce(m);
  send(m);
  ....
end;
procedure consumer;
var m: line;
begin
  ....
  receive(m);
  consume(m);
  ....
end;

begin                                   (*glavni program*)
  cobegin
    producer; consumer
  coend
end.
```

Monitor je strukturirano orodje za medprocesno komunikacijo. Sestavljen je iz deklaracij globalnih spremenljivk in iz množice procedur, ki izvajajo določene operacije nad temi spremenljivkami. Procesi nimajo direktnega dostopa do globalnih spremenljivk, temveč samo preko klicanja monitorjevih procedur. Monitorjeve procedure imajo to lastnost, da lahko v nekem času samo en proces aktivno izvaja proceduro znotraj danega monitorja. Monitor ima tudi del, ki ga imenujemo telo in ki se uporablja za inicializacijo monitorjevih spremenljivk.

Monitorski koncept združuje dve ideji: vse kritične funkcije so zbrane na enem mestu in globalni podatki ter dostop do njih sta strukturirana.

Koncept monitorja omogoča, da prevajalnik testira, če se v programu nad deljenimi spremenljivkami izvajajo samo tiste operacije, ki so definirane v monitorju. Zgrajeni monitor lahko sistematsko in temeljito testiramo, potem pa prevajalnik preprečuje, da bi ga ostali programski moduli nepravilno uporabljali. Vsak proces s pripadajočimi lokalnimi spremenljivkami deklariramo kot poseben programski modul. Lokalne spremenljivke procesa morajo biti nedostopne drugim procesom (v našem primeru imata producer in consumer lokalno spremenljivko m). Prav tako testira prevajalnik pravilno uporabo lokalnih spremenljivk.

Testi pri prevajanju lahko nadomestijo teste pri izvajanju in zaščitne mehanizme v materialni opremi. Namen odpravljanje testov pri izvajanju ni samo doseganje učinkovitejše prevedene kode. Testi pri prevajanju preprečujejo nastop napak, testi pri izvajanju pa lahko samo sporočijo, zakaj se je sistem podrl. To je bistveno pri sistemih za delo v realnem času, ki opravljajo pomembne upravljalske funkcije.

Kot primer jezikov, ki temeljijo na monitorjih lahko navedemo Concurrent Pascal, ki so ga definirali in implementirali leta 1974 in Modula, ki so ga razvili leta 1977.

Jezikovni elementi so se razvijali v smer, kjer opis sinhronizacije med procesi podamo z opisom abstraktnega podatkovnega tipa. Pri zgoraj opisanem monitorju je na primer sinhronizacija med procesi v primeru praznega ali zapolnjenega izravnalnika implementirana v monitorjevih procedurah s stavkom "when pogoj do". Programer definira abstraktni podatkovni tip (izravnalnik) in dovoljene operacije nad njim. V opisu operacij je zajeta tudi potrebna sinhronizacija med procesi. Z uporabo takšnega podatkovnega tipa se izognemo opisu načina pravilne uporabe podatkovnega tipa.

Opis poti

Campbell in Habermann sta 1974 predlagala opis poti. To je abstraktni tip podoben monitorju, le da znotraj njega dodatno podamo izraz, ki določa zaporedje dostopa do podatkov. Oblika opisa poti je naslednja:

```
type
  ime = begin
          opis podatkovne strukture
          path opis zaporedja dostopa do podatkov end
          operation opis operacij nad podatki
        end
```

Izraz path opisuje dovoljeno zaporedje operacij nad podatki. V izrazu lahko opišemo tudi ponavljanje ali izbiro neke operacije.

2.5 Distribuirani sistemi

Doslej opisani mehanizmi za medprocesno komunikacijo so se večinoma zanašali na določeno implementacijo. čeprav izgleda, da višji programski jeziki z abstrakcijo prikrivajo razlike v implementaciji, implicitno predpostavljajo da se podatki med procesi prenašajo preko skupnega pomnilniškega območja. Na primer, vsak proces ima pri izvajanju monitorske procedure dostop do monitorjevnih spremenljivk. Pri tem mora biti zagotovljeno vzajemno izključevanje. To se da zagotoviti pri sistemih, ki imajo skupen pomnilnik, implementacija pa je težka pri distribuiranih sistemih, ki so povezani preko vhodnih in izhodnih linij, preko katerih si procesorji izmenjujejo sporočila. S pocenitvjo materialne opreme so se začeli takšni sistemi vse bolj uveljavljati.

Medprocesna komunikacija in sinhronizacija z rendezvous-jem

Hoare in Brinch Hansen sta 1978 leta pokazala, da je pri distribuiranih sistemih bolj naraven pristop k medprocesni komunikaciji, če sinhronizacijo in prenos podatkov med procesi jemljemo kot povezani dejavnosti. Osnovna ideja je prenos podatkov v določenih točkah posameznih procesov. Pred prenosom podatkov se oba procesa sinhronizirata. Ta način komunikacije imenujemo rendezvous. Na primer, če želi proces A poslati podatke procesu B, morata oba procesa soglašati s komunikacijo

tako, da pošljeta zahtevek za oddajanje oziroma za sprejem. če proces A prvi zahteva oddajanje podatkov, mora počakati, da proces B zahteva sprejem podatkov. če proces B prvi zahteva sprejem podatkov, mora počakati, da proces A zahteva oddajanje podatkov. Ko se procesa sinhronizirata, se podatki prenesejo in procesa nadaljujeta izvajanje.

Ta princip rendezvous-ja je izbran pri programskem jeziku Ada, za katerega so 1980 leta izdali standardni priročnik.

Ada nudi bogate možnosti za določanje in obvladovanje paralelnega izvajanja. Da dobimo predstavo o glavnih idejah, bomo v tem poglavju opisali nekatere možnosti. Zaradi jasnosti bomo ponekod podali samo nekatere oblike stavkov.

Za pojasnitev bomo vzeli primer omejenega izravnalnika za celoštevilčne podatke.

```
procedure PRODUCERCONSUMER is

      --specifikacija procesa izravnalnika
task BOUNDEDBUFFER is
  entry APPEND(V: in INTEGER); --deklaracija procesnih
  entry TAKE(V: out INTEGER);  --vhodov s parametri
end BOUNDEDBUFFER;             --konec specifikacije

      --specifikacija procesa, ki pise v izravnalnik
task PRODUCER;

      --specifikacija procesa, ki cita iz izravnalnika
task CONSUMER;

      --telo procesa izravnalnika
task body BOUNDEDBUFFER is
  SIZE: constant:= ...;
  B: array(0..SIZE) of INTEGER;
  INPTR,OUTPTR: INTEGER;
  N: INTEGER;
begin
  N:=0; INPTR:=0; OUTPTR:=0;   --inicializacija
  loop                         --omejenega izravnalnika
      --izbirni stavek
    select
      when N <= SIZE =>        --pogoj (varovalo)
        accept APPEND(V: in INTEGER) do  --prejemni st.
          B(INPTR):=V;         --zapisovanje v izravn.
        end APPEND;
        N:=N+1;
        INPTR:=(INPTR+1) mod SIZE;
    or
      when N > 0 =>            --pogoj
        accept TAKE(V: out INTEGER) do   --prejemni st.
          V:= B(OUTPTR);       --citanje elementa izrav.
        end TAKE;
        N:=N-1;
        OUTPTR:=(OUTPTR+1) mod SIZE;
    end select;                --konec izbirnega stavka
  end loop;
end BOUNDEDBUFFER;
      --telo procesa, ki pise v izravnalnik
task body PRODUCER is
  ELEM: INTEGER;
begin
  loop
    PRODUCE(ELEM);
    APPEND(ELEM);
  end loop;
end PRODUCER;
      --telo procesa, ki cita iz izravnalnika
task body CONSUMER is
  ELEM: INTEGER;
begin
  loop
    TAKE(ELEM);
    CONSUME(ELEM);
  end loop;
end CONSUMER;

      --glavni program
begin
  null;
end PRODUCERCONSUMER;
```

Prenos podatkov pri rendezvous-ju poteka preko parametrov in ne preko izravnalnika. Prenos podatkov med dvema asinhronima procesoma (v našem primeru PRODUCER in CONSUMER) zato izvedemo z uporabo dodatnega procesa (BOUNDEDBUFFER), v katerem programiramo omejeni izravnalnik. Rešitev s pasivnim monitorjem, ki se izvaja samo ko je klican, tu nadomestimo s procesom.

Procesi v Adi so programske enote, ki se izvajajo paralelno (task). Sestavljeni so iz specifikacije procesa in telesa procesa. Procesi se začnejo izvajati takoj po deklaraciji. Zato je v našem primeru glavni program lahko "null" (Ada uporablja "glavno proceduro" kot Pascal glavni program).

Komunikacija med procesi se izvaja na nadzorovan način preko procesnih vhodov (entry). V Adi je implementiran asimetričen rendezvous, pri katerem lahko ločimo kličoči proces (pri nas sta to PRODUCER in CONSUMER) in klicani proces (BOUNDEDBUFFER). Bistvo asimetričnosti je v tem, da v točkah rendezvous-ja samo kličoči proces navaja klicani proces oziroma njegove procesne vhode (APPEND, TAKE). Klicani proces ne navaja, kdo ga kliče, temveč izvede prejemni stavek (accept) na svoje vhode (APPEND, TAKE).

Oglejmo si obliko deklaracij in stavkov. Procesni vhodi so deklarirani v specifikaciji klicanega procesa:

```
entry identifikator [parametri_vhoda];
```

Pri tem so lahko parametri vhoda vhodni (in) ali izhodni (out).

V točki rendezvous-ja kličoči procesi navedejo ime vhoda klicanega procesa:

```
ime_vhoda [(dejanski_parametri)];
```

klicani proces pa navede prejemni stavek (accept):

```
accept ime_vhoda [parametri_vhoda]
  [do stavki
    end [identifikator]];
```

Asimetričnost omogoča, da procese razdelimo na uporabniške (kličoče) in servisne (klicane). Uporabniški procesi morajo poznati vhode servisnih procesov, ki jih kličejo. Servisnim procesom ni potrebno poznati uporabniških procesov. Prejemni stavek servisnega procesa namreč ne navaja imena uporabnika. Zato lahko zgradimo knjižnice servisnih procesov, ki jih lahko uporabljajo vsi uporabniki. Uporabniškim procesom zaradi njihove narave lahko rečemo tudi aktivni procesi, servisnim pa pasivni procesi.

Ada ima na tem področju še dodatno prednost: ker ima klic vhoda enako sintakso kot klic procedure, je lahko klicana operacija izvedena kot proces ali kot navadna sekvenčna procedura.

Na primeru izravnalnika vidimo, da mora imeti proces možnost rendezvous-ja na različnih vhodih, pri tem pa ne pozna vrstnega reda klicanja posameznih vhodov. Ne moremo namreč vnaprej predvideti zaporedja pisanja v izravnalnik in čitanja iz izravnalnika. Potrebujemo možnost nedeterminističnega prenosa podatkov. V Adi nam to možnost nudi izbirni stavek (select), ki omogoča alternativno izbiro med različnimi prejemnimi stavki.

Problem predstavlja tudi omejena velikost izravnalnika. če je izravnalnik poln, ne more sprejeti novega elementa. če je prazen, ne more oddati elementa. Zato pred posamezno alternativo v izbirnem stavku kot varovala postavimo pogoje (when). Takšen izbirni stavek z varovali in izbirami med alternativnimi prejemnimi stavki ima obliko:

```
select
  [when pogoj =>]
     alternativa s prejemnim stavkom
or
  [when pogoj =>]
     alternativa s prejemnim stavkom
.....

[else
     stavki]
end select;
```

Ob izvajanju izbirnega stavka se najprej ovrednotijo vsi pogoji in tako določijo odprte alternative. če je klican nek prejemni stavek v odprtih alternativah, se rendezvous izvede. če ni odprte alternative ali ni klicanih prejemnih stavkov odprtih alternativ in obstaja stavek else, se izvedejo pripadajoči stavki. če obstajajo odprte alternative, noben pripadajoči prejemni stavek pa ni klican in ni stavka else, proces počaka na zahtevo po randezvous-ju. če pa ni odprte alternative ne stavka else nastopi napaka.

Za raznovrstne probleme paralelnega programiranja mehanizem rendezvous-ja nudi primernejše rešitve od mehanizmov na osnovi vzajemnega izključevanja (kritične sekcije, semaforji, monitorji), ki so bolj prilagojeni reševanju določenih problemov. Rendezvous je primeren za implementacijo na enoprocesorskih sistemih, multiprocesorskih sistemih s skupnim pomnilnikom in distribuiranih multiprocesorskih sistemih. Ugotovimo, da rešitve z uporabo rendezvous-ja ponavadi zahtevajo več procesov, kot pri uporabi drugih mehanizmov. Procesi se zato pogosteje izmenjujejo, za kar se porabi določen čas. Pri našem primeru proizvajalca in potrošnika smo tako namesto izravnalnika, implementiranega s pasivnim monitorjem, pri uporabi rendezvous-ja vstavili servisni proces, ki vrši funkcijo izravnalnika. Pri uporabi novejših procesorjev je čas preklopa med procesi tako hiter, da ni kritičen.

2.6 Zaščitni mehanizmi

Videli smo, da monitor lahko definiramo kot abstraktni podatkovni tip. V monitorju so opisane podatkovne strukture in procedure, ki so edine dovoljene operacije nad podatki. V procedurah je zajeta tudi pravilna sinhronizacija med procesi. Procesi kličejo procedure monitorja, pri tem pa nimajo vpliva in jih tudi ne zanima način izvajanja operacij, temveč samo učinek operacij na podatkovne predmete. Vidimo, da monitor

deluje kot nekakšna zaščita podatkovnega predmeta. Pri uporabi izrazov za opis poti smo videli, kako lahko definiramo tudi dovoljeno zaporedje izvajanja operacij nad podatkovnimi predmeti.

Razvoj jezikovnih konceptov se je nadaljeval v smeri abstrakcije podatkovnih predmetov. Definiramo lahko splošni mehanizem za zaščito. Zaščitni mehanizem podatkovnega predmeta je proces, sestavljen iz skupine podatkov, ki jih ščiti in množice procedur za dostop do teh podatkov. Drugi procesi lahko izvajajo operacije nad podatki s klicanjem procedur procesa zaščitnega mehanizma preko njegovih procesnih vhodov.

Pri uporabi zaščitnega mehanizma lahko prevajalnik ugotavlja, ali procesi izvajajo nad podatkovno strukturo samo tiste operacije, ki jih mehanizem dovoljuje. Med izvajanjem programa zaščitni mehanizem izvede, zakasni ali zavrne operacije, ki niso takoj izvedljive zaradi dinamike celotnega sistema.

Vsak mehanizem za komunikacijo in sinhronizacijo med procesi, na primer monitor, izvaja ali zakasni zahtevane operacije. Enak moritorski klic bo vedno sprožil izvajanje enake procedure.

Zaščitni mehanizem lahko izvaja tudi aktivno filtriranje vhodnih klicov. Proces zaščitnega mehanizma izvaja nek program in ima lahko tudi lastne spremenljivke za shranjevanje stanja sistema. V različnih točkah program sprejema klice ostalih procesov na procesnih vhodih. Pri sprejemu klicev pa zaščitni mehanizem lahko izvede različne operacije nad podatkovnim predmetov, odvisno od tega, v katerem delu programa je klic sprejet in v odvisnosti od stanja sistema.

Pri uporabi jezika Ada lahko z elementi jezika zgradimo zaščitni mehanizem nekega podatkovnega predmeta na popoln način, tako da za ta namen ne potrebujemo kakšnega posebnega jezika.

Kot primer lahko vzamemo implementacijo omejenega izravnalnika, ki smo ga podali pri opisu rendezvous-ja. Formirali smo servisni proces (BOUNDEDBUFFER), v katerem smo programirali omejeni izravnalnik, ki ga uporabljamo za prenos podatkov med dvema asinhronima procesoma. Ta servisni proces je hkrati zaščitni mehanizem omejenega izravnalnika. Program procesa je neskončna zanka, v kateri se izvaja izbirni stavek (select). Znotraj izbirnega stavka se lahko izvedeta dva prejemna stavka (APPEND, TAKE) z rendezvous-jem. Ta prejemna stavka sta procesna vhoda. Pri rendezvous-ju na nekem procesnem vhodu ustrezni prejemni stavek izvede določeno operacijo nad omejenim izravnalnikom. To je edini način dostopa drugih procesov do izravnalnika. Če bi bilo potrebno, bi znotraj servisnega procesa zapisali enak prejemni stavek na različnih mestih programa, pri čemer bi se ob posameznem rendevous-ju izvedle različne operacije nad podatkovnim predmetom.

3 Zaključek

Paralelno programiranje uvedemo zato, da enostavneje in hitreje rešimo neko nalogo, ki ima možnost paralelnega izvajanja in da zvečamo izkoriščenost sistema. Če paralelni procesi med sabo sodelujejo, moramo zagotoviti pravilen način interakcije med procesi. Uporabimo lahko več ustreznih primitivov, kot so kritične sekcije, semaforji, monitorji, rendezvous-ji itd. Več jezikov vsebuje jezikovne konstrukte za deklariranje procesov in medprocesno komunikacijo in sinhronizacijo z opisanimi primitivi. Večina takšnih jezikov odraža stanje materialne opreme v času razvoja in podpira primitive (semaforji, kritične sekcije, pogojne kritične sekcije, monitorji), primerne za implementacijo na enoprocesorskih sistemih ali večprocesorskih sistemih s skupnim pomnilnikom. Takši jeziki so na primer Concurrent Pascal in Modula. Pri distribuiranih večprocesorskih sistemih, pri katerih so posamezni procesorji povezani preko vhodno-izhodnih linij, pa je nastal problem implementacije teh primitivov. V novejšem jeziku Ada so komunikacijo in sinhronizacijo med procesi povezali v mehanizmu rendezvous-ja, ki ga lahko enostavno implementiramo tudi v distribuiranih sistemih.

Uporaba višjih programskih jezikov poenostavi in skrajša čas razvoja programov. Pri tem nastopi problem učinkovitosti prevedene kode. Večina današnjih arhitektur ne podpira učinkovito abstraktne jezike v primerjavi s strojno kodo. Pri programih, ki morajo delati v realnem času smo zato soočeni z izbiro med učinkovitostjo, ceno in zanesljivostjo. Zato je prisoten trend razvijanja računalniških arhitektur, ki direktno podpirajo koncepte programskih jezikov. V procesorskih elementih se z materialno opremo oziroma mikroprogrami implementirajo specifične funkcije jezikov za paralelno izvajanje procesov: preklapanje procesorja med procesi, medprocesna komunikacija in sinhronizacija in podobne. S tem se občutno zveča hitrost izvajanja teh funkcij. Takšne funkcije ima na primer VAX procesor. Po drugi strani razvijajo sisteme z več procesorji, pri katerih posamezni procesi tečejo na lastnih procesorjih, s čemer se zveča hitrost izvajanja paralelnih programov. Takšno izvajanje na primer podpira arhitektura sistema iAPX 432.

Lahko povzamemo, da so nam danes na razpolago jeziki z jezikovnimi elementi, primernimi za paralelno programiranje, razvijajo pa se arhitekture, ki podpirajo učinkovito implementacijo jezikov na sistemih z več procesorskimi elementi.

Pri obravnavanju računalniških jezikov smo v članku upoštevali imperativne jezike. Programi v funkcionalnih jezikih vsebujejo visoko stopnjo inherentne paralelnosti, načine automatskega izkoriščanja in učinkovite implementacije te paralelnosti na večprocesorskih sistemih pa še raziskujejo.

Literatura

/1/ M. Ben-Ari, "Principles of Concurrent Programming", Prentice/Hall International, London, 1982

/2/ R.C. Holt, G.S. Graham, E.D. Lazowska, M.A. Scott, "Structured Concurrent Programming with Operating Systems Applications", Addison-Wesley, Reading, Mass., 1978

/3/ P. Brinch Hansen, "A Keynote Address on Concurrent Programming", IEEE Computer, May 1979, pp. 50-56

/4/ D.A. Anderson, "Operating Systems", IEEE Computer, June 1981, pp. 69-82

/5/ S.J. Young, "Real Time Languages: Design and Development", John Wiley & Sons, New York, 1982

/6/ I.C. Pyle, "The Ada Programming Language", Prentice/Hall International, 1981

/7/ M. Exel, F. Prijatelj, "Programiranje sprotnih in vgnezdenih sistemov: procesi v Adi", Informatica, No. 2, 1981, pp. 8-18

/8/ G. Štrkić, D. Novosel, "O jezicima za konkurentno programiranje kao sredstvu za projektovanje upravljačkih sistema", Informatica, No. 2, 1985, pp. 16-18

/9/ M. Kapus, "Pregled jezikovnih elementov za opis sinhronizacije paralelnih procesov", Informatica, No. 1, 1982, pp. 71-77

/10/ E.T. Fathi, M. Krieger, "Multiple Microprocessor Systems: What, Why and When", IEEE Computer, March 1983, pp. 23-32

/11/ J.L. Hennessy, "VLSI Processor Architecture", IEEE Trans. on Computers, Vol. C-33, No. 12, December 1984, pp. 1221-1246

PRIMJENA MEHANIČKOG DOKAZIVANJA TEOREMA U RJEŠAVANJU IMPLIKACIONOG PROBLEMA ZA T–ZAVISNOSTI U RELACIONIM BAZAMA PODATAKA

INFORMATICA 2/86

Mirko Maleković
CVVTŠ „General arm. Ivan Gošnjak", Zagreb

UDK: 681.3.01:519

Klasa T-zavisnosti sadrži, kao specijalne slučajeve, višeznačne zavisnosti, zavisnosti spajanja, te podskup-zavisnosti. U ovom radu, rješavajući implikacione probleme za pravila formalnog sistema za T-zavisnosti, dokazujemo točnost navedenog formalnog sistema. Rješavanje se bazira na primjeni rezolucijskih procedura dokazivanja.

THE APPLICATION OF MECHANICAL THEOREM PROVING TO IMPLICATION PROBLEM SOLVING FOR T-DEPENDENCIES IN RELATIONAL DATABASES: The class of T-dependencies includes the following dependency classes as special cases: multivalued dependencies, join dependencies and subset-dependencies. In this work we have solved implication probleme for rules of formal system for T-dependencies, that is we have proved soundness of formal system for T-dependencies. The method is based on application of resolution proof-procedures.

0. Uvod

Centralni problem u dizajniranju relacione baze podataka jeste kako izabrati "dobar" skup relacionih šema. Ovdje, pod dobrim skupom relacionih šema smatramo onaj skup koji zadovoljava uvjete čuvanja zavisnosti i čuvanja informacije. Osim toga, zahtjeva se nepostojanje anomalija brisanja, upisivanja i ažuriranja. Navedeni problem vodi na implikacioni problem za danu familiju zavisnosti, što je rješavano u [1],[2],[3],[5],[7],[8], [9],[10],[11],[12]. U ovom radu, tretiramo T-zavisnosti uvedene u [12]. Rješavajući implikacione probleme za pravila formalnog sistema, predloženog također u [12], dokazujemo njegovu točnost. Postupak rješavanja se bazira na primjeni rezolucijskih procedura dokazivanja. Organizacija članka je kao što slijedi; Osim uvodnog dijela i zaključka, rad ima četiri sekcije. U prvoj sekciji uvodimo bazične pojmove, vezane za relacioni model, koje ćemo upotrebljavati u preostalim sekcijama. U sekciji 2. karakteriziramo T-zavisnosti. Reprezentacija T-zavisnosti pomoću Skolemove standardne forme dana je u sekciji 3. Implikacioni problemi za pravila formalnog sistema predloženog u [12], riješeni su u sekciji 4. Pretpostavljamo poznavanje teorije baza podataka i rezolucijskih procedura dokazivanja na nivou [13], odnosno [4].

1. Bazični pojmovi

Neka je U_∞ beskonačan skup apstraktnih elemenata. Elemente skupa U_∞ zovemo atributi. Dalje, neka je $\mathcal{F} = \{D_A / A \in U_\infty\}$ familija nepraznih skupova, gdje D_A zovemo domenom od atributa A.

Definicija Neka je $R \subseteq U_\infty$ konačan, neprazan skup, $D = \bigcup_{A \in R} D_A$. Tipl u je funkcija $u: R \to D$, sa svojstvom $u(A) \in D_A \forall A \in R$. Sa $Tipl(R)$ označavamo skup svih tiplova nad R.

Definicija Relacija je uređen par (R,r), gdje je $r \subseteq Tipl(R)$ konačan skup. R zovemo relacionom šemom, a r primjerom relacione šeme.

U slučaju da se relaciona šema podrazumijeva, relaciju (R,r) ćemo označavati kraće sa r. Uređenje relacione šeme omogućuje reprezentiranje relacije r pomoću tabele; redovi tabele su tiplovi elementi od r, a stupci su imenovani atributima iz R. Slijedeći uobičajenu notaciju u teoriji baza podataka, jednočlan skup $\{A\}$ pišemo kao A, uniju skupova $X \cup Y$, kao XY, a komplement skupa X u odnosu na R, $R \setminus X$, kao $\overline{X}$.

2. T-zavisnosti

U ovoj sekciji karakteriziramo T-zavisnosti. Pridružimo svakom $A \in R$, osim D_A, i skup D'_A apstraktnih simbola. Skup D'_A zovemo aps-

traktna domena. Pretpostavljamo, $D'_A \cap D'_B = \emptyset$ za $A \neq B$.

Definicija Neka je $D' = \bigcup_{A \in R} D'_A$. TD-element je funkcija $v: R \to D'$ sa svojstvom $v(A) \in D'_A \; \forall A \in R$

Iz definicije vidimo da je TD-element analogon tipla. Neka je
(1) $r_i = v_i(A_1)\, v(A_2) .. v_i(A_k)$, $i = 1,..,n+1$;
gdje je $R = \{A_1, A_2, .., A_k\}$, a v_i je TD-element.

Definicija T-zavisnost je izraz oblika $t: (r_1,..,r_n)/r_{n+1}$. t je ime T-zavisnosti, $r_1,..,r_n$ su hipoteze, a r_{n+1} je zaključak.

Identifikacijom TD-elementa v_i i reda r_i, $i = 1,..,n+1$ (1) pišemo u obliku
(2) $r_i = r_i(A_1)..(A_k)$, $i = 1,..,n+1$.
T-zavisnost $t: (r_1,..,r_n)/r_{n+1}$ predstavljamo tabelom čiji su redovi $r_1, r_2, .., r_{n+1}$, a stupci su imenovani atributima iz R. Posljednji red je zaključni TD-element.
Neka je $t: (r_1,..,r_n)/r_{n+1}$ T-zavisnost nad R. Uvedimo skupove $S_t(i,j) = \{A \in R / \; r_i(A) = r_j(A)\}$, $i,j = 1,..,n+1$.
Uočimo da je $S_t(i,j) = S_t(j,i) \; \forall i,j \in \{1,..,n+1\}$, te $S_t(i,i) = R \; \forall i \in \{1,..,n+1\}$.

Definicija Neka je $t: (r_1,..,r_{n+1})/r_{n+1}$ T-zavisnost nad R. Kažemo da relacija (R,r) zadovoljava t, ili da t vrijedi u (R,r), ako i samo ako
(3) $\forall t_1,..,t_n \in r \left[\forall i,j \in \{1,..,n\} \; E_{S_t(i,j)}(t_i,t_j) \Rightarrow \right.$
$\left. \Rightarrow \exists t_{n+1} \in r \left[\forall i \in \{1,..,n\} \; E_{S_t(n+1,i)}(t_{n+1}, t_i) \right] \right]$.

U formuli (3), $E_{S_t(i,j)}(t_i,t_j)$ znači jednakost tiplova t_i i t_j na skupu atributa $S_t(i,j)$.

Iskazana definicija kaže da r zadovoljava TD-zavisnost $t: (r_1,..,r_n)/r_{n+1}$ ako i samo ako vrijedi: kad god su hipoteze od t (poslije proizvoljnog preimenovanja) u r, onda je i zaključak od t (poslije odgovarajućeg proširenja ovog preimenovanja) također u r.

3. Skolemova standardna forma za T-zavisnosti

U ovoj sekciji reprezentiramo T-zavisnost u standardnoj formi. U sekciji 2. smo rekli da T-zavisnost $t: (r_1,..,r_n)/r_{n+1}$ vrijedi u relaciji r ako i samo ako vrijedi (3).
Ako sa $\bigwedge_{i,j=1}^{n} E_{S_t(i,j)}(t_i,t_j)$ označimo konjunkciju formula $E_{S_t(i,j)}(t_i,t_j)$ po svim parovima $i,j \in \{1,..,n\}$, formula (3) poprima oblik
$$(4) \; \forall t_1 .. \forall t_n \left[\bigwedge_{i,j=1}^{n} E_{S_T(i,j)}(t_i,t_j) \Rightarrow \right.$$
$$\left. \Rightarrow \exists \; t_{n+1} \left[\bigwedge_{k=1}^{n} E_{S_t(n+1,i)}(t_{n+1}, t_k) \right] \right].$$
Ako, interpretirajući tipl-varijable $t_1,..,t_{n+1}$ na r, (4) postane istinita propozicija, onda kažemo da (4) vrijedi u r, odnosno da je r model za (4).
Označavajući sa $\bigvee_{i,j=1}^{n} E_{S_t(i,j)}(t_i,t_j)$ disjunkciju formula $E_{S_t(i,j)}(t_i,t_j)$ po svim parovima $i,j \in \{1,..,n\}$, te standardizacijom formule (4) nalazimo Skolemovu standardnu formu za T-zavisnost t:

$$s(t): \begin{cases} (1) \bigvee_{i,j=1}^{n} \sim E_{S_t(i,j)}(t_i,t_j) \vee E_{S_t(n+1,1)}(f(t_1,..,t_n), t_1) \\ \cdot \\ \cdot \\ (n) \bigvee_{i,j=1}^{n} \sim E_{S_t(i,j)}(t_i,t_j) \vee E_{S_t(n+1,n)}(f(t_1,..,t_n), t_n) \end{cases}$$

Dalje, negirajući (4), te stadardizacijom dobivamo standardnu formu za $\sim t$:

$$s(\sim t): \begin{cases} (1) \bigwedge_{i,j=1}^{n} E_{S_t(i,j)}(a_i,a_j) \\ (2) \bigvee_{k=1}^{n} \sim E_{S_t(n+1,k)}(t_{n+1}, a_k) \end{cases}$$

4. Točnost formalnog sistema za T-zavisnosti

U ovoj sekciji, dokazujemo točnost formalnog sistema za T-zavisnosti predloženog u [12]. Dokazi se baziraju na rješavanju implikacionih problema za pravila formalnog sistema. Točnost formalnog sistema se karakterizira preko logičke konzekvence.

Definicija Neka je C skup zavisnosti za relacionu šemu R. Neka je c pojedinačna zavisnost. Kažemo da je c logička konzekvenca od C, ili da C logički implicira c, u oznaci $\frac{C}{c}$, ako i samo ako svaki primjer (relacija) od R koji zadovoljava C također zadovoljava i c. Kažemo, dalje, da primjer r od R zadovoljava skup zavisnosti C za R ako i samo ako r zadovoljava svaku zavisnost iz C.

Formalni sistem teorije zavisnosti se sastoji od pravila (aksioma) koja omogućuju izvođenje novih zavisnosti iz zadanih zavisnosti.

Definicija Neka je F_D formalni sistem, C i c kao u prošloj definiciji. Kažemo da C dokazuje c ako i samo ako je moguće upotrebom aksioma iz F_D na zavisnosti iz C izvesti

zavisnost c . Dokazivost c iz C u formalnom sistemu F_D označavat ćemo sa $C \vdash_{F_D} c$. Slijedi definicija točnosti formalnog sistema

Definicija Za formalni sistem F_D kažemo da je točan ako i samo ako za bilo koji skup zavisnosti C i za bilo koju pojedinačnu zavisnost c vrijedi $C \vdash_{F_D} c \Longrightarrow \frac{C}{c}$.

Formalni sistem za T-zavisnisti, u oznaci F_{TD}, će sastoji od slijedećih pravila:

TD1: $t:(r_1,..,r_n)/r_{n+1} \vdash t':(g(r_1),..,g(r_n))/g(r_{n+1})$.

U navedenom pravilu, imamo da je $g:D' \longrightarrow D'$ sa slijedećim svojsvima:

(a) $g(r_i(A)) \in D'_A \ \forall i \in \{1,..,n+1\} \forall A \in R$.

(b) Ako za neke simbole a iz r_{n+1} i b iz r_i za neki $i=1,..,n$, vrijedi $g(a)=g(b)$, onda postoji j , $1 \leq j \leq n$, takav da je a iz r_j .

Zahtjev (b) kaže da g ne identificira bilo koji simbol koji se pojavljuke samo u r_{n+1} sa drugim simbolom. Zavisnost t zovemo preimenovanjem zavisnosti t , a pravilo TD1 zovemo preimenovanje i identifikacija simbola.

TD2: $t:(r_1,..,r_n)/r_{n+1} \vdash t':(r_o,r_1,..,r_n)/r_{n+1}$,

uz uvjet da vrijedi:

(c) $r_o(A)=a \Longrightarrow \left[a \in D'_A \wedge (r_{n+1}(A)=a \Rightarrow \exists i \in \{1,..,n\} r_i(A)=a)\right]$.

Zahtjev (c) nam kaže da r_o ne sadrži niti jedan simbol iz r_{n+1} , osim ako se taj simbol ne pojavljuje u nekom r_i , $i=1,..,n$.
Pravilo TD2 se zove proširenje (zavisnosti).

TD3: $t:(r_1,..,r_n)/r_{n+1} \vdash t':(r_1,..,r_n)/p(r_{n+1})$,

gdje za preslikavanje p vrijede svojstva:

(d) $p(r_{n+1}(A)) \in D'_A \forall A \in R$.

(e) $p(r_{n+1}(A))=r_{n+1}(A) \vee (p(r_{n+1}(A))=a \wedge \wedge \forall i \in \{1,..,n+1\} \ p(r_i(A)) \neq a) \forall A \in R$.

Pravilo TD3 se zove oslabljenje, a za zavisnost t' kažemo da je oslabljenje od zavisnosti t .

TD4: $t_1:(r_1,..,r_{n-1})/r_n$, $t_2:(r_1,..,r_n)/r_{n+1} \vdash$
$\vdash t:(r_1,..,r_{n-1})/r_{n+1}$.

Pravilo TD4 predstavlja tranzitivnost za T-zavisnosti.

TD5: $t:(r/r)$ vrijedi u bilo kojoj relaciji, za bilo koji red r .

Pravilo TD5 zovemo trivijalna T-zavisnost.

Prijeđimo sada na dokaz točnosti navedenih pravila.

Dokaz za TD1:

Trebamo dokazati $\dfrac{t:(r_1,..,r_n)/r_{n+1}}{t':(g(r_1),..,g(r_n)/g(r_{n+1})}$,

gdje g ima svojstva (a) i (b) u TD1 . Standardizacijom $t \wedge (\sim t')$ dobivamo skup rečenica S :

S

(1) $\bigvee_{i,j=1}^{n} \sim E_{S_t(i,j)}(t_i,t_j) \vee E_{S_t(n+1,1)}(f(t_1,..,t_n),t_1)$

.

(n) $\bigvee_{i,j=1}^{n} \sim E_{S_t(i,j)}(t_i,t_j) \vee E_{S_t(n+1,n)}(f(t_1,..,t_n),t_n)$

(n+1) $\bigwedge_{i,j=1}^{n} E_{S_t'(i,j)}(a_i,a_j)$

(n+2) $\bigvee_{k=1}^{n} \sim E_{S_t'(n+1,k)}(t_{n+1},a_k)$

Skup S proširujemo pravilima koja karakteriziraju odnos skupova $S_t(i,j)$ i $S_t(i,j)$, te svojstvo funkcije g .

Pravilo (A): $S_t(i,j) \subseteq S_t(i,j) \forall i,j \in \{1,..,n+1\}$.

Iz pravila (A) dobivamo rečenicu

(n+3) $\sim E_{S_t'(i,j)}(t_i,t_j) \vee E_{S_t(i,j)}(t_i,t_j)$

Pravilo (B): $\forall A \in R \forall j \in \{1,..,n\} \left[A \in S_t'(n+1,j) \Rightarrow \right.$
$\left. \Rightarrow \exists k \in \{1,..,n\} (A \in S_t(n+1,k) \wedge r_k(A)=r_j(A))\right]$.

Pišući $A \in S_t(i,j)$ kao $S_t(i,j)(A)$, te standardizirajući pravilo (B) nalazimo rečenice:

(n+4) $\sim S_t'(n+1,j)(A) \vee S_t(n+1,f(A,j)(A)$

(n+5) $\sim S_t'(n+1,j)(A) \vee E_A(a_{f(A,j)},a_j)$

Sada pokazujemo kontradiktornost skupa $S'=S \cup \{(n+3), (n+4), (n+5)\}$. Stablo dokaza je dano na slici 1.

Dokaz za TD2:

Trebamo dokazati $\dfrac{t:(r_1,..,r_n)/r_{n+1}}{t':(r_o,r_1,..,r_n)/r_{n+1}}$.

Standardizacijom $t \wedge \sim t'$ nalazimo skup rečenica S:

S

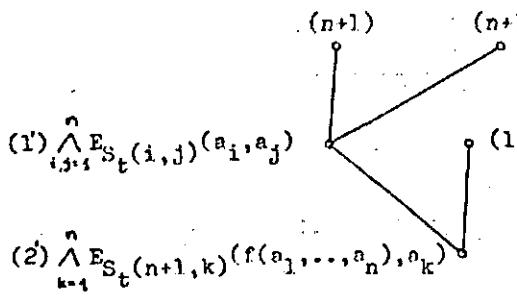
(1) $\bigvee_{i,j=1}^{n} \sim E_{S_t(i,j)}(t_i,t_j) \vee E_{S_t(n+1,n)}(f(t_1,..,t_n),t_1)$

.

(n) $\bigvee_{i,j=1}^{n} \sim E_{S_t(i,j)}(t_i,t_j) \vee E_{S_t(n+1,n)}(f(t_1,..,t_n),t_n)$

(n+1) $\bigwedge_{i,j=0}^{n} E_{S_t'(i,j)}(a_i,a_j)$

(n+2) $\bigvee_{i=0}^{n} \sim E_{S_t'(n+1,j)}(t_{n+1},a_j)$



/Cilj je da pokažemo da vrijedi (3'): $\bigwedge_{j=1}^{n} E_{S_t(n+1,j)}(f(a_1,..,a_n),a_j)$ /

Rečenica (3') je ekvivalentna sa (4') $\forall A \forall j \left[S_t(n+1,j)(A) \Rightarrow E_A(f(a_1,..,a_n),a_j) \right]$.

Sada pokazujemo da je (4') logička konzekvenca od pravila (B) tj. rečenica (n+4) i (n+5) . Negiranjem rečenice (4') dobivamo

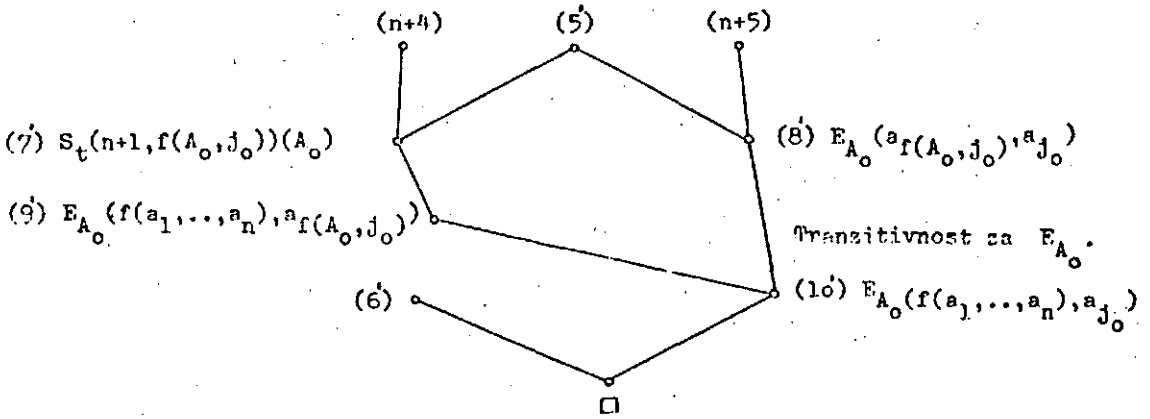
$\sim$(4'): $\exists A \exists j \left[S_t(n+1,j)(A) \wedge \sim E_A(f(a_1,..,a_n),a_j) \right]$.

Standardizacijom iz $\sim$(4') nalazimo:

(5') $S_t(n+1,j_o)(A_o)$

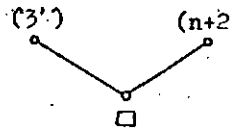
(6') $\sim E_{A_o}(f(a_1,..,a_n),a_{j_o}))$

Sada nastavljamo rezoluciju:



/ Izveli smo (3') $\bigwedge_{j=1}^{n} E_{S_t(n+1,j)}(f(a_1,..,a_n),a_{j_o})$ /

Konačno,



Sl. 1.

Skup rečenica S proširujemo pravilom (A) iz dokaza TD1 . Dobivamo rečenicu

(n+3) $\sim E_{S_t(i,j)}(t_1,t_j) \vee E_{S_t(i,j)}(t_i,t_j)$

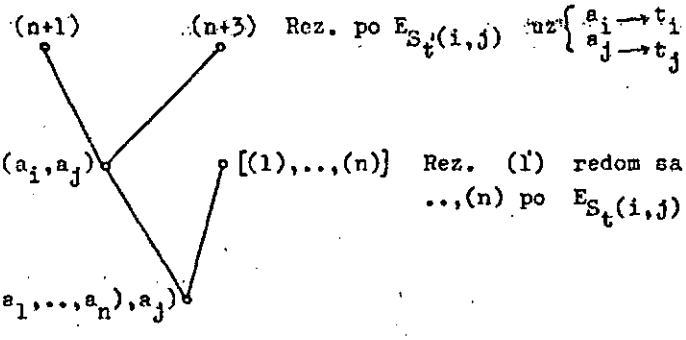
Također, trebamo pravilo koje karakterizira TD-element r_o .

Pravilo (C):
$\forall A \left[E_A(r_{n+1},r_o) \Rightarrow \exists i \in \{1,..,n\} (E_A(r_i,r_{n+1}) \wedge \right.$
$\left. \wedge E_A(r_i,r_o)) \right]$.

Standardizacijom pravila (C) dobivamo rečenice:

(n+4) $\sim E_A(r_{n+1},r_o) \vee E_A(r_{n+1},r_{f(A)})$

(n+5) $\sim E_A(r_{n+1},r_o) \vee E_A(r_{f(A)},r_o)$. Prelazimo na dokaz kontradiktornosti skupa $S' = S \cup \{(n+3), (n+4), (n+5)\}$. Stablo dokaza dajemo na slici 2.



/ Koristeći dokaz za TDl , znamo da iz (2') slijedi

(3') $\bigwedge_{j=1}^{n} E_{S_t'(n+1,j)}(f(a_1,..,a_n),a_j)$ /

Slijedeći cilj je da pokažemo da vrijedi:

(4') $E_{S_t'(n+1,o)}(f(a_1,..,a_n),a_o)$

Rečenica (4') je ekvivalentna sa

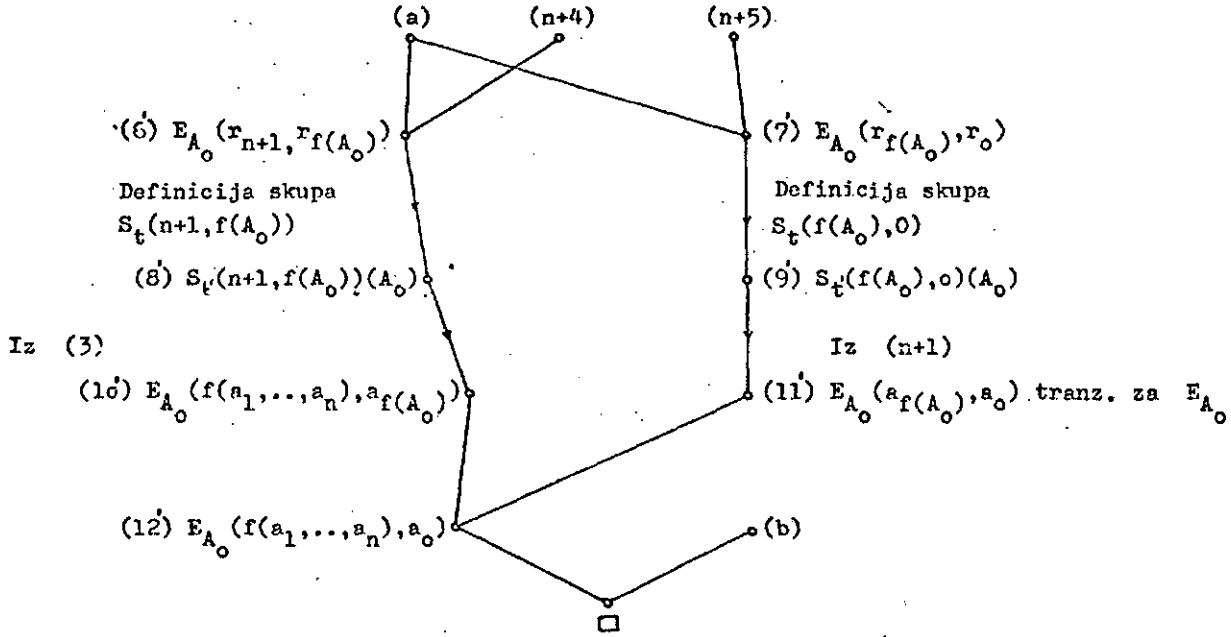
(5') $\forall A\left[S_t'(n+1,o)(A) \Rightarrow E_A(f(a_1,..,a_n),a_o)\right]$.

Sada ćemo pokazati da (5') slijedi iz pravila (C) tj. rečenica (n+4) i (n+5), te rečenica (3) i (n+1). Trebamo pokazati da je skup S= {(5), (n+4), (n+5), (3), (n+1)} kontradiktoran.

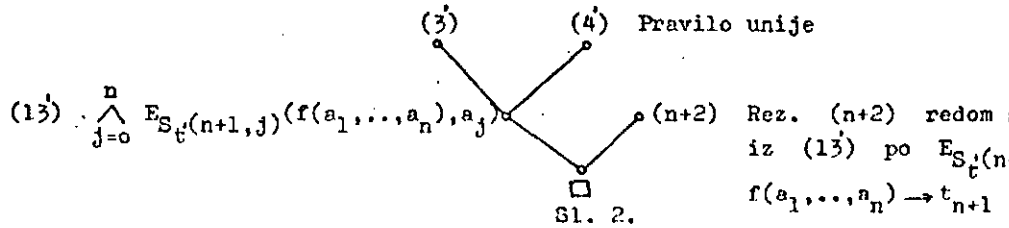
$\sim(5'):$ (a) $E_{A_o}(r_{n+1},r_o)$

(b) $\sim E_{A_o}(f(a_1,..,a_n),a_o)$

Dokaz kontradiktornosti skupa S" je kao što slijedi:



Dokazali smo da vrijedi (4') . Slijedi kompletiranje dokaza.



Sl. 2.

Prelazimo na dokaz pravila TD3 tj. dokazujemo da vrijedi: $\dfrac{t:(r_1,..,r_n)/r_{n+1}}{t:(r_1,..,r_n)/p(r_{n+1})}$

Standardizacijom formule $t \wedge \sim t$ nalazimo skup rečenica S:

S

(1) $\bigvee_{i,j=1}^{n} \sim E_{S_t(i,j)}(t_i,t_j) \vee E_{S_t(n+1,1)}(f(t_1,..,t_n),t_1)$

(n) $\bigvee_{i,j=1}^{n} \sim E_{S_t(i,j)}(t_i,t_j) \vee E_{S_t(n+1,n)}(f(t_1,..,t_n),t_n)$

(n+1) $\bigwedge_{i,j=1}^{n} E_{S_{t'}(i,j)}(a_i,a_j)$

(n+2) $\bigvee_{j=1}^{n} \sim E_{S_t(n+1,j)}(t_{n+1},a_j)$

Na osnovi zadanih zavisnosti t i t' , odnosno na osnovi svojstava preslikavanja p , imamo dva pravila:

P1: $S_t(i,j)=S_{t'}(i,j)$, $i,j=1,..,n$.

P2: $S_{t'}(n+1,j) \subseteq S_t(n+1,j)$, $j=1,..,n$.

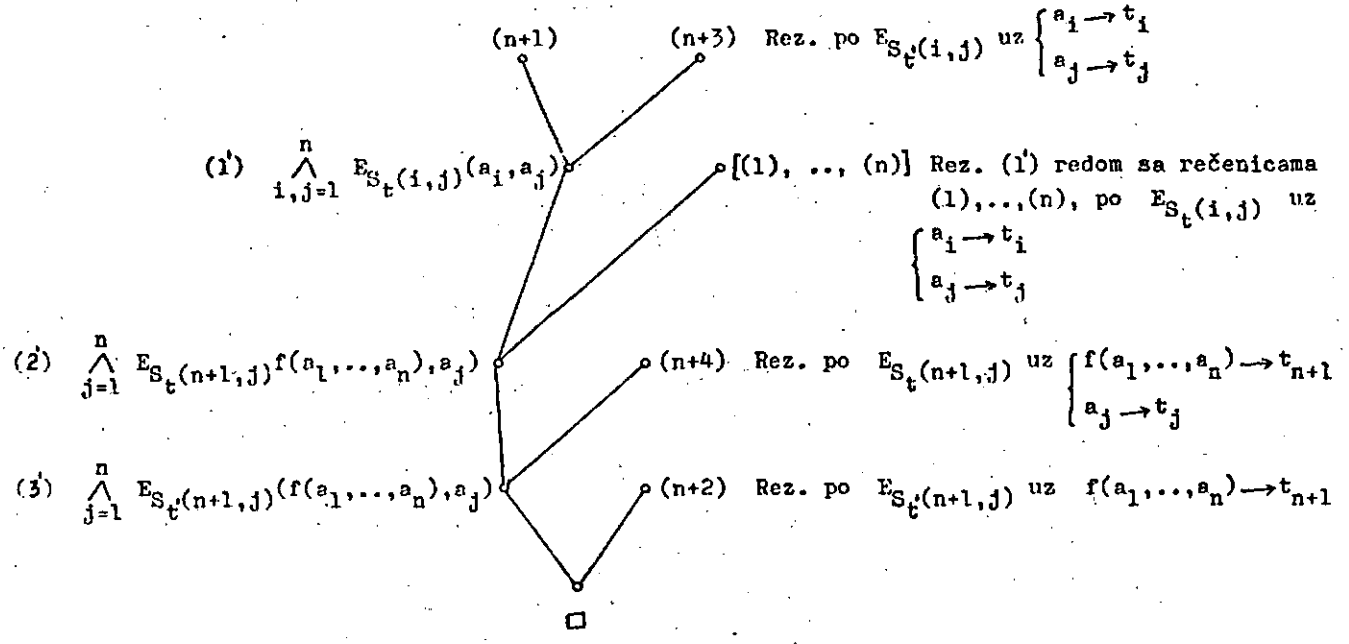
Iz pravila P1 nalazimo rečenicu:

(n+3) $\bigwedge_{i,j=1}^{n} (\sim E_{S_t(i,j)}(t_i,t_j) \vee E_{S_{t'}(i,j)}(t_i,t_j))$.

Iz pravila P2 dobivamo rečenicu:

(n+4) $\bigwedge_{j=1}^{n} (\sim E_{S_t(n+1,j)}(t_{n+1},t_j) \vee E_{S_{t'}(n+1,j)}(t_{n+1},t_j))$

Stablo dokaza kontradiktornosti skupa $S'=S \cup \{(n+3), (n+4)\}$, dano je na slici 3.



Sl. 3.

Na kraju ove sekcije , dokazujemo pravilo tranzitivnosti:

TD4: $\dfrac{t:(r_1,..,r_{n-1})/r_n;\ t_2:(r_1,..,r_n)/r_{n+1}}{t:(r_1,..,r_{n-1})/r_{n+1}}$

Transformiramo $t_1 \wedge t_2 \wedge \sim t$ u standardnu formu te dobivamo skup rečenica S :

$$
\left.
\begin{array}{ll}
(1) & \bigvee_{i,j=1}^{n-1} E_{S_{t_4}(i,j)}(t_i,t_j) \vee E_{S_{t_4}(n,1)}(f(t_1,..,t_{n-1}),t_1) \\
(n-1) & \bigvee_{i,j=1}^{n-1} \sim E_{S_{t_4}(i,j)}(t_i,t_j) \vee E_{S_{t_4}(n,n-1)}(f(t_1,..,t_{n-1}),t_{n-1}) \\
(1') & \bigvee_{i,j=1}^{n} \sim E_{S_{t_2}(i,j)}(t_1,t_j) \vee E_{S_{t_2}(n+1,1)}(g(t_1,..,t_n),t_1) \\
(n') & \bigvee_{i,j=1}^{n} \sim E_{S_{t_2}(i,j)}(t_1,t_j) \vee E_{S_{t_2}(n+1,n)}(g(t_1,..,t_n),t_n) \\
(1'') & \bigwedge_{i,j=1}^{n-1} E_{S_t(i,j)}(a_i,a_j) \\
(2'') & \bigvee_{j=1}^{n-1} \sim E_{S_t(n+1,j)}(t_{n+1},a_j)
\end{array}
\right\} S
$$

Skup S dopunjujemo pravilom:
P: $S_t(i,j)$, $S_{t_1}(i,j)$ i $S_{t_2}(i,j)$ su poparno jednaki kad god su definirani. Dokaz je dan na slici 4., kao što slijedi:

Iz pravila P i rečenice (1'') dobivamo

(3'') $\bigwedge_{i,j=1}^{n-1} E_{S_{t_1}(i,j)}(a_i,a_j)$

(3'') [(1),..,(n-1)] Rez. (3'') redom sa rečenicama (1),..,(n-1)

(4'') $\bigwedge_{j=1}^{n-1} E_{S_{t_1}(n,j)}(f(a_1,..,a_{n-1}),a_j)$ Iz pravila P i rečenice (4'') nalazimo:

(5'') $\bigwedge_{j=1}^{n-1} E_{S_{t_2}(n,j)}(f(a_1,..,a_{n-1}),a_j)$, a iz pravila P i rečenice (1) dobivamo:

(6'') $\bigwedge_{i,j=1}^{n-1} E_{S_{t_2}(i,j)}(a_i,a_j)$ Uz oznake $w_1=a_1,..,\ w_{n-1}=a_{n-1},\ w_n=f(a_1,..,a_{n-1})$, iz (5'') i (6'') po pravilu unije slijedi:

(7'') $\bigwedge_{i,j=1}^{n} E_{S_{t_2}(i,j)}(w_i,w_j)$ Dalje, nastavljamo dokaz:

(7'') [(1'),..,(n')] Rez. (7'') redom sa rečenicama (1'),..,(n')

(8'') $\bigwedge_{j=1}^{n} E_{S_{t_2}(n+1,j)}(g(a_1,..,a_{n-1},f(a_1,..,a_{n-1})),a_j)$

Iz pravila P imamo $S_{t_2}(n+1,k)=S_t(n+1,k)$, $k=1,..,n-1$, pa iz (8'') dobivamo:

(9'') $\bigwedge_{j=1}^{n-1} E_{S_t(n+1,j)}(g(a_1,..,a_{n-1},f(a_1,..,a_{n-1}),a_j)$. Konačno,

(9'') (2'') Rez. po $E_{S_t(n+1,j)}$ uz $g(a1,..,a_{n-1},f(a_1,..,a_{n-1})) \to t_{n+1}$

Sl. 4.

5. Zaključak

U ovom radu, razmatrali smo T-zavisnosti u relacionim bazama podataka. Rješavajući implikacione probleme, za pravila formalnog sistema predloženog u[12] dokazali smo točnost navedenog formalnog sistema. Kao i u ranijim radovima, koji se odnose na funkcionalne, višeznačne i podskup zavisnosti, i ovdje, metod rješavanja se bazira na primjeni rezolucijskih procedura dokazivanja. Navedenim radovima, pokazali smo mogućnost tretiranja implikacionog problema, za skoro sve najvažnije zavisnosti u relacionim bazama podataka, u okviru jedinstvenog konceptualnog aparata koji daje teorija mehaničkog dokazivanja teorema.
Kuda dalje? Sintetizirajući formule koje reprezentiraju spomenute zavisnosti i pravila koja smo upotrebljavali u proširenju skupa S, pravila koja karakteriziraju pojedine zavisnosti, možemo dobiti deduktivnu bazu podataka koja bi mogla predstavljati podršku u automatizaciji dizajna relacionih šema, kao što se razmatra u [6].

Literatura

1. Aho, A. V., Beeri, C., Ullman, J.D.: The theory of joins in relational databases. ACM Trans. Database Syst. 4,3 (Sept. 1979), 297-314.

2. Armstrong, W.W., Delobel, C.: Decompositions and functional dependencies in relations. ACM Trans. Database Syst. 5,4 (Dec. 1980), 404-430.

3. Beeri, C., Vardi, M.Y.: A proof procedure for data dependencies. J. ACM 31,4 Oct. 1984), 718-741.

4. Chang,C.L., Lee, R.C.T.: Symbolic Logic and Mechanical Theorem Proving, Compt. Sci. Apll. Math. Academic Press, 1973.

5. Fagin, R. Horn clauses and database dependencies. J. ACM 29,4(Oct. 1982), 952-985.

6. Gallaire, H., Minker, J., Nicolas, J.M.: Advances in Data Base Theory, vol.1. Plenum Press, New York, 1981.

7. Maier, D., Mendelzon, A.O. and Sagiv, Y.: Testing implications of data dependencies ACM Trans. Database Syst. 4.4(Dec. 1979) 445-469.

8. Maier, D., Mendelzon, A.O., Sadri, F., and Ullman, J. D.: Adequasy of decompositions of relational databases. In Advances in Database Theory, H. Gallaire, J. Minker, and J.M. Nicolas, Eds. Plenum Press, New York, 1981., 101-114.

9. Mendelzon, A.O.: On axiomatizing multivalued dependencies in relational databases. J. ACM 26.1(Jan. 1979), 37-44.

10. Paredaens, J., and Jannsens, D.: Decompositions of relations: A comprehensive approach. In Advances in Database Theory, H. Gallaire, J. Minker, and J.M. Nicolas, Eds. Plenum Press, New York, 1981.,101-114.

11. Sciore, E. A complete axiomatizations of full join dependencies. J. ACM 29.2(Apr. 1982), 373-393.

12. Sadri, F., and Ullman, J.D.: Template Dependencies: A large Class of Dependencies in Relational Databases and its Complete Axiomatization. J. ACM 29, 2(Apr. 1982), 363-373.

13. Ullman, J. D.: Principles of Database Systems. Computer Science Press, Potomac, Md., 1980.

Od Sappora do Tokia nazaj v Ljubljano

Anton P. Železnikar

Ari no michi kumo no mine yori tsuzukiken

Mravljična steza se spušča z daljnjih vrhov oblakov

(Kobayashi Issa)

1. Uvod

Pred nekako desetimi leti, ko je postala nesposobnost že premisa in organizacija znanstvenoraziskovalnega dela, sem jasno občutil potrebo, da bi se odpravil na Japonsko. Njen gospodarski napredek kot posledica raziskovalnorazvojne in tehnološke sposobnosti se mi je zdel vreden posebne pozornosti. Od kje in zakaj so se Japonci zavihteli na tehnološki vrh? Kakšno je bilo in še vedno je ozadje nenehnega vzpenjanja, ki vznemirja sklerotično zadržano Evropo in kapitalno vladajoče ZDA? Na Japonskem sem imel enega samega znanca še iz časov moje dejavnosti v Generalni skupščini IFIP konec šestdesetih in v začetku sedemdesetih let: profesorja Eiichija Gotoja s tokijske univerze; žal sem v osemdesetih letih na to dobro in prijateljsko znanstvo skorajda pozabil. Prof. E. Goto je bil vendar podpredsednik IFIPa v času kongresa IFIP v Ljubljani leta 1971. Iz te moje pozabljivosti se je kasneje izcimila kar občutna organizacijska škoda.

Že v poletju 1985 in morda še prej so me začeli prijatelji vzpodbujati nekako takole: "Odpravite se vendar na Japonsko in navežite stike z japonskimi univerzami in inštituti pa tudi s tistimi japonskimi podjetji, ki bi bila primerna za sodelovanje z nami." Tako je bila oblikovana ekspedicijska skupina in v sredini septembra sprejeta dokončna odločitev, da se na Japonsko odpravimo 1. novembra 1985 in da nekako v dobrem tednu opravimo čimveč tega, kar je bilo z načrtom določeno. Ob tem se je pojavila tudi zahteva, da moram za obisk na Hokkaido univerzi v Sapporu pripraviti "protokolarno" predavanje. Tako so se dejanske organizacijske in moje strokovne priprave za razgledovanje in obiskovanje na Japonskem lahko začele.

2. Priprava na zahtevno pot

Inazuma ni satoranu hito no totosa yo

Kako je vzvišen, ki ob blisku ne pomišlja na minljivost življenja

(Matsuo Basho)

* Vsi haiku motivi so privzeti iz knjige Vladimirja Devideja "Japanska haiku poezija," Cankarjeva založba, Zagreb 1985. Prevod v slovenščino je svoboden (iz srbohrvaščine). Motivi naj spodbujajo bralca pri razumevanju japonske in naše poti v jutrišnji dan.

Potovanje na Japonsko je moralo biti dobro in vnaprej pripravljeno, saj bi vsaka improvizacija lahko bila neuspešna. Tako sem 15. 9. začel razmišljati o osnutku svojega predavanja, hkrati pa smo se dogovarjali tudi za obisk različnih podjetij in institucij. Moj cilj je bil, da predstavim Japoncem nekaj, kar je povezano z novogeneracijskim računalništvom, zato sem se lahko odločil za tematiko "Prekrivanje: vzorec za paralelno in zaporedno procesiranje" (glej avtorjev članek v angleščini v tej številki Informatice). Ker sem želel s seboj na Japonsko odnesti tudi razmnoženi rokopis tega predavanja, sem se moral lotiti dela z vso intenzivnostjo. Intuitivni model predavanja sem izdelal v 6-dnevni osamitvi tako, kot je opisan v članku (odstavki 1 do 30 poglavja 2 so bili pisani v angleščini pod vedrim nebom v bližini Ribniške koče na Pohorju v dneh od 20. do 25. 9. 1985). Kasneje se je izkazalo, da je bila ta temeljitost potrebna, ker brez nje ne bi prišel v razvojno središče japonske pete računalniške generacije, tj. v ICOT.

V Sapporu je na Hokkaido univerzi delal naš raziskovalni študent mgr. Drago Novak, na tokijski univerzi pa naš doktorand Andrej Bekeš; oba sta se intenzivno vključila s svojim znanjem japonščine v dogovore s podjetji in institucijami. Na tokijski univerzi je delal tudi drugi raziskovalni študent Dragan Milutinović z beograjske strojne fakultete, ki je prispeval sugestijo o možnosti obiska laboratorija Sata/Kimura. Zaradi dobre povezanosti tega laboratorija z ljubljansko strojno fakulteto sva si z Dragom Novakom (s priporočilom prof. Janeza Peklenika) lahko ogledala nekatere projekte, ki se tam izvajajo.

Potovanje na Japonsko je potekalo na relaciji Ljubljana - Zagreb - Pariz - London - Anchorage - Tokyo - Sapporo, vrnitev pa na relaciji Tokyo - Anchorage - Pariz - Milano - Trst - Nova Gorica - Ljubljana. Ekspedicijo so sestavljali še S. Hadži, M. Kovačević in D. Šalehar.

3. Sapporo, severna prestolnica

Sapporo je sodobno, po ameriških standardih zgrajeno mesto. Izven mestnega središča je obsežen univerzni tabor, v katerem so združene vse fakultete; tabor ima tudi vso pripadajočo infrastrukturo (športna igrišča, gostinske obrate, radioamaterski klub itd.). Cilj obiska v Sapporu je bila demonstracija nekaterih dosežkov Iskre Delte pri tamkajšnjih mikroračunalniških podjetjih, izmenjava tehnoloških in tržnih izkušenj s temi podjetji in seveda obisk s predavanjem in ogledom nekaterih laboratorijev na univerzi.

Sreda, 6. 11. 1985 je bila namenjena obisku na Hokkaido univerzi. V odsotnosti prof. Y. Aokija nas je sprejel prof. Tsuyoshi Yamamoto, ki je vodil ogled laboratorijev in organiziral moje predavanje v okviru podiplomskega študija na računalniški katerdri v okviru oddelka za elektrotehniko. Prof. Y. Aoki je sicer direktor laboratorija za procesiranje valovnih informacij.

Pri razgovorih in ogledu laboratorijev smo sodelovali vsi člani ekspedicije, z japonske strani pa razen prof. Yamamote še asistenti katedre in študentje podiplomskega študija.

Najprej smo si ogledali laboratorij za barvno grafiko, potem laboratorij za razvoj kompilatorjev (Lisp) in nato še laboratorij odsotnega prof. Yuzuru Tanake (posebni konverzacijski jeziki) in laboratorij za govorne komunikacije. Vsi laboratoriji so izrazito praktično usmerjeni in delajo v tesni povezavi z japonsko računalniško industrijo. Barvna grafika in njena izdelanost sta na visoki ravni. Poudariti velja, da postaja jezik Lisp na Japonskem vse bolj komercialni izdelek in da je njegova uporaba pri reševanju inženirskih nalog vredna vse pozornosti. Pri nas mislimo, da je Lisp predvsem akademsko orodje, vendar smo kasneje na Japonskem dobili še dodatna potrdila o uporabnosti in tržnosti Lispa. Hokkaido univerza ima očitno specializacijo na področju implementacije in uporabe Lispa in sodeluje na tem področju zlasti s podjetjem BUG v Sapporu (to podjetje smo obiskali) in s podjetjem Fujitsu, ki lispovski projekt sofinancira in podpira s svojo sistemsko računalniško opremo.

Prof. T. Yamamoto mi je podaril tudi svojo knjigo "The 3-Dimensional Computer Graphics", ki je napisana v japonščini in vsebuje zares čudovite barvne posnetke; v knjigi je objavljenih veliko algoritmov, ki so napisani v navadnih programirnih jezikih in so bralni tudi za evropskega uporabnika.

Naslov mojega predavanja je bil "Overlapping: A Paradigm of Parallel and Sequential Processing". Predavanje je bilo napovedano s posebnim plakatom v središču avle tehnične fakultete, za predavanje pa sem imel razen manuskripta pripravljenih tudi 20 barvnih prosojnic. Izdatno sem lahko uporabljal tudi kredo in tablo. Po predavanju se je razvila razprava, v kateri je sodeloval profesor Yamamoto in študentje. Profesorja Aoki in Tanaka sta bila žal odsotna (predavanja na Kitajskem in specializacija v ZDA).

V Sapporu sem se prvič srečal z japonsko delavnostjo, njihovimi družabnimi običaji (sezuvanje čevljev, hranjenje s pomočjo palčk), z njihovim nenadkriljivim smislom za praktično uporabnost razvojnega in raziskovalnega dela.

Ker iz Ljubljane ni bilo mogoče urediti obiska nekaterih za naše pojme izredno pomembnih institucij, smo se morali s pomočjo naših novih japonskih znancev o tem še naprej dogovarjati. Tu bi rad omenil predvsem možnost obiska na ICOT v Tokiu, ker nam tega obiska ni bilo mogoče poprej zagotoviti. Pri tem problemu se je pokazala velika ustreljivost naših znancev s Hokkaido univerze in iz podjetja BUG. Kljub urgencam naših japonskih znancev (profesorjev, direktorjev) pri dr. K. Furukawi nam iz Sappora ni bilo mogoče urediti obiska na ICOT. Z ICOT so prihajali odgovori, da to ni mogoče. Šele kasneje v Tokiu se je pokazalo, da je obstajal način, kako priti v to centralno državno inštitucijo pete računalniške generacije.

4. Tokijsko mravljišče

4.1. Obisk v podjetju Ampere

Obisk v Tokiu se je začel 8. 11. zjutraj. Ta dan smo obiskali v polni sestavi podjetje Ampere, ki sodeluje z BUG iz Sappora, vendar samo razvojno in nekonkurenčno. V podjetju Ampere sem ostal nekako do 12h, ker sem imel ob 14h sestanek na tokijski univerzi. Po predstavitvi Iskre in podjetja Ampere so bile izmenjane tehnične informacije, nadaljnji razgovori so bili opravljeni popoldne in v soboto 9. 11. (na teh pogovorih nisem sodeloval, ker sem se v soboto mudil v institutu RIKEN). Med drugim nam je podjetje Ampere pokazalo svoj najnovejši mikroračunalniški produkt WS-1, ki med drugim uporablja tudi jezik APL (posebej označena tastatura), namenjen pa je direktorjem, prodajalcem in tudi univerzam (zaradi APL). Ta sistem je zanimiv zaradi tega, ker omogoča tudi prenos po govornem (telefonskem) kanalu, tako da se npr. pri izmenjavi podatkov (modemski kanal) lahko komunicira tudi neposredno z govorom. Ta mikroračunalnik je izveden v CMOS tehnologiji s procesorjem 68000 (8 MHz), je razširljiv in povezljiv v mrežo (pisarniška avtomatizacija). Uporablja prikaz s tekočimi kristali z 80 stolpci in 25 vrsticami, z grafiko pa z 480 krat 200 točkami. Njegova potrošnja znaša le 1,2W (448 kB RAM, 128 kB ROM in operacijski sistem BIG.DOS).

Pri pogovorih v podjetju Ampere je bilo prisotnih od 8 do 10 njihovih vodilnih uslužbencev na čelu s predsednikom Takashijem Kusanagijem. Ryu Osaki, direktor za mednarodne operacije je prevajal iz angleščine v japonščino in tudi obratno. Po dopoldanskih pogovorih nas je predsednik podjetja Ampere odpeljal na kosilo. Zanimivo je, da se mi je prav na tem kosilu ponudila priložnost, da vendarle obiščem ICOT.

V razgovorih s predsednikom Kusanagijem je bilo ugotovljeno, da dobro pozna dr. K. Furukawo. Takoj sem mu obrazložil svoj problem, ob tem pa sem mu tudi pokazal manuskript predavanja, ki sem ga bil imel v Sapporu. Izgleda, da je bilo to odločilno, ker mi je g. Kusanagi obljubil, da bo posredoval pri dr. Furukawi in da mu bo pokazal moj manuskript. Zaradi obiska na tokijski univerzi sem moral kosilo prekiniti. Kasneje se je ponovno izkazalo, kako so Japonci dosledni v podrobnostih, v njihovi ustreljivosti in prijaznosti, še posebej, če ugotovijo ali samo slutijo, da bi določena informacija lahko koristila katerikoli japonski organizaciji.

4.2. Obisk laboratorijev Sata/Kimura na tokijski univerzi

Točno ob 14h (8. 11.) me je sprejel v svoji pisarni na tokijski univerzi prof. Toshio Sata. Sestanek in ogled laboratorijev sta bila pripravljena do podrobnosti. Tu sta sodelovala še D. Novak in Dragan Milutinović s strojne fakultete v Beogradu, ki opravlja pri prof. Sati prakso (doktorat) raziskovalnega študenta. Milutinović je s svojim predlogom (priporočilo prof. J. Peklenika) dejansko organiziral obisk in ogled.

Prof. T. Sata uživa svetovni sloves na področju strojniške avtomatizacije (machining automation, robotics) in sodi v ožji intelektualni vrh japonskega tehnološkega preboja (čudeža). Ko je pojasnjeval strategijo ICOTa, je jasno izpostavil, da gre pri tem za koncentracijo japonskega intelekta in sposobnosti in da je strategija ICOT proizvod velikega števila ekspertov z različnih področij. Očitno je bilo, da tudi sam sodeluje pri oblikovanju te strategije. Formula, ki jo uporabljajo Japonci v okviru ICOT je tale: naloga ICOT je, da zabija kline ali vodilni klin v neraziskana tehnološka in konceptualna področja prihodnjih računalniških sistemov in da s svojo organizacijo, vladno in podjetniško podporo skrbi za š i r j e n j e (razširjanje, obveščanje, organizacijo, delovne naloge, poslovne dogovore) svoje dejavnosti, skupnih rezultatov v najširšo industrijsko bazo. Na ta način rešuje ICOT sicer zelo težko rešljiv problem hitre vleke japonske industrije v nova tehnološka področja. Le nekaj sto sodelavcev ICOT vleče na ta način (po lanskih podatkih) armado 11000 industrijskih raziskovalcev

in razvijalcev. V ICOT se za določen čas rekrutirajo tudi najboljši znanstvenoraziskovalni organizatorji in eksperti iz industrije. ICOT je tudi glavni organizator razvejane mednarodne raziskovalne dejavnosti (o tem pozneje).

Kot zanimivost je prof. T. Sata obrazložil še tole: profesor tokijske univerze ima praviloma docenta in dva asistenta. Država prispeva letno cca. 2 milijona jenov za raziskave na profesorja. Nadaljna 2 milijona jenov priteče iz neke vrste raziskovalne skupnosti. Od posameznega podjetja se lahko vzame brez obvez še 2 milijona jenov. Takih podjetij je od 10 do 20. To znese letno za svobodne raziskave približno 24 do 44 milijonov jenov (120 do 220 tisoč dolarjev) v gotovini. V soglasju z fakulteto lahko profesor vzame ali si sposodi računalniško opremo, s katero razpolaga poseben industrijsi konzorcij. Fakultete so popolnoma neodvisne in se ne smejo tesneje vezati na nobeno podjetje. Lahko pa se laboratoriji različnih fakultet povezujejo med seboj in tesno sodelujejo na problematiki skupnega interesa. Tak je tudi primer povezave laboratorijev prof. T. Sate in prof. Kimure, ki opravljata skupne raziskovalne dejavnosti.

Ogled laboratorijev je bil povezan z razlago tele problematike, ki je bila praktično demonstrirana na štirih lokacijah obsežnega laboratorija:

-- sistemska arhitektura naprednega produkcijskega sistema:
   sistemski model produkcije; programska arhitektura produkcijskega sistema; programska orodja za sistemsko konfiguriranje: prototipiziranje; distribuirani sistem in podatkovna baza
-- geometrijsko modeliranje:
   - osnovni geometrijski algoritem in podatkovna struktura:
     dvodimenzionalno slikovno urejanje in barvanje; poenotena podatkovna struktura za trodimenzionalne geometrične objekte;
   - trdnostno modeliranje (GEOMAP-III):
     osnovne operacije in podatkovna struktura za trdne snovi; prostorsko indeksiranje za učinkovito procesiranje; integracija prostooblikovnih površin;
   - modeliranje prostooblikovnih površin:
     večstranične površinske (ploskovne) prilagoditve; metode oblikovanja lepotnih površin;
   - uporaba geometrijskih modelov:
     geometrični izračuni; raznovrstne analize;
-- izdelčno modeliranje:
   - analiza zahtev izdelčnega modela:
     analiza oblikovalnega in proizvodnega procesa;
   - predstavitev izdelčnega modela:
     objektno usmerjena metoda in Hornova logika; domenskoznačilno znanje: geometrija, manipulacija formul, mehanika;
-- obravnava tehničnega znanja:
   - klasifikacija tehničnega znanja;
   - splošni okvir predstavitve tehničnega znanja;
   - arhitektura podatkovne baze;
-- povezava človek-stroj:
   - tehnične V/I funkcije: risanje, simbolni slovar itd.;
   - upravni sistem uporabniške povezave;
   - primer interactivnega geometričnega modeliranja;
   - arhitektura tehnične delovne postaje za eksperte;
-- načrtovalne in proizvodne aktivnosti:
   - načrtovanje izdelka: dimenzijska in tolerančna analiza; parametrično načrtovanje; modelna transformacija in oblikovanje: iz konceptualnega v podrobno;
   - planiranje procesa: uporaba ekspertnega sistema;
   - strojništvo: planiranje operacij za grobo obdelavo;
   - inteligentni senzorski robotski sistem: sistemska arhitektura senzorskega robota; interaktivno načrtovanje robotskih operacij; izvajalni sistem robotskih operacij: HW in SW; senzorski sistem: TV kamera (sestavljeno vizualno razpoznavanje), taktilni senzor, senzor sile itd.; uporaba pri nalogah strojnega sestavljanja;
-- diagnostika proizvodnega sistema:
   - zaznavanje napak strojnega orodja s tokom motorja;
   - vodenje strojev z zvočnimi signali;
   - diagnostika strojnih napak z ekspertnim sistemom.

Vsi prostori laboratorija so bili močno populirani s študenti in zunanjimi sodelavci, delo je potekalo na številnih terminalih in delovnih mestih, delovni čas traja navadno od 9h do 22h, v nekaterih prostorih so bila opazna tudi ležišča za prenočevanje (za nočno delo ob dolgotrajnih eksperimentih, ki zahtevajo prisotnost raziskovalcev).

Na koncu obiska je prof. T. Sata še razpredal svojo filozofijo razvoja, ki naj bi temeljila na zaupanju, razumevanju in pomoči med različnimi populacijami. Zlasti je pomembno za Japonce, da razumejo ostali svet, pa tudi za druge, da razumejo Japonce. Obisk pri prof. T. Sati je izzvenel zelo harmonično in je bil dejansko na visoki intelektualni ravni. To je bilo mogoče predvsem zaradi Satove življenske modrosti in zaradi njegovega izrednega obvladanja angleščine.

A s o c i a c i j a v povezavi z obiskom laboratorija Sata/Kimura

Ob obisku laboratorija Sata/Kimura na tokijski univerzi sem se seveda vprašal, kje je ob izredno dobri povezanosti strojne fakultete v Ljubljani in laboratorija Sata/Kimura usmeritev, predvsem pa vpliv in možnosti fakultete v Ljubljani. Japonske raziskave so izredno racionalne in usmerjene na nujen minimum stroškov v strojniški avtomatizaciji. Takšen je primer elektronskega vida, ko se pri znanem predmetu na traku ugotavlja položaj predmeta in se določajo podatki za manipulator (za roko, ki predmet premešča in obrača). Nepreconljive vrednosti je tudi ekspertni sistem oziroma orodja za načrtovanje proizvodnega procesa, ki so instalirana na stroju tipa VAX. Bilo bi smiselno, da se strojna fakulteta v Ljubljani usposobi za prevzem te vrste tehnologije prek tkim. akademske povezave (academic link), saj se bo prof. Sata kmalu upokojil in te zveze morda ne bodo tako trdne kot so sedaj. Pri tem bi morali strojni fakulteti v Ljubljani čimprej zagotoviti ustrezno računalniško opremo (verjetno tudi iz domače računalniške proizvodnje).

4.3. Sporočilo o obisku na ICOT

Kojiki kana
tenchi wo kitaru
natsu-goromo

Glej berača:
Nebo mu je tudi Zemlja
v letnem oblačilu

(Takarai Kikaku)

Po vrnitvi iz Sata/Kimurovih laboratorijev me je ob 18h poklical v hotel g. Ryu Osaki iz podjetja Ampere in mi sporočil, da me bo v ponedeljek 11. 11. ob 10h sprejel dr. K. Furukawa (ICOT) na dvourni razgovor. Ta sestanek je

uredil predsednik podjetja Ampere, g. Takashi Kusanagi, kot mi je obljubil na kosilu. Tako je tretja urgenca na ICOT končno uspela.

4.4. Obisk instituta RIKEN

V soboto 9. 11. sva se s Š. Hadžijem odpravila na obisk v institut za fizikalne in kemijske raziskave RIKEN v Saitami (približno uro vožnje z vlakom iz Tokia). Ta sestanek je bil organiziran na osnovi mojega poznanstva s prof. Eiichijem Gotom, s katerim sva sodelovala v okviru IFIP v 60-ih in 70-ih letih; on je bil v času kongresa IFIP '71 v Ljubljani podpredsednik IFIPa. Obisk je bil namenjen njegovemu laboratoriju za informacijske znanosti v RIKEN.

Ob odsotnosti prof. E. Gota sta naju sprejela višji raziskovalec dr. Takashi Soma in raziskovalec dr. Masanori Idesawa. Laboratorij za informacijske znanosti se ukvarja s temi raziskovalnimi področji: ekspozicijski sistem z elektronskim curkom, računalniška algebra, nove programirne metode, procesiranje in generiranje slik, avtomatično merjenje trodimenzionalnih predmetov in logični elementi z Josephsonovim spojem. Zanimivo je, kako so naštete problematike laboratorija medseboj organsko povezane. To povezanost bom opisal na osnovi nekaterih raziskovalnih dosežkov oziroma razvojnih produktov tega laboratorija, ki so bili predani v proizvodnjo raznovrstni japonski industriji.

Prof. Eiichi Goto vodi tudi laboratorij za računalniške znanosti na tokijski univerzi; problematiki obeh njegovih laboratorijev sta prepleteni in soodvisni. V obeh Gotovih laboratorijih se raziskuje vrhunska mikroelektronska in računalniška tehnologija, ki je neposredno predmet novih računalniških generacij (prof. Goto poudarja, da je vsaka nova generacija samo /enostavno/ zadnja generacija, kar je v bistvu v nasprotju s pojmovanjem prihodnje /zlasti pete/ generacije, ki je ne priznava).

V preteklih 10 letih je laboratorij razvil litografski sistem z elektronskim curkom, s katerim je moč realizirati 0,1-mikronsko tehnologijo. Elektronsko ležje te naprave so preračunavali več let in rešitev poljskih enačb je lahko bila najdena šele s pomočjo močnega lispovskega stroja. Pokazali so polinom ležja v simbolni obliki z dolžino približno osemdesetih strani (listing) in takšen polinom je bil lahko izpeljan samo z dovolj zmogljivim strojem za jezik Lisp. Litografski sistem ima naslednje zmogljivosti:

-- proizvodnja mask: 10 ravnin na uro;
-- proizvodnja mrežic (retiklov): 8 do 12 ravnin na uro;
-- neposredno pisanje: 10 rezin na uro;
-- natančnost obsega vzorca: 0,1 mikrona;
-- natančnost prepletanja (stitching): 0,1 mikrona;
-- medplastna natančnost: 0,15 mikrona;
-- plastna (overlay) natančnost: 0,15 mikrona;
-- vzorčna dodelitev: v 0,05-mikronskih segmentih.

Ta naprava je bistveno prispevala k razvoju 256kb dinamičnih RAMov in k 512kb ROMov, uporablja pa se tudi pri razvoju tehnologije največjih gostot. Naprava se danes proizvaja serijsko, in sicer pod imenom JBX-6AII (podjetje Jeol). Ležje je bilo zrisano s posebno 3D grafiko, ki so jo razvili v tem laboratoriju. Nazadnje pa so prav to napravo uporabili tudi pri VLSI realizaciji lispovskega stroja FLATS, ki je izračunal ležje. Tako se je raziskovalni/razvojni krog sklenil na neverjetno učinkovit način.

Drugi pomemben izdelek laboratorija je lispovski stroj FLATS (Formula Lisp Association Tuple Set). Ta računalnik so razvili z ECL tehnologijo in prototip je nameščen v sedmih velikih omarah. Za komunikacije uporablja VAX-11, za podporo pri razvoju materialne opreme na tem lispovskem računalniku se uporabljata kot periferni enoti še en VAX-11, M380 in pravtako za podporo pri programskem razvoju. FLATS je najhitrejši lispovski računalnik na svetu, je pa tudi med največjimi. Kot tak je povezan prek satelita z raziskovalnimi centri po svetu (Rand Corp., Illinois University, Texas A&M University, Max Planck Gesellschaft, Pasteur Institute, Cambridge University itd.). Lisp postaja vse bolj pomembno orodje pri tehnološkem načrtovanju in pri raziskavah v fiziki, kemiji, ladjedelništvu, biologiji, medicini in še kje.

Ogled lispovskega stroja je bil mogoč pri odprtih omarah, kjer je bilo moč videti na desettisoče vodov, ki spajajo posamezne enote. Stroj je seveda deloval v mednarodni mreži. Njegov mikrocikel znaša 50 ns. Primerjava z drugimi Lisp stroji je prikazana v tabeli 1.

FLATS je rezultat razvoja Gotojevih laboratorijev v RIKEN in na tokijski univerzi. Ta stroj uporabljajo pri razvoju novih Josephsonovih spojev in tudi pri načrtovanju najhitrejšega paralelnega računalnika na svetu (taktne frekvence in operacijske hitrosti so v področju mikrovalov, tj. v intervalu med 10 in 100 GHz). FLATS je rezultat dolgoletnega, usmerjenega dela, vendar ni samo akademski dosežek. Zanimivo je, da ta stroj v visokointegrirani obliki proizvaja in trži Mitsui Engineering & Shipbuilding Co. za potrebe tehniškega, tehnološkega in razvojnega načrtovanja. Ta komercialni FLATS je seveda kompakten računalnik in delovna postaja.

V Gotojevem laboratoriju v RIKENu so nama pokazali tudi procesiranje in generiranje slik in grafiko, ki je bila uporabljena pri načrtovanju elektronskega ležja. Za ogled laboratorijske

Tabela 1

Ime stroja	CDR kodiranje	Logika	Celični pomnilnik	Cashe pomnilnik	Mikrocikel
CADR	2 bita	TTL	16 M	nima	180 ns
Dolphin	8 bitov	TTL	16 M	nima	200 ns
Dorado	8 bitov	ECL	16 M	120 ns	60 ns
3600	2 bita	TTL	64 M	200 ns	200 ns
ELIS	nima	TTL	16 M	nima	180 ns
EVLIS	nima	TTL	64 k	nima	100 ns
ALPS2	nima	TTL	500 k	nima	300 ns
Kobe	nima	TTL	64 k	nima	300 ns
FLATS	2 bita	ECL	32 M	50 ns	50 ns

proizvodnje Josephsonovih spojev je šal zmanjkalo časa, saj se je ura pomaknila že v pozno sobotno popoldne. Za konec smo se s sodelavci laboratorija tudi slikali in dobila sva spominska posnetka.

A s o c i a c i j a v povezavi z obiskom v RIKENu

Ichiban ni
kagashi wo tabsu
nowaki kana

Kot prvo je
podrla strašilo
jesenska nevihta

(Morikawa Kyoroku)

Pri ogledu laboratorija v RIKENu je ostalo odprtih več vprašanj, ki zadevajo japonsko/slovensko in slovensko/japonsko projekcijo. Če primerjamo, kaj so naredili podobni instituti pri nas pri približno enaki kadrovski in raziskovalni strukturi in kaj so emitirali v širšo reprodukcijo v določenem razdobju, je vredno iskati odgovore. Motivacija japonskih raziskovalcev je posledica visokosposobnih kadrov, ki se utemeljuje s težnjo, da je potrebno prizvajati najboljše na svetu. Imeti najboljše, najzmogljivejše, najhitrejše je osnovni imperativ, ki temelji seveda na visokem intelektualizmu raziskovalnih in vodilnih kadrov. Izgleda, kot da je pri nas nesposobnostna kadrovska selekcija izničila raziskovalno sposobnost in intelektualizem, ki sta osnovna faktorja raziskovalnega in sicerànjega napredka določene populacije. Pri tem nam je ostala kot sedativ le inteligenca (neintelektualizem, priučenost, nesposobnostna prilagodljivost), ki je zmožna le še nesposobnostnega prilagajanja in drsenja v neperspektivni razkroj. Takšna naravnanost domačih znanstvenih institucij je še posebej vidna v njihovih domala mrtvičnih in neustrezno sestavljenih managementih in seveda tudi v raziskovalcih, ki za svojo funkcijo v mednarodnem znanstvenem prostoru niso več usposobljeni. Kako je sicer mogoče, da je skupina 6 ljudi v RIKENu v desetih letih prispevala več v svetovni tehnološki prostor kot naš 600 članski institut, ki bi nazadnje lahko raziskoval tudi za potrebe razvitega sveta (pri deklarirani sposobnosti znanstvenega menažmenta in raziskovalcev)?

5. Kamakura, sveti kraj Japoncev

Hana chiru ya
garan no hitsugi
otoshi yuku

Cvetje odpada -
zapira vrata hrama
in odhaja

(Boncho)

V nedeljo 10. 11. smo obiskali sveti kraj v bližini Tokia z imenom Kamakura. Tu se nahaja množica Budhovih templjev. Izlet je bil zanimiv zaradi opazovanja in raziskovanja japonske mentalitete in običajev in njihove projekcije na naše navade. Povzpeli smo se tudi na sveto goro (kot je naša Šmarna gora) in se vrnili na pacifiško obalo. Tako smo si nabrali še nekaj potrebne kondicije za obiske v ponedeljek in torek.

Japonska verska zbranost je temeljita in kaže na sposobnost miselne koncentracije v valujoči množici in okoliškem hrupu. Značilna je tudi japonska dekoncentracija, ki se kaže v zaspanosti ob vsaki, za to primerni priložnosti. Japonec spi oziroma dremlje skorajda v vsakem sedečem položaju, če se od njega ne pričakuje delovni učinek. Tokijska podzemska železnica je slika spečih Japoncev. Mati, ki pripelje otroka v trgovino z igračami, v trenutku zakinka na stolčku. Odkrito zehajoči Japonci na tokijskih ulicah so običajen pojav. Japonska zaspanost je kot počitek pred velikimi napori, ki so za preživetje japonske populacije nujni.

Na tokijskih ulicah je mogoče razumevati občutje japonske ogroženosti. To je ogroženost zaradi prenaseljenosti, ki pogojuje občutke revnosti (nezadostnosti življenskih virov), katastrofalnosti (potresov, tajfunov, povodnji) in nerazvitosti (tudi najsodobnejša tehnologija in vrhunska delovna usposobljenost ne zagotavljata več golega preživetja). Delovna sposobnost je za Japonca imperativ, iz katerega izvira tudi izreden smisel in pripravljenost za medsebojno pomoč in za skupinske podvige. Japonska zavest je prepojena s prepričanjem, da sposobnost ni samo nujnost, marveč mora biti in ostati osnovna potreba. Zadovoljevanje te potrebe je pripomoček, ki zagotavlja preživetje za daljše razdobje.

Japonska ustrežljivost in pripravljenost za pomoč se kaže tudi v stikih s tujci. Ta pripravljenost je tem večja, na čim višjem položaju je Japonec. Ker je japonska upravljavska hierarhija izrazito sposobnostna (čim višji položaj tem večja sposobnost), je komunikacija najlažja in najustreznejša pri vrhu. Japonec v nekem upravljavskem vrhu je za svojo funkcijo primerno izobražen, razgledan, obziren, kulturen, toleranten in ustrežljiv. Naša upravljavska hierarhija je zgrajena nesposobnostno: čim višji položaj tem večja nesposobnost. Sposobnostna komunikacija na vrhu je praktično nemogoča: napihnjenost, odrezavost, poklicna popačenost, strah in domala neverjetna brezbrižnost za usodo rezultatov in dosežkov trdega dela utrjujejo in utemeljujejo visoke položaje. Ta inverzija življenske strategije, ki temelji na popačenih ciljih in brezciljih, je značilna, je nasprotje nečesa, kar je kot občutena nujnost usmerjeno v odgovornost za preživetje.

Japonski intelektualizem (zavestna naravnanost v vzdrževanje in rast sposobnosti japonske populacije kot posameznika in celote) se mi je potrjeval v številnih osebnih stikih, zlasti v razgovorih s prof. T. Sato, predsednikom družbe g. T. Kusanagijem, z raziskovalcema dr. M. Somo in dr. M. Idesawo in še posebej v razgovorih z dr. K. Furukawo (ICOT) in prof. E. Gotom. Japonska delovna motiviranost je intimna in ima bistvene notranje (individualne) korenine, ki mi jih je najbolj nazorno pojasnil prof. E. Goto (o tem pozneje). Ob tem sem se upravičeno spraševal, kje hodi in kam je zašel naš profesionalni (poklicno strokovni) intelektualizem, kot brezkompromisni moralni atribut razvitega človeka naše dobe.

6. Novogeneracijsko računalništvo: tokrat zares!

Shiroki kyosen
kitareri haru no
tokarazu

Prihaja velika
bela ladja. Kmalu
bo tudi pomlad

(Rinka)

Japonci razvijajo pravo novo računalniško generacijo. Ta izvirnost se kaže dejansko tudi v načelu, da je nedoločnost japonska krepost, ki bo prešla v določnost, ko pride čas. Japonski razvojni projekt namreč ne pristaja na kompromis spajanja starega z novim ali polnovim. Japonski način mišljenja z izdatno uporabo desne možganske poloble prihaja naposled tudi na področju visoke tehnologije do svojega izraza.

Japonci že postavljajo temelje nove računalniške taksonomije, ki je Nejaponcu zaenkrat še nedostopna, saj se ne prevaja v angleščino.

Sprašujem se, ali bodo Evropci v letu 1995 sploh še sposobni razumeti japonsko računalniško tehnologijo. Ali ne bo tedaj zaostanek Evrope in ZDA v bistvu močno drugačen, različen? Ko danes razmišljamo o Eureki, kar pozabljamo, da je Eureka komercialni program z izdelki, ki imajo nizko rizično stopnjo in ki dejansko niso novogeneracijski. Eureka je le kompromis med spodbudo in kratkoročno perspektivo, s katerim bi Evropa rada oživila svoje dokaj statične in starostno vodene velike gospodarske organizacije. Ali ne bi bilo pametneje, da pošiljamo izdatneje v uk naše raziskovalne študente na Japonsko?

Japonska peta računalniška generacija je po zamisli (in organizacijskem konceptu) nova, brezkompromisna generacija, ki nosi v sebi veliko poslovno tveganje, ima pa tudi dokaj visoko verjetnost za bistveni in dolgoročni uspeh. Gre za naporno in dolgotrajno delo. Kar poskusite pripraviti Evropca na japonski način življenja: prav gotovo se bo upri. Razlike v intelektualnih in fizičnih zmogljivostih so prevelike: Evropa je miäljensko, starostno in delovno pomehkužena, če ne dekadentna.

6.1. Obisk ICOTa

Taiboku wo nagamete itari shita-suzumi

Sedim v hladu in gledam navzgor: Veliko deblo!

(Morikawa Kyoroku)

ICOT (Institute for New Generation Computer Technology) je lociran v Tokiu (Mita Kusakai Building) in je operativno središče, iz katerega se upravlja, vodi, raziskuje in pospešuje državni in podjetniški projekt pete računalniške generacije. ICOT je de facto pojem tehnološkega napredovanja in državne organiziranosti v japonski univerzitetni, raziskovalni in podjetniški zavesti. ICOT je kot tiho ozadje, ki ždi in je prisotno v izredno občutljivem in perceptivnem mehanizmu japonskega podjetniškega in univerzitetnega življenja. Tudi v najmanjših podjetjih vedo, kakšen je smisel te institucije. V tem pomenu ni nikakršnega oklevanja ali posmehovanja o potrebnosti in namembnosti te ustanove. Tudi ICOT sam je izredno racionalna organizacija: je samo maloštevilni, toda visoko usposobljeni klin, ki trasira in organizira pot v novo računalniško generacijo. ICOT je že postal del potrebe, ki jo Japonci izražajo kot nujnost.

Zanimivo je, da je bil ICOT financiran v prvi triletni fazi (1982 - 1984) le z zneskom 42 milijonov dolarjev (v prvem letu le 2 milijona dolarjev), vendar je bil ta kapital multipliciran z industrijskim sodelovanjem približno stokrat. V ICOT so bili rekrutirani tudi najboljši industrijski raziskovalci in organizatorji iz znanih velikih japonskih podjetij računalniške industrije. Ti kadri so ob svojem raziskovalnem delu zadolženi za brezhiben in takojšen prenos raziskovalnih dosežkov v industrijo. Zanimivo je, da morajo vsi vodilni delavci ICOTa opravljati tudi samostojno raziskovalno delo (od direktorja navzdol), kar se dokazuje s samostojnimi znanstvenimi publikacijami. Torej tu ni možen naš princip, ko se vodje raziskovalnih enot (ki že desetletja praktično niso sposobni znanstvene aktivnosti) enostavno podpisujejo na dosežke svojih sodelavcev.

V ponedeljek, 11. 11. 1985 sem počakal predsednika podjetja Ampere, g. T. Kusanagija, ki je prišel pome s taksijem v Tobu Hotel in me odpeljal na drugi konec Tokia v ICOT (več kot polurna vožnja). Mita Kusakai Building je nov nebotičnik, ki med drugimi ne vzbuja posebne pozornosti. Vsi zunanji napisi in napisi v avli tega poslopja so v japončini, tako da sem zaman iskal črke ICOT. V 21. nadstropju naju je sprejel dr. K. Furukawa v svoji dokaj razkošni delovni sobi in sprejemnici, ki je očitno namenjena tudi skupinskim sestankom. Pogovor sva začela dokaj neformalno, pri tem pa je bila izmenjana kopica informacij (seveda tudi vljudnostnih). "Najbrž ne veste, da ste drugi Jugoslovan, ki je obiskal ICOT; prvi je bil profesor Suad Alagić," mi je na začetku polslovesno sporočil prijazni dr. Furukawa.

V ICOT sem odšel z dobro izdelanim programom, ki je vseboval vrsto vprašanj. Po začetnem ogrevanju je dr. Furukawa predlagal, da mu razložim svoj koncept prekrivnega stroja (overlapping abstract machine). Ta predstavitev je trajala z vmesnimi vprašanji in odgovori debelo uro. Zlasti je bilo očitno, da je zanj (ali za njegove sodelavce) že posebej zanimiv koncept metaprekrivnega stroja (metaoverlapping abstract machine), ki povečuje stopnjo paralelnosti. Paralelizem (aparaturni in programirni) je danes osnovni (nerešeni) problem pete in seveda naslednjih generacij. Različne abstraktne podobe strojev in postopkov (vključno v okviru paralelnega Prologa) so sicer zanimive, toda ne rešujejo temeljnega problema o paralelni dekompoziciji problemov. Ta dekompozicija je še vedno lahko le rezultat izredne iznajdljivosti in domiselnosti posameznika v vsakem posebnem primeru, ni pa metodologija, ki bi lahko dosegala avtomatično dekompozicijo s človekom ali s strojem. Prav zaradi tega so novi paralelni koncepti zanimivi, ker morda prinašajo novitete v semantično razreševanje dekompozicijskega problema. Dr. Furukawa me je opozoril na najnovejšo japonsko raziskavo s področja paralelnega Prologa, ki se izvaja na računalnikih v ICOT in temelji na konceptu tkim. varovalnih Hornovih klavzul (Guarded Horn Clauses). Ob tem sem izrazil dvom v kakšno bistveno podobnost med temi klavzulami in prekrivnim strojem, saj gre pri prvem za izrazito lingvistični koncept, pri drugem pa za posebno topološko strukturo, ki v bistvu združuje problematiko reševanja problema z distribucijo paralelnih procesov v procesorski mreži (processor grid). Nazadnje je dr. Furukawa priznal, da je tkim. metaprikrivni koncept presenetljiv, saj ga je mogoče širiti naprej v metametakoncept itd., kar lahko pripelje do ekstremno visoke stopnje paralelnosti.

V nadaljevanju razgovora z dr. Furukawo so bila pojasnjena tale vprašanja:

- -- Kakšne so možnosti, da se povabi eksperta iz območja dejavnosti ICOT kot predavatelja na mednarodni simpozij v SFRJ?
- -- Ali je mogoče objavljati prispevke v časopisu "New Generation Computing" (izdaja ga Ohmsha v Tokiu, distribuira pa Springer-Verlag)?
- -- Ali je mogoče individualno sodelovanje (dopisovanje, izmenjava znanstvenih informacij) z ICOT?
- -- Ali je mogoče medinstitucionalno (univerzitetno, academic link, podjetniško) sodelovanje z ICOT?

Povabilo eksperta za uvodno predavanje in vodenje seminarja iz določenega področja v SFRJ je mogoče. Ker sem izrazil, da bi želeli imeti v uvodnem referatu v bistvu pregled japonske strategije in dosežkov v napredovanju v peto generacijo, je dr. K. Furukawa izrazil pripravljenost, da pride sam v Opatijo (Mipro). Kasneje jo povedal, da so težave z rokom (v maju 1985), ker imajo v aprilu fiskalno leto in

sklepajo pogodbe za novo fiskalno obdobje. Pri tem se je potrebno obračati na njihov International Relations Dept., ki ga vodi (administrira) gospa Ami Semba, ki mi je bila tudi predstavljena.

Časopis "New Generation Computing", ki ga ureja mednarodni odbor pod okriljem ICOT, je časopis odprtega tipa in recenziranje prispevkov je podobno kot v znanstvenih časopisih, ki jih izdaja Springer-Verlag. Za SFRJ bi tak prispevek lahko bil sprejet, če ustreza recenzentskim kriterijem.

Individualno sodelovanje z ICOT je edina svobodna oblika sodelovanja, ki je mogoča. To je osebno sodelovanje med posamezniki v ICOT in izven njega. Tu so seveda določene omejitve (dežele realnega socializma).

Medinstitucionalno sodelovanje med ICOT in tujo organizacijo je mogoče samo na osnovi predhodnega medvladnega sporazuma. Doslej takih sporazumov še ni, vendar bo prvi sklenjen med vladama Japonske in Velike Britanije. Drugi medvladni sporazumi so v fazi priprav (Francija, ZRN, Švedska itd.).

Po teh razgovorih mi je dr. Furukawa pokazal še delovanje PSIM (Personal Sequential Inference Machine), ki jo izdelujejo že serijsko. PSIM je orodje za razvoj programov in arhitekture japonskega računalniškega sistema pete generacije. Primarni (strojni) jezik tega stroja je logični programirni jezik (kernel language tipa Prolog). Operacijski sistem je SIMPOS. S tem sistemom doseže PSIM 30k LIPS (logical inferences per second). Uporablja visoko interaktivne V/I naprave (bitno preslikan prikaz, miška) in LAN za inter-PSIM in druge komunikacije. Ta postaja je praktična in cenena in je dejansko masovno razvojno orodje.

Obisk v ICOT je trajal dve uri, kar je bilo zaradi zasedenosti (in prejšnjih reakcij) dr. Furukawe opravičljivo. Zaradi tega kratkega časa tudi nisem imel priložnosti, da bi si ogledal še kakšne druge dosežke ICOT. Na splošno pa velja za v bodoče, da je potrebno obisk pripraviti vnaprej in se natanko dogovoriti o namenu in ciljih obiska z Japonci. Balkanska improvizacija zbuja namreč nezaupanje in občutek, da posli niso čisti. Tu pa so Japonci natančni in zelo občutljivi.

6.2. Razgovor s prof. E. Gotom

Aki fukaki
tonari wa nani wo
suru hito zo

V pozni jeseni:
je neki ta moj prijatelj,
kako mu je?

(Matsuo Basho)

Zvečer 11. 11. me je okoli 22h poklical še prof. E. Goto, pionir in neutrudni organizator japonskega računalništva, enfant terrible, dvomljivec in upornik japonske informatike. Leta 1969 mi je po predavanju nekega avstrijskega ibmovčka v Amsterdamu na temo "Računalniki in družba" potožil, da evropske filozofije kratkomalo ne razume. Tedaj sem mu odgovoril, da se to dogaja tudi meni, zato se takih predavanj ne udeležujem. Nakladanje besed je prav gotovo evropska tradicija, ki je dosegla svoj višek v lingvističnih filozofizmih prejšnjega stoletja.

Oba sva izrazila obžalovanje, da je ostalo premalo časa za ogled laboratorija za računalniške znanosti na tokijski univerzi, kjer razvija prof. Goto s svojimi sodelavci najhitrejši računalnik na svetu. Kot je bilo že omenjeno (RIKEN), temelji ta računalnik na posebnem Josephsonovem spoju in na ostali tehnologiji, ki so jo deloma razvili v RIKEN. Teoretična meja tehnologije je 0,1-mikronska, ki je potrebna predvsem tudi zaradi visokih taktnih frekvenc novega računalnika. V tem okviru se raziskujejo intenzivno zlasti problemi prenosa signalov (antene, valovodi, optična vlakna, hlajenje) itd.

Ko sem prof. Gota povprašal po njegovem zdravju, mi je odvrnil, kako bi zdravje lahko bilo slabo, ko pa gradi najhitrejši računalnik na svetu in se noč in dan vživlja v to izredno zapleteno in novo problematiko. Zatrdil mi je, da je to lahko najvišja oblika motivacije, ki jo je kdajkoli dosegel. Naloga je neverjetno naporna in vredna izrednega truda. To mi je povedal enostavno in osupljivo. Hkrati me je povabil, da si ob letu ta stroj tudi ogledam. Pri tem je poudaril, da je potrebno stvari predhodno urediti in se natanko dogovoriti (v primeru našega obiska na Japonskem namreč nisem imel v načrtu obiska pri prof. Gotu, ker nisem bil prepričan, ali je bil moj znanec iz IFIPa prav on). Tako sva se poslovila z najboljšimi željami in z upanjem, da se kmalu srečava.

Klic ob pozni uri je za Japonce običajen. Japonci namreč delajo pozno v noč.

7. Japonska sposobnost ali naša nesposobnost?

Asagao wa
saki narabete zo
shibomi keru

Cveti slak
eden zraven drugega
- in vene

(Tachibana Hokushi)

Kaj bo v prihodnosti odločilno? Ali je preživetje človeških populacij v naslednjih sto ali tisoč letih res zagotovljeno? Ali niso cilji iz domene tkim. socialne varnosti utopični, da ne rečemo ideološki? Ali je pomirjajoče in nekritično obravnavanje kot slepilo zares boljše od možnega povečevanja naporov za hitrejše populacijsko usposabljanje?

Ta in podobna vprašanja so na področju visokih tehnologij - in te so prav preživetvene tehnologije - bistvena, saj teh tehnologij ni moč obvladovati z nižjimi človekovimi sposobnostnostmi. Vprašanja in probleme sposobnosti pri človeku kot njegove najvišje informacijske (biološke, psihološke) organiziranosti je potrebno sproti razčiščevati in dograjevati. Sposobnost človeka-posameznika na njegovem področju dela mora postati osnovna potreba v enskem pomenu, kot ga npr. imajo osnovne potrebe v motivacijskem modelu Abrahama Maslowa. Če namreč sposobnost ne bo osnovna potreba prihodnjega človeka, ne bo obstajala nujna povratna povezava pri proizvodnji virov za pokrivanje človekovih fizioloških in varnostnih potreb.

Na Japonskem sem zavest o nujnosti usposobljenega Japonca lahko občutil tudi kot resnično potrebo, zaradi katere se Japonec lahko obpoveduje nepotrebno visokemu standardu, vztraja v skromnosti in sicer nerazumljivi varčnosti, v kopičenju varnostnih dejavnikov in v nenehnem vpraševanju, kaj se mu lahko zgodi jutri. Skozi to perspektivo je že vidna sposobnostna povratna povezava, ki določeno pomanjkanje razume kot neizogibno nujnost in usmerja Japonca tega stoletja v iskanje izvirnih, vzhodnjaških tehnoloških rešitev.

Kako si Američani predstavljajo peto računalniško generacijo

Ta generacija naj bi se kmalu pojavila v ZDA. Njen prihod naj bi se začel preprosto s presaditvijo umetne inteligence na paralelne računalnike, ki se pojavljajo kot gobe po dežju pri različnih prizvajalcih. Problem pa je brškone v tem, da je potrebno za paralelne računalnike spremeniti večino dosežkov današnje umetne inteligence in jo transformirati v paralelne koncepte. Druga težava pa je tudi v tem, da so vozliščni procesorji v današnjih paralelnih sistemih klasični von-neumanski računalniki. Verjetno bi tudi ti procesorji morali biti koncipirani na kakšnem višjem strojnem jeziku, kot sta npr. Prolog in Lisp.

Intel je v februarju 1985 predstavil svoj paralelni računalnik iPSC za numerično procesiranje. Zdaj pa so se pojavile tudi zahteve, da naj bi ta računalik zmogel tudi obdelavo simboličnih programov. Tako naj bi nastala paralelna različica jezika za umetno inteligenco na računalniku iPSC z imenom Common Lisp.

Intel tudi izboljšuje zmogljivost in povečuje pomnilnik za iPSC in ga tako pripravlja za uporabo v umentni inteligenci. Trenutni iPSC ima 128 procesorjev in dosega zmogljivost 100 milijonov ukazov na sekundo. Druga generacija tega Intelovega računalnika naj bi imela 1024 hitrejših procesorjev in bi bila dobavljiva v letu 1988, računalniška zmogljivost pa naj bi narasla na 10000 milijonov ukazov na sekundo. To naj bi bil po ameriških merilih že kar računalnik pete generacije.

Seveda pa kritični opazovalci zatrjujejo, da to vendarle še ne bo računalnik prave pete generacije in da nekatera podjetja v ZDA pripravljajo nekaj, kar bo bo moč uvrstiti med prave dosežke pete generacije.

Paralelno procesiranje: ali se res približuje?

Več kot trideset proizvajalcev v ZDA ponuja danes svoje paralelne računalnike. Glavna ovira za večji prodor teh računalnikov je strah uporabnikov pred poplavo različnih paralelnih arhitektur, ko ni jasno, kaj se bo na trgu uveljavilo oziroma katera paralelna arhitektura bo sprejeta kot uporabniški in industrijski standard.

Eden temeljnih problemov je meritev zmogljivosti različnih paralelnih arhitektur s standardnim paketom programov. Takega standardnega paketa namreč še ni, izdelal pa naj bi ga nacionalni urad za standarda (NBS). Večina današnjih paralelnih računalnikov išče svojo uporabo v numeriki oziroma v tkim. znanstvenih izračunih. Zaradi tega ima vrsta teh paralelnih arhitektur v vozliščih mrež tudi aritmetične koprocesorje.

Navdušenje za take računalnike med znanstveniki in akademiki ne pojenja, saj so ti računalniki še vedno občutno cenejši od tkim. superračunalnikov, katerih cene znašajo nekaj milijonov dolarjev. Ti paralelni sistemi uporabljajo namreč cenene vendar številne, počasnejše mikroprocesorje, ki izvajajo programe oziroma dele programov paralelno (hkrati, istočasno).

Vendar se zmeda na tržišču paralelnih računalnikov nadaljuje. Trenutna poplava nenavadnih mrežnih (array, grid, net) arhitektur ni več pregledna in ne daje odgovorov, kakšni so dejanski potenciali in zmogljivosti posameznih arhitektur na različnih uporabnostnih področjih. Seveda pa bi bilo hkrati potrebno razviti na univerzah oziroma v raziskovalnih središčih tudi metode paralelnega programiranja, med katere sodi zlasti dekompozicija vhodnega problema v paralelno-serijske podprobleme oziroma programe. Glavna pobudnika na tem novem področju sta ustanovi NSF (National Science Foundation) in NBS (National Bureau of Standards).

Tabela 1 prikazuje seznam različnih paralelnih računalnikov, ki so klasificirani cenovno, paralelnostno, povezovalno, pomnilniško in procesorsko. Vozliščni procesorji so različnih zmogljivosti, njihovo število pa sega od nekaj kosov do milijon.

Za paralelne sisteme v tabeli 1 tudi ni jasno, kakšna programska oprema je na razpolago za njihovo uporabo. Razen v tabeli naštetih paralelnih sistemov pripravljajo podobne sisteme tudi veliki proizvajalci, kot so npr. IBM, Cray, Perkin Elmer, Motorola, Norsk Data, ITT in drugi. Cene teh sistemov so primerno visoke. Seveda pa je zelo vprašljivo, katera od ponujenih arhitektur bo preživela daljše obdobje. V prihodnosti naj bi bila edina smiselna oziroma dovolj univerzalna arhitektura vodilo, prek katerega je vozliščni procesor neposredno dostopen za drugi vozliščni procesor. Pri sistemu z vodilom ima namreč vsak procesor za sosede vse preostale procesorje mreže, pri hiperkocki pa je število sosedov odvisno od dimenzije (npr. hiperkocka dimenzije 10 ima 1024 vozliščnih procesorjev, procesor v tej kocki pa ima le 10 sosedov, s katerimi je neposredno povezan). V primeru vodila ima procesor 1023 sosedov.

Paralelni sistemi, kot so tukaj opisani, naj bi oblikovali posebni tržni segment, tj. segment tkim. minisuperračunalnikov. To pa je področje, ki je zanimivo tudi za manjša podjetja. To tržišče naj bi se iz kapitalne vrednosti $125 milijonov v letu 1985 povzpelo na $1100 milijonov v letu 1990. Zaradi tega je razumljivo, da bo na tem področju prisotna tudi ponudba podjetja IBM, ki v svoji 100 milijardni letni strategiji (dosegljivi v letu 1990) išče tržne segmente, ki so vredni vsaj $100 milijonov letno.

V 80-tih letih so izdelki za paralelno obdelavo zanimivi predvsem za univerze in za znastveno raziskovanje. Na komercialnem področju pa se bodo ti izdelki uveljavili v 90-tih letih. Vendar trdi podjetje Loral Instrumentation iz San Diega, da ima že danes komercialno dobavljive podatkovno pretočne računalnike z imenom Loral Dataflo LFD100. Pilotska proizvodnja teh sistemov z vodilom se je že začela.

Računalnik Dataflo LFD100 uporablja 16-bitne oznake (simbole, priveske), ki so pridruženi 16-bitnim podatkovnim besedam; z njimi se poučujejo (obveščajo) številni paralelni procesorji. Vsak vozliščni procesor vsebuje tkim. vozliščni kodni soprocesor, ki opravlja funkcijo identifikacije (kot sprejemnik) in zbira v lokalnem pomnilniku pravilne podatkovno-privesne skupine. Ko so bile v pomnilniku akumulirane vse skupine, sproži kodni soprocesor programski procesor in pridruži kasneje nove 16-bitne priveske k procesiranim podatkom.

Tabela 1

Podjetje	Oznaka	Cena	Paralelizem	Povezanost	Pomnilnik	Procesor
Accelerated Processors	Model 10	$88000	4 do 12 skupin s po 8 ALE	rekonfigurabilna	ni podatkov	ni podatkov
Alliant Computer Systems	FX18	$270000	do 20	vodilo	globalni do 64 Mzlogov	64-bitni, CMOS, polje vrat
Ametek Computer	System 14	$75000 in več	16 do 256	hiperkocka	lokalni, do 256 Mzlogov	16-bitni, 80286/80287
Bolt, Beranek & Newman	Buterfly	$40000 in več	1 do 256	preklopni sistem	deljeni in lokalni	16-bitni, MC 68000
Denelcor	HEP 1	$1mil do $3mil za izvrš. modul	1 do 16 izvršilnih modulov	dvosmerna mreža	globalni	64-bitni, ECL
ELXSI	6400	$600000	do 12	vodilo	globalni, z lokalnimi do 800 Mzlo.	64-bitni, ECL, polje vrat
Encore Computer	Multimax	$114000	do 20	vodilo	globalni, lokalni do 32 Mzlogov	32-bitni, NS 32032
Flexible Computer	Flex-32	$150000 in več	do 20/box, 2480 vseh	vodilo	globalni, lokalni, 98M zl./kabinet	32-bitni, NS 32032
Gemini Computers	Trusted Multiple Microcomp.	$47500 in več	1 do 8	vodilo	deljeni, lokalni, do 128 Mzlogov	16-bitni 80?86
Intel Scientific Computers	iPSC	$150000 do $520000	32 do 128	hiperkocka	lokalni, 288 Mzlogov	16-bitni, 80286 80287
International Parallel Machines	IP-1	$50000	1 mojster, do 8 procesorjev	krosbar preklopnik	globalni, do 40 Mzlogov	32-bitni
Loral Instrumentation	Dataflo	$65000 in več	5 do 256 dataflow procesorjev	vodilo	deljeni, lokalni do 14,5 Mzlogov	16-bitni, NS 32016
Meiko	Computing Surface	$220000 do $300000	do 128	4 najbližji sosedi	lokalni, 48k zlogov na 4 procesorje	32-bitni, Inmos, T414 Transputer
Multiflow Systems	ni podatkov	v obsegu VAXa	večregistrski	ni podatkov	globalni, več kot 1Gzlog	polje vrat
Ncube	Ncube/Ten	$100000 in več	16 do 1024	hiperkocka	lokalni, do 160 Mzlogov	32-bitni, po naročilu VLSI
Saxpy Computer	Saxpy-1M	s 2 mil	32	paralelno sistolična	deljeni, do 512 kzlogov	32-bitni, po naročilu
Sequent Computer System	Balance 8000	$60000	do 12	vodilo	globalni, do 28 Mzlogov	32-bitni, NS 32032
Sequoia Systems	Sequoia System	$200000	do 64	vodilo	globalni, do 252 Mzlogov	16-bitni, MC 68010
Thinking Machines	Connection Machine	ni podatkov	64000 do 1000000	hiperkocka	globalni, 500 Mzlogov	1-bitni, po naročilu

Danes torej še ni jasno, kateri paralelno obdelovalni standardi bodo preživeli. Verjetno bodo tržno uspešni paralelni sistemi z lokalnimi pomnilniki v vozliščih. Hiperkocka je preveč naključna (zgolj kombinatorna) arhitektura, da bi lahko preživela (razvita je bila na kalifornijskem tehnološkem institutu CIT). Nekatera podjetja (Cray) pripravljajo multiprocesorske sisteme z deljenim globalnim sistemskim pomnilnikom. Tu naj bi prišli do izraza programi s sinhronizacijskimi osnovnimi funkcijami (v bistvu je to starejši sistem z znanimi pomanjkljivostmi). Za različne aplikacije pa bodo kot vseloj najbolj primerni posebni paralelni sistemi. Paralelni sistemi bodo morali biti bolj natančno prilagojeni problemom realnega sveta. Kot doslej ne bo mogoče zgraditi paralelnega sistema, ki bi bil primeren za vakršnjo uporabo oziroma za vsakršnje tržišče.

Paralelno procesiranje: resničnost ali fantazija?

Paralelno procesiranje je kot zaklad, ki je še globoko zakopan v naši podzavesti. Ta zaklad obljublja možne rešitve tistih problemov, ki jih z današnjo računalniško tehnologijo ni mogoče razrešiti, obljublja pa tudi reševalno hitrost, ki je za nekaj velikostnih razredov večja od danes znanih reševalnih hitrosti.

Komercializacija zamisli paralelnega procesiranja se je že začela. Tako so npr. podjetja Intel, Ametek Computer in NCube uresničila tkim. koncept s CITjevo hiperkocko (glej prejšnjo novico). Podjetje Bolt, Beranek & Newman že dobavlja svoj sistem Buterfly. Podjetje Thinking Machines še vedno razvija "drobnozrnati" sistem s 64000 procesorji. Massively Parallel Processor podjetja Goodyear Aerospace, ki ima 16384 procesorjev, je v uporabi pri NASA že od leta 1983.

Naraščajoče število navadnih računalniških proizvajalcev, kot so Sequent Computer Systems, ELXSI in Perkin-Elmer, že modificirajo svoje navadne multiprocesorske sisteme za uporabo pri paralelnem procesiranju.

Raziskovalne napore na področju paralelnega procesiranja financirata tudi DARPA in NSF. Konec avgusta 1985 je celo podjetje IBM napovedalo realizacijo svojega sistema s 512 procesorji z imenom RP3. Ta sistem temelji na rezultatih projekta ultraračunalnika newyorške univerze. IBM razvija še šest drugih različic za paralelno procesiranje. Številni drugi raziskovalni projekti se izvajajo v okviru akademskih institucij na področju podatkovnopretočnih jezikov in strojev, ki naj bi se alternativno uporabljali v okviru paralelnega procesiranja.

Dve sili usmerjata pobudo v okviru paralelnega procesiranja. Prva je ugotovitev, da se zmogljivosti običajnih arhitektur npr. v okviru VLSI tehnologij že približujejo svoji končni točki in da nadaljne arhitekturne izboljšave ne dajejo več bistvenih prispevkov. Sama VLSI tehnologija pa je sprožila tudi povsem drugačen pristop. Dobavljivost zmogljivih 32-bitnih mikroprocesorjev pa je navedla načrtovalce na zamisel, da iščejo procesorje s skromnejšimi ukaznimi zalogami. Na ta način je mogoče dobiti cenene in hitre 32-bitne mikroprocesorje, ki so primerni za masovno uporabo v paralelnem procesiranju.

V najširšem semantičnem smislu je paralelno procesiranje prisotno praktično v vsaki računalniški arhitekturi, npr. pri prekrivanju V/I z aritmetičnimi in logičnimi operacijami. Druga pomembna oblika paralelizma je cevenje (pipelining) ali prekrivanje izvajanja ukaza v aritmetični in logični enoti z doseganjem in dekodiranjem naslednjega ukaza v posebni ukazni enoti. Cevenje je bilo uporabljeno v računalniku IBM 7030 Stretch že v letu 1961.

V mrežnih (array) procesorjih se lahko uporabljajo splošnejše sheme cevenja in v teh shemah se številne operacije prekrivajo v tkim. semiavtonomnih materialnih elementih. Te sheme podpirajo navadno tudi vektorske operacije, ki jih najdemo v računalnikih podjetja Cray, v superračunalnikih in v novem valu Crayet (npr. Convex in Alliant Computers). Toda cevenje se uporablja tudi v večini superminijev (npr. v DECovem VAXu in 8600, v DGjevih MV, v Perkins-Elmerjevem 3200XP) in postaja standardno tudi v novem valu 32-bitnih mikroprocesorjev. Cevenje v nekem računalniku tako še ne pomeni avtomatično, da je ta računalnik že paralelni sistem.

Trenutno poudarjanje paralelnega procesiranja je usmerjeno že posebej na uporabe multiprocesorske (MP) in multimikroprocesorske (MMP) tehnologije za namene paralelnega procesiranja. Ker uporabljajo praktično vsi paralelni procesorji MP ali MMP tehnologijo, pa ni vsak MP ali MMP avtomatično tudi paralelni procesorski sistem. Nekatere multiprocesorske sisteme je moč modificirati, da zmorejo nekatera bremena paralelnega procesiranja. Sistem FX/8 podjetja Acton (Mass) je prvi računalnik, ki združuje možnosti MMP in paralelne procesorske tehnologije. Ni enostavno določiti, ali je sistem "navadni" MP ali "pravi" paralelni sistem.

Klasifikacija po namenu. Bistvena razlika v razpoznavanju splošnega multiprocesorskega sistema (SMP) od pravega paralelnega sistema je tale: SMP sistem izboljšuje sistemski pretok s prekrivanjem izvajanja številnih, medsebojno neodvisnih opravil v različnih procesorjih. Posamezna opravila niso pospešena (s paralelizmom); lahko so celo upočasnjena zaradi znane multiprocesorske obremenitve (overhead). V paralelnem procesorskem sistemu pa se številni procesorji uporabljajo za sočasno izvajanje delov nekega logičnega opravila in tako dosegajo zmanjševanje izvajalnega časa.

To razlikovanje je pomembno in poudarja, da so paralelni sistemi primerni za znastveno in tehniško uporabo, kjer so opravila zapletena in obsežna, V/I pa je relativno skromen. Trenutno so v središču pozornosti razprave, ali so paralelne arhitekture primerne tudi za uporabo v simbolični logiki in umetni inteligenci. SMP sistemi pa so trenutno učinkovitejši v komercialnih okoljih za sprotno transakcijsko obdelavo, kjer se od sistema zahteva veliko število neodvisnih opravil, ki so na splošno povezana z visoko diskovno aktivnostjo in z nezahtevnimi potrebami podatkovne obdelave.

Arhitekturna klasifikacija. Trenutno dobavljive MP, MMP in paralelne arhitekture se sicer bistveno razlikujejo v arhitektonskih in implementacijskih podrobnostih, pripadajo pa tkim. razredu VUVP (večukazni tok, večpodatkovni tok). VUVP je le ena od štirih arhitekturnih klasifikacij, ki so bile opredeljene že v letu 1960. Navadni uniprocesor pripada razredu EUEP (en sam ukazni tok, en sam podatkovni tok). VUVP obsega nominalno vsako arhitekturo, v kateri so medsebojno povezani številni in neodvisni procesorji na določen način; geografsko porazdeljeno računalniške mreže navadno ne pripadajo temu razredu.

Razred EUVP (en sam ukazni tok, večpodatkovni tok) je dokaj redek. Primer take arhitekture je danes že upokojeni Illiac, ki je bil paralelni procesor z rojstno letnico 1972. V Illiacu je razpošiljala centralna krmilna enota identične ukaze k 64 identičnim procesorskim elementom. Sodobnejši računalnik iz razred EUVP je Massively Parallel Processor podjetja Goodyear Aerospace s 16384 procesorskimi elementi. Razred EUVP se na splošno prišteva k paralelnim procesorjem. Zaenkrat ni zanesljivih podatkov o četrtem razredu VUEP (večukazni tok, en sam podatkovni tok).

Zrnatost paralelizma. Paralelizem je le redkoma problem tipa da/ne. Različni sistemi lahko imajo različne stopnje zrnatosti paralelizma. Tristopenjska sistemska klasifikacija, ki je splošno prizнana, razlikuje

-- drobnozrnati,
-- srednjezrnati in
-- grobozrnati paralelizem.

Splošna zamisel je, da opisuje drobnozrnati paralelizem vzporednost na ukazni ravnini (npr. cevno vektorsko procesiranje) ali celo pod to ravnino, npr. na bitni ravnini (sistemi inteligentnega pomnilnika, kot je npr. Connection Machine podjetja Thinking Machines, glej tabelo 1 v prejšnji novici). Različni procesorski elementi ali cevne stopnje morajo biti usposobljeni za usklajevanje (sinhronizacijo) svojih akcij v podmikrosekundnem časovnem razdobju.

Srednjezrnati paralelizem najdemo npr. pri hkratnem izvajanju zaporednih zančnih iteracij (DO-loop iterations). Posebni paralelizacijski prevajalniki (kot sta Parafrase D. Kucka in Alliantov Fortran) lahko reformatirajo izračun v tem pomenu. Za podporo te sheme mora imeti izvajalni računalnik sinhronizacijske osnovne funkcije, tako da je mogoče postavljanje zastavic in njihovo testiranje v največ nekaj mikrosekundah. Sinhronizacija je bistvena pri upoštevanju podatkovnih in krmilnih odvisnosti, tj. v primerih, ko podopravilo ne more biti nadaljevano preden drugo opravilo ni opravilo izračuna podatkov ali vejitvenega naslova ali ko procesor izvaja iteracijo in potrebuje vrednost indeksa za naslednjo zanko. Ultraračunalnik newyorške univerze in IBMov RP3 vsebujeta pomembno funkcijo fetch-and-op za te sinhronizacijske namene.

N a v a d n o m u l t i p r o c e s i r a n j e . Grobozrnati paralelizem se ukvarja z delitvijo problema v logične podbloke, ki jih je mogoče izračunavati neodvisno enega od drugega. Ta termin se uporablja navadno za opisovanje obdelave medsebojno neodvisnih opravil v VUVP arhitekturi. Termin navadno multiprocesiranje razsteza spekter paralelnega procesiranja do ekstrema, ki navaja k intelektualni pasivnosti in vnaša zmedo v terminologijo. Ta tip sistema je primer navadnega ali splošnega multiprocesiranja in ne paralelizma. Splošen MP je kabinetni sistem (IBMovi diadni in kvadratni sistemi 300X-3090), supermini (Tandem, ELXSI, Perkin-Elmer) ali multimikroprocesorski sistem (Stratus Computer, Sequent, Arete Systems, Flexible Computer, Encore Computer, En-Masse Computer).

Omeniti velja, da je vrsta dobaviteljev, kot so ELXSI, Sequent itd., implementirala modifikacije svojih arhitektur tako, da so te primerne za prehod iz grobozrnatega na srednjezrnati paralelizem. Tudi Perkin-Elmer pripravlja podobno modifikacijo za svojo družino 3200MPS.

Na Kalifornijski univerzi v Berkeleyu je bil razvit simulacijski program za vezja, imenovan Spice je eden najpopularnejših in najuspešnejših pripomočkov za paraleliziranje obstoječih programov na ustrezno modificiranih splošnih multiprocesorskih sistemih. Spice se intenzivno uporablja v razvojnih ciklih novih računalniških sistemov; v glavnem je napisan v jeziku Fortran in porabi znaten delež procesor-

Dimenzija	Stevilo vozlišč	Stevilo kanalov	Topologija
D0	1	0	
D1	2	1	
D2	4	4	
D3	8	12	
D4	16	32	
D5	32	80	

Slika 1. Topologija hiperkocke: vozlišèni procesorji so povezani z majhnim številom sosedov (pri dimenziji Dm ima vsak procesor 'm' sosedov) in vsak procesor v hiperkocki je dostopen neposredno ali skozi enega ali več procesorjev; hiperkocka je znana tudi pod imenom kosmièna kocka; pri dimenziji Dm je število procesorjev v hiperkocki enako 2^m.

skega časa, ko simulira vezja z več sto ali tisoč procesorskimi elementi. Spice odlično podpira samega sebe pri paralelizaciji, saj je število modulov glede na število procesorskih enot majhno, pa tudi enote niso kompleksno povezane in podatkovno odvisne. ELXSI in Alliant poročata o uspešni paralelizaciji s programom Spice že po nekajdnevni uporabi, ko so bile dosežene bistvene izboljšave sistemskih zmogljivosti.

Podjetje Shiva Multisystems (Mountain View, Ca) je ubralo drugo pot. Za svojega glavnega jezdeca je proglasilo PowerSpice, tj. vezje, ki je razdeljeno na podvezja, ki so med seboj relativno šibko povezana. Ta podvezja razrešujejo paralelno (hkrati) določene probleme v multiprocesorskem sistemu; tu se npr. opravlja podmilisekundna sinhronizacija. Tako lahko Shiva vstavi svojo programsko opremo v Sequent stroje, ko so ti stroji opremljeni še s tkim. Atomic Lock Memory, ki je v bistvu materializirani hitri pripomoček za sinhronizacijo.

G l o b a l n i i n l o k a l n i p o m n i l n i k . Način uporabe pomnilnika v paralelnih sistemih je ključna značilnost za razlikovanje in vsebino paralelizma. Pri porazdeljenem ali lokalnem pomnilniku se predpostavlja, da ima vsak procesor svoj lastni lokalni pomnilnik in da se sporazumeva z drugimi procesorji z uporabo mreže. Sheme z arhitekturo hiperkocke pripadajo temu tipu. Kadar je pomnilnik globalen, torej deljeni vir, imamo podobne razmere kot pri splošnih multiprocesorskih sistemih; primer takšne arhitekture je ultraračunalnik newyoške univerze. V IBMovem RP3 in v Butterflyju (Bolt, Beranek & Newman) sta na volje obe vrsti pomnilnika: globalni in lokalni; pomnilnik je tako dodan vsakemu vozlišču nemu procesorju in tudi vsak procesor lahko dostopa v nekatere dele drugih pomnilnikov.

Glavna prednost modela z globalnim pomnilnikom je možnost le enkratne naložitve podatkov (npr. z diska); različni procesorji lahko potem pošiljajo podatke in uporabljajo pomnilnik za medsebojno sporočanje. V sistemih s porazdeljenim pomnilnikom uporabijo procesorji več časa za sprejemanje začetnih podatkov in z medsebojnim razpošiljanjem vmesnih rezultatov.

Porazdeljeni (zasebni) pomnilniški sistemi lahko enostavno oskrbujejo veliko število procesorjev; v sistemih z globalnim pomnilnikom je število procesorjev bistveno odvisno od značilnosti medprocesorskega povezovalnega vezja.

P o v e z o v a l n i s i s t e m . Praktično uporabljajo vsi splošni multiprocesorski sistemi neke vrste skupno vodilo za medsebojno povezavo procesorjev in pomnilnega sistema. Vodila so seveda zelo draga in tako je navadno število procesorjev omejeno na 20. Tudi krosbar stikala so nad tem številom procesorjev neprimerna zaradi visoke cene. Paralelni sistemi, ki imajo 100 ali 1000 procesorjev tako ne morejo uporabljati skupnega vodila ali krosbar stikala.

Na prizorišču paralelnih sistemov sta se uveljavili dve povezovalni shemi. Prva je večstopenjska mreža, ki jo najdemo v ultraračunalniku, v RP3 in v Butterflyju; ta mreža povezuje vsak procesor z vsakim pomnilnim modulom, in sicer tako, da vstavi dve ali več ravnin vmesnih preklopnih vozlišč. Podobno kot v telefonski mreži je število takih vozlišč majhno v primerjavi s številom možnih oddajnikov (procesorjev) in spejemnikov (pomnilnikov); s primerno izbiro medvozliščnih povezav je moč doseči, da obstaja vsaj ena pot med oddajnikom in sprejemnikom.

Druga shema se uporablja v hiperkocki in v njej so povezovalni členi med procesorji kar procesorji sami (glej sliko 1). Vsak procesor je povezan samo z določenim številom sosedov. Komunikacija z oddaljenimi (nesosedskimi) procesorji je mogoča le skozi eno ali več vozlišč z načinom "shrani in pošlji naprej."

Tretja povezovalna shema je drevesna in se uporablja v Columbia University NonVon, v projektu FermiLab's Advanced Computer in v podatkovnobaznem stroju Teradata. Privlačnost te povezovalne različice je omejena zaradi majhnega števila računalnih procedur, ki jih je mogoče učinkovito preslikati na ta model.

K r a t e k p r e g l e d d o s e ž k o v p r a l e l n e g a p r o c e s i r a n j a . V prejšnji novici smo v tabeli 1 prikazali dosežke na področju mrežnih (vektorskih) računalnikov. Tu dodajmo in dopolnimo prejšnje podatke.

Ultraračunalnik newyoške univerze ima klasifikacijo MMP/VUVP/globalni pomnilniški sistem z medsebojno procesorsko povezovalno mrežo tipa omega. Za razvoj tega računalnike je bila samo v letu 1985 dodeljena dotacija $2,5 mil, razvijali pa so ga 5 let v Courantovem institutu newyoške univerze. Doslej so zgradili osemvozliščni prototip, v vozliščih pa so uporabili procesorje MC68010. Do leta 1990 naj bi zgradili na tej osnovi 4096 vozliščni sistem. Projekt vodita profesorja Allan Gottlieb in Malvin Kalos.

Značilnost ultraračunalniške povezovalne mreže je možnost razvrščanja paketov s konfliktnimi zahtevami, tj. tistih, ki so prispeli na enaka izhodna vrata. Pri mrežah brez vmesnikov so posledice takih konfliktov ponovno pošiljanje paketov in znižanje učinkovitosti.

Eden najpomembnejših dosežkov ultraračunalniške povezovalne mreže je mehanizem za sinhronizacijo pomnilniškega dostopa, ki se imenuje vzemi-in-dodaj (fetch-and-add ali F&A). Ta mehanizem je posplošenje atomarnih (neprekinljivih) operacij tipa preizkusi-in-postavi (test-and-set), ki se navadno uporabljajo za manipulacijo v pomnilniku nameščenih semaforjev pri koordinaciji dostopa k deljenim podatkovnim strukturam ali v kritičnih ukaznih območjih. F&A vrne prejšnjo vrednost celoštevilske spremenljivke in prišteje k njej nek inkrement v nekem atomarnem zaporedju. F&A se lahko uporablja neposredno za samodejno razvrščanje več procesorjev, ko se ti indeksirajo v skupni izvajalni vrsti.

V avgustu 1985 je IBM formalno napovedal raziskovalni projekt paralelnega procesorja. IBM poudarja, da je RP3 samo raziskovalni pripomoček in da ni vključen v noben trenutni proizvodni načrt, da pa bodo dosežki te raziskave verjetno upoštevani v komercialno dobavljivih izdelkih podjetja.

Nova lastnost RP3 v primerjavi z ultraračunalnikom je v tem, da ima IBMov sistem v vozliščih pomnilnike z obsegom do 4 Mzloge. Ti pomnilniki so uporabljivi kot lokalni, globalni ali kot obojevrstni pomnilniki, njihova delitev pa se določa dinamično v izvajalnem času. To naj bi raziskovalcem omogočilo ocenjevanje zmogljivosti sistema pri različnih pristopih.

Povezovalna mreža v RP3 ima dva podsistema, in sicer mrežo tipa banian, ki se uporablja pri nalaganju in shranjevanje in mrežo tipa omega, ki vsebuje logiko in pomnilnik za operacijsko kombiniranje. Računalniška vozlišča v RP3 bodo uporabljala poseben riscovski procesor, ki ga je razvil IBM v FET tehnologiji. Cilj trenutne-

ga projekta je razvoj 64-vozliščnega podsistema 512-vozliščnega sistema. IBM ocenjuje, da bo 512-vozliščni RP3 dosegel hitrost 1,3 milijarde ukazov na sekundo in 800 Mflops. 19 pomembnih aplikacij se že uspešno izvaja na RP3 in za konverzijo programov je potrebnih le nekaj dni.

Druga zanimiva arhitektura je hiperkocka (tudi kozmična kocka). Ta arhitektura temelji na ščibko povezanih VUVP sistemih, kjer ima vsako vozlišče svoj zasebni pomnilnik, ki je dostopen tudi za neposredne sosede prek dvosmerne serijske povezave. Obstaja pa tudi globalni kanal, po katerem dobivajo vozlišča svoje programe od krmilnega računalnika in po katerem vračajo svoje rezultate. Ta koncept je bil prvotno razvit v Pasadeni na CIT. Arhitektura je dobila svoje ime po načinu vozliščne povezave (glej sliko 1).

Dva hiperkockasta modela sta bila doslej zgrajena na CIT, uporabljata pa procesorje 8086. Tretji model je bil zgrajen v Pasadeni v Jet Propulsion Laboratory z vozliščnimi procesorji 68020.

Intlov iPSC (okrajšava za Intel Personal Supercomputer) je vodilni komercialni izdelek z arhitekturo hiperkocke. V vsakem vozlišču iPSC se nahaja procesor 80286 s koprocesorjem 80287, uporablja pa se taktna frekvenca 8 MHz. Medvozliščne komunikacije so dvosmerne in bitnoserijske pri hitrosti nad 10 Mbit/s in se izvajajo s pomočjo Ethernet vezja 82586 v vsakem vozlišču. Intlovi preizkusi kažejo, da zmore vsako vozlišče hitrosti med 35 kflops do 50 kflops.

iPSC je dobavljiv v izvedbah z 32, 64 in 128 vozlišči, njihove cene pa so $150000, $275000 in $520000. Intel uporablja tri take sisteme, 14 pa jih je bilo dobavljenih velikim naročnikom.

Drugi sistem s hiperkocko proizvaja Ametek Computer (Arkadia, Ca). Tudi ta sistem uporablja kombinacijo procesorjev 80286/287 v vozliščih in se izdeluje v izpeljankah s 16, 32, 64, 128 in 256 vozlišči. Vsako vozlišče lahko ima do 1Mzlog pomnilnika, ima pa tudi že procesor 80186, ki je namenjen medvozliščni povezavi in protokolom. Cene se nekoliko nižje kot pri Intlu.

Omeniti velja še non-vonski projekt kolumbijske univerze, ki temelji na velikih procesirnih elementih, ki se nahajajo v vmesnih korenih binarnega drevesa, kjer so binarna drevesa majhnih procesirnih elementov. Tkim. velika vozlišča imajo zasebne pomnilnike in V/I zmogljivosti; ta vozlišča so medsebojno povezana in so kontrolirana z računalnikom VAX 11/780. Majhni procesirni elementi so sestavljeni iz 4-bitnih procesorjev in RAMa s samo 64 zlogi in delujejo kot inteligentni pomnilni stroji.

Splošna operacijska zamisel tega sistema je, da delujejo veliki procesirni elementi kot VUVP sistemi, ki usmerjajo ukazne tokove k majhnim procesirnim elementom; ti pa delujejo kot EUVP sistemi. Tako se lahko številna binarna iskanja, ki so značilna za umetno inteligenco, izvajajo paralelno.

Bostonsko podjetje Bolt Beranek & Newman dobavlja paralelni procesni sistem Butterfly, ki je namenjen uporabi v umetni inteligenci. Sistem ima 256 vozlišč, ki so povezana s tkim. metuljčno mrežo, ki je izpeljanka mreže tipa banian. V vozliščih se nahajajo procesorji 68020 s pomnilniki od 1 do 4 Mzlogov. Ti pomnilniki so lahko dostopani lokalno in globalno. Vsak procesor izvaja posebno kopijo jedra operacijskega sistema, ki se nahaja izven lokalnega dela njegovega pomnilnika. Doslej je bilo instaliranih 20 teh sistemov, 45 pa je še naročenih.

Podjetje Connection Machine prodaja drobnozrnati sistem, katerega arhitektura naj bi bila primerna za lispovski stroj; vsebuje 64000 procesorjev oziroma vozlišč oziroma natančneje inteligentnih pomnilnih vozlišč, ki so enobitni procesorji s pomnilnikom 384 bitov. Povezovalno vezje je tipa n-kocke (hiperkocka), metoda sporočanja pa je drugačna.

Projekt Cedar illinoiške univerze izvira iz prevajalnika Parafrase, ki lahko strukturira fortranski program v "usmerjene krmilne grafe," kjer je vsako vozlišče grafa opravilo, ki bo izvajano, vejitve pa nakazujejo prednostne relacije in podatkovne odvisnosti. Namen projekta Cedar je gradnja multiprocesorske konfiguracije za paralelno izvajanje vozlišč dobljenega grafa. Projekt vodi prof. David Kuck.

MITjeva raziskovalca (J. Dennis in Arvind) sta začetnika dela na področju računanja z modeli podatkovnega pretoka. Takšen model omogoča paralelno izvajanje drobnozrnatih opravil pri uporabi posebnega funkcionalnega ali aplikativnega jezika, ki je visoko strukturiran in veliko bolj omejen kot visoki imperativni jezik Fortran. Paralelizem se doseže s konverzijo programa v usmerjeni podatkovnopretočni graf, katerega vozlišča so čiste funkcije (brez stranskih učinkov) in katerega poti (loki) prikazujejo podatkovne odvisnosti. Konverzija se doseže z jezikovnim prevajalnikom in uporabniško sodelovanje ni potrebno.

Podatkovnopretočni računalnik ima več izvajalnih enot različnih tipov (npr. za seštevanje, odštevanje, množenje) in programsko krmiljeni procesor. Izhodi vseh izvajalnih enot se lahko povežejo na njihove vhode s preklopno povezovalno mrežo. Program, ki poganja krmilni procesor, je določena predstavitev podatkovnopretočnega grafa. To ni sekvenčni program: ukaz postane izvršljiv, kakorhitro so prisotne vse vhodne vrednosti na izhodih ustreznih izvajalnih enot. V tej točki se ukaz lahko priredi izvajalni enoti s povezavo izhodov na vhode te enote. Veliko ukazov se lahko izvaja paralelno v različnih izvajalnih enotah.

Trenutno je v razvoju več podatkovnopretočnih računalnikov; takšen stroj se razvija na MIT, pri NTT (japonsko podjetje) pa Sigma .. Razvitih je bilo tudi več podatkovnopretočnih programirnih jezikov.

Najbolj kritično vprašanje paralelnega procesiranja je, ali je mogoče podatkovnopretočne metode uspešno uporabiti pri reševanju potreb komercialne obdelave podatkov. Nakljućne raziskave kažejo npr., da bi bila priprava plačilnih list idealna za paralelno procesiranje: plača kakega zaposlenega se namreč računa neodvisno od drugega zaposlenega; ti izračuni bi tako lahko potekali paralelno na več procesorjih.

Vendar se tudi pri paralelnem procesiranju lahko pojavi problem zasedenosti V/I virov. Če preveč procesorjev čaka na V/I, postane veliko število procesorjev nesmiselno. Podoben je problem pri paralelnem sortiranju zbirk: sortiranje je lahko paralelno, bralni čas zbirke pa je predolg in tako je paralelizem brez pravega pomena. Celo pri znanstvenih in tehničnih problemih, kjer je obilo paralelizma, obstaja gornja meja izboljšanja zmogljivosti. Če je v programu mogoče paralelizirati njegovih 95% in ostane le 5% serijskega dela programa, bo pohitritev s paralelnim izračunom lahko kvečjemu 20-kratna neglede na število procesorjev in njihovo hitrost. Vendar je navadno mogoče paralelizirati le 20% do 30% programa.

Navadno je potrebno pisati programe v obliki,

iz katere je razvidna prisotnost več procesorjev, ker sicer dodajanje ali odvzemanje procesorjev (vozlišč) ni transparentno. Navadno je nemogoče opraviti multiprogramiranje, tj. dinamično delitev kompleksa za paralelno izvajanje oziroma na neodvisna opravila. Vendar se načrtuje, da bo z RP3, Butterflyjem in hiperkockami mogoče pokazati, da je taka dinamična delitev možna. Večkrat paralelni procesorji ne morejo delovati neodvisno, saj se obravnavajo kot periferne naprave glede na gostiteljski računalnik, ki razdeljuje programe in podatke po vozliščih mreže in zbira rezultate.

Glavna ovira v napredovanju paralelnega procesiranja je pomanjkanje učinkovitih uporabniških razvojnih orodij in pomanjkanje paralelnih popravijalnih (debugging) sistemov. Podatkovno-pretočni sistemi obljubljajo nekaj v tej smeri, vendar so ti sistemi drugače omejeni (prepis obstoječih programov v nov jezik). Mehanično je mogoče paralelizirati fortranske programe, vendar je to paralelizacija na nizki ravni.

A. P. Železnikar

NOVICE IN ZANIMIVOSTI

ZDA in Japonska ukinjata carine

ZDA in Japonska sta podpisali sporazum, s katerim se ukinjajo carine na področju računalništva. ZDA se obvezujejo, da ukinjajo carine (na 0%) za računalniške komponente z izjemo prikazovalnih elektronk. Kot protiuslugo ukinja Japonska carine za računalniške komponente, periferne naprave in centralne enote. Pred tem sporazumom so bile carine za te izdelke na obeh straneh v območju 4,5 do 6%.

Siemens se hitro razvija

V preteklem poslovnem letu 1984/85 javlja podjetje Siemens prihodek 51,7 milijard (M) DM (plus 7%). Doma so naročila narasla za 2% na 23,9 MDM, za inozemstvo pa za 12% na 27,8 MDM. Nadpovprečna rast prihodka je bila dosežena v energetski in avtomatizacijski tehniki, komunikacijski in podatkovni tehniki kot tudi v medicinski tehniki.

V izdelavi je superračunalnik z 61,4 gigaflops

Na princetonski univerzi razvijajo superračunalnik, ki naj bi bil najzmogljivejši na svetu. Ta računalnik naj bi imel 128 vozlišč in naj bi dosegel hitrost 61,4 milijard operacij v plavajoči vejici na sekundo (gigaflops) in trajno zmogljivost 51,2 gigaflops. Povezava med vozlišči temelji na tkim. arhitekturi tipa cosmic cube, ki je bila razvita na kalifornijskem tehnološkem institutu. Doslej so razvili hitro 16-krat-16-krat-32-linijsko mrežo za prenos podatkov med vozlišči. Ta mreža ne zahteva dodatne podpore (no overhead) pri medprocesorskih komunikacijah. V vozlišču bo uporabljenih do 24 32-bitnih procesorjev tipa Am29325 (emitorsko sklopljeni procesorji za operacije v plavajoči vejici) s taktno frekvenco 20 MHz. Vsako vozlišče bo imelo trajno zmogljivost 400 Mflops in vrhunsko zmogljivost 480 Mflops (to je trikrat več kot sistem Cray-1).

Quo vadis Austro-Chip?

Minilo je le 5 let, ko so v centrali podjetja Voest v Linzu slovesno najavili, da vstopajo na področje mikroelektronike. Danes pa se avstrijski politiki, gospodarstveniki in majhni davkoplačevalci ustavljajo ob ruševinah diverzifikacijske strategije največjega avstrijskega koncerna: mikroelektronska aktivnost tega koncerna izkazuje velike izgube.

Vstop podržavljenega jeklarskega koncerna v polprevodniško tehnologijo je bil že na samem začetku problematičen. Poslovni in tehnološki partner American Microsystems je bil že pred otvoritvijo skupnega obrata v Unterpremstaetten pri Grazu absorbiran v podjetje Gould. Austria Mikrosysteme životari že od samega začetka, medtem pa je tudi podjetje Gould zašlo v rdeče številke. Pri tem velja poudariti, da se tržišče vezij po naročilu, ki je bilo domena ameriškega partnerja, stalno povečuje. Vzroke neuspeha torej ni mogoče iskati na tržišču.

Podjetje Siemens je opozarjalo že na samem začetku, da naj se jeklokuharji nikar ne lotevajo stvari, ki jih ne razumejo. Vendar je zašel v težave tudi Siemensov beljaški polprevodniški obrat. V trenutku, ko so odprli "256-k-obrat," se je podrlo svetovno pomnilniško tržišče. Glavni OEM partner IBM je občutno zmanjšal svoja naročila in rezultat je bil znan: Siemens je v Beljaku uvedel skrajšani delovni čas. Direktor in soustanovitelj tega obrata se je takoj preselil v Braunschweig, seveda iz povsem osebnih razlogov.

Ob tem se patetično postavlja vprašanje, ali je Austro-Chip res pred izumrtjem. V podjetju Voest je položaj mikroelektronskega obrata povsem nejasen tudi zaradi izgub tega podjetja v drugih njegovih branžah. Podjetje Siemens lahko reši finančno položaj svojega beljaškega obrata le z "modrim očesom." Vendar so pri Siemensu že vidni znaki umika s področja pomnilniških vezij na proizvodnjo vezij tipa Telecom in ASIC.

Avtorsko abecedno kazalo časopisa Informatica, letnik 9 (1985)

Članki

i korišćenju prognostičkih modela. Informatica 9 (1985), št. 4, str. 200-203.

Jelovica B.: Interaktivni rad posredstvom telex mreže. Informatica 9 (1985), št. 4, str. 327-329.

Jenko M.: Programska oprema mikroračunalniškega sistema za nadzor in vodenje visokoregalnega skladišča. Informatica 9 (1985), št. 4, str. 242-243.

Jovanoski K.: Klasični algoritmi za iskanje minimalnih dreves. Informatica 9 (1985), št. 2, str. 52-55.

Jurkovič F., D. Donlagić, B. Tovornik: Jezikovno modeliranje procesov. Informatica 9 (1985), št. 4, str. 198-199.

Karba Neda: Standardi v računalništvu. Informatica 9 (1985), št. 4, str. 125-127.

Kastelic M., B. Grilec, P. Peterlin: Barve v računalniški grafiki. Informatica 9 (1985), št. 4, str. 273-277.

Koch G.: Mesto i uloga personalnih računara u informacijskom sistemu privredne organizacije. Informatica 9 (1985), št. 4, str. 163-166.

Kocuvan E.: Operacijski podsistem za programiranje. Informatica 9 (1985), št. 4, str. 121-124.

Kokol P., M. Ojsteršek, V. Žumer: Izbira jezika za podatkovno vodene računalnike. Informatica 9 (1985), št. 4, str. 94-98.

Kolar R., M. Toni: Izdelava paketa "Menično poslovanje" za Ljubljansko banko Gospodarsko banko. Informatica 9 (1985), št. 4, str. 180-181.

Kolbezen P., S. Mavrič, J. Šilc, B. Mihovilović: Metodologija testiranja magnetnig mehurčnih pomnilnikov. Informatica 9 (1985), št. 1, str. 67-72.

Komprej I., D. Čuk: Komunikacija operaterja z lokalno konzolo perifernega mikroračunalnika. Informatica 9 (1985), št. 3, str. 56-58.

Kononenko I.: Strukturno avtomatsko učenje. Informatica 9 (1985), št. 3, str. 44-55.

Kukrika M.: Primjer dinamičkog rasporedjivanja zadataka u višeračunarskim sistemima sa radom u stvarnom vremenu. Informatica 9 (1985), št. 2, str. 66-71.

Kukrika M.: Elementi arhitekture raspodijeljenog sistema za rad u stvarnom vremenu. Informatica 9 (1985), št. 3, str. 59-65.

Kurtanjek Ž.: Primjena kompjutera u nastavi na prehrambeno-biotehnološkom fakultetu. Informatica 9 (1985), št. 4, str. 340-342.

Lazarov Z.: Prilagodavanje računarskog sistema proširenju terminalske mreže. Informatica 9 (1985), št. 4, str. 89-91.

Lečić D.: Multiprogramiranje i merenje I/O čekanja sistema. Informatica 9 (1985), št. 3, str. 66-67.

Lozica M.: Programski sustav za proračun električnog polja uz namote transformatora. Informatica 9 (1985), št. 4, str. 256-258.

Mahnič V., S. Eržen, P. Beltram: Programski paket za vodenje evidence investicijskih programov. Informatica 9 (1985), št. 4, str. 177-179.

Manasiev L., A. Petkov, K. Boyanov: A Formal Approach to the Problems of Microcode Compaction Caused by Transitory Data Resources of the Microarchitectures. Informatica 9 (1985), št. 4, str. 109-112.

Mančić Vesna: Programski realizovano nalaženje složenosti Bulove funkcije. Informatica 9 (1985), št. 4, str. 140-143.

Marc M.: Uvajanje teleinformatskih storitev v PABX. Informatica 9 (1985), št. 4, str. 315-322.

Marić I.: Prepoznavanje glasova sekvencijskom analizom autokorelacijskih vektora ograničenog broja uzastopnih sekcija. Informatica 9 (1985), št. 4, str. 360-364.

Mihelič M., U. Miklavžič, Z. Rupnik, P. Satalić: Mikroračunalniško vodeni TL analizator. Informatica 9 (1985), št. 4, str. 211-214.

Mihovilović B., P. Kolbezen: Večprocesorski sistemi. Informatica 9 (1985), št. 2, str. 48-51.

Milenković D.: Projekt logičke strukture baze podataka o biblioteci. Informatica 9 (1985), št. 4, str. 152-156.

Miljan D., J. Šilc: Govor v komunikaciji med strojem in človekom. Informatica 9 (1985), št. 2, str. 56-62.

Mitić N., V. Stojković: Proširenje Lisp kit Lisp jezika funkcijama Explode in Implode. Informatica 9 (1985), št. 1, str. 19-25.

Mittermeir R. T.: Requirements Elicitation by Rapid Prototyping. Informatica 9 (1985), št. 4, str. 105-108.

Murn R., S. Prešeren, D. Peček, B. Kastelic: Odkrivanje napak z Bergovim in podobnimi kodi II. Informatica 9 (1985), št. 3, str. 15-21.

Nezić H.: Strukturirano programiranje u asembleru. Informatica 9 (1985), št. 1, str. 52-60.

Ogrinc Tatjana, B. Mihevc: Priprava vzgojno-izobraževalnih mikroračunalniških programov in njihova uporaba pri pouku geografije. Informatica 9 (1985), št. 4, str. 349-350.

Pagon D.: Namestitveni algoritem. Informatica 9 (1985), št. 4, str. 285-286.

Panić N., O. Mikulič, V. Kocmač: Arhitektura lokalne (mikro)računalniške mreže TI-30 za vodenje procesov. Informatica 9 (1985), št. 4, str. 330-337.

Paulin A., N. Bezič: Relaxation Mesh Dynamics in the Method of Finite Differences. Informatica 9 (1985), št. 4, str. 48-49.

Peček D., R. Murn, B. Kastelic: Implementacijske zasnove za izboljšanje zanesljivosti delovanja polprevodniških pomnilniških sistemov. Informatica 9 (1985), št. 4, str. 62-63.

Pehani B., B. Pohar, J. Beštet: Povečanje informacijskih pretokov s tranzitnim pretvornikom. Informatica 9 (1985), št. 4, str. 311-314.

Petković D.: Analysis of Code Generation for a Commutative One-Register Machine. Informatica 9 (1985), št. 2, str. 26-29.

Plavc J., M. Maher, B. Černivec, B. Delak, J. Zajc: Mikroračunalniški sistem za prenos in ob-

delavo informacij pri požarni zaščiti. Informatica 9 (1985), št. 4, str. 233-235.

Prešern S.: Mikroračunalniški sistem za kontrolo procesa vulkanizacije. Informatica 9 (1985), št. 4, str. 215-218.

Redmond J. A.: Esprit Project 510 "Tooluse". Informatica 9 (1985), št. 4, str. 26-35.

Redmond J. A., J. F. J. Gleeson: Expert Systems - In Computer and Electronic Systems. Informatica 9 (1985), št. 4, str. 36-45.

Ribarić S.: Model procesora za obradu i raspoznavanje slika. Informatica 9 (1985), št. 4, str. 64-68.

Robič B., J. Šilc, B. Mihovilović: Funkcionalno programirani sistemi. Informatica 9 (1985), št. 4, str. 366-370.

Rozman I., M. Colnarič, B. Stiglic: Efficiency of Multiple Bus Structure. Informatica 9 (1985), št. 4, str. 54-57.

Rupnik V.: O sintezi generaliziranega vrednotenja informacijskih sistemov. Informatica 9 (1985), št. 4, str. 160-162.

Skočir E.: Informacijski sistem za delo republiških organov in organizacij, izkušnje pri uvajanju ter možnosti za nadaljnji razvoj. Informatica 9 (1985), št. 4, str. 173-176.

Skumavc M., J. Plavc: Multimikroračunalniški sistem za vodenje in nadzor HE Mavčice. Informatica 9 (1985), št. 4, str. 236-230.

Skumavc M.: Multiračunalniški sistem za nadzor in vodenje visokoregalnih skladišč. Informatica 9 (1985), št. 4, str. 239-241.

Slatinek R., B. Horvat, N. Črnko: Komunikacijski krmilnik. Informatica 9 (1985), št. 4, str. 296-300.

Smailagić A.: Hardversko poboljšanje kod mikroprocesora radi realizacije interlivinga adresa pri particitivnom alociranju memorije u multiprocesorskom sistemu. Informatica 9 (1985), št. 4, str. 58-61.

Sovdat B.: 32-bitni mikroprocesor NS32032. Informatica 9 (1985), št. 1, str. 39-47.

Stajić D.: Primena mikroračunarskog simulatora transportnog kašnjenja u Smitovoj metodi automatske regulacije. Informatica 9 (1985), št. 4, str. 136-139.

Stamenković S., S. Maksimović: Neke primene Apple II mikroračunara u nastavi fizike. Informatica 9 (1985), št. 4, str. 347-348.

Starič V.: Komunikacijsko usmerjen informacijski sistem za upravljanje s proizvodnjo - IBM Copics. Informatica 9 (1985), št. 1, str. 61-66.

Steblovnik K., V. Lončarič, A. Križnik: Napredna logična kartica terminala Paka 300. Informatica 9 (1985), št. 1, str. 19-30.

Šilc J., B. Robič: Osnovna načela DF sistemov. Informatica 9 (1985), št. 2, str. 10-15.

Šilc J., B. Robič, B. Mihovilović: Podatkovno vodene računalniške arhitekture. Informatica 9 (1985), št. 4, str. 371-376.

Šoštarič D.: Nekatere izkušnje pri delu z računalniškimi mrežami. Informatica 9 (1985), št. 4, str. 301-303.

Štrkić G., D. Novosel: O jezicima za konkurentno programiranje kao sredstvu za projektovanje upravljačkih sistema. Informatica 9 (1985), št. 2, str. 16-18.

Štrkić G., D. Novosel: Analiza raspoređivanja zadataka u multiprogramskom sistemu (MPS-u) korištenjem simulacije u Pascalu. Informatica 9 (1985), št. 2, str. 42-47.

Šubic M.: Procesorji v Iskrinih telefonskih SPC centralah. Informatica 9 (1985), št. 4, str. 204-207.

Tomić M.: Iterativna metoda za brzu realizaciju matematičkih funkcija na elektroničkom računalu. Informatica 9 (1985), št. 4, str. 134-135.

Toni M., R. Kolar, M. Marušič, U. Marušič, M. Gril: Paket za projektiranje panoramskih plošč. Informatica 9 (1985), št. 4, str. 229-232.

Tvrdy I., F. Rozman, R. Sabo, M. Tvrdy: Uvajanje novih funkcij v teleinformatske sisteme. Informatica 9 (1985), št. 4, str. 323-326.

Vintar M.: Programski paket APP-1. Informatica 9 (1985), št. 4, str. 182-183.

Vraneš P., M. Bajčetić: Softver grafičkog terminala prema GKS standardu. Informatica 9 (1985), št. 4, str. 260-262.

Vraneš P., I. Vojvodić: Projekat terminalskog grafičkog modula srednje rezolucije. Informatica 9 (1985), št. 4, str. 263-267.

Vukajlović Z.: Neintegrisano okruženje programskog jezika Pascal. Informatica 9 (1985), št. 2, str. 39-41.

Vukajlović Z.: Integrisano okruženje programskog jezika - alat za udobno i efikasno programiranje. Informatica 9 (1985), št. 3, str. 22-24.

Vukajlović Z., A. Zele, M. Ovčina: Formater teksta izvornog programa za Pascal. Informatica 9 (1985), št. 4, str. 99-100.

Welzer Tatjana: Podatkovne zbirke. Informatica 9 (1985), št. 4, str. 157-159.

Welzer T., B. Horvat: Načrtovanje in proizvodnja izdelkov. Informatica 9 (1985), št. 4, str. 291-293.

Yankov B., L. Nikolov, S. Bonev: MBPL Microprocessor Development Cross-System: The Emulator Subsystem. Informatica 9 (1985), št. 4, str. 113-116.

Zorman S.: Uporaba digitalnega inkrementalnega dajalnika kota v mikroračunalniškem krmilju industrijskega robota. Informatica 9 (1985), št. 2, str. 63-65.

Zvizdić D.: Koncepcija CAD sistema za proračun zagrijanja transformatora u protueksplozivnoj izvedbi. Informatica 9 (1985), št. 4, str. 287-290.

Železnikar A. P.: Ibmovski osebni računalnik Petra II. Informatica 9 (1985), št. 1, str. 9-18.

Železnikar A. P.: Ibmovski osebni računalnik Petra III. Informatica 9 (1985), št. 2, str. 30-38.

Uporabni programi

Batagelj V.: Prilagodljivo numerično integriranje. UP 18. Informatica 9 (1985), št. 1, str. 73-75.

Ficzko Jelena, A. P. Železnikar: Besedilni oblikovalnik v jeziku Pascal. Informatica 9 (1985), št. 3, str. 71-86.

Železnikar A. P.: Magični kvadrati sode in lihe stopnje. UP 19. Informatica 9 (1985), št. 1, str. 75-78.

Železnikar A. P.: Warshallov algoritem. UP 20. Informatica 9 (1985), št. 2, str. 86-87.

Železnikar A. P.: Položaj točke glede na mnogokotnik. UP 21. Informatica 9 (1985), št. 2, str. 88-90.

Nove računalniške generacije

Železnikar A. P.: Uvod k novi rubriki časopisa Informatica. št. 2, str. 72-73.

Železnikar A. P.: Doslej objavljene novice in prispevki s področja novih računalniških generacij v časopisu Informatica. št. 2, str. 73.

Železnikar A. P.: Izvedeniški sistemi, ki temeljijo na znanju I. št. 2, str. 73-83.

Železnikar A. P.: Bibliografija s področja novih računalniških generacij I. št. 2, str. 83-85.

Železnikar A. P.: Mednarodna konferenca o računalniških sistemih pete generacije v Tokiu. Informatica 9 (1985), št. 3, str. 68-70.

Železnikar A. P.: MProlog, jezik za umetno pamet. Informatica 9 (1985), št. 3, str. 70.

Polemika

Železnikar A. P.: Umetna inteligenca: polemični zapis. Informatica 9 (1985), št. 3, str. 87-89.

Novice in zanimivosti

Blatnik B., A. P. Železnikar: Stopnjevanje študijskih programov računalniških znanosti. št. 2, str. 91-92.

Blatnik B.: Podiplomski študij računalništva na ameriški univerzi. št. 2, str. 93-96.

Železnikar A. P.: Jezik Lisp za mikroračunalnike. št. 1, str. 79.

Železnikar A. P.: Kaj pomeni ibmovski osebni računalnik? št. 1, str. 79-82.

Železnikar A. P.: Kako si zgradite ibmovski PC? št. 1, str. 82-84.

Železnikar A. P.: Ibmov PC AT. št. 1, str. 84-85.

Železnikar A. P.: Šaljivo in tragično. št. 3, str. 90.

Železnikar A. P.: Nove knjige. št. 3, str. 90.

Vinčestrski diski z vdelanimi krmilniki

Dva ameriška proizvajalca sta začela v svoje visokozmogljive 5-1/4-colske vinčestrske diskovne enote vgrajevati tudi krmilnike in sta tako dosegla tkim. SCSI implementacijo. Ta proizvajalca sta Maxtor Corp. in Priam Corp. iz San Joseja v Kaliforniji. Maxtorjeva družina z oznako XT-3000 ima pomnilni obseg 170 ali 280 Mzlog s poprečnim časom dostopa 30 ms. Cena teh enot bo približno 10$ na Mzlog. Priamov model 725 ima obseg 255 Mzlog in čas dostopa 20 ms. Cena teh enot bo $2185 pri 1000 kosih.

Svetovni hitrostni rekord

V Bellovih laboratorijih gradijo tranzistor s časom preklopa 5,8 ps. Ta hitrost je za 32% nižja od hitrosti 2,7 ps, ki si jo lani dosegli pri podjetju Honeywell. Ti tranzistorji so izdelani v GaAs in AlGaAs tehnologiji. Ta hitrost je bila dosežena pri nizki temparaturi 77 stopinj Kelvina. Pri sobni temperaturi znaša preklopna hitrost tranzistorja že 10,2 ps, izgubna moč pa 1,03 mW.

PRVO OBVESTILO O SEMINARJU "MIKROFILM V PRAKSI", KI BO V LJUBLJANI NA GOSPODARSKEM RAZSTAVIŠČU 19. IN 2o/4-1983

1. dan:

Skupne teme:

- Mikrofilm v informacijskih sistemih
- Vrste mikrofilmov in njihova uporaba
- Razvoj mikrofilma v svetu in pri nas
- Mirkofilm kot sredstvo organizacije in racionalizacije poslovanja z dokumentacijo
- Arhivska trajnost dokumentacije na papirju
- Pravni vidik mikrofilma in možnosti ponarejanja in poneverjanja mikrofilma

2. dan:

Skupna tema: Povezava računalnika in mikrofilma

Seminar A: Uporaba mikrofilma v splošni poslovni in drugi dokumentaciji

- Uporaba mikrofilma v komerciali, splošnih službah, kadrovski evidenci, računovodstvu ipd.
- Uporaba mikrofilma v javni upravi, arhivih in delovnih organizacijah ter ustanovah, ki imajo trajno dokumentacijo
- Uporaba mikrofilma v bančništvu, SDK in podobnih dejavnostih
- Ogled mikrofilmskega centra

Seminar B: Uporaba mikrofilma v tehnični dokumentaciji

- Uporaba mikrofilma v tehnični dokumentaciji
- Priprava dokumentacije za mikrofilmsko obdelavo
- Mikrofilmska obdelava tehnične dokumentacije, razmnoževanja in služba sprememb
- Praktični prikaz uvedbe 35 mm filma v poslovanje z ogledom mikrofilmskega centra

Seminar C: Računalniški izhod na mikrofilm

- Uporaba COM sistema za izhodno enoto računalnika
- Priprava in organizacija dokumentacije za prehod na računalniško-mikrofilmsko obdelavo
- Racionalizacija poslovanja s COM-om in razvoj COM sistemov
- Praktični prikaz uporabe računalniško - mikrofilmske povezave z ogledom mikrofilmskega centra

Za vse informacije o seminarju se obrnite na tov. ČUFER Stanka, RSNZ SRS Kidričeva 2, Ljubljana, tel. 325-361.