An experimental computer architecture supporting expert systems and logic programming

by H. Diel
N. Lenz
H. M. Welsch

This paper describes a set of function primitives which have been designed for support of expert systems and logic programs. The functions could be offered as part of the computer architecture by implementing them in microcode and partially in hardware. The functions are primarily (but not exclusively) oriented towards support of logic programming languages such as Prolog for implementing expert systems. Particular emphasis is given to supporting the parallel execution of expert system applications by multiple processors. The concepts described are based on the Concurrent Data Access Architecture (CDAA). It is shown that OR-parallelism, as well as AND-parallelism, can be supported.

1. Introduction

In recent years, several proposals for computer architectures intended to support artificial intelligence applications have been published and in some cases implemented [1, 2]. This paper describes a computer architecture having a similar aim but which has two specific additional goals:

1. The proposed computer architecture is an evolutionary extension of the standard von Neumann computer.
2. The proposed computer architecture is intended to be the basis for further extensions and modifications (i.e., it is *experimental*). The authors believe that too much rigidity in such an architecture is undesirable at this time. This idea is similar to that of the CM* Testbed [3].

The computer architecture described here is called AIK-0. It is oriented primarily towards support of logic programming languages such as Prolog [4] and is intended to permit efficient implementations of Prolog. Furthermore, the authors assume that languages other than Prolog (e.g., LISP, OPS5) will continue to be used (not only for performance reasons) for implementing expert systems. Therefore, AIK-0 aims at providing support for such languages as well. However, the support offered by AIK-0 is described here only in general terms.

Particular emphasis is put on support of parallel processing. The performance improvements which are necessary to extend the range of feasible expert systems applications can only be achieved by utilizing a high degree of parallel processing. The authors believe that significant,

but not necessarily revolutionary, improvements over traditional computer architectures are required.

The two major items that support our aim to keep AIK-0 flexible for future adaptations are the following:

1. The fact that it supports relatively low-level functions allows considerable flexibility for the implementation of the higher-level functions.
2. The fact that it is not exclusively oriented towards support of logic programming results in the provision of functions which are more generally useful.

2. Overview of AIK-0

Figure 1 shows the overall structure of AIK-0.

In the general case a cluster of processors works together to execute an expert system application. The processors are standard von Neumann computers interconnected by some fast interconnection media.

Parallel processing within the cluster of processors is supported at two levels:

1. At a basic level, which is suitable not only for expert system functions. This level of support is provided by the CDAA (Concurrent Data Access Architecture) [5], which has been designed with a view towards artificial intelligence.
2. The functions of AIK-0, in turn, are based on CDAA. One of the major goals of AIK-0 was to design the functions with optional parallel processing capabilities but with no significant performance degradation for uniprocessing.

Although the CDAA functions and certain other features of AIK-0 have been introduced primarily for support of parallel processing, the respective interfaces (e.g., machine instructions) are, of course, also available in a uniprocessor configuration. In the uniprocessor configuration the functions can be used to control the concurrent access to memory data and other resources by multiple processes running on the same processor.

In summary, the following layers can be distinguished in a system based on AIK-0 (as shown in Fig. 1):

- A standard machine architecture.
- CDAA as a machine architecture extension supporting parallel processing.
- AIK-0 as a further machine architecture extension providing a set of machine instructions (including the CDAA instructions) which are suitable for the implementation of expert systems and logic programming.
- Typically running on top of AIK-0, a logic program application or an expert system (possibly based on logic programming, as shown in Fig. 1).

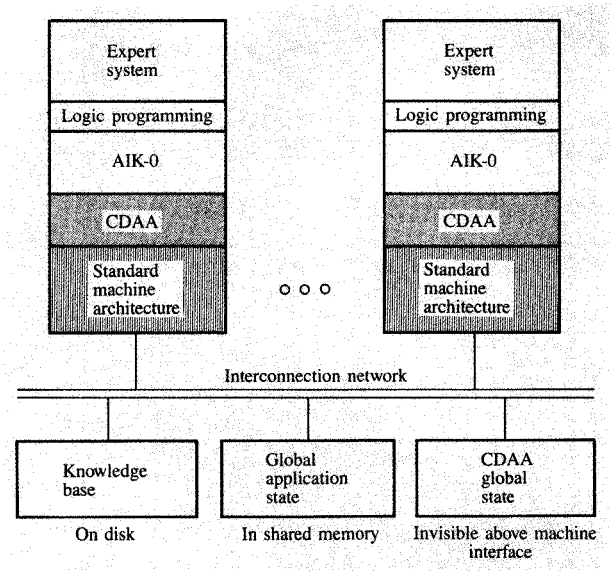

Figure 1

Overview of AIK-0 structure in a parallel processing configuration.

Shared among the multiple processors are the database/knowledge base, global state information of the expert system application, and global state information required by CDAA (invisible above the AIK-0 interface).

3. Overview of CDAA

The overall concept of the Concurrent Data Access Architecture, an overview of which is shown in **Figure 2**, is composed of four basic ideas:

- The totality of memory data which can be accessed by a processor is divided into Processor Private Data, Shared Read-only Data, and Shared Read/Write Data.
- The Shared Read/Write Data can only be accessed in a controlled way. This means that prior to accessing the data, a machine instruction (LOCK-Datafield) must be issued and the type of access (READ or WRITE) must be specified. Upon completion of access, the data must be explicitly released by an UNLOCK-Datafield machine instruction. The arrival of data values for a particular data field can be determined by explicitly clearing the data field by a CLEAR-Datafield instruction. Subsequent attempts to read such a "cleared" data field result in setting the accessing process into wait state until the data field is updated. This provides the capability of data-driven execution comparable to the dataflow concept.
- Each processor has an associated cache, which is an explicitly controlled resource. Besides the traditional purpose of a cache (to allow faster access by the processor

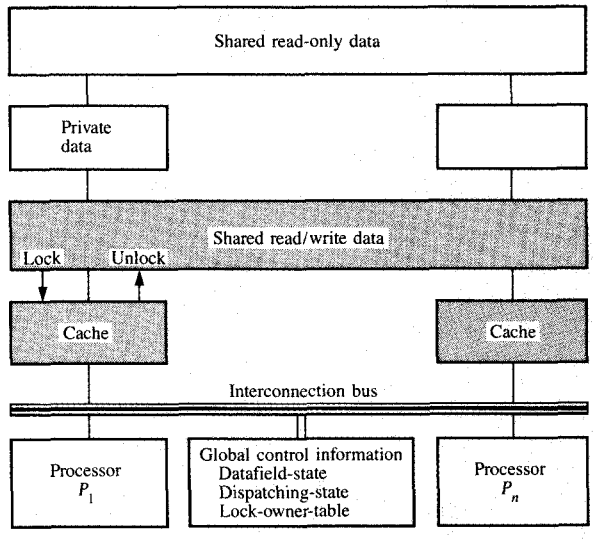

Figure 2

Overview of Concurrent Data Access Architecture.

to data), in CDAA the cache holds a local copy of the Shared Read/Write data. This allows an efficient realization of the explicitly controlled data access mentioned above by assuming that such data are accessible only when they reside in the cache. The data are moved into the cache only as the result of the LOCK-Datafield instruction. In addition, the concept of an explicitly controlled cache supports versions of data and thereby the facilities to back out (transaction processing) or backtrack (logic programming) to previous states.

- Dispatching of processes is controlled by CDAA. This means that a process is set to the wait state when it tries to obtain an inaccessible lock or to read uninitialized data. The process is resumed when a data field it has been waiting for becomes available (unlocked or instantiated).

- *CDAA instructions*

Two groups of CDAA instructions are distinguished:

- Data field management instructions — (CLEAR-DF, LOCK-DF, UNLOCK-DF, READ-UNSAFE, COMPARE-AND-SWAP).
- Cache management instructions — (CLEAR-CACHE, COMMIT-CACHE, CHECKPOINT-CACHE, RESTORE-CACHE).

An overview of these instructions is given below.

CLEAR-DF (address,length)

The CLEAR-DF instruction prepares for the detection of read-type references to the specified data field before this data field is initialized. The instruction changes the state of the data field such that CDAA can audit read attempts to the data field. Such read references are either rejected or result in setting the process issuing the read request to the wait state. Update of the data field (by use of LOCK-DF with WRITE access or by COMPARE-AND-SWAP) resets the data field state such that read references are no longer supervised.

If a data field to be cleared is locked by another process, the process which issued the CLEAR-DF instruction is set to the wait state.

LOCK-DF (address,length,type)

The LOCK-DF instruction must be issued before a data field in the Shared Read/Write Data can be accessed by a processor. The "type" field may be READ or WRITE. WRITE locks are exclusive locks; READ locks can be shared by multiple processors.

The instruction causes the specified data fields to be copied from the Shared Read/Write Data to the cache. If a data field to be locked has already been locked by another process (except when it is a READ-lock request for a READ-locked data field), the current process is set to the wait state.

UNLOCK-DF (address,length,type)

The UNLOCK-DF instruction releases locks held for data fields and in case of WRITE locks *commits* the data field values. By use of a type parameter (whose value may be either READ or 0), it is possible to change a lock from WRITE to READ.

Committing data means that the data fields are copied from the cache back to the Shared Read/Write memory.

A further result of the UNLOCK-DF instruction is that processes waiting for the unlocked data fields are made dispatchable.

READ-UNSAFE (source-address,target-address,length)

The contents of the source data field are copied to the target data field. The source data field may be located in the Shared Read/Write Memory; the target data field must be in Private Memory. The source data field is not locked (i.e., for Read access). Therefore, the value copied may change before it is further processed (it is unsafe). The instruction is primarily useful in connection with the Compare-and-Swap instruction described next.

COMPARE-AND-SWAP (source-addr,comp-addr,target-addr,length)

The target data field gets the value of the source data field, provided its present value is equal to that of the comparison data field. The combined function of comparison and value update is executed as an atomic operation.

If the specified data field is locked, the instruction is rejected.

CLEAR-CACHE
The contents of the cache are cleared and all locks held by the process are released.

COMMIT-CACHE
All data field values of the cache are written (committed) to the Shared Read/Write Data.

CHECKPOINT-CACHE (address,length)
The contents of the cache are saved at the specified area. By the use of the RESTORE-CACHE instruction, the saved copy can be reinstalled at a later time. This facility is useful to establish versions of data and to back out or backtrack to previous versions.

RESTORE-CACHE (address,option)
The cache is loaded with the data fields contained in the specified area. The specified area must have been previously loaded by a CHECKPOINT-CACHE instruction.

Two cases are distinguished by the option parameter. Option = RESET indicates the case where a previously checkpointed status is reestablished. Option = COPY is provided primarily for support of OR-node parallelism and means that a previously checkpointed status is used; however, it is mapped to a new area and therefore does not interfere with the original data fields.

For a more detailed description of the Concurrent Data Access Architecture, see [5].

CDAA is assumed to be used not only for the implementation of the AIK-0 instructions but also directly for controlling parallelism by an expert system application. Therefore, because it might be utilized in other application areas (e.g., transaction processing), the CDAA instructions are part of the AIK-0 machine interface.

4. AIK-0 instructions

This section describes the machine instructions which represent AIK-0, except for the CDAA instructions, which have already been described. The AIK-0 machine instructions provide support in areas such as parallel processing, process management, storage management, logic programming, and pattern matching. **Table 1** summarizes the AIK-0 machine instructions.

The reasons for including particular instructions in AIK-0 differ. In order of priority, the following four reasons are distinguished:

1. To provide basic functions which are not available with typical von Neumann computers. The CDAA functions belong to this category.
2. To provide functions whose performance is of critical importance to the overall performance of expert systems and logic programming applications. The GET, PUT, and PMATCH instructions are in this category.

Table 1 Summary of AIK-0 instructions.

ATTACH	Start a new process(or)
CHECKPOINT-CACHE	Checkpoint cache contents
CLEAR-CACHE	Clear cache contents
CLEAR-DF	Clear data field
COMMIT-CACHE	Commit complete cache
COMPARE-AND-SWAP	Do an atomic compare and update
DEREF	Dereference a variable
FREM	Free allocated storage
GET	Get parameter bindings
GETM	Allocate storage
LOCK-DF	Lock data field
MAKE-DISPATCHABLE	Make process(es) dispatchable
PMATCH	Do pattern matching
PUT	Put parameter bindings
READ-UNSAFE	Read shared memory w/o lock
RESET	Terminate process(or)
RESTORE-CACHE	Restore cache contents
UNBIND	Undo variable bindings
UNLOCK-DF	Unlock data field
WAIT	Wait for a resource

3. To provide functions which are heavily impacted by parallel processing and where there is a chance to handle this impact more efficiently in microcode. With GETM, FREM, and part of the process management instructions, this is the case.
4. To provide functions in AIK-0 to establish a clean layering structure. By inclusion of UNBIND, DEREF, and some of the process management instructions, it is possible to hide certain control information from the AIK-0 user.

The individual AIK-0 instructions are now described.

ATTACH (att_list,pass_list_ptr,process_id,)
A new process is started and a process_id uniquely identifying the process is returned.

All parameters defining the type of process being started are passed in att_list; e.g., the entry ı oint address, where control is to be passed; pass_list_ptr points to a parameter list to be passed to the newly created process.

If a free processor is available (i.e., free_processor_count > 0), the process is started on a new processor and the free_processor_count is updated. If there is no free processor available, the process is created; however, it is held dormant until a processor becomes free.

RESET (process_id)
The process denoted by the process_id is terminated. If process_id = 0, the process issuing the RESET request itself is reset. Termination of the process means that its execution is stopped and all resources allocated by the process are freed.

If there is no other process waiting to be dispatched on the present processor, the processor is stopped and the free_processor_count is updated.

WAIT (waitcondition,datafield)
The current process is set into wait state until the specified wait condition is signaled for the specified data field.

In general, the wait conditions are numbers whose meaning is not known to AIK-0 (e.g., wait-condition-1 = wait for terminal input, wait-condition-2 = wait for messages from process-x). The numbers get their desired meaning from the fact that a complementary MAKE-DISPATCHABLE instruction with the same wait condition (number) has to be issued. In addition, some wait condition numbers are preassigned by AIK-0, such as "wait for data field to become unlocked."

MAKE-DISPATCHABLE (process-id, waitcondition,datafield)
The specified process (or all processes if process-id = 0) is made dispatchable if it is waiting for the specified wait condition and data field.

GETM (subpool_no,area_addr,area_length)
A storage area of the specified area_length is allocated for the issuing process within the subpool specified by subpool_no.

A subpool identifies a collection of storage with common characteristics, such as lifetime, protection, access rights. These subpools are allocated in either of the following parts of the virtual memory:

- Private Virtual Storage (PVS) $0 \leq$ subpool_no ≤ 127
- Shared Virtual Storage (SVS) $128 \leq$ subpool_no ≤ 255

If no more storage of the requested type (PVS or SVS) is available, the condition code is set to 1 and area_addr is zeroed on return.

If area_length = 0 is specified, the maximum available space is returned in the area_length parameter. In this case, area_addr = 0 is returned and the condition code is set to 2. No space is allocated in this case!

FREM (subpool_no,area_addr,area_length)
The allocated storage within the given subpool, starting at area_addr and having the indicated length, is freed. The area to be freed must fit completely into an area allocated with previous GETM instructions.

With shared subpools it is the responsibility of the user to ensure that the allocated storage is not freed while there are still users accessing that area.

If area_length = 0 and area_addr = 0, then the complete subpool is freed.

GET (context_ptr,parm_block,arg_block,gst_ptr,lst_ptr, tot_ptr)
The GET instruction unifies the parameters of a clause head with the arguments of a call. Parameters are assumed to be given in a parm_block, arguments are assumed to be given in an arg_block. The context describes the logic program state. If no parallelism occurs, the context consists of the global stack, local stack, and trail. In the case of parallelism, these stacks are tree-shaped. Nodes in these trees occur whenever parallelism is started. The corresponding pieces of global stack, local stack, and trail together with some header information form a context. The contexts are backward chained.

Gst_ptr, lst_ptr, and tot_ptr point to the global stack, the local stack, and the top of the trail, respectively. If lst-ptr is zero, all bindings are made in the global stack. If tot_ptr is zero, bindings are not logged in the trail. This makes sense if no backtracking is supported or if backtracking is implemented without using a trail area.

A parm_block consists of information which can be built by a Prolog compiler. On the basis of this information, the GET instruction can execute all loading and unifying for each of the parameters in the parm_list:

```
parm_block   := nop,          /*number of parameters*/
                parm_list;

parm         := parm_tag,     /*identifies type of item*/
                mode,         /*read-only or read/write*/
                parm_body (parm_tag);
```

Read-only variables are required for support of languages such as Concurrent Prolog and Parlog.

It is assumed that all constants referenced within a program are numbered. Their corresponding reference number const_no is used to identify them uniquely; const_no may be a pointer. The constant nil is handled the same way.

Functors of structures are referenced by the same scheme, and a list may be viewed as a structure with a special functor.

```
parm_tag                    := perm_var_tag   |
                               temp_var_tag   |
                               unsafe_tag     |
                               perm_val_tag   |
                               temp_val_tag   |
                               const_tag      |
                               struct_tag     |
                               list_tag       ;

                               /*see Warren [7]*/
parm_body(perm_var_tag)     := perm_var_index;

parm_body(temp_var_tag)     := temp_var_index;
```

```
parm_body(unsafe_tag)      := unsafe_index;
parm_body(perm_val_tag)    := perm_val_index;
parm_body(temp_val_tag)    := temp_val_index;
parm_body(const_tag)       := const_no;
parm_body(struct_tag)      := func_no,
                              parm_block;
parm_body(list_tag)        := parm_block;
```

The parm_block parameter represents the actual code to be executed, whereas all the run time information is passed to AIK-0 by the context_ptr. The variable bindings are grouped into local_stack and global_stack:

```
context      := context_head,
                local_stack,
                global_stack,
                trail;

context_head := cont_father,   /*ptr. to father context*/
                loc_anc,       /*ptr. to local stack*/
                glob_anc,      /*ptr. to global stack*/
                trail_anc;     /*ptr. to trail*/

local_stack  := ref_list,
                other-info_1;  /*other information*/
                               /*unknown to AIK-0*/

global_stack := ref_list;
                other-info_2;  /*other information*/
                               /*unknown to AIK-0*/

trail        := ref_list;

ref          := ref_type,
                ref_mode,
                ref_level,
                (ref_offset | get_ptr_offsets);

ref_type     := local_var  |   /*reference to local var*/
                global_var |   /*reference to global var*/
                const      |   /*reference to constant*/
                struct     |   /*reference to structure*/
                list       |   /*reference to list*/
                get_ptr    ;   /*reference to copied var*/
```

Get-pointers (ref_type = get_ptr) are required to support the copying of goal variables with OR-parallelism (see Section 5 and [6]):

```
ref_mode     := read       |   /*variable is read only*/
                write      ;   /*variable may be bound*/
```

All the unifications and therefore all the bindings take place within AIK-0. When variables are to be bound in a context located in the shared read/write memory, GET uses the appropriate CDAA instructions (LOCK-DF, READ-UNSAFE, COMPARE-AND-SWAP) to access the data fields.

The GET instruction is a generalization of the GET instructions of Warren [7]. Since control flow will take place above the AIK-0 interface, a compiler producing code for sequential processing can produce code equivalent to that of the Warren machine. All register optimizations are possible, as in [7]. The identity of argument registers and temporary variable registers can be exploited; temporary and permanent variables can be distinguished.

When AIK-0 is implemented in microcode, the data representation is fixed in the hardware. It is possible to enrich the set of tags to support a greater diversity of data types, e.g., integers, floating point numbers, strings, etc. Generic data types supporting several different tags for types with identical internal data representation would allow the user to distinguish between different kinds of data of the same basic type.

Whether the unification algorithm used by the GET instruction should incorporate the occur check is left open. It is a question of performance. If it is available, whether the occur check should take place or not may be specifiable by a parameter. In a software solution a unification error due to the missing occur check results in a stack overflow; in a hardware solution it results in an exception.

PUT (context_ptr,parm_block,arg_block,gst_ptr)

The PUT instruction builds an arg_block for a call of a procedure corresponding to a goal in the body of a clause. This may include creation of a reference in the global stack in case that parm_tag = unsafe_tag or parm_tag = temp_var_tag. In the cases of structures or lists, copies are made on the global stack as proposed by Warren [7] ("unification in write mode").

UNBIND (context_ptr,ref,length)

The flow of control of a logic program (especially the initiation of backtracking) is not controlled by AIK-0. Therefore, the AIK-0 instruction UNBIND is provided to undo previous bindings.

The UNBIND instruction removes the bindings of all references listed between ref and ref + length. All the unbound references have the same status as if they had never been bound before.

DEREF (context_ptr,ref,r1,r2)

The DEREF function scans the chain of references (through all levels) and returns in r1 the term to which the reference ref is bound. This includes the handling of read-only variables and of get-pointers. The AIK-0 concept for the treatment of these two types of references is similar to the one described in [6]. Field r2 points to the first get-pointer or the last read-only variable.

When moving through the reference chain, DEREF uses the appropriate CDAA instructions (LOCK-DF, READ-UNSAFE, COMPARE-AND-SWAP) to access contexts located in the shared read/write memory.

PMATCH (search-area,format-descr,pattern-spec,result-area)
The PMATCH instruction supports pattern matching in a general way. Pattern matching is applied to an area in virtual storage. The area may be structured, for example, into fixed- or variable-length character strings. This is specified by the format-descr parameter:

```
format-descr := UNSTRUCTURED          |
                FIXED-LENGTH(1)       |
                VARIABLE-LENGTH-1     |
                LINKED-LIST-1         |
                other;
```

With VARIABLE-LENGTH-1 and LINKED-LIST-1 a specific format for the representation of the string length and the list chaining is assumed. The pattern specification "pattern-spec" supports operations known from string manipulation languages, such as SNOBOL and ICON, and from A.I. languages, such as Planner and Popler (see [8–10]):

```
pattern-spec := p-match-op-list   |
                (pattern-spec)    |
                pattern-spec OR pattern-spec;

p-match-op := string      |
              SKIP(1)     |
              POS(n)      |
              VAR         |
              ANY(string) |
              READ;
```

The meanings of these operations are as follows:

string	match against the specified character string
SKIP(1)	skip 1 character
POS(n)	match only if at position n
VAR	skip a variable number of characters until the next operation matches
ANY(string)	match current character against any of the characters in string
READ	skip a variable number of characters until the next operation matches and read the characters in between into the result area

Examples:

Pattern-spec := 'Manager of' READ 'is' READ '.'
would return 'Smith', 'Jones' if applied to the string 'Manager of Smith is Jones.'

Pattern-spec := '.' 'Therefore' finds all sentences starting with 'Therefore'.

A pattern-spec in the syntax described above has to be translated (compiled) into a mask before the PMATCH instruction can be invoked. This results in operations which offer the potential for microcode and hardware assistance (see [11, 12]).

The result of the search is returned in result-area as a list of character strings and the position where the search finished. If the result-area overflows, search is interrupted and the position of interruption is returned in result-area. This position may then be used to continue search with a repeated invocation of PMATCH.

5. Logic programming with AIK-0

One of the primary objectives of AIK-0 is to permit support of a large variety of logic programming implementations and of different language features. This section shows the spectrum of solutions supported by AIK-0, rather than the details of a specific solution.

• *Unification, dereferencing*
Unification is the function which matches the arguments of an actual goal against the parameters of a clause head. Its efficiency largely determines the efficiency of the whole logic programming implementation.

AIK-0 supports this function by the GET instruction. This requires some assumptions with respect to the format in which the logic programming variables are stored in memory and the types of variables to be distinguished. Besides the logic program constructs, such as variable, constant, structure, and list, AIK-0 also allows one to distinguish cases which have proven to be useful (see [7, 13, 14]) for the implementation of Prolog compilers, such as local variable, global variable, permanent variable, temporary variable, etc. In addition, Read-only variables are supported as required with certain parallel logic programming languages such as Concurrent Prolog and Parlog (see below). However, this does not mean that all logic programming implementations based on AIK-0 have to distinguish all these types of variables and references. Implementations with less sophisticated variable types are imaginable. Conversely, it may also be reasonable to handle simple types of unification, e.g., void variables (variables which occur only once in the clause) without the use of the GET instruction of AIK-0.

Dereferencing as a separate function (DEREF) is required when variables are accessed for purposes other than unification, for example, inspected or copied (see OR-parallelism).

• *Storage management*
The AIK-0 instructions GET, PUT, DEREF, UNBIND support a structuring of logic programming data into Local Stack, Global Stack, and Trail, as known from literature on Prolog implementations [7, 15]. But these instructions can also be used to operate on less complex structures, e.g., on a single stack.

When parallel processing is supported, it is useful to structure the logic program state into multiple contexts, such that a single context comprises the information which is uniquely related to a set of processes. Shared contexts should be allocated in the Shared Read/Write Memory. Private contexts may initially (as long as they are private) be allocated in Private Memory and then, when additional parallelism is started, moved to the Shared Read/Write Memory; or they can be allocated in Shared Read/Write Memory right at the beginning.

- *Parallel processing*

The AIK-0 instructions described in Section 4 have been designed with the view of supporting AND-parallelism as well as OR-parallelism. However, only the low-level part of the parallel processing support is covered by the AIK-0 instructions. The layer above AIK-0 must include the high-level parallel processing support, e.g., support of control flow.

Besides taking the unchanged, original Prolog (so-called Pure-Prolog) as the logic programming language for parallel execution, Prolog derivatives such as Concurrent Prolog [16] and Parlog [17] are especially aimed at support of parallel processing. These various approaches and languages differ in

- The way parallelism can be specified and controlled.
- The way access to shared variables is controlled.
- Whether single and/or multiple solutions are supported.
- Some other language features.

AIK-0 is not tailored towards any specific alternative within this spectrum of different concepts but aims to be suitable for support of different approaches.

- *Process management*

The efficiency of parallel processing support depends heavily on the efficiency with which processes can be created and dispatched. The instructions ATTACH, RESET, WAIT, and MAKE-DISPATCHABLE have been included in AIK-0 to allow an optimal realization of these functions by use of microcode and/or hardware assistance.

Nevertheless, it is anticipated that the creation of new processes and their dispatching on multiple processors will still result in a noticeable overhead. It is therefore recommended that the creation of parallel processes be constrained by the following three means:

1. *By language features.* In Parlog it is possible to distinguish parallel and sequential execution by use of different syntactical symbols. In Concurrent Prolog and Parlog clause bodies may be structured such that they start with guards. Guards constrain the parallelism in that only the guards are executed in parallel. As soon as one guard out of a set of alternative clauses matches (i.e., unifies) the goal, the execution of all alternative clauses is discontinued.
2. *By implementation-specific considerations.* It may be reasonable to exploit parallelism only when certain conditions are met, rather than in all instances allowed by the language. As an example, with Pure Prolog it may be reasonable to exploit AND-parallelism only when there are no shared variables between the AND-node branches (see Restricted AND Parallelism in [18, 19]).
3. *By observing the processor workload.* New processes should only be generated if the available processors are not too heavily utilized by the existing processes. AIK-0 maintains a free processor count which can be used to decide whether or not creation of additional processes is useful.

- *OR-parallelism*

For support of OR-parallelism it is necessary to copy the program state to prevent conflicting initialization attempts of the same variable by multiple alternative clauses. There are three major alternatives for copying the program state:

1. Copying of complete state.
2. Copying of goal variables only.
3. Demand-driven copying (see Levy [6]).

Copying of goal variables (including demand-driven copying) is supported by AIK-0 in terms of get-pointers as a special type of reference (see Section 4, GET instruction).

Copying of the program state has to include the local copy of the shared read/write data in the cache of the CDAA. Copying of the cache data can be accomplished by the instructions CHECKPOINT-CACHE and RESTORE-CACHE.

- *AND-parallelism*

Multiple conjunctive goals of a clause body are executed in parallel. Support of AND-parallelism has to address the following two interrelated problems:

1. *Concurrent access to variables shared by multiple goals.* Data which are shared between parallel processes should reside in the Shared Read/Write Memory. Access to these data is performed by CDAA instructions. Simple tests (e.g., whether a certain branch is already started) can be performed by use of COMPARE-AND-SWAP. More complex tests and updating have to use LOCK-DF and UNLOCK-DF. Waiting for a data field to get a specific value is implemented by use of CLEAR-DF and LOCK-DF.
2. *Backtracking.* When multiple conjunctive goals share a variable, the variable might be instantiated by one goal to a value which might not satisfy another goal. With nonparallel logic programming, the commonly acceptable

variable value is found by backtracking, provided there is one at all. With parallel logic programming, simple forms of backtracking, e.g., backtracking only the individual goals in an uncoordinated way, is not sufficient. Concurrent Prolog supports only so-called *determinate AND-parallelism*, which does not perform any backtracking. If backtracking is still to be supported, this can be accomplished by more complex schemes, such as the generation of variable streams as proposed by Conery [20].

6. Support of other programming languages for writing expert systems

It is conceivable that many expert systems will still be written in languages other than Prolog. Three types of languages are anticipated:

1. *Languages which have been designed for support of artificial intelligence applications.* OPS5 [21], derivations of Planner, such as MicroPlanner and CONNIVER, and similar languages, e.g., Popler and POP2 (see [9, 10, 22, 23]) belong to this category.
2. *LISP* [24]. This is a language which, although not specifically aimed at support of A.I., is very popular for writing A.I. applications.
3. *General-purpose languages.* In order to achieve optimal execution performance, it is also the practice to write expert system shells in a general-purpose programming language, such as Pascal.

Although AIK-0 puts emphasis on supporting logic programming languages, it aims at offering assistance to all three types of languages listed above. There are some common characteristics and requirements which can be seen for all these languages (at least when used for the implementation of expert systems). There are four areas which could probably benefit from an extended computer architecture such as AIK-0:

- *Parallel processing.* The Concurrent Data Access Architecture, CDAA, provides flexible and powerful functions for support of different parallel processing applications and for controlling the concurrent access to shared data.
- *Backtracking.* Some of the languages mentioned above (e.g., Planner, POP-2) support backtracking as an explicit language feature. The CDAA cache management instructions (CLEAR-CACHE, COMMIT-CACHE, CHECKPOINT-CACHE, RESTORE-CACHE) are very powerful for implementing backtracking of different styles.
- *Storage management.* With dynamic variable binding, which most of the above languages support, storage allocation and deallocation is a key area where assistance by a suitable hardware architecture is desirable, especially in a parallel processing environment. The GETM and FREM instructions described in Section 4 offer this assistance.
- *Pattern matching.* Pattern matching functions are required to
  1. Search through a data base or knowledge base for specific patterns.
  2. Support pattern-directed invocation schemes, e.g., for A.I. languages such as Planner, CONNIVER, and Popler.
  3. Realize part of the GET instruction.

The PMATCH instruction described in Section 4 has been designed for support of these applications.

7. Summary

In the preceding sections a computer architecture called AIK-0 for support of expert systems and logic programming has been described. AIK-0, which is represented primarily by a set of machine instructions, could be an extension of most traditional von Neumann computers. AIK-0 is called an experimental computer architecture because the area of expert systems and logic programming is not yet considered to be stable enough to aim for a firm computer architecture. Areas where the utilization of AIK-0 may help in achieving some consolidation are

- Finding the right degree of compilation and degree of microcode or hardware assistance.
- Determining the extent to which parallel processing may speed up expert system applications and suitable solutions for doing this.
- Determining heuristic methods for the optimal scheduling of AND-parallelism and OR-parallelism.

8. References

1. T. Moto-oka and K. Fuchi, "The Architecture in the Fifth Generation Computers," *Proceedings of IFIP 1983*, North-Holland Publishing Co., Amsterdam, p. 589.
2. J. A. Crammond and C. D. F. Miller, "An Architecture for Parallel Logic Languages," *Proceedings of the Second International Logic Programming Conference,* Uppsala, Sweden, 1984, p. 183.
3. Z. Segall, A. Singh, R. T. Snodgras, A. K. Jones, and D. P. Siewiorek, "An Integrated Instrumentation Environment for Multiprocessors," *IEEE Trans. Computers* **C-32**, 4 (1983).
4. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer-Verlag New York, 1981.
5. H. Diel, "Concurrent Data Access Architecture," *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, 1984, p. 373.
6. J. Levy, "A Unification Algorithm for Concurrent Prolog," *Proceedings of the Second International Logic Programming Conference*, Uppsala, Sweden, 1984, p. 331.
7. D. H. D. Warren, "An Abstract Prolog Instruction Set," *Technical Report 309*, SRI International, Menlo Park, CA, 1983.
8. R. E. Griswold, J. F. Poage, and I. P. Polonsky, *The SNOBOL4 Programming Language*, Second Edition, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1971.

9. C. Hewitt, "Planner: A Language for Manipulating Models and Proving Theorems in a Robot," *AI Memo 168*, Massachusetts Institute of Technology, Cambridge, MA, 1970.
10. D. J. M. Davies, "Popler 1.5 Reference Manual," *TPU Report No. 1*, Theoretical Psychology Unit, Edinburgh, Scotland, 1973.
11. S. Arikawa and T. Shinohara, "A Run-Time Efficient Realization of Aho Corasick Pattern Matching Machines," *New Generation Computing* **2**, 171 (1984).
12. R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Commun. ACM* **20**, 762 (1977).
13. D. L. Bowen, L. M. Byrd, and W. F. Clocksin, "A Portable Prolog Compiler," *Logic Programming Workshop '83*, Universidade Nova de Lisboa, Portugal, June 1983, pp. 74-83.
14. D. H. D. Warren, "Implementing Prolog—Compiling Predicate Logic Programs," *Research Reports 39 and 40*, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1977.
15. M. Bruynooghe, *The Memory Management of Prolog Implementations, Logic Programming*, K. L. Clark and S.-A. Taernlund, Eds., Academic Press, Inc., New York, 1982, p. 83.
16. E. Y. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter," *Technical Report TR003*, ICOT, Institute for New Generation Computer Technology, Tokyo, Japan, 1984.
17. K. Clark and S. Gregory, "PARLOG: Parallel Programming in Logic," *Research Report DOC 84/4*, Department of Computing, Imperial College, London, April 1984.
18. D. DeGroot, "Restricted AND Parallelism," *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, 1984, p. 471.
19. J. H. Chang and D. DeGroot, "AND-Parallelism of Logic Programs Based on Static Data Dependency Analysis," *Proceedings of AFCET Conference on Fifth Generation Computer Systems* (Agence de l'Informatique Centre National d'Etudes des Télécommunications), Paris, 1985, p. 271.
20. J. S. Conery, "The AND/OR Model for Parallel Interpretation of Logic Programs," Ph.D. Thesis, University of California, Irvine, CA, 1983.
21. C. L. Forgy, "OPS5 User's Manual," *Technical Report CMU CS-81-135*, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1981.
22. G. J. Sussman and T. Winograd, "MicroPlanner Reference Manual," *AI Memo 203*, Massachusetts Institute of Technology, Cambridge, MA, 1970.
23. D. V. McDermott and G. J. Sussman, "The CONNIVER Reference Manual," *AI Memo 259*, Massachusetts Institute of Technology, Cambridge, MA, 1972.
24. W. Teitelman, "INTERLISP Reference Manual," XEROX Palo Alto Research Center, Palo Alto, CA, 1974.

Received March 5, 1985; revised August 8, 1985

Hans H. Diel *IBM Germany, Böblingen, Federal Republic of Germany.* Mr. Diel is currently senior programmer in the Advanced Technology Group in DSD Böblingen working on advanced system structures. He studied mechanical engineering in Saarbrücken. Mr. Diel joined IBM in 1965 and has worked in areas such as compiler development, language design, performance analysis, and operating system development.

Norbert G. Lenz *IBM Germany, Böblingen, Federal Republic of Germany.* Dr. Lenz is a system development engineer in the Advanced Technology Systems I Department working on advanced system structures. He received his doctor's degree in mathematics at the University of Mainz, Federal Republic of Germany, in 1982 and joined IBM the same year.

H. Martin Welsch *IBM Germany, Böblingen, Federal Republic of Germany.* Dr. Welsch is a system development engineer in the Advanced Technology Systems I Department working on advanced system structures. He received his doctor's degree in physics at the University of Marburg, Federal Republic of Germany, in 1984 and joined IBM the same year.