PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT.

CONCURRENT DATA ACCESS ARCHITECTURE

H. Diel, IBM Laboratory, Boeblingen, Germany

ABSTRACT

The Concurrent Data Access Architecture (CDAA) has been designed with the objective to support a high degree of parallelism. This goal is similar to that of data flow architecture and other computer architectures which have been proposed in recent years. CDAA, in addition aims at providing further support for those application areas that could probably utilize best the increased parallelism. Therefore, additional features such as backtracking (for Artificial Intelligence applications) and BACKOUT/COMMIT (for Transaction Processing) are embedded into the concept. The Concurrent Data Access Architecture is an evolutionary extension of existing machine architectures. This results in the advantage that traditional applications can still run on computers supporting CDAA.

1 OVERVIEW

With the emergence of VLSI, which permits more and more functions to be packed into a single chip, the time has come to direct the attention of processor designers away from the goal to further optimize individual functions,and to search for solutions which allow the execution of a large number of such functions in parallel. The Concurrent Data Access Architecture CDAA has been designed with the goal to support drastically increased parallelism.

In order to achieve this, it is believed that the primary problem to be solved is that of concurrent data access to shared data items by multiple processors. Besides supporting increased parallelism, CDAA offers additional support for those application areas which could probably best utilize the increased parallelism, namely artificial intelligence applications and transaction processing.

The overall concept of CDAA can be described by four items:

1. The totality of memory data which can be accessed by a processor is divided into the following parts:

   a. Processor Private Data

   b. Shared Read only Data

   c. Shared Read/Write Data

   The collection of this data is henceforth called the data base and it is the main subject of this paper.

   With an optimal implementation of CDAA, these three logical types of memory would probably be implemented by different physical memories. However, less optimal implementations may still be useful. One could imagine CDAA-supported processor clusters with (physically) shared memory only,or vice versa which have (physically) private memory only. For CDAA it is only required that it is possible to distinguish the above logical parts,for example by specific address ranges.

2. The data base can only be accessed in a controlled way. This means

   a. Prior to accessing the data, a machine instruction LOCK Datafield must be issued and the type of access READ or WRITE must be specified.

   b. When the data has been accessed, it has to be explicitly released by the UNLOCK Datafield machine instruction.

   c. Initialization of data and the arrival of data values for a particular datafield can be determined by explicitly clearing the datafield by the CLEAR datafield instruction. Together with the ability to wait for unitialized datafields this provides a facility comparable with the concept of data flow architectures.

3. Each processor has a cache associated with it. The association of cache to

processor may be fixed, i.e. there are as many caches as processors, or one could even have a larger number of caches than processors and perform a dynamic association of cache to processor.

Besides the traditional purpose of a cache (which is to allow faster access to data), with CDAA the cache has the purpose to hold a local copy of the data base.This supports two goals:

a. It enables an efficient realization of the explicitly controlled data access described above. This is achieved by assuming that data base data is accessible only when it resides in the cache; and the data is moved into the cache only as the result of the LOCK datafield instruction.

b. It supports versions of data and thereby the facilities to backout (i.e. transaction processing) or backtrack (i.e. artificial intelligence) to previous states. These facilities are provided by cache management instructions COMMIT-CACHE, CHECKPOINT-CACHE, and RESTORE-CACHE.

Instead of having a true cache one could imagine CDAA implementations in which the local data base copy is kept in normal memory (without cache-like characteristics). In this case,however it would be very difficult to detect access to data which is not yet locked or initialized. This may still represent a useful subset of CDAA, however the full CDAA described in this paper assumes the detection of access to unlocked and uninitialized data. In the following the terms 'cache' and 'Local-DB-copy' are used interchangeably.

4. The dispatching of processes is controlled by CDAA.This means

   a. A process is set into wait state when it tries to obtain an inaccessible lock or to read uninitialized data.

   b. A process is resumed when data it has been waiting for becomes available.

It is assumed that the implementation of CDAA supports these functions by direct hardware to the extent necessary to provide suitable performance.

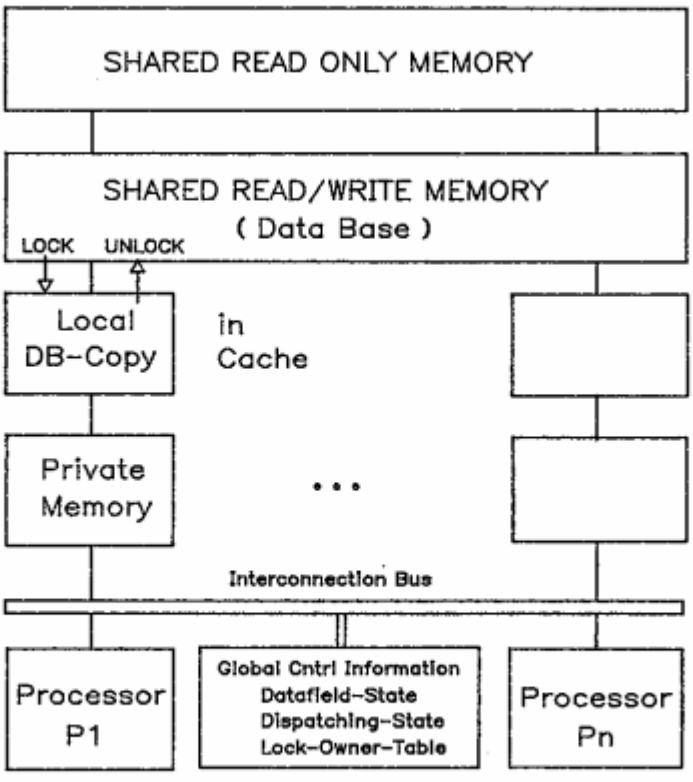

Figure 1: Overview on CDAA

2 CDAA INSTRUCTIONS

This section describes the CDAA instructions. Three groups of instruction are distinguished:

- Datafield Management Instructions
- Cache Management Instructions
- Process Management Instructions

The instructions are described in a kind of pseudocode to keep the description compact by still maintaining a reasonable degree of detail and preciseness.

2.1 MACHINE STATE

The state of a CDAA controlled machine consists of the following information:

```
Machine-State:=
   Processor-state-list,
   Local-DB-copy-list,
   Traditional-state-list,
   Datafield-state,
   Dispatching-state,
   Lock-owner-table;
```

The first three items (the lists) denote distributed information, whereas the latter items denote global control information. Traditional-state comprises all state information required to describe the general, non-CDAA related part of the machine semantic. It is not further described here.

```
Processor-state:=
   CDAA-action,
   Current-process-id,
   Current-memory-unit-id;
```

The CDAA-action defines for an individual processor how this processor reacts when a referenced datafield is not accessible (e.g. not initialized or locked). Two alternatives WAIT and signal ERROR are supported.

```
CDAA-action:=
   Initialize-action:= WAIT v ERROR ,
   Lock-action:= WAIT v ERROR;

Current-process-id:= Number;
Current-memory-unit-id:= Number;
```

The Datafield-state defines for **all datafields of the data base** , the information CDAA needs in order to control access to the datafield.

```
Datafield-state:=
      Datafield-attribute -list;

Datafield-attribute:=
   CLEARED v
   WR-CLEARED v
   READ-LOCKED v
   WRITE-LOCKED v
   EMPTY;
```

Dispatching-state describes the information required by CDAA to perform the deactivation (wait) and dispatching of processes known by CDAA.

```
Dispatching-state:=
       Dispatch-descr-list;

Dispatch-descr:=
   Process-id:= Number,
   Memory-unit-id:= Number v 0,
   Process-status:= WAIT v
             DISPATCHABLE v
             ACTIVE,
   Wait-condition:= W-LOCK v
             other-wait-condition v
             NOWAIT,
   Datafield-address:= address;

Local-DB-copy:= Datafield-list;
Datafield:=  address, value;
```

2.2 DATAFIELD MANAGEMENT

The three instructions described below control access to the datafields of the data base. The instructions apply to areas of arbitrary length and alignment. A concrete realization of CDAA, however,will probably work on fixed size datafields (e.g. 32 bytes). If the area specified by an instruction does not match the datafield boundaries, CDAA will extend it to the next boundaries. This has no effect on the semantics of the instructions, but affects performance only. If the specified area spans multiple datafields, CDAA will apply the instructions automatically to the multiple datafields.

2.2.1 CLEAR-DF (address,length)

The purpose of this instruction is the detection of read type references to the datafield before this datafield is initialized. After issuing a CLEAR-DF instruction, the specified datafield is still not accessible by the process, but an additional LOCK-DF with WRITE access must be issued before the

datafield can be accessed (e.g. initialized). If a data field to be cleared is locked by another process, the current process is set into wait state. Deadlocks are detected by use of the lock-owner-table and will be signalled via a special condition code value.

```
CLEAR-DF (address, length):=
For all data-words Wi
      within Range (address, length)
DO;
IF ¬ Accessible (Wi, CLEAR) THEN
   DO;
   CALL CDAA-action (Wi, CLEAR);
   RETURN;
   END;
ELSE Datafield-attribute(Wi)=CLEARED;
END;
```

2.2.2 LOCK-DF (address,length,type)

The LOCK-DF instruction must be issued before a datafield can be accessed by a processor. 'type' 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 data base to the local data base copy (i.e. cache). If a data field to be locked is already locked by another process (except if it is a read lock request for a read-locked data field) ,the current process is set into wait state. Deadlocks are detected by use of the lock-owner-table and will be signalled via a special condition code value.

```
LOCK-DF (address, length, type):=
FOR all datafields Wi
     within Range (address, length)
DO;
IF ¬ Accessible (Wi, type) THEN
   DO;
   CALL CDAA-action (Wi, type);
   RETURN;
   END;
ELSE DO;
     IF type = READ THEN
     Datafield-attribute (Wi) =
                    READ-LOCKED;
     IF type = WRITE THEN
     IF Datafield-attribute (Wi) =
                    CLEARED THEN
     Datafield-attribute (Wi) =
                    WR-CLEARED;ELSE
     Datafield-attribute (Wi) =
                    WRITE-LOCKED;
     Local-DB-copy =
     Local-DB-copy || Datafield (Wi);
     END;
END;
```

2.2.3 UNLOCK-DF(address,length,type)

This instruction is used to release locks held for datafields and to commit the data field values in case of WRITE locks. UNLOCK-DF for a read lock eliminates the current process from the lock-owner-table.If the lock-owner-list for a data word becomes empty the read lock for this data word is released. 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 local data base copy back to the shared data base. A further result of the UNLOCK-DF instruction is that processes waiting for the unlocked data fields are made dispatchable.

```
UNLOCK-DF(address, length, type):=
FOR all data words Wi
     within Range (address, length)
DO;
IF ¬ Accessible (Wi, UNLOCK) THEN
     DO;
     CALL CDAA-action(Wi,UNLOCK);
     RETURN;
     END;
IF Datafield-attribute(Wi)=READ-LOCKED
THEN DO;
     IF type = READ THEN DO;
          CALL EXCEPTION (E3);
          RETURN; END;
     Remove process from lock-owner-list;
     IF last-owner THEN DO;
        Release from local-DB-copy(Wi);
        Make-Dispatchable(0,W-LOCK,Wi);
        END;
     END;
IF Datafield-attribute(Wi)=WRITE-LOCKED
v  Datafield-attribute(Wi)=WR-CLEARED
THEN DO;
     IF type = READ THEN
     Datafield-attribute(Wi)=READ-LOCKED;
     Write-Back-to-DB (Wi);
     Make-Dispatchable(0,W-LOCK,Wi);
     END;
END;
```

2.3 CACHE MANAGEMENT

2.3.1 CLEAR-CACHE

The contents of the cache are cleared and all locks held by the processor are released.

```
CLEAR-CACHE:=
FOR all datafields Wi in Local-DB-copy
RELEASE(Wi);
```

2.3.2 COMMIT-CACHE

The datafield values of the cache are written (committed) to the database.

```
COMMIT-CACHE:=
FOR all datafields Wi in Local-DB-copy
Write-Back-to-DB(Wi);
```

2.3.3 CHECKPOINT-CACHE (address,length)

The contents of the cache are saved at the specified area. By use of the RESTORE-CACHE instruction the saved copy can be reinstalled at a later point in time. This facility is useful for two purposes:

- To establish versions of data and to be able to backout (Transaction Processing) or backtrack (Artificial Intelligence) to previous versions.
- To support process switching in cases where this might be requested on top of the automatic process management provided by CDAAA.

```
CHECKPOINT-CACHE (address, length):=
IF cache-size > length THEN
     CALL EXCEPTION;
ELSE DO;
     Area (address, cache-size) =
                 Local-DB-copy;
     length = cache-size;
     END;
```

2.3.4 RESTORE-CACHE (address,option)

The cache is loaded with the datafields 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 exactly 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 (i.e. duplicated). Thus, it does not interfere with the original datafields.

```
RESTORE-CACHE (address,option):=
Local-DB-copy =
   Area (address, saved-length);
IF option = COPY THEN
   Relocate-cache-addresses(address);
```

'saved-length' is part of the checkpoint data stored at the specified address.Relocate-cache-addresses modifies the addresses of the datafields such that they point to the checkpoint copy.

2.4 PROCESS MANAGEMENT

The dispatching and quiescing of processes as a result of CDAA controlled events (e.g. data fields being locked/unlocked or uninitialized/initialized) is performed automatically by CDAA and does not require any explicit instructions. However, since CDAA contains the necessary functions anyway,it is reasonable to offer them explicitely for control of non-CDAA related process management.

2.4.1 WAIT (process-id,waitcondition,datafield)

The specified process (or present process,if process-id=0) is set into wait state until the specified wait condition is signalled for the specified data field.The instruction can be invoked explicitely or implicitely by CDAA-Action (see section 2.5).

```
WAIT (process-id,WC,DF) :=
IF process-id = 0 THEN
          pid = current-process-id;
          ELSE pid = process-id;
  Process-status(pid)= WAIT;
  Datafield-address(pid)= DF;
  Wait-condition(pid)= WC ;
  /* Activate other processes */
  CALL ACTIVATE-PROCESS;
```

After a process has entered the Wait State,ACTIVATE-PROCESS is invoked to determine whether another process is ready to be dispatched on the present processor. Two cases are distinguished with respect to dispatchable processes:

1. The process has a memory unit associated, i.e. the processes data is still in real memory and cache.
2. The process has no memory unit associated. This case happens if the number of processes is greater than the number of memory units and therefore data has to be

swapped out of the memory units to service all processes. The functions of swapping out processes from memory units and swapping them in again have to be provided by software and are not described here. They can be implemented by use of the cache management instructions CHECK-POINT-CACHE and RESTORE-CACHE.

```
ACTIVATE-PROCESS :=
/* First scan for processes which have
   already a memory unit associated */
FOR all elements I of Dispatch-descr-list
DO;
IF process-status (I) = DISPATCHABLE AND
   Memory-unit-id (I) >  0 THEN
   DO;
   process-status (I) = ACTIVE;
   Current-process-id = I;
   Current-memory-unit-id =
               Memory-unit-id (I);
   RETURN;
   END;
   END;
/* Now scan for processes w/o
   associated memory unit */
FOR all elements I of Dispatch-descr-list
DO;
IF process-status (I) = DISPATCHABLE THEN
   DO;
   Signal-special-event;
   /* i.e. interrupt which causes logic
      for process switch to be invoked
      on current processor */
   RETURN;
   END;
END;
```

2.4.2 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. The instruction can be invoked explicitely or implicitely by the UNLOCK-DF instruction. The function will not necessarily be executed by the processor which executed the UNLOCK-DF instruction, but it may for example, run on a dedicated processor, or at a processor which is waiting anyway.

```
Make-Dispatchable(process-id,WC,DF) :=
   IF process-id = 0 THEN
   process-list = Dispatch-descr-list;
   ELSE process-list = process-id;
   FOR all elements of process-list
   DO;
   IF process-status (I) = WAIT  AND
      Datafield-address (I) = DF AND
         WC = Wait-condition (I)  THEN
      DISPATCH (I);
   END;

DISPATCH (I):=
   Process-status (I) = DISPATCHABLE;
   Datafield-address (I) = EMPTY;
   Wait-condition (I) = NOWAIT;
   FOR all processors J DO;
      IF current-process-id (J) = 0  THEN
      DO;
      Start-process-on-processor(I,J);
      RETURN;
      END;
   END;
```

2.5 GENERAL FUNCTIONS AND PREDICATES

In the preceding sections 2.2, 2.3, 2.4 certain functions and predicates are used which are described in some more detail below.

ACCESSIBLE (W, Type)

This predicate checks if a certain datafield W is accessible in a specific access mode. The predicate is invoked by the datafield management instructions, but in addition also by all general purpose instructions which reference datafields in memory. This leads to the following possible values for the Type parameter: CLEAR, READ, WRITE, UNLOCK, READ-REF (for read references from general purpose instructions), and WRITE-REF (for write references from general purpose instructions). The validity of accessing a datafield depends on the access mode, the datafield-attributes of the referenced datafield, and on whether or not the datafield is available in the cache.

```
Accessible (W,Type):=

IF is-element-of (Local-DB-copy, W) THEN
   SELECT(Type);
   WHEN( READ-REF )  --> true,
   WHEN( WRITE-REF v CLEAR)  -->
   IF Datafield-attribute(W)=READ-LOCKED
                        THEN false,
                        ELSE true,
   WHEN( UNLOCK   )  --> true,
   OTHERWISE  --> false ;
   END;
ELSE
  /*i.e. not contained in Local-DB-copy*/
   SELECT(Type);
   WHEN( WRITE v CLEAR) -->
   IF Datafield-attribute(W)=CLEARED v
      Datafield-attribute(W)=EMPTY
                           THEN true,
                           ELSE false;
   WHEN( READ)  -->
   IF Datafield-attribute(W)=READ-LOCKED
   v  Datafield-attribute(W)=EMPTY
                           THEN true,
                           ELSE false;
   OTHERWISE   --> false;
   END;
```

CDAA-action (W, type)

This function is invoked whenever it is detected (by a datafield management instruction or by a general purpose instruction) that a requested datafield is not accessible.

```
CDAA-action (W, type):=
IF is-element-of (Local-DB-copy, W) THEN
DO;
IF type =CLEAR THEN CALL EXCEPTION(E1);
IF type =READ THEN CALL EXCEPTION(E3);
IF type =WRITE THEN CALL EXCEPTION(E3);
IF type =WRITE-REF THEN CALL EXCEPTION(E1);
END,
ELSE DO;
IF type =CLEAR v type = READ v type =WRITE
             THEN CALL WE(W-LOCK,W);
IF type =UNLOCK THEN CALL EXCEPTION(E2);
IF type =READ-REF THEN
   DO;
   IF Datafield-attribute (W)= CLEARED
   v  Datafield-attribute (W)= WR-CLEARED
   THEN IF
   Initialize-action (current-process-id)=
   ERROR  THEN CALL EXCEPTION(E4);
          ELSE CALL WAIT(0,W-LOCK,W);
   END; ELSE CALL EXCEPTION(E1);
IF type = WRITE-REF THEN CALL EXCEPTION(E1);
END;
/* E1: Access Exception: Reference w/o
       correct lock
   E2: UNLOCK for datafields not access-
       ible by present process
   E3: Multiple lock requests by same
       process for same datafield
   E4: Reference to unavailable or
       uninitialized datafield    */

WE (type,W ):=
IF Lock-action(current-process-id)=ERROR
      THEN CALL EXCEPTION(E4);
      ELSE CALL WAIT(0,W-LOCK,W);
```

3 IMPLEMENTATION CONSIDERATIONS

3.1 MEMORY ORGANIZATION

As already mentioned in Section 1,'Overview', an optimal implementation of CDAA would support the three logical types of memory, processor private data, read only shared data,and the data base by different physical memories. However, less optimal implementations may still be useful. For example, the processor private data and the read only data could easily be supported by a common,physically shared memory. It is a special characteristic of CDAA, that it is also feasible to represent the logically shared data (the data base) by multiple, processor-local memories. It requires that, whenever data is committed (by use of UNLOCK-DF), the data has to be communicated to the multiple, processor-local memories.

Section 1 also mentions, that cache-less implementations of CDAA may still be useful if the automatic detection of unlocked and uninitialized datafields is relinquished. If a cache is available (as with full CDAA), it need not be restricted to support of the local data base copy, but may also be applied to the processor private data and the read-only shared data. Separate read and write caches per processor might also be useful.

The value or necessity of the above described implementation alternatives depends very much on the type of application for which the CDAA-supported processors will be used (see section 4).

3.2 PROCESS DISPATCHING

Besides efficient memory organization, this is the second area of CDAA where an efficient implementation is critical for the feasibility of CDAA in general. It is pro-

posed that the CDAA perform a certain degree of saving and restoring of process status (e.g. Program Status Word, registers, cache contents) automatically. In addition, a software-provided (small) exit routine is invoked which performs additional operating-system-dependent saving or restoring. Three cases have to be considered:

1. Setting processes into wait state
   This is performed automatically when a data field which is attempted to be accessed is not accessible (locked or uninitialized) or when the Wait instruction is issued. The process status in the dispatching table is changed from 'Active' to 'Waiting'. The saving of the process state and the invocation of the Save-exit-routine however, is deferred until another process gets dispatched on that processor.

2. Dispatching a process
   When a process is dispatched at a processor (because that processor became free and/or the process became dispatchable), CDAA has to update the dispatching table, invoke the Save-exit-routine for the old process and the Restore-exit-routine for the new process, and resume execution according to the Program Status Word.

3. Making processes dispatchable
   It is assumed that the function of scanning and updating the dispatching table is not performed by the processor that caused the datafield to become available, but that a message is sent to a processor which performs this work. In the simplest case this processor will be a dedicated processor (with possibly additional functions). More sophisticated schemes may utilize the fact that certain processors may be in wait state anyway.

3.3 MESSAGE-BASED VERSUS SHARED-MEMORY-BASED PARALLEL PROCESSING.

There is a lot of discussion among computer architects about the value of message-based parallel computer architectures versus parallel processing based on shared memory. Even the argument that the advantage of one type of architecture over the other depends on the type of application for which the processors will be used does not terminate the discussion, but at best restricts it to specific application types.

CDAA cannot properly be associated with one or the other of these two types of parallel processing. It has the advantage that the question of message-based versus shared-memory-based parallel processing does not relate to CDAA as a computer architecture, but is rather a CDAA internal implementation issue. Both types of processing could be supported by CDAA. As already mentioned, CDAA implementations which do not have a (physically) shared memory may be reasonable, depending on the type of application. It requires that the committment of data (when UNLOCK-DF is issued) has to be communicated to the other processors (i.e. messages have to be sent). It seems that, in the same way as this kind of CDAA implementation is not optimal for all applications, message-based parallel processing in general might not be optimal for all kinds of application.

4 CDAA APPLICATIONS

4.1 GENERAL MULTIPROCESSING

Host operating systems (e.g. UNIX, VM/CMS, OS/VS2 MVS) do not allow the read/write sharing of memory data directly by multiple user applications. Distribution of an operating system with this restriction to multiple CDAA-controlled processors would require modifications in the respective operating system only (but not in the user applications). It would provide the image of tightly coupled multiprocessing without implying all the hardware burden of tightly coupled multiprocessing (e.g. cache broadcasting).

4.2 DATA FLOW LANGUAGES

Programs written in a data flow language (see Ackermann 1982) or in a functional programming language could be compiled such that they could run on CDAA utilizing parallel processing. The following CDAA-related work would have to be performed by the compiler:

1. Initialization

   At the beginning of the object program

   - the program pieces which are executable in parallel must be distributed to multiple processes (statically or dynamically),

   - the cache must be cleared by use of the CLEAR-CACHE instruction,

- the datafields being subject to concurrent data access must be marked as not being initialized by use of the CLEAR-DF instruction.

2. Whenever a datafield obtains a value which is input to other processes, the UNLOCK-DF instruction has to be issued.

Processing then functions similarily to data flow architecture,namely when a process references uninitialized data it waits automatically until this data becomes available. This kind of processing could also be achieved with programming languages other than data flow languages,if the compilers were extended accordingly. For example, as pointed out in Shapiro 1983, Concurrent Prolog could be utilized to obtain the behaviour of data flow languages.

4.3 TRANSACTION PROCESSING

In order to discuss how CDAA can be used to support transaction processing, it might be useful to consider an operating system kernel supporting transaction processing as described in H.Diel et al.,1984. The data management kernel described there offers transaction processing facilities (sharing, locking, committment of data, backout to previous state) by a close cooperation between these functions and the virtual memory management functions. As a consequence, the internal sharing/locking granularity is equal to the page size (e.g. 4K bytes). For certain applications this may result in less than optimal performance.

CDAA would be an excellent supplement to such a data management kernel by allowing locking/sharing granularity < page size, and by supporting the process dispatching by the processor architecture. Since it is neither necessary nor feasible to apply the smaller locking granularity to a complete large data base, one could apply CDAA to small data bases or to small parts of data bases with a high degree of concurrent data access.

It should be noticed that the consistency considerations related to distributed database concepts do not affect CDAA support of transaction processing, because the database managed by a single CDAA complex is not a distributed database. Support of 'Two-Phase-Commit', for example,is not required.Consistency control and sharing among multiple CDAA controlled complexes (e.g. multiple network nodes) would be the task of the data base system or data management kernel above CDAA,because these functions also manage the non-volotile storage.

4.4 ARTIFICIAL INTELLIGENCE

The parallel execution of artificial intelligence applications such as Problem Solving, Logic Programming and Theorem Proving distinguishes two types of parallelism:

OR - parallelism:

Alternative paths are investigated. The paths have identical initial state and it is possible to prevent interferences between the alternative paths by working on local copies of the application state only.

To implement OR-parallelism with CDAA it is useful to copy the process state to the multiple processors.This can be done by an instruction sequence such as the following:

```
CHECKPOINT-CACHE( Area, L1 );
Start-Process( Pn );
/* At process  Pn  */
RESTORE-CACHE( Area, L1, COPY);
```

Depending on the actual application, additional functions would have to be performed to start OR-parallelism. OR-parallelism with structure sharing instead of copying,of course could be implemented with CDAA as well.

In case where constrained or designated OR-parallelism is supported, Backtracking may have to be implemented as well.Backtracking can be done by an instruction sequence such as the following:

```
/* At OR-Node          */
CHECKPOINT-CACHE( Area, L1 );
Perform function of individual
OR-Node branch
IF  failure  THEN
RESTORE-CACHE( Area, L1,RESET);
```

AND - parallelism

Multiple paths are executed to evaluate a common goal. In general, the multiple paths share data and thus interfere with each other when the shared data is updated. Different schemes have been proposed to weaken the concurrency problem by restricting the AND-parallelism. For example,Concurrent Prolog (see Shapiro 1983) supports so-called Determinate AND-parallelism. PARLOG (see Clark and Gregory 1984) supports one-way communication between a producer of a variable and a consumer only.L.V.Kale and David S. Warren (see Kale and Warren 1984) have introduced Reduce-Nodes instead of AND-Nodes.

It is believed that given a particular definition of AND-parallelism, CDAA with its controlled access to shared data and its support for backout or committment of data is very suitable for support of AND-parallelism.For example,the one-way communication can be implemented in CDAA by instructions such as the following:

```
CLEAR-DF(X,L)       /* prior to forking     */
LOCK-DF(X,L,WRITE)/* at producer process */
instantiate X       /* at producer process */
UNLOCK-DF(X,L,0)    /* at producer process */
LOCK-DF(X,L,READ)  /* at consumer process */
use X               /* at consumer process */
UNLOCK-DF(X,L,0)    /* at consumer process */
```

5 SUMMARY

The paper describes a computer architecture that supports a high degree of parallel processing. CDAA is suitable for a wide range of application areas (e.g. Transaction Processing, Artificial Intelligence) which could utilize the increased parallelism. In addition CDAA supports further features (e.g. Backout, Checkpointing, Backtracking) which are essential for such application areas.

CDAA leaves much freedom with respect to possible implementation alternatives. Although implementations with shared memory are probably most efficient, implementations without shared memory, but with message communication instead may be appropriate for specific kinds of applications, as well.

6 REFERENCES

1. W.B. Ackermann, Data Flow Languages, IEEE Computer 15(2),1982

2. A. Ciepielewski, S. Haridi, A Formal Model of OR-parallel Execution, IFIP 1983, North Holland

3. K.Clark,S.Gregory, PARLOG:Parallel Programming in Logic, Reaearch Report Doc 84/4, April 1984.

4. J.A. Crammond, C.D.F. Miller, An Architecture for Parallel Logic Languages,Proceedings of Second International Logic Programming Conference, Uppsala,1984

5. H. Diel, N. Lenz, G. Kreissig, M.Scheible, B. Schoener, Data Management Facilities of an Operating System Kernel,Proceedings of SIGMOD 84.

6. L.V. Kale, David S. Warren, A Class of Architectures for a Prolog Machine,Proceedings of Second International Logic Programming Conference, Uppsala,1984

7. I. Lima, D. Mundy, P. Treleaven, Decentralized Control Flow Programming, IFIP 1983, North Holland

8. T. Moto-oka, K. Fuchi, The Architecture of the Fifth Generation Computers, IFIP 1983, North Holland

9. D.P.Reed, Naming and Synchronization in a Decentralized Computer System, Technical Report MIT/LCS/TR-205,Sept. 1978

10. E.Y. Shapiro, A Subset of Concurrent PROLOG and its Interpreter, Technical Report TR003,ICOT Institute for New Generation Computer Technology, 1983.