AN ARCHITECTURAL MODEL FOR A FLAT CONCURRENT PROLOG PROCESSOR

Leon Alkalaj
Ehud Shapiro

February 1988
CSD-880007

An Architectural Model for a Flat Concurrent Prolog Processor

Leon Alkalaj
University of California, Los Angeles
Ehud Shapiro
Weizmann Institute of Science, Rehovot

ABSTRACT

We propose an execution model and a special-purpose processor architecture for the execution of Flat Concurrent Prolog (FCP). The execution model defines internal concurrency inherent to the execution of FCP on a single processor. It is derived by partitioning the FCP Sequential Abstract Machine into concurrently executing units. To support this execution model, the FCP Processor architecture consists of the following concurrent functional processors: Reduction Processor, Tag Processor, Goal Management Processor, Instruction Processor and Data-Trail Processor. The Goal Management Processor performs the efficient management of concurrent FCP goals reduced by the Reduction Processor. The Data-Trail Processor implements a novel cache management algorithm which supports shallow backtracking. The FCP Processor architectural model is specified in FCP itself and is part of a working simulator. The attainable performance of the FCP Processor architecture is currently under investigation.

1 Introduction

Flat Concurrent Prolog (FCP) is a concurrent logic programming language that uses read-only variable annotations to implement data-flow synchronization and allows only simple test-predicates in the Horn clause guards. The language design and an interpreter are specified in [Mier85] and a Sequential Abstract Machine (SAM) similar to the abstract machine for Prolog defined by [Warren83] is proposed in [Houri86]. Both the sequential interpreter and the abstract machine implementations of FCP perform inferior to conventional sequential languages. Improved results are reported in [Klig87] using target host compilation techniques. However, to achieve performance comparable to conventional languages, a special-purpose environment for the execution of FCP is required.

Since FCP is a concurrent programming language, one way of improving performance is to define a concurrent execution model and distribute the execution of an FCP program on multiple processors. For example, preliminary results of implementing FCP on a Hypercube architecture are reported in [Taylor87]. Besides *inter-processor* concurrency, further performance improvements are possible by exploiting parallelism at the processor architecture level, that is the *intra-processor* concurrency. By supporting the parallelism inherent to the execution of FCP on a single processor, one may obtain high-performance processors which could also be used as building blocks in a multiprocessor environment.

The main goal of our research is the design of a high-performance processor for the execution of FCP. It is modelled with the following considerations:

- Internal functional concurrency within the processor is exploited to achieve high-performance.
- A wide-bandwidth memory hierarchy is integrated into the processor architecture to reduce memory response time.

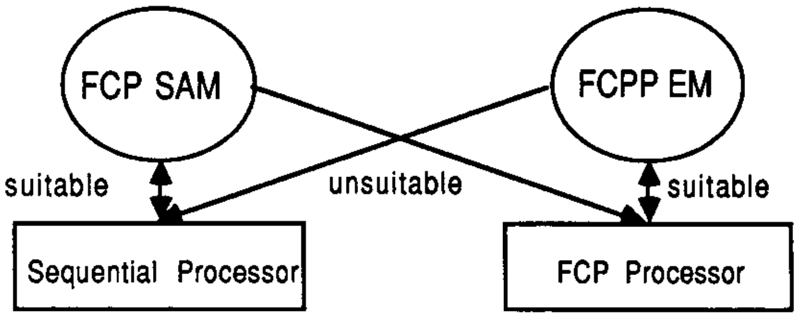

Figure 1: An Execution Model for the FCP Processor

In this paper we describe the architectural model for the FCP Processor. It consists of a set of hierarchally structured communicating functional units. In Section 2 we describe the execution model of the FCP Processor. It is derived by modifying the SAM for FCP to execute on multiple functional units. In Section 3 we describe the Reduction Processor which is the main unit in the FCP Processor. Two features which distinguish FCP from other logic programming languages are supported at the functional unit level:

- The use of *goal invocation* or *process-call* as the basic control mechanism.
- The use of read-only unification as the basic data manipulation primitive.

The former feature is supported by the Goal Management Processor which manipulates goal structures using a Goal Cache. The functionality of the Goal Cache described in Section 4, is similar to the use of multiple-windows for nested procedure calls in RISC processor architectures [Pat82]. The latter feature is supported by the Data-Trail Processor which performs data trailing and the undoing of the trail using a data cache. In Section 5 we describe the Data-Trail Cache policy we call *Delayed Binding.* We conclude this paper by summarizing the properties of the proposed FCP Processor execution model and architecture.

2 The FCP Processor Execution Model

It is common practice in the design and implementation of a high level language to first specify an appropriate, machine independent, abstract machine and instruction set suitable for execution in an existing, often general-purpose processing environment. Since these environments are currently sequential, one readily defines a sequential abstract machine, followed by optimization techniques which may lead to an implementation with acceptable performance.

However, if one is to specify a special-purpose processor for the execution of the target language, one need not inherit the restrictions of the execution model specified for a general-purpose processing environment. Thus, if one were to consider the SAM for FCP as the execution model of the FCP Processor, one is limited by its inherent sequentiality. In Figure 1 we symbolically imply that the FCP SAM may be adequate for implementation on a general-purpose sequential machine. However, for a special-purpose processor, an alternative execution model should be considered; one that suites the underlying architecture. We propose modifications to the SAM and define the FCP Processor Execution Model (EM) executable on a single processor which consists of concurrent communicating processing elements or functional units. The transformation of the SAM into the FCP Processor EM is simple and intuitive. Many features of the EM may also apply to other concurrent logic programming languages.

2.1 FCP Processor Organization

The FCP Processor consists of a set of functional units or processing elements for the single-reduction execution of FCP. Figure 2 depicts the three-layer hierarchal structure of the FCP Processor organization. The

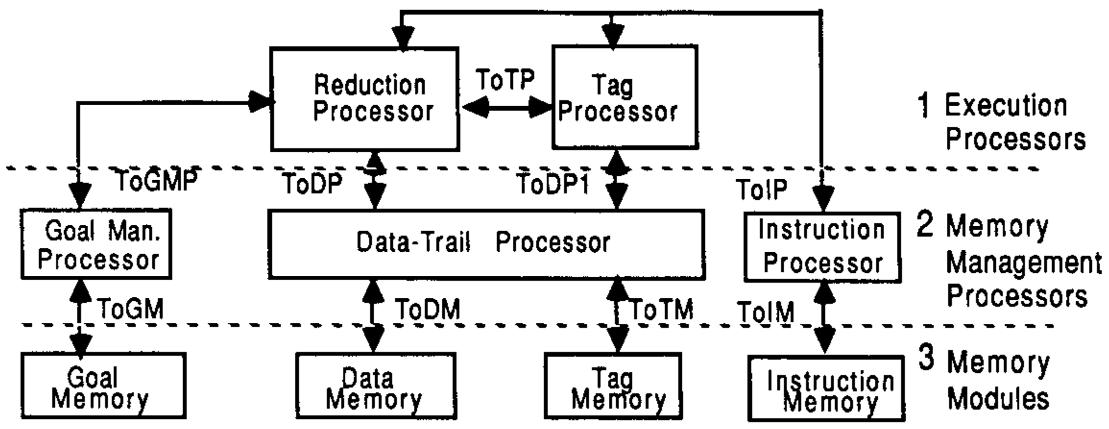

Figure 2: The Organization of the FCP Processor

top of the hierarchy contains the tightly coupled execution processors, the second level consists of specialized memory management processors and the third level contains the special-purpose memory modules. The following are the FCP Processor concurrent functional units:

1. Execution Processors: Reduction (RP) and Tag (TP) Processors;
2. Memory Management Processors: Goal Management (GMP), Data-Trail (DTP) and Instruction (IP) Processors;
3. Memory Modules: Goal, Data, Tag and Instruction Memory;

Program 1 describes the organization of the FCP Processor, using FCP. The parallel execution of communicating processors is described using concurrent FCP goals. The communication protocol between the functional units is modelled using shared variables as communication channels and read-only variable annotation *(?)* to model the direction of communication and synchronization. The predicate *fcp_processor* has four argumets representing the initial values of the four memory modules. It spawns goals corresponding to the parallel functional units: *rp, tp, gmp, ip, dtp,* and memory modules: *instruction_memory, data_memory, tag_memory, goal_memory.* The *rp* predicate contains arguments *ToGMP, ToDTP, ToIP, ToTP* which correspond to the inter-processor communication channels. The predicate *gmp* shares a read-only version of the variable *ToGMP* with the *rp*. This means that the GMP waits for a message from the RP.

```
fcp_processor(I_IM,I_DM,I_TM,I_GM) :-
    rp(ToGMP,ToDTP,ToIP,ToTP?),
    tp(ToIP?,ToTP,ToDTP1),
    gmp(ToGMP?,ToGM),
    ip(ToIP?,ToIM),
    dtp(ToDTP?,ToDTP1?,ToDM,ToTM),
    instruction_memory(I_IM,ToIM?),
    data_memory(I_DM,ToDM?),
    tag_memory(I_TM,ToTM?),
    goal_memory(I_GM,ToGM?).
```

Program 1: FCP Processor Organization

The next subsection describes the FCP Processor functional units.

2.2 FCP Processor Functional Description

Reduction Processor

The RP performs goal reduction using the FCP read-only unification algorithm [Shapiro83]. The RP requests and manipulates three types of operands: *Goals*, *Instructions* and *Data*. A goal is an abstract data structure that contains a pointer to a sequence of instructions (called the goal's program counter) and a set of pointers to the goal's arguments. The RP reduces goals by executing the instructions denoted by the current goal's program counter. The instructions are requested and received from the IP. The RP executes instructions, thus requesting data manipulation from the DTP or goal management from the GMP.

Tag Processor

FCP incorporates polymorphic operations on primitive data types. Unless some architectural support is provided, our observations indicate that tag processing consumes a significant part of program execution and compiled code size. In the FCP Processor tags are separated form data objects and stored in the Tag Memory. A separate processor-memory path for tags enables concurrent tag access and the TP performs concurrent tag processing. The instruction requested by the RP from the IP contains two fields indicating concurrent operations for the RP and TP. The RP does not execute the next instruction until both the RP and the TP are finished executing the current instruction.

Goal Management Processor

In FCP and in other committed-choice concurrent logic programming languages, spawning and halting of concurrent goals represents the main control mechanism. This is analogous to the procedure call and return in conventional languages. We propose a novel mechanism for the management of concurrent goals on a single processor. Using a Goal Cache structure described in Section 4, the GMP implements efficient goal switching, goal creation, suspension, activation and termination.

Instruction Processor

The IP fetchs instructions from the IM when requested by the RP. Additional features could be added to the functionality of the IP: prefetching, instruction caching etc.

Data-Trail Processor

Goal reduction requires the unification of goal head arguments with arguments of a matching clause in the FCP program, followed by the successful evaluation of clause guards. These two steps are jointly referred to as a *clause-try*. If there are several matching clauses, a clause-try is attempted for each clause, until a successful clause-try is found, otherwise the goal fails. During a clause-try, variables in the clause may have values assigned to them. If a clause-try fails, memory must be restored to the state preceeding the clause-try, so that another clause-try is attempted. This is referred to as *shallow backtracking*. If the clause-try is successful, the bindings performed during the clause-try are available to the body of the clause.

To restore a previous memory state, variable assignments performed during a clause-try are *trailed*. Trailing consists of saving the address and value of the trailed variable in a *trail* structure. Restoring the previous memory state is then performed by reading the old variable values from the trail and writing them to memory.

In the FCP Processor, a novel trailing mechanism is performed by the Data-Trail Processor. It records the changes made in memory during a clause-try, restores them upon a clause failure and makes them permanent upon clause commit. The data trailing algorithm is performed concurrently with RP execution.

Memory Modules

The FCP Processor memory modules service the *read(Address, Value)* and *write(Address, Value)* memory requests from the GMP, DTP and IP. The address space of a FCP program is partitioned into four areas: *Code*, *Goal*, *Memory* and *Tag Memory*. The compiled program is stored in the Code Memory which is accessed and managed by the IP. The Goal Memory is used for storing goal structures. It is accessed and managed only by the GMP. Tags are stored in the Tag Memory whereas the Data Memory is used for storing objects like lists, variables, tuples, integers etc. It is managed by the DTP.

2.3 FCP Processor Execution

The FCP Processor maintains a consistent hierarchy of processor execution. The RP requests goals, instructions and data values from the GMP, IP and DTP respectively. Each of these processing units acts as a cache of objects requested by the RP. If there is a cache hit, the RP will be serviced immediately, and if there is a cache miss, the corresponding cache processor will perform the necessary memory request. There are no explicit requests by the RP to the memory modules.

The RP executes FCP programs by requesting goals from the GMP. The received goal becomes RP's current goal. The RP requests from the IP the instruction corresponding to the goal's program counter. The received instruction contains instruction fields that determine the operation of the RP and TP. The execution of the instruction manipulates structures in the Data and Tag Memory by requesting changes to the memory state from the DTP. All read or write requests to the DTP fetch or store the appropriate data and tag pair of values.

Some of the instructions that the RP receives from the IP require the management of the current goal. These instructions are executed by the GMP while the RP continues executing the next instruction. Therefore, the RP continuously reduces goals while the GMP manages the goal structures in an overlapped mode. The communication protocol between the RP and the GMP allows a single GMP operation to be requested at a time. Using FCP, this protocol is modelled in the following way: The RP sends the GMP an operation together with a *Busy* variable, shared by the two processors. When the GMP receives the message, it performs the requested operation, and upon completion, assigns it the value *done*. Meanwhile, the RP continues to execute instructions. Instructions that are executed only by the RP, do not depend on the status of the Busy flag, and thus they ignore it. When a GMP instruction is encountered in the RP, the Busy status flag is first tested by attempting to assign the *done* value. If the GMP finished its previous operation, this shared variable is bound to *done*, thus leading to successful unification which results in continuous RP execution. If the GMP is still busy, the RP will suspend waiting for the GMP to finish the current instruction.

3 Reduction Processor

The RP is a special-purpose processor for the reduction of FCP goals. It is the main processing unit in the FCP Processor. The purpose of the remaining functional units is to enhance the performance of the RP.

3.1 Goal Reduction

Goal reduction in FCP consists of:

1. selecting a clause whose head unifies with the goal and whose guard succeeds,
2. commiting to this clause, and
3. spawning the body of the committed clause.

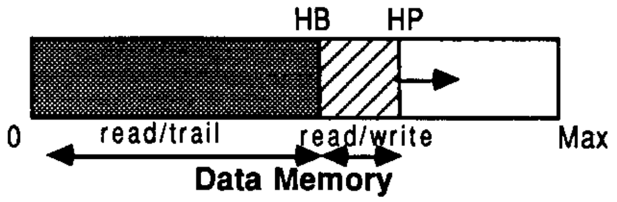

Figure 3: Read, Write and Trail Memory Access

3.1.1 Clause Selection

In FCP, given a goal and a set of matching clauses, any clause may be selected for goal reduction, conditioned that the clause-head arguments unify with the goal arguments, and the clause-guard evaluates to true. Selection of a clause may consist of a sequence of unsuccessful *clause-tries*. If a clause-try creates structures in the Data Memory, then, if the clause-try fails, these structures become invalid. If the clause-try succeeds, the structures are valid. In other words, during clause selection, all bindings in the Data Memory are temporary until a successful guard evaluation marks the end of the selection phase, and the beginning of the commit phase, which makes the bindings permanent.

Data Trailing

Structures in the Data Memory area allocated as a heap. Let HP denote the top of the heap and HB the top of the heap prior to the last clause-try. During a clause-try, new data structures are allocated between HB and HP. If a clause-try succeeds, the HB becomes HP and a new clause-try may begin. Otherwise, the newly allocated structures are discarded by assigning HB to HP. But, this does not undo any changes performed in memory below HB. That is, memory assignments below HB must be *trailed.*

In the FCP Processor, data trailing is performed in the DTP concurrently with RP execution. The RP distinguishes memory assignments below HB by comparing the address of a memory access with the value of HB. Thus, there are three different types of memory accesses: *read(Address)*, *write(Address, Value)* and *trail(Address, Value)*. The *write* memory access is performed when $HB < Address < HP$ and *trail* is performed when $Address < HB$. This is shown in Figure 3.

Goal Suspension

In FCP, during clause-head unification the current goal suspends if an attempt is made to bind a read-only variable. The goal suspension mechanism is implemented by the GMP. The RP supports goal suspension by using two suspension tables, ST1 and ST2. During the clause-try, a suspension table (ST) is used to store the variables that the current goal may suspend on. If the outcome is anything other than *suspend*, the suspension table is cleared and reused. Otherwise, it is used by the GMP to implement the goal suspension mechanism, while the RP continues with program execution using the alternative suspension table.

3.1.2 Clause Commit

Following a successful clause-try is the commit phase of goal reduction. Committing to a clause means that all the assignments made during the clause-try become permanent. What remains to be done at commit time is to verify if assignments were made to variables that had goals suspended on them. If there were goals suspended on these variables, they are given to the GMP to schedule for execution. This is done using a goal *Wakeup Queue* (WQ).

In section 6 we briefly describe the goal suspension mechanism performed by the GMP. The relevant part to the clause-commit algorithm is that goal suspension is implemented by assigning the read-only variable a pointer to the suspended goal. Therefore, at commit time, the trailed value of a read-only variable assignment points to the goal to be woken up. This pointer is given to the GMP via the WQ. The RP repeats this for each trailed entry and terminates the commit phase by enqueueing a special marker onto the WQ.

Goal wake-up may be a time consuming operation. Enabling the GMP to perform this operation con-

$p(...)$:- $g_1,...,g_n$ | true.

$p(...)$:- $g_1,...,g_m$ | $q(...)$.

$p(...)$:- $g_1,...,g_k$ | $q_1(...),q_2(...),...,q_n(...)$.

Figure 4: FCP Clause Types

currently with RP execution is a significant architectural feature. The number goals typically activated at commit time is of the order: zero, one or two.

3.1.3 Spawning the Clause Body

After the commit phase the RP spawns goals corresponding to the body of the committed clause. For each goal in the clause body, it creates the goal arguments and places pointers to them into a goal structure. After spawning all of the goals in the clause body, the current goal is considered reduced.

3.2 RP Instruction Execution

The RP reduces goals supplied by the GMP. Goal reduction consists of executing instructions corresponding to compiled goals stored in the IM and managed by the IP. In FCP, a goal corresponds to a set of Guarded Horn Clauses which may take one of the three forms shown in Figure 4.

If we symbolically represent goal-head unification with the *get* instruction and the formation of goal arguments with the *put* instruction, the clause types are compiled to the sequence of instructions shown in Figure 5. The *commit* instruction denotes the commit phase of goal reduction, the *halt* instruction terminates the current goal, and the *spawn* instruction tells the GMP to schedule the newly created goal. The first clause type represents goal termination, the second goal iteration and the third shows goal p iterating on goal q_1 while spawning goals $q_2, q_3, ..., q_n$.

The RP executes all the *get* and *put* type of instructions, but not the instructions that deal with the actual manipulation of the goal structures (highlighted). These instructions are executed by the GMP. In Figure 6 we describe RP instruction execution. Instructions denoted as *gmpop* are sent to the GMP to be executed while the RP continues to fetch and execute instructions.

4 Goal Management Processor

The purpose of the GMP is to reduce the effective time spent performing goal management operations. By effective time, we imply the time as seen by the RP, which also includes the overhead of communication and synchronization. This is achieved in the following way:

- The FCP Processor execution model allows the execution of GMP operations concurrently with goal reduction in the RP.
- The GMP is designed for the efficient execution of goal management operations using a Goal Cache (GC).

4.1 Overlapped GMP Execution

The following features of the FCP Processor execution model enable the concurrent execution of the GMP:

get	*unify the head of clause p*
commit	*wake up suspended goals*
halt	*terminate goal*

(a) Compiling clause type p().

get	*unify the head of clause p*
commit	*wake up suspended goals*
put q	*form the arguments of goal q*
execute q	*reduce goal q*

(a) Compiling clause type p() :- q().

get	*unify the head of clause p*
commit	*wake up suspended goals*
put q_1	*form the arguments for goal* q_1
put q_2	*form the arguments for goal* q_2
spawn q_2	*spawn goal* q_2
...	
put q_n	*form the arguments for goal* q_n
spawn q_n	*spawn goal* q_n
execute q_1	*iterate on goal* q_1

(c) Compiling clause p() :- q_1(), q_2(), ... , q_n().

Figure 5: Compiling FCP clause types

```
rp(ToGMP,ToIP,ToDTP,ToTP) :-
        rp(done,done,ToGMP,ToIP,ToDTP,ToTP,initial_state(...)).

rp(done,done,[G|Gs?],[I|Is?],[D|Ds?],[T|Ts?],State):-
        fetch(I,State,Instruction),
        execute(Instruction?,done,Done,done,Busy,G,D,T,State?,NewState),
        rp(Done?,Busy,Gs,Is,Ds,Ts,NewState?).
rp(done,done,[close],[close][close],[close],_state()).

execute(store(A,T,V),D,D,B,B,noop,noop,write(A,T,V),set(T),state(..PC..),state(..PC1..):-
        PC1 := PC + 1 | true.
execute(gmpop,Done,Done,Busy,Busy1,gmpop(Busy1),noop,noop,state(..PC..),state(..PC1..)):-
        PC1 := PC + 1, ground(Busy) | true.

fetch(read(PC,Instruction),state(...,PC,...),Instruction).
```

Figure 6: RP Instruction Execution Cycle

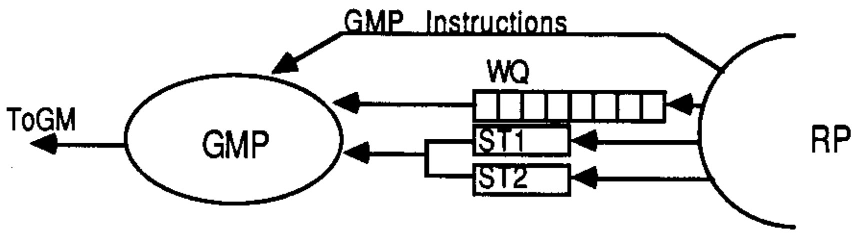

Figure 7: Enabling Concurrent GMP Execution

```
gmp(ToGMP,ToGM):-
    gmp(ToGMP,done,ToGM,state(initial)).

gmp([gmp_op(Busy)|ToGMP],done,ToGM,State):-
    step1(ToGM,ToGM1?,State?,State1,done/Done1),
    step2(ToGM1,NewGM?,State1?,NewState,Done1?/Busy),
    gmp(ToGMP?,Busy?,NewGM,NewState?).
```

Figure 8: GMP Instruction Execution

- The goal management operations *Halt*, *Spawn*, *Suspend* and *Commit* are identified as separate high-level instructions in the RP instruction set.
- Their execution manipulates goal structures which are accessed only by the GMP.
- The nature of the data dependencies between the GMP instructions and the following RP instructions is well defined. For the *Halt* and *Spawn* operations there is no data dependency, thus they may execute concurrently without any special support. The concurrent execution of the *Suspend* instruction is enabled by adding a second Suspension Table and the *Commit* operation is supported by adding the Wake-up Queue.

In Figure 7 we symbolically represent the GMP and its interface to the Goal Memory and the RP. The GMP receives instructions from the RP via *ToGMP* and communicates with the Goal Memory using the *ToGM* channel. The two suspension tables ST1 and ST2 are alternatively used for implementing the goal suspension algorithm and the WQ enqueues goal pointers received from the RP denoting goals to be scheduled for reduction.

In Figure 8 the GMP concurrent execution is described using FCP. The GMP receives from the RP a message containing the operation and the shared *Busy* flag used for synchronization. When the GMP completes the requested operation, it will set the shared flag to *done*. If the RP decodes another GMP operation before the GMP finishes the current operation, the RP suspends.

4.2 Efficient Goal Management

The GMP performs efficient goal management by manipulating a Goal Cache (GC) used for storing FCP goals. The GC consists of a set of N goal windows marked *active*, *ready*, *free* or *spawn*. The *active* window contains the currently executing goal, *ready* windows contain goals that are ready to be reduced, *free* windows are vacant and *spawn* denotes the window used by the RP to spawn a new goal. The windows that are addressable by the RP are the *active* and the *spawn* windows. The GC is shown in Figure 9.

The GMP manipulates the GC so that it always has at least one free window used for fast spawning, and one prefetched goal for fast goal scheduling. A goal is spawned in the cache by marking it *ready*. If the GC overflows, a goal is selected in the cache and moved to a queue in the Goal Memory. A goal is terminated in the cache by marking it *free*. Successive goal termination may cause *underflow*. In this case the GMP dequeues a goal from the queue in the Goal Memory, and stores it in the GC.

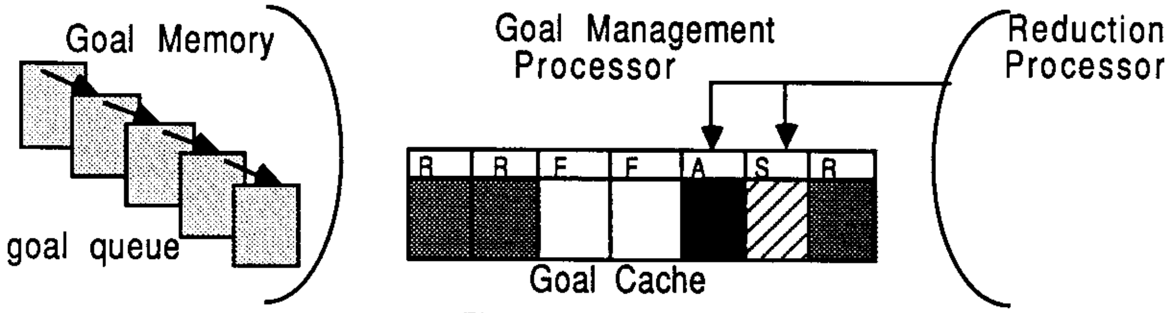

Figure 9: GMP Goal Cache

When a goal suspends, it is moved to the goal queue in memory, and when goals are woken up, they are added to the end of the goal queue. All operations in the GC consist of changing the status of the goal windows and manipulating goal window pointers.

5 Data-Trail Processor

In FCP, given a goal query, any clause from a set of matching clauses may be selected for a clause-try. The outcome is *success* or *failure*. A number of unsuccessful clause-tries may be attempted before a successful clause-try is found. We consider as overhead all the time spent executing clause-tries that lead to failure. To reduce their effect on performance one may:

- Improve the clause selection strategy;
- Provide architectural support for data trailing;

Improving the clause selection strategy is possible if more processing is performed at compile time, and if the user provides some "inside information" regarding program behaviour. Because of data dependencies and lack of knowledge about program behaviour, clause-try failures are unavoidable. Therefore, given that the clause selection strategy is not ideal, we are concerned with reducing the overhead of data trailing in the FCP Processor.

There are three parts to the clause-try overhead: processing necessary to determine clause-try failure, saving a previous memory state and restoring a previous memory state when failure occurs. The processing that is performed to determine whether the selected clause succeeds is unavoidable, given that the compiler or user were not able to provide the program execution with such information.

The DTP provides architectural support for both saving and restoring the memory state prior to and after a clause-try failure. We refer to the following DTP policy as *Delayed Binding*:

All assignments performed during a clause-try are delayed until the outcome of the clause-try is known. If the outcome is successful, the bindings become permanent otherwise they are cleared.

5.1 Data-Trail Cache Algorithm

The DTP is a data cache that implements the Delayed Binding policy. In this paper we do not discuss the details of the data cache organization or mapping policy. We only emphasize those features that relate to the data trailing property. For simplicity, we assume that the data cache is fully associative with a write-back memory policy. The data cache stores the address, the value and the status of the cached elements. The status may be: *Empty, Clean, Dirty* or *Trailed.* An entry labeled *Empty* is vacant. If the status is *Clean* the cached element is identical to the corresponding value in the DM. A *Dirty* status indicates that the stored

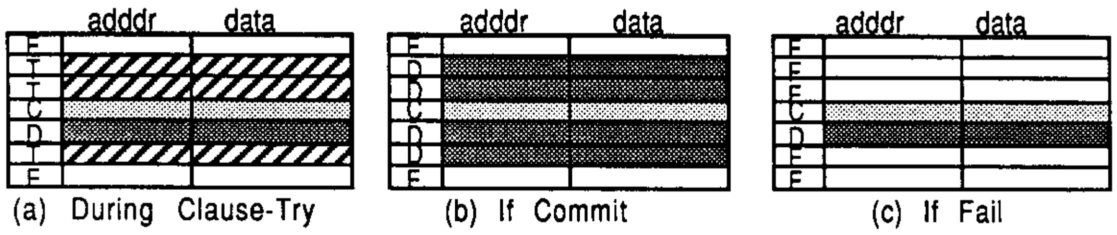

Figure 10: Data-Trail Cache Policy: Delayed Binding

value in the cache differs from the value in memory, and the *Trailed* status indicates that the value in the cache is temporary.

The DTP receives read, write and trail memory requests. The read and the write requests are treated in the conventional way. Upon receiving the trail memory request, the DTP performs the following operation:

- **trail(Address,Value):** If there is a cache hit, the following cases may occur depending on the status of the cached element. If it is *Clean*, the new value is stored in the cache and marked as *Trailed.* If it is *Dirty* the cached value is written back to the DM and the new value is stored in the cache and marked as *Trailed.* If it was already *Trailed,* the new value is written in the cache and remains Trailed.

  In case of a cache miss, the cache replacement policy vacates an entry that is not *Trailed,* writes the new value in the cache and marks it as *Trailed.*

The following operations are performed when the control signals *fail* and *commit* determine clause-try failure or success.

- **fail:** All Trailed entries are marked Free.
- **commit:** All Trailed entries are marked Dirty.

In Figure 10a we show the DTP cache during a clause-try. The elements marked as *T* are being trailed. In Figures 10a and 10b we show the contents of the cache after a clause success and failure.

Therefore, the proposed data cache policy implements the Delayed Binding approach to data trailing by keeping the trailed values in the cache and either commiting them to Dirty upon clause-try success, or resetting them upon clause-try failure. One should note that Trailed values are never replaced by the cache replacement policy. Furthermore, trailed values are accessible during the clause-try even before they commit or fail. Trailed values that commit to Dirty remain in the data cache as valid cache entries.

6 Conclusion

In this paper we propose a FCP Processor execution model which exhibits intra-processor concurrency. The execution model is derived by modifying the SAM for FCP. We also describe the organization of the special-purpose FCP Processor architecture which consists of the following concurrent functional units: Reduction Processor, Tag Processor, Goal Management Processor, Instruction Processor and Data-Trail Processor. The Reduction and Tag Processors execute instructions received from the Instruction Processor, reducing goals supplied by the Goal Management Processor and sending data requests to the Data-Trail Processor. We propose architectural support for goal management in the form of a Goal Cache. We define the Delayed Binding data cache policy used by the Data-Trail Processor to reduce the overhead of saving and restoring a previous memory state upon clause-try failures. The FCP Processor architecture and execution model are described using FCP. We are currently investigating the attainable performance of the FCP Processor.

References

[Mier85] C. Mierowsky et al., "The Design and Implementation of Flat Concurrent Prolog", Department of Applied Mathematics, Weizmann Institute of Science, Israel. CS85-09, July 1985.

[Houri86] A. Houri and E. Shapiro, "The Sequential Abstract Machine for Flat Concurrent Prolog", Department of Applied Mathematics, Weizmann Institute of Science, Israel, CS86-20, July 1986.

[Klig87] S. Kliger, "Towards a Native-Code Compiler for for Flat Concurrent Prolog", Department of Applied Mathematics, Weizmann Institute of Science, Israel. Master Thesis, 1987.

[Shapiro83] E. Shapiro, "A Subset of Concurrent Prolog and Its Interpreter", Department of Applied Mathematics, Weizmann Institute of Science, Israel. February 1983.

[Taylor87] S. Taylor et. al. "FCP: Initial Studies of Parallel Performance", Department of Applied Mathematics, Weizmann Institute of Science, Israel, CS87-19, October 1987.

[Warren83] D. H.D. Warren, "An Abstract Prolog Instruction Set", SRI International, Project 4776, Technical Note 309, October 1983.

[Pat82] D. A. Patterson and C. H. Sequin, "A VLSI RISC", IEEE Computer, Vol 15, No. 9, pp.8-21, Sept.1982.

Acknowledgement

We acknowledge the contributions of Prof. Milos Ercegovac and Prof. Tomas Lang of the UCLA Computer Scince Department. They have significantly contributed to the research direction reported in this paper. We are also grateful to Avshalom Houri from the Weizmann Institute of Science for numerous discussions and to Miquel Huguet from UCLA for useful comments reviewing this publication.