Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

A PROPOSAL FOR THE FLAT CONCURRENT PROLOG PROCESSOR ARCHITECTURE

Leon Alkalaj

April 1988
CSD-880035

A Proposal for the Flat Concurrent Prolog Processor Architecture

Leon Alkalaj
University of California, Los Angeles
A Thesis Proposal

May 3, 1988

ABSTRACT

We propose to define, simulate and evaluate a special-purpose, high-performance processor architecture for the execution of Flat Concurrent Prolog (FCP). Our objective is to achieve at least two orders of magnitude performance increase compared to current implementations on general-purpose processors.

We propose a novel FCP Processor Execution Model that defines the overlapped execution of Goal Management operations and Goal Reduction. It is derived by partitioning the FCP Sequential Abstract Machine into concurrently executing units.

We propose a special-purpose FCP Processor organization to support the internal concurrency defined in the FCP Execution Model. The FCP Processor architecture consists of the following concurrent functional units or processing elements: Reduction Processor, Tag Processor, Goal Management Processor, Instruction Processor and Data-Trail Processor. We identify two features in the execution of FCP: *Goal Management* and *Data Trailing*, which are supported at the functional unit level by the Goal Management Processor and Data-Trail Processor respectively.

The Goal Management Processor performs the efficient management of concurrent FCP goals reduced by the Reduction Processor. Goals are stored in a special-purpose Goal Cache. The Data-Trail Processor implements a Data Cache with a novel cache management algorithm called *Delayed Binding* which supports shallow backtracking.

We specify the FCP Processor Execution Model and architecture using FCP. The specification is part of a working simulator. We find the concurrency implicitly defined in FCP suitable for modeling the execution of concurrent functional units.

We are currently investigating the attainable performance of the FCP Processor architecture. We propose a novel methodology to evaluate the contribution of each concurrent functional unit in the FCP Processor.

1 Introduction

Flat Concurrent Prolog (FCP) is a concurrent logic programming language proposed by Shapiro [25]. It defines non-deterministic goal reduction and data-flow synchronization using annotated shared variables. An FCP program is written as a set of guarded Horn clauses. The term *Flat* denotes that only simple test primitives are allowed in the clause guard.

The design and implementation of FCP using a sequential interpreter is proposed in [Mier] and a Sequential Abstract Machine, similar to the abstract machine for Prolog define by Warren [35], is proposed in [15]. Both implementations exhibit performance inferior to implementations of conventional languages. Improved results are reported in [18] using target host compilation techniques. Nevertheless, the current implementations of FCP exhibit inadequate performance. 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 the performance is to define a concurrent execution model and distribute the execution of an FCP program on multiple processors. Parallel Inference Machines for the execution of Concurrent Logic Programming languages have been proposed in [16], [22]. Similarly, preliminary results of implementing FCP on a Hypercube architecture are reported in [30].

Besides *inter-processor* concurrency, further performance improvements are possible by exploiting the 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 goal of our research is to investigate the prospects of a special-purpose, high-performance, single-processor architecture for the execution of FCP. Our objective is to achieve at least two orders of magnitude performance improvements compared to the currently available implementations of FCP. We divide our research effort into the following three phases:

- Propose a special-purpose FCP Execution Model and processor architecture.
- Simulate the functionality of the FCP Processor.
- Evaluate the performance of the proposed architecture.

Execution Model and Processor Architecture

In the first phase we propose an execution model for FCP and a corresponding FCP Processor architecture, designed for high-performance. We propose an *overlapped* execution model that exhibits *internal parallelism*. It is derived by modifying the Sequential Abstract Machine for FCP and overlaps the execution of Goal Management and Goal Reduction. We also define a suitable FCP Processor architecture designed with the following characteristics:

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

The following two features that distinguish FCP from other logic programming languages deserve special attention:

- 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 first feature is supported at the functional level by a separate Goal Management Processor using a special-purpose Goal Cache. The motivation for introducing a separate Goal Cache for the manipulation of concurrent FCP goals is analogous to the use of multiple windows to support nested procedure calls in procedure-oriented architectures [Patterson]. The second feature is also supported by a separate functional unit called the Data-Trail Processor. It manipulates a special-purpose Data Cache which implements a cache policy we termed *Delayed Binding*.

FCP Processor Functional Simulation

The second phase of our work consists of functionally specifying the FCP Processor architecture using a suitable simulation language. The main reason for this is to verify the functionality of the proposed execution algorithms. We use FCP to functionally specify the processor architecture. We find the concurrency implicitly defined in FCP suitable for simulating concurrent, communicating functional units.

FCP Processor Evaluation

In the final phase of our work we determine the performance of the FCP Processor. We evaluate the FCP Processor architecture and compare its performance to other existing implementations. We propose a unique methodology to determine the contribution of each concurrent functional unit in the FCP Processor. We find this an important aspect of our research since it will evaluate the design choices made.

1.1 Research Proposal Organization

This report is organized in the following way: In Section 2 we describe the scope of the proposed research. We identify the main research problems and our approach to resolve them.

In Section 3 we describe how our research compares to other related work. As of yet, we are unaware of any published results in the area of processor architectures for Flat Concurrent Prolog. However, we do not restrict the scope of related work to just the implementation of FCP since the implementations of other logic programming languages have provided us with useful experience.

In Section 4 we propose and evaluate solutions to problems identified in Section 2. We describe the state of the current research and the future work.

Finally, in Section 5 we summarize our research goals and the current state of the research. We conclude by specifying what results we expect to obtain.

2 Scope of Proposed Research

The goal of our research is to define, simulate and evaluate a high-speed single-processor architecture for the execution of FCP. In this section we define the scope of the proposed research. To understand issues concerning the implementation of FCP, we find it necessary to first discuss how FCP relates to other logic programming languages. In the following subsection we give a simple abstraction of the FCP programming language and discuss features that distinguish it from other logic programming languages. After identifying FCP in the spectrum of logic programming languages, we discuss the following focal points of our research.

- FCP Execution Model
- FCP Processor Architecture:
  - Organization
  - Specification
  - Evaluation
- Goal Management in FCP
- Data Trailing in FCP

For each issue we describe the problem, our approach to solving it and the scope of research.

2.1 FCP and Logic Programming

Even though logic programming evolved from research in the field of theorem proving, its applications today range over diverse areas of computer science such as data-base systems [10], operating systems [27] and problem solving. Furthermore, the declarative semantics of first-order predicate calculus logic has been proposed as a means for representing Declarative Knowledge in articifical intelligence [12]. For a concise review of the foundations of logic programming, refer to the book by Lloyd [20].

Horn Clause Form

The use of logic as a computer programming language was first proposed by Kowalski [19] and Colmeraurer [5]. The first practical logic programming language, Prolog, is based on a subset of logic expressed in the Horn clause form. A Horn clause is an implication of the form

$$A(t_1, ..., t_n) \leftarrow B_1 and B_2 and ... and B_m., \; n, m \geq 0.$$

$A(t_1, ..., t_n)$ is a distinguished positive literal referred to as the clause-head and the B_i's literals are the body goals. All free variables in the clause are implicitly universally quantified. The clause form is read as: "A is true if all the B_i" are true.

From the logic point of view, there is no constraint imposed on the ordering of goals B_i. This form of concurrency is referred to as AND-parallelism. Moreover, given a goal query B_i, any one of

the matching clauses from the set of clauses in the logic program may be chosen for goal reduction. This form of concurrency is called OR-parallelism.

Sequential Logic Programming Languages: Prolog is an example of a sequential logic programming language. The goals B_i are reduced sequentially from left to right and the clauses are selected in textual order. If a goal fails, the control execution backtracks to an alternative path in the AND/OR search space. Alternative sequential search strategies are possible.

Parallel Logic Programming Languages: Define the same semantics as the sequential logic languages, only attempt the search in parallel. Parallel execution models have been proposed in [36], [6], [2].

Guarded Horn Clause Form

A gaurded Horn clause has the following form:

$$A(t_1, ..., t_n) \leftarrow G_1, ..., G_j \mid B_1 and B_2 and ... B_m., \; j, n, m, \geq 0.$$

The G_js are clause-guards and | denotes the *commit* operator. A guarded Horn clause is read as: "*A is true if the conditions G_j are all satisfied and all B_i evaluate to true.* The | operator separates the clause-guard from the clause-body. The implication of the clause-guard is that once the guards evaluate to true, the control execution "commits" to the selected clause and cannot backtrack.

Committed Choice Languages, (CCL): The logic programming languages that use the guarded Horn clause form are generally referred to as Committed Choice Languages since at some point in the search path the control *commits* and cannot backtrack. Examples are Concurrent Prolog (CP) [25], Parlog [4] and Guarded Horn Clause (GHC) [34].

Flat Committed Choice Languages, (FCCL): Define a subset of CCL that allow only simple test primitives defined in the clause-guard. FCCL have been introduced to circumvent implementation problems encountered in CCL. Examples of FCCL are: Flat Concurrent Prolog (FCP), Flat Parlog (FP) and Flat Guarded Horn Clause (FGHC).

2.1.1 Flat Concurrent Prolog (FCP): A Simple Abstraction

In [26], FCP is described as a *process-oriented* language. An analogy is drawn between the non-deterministic scheduling of processes in a multi-processing system and the non-deterministic scheduling of goals in FCP. The concurrent goals in FCP communicate via shared variables. The data-flow synchronization is implemented by annotating shared variables with the *read-only* construct "?". For example, the shared variable X in goal $P(X)$ annotated as read-only, $X?$, in goal $Q(X?)$, implies that goal P is the *producer* of the value for X and goal Q is the *consumer*. Goal Q will *suspend* if it attempts to assign a value to the shared variable X. The suspended goal Q is *woken-up* when goal P produces a value for the shared variable X.

2.2 FCP Execution Model

It is common practice in the design and implementation of high-level languages to first specify an abstract machine implementation and a corresponding machine instruction set. Abstract ma-

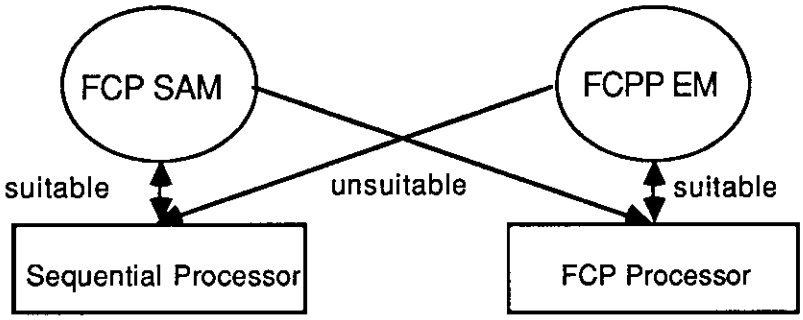


Figure 1: FCP Execution Model

chines are defined as machine independent implementations used for fast prototyping and simple compilation. However, to execute the high-level language, the proposed abstract machine is then emulated on the host, physical machine using a suitable *emulation language*. At this point, the user is obviously concerned with the efficiency of program execution, which may raise questions regarding the "machine independence" of the proposed execution model.

Therefore, even though abstract machines represent machine independent execution models, they are ultimately judged by their performance, which translates to issues of "machine suitability". Since the currently available execution environments are commonly sequential, one readily defines a sequential abstract machine followed by optimization techniques which may lead to an implementation with acceptable performance.

Current implementations of FCP are based on the Sequential Abstract Machine (SAM) defined in [15]. This execution model is similar to the one defined by Warren for the Prolog programming language [35]. Both implementations on off-the-shelf processors report similar performance results [15].

Since our goal is to define a high-speed, special-purpose processor and since the SAM for FCP is designed for execution in a general-purpose environment, there is no reason for us to use an execution model that was constrained by architectural features that may not be part of the FCP Processor architecture. Rather than building the FCP Processor and then finding a suitable "machine independent abstract machine", we first propose a special-purpose execution model with features for higher performance, and then define a suitable processor architecture for the defined execution model.

Therefore, our approach is to actually emphasize the suitability of the execution model and processor architecture, rather than claiming "abstractness of implementation". We do not refer to the FCP Processor execution model as an abstract machine, but rather as an execution model suitable for the special-purpose FCP Processor. In Figure 1 we symbolically represent the unsuitability of the SAM for the FCP Processor and the FCP Execution Model for an off-the-shelf sequential processor. As the FCP SAM may be suitable for implementation on sequential machines, so we expect the FCP Execution Model to execute efficiently on the FCP Processor architecture.

2.3 FCP Processor Architecture

The following issues regarding the FCP Processor are within the scope of our research:

- Processor organization;
- Functional specification and simulation of processor execution;
- Processor Evaluation using defined metrics;

Since the question is often raised, we find it necessary to explicitly state that the implementation of the FCP Processor into a VLSI design is not within the scope of the proposed research. Similarly, implementation details will be addressed only within the context of the scope of research. Therefore, our approach is to explore the space of possible architectural solutions so that the appropriate design decisions are made depending on the state-of-the-art of VLSI processor implementations.

FCP Processor Organization

The FCP Processor organization depends on the design of the execution model discussed earlier in this section. Since we are not concerned with the generality of the proposed architecture (which may be addressed later), we explicitly tailor the FCP Processor to execute the execution model in an efficient way. Therefore, the FCP Processor organization reflects the structure of the FCP Processor execution model. We propose to identify the main components of the processor architecture, define their functionality and interface to other units.

FCP Processor Simulation

The functionality of the FCP Processor components should be specified using an appropriate specification language. Our concern is not to determine the suitability of the simulation language but to use the existing environment for fast prototyping. Using the functional specification of the FCP Processor architecture we propose to simulate the execution of the FCP Processor. We are concerned with features such as simulation speed and the class of programs that execute on the simulated architecture.

FCP Processor Evaluation

After proposing and simulating the FCP Processor we plan to verify the following two claims set throughout our research.

- The FCP Processor and Execution Model are **suitable** for the execution of FCP.
- The FCP Processor is a **high-speed** processor architecture for the execution of FCP.

We distinguish the two claims in the following way. The first claim relates to the design of the FCP EM and the suitability of the corresponding processor architecture. Within this claim we are not concerned with the physical performance of the FCP Processor but rather the use of resources and other defined metrics. That is, by verifying the first claim we are evaluating the FCP Processor organization.

The second claim states that the proposed FCP Processor is superior to other existing implementations of FCP. Particularly, we refer to two different implementations of the SAM for FCP on a conventional processor. One implementation compiles FCP to the SAM instruction set which is

Goal Management Operations	Boot Logix	Compiler1	Compiler2	Compiler3
Creations	1126	81251	8838	231165
Suspensions	1328	75039	8654	158008
Activations	1328	74910	8654	156865
Switches	3	285	15	1018
Reductions	2399	272 360	29205	736748
Terminations	1126	81122	8830	1 230020
CPU time (ms)	1980	101900	10940	271680
LIPS	1211.62	2672.82	2669	2711.82

Table 1: GMP Operations

interpreted at the abstract machine emulation level, and the other uses a host machine compiler [18].

To verify both claims we need to determine a precise and meaningful methodology and define the evaluation metrics. The performance of logic programming languages is commonly measured using a very vague and deceptive metrics that counts the number of Logical Inferences Per Second (LIPS). Given the fact that logical inferences may vary in size and complexity, quoting the number of LIPS is meaningless unless one also cites the evaluated program. Selecting meaningful benchmarks for the evaluation of the FCP Processor architecture is also within the scope of our research.

2.4 Goal Management in FCP

A significant part of our research is devoted to the study of *Goal Management* in FCP. The management of logical goals in FCP represents the main control mechanism analogous to procedure call/return used in procedure-oriented languages. The efficient management of goals in FCP is essential to achieve higher performance. A typical FCP program spawns numerous reducible goals that may be scheduled non-deterministically. Furthermore, FCP goals communicate and synchronize execution using shared communication channels. This implies that goals may suspend waiting for messages, and are activated upon their arrival.

In Table 1, we show the number of goal management operations performed during various executions of the Logix operating system [27] written in FCP. The first column shows Logix being booted and exited while the other columns were obtained by having Logix run the FCP Compiler (written in FCP) compiling portions of the FCP Processor Simulator (also written in FCP).

From the above example, one can note that for each goal reduction, at least two goal management operations are performed. The goal management operations may be quite complex especially goal suspension and goal activation. In [9], it is reported that two of the most expensive operations in the implementation of Flat Committed Choice Languages, FCCL, are related to goal management. We take this observation even further and claim that goal management may in fact pose a bottleneck in FCP execution, if not architecturally supported.

In our research we propose a method to reduce the effect of goal management on overall performance. We are also concerned with evaluating alternative goal management strategies that may algorithmically improve the execution of FCP programs.

2.5 Data Trailing in FCP

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 preceding 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.

Trailing in the SAM

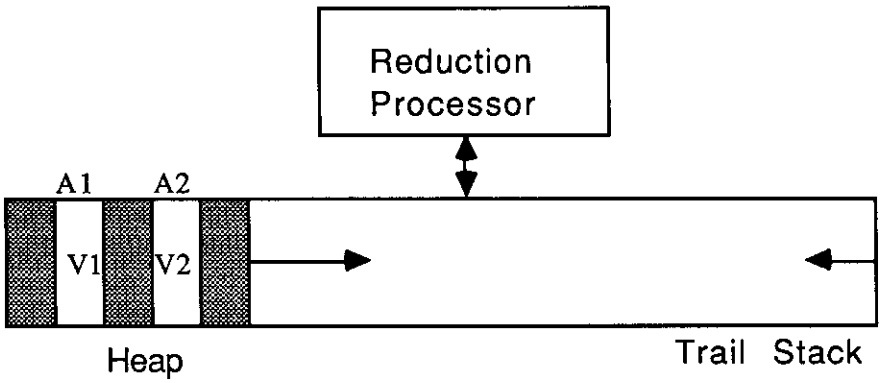
To restore a previous memory state, variable assignments performed during a clause-try are *trailed*. Trailing consists of saving the address and value of the trailed variable in a *trail* structure. Restoring the previous memory state is then performed by reading the old variable values from the trail and writing them to memory. In Figure 2 we show a Reduction Processor for the execution of FCP programs performing a clause-try in memory. Figure 2a shows the state of memory and *trail* prior to the clause-try. During the clause-try, the RP stores the address and the old value of the variable in the *trail*. This is shown in Figure 2b. These values are read and restored in case of a clause-try failure.

We propose a novel trailing mechanism to reduce the overhead of saving and restoring a memory state upon a clause-try failure. This mechanism is defined as an integral part of the FCP Processor architecture.

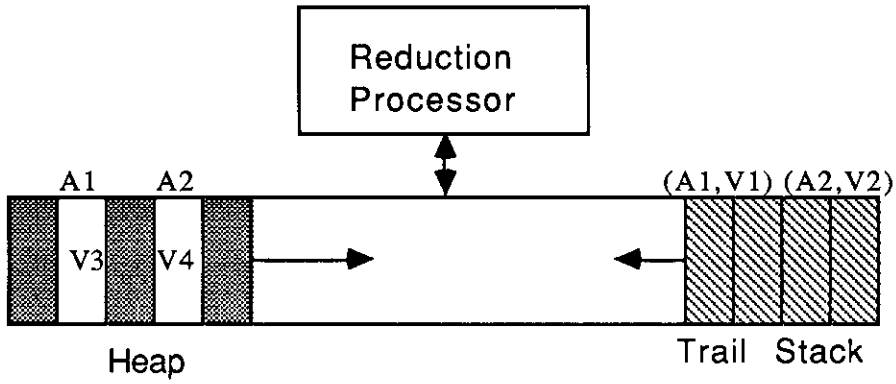
2.6 Summary of Expected Contributions

We expect the following contributions from the proposed research:

- A novel execution model for FCP.
- A high-performance FCP Processor architecture.
- An analysis of suitable Goal Management Strategies.
- A novel Data-Trail Algorithm.
- A working FCP Processor simulator.
- A set of meaningful evaluation benchmarks.
- A methodology for the evaluation of the FCP Processor architecture.
- A performance comparison between the FCP Processor and other existing implementations.
- A processor architectural model suitable for other FCCL.



a) Memory, prior to clause-try



b) During Clause-try

Figure 2. Trailing During a Clause-Try

3 Related Work

Architectural support for the execution of logic programming languages is an active area of research [33]. We distinguish two main research directions. First, a number of parallel machines for the execution of logic programming languages, called *Parallel Inference Machines*, have been proposed [3], [7], [11], [13], [16], [17], [22], [23], [28]. The main issues that these architectures are concerned with are the type of parallelism to support (OR, AND or perhaps AND/OR) and how to implement an efficient multi-processor execution model.

A comparably smaller research effort is directed towards the design of high-performance sequential processors for logic programming languages, called *Sequential Inference Machines* [8], [14], [29], [31]. All of these machines are designed for the execution of Prolog and they are all based on the sequential execution model proposed by Warren [35].

Our research effort consists of designing a special-purpose processor for FCP [1], which is quite different from Prolog. As it is explained in Section 2, FCP belongs to the FCCL class of languages. Sine FCP does not allow backtracking, there is no need to use a stack to store the goal control records, alternative choice-points or data environments, as is the case for Prolog. Furthermore, Prolog requires a stack of trailing environments, whereas trailing is only one level deep in FCP.

In [32], Tick investigates one aspect of Prolog architectures, namely, the processor memory-referencing behavior and the use of specialized buffers and caches. Stack and choice-point buffer hit ratios as well as traffic ratios are reported. We also suggest and investigate the use of specialized caching functions to increase the processor-memory bandwidth and thus processor performance. The functions we define are specialized for the execution of FCP. Furthermore, we define a processor execution model which enables the overlapped execution of the specialized caching units.

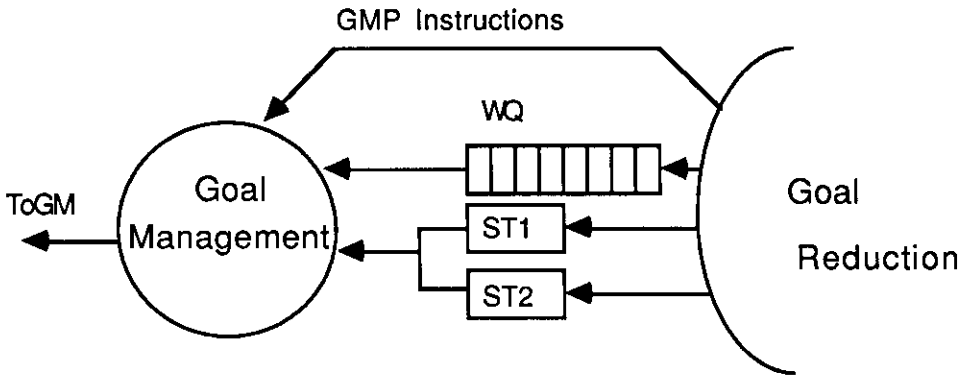


Figure 3: FCP PEM: Overlapped Goal Management and Goal Reduction

4 FCP Processor Execution Model

We propose a special-purpose FCP Processor Execution Model (EM) that uses **internal parallelism** to achieve high performance. It enables the overlapped execution of Goal Reduction and Goal Management. The FCP EM is derived by modifying the SAM for FCP in the following way:

- All goal structures are contained within a separate Goal Memory (GM) area;
- A Goal Heap Pointer (GHP) and a Goal Free List (GFL) are used for memory allocation and garbage collection in the Goal Memory;
- There are two, (instead of one in the SAM), *Suspension Tables* (ST) that enable the overlapped execution of the goal suspension mechanism. While a goal is being suspended using one ST, the other is used for overlapped goal reduction.
- A *Wake-up Queue* (WQ) is used to enable the concurrent execution of goal wake-up and continuous goal reduction.

In Figure 3 we symbolically denote the overlapped execution of Goal Management and Goal Reduction. We allow a single goal management operation to be overlapped at a time. An alternative is to queue goal management requests. We have considered this approach and observed significant implementation complexity that would not yield a gain in performance. In the following subsection we describe the overlapped execution of goal management and goal reduction.

4.1 Overlapped Goal Management and Goal Reduction

In this subsection, we describe how the FCP Processor EM defines the overlapped execution of goal management operations and goal reduction execution. We do so by first describing goal reduction as it is performed in the SAM. Following this we identify the main goal management operations and describe how we propose to support their overlapped execution.

Goal Reduction

Goal reduction in FCP consists of the following three steps:

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

1. Selecting a clause whose head unifies with the goal and whose guard succeeds.
2. Commiting to the selected clause.
3. Spawning the body of the committed clause.

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-guards evaluate to true. Selecting a clause may consist of a sequence of unsuccessful *clause-tries*, that is, clause-tries that have failed. If there were assignments made during a failed clause-try, they are restored to the values prior to the clause-try.

Following a successful clause-try is the commit phase of goal reduction. Committing to a clause implies that all bindings created 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 so, these goals must be scheduled for execution, that is woken-up.

After clause commitment the non-empty body of the selected clause is spawned. This is performed by allocating goal structures, forming the goal arguments and scheduling the goal for execution. If the clause body is empty, the goal terminates and another reducible goal is scheduled for execution.

Goal Management Operations

To understand how the goal management operations are overlapped with goal reduction execution, consider the three types of FCP clauses shown in Figure 4. The first clause is called the *halting clause*, the second *iterating clause* and the third is referred to as the *spawning clause* type. Let us symbolically represent the operations performed during goal-head unification with the *get* abstract instruction and the formation of goal arguments with the *put* instruction. We represent the three possible outcomes of goal-head unification with the following abstract instructions:

- *commit*
- *suspend*
- *fail*

Following successful goal-head unification the *commit* instruction denotes the waking-up of suspended goals that received data during the clause-try. If, during goal head-unification, an attempt is made to bind read-only variables, the current goal suspends. This is denoted by the *suspend*

instruction. Finally, if goal-head unification does not succeed, the *fail* instruction implies that an alternative clause must be considered for goal reduction.

After successful goal-head unification and commitment, the goals in the body of the clause are spawned using the *put* and *spawn* instructions. If there are no goals in the body, the *halt* instruction denotes the termination of the goal. If there is only one goal in the clause body, the *iterate* instruction implies *tail recursion optimization* by reusing the old goal instead of spawning a new goal. This feature is merely an optimization and the *iterate* instruction may be replaced by *spawn* instruction.

In Figure 5 we show the three FCP clause types compiled using the abstract instructions: *get, put, commit, fail, suspend, spawn, iterate* and *halt.* We highlight the abstract instructions *switch, spawn, suspend, halt* and *commit* that manipulate FCP goals. In the SAM these instructions are executed sequentially. In the FCP Processor Execution Model, we propose the overlapped, or concurrent, execution of goal management operations and goal reduction instructions.

Overlapped Execution

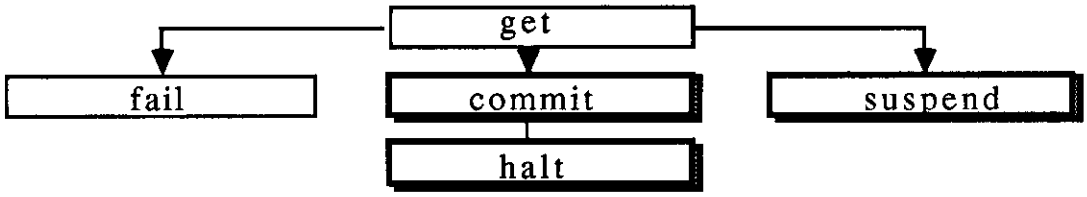
In Figure 6a we show that there is no data dependency between the goal management instruction *halt* and the *get* instruction of the following goal reduction. Therefore, these two operations could overlap. Similarly, in Figure 6b we show that there is no data dependency between the *put* instruction during the spawning of the clause-body, and the *spawn* instruction of the previously "put" goal. That is, the successive putting and spawning of goals in the clause-body could overlap.

However, allowing the overlapped execution of the *suspend* and *commit* instructions is not as simple. During the clause-try, the *get* instruction uses a Suspension Table defined in the SAM, to store the addresses of variables that the goal may suspend on. If the outcome of a clause-try is indeed "suspend", the *suspend* instruction accesses the Suspension Table and implements the goal suspension mechanism. Therefore, the *suspend* instruction can not overlap execution with the *get* instruction of the next goal reduction, since they would both access the same structure. We show this conflict of access in Figure 7a.

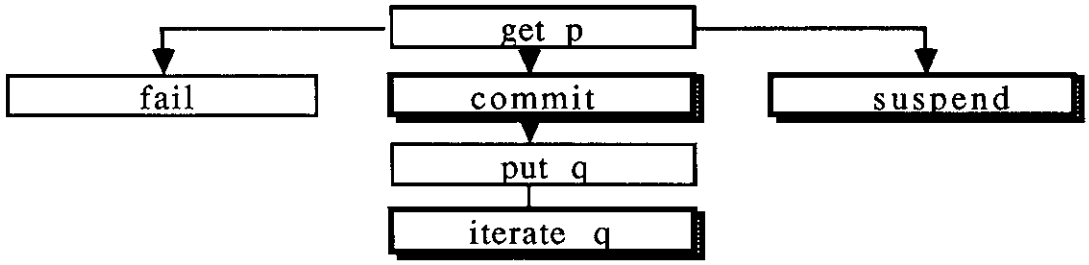
We propose the use of two suspension tables in an alternating manner. When the *suspend* instruction is encountered, goal reduction continues with the free Suspension Table, while the goal suspension is implemented concurrently using the old Suspension Table. This is shown in Figure 7b.

The *commit* instruction consists of waking up goals and scheduling them for execution. In Figure 8a we show two shared variables that have goals suspended on them. This is denoted by storing the goal pointer in the variable location. Goal reduction may not continue since it may overwrite the pointers to the goals that should be woken up.

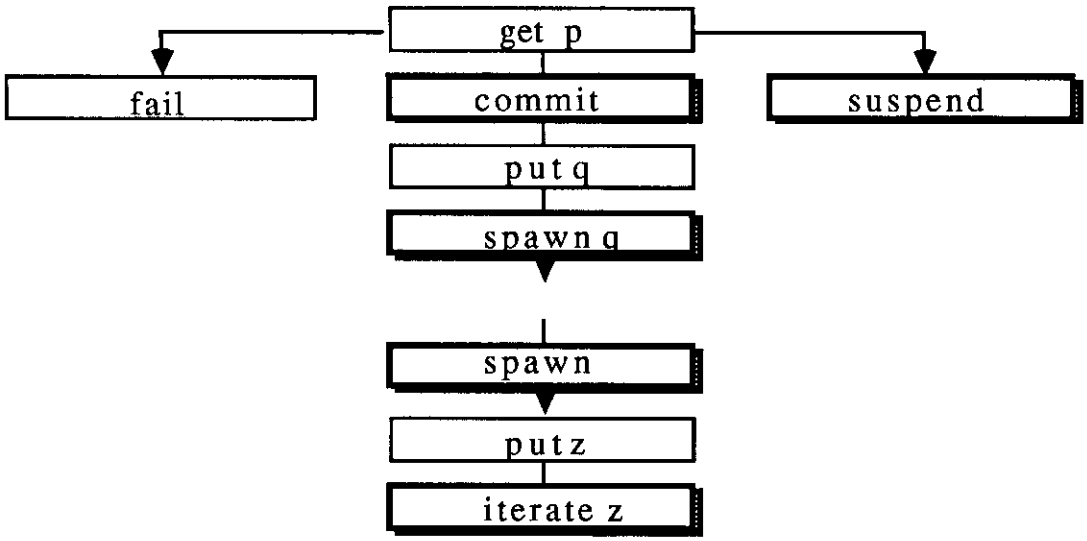
We propose a Wake-up Queue (WQ) to store the pointers to the goals that should be woken up. The commit instruction then consists of enqueueing the goal pointers onto the WQ, after which goal reduction may proceed while the scheduling of goals in performed in an overlapped mode. This is shown in Figure 8b.



Compiling the Halting clause p(...) :- g(...) | true.



Compiling the Iterating clause p(...) :- g(...) | q(...)..



Compiling the Spawning clause p(...) :- g(...) | q(...), ... , z(...).

Figure 5: Compiling FCP clause types

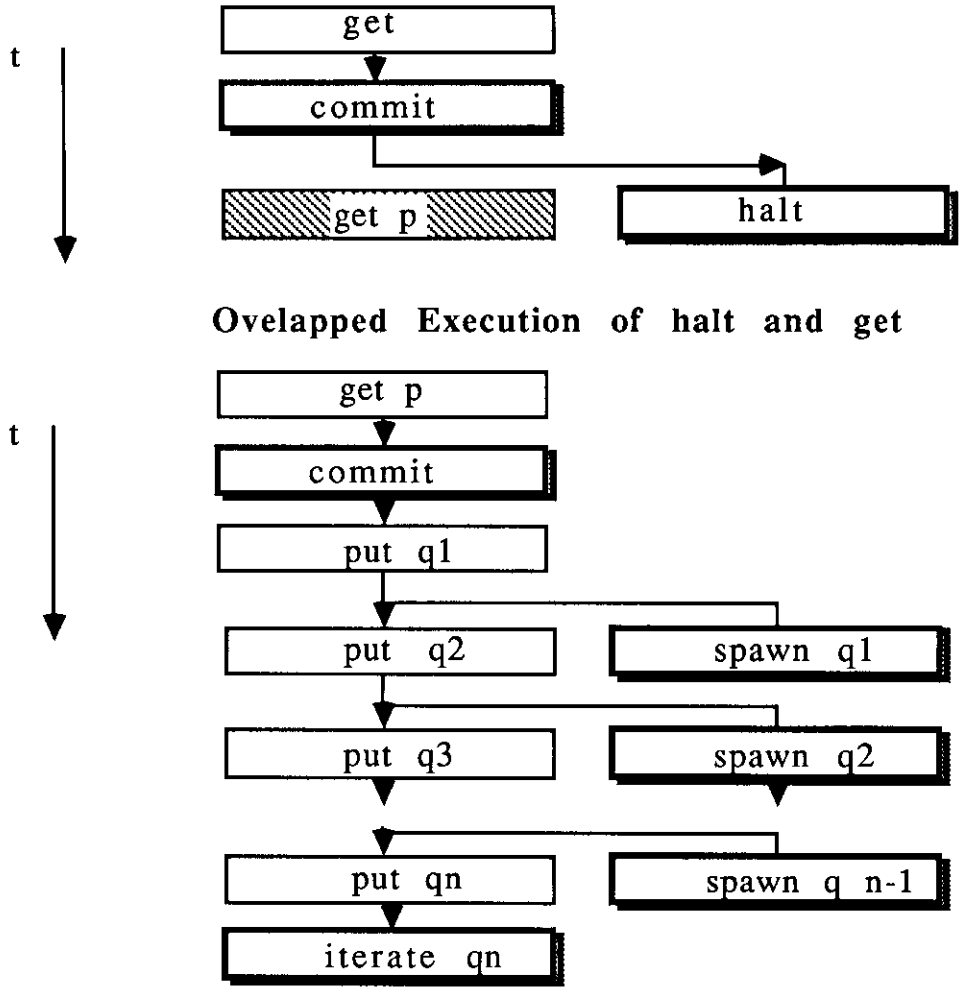


Figure 6: Overlapped get/halt and put/spawn

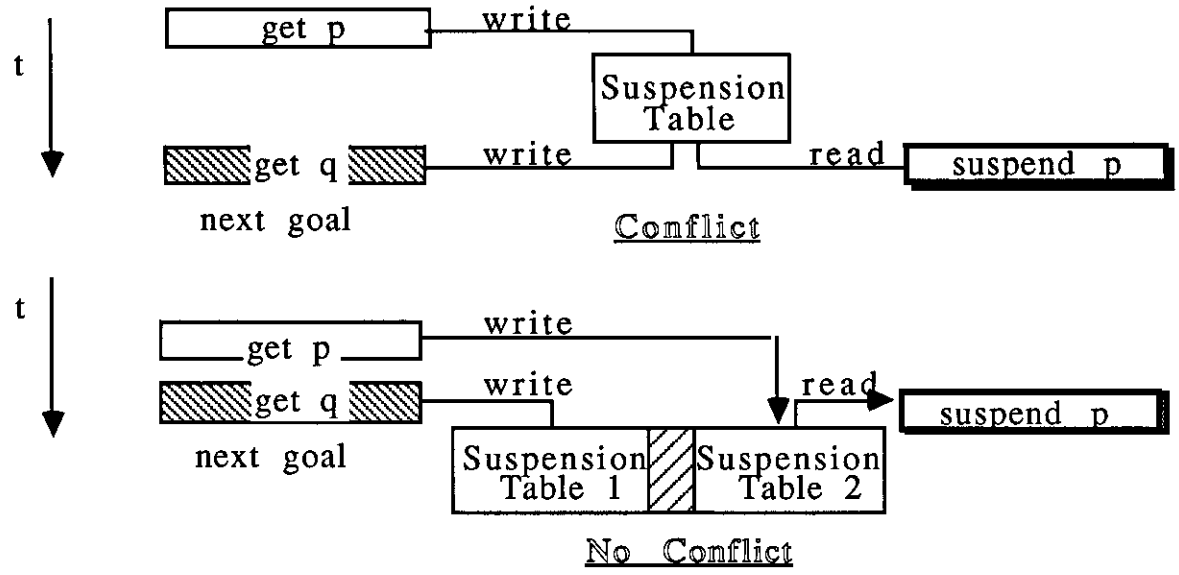


Figure 7: Enabling the Overlapped Execution of Suspend

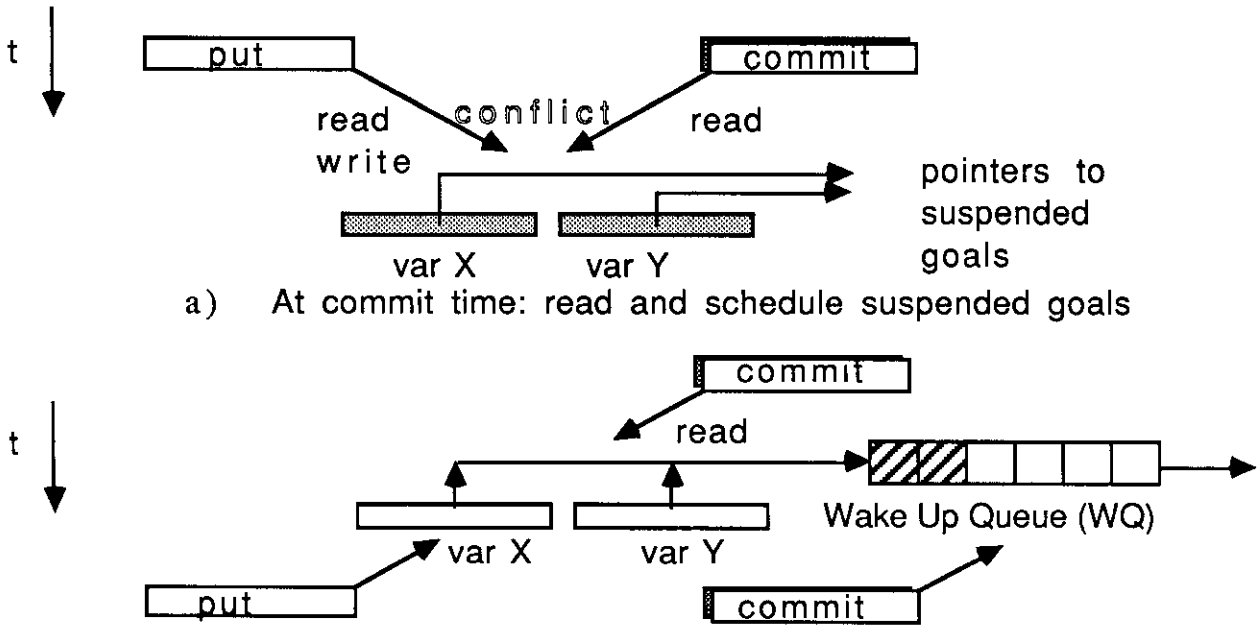


Figure 8: Enabling the Overlapped Execution of Commit

4.2 Discussion, Status and Future Work

We have proposed a novel FCP Processor Execution Model which defines the concurrent execution of goal management and goal reduction. This is achieved due to the following observations:

- The goal management operations *Halt*, *Spawn*, *Suspend* and *Commit* are identified as separate high-level instructions in the abstract instruction set.
- Their execution manipulates goal structures stored in a separate Goal Memory.
- The nature of the data dependencies between the GMP instructions and the goal reduction 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.
- The *Commit* operation is supported by adding the Wake-up Queue.

We have implemented a working simulator of the overlapped execution of goal management and goal reduction operations in the FCP Processor. In our future work we intend to evaluate the performance gain due to the overlapped execution model.

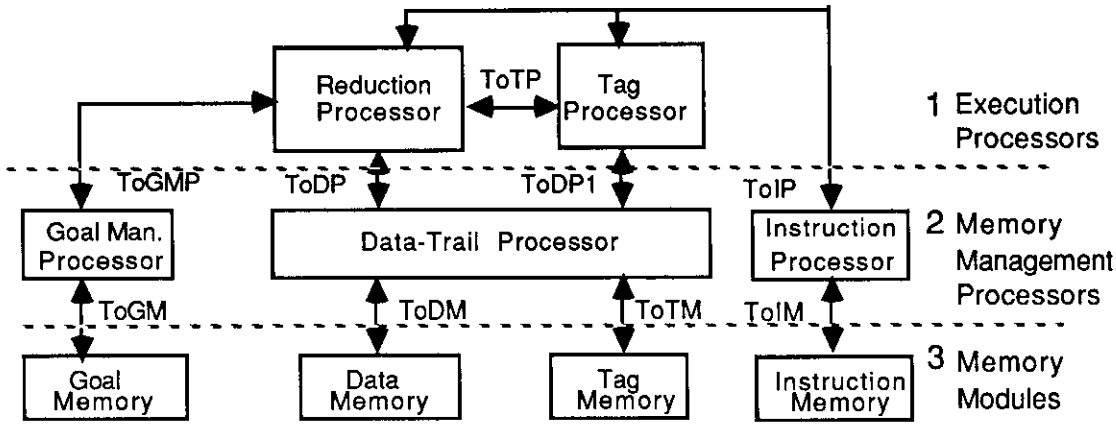


Figure 9: The Organization of the FCP Processor

5 FCP Processor Architecture

In this section we describe the FCP Processor organization, the processor simulator and evaluation methodology. The FCP Processor simulator is written in FCP. We find the concurrency of FCP goals implicitly defined in the FCP execution model suitable for describing the concurrent execution of functional units in the FCP Processor. The communication and synchronization between functional units is modeled using the FCP read-only variables. In this section we use portions of the simulator where appropriate to describe the FCP Processor architecture.

5.1 FCP Processor Organization

The FCP Processor consists of a set of functional units or processing elements for the single-reduction execution of FCP. Figure 9 depicts the three-layer hierarchal structure of the FCP Processor organization. The 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;

In Figure 10 we describe 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 modeled 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 arguments 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

```
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?).
```

Figure 10: Program 1: FCP Processor Organization

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.

The next subsection describes the FCP Processor functional units.

5.2 FCP Processor Functional Description

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. The RP performs goal reduction using the FCP read-only unification algorithm [25]. 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

The data trailing algorithm is performed by the DTP, concurrently with RP execution. The DTP is a special-purpose Data Cache with a cache policy that supports the efficient saving and restoring of a previous memory state.

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.

5.2.1 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

```
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 11: RP Instruction Execution Cycle

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 modeled 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.

In Figure 11 we describe the RP execution using FCP. The *fetch* predicate requests an instruction from the IP, which is received by the *execute* predicate. We show the execution of two types of instructions: *store(Address, Tag, Value)* is an example of an instruction executed only by the RP and the instruction *gmpop* is given to the GMP for execution. The *Busy* variable denotes the shared flag used for synchronization.

5.3 A Proposed Methodology for FCP Processor Evaluation

Let P_{fcpp} denote the FCP Processor, where $fcpp = \{rp, gmp, tp, dtp, ip\}$. We denote each concurrent processor in the FCP Processor as P_i where $i \in fcpp$. The proposed methodology consists of three phases.

Phase 1

Let T_i^I, $i \in fcpp - rp = \{gmp, tp, dtp, ip\}$, be the time to execute an FCP program on P_{fcpp} with the operations of processor P_i taking zero time. Let T_i^E, $i \in fcpp - rp = \{gmp, tp, dtp, ip\}$, denote the time to execute the same FCP program on the FCP Processor with processor P_i removed and its functionality emulated by the remaining architecture denoted as $P_{(fcpp-i)}$. We then represent the maximum or *ideal* performance benefit due to processor P_i as

$$S_i^I = T_i^E / T_i^I \tag{1}$$

where $i \in fcpp - rp = \{gmp, tp, dtp, ip\}$.

Phase 2

Let T_i^R, $i \in fcpp - rp = \{gmp, tp, dtp, ip\}$, represent the *realistic* (non zero) time it takes to execute an FCP program on P_{fcpp} and T_i^E the time to run the same program with processor P_i emulated. The *realistic* performance benefit due to processor P_i is:

$$S_i^R = T_i^E / T_i^R \tag{2}$$

where $i \in fcpp - rp = \{gmp, tp, dtp, ip\}$.

Phase 3

Let S_i^I represent the set of values for the ideal and S_i^R realistic performance speedups due to each processor P_i where $i \in fcpp - rp = \{gmp, tp, dtp, ip\}$. The FCP Processor evaluation factor measuring performance improvement due to the concurrent execution of FCP on communicating processors is modeled as:

$$E^I = \prod_i S_i^I = \prod_i (T_i^E / T_i^I) \tag{3}$$

$$E^R = \prod_i S_i^R = \prod_i (T_i^E / T_i^R) \tag{4}$$

where $i \in fcpp - rp = \{gmp, tp, dtp, ip\}$.

5.3.1 Discussion

A discussion of the proposed methodology for the evaluation of the FCPP architecture is appropriate at this point. We have suggested a method for evaluating the performance benefit of functional units by successively removing a single unit and emulating its functionality on the remaining architecture. As we have previously objected to the approach of comparing emulated level language implementation with other machine level implementations, the following question must be answered:

- *What if the architecture denoted as P_{fcpp-i} is particularly unsuitable for the emulation of processor P_i?*

Or, for example, what if the operations of the GMP are particularly time consuming when emulated on the $P_{fcpp-gmp}$ architecture? Moreover, if this were the case, one would readily modify the $P_{fcpp-gmp}$ architecture to emulate the GMP more efficiently.

To avoid the *circulus viciosus* situation of first removing a processor P_i and then creating it for the sake of efficient emulation on P_{fcpp-i}, we find it necessary to verify whether indeed the P_{fcpp-i} enables efficient emulation within its execution environment. To further illustrate this problem, consider a processor architecture where one of the units is specialized to perform trigonometric operations. If we then emulate these operations on the remaining architecture, we are concerned that the emulation is performed efficiently. If this is true, the emulation approach is evaluated, otherwise, additional features are considered to further optimize emulation execution.

The following question regarding the proposed methodology should also be discussed:

- *How does it model domains of specialized functional units that are not disjunctive?*

For example, it is possible that emulating processor P_i on P_{fcpp-i} executes part of the domain of processor $P_j \in P_{fcpp-i}$ where $i \neq j$. For example, if a significant part of goal management consists of tag processing, the emulated GMP operations would benefit considerably from the concurrent Tag Processor, TP.

In the given example, we are concerned with the GMP operations that are disjunctive with the operations of the $P_{fcpp-gmp}$ architecture. The tag operations would lead to a more efficient emulation of the GMP and diminish its contribution to the performance evaluation of the P_{fcpp}. Therefore, we consider such cases to be correctly modeled by the proposed evaluation methodology.

5.4 Future Work

We propose to extend the currently available FCP simulator to obtain performance evaluation parameters. For benchmarks, we propose to use programs developed for FCP Processor simulation or perhaps parts of the Logix operating system written in FCP. Furthermore, we propose to use programs which are particular for the FCP programming language. For example, a termination detection meta-interpreter using the short-circuit technique.

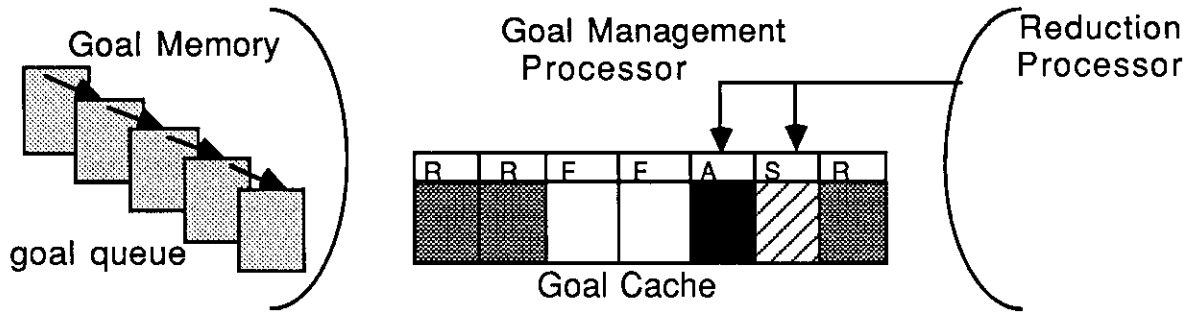


Figure 12: GMP Goal Cache

6 Goal Management In The FCP Processor

A separate functional unit, the Goal Management Processor GMP, is dedicated to the management of FCP goals. 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).

In Section 4 we have discussed in detail the proposed overlapped execution model, and now we show the efficient implementation of goal management operations.

6.1 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 12.

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*. The successive spawning may cause *overflow*. 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.

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.

6.2 Discussion, Status and Future Work

We propose a novel policy for the efficient implementation of concurrent goal management operations using a Goal Cache. The GC policy has the following properties:

- It enables the fast spawning, halting and scheduling of goals by manipulating the goal status bits.
- Supports the concurrent execution of goal suspension and goal wake-up.
- Buffers the frequent halting and spawning of FCP goals.
- Captures the locality of inter-goal communication.

Currently, a working simulator of the GMP is available. We propose to evaluate the proposed GC policy and investigate other cache strategies. We are concerned to determine how parameters vary with the size of the GC and to investigate the complexity of the GC implementation. Furthermore, we are concerned to evaluate the load balance between the GMP and the RP. The load balance will determine to which degree one needs to optimize the execution of the GMP. For example, if the GMP creates a bottleneck for RP execution, it may be effective to further increase the speed of goal management operations with hardware support.

7 Data Trailing In The FCP 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 clause-try success commits to the body of the selected clause whereas a clause-try failure leads to an alternative clause-try attempt.

Trailing data assignments in the FCP Processor is necessary since one does not know in advance which clause-try will succeed. This non-determinacy of clause selection is a necessary and powerful feature of logic programming. It implies that 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;

Clause Selection

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 behavior. For example, if the implicit clause selection strategy is determined by the textual order of clauses in a program, the user may order the clauses according to his/her knowledge of the probability of clause failure or success. Alternatively, in [Kliger], input mode declarations are used to aid the clause selection strategy. In this case, the clause selection is compiled to a sequence of argument dereferencing and testing, until a single clause is selected. In cases where this knowledge is not available to the user or the program i.e. it is data dependent, a clause-try may lead to failure. In these situations the overhead of data trailing is incurred.

Program analysis and transformations should be used extensively to reduce the overhead of clause selection in FCP, since they lead to a compile time instead of run-time overhead. Nevertheless, because of data dependencies and lack of knowledge about program behavior, 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. Improving the clause selection strategy is beyond the scope of this work, but it should be combined with the data trailing mechanism proposed here.

Clause-try Overhead

There are three parts to the clause-try overhead:

- Processing necessary to determine clause-try failure;
- Saving a previous memory state;
- Restoring a previous memory state when failure occurs;

The processing that is performed in order to determine whether the selected clause may succeed, is unavoidable, given that the compiler or user were not able to provide the program execution with

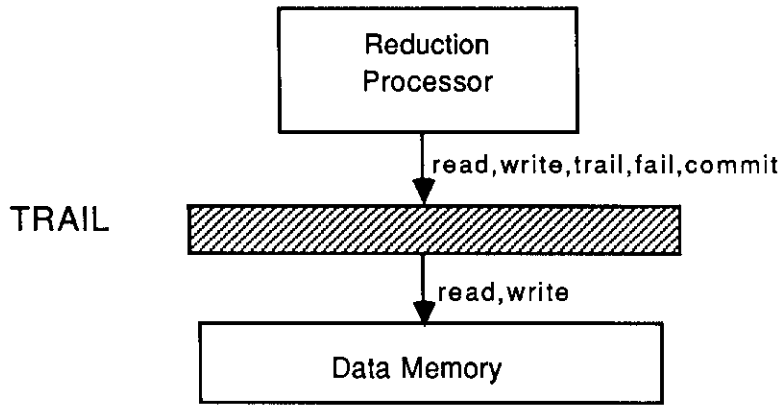


Figure 13: DTP: Design Approach

such information. This execution consists of operations such as argument unification, argument testing and general data processing.

In addition to clause-try processing, the state of memory before the clause-try must be saved and restored in case of clause failure. Saving a memory state consists of saving the address and old value of bindings performed during a clause-try. Restoring a previous state consists of reading the stored addresses and reassigning the old values into memory.

DTP Objective and Design Approach

The purpose of the Data Trail Processor described in this section is to provide hardware support for both saving and restoring the memory state prior to and after a clause-try failure.

This is achieved by delaying all the bindings into the Data Memory until the outcome of the clause-try is known. In this case, if the outcome is successful, the bindings are included as part of the memory hierarchy, otherwise, the bindings are ignored. During the clause-try, memory assignments are kept in a structure used for storing temporary bindings which are accessible to the RP. If a clause-try fails, the temporary bindings are ignored and if the clause-try succeeds the bindings become valid.

Therefore, the design approach allows the RP to "see" the the Data Memory, the way it would look if the clause-try were to succeed. This design approach is depicted in figure 13.

7.1 Data-Trail Cache Policy

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.

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

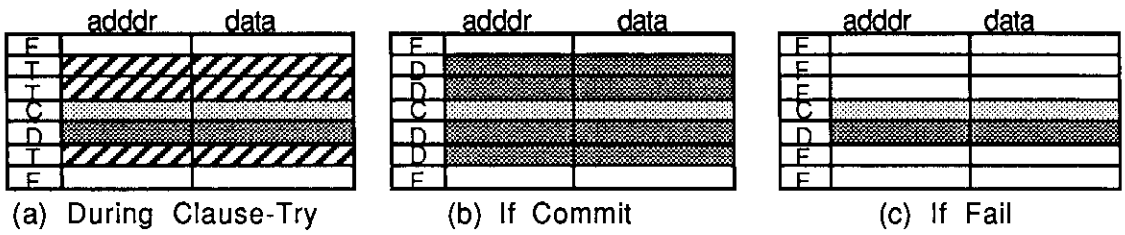


Figure 14: Data-Trail Cache Policy: Delayed Binding

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 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 14a we show the DTP cache during a clause-try. The elements marked as T are being trailed. In Figures 14a and 14b we show the contents of the cache after a clause success and failure. The Data Trail Cache Policy is described in 15.

Therefore, the proposed data cache policy implements the Delayed Binding approach to data trailing by keeping the trailed values in the cache and either committing 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

One should note that there is no need to replace any of the trailed values that have committed to dirty. They remain valid entries in the cache and may be replaced using the general cache replacement policy. With the implementation proposed in this report, the RP does not need to perform the trailing or the undoing of the trail.

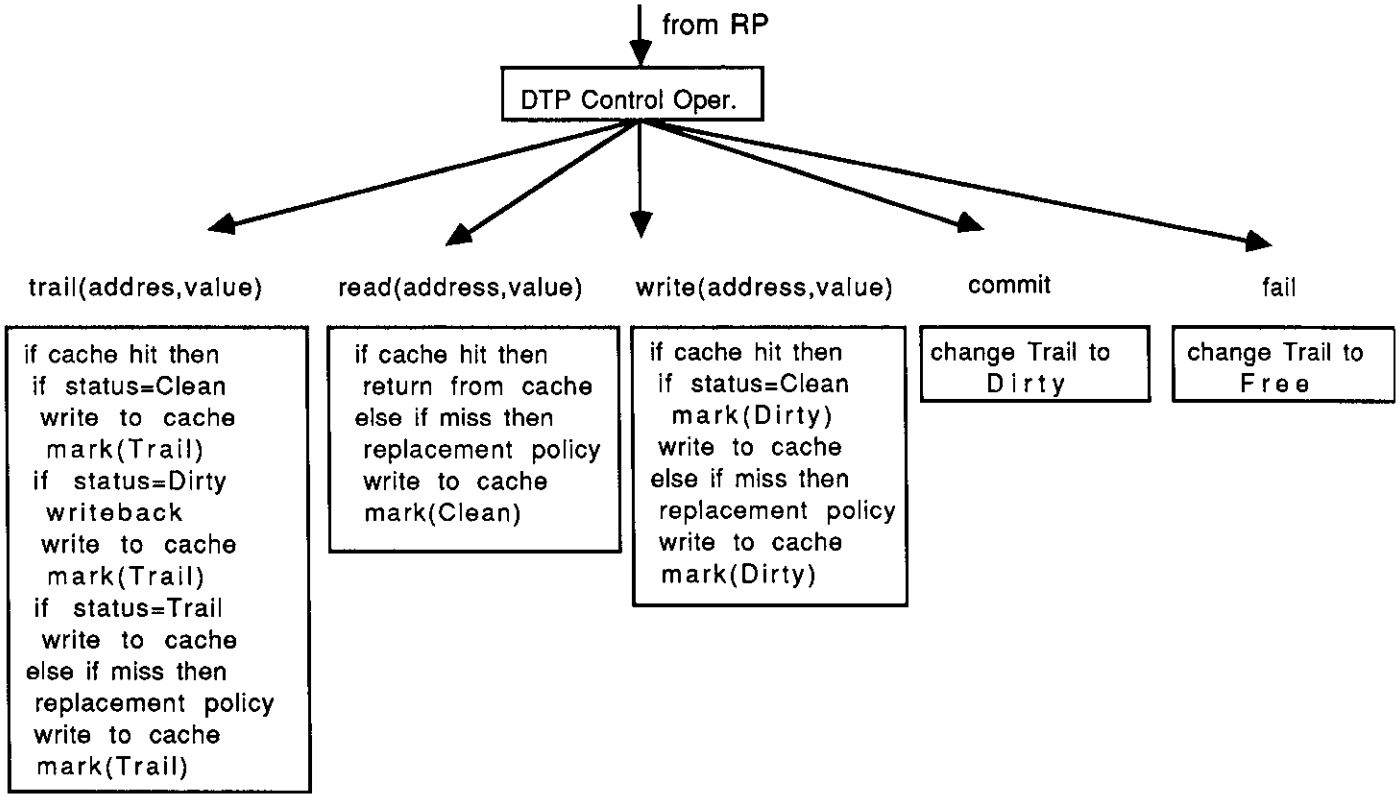


Figure 15: Data Trail Policy

7.2 Status and Future Work

We propose to simulated and evaluate the DTP Delayed Binding cache policy and determine the performance gains due the support for data trailing.

8 Conclusion

We propose to define, simulate and evaluate a special-purpose, high-performance single-processor for the execution of FCP. We define an execution model which exhibits internal, intra-processor concurrency. The execution model is derived by modifying the SAM for FCP. We also propose 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

[1] L. Alkalaj and E. Shapiro, "An Architectural Model for A Flat Concurrent Prolog Processor", UCLA TR-CSD-880007, February 1988.

[2] A. Ciepeilewski and B. Hausmann, "Performance Evaluation of a Storage Model for OR-Parallel Execution of Logic Programs", IEEE 1986.

[3] A. Ciepeilewski and S. Haridi, "Execution of Bagof on the OR-Parallel Token Machine", Proceedings of the International Conference on the Fifth Generation Computer Systems, 551-562 (1984).

[4] K. Clark and S. Gregory, "A Relational Language for Parallel Programming", Proc. ACM Conf. on Functional Programming Languages and Computer Architecture, 1981, 171-178.

[5] A. Colmerauer et al. "Un Systeme de Communication Homme-Machine en Francais", Groupe de Reserche en Intelligence Artificielle, Universite d'Aix-Marseille, 1973.

[6] J. Conery, "Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors", University of Oregon, 1987.

[7] T. Dobry et al. "Performance Studies of a Prolog Machine Architecture", 12th Annual International Symposium on Computer Architecture, 180-190, 1985.

[8] T. Dobry "A High Performance Architecture For Prolog" PhD Thesis, University of California, Berkeley, May 1987.

[9] I. Foster and S. Taylor et. al. "FCP and Parlog: A Basis for Comparisom", Department of Applied Mathematics, Weizmann Institute of Science, Israel, CS87-19, October 1987.

[10] H. Gallaire and J. Minker, "Logic and Data Bases", Plenum Press, New York 1978.

[11] A. Goto et al. "Highly Parallel Inference Engine: PIE.", New Generation Computing Vol. 2. 37-85 (1984).

[12] Genesereth ,"Logic Programming in Artificial Intelligence" M. Kaufmann, 1987.

[13] M. Hermenegildo, "An Abstract Machine Based Execution Model For Computer Architecture Design And Efficient Implementation Of Logic Programs in Parallel", PhD. Thesis, Austin Texas, TR-86-20, August 1986.

[14] R. Nakazaki et al. "Design of a High-speed Prolog Machine (HPM)", 12th International Conference on Computer Architecture, 191-197, 1985.

[15] 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.

[16] M. Kishi et al. "The Dataflow-based Parallel Inference Machine To Support Two Basic Languages in KL1", ICOT Research Center, Japan, TR-114, July 1985.

[17] M. Kishishita et al. "Distributed Implementation of FGHC", ICOT Japan, TR-159, March 1986.

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

[19] R. Kowalski, "Logic For Problem Solving", Elsevier North Holland, New York, 1979.

[20] J. Lloyd, "Logic For Problem Solving", Elsevier North Holland, New York, 1979.

[21] 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.

[22] R. Onai et al. "Architecture and Evaluation of a Reduction-Based Inference Machine: PIM-R", ICOT Research Center, Japan, TR-138.

[23] R. Overbeek et al. "Prolog on Multiprocessors", Argonne National Laboratory, Technical Report 1985.

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

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

[26] E. Shapiro, "Concurrent Prolog: A Progress Report", IEEE Computer, August 1986, Vol.19 No.8.

[27] W. Silverman, M. Hirsch et al. "The Logix System User Manual, Version 1.21", Department of Applied Mathematics, Weizmann Institute of Science, Israel, TR CS-21, July 1986.

[28] Y. Sohma, "A New Parallel Inference Mechanism based on Sequential Processing", IFIP TC-10, Working Conference on Fifth Generation Computer Architecture, (UMIST), Manchester 1985.

[29] K. Taki et al., "Hardware Design and Implementation of PSI", Proceedings of International Conference o Fifth Generation Computer System 1984, ICOT Japan.

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

[31] E. Tick and D.H.D. Warren "Towards a Pipelined Prolog Processor", New Generation Computing, 321-345 (1984).

[32] E. Tick "Prolog Memory-Referencing Behavior", Stanford TR No. 85-281, September 1985.

[33] P. Treleaven et al. "Computer Architectures for Artificial Intelligence" in Future Parallel Computers, Lecture Notes in C.S. No.272, Eds. P. Treleaven and M. Vanneschi, 1987.

[34] K. Ueda, "Guarded Horn Clauses", ICOT Japan, TR -103.

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

[36] D. H.D. Warren, "OR-Parallel Execution Models for Prolog", Theory and Practice of Software Developement, TAPSOFT 1987, Lecture Notes in Computer Science, No. 250, Springer-Verlag.