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

FLAT CONCURRENT PROLOG SEQUENTIAL ABSTRACT MACHINE CHARACTERISTICS

Leon Alkalaj

April 1989
CSD-890018

Abstract

Our motivation for performance analysis of existing FCP implementations is the design a special-purpose processor architecture for the efficient execution of FCP. We claim that the implementation level selected for analysis must yield results that are independent of the abstract machine emulation language and the host machine on which it executes. Therefore, we characterize algorithmic features of FCP program execution at the abstract machine level using an instrumented version of the existing abstract machine, called *Slogix* or Statistics Logix. Empirical results are obtained for a set of benchmark programs that are representative of a *System's Development Workload* currently in use.

The analysis of the dynamic goal management behavior shows a significantly higher complexity of goal *suspension* and *activation* compared to the previously reported behavior of simpler programs. In addition to their complexity, goal management operations occur on the average 1.3 times per goal reduction. These results motivate architectural support for goal management in a special-purpose environment. Furthermore, our results indicate that the average fraction of *redundantly selected clauses* during a goal reduction is 76%. This seriously motivates the need for advanced compilation techniques for clause-indexing. Other results that characterize goal management and goal reduction are also presented.

1 Introduction

Current compiler implementations of Flat Concurrent Prolog (FCP) on general-purpose processors emulate the execution of the FCP sequential abstract machine described in [8]. The same abstract machine is enhanced with a distributed unification algorithm and used in a distributed implementation of FCP [11]. Moreover, it is used as the basis for the design of a special-purpose processor architecture [2]. Similar abstract machines have been proposed for the implementation of other concurrent logic programming languages such as Parlog [6] and GHC [13].

Motivation for Performance Analysis

The ability to *systematically* enhance computer system performance by previously characterizing system behavior is the main motivation for performance analysis. The motivation may further be divided into the following two categories:

- Improving a *specific machine* implementation.
- The design of a *special-purpose* implementation.

In both cases, one way of enhancing performance is by providing architectural support in either of the following two forms:

- *General-purpose architectural support* is an effective way of improving performance using general-purpose components. For example, adding a memory cache, floating-point functional unit or upgrading the existing machine with a faster general-purpose processor is considered general-purpose support.
- *Special-purpose architectural support* involves the addition of components that are specifically *designed* for the performance improvement of the target language implementation. For example, a specialized cache with a cache-policy designed to match the characteristic memory referencing behavior is considered special-purpose architectural support.

Our motivation for performance analysis is the design of a *special-purpose* FCP implementation using *special-purpose architectural support.* Results of the analysis are used to design a processor architecture for the efficient execution of FCP [1]. Moreover, they provide insight on the operational behavior of FCP for a specific class of applications. As such, they may be used to improve system performance of existing general-purpose implementations.

Implementation Level Analysis Tradeoffs

Figure 1 depicts current implementations of the FCP abstract machine emulator on general-purpose processors. The emulator is implemented using an *emulation language* which executes on the *host machine.* The choice of the emulation language is usually made based on portability, programmability and implementation efficiency. The choice of the host machine is often based on availability and general-purpose performance.

Analyzing performance characteristics of an implementation may be conducted at various levels ranging from the very low, machine-level to the higher abstract machine (or even meta-interpreter)

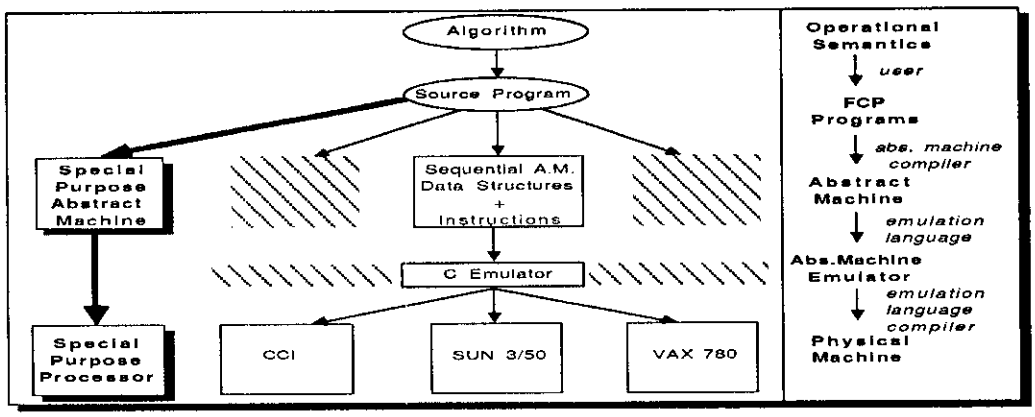

Figure 1: Abstract Machine Implementation and Emulation of FCP Operational Semantics

level. Selecting the appropriate implementation level to characterize program execution depends on the objectives of the performance analysis.

If the motivation is to improve the performance of the specific implementation that is being analyzed, then both low-level and high-level characteristics are meaningful to an optimization procedure. For example, analyzing the low-level memory reference behavior, one may suggest ways of reducing the virtual memory page-fault frequency which is specific to the measured system.

However, if the objective is to suggest ways of improving the performance of other systems that are based on the same abstract machine but use different emulation languages or host machines, the performance analysis must yield results that are applicable to the other implementations as well. For example, the previously described virtual memory optimization strategy can not be applied to implementations that do not use virtual memory.

The same argument becomes even more apparent if the objective is to design a special-purpose processor architecture for the efficient execution of the target language. Using an existing implementation to conduct the performance analysis, one must obtain results that are general enough and invariant of the lower-level implementation details.

The tradeoff involved is between the amount of detail that one can measure at the specific implementation level, and the generality of the analysis results. Low level results such as register allocation performance or processor memory bandwidth are dependent on the physical machine organization, the selection of the abstract machine emulation language and how it is implemented on the physical machine (for example which emulation language compiler is used). High-level abstract machine analysis, on the other hand, enables a less detailed performance analysis, but the results do not depend on any of the previously mentioned features.

Previous Analysis Approaches

Two previously reported performance evaluations of concurrent logic programming languages are discussed here. In [5], an approach for the analysis of FCP program execution is described. The motivation for the analysis is the design of a special-purpose processor architecture for FCP [7]. However, program execution is analyzed at the host machine level using the Unix *gprof* profiling tools to determine time consuming events of the emulator. As it was previously argued, these results depend on the implementation the analysis was performed on (C was used as the emulation language executing on the Vax). For example, the timing result of a *dereference* function depends

on how it was implemented in C (macro or function call), which C compiler was used and how it is supported on the host machine (some machines may have an implicit *deref* instruction mode).

In [12], an analysis of AND-parallel committed-choice execution of FGHC is compared to the OR-parallel uncommitted-choice execution of Aurora, running on the same multiprocessor machine. Program analysis is performed at the abstract machine level but the main emphasis is at the machine-level. This analysis approach *agrees* with the implementation level analysis tradeoffs previously described. Since both FGHC and Aurora are being optimized on the same multiprocessor machine, low-level analysis is meaningful. At the end, the raw program execution timings of the two systems are evaluated and compared.

The main observation is that Aurora has *better* memory performance characteristics than FGHC. The reason given is that Aurora makes better use of the more efficient stack-based storage model compared to the heap-based storage model of FGHC. However, it is noted that FGHC makes better use of parallelism in single-solution problems.

System's Development Workload Performance Analysis

Since our motivation is to design a special-purpose processor architecture and given the analysis tradeoffs described earlier, we conduct our performance analysis by characterizing program execution at the abstract machine level in a way that is independent of the selected emulation language and its implementation on the host machine.

To gather abstract machine level characteristics a new emulator, called *Slogix* or Statistics Logix was developed. Logically, existing FCP programs execute as in the original emulation environment. Program characteristics are obtained empirically by executing 7 benchmarks. The programs are authentic, unoptimized, written by various programmers and exhibit a variety of programming styles. They were not selected for performance but for their characteristic behavior and use of FCP parallel programming techniques. All programs are *real* applications used in the Logix environment. They are large in terms of source code size and execute over lengthy periods of time, thus achieving a reasonable average of program behavior.

The selected programs are specific of a single environment. That is, due to the lack of a large spectrum of applications, the selected programs are limited in their scope to applications for program development. We refer to the selected set of programs as the *Sytem's Development Workload.* In Table 1 we show the size of the programs and the number of goal reductions performed. Programs written using read-only unification are labeled as fcp(?) and those that use implicit input unification in the guard as fcp(:) [10]. Programs executing in *interpret* mode report failure and can thus be interpreted and debugged. Programs in *trust* mode execute expecting all goals to succeed.

A *typical* workload is described as follows. FCP programs for system prototyping and simulation

Info	FCP Benchmark Programs						
	Compiler	Simulator1	Simulator2	Debug	Solver	Distribute	Logix
Lang.	fcp(?)	fcp(?)	fcp(?)	fcp(:)	fcp(?)	fcp(:)	fcp(?)
Mode	trust	trust	interpret	trust	interpret	interpret	trust
Size	3885	2066	2066	1670	676	6376	10000
Red.	7075380	3009280	7722376	1593286	409075	259283	1481900

Table 1: FCP Benchmark Information

are developed under the *Logix* system [14]. For example, two different simulation environments are being modeled. One is a concurrent, multi-functional unit environment for the execution of FCP, referred to as *Simulator1* (a second version called *Simulator2* is also used). The second is a simulator of a distributed FCP implementation executing on a ring architecture, called *Distribute*. They are debugged using the *Debugger* program and compiled using the FCP *Compiler*. Program control flow analysis using abstract interpretation is performed by the *Solver* program. The characteristics of the 7 programs are equally weighed even though they vary in size and execution time.

Our approach for performance analysis differs from the two previous approaches described. With the work described in [5] we share the motivation but not the approach. Our analysis is only at the abstract machine level rather than being machine dependent. With the analysis performed in [12] we share the approach but not the motivation, since we are concerned with a special-purpose processor architecture.

Our work differs significantly from both approaches in the selection of the workload used for empirical evaluation. In [5] small applications are used that are not representative of a specific workload. In [12], larger program applications are used but the author correctly notes that the main drawback of the analysis is the use of "small symbolic manipulation problems". We emphasize that our selected programs characterize a specific workload that is representative of an existing working environment.

Summary of Results and Outline of Paper

Several features of FCP program execution are given special attention in our analysis. First, we characterize the *dynamic behavior of goals*, including goal creation, termination and asynchronous communication via goal suspension and data-flow activation. We reaffirm the suggestion made in [4] that the complexity and frequency of goal suspension and activation is a potential source of implementation inefficiency. Moreover, we quantify our results using metrics that describe the frequency and complexity of goal management in general. Our measurements indicate that the System Development Workload exhibits a degree of goal management activity significantly higher than previously observed in simple applications.

The second set of features quantified in our analysis describe the characteristic *behavior of goal reduction*. Particularly, we evaluate the degree of redundant clause selection performed attempting to unify a goal with a set of potentially matching clauses. Since FCP is a committed-choice language, a goal may commit to only one clause. In the FCP abstract machine clauses are selected sequentially, in textual order. Quantifying redundant clause selection is important to understand the potential for performance improvement available if advanced clause-indexing compilation techniques are applied. For example, the *decision-tree* compiler approach described in [9] performs *inter-clause* analysis of argument structures to eliminate redundant argument checking. Our measurements indicate that an average of 76% of all clause-tries are redundant in the considered workload. We also investigate the complexity of other goal reduction features such as *argument dereferencing*, *branching on data types* and *clause-trailing*.

In the remainder of this paper we first briefly review the FCP sequential abstract machine. Following this we present our results and concluding remarks.

2 An Overview of the FCP Sequential Abstract Machine

The FCP abstract machine was significantly influenced by the design of the Prolog abstract machine (WAM) [15]. General unification is implemented using the same get, put and unify instructions extended to handle the unification of *read-only* annotated variables [8]. Several important features distinguish the FCP abstract machine from WAM.

- Non-deterministic goal scheduling.
- Goal suspension and data flow activation.
- Heap storage model.

As a concurrent logic programming language intended for parallel processing, the program *resolvent* is modeled as a set of non-deterministically scheduled goals, much like processes in a multiprocessing system. In the FCP abstract machine, a queue of goals is used instead of the stack-based control used in WAM.

To model inter-goal communication and synchronization, FCP and concurrent logic programming languages in general, define *rules of suspension.* A goal *suspends* if there is insufficient data to perform successful goal-head unification whereas the reduction may succeed when the data becomes available. In the FCP abstract machine, goal suspension is implemented using *suspension lists* associated with each suspending variable. Data flow activation is implemented by rescheduling the suspended goals when the suspending variables receive data from other active goals. In Figure 2, we show goal **p** suspended on variable **X** and goal **q** on variables **X** and **Y**. *Suspension records* are linked into suspension lists if more than one goal is suspended on the same variable. Single *activation records* per suspended goal prevent multiple activation of the same goal.

The third important feature that distinguishes the FCP abstract machine from WAM is the storage model. FCP uses a *heap-based* model as opposed to the stack-based model, to store both program and control structures. The common heap is an essential part of the parallel programming model. It enables the implementation of asynchronous inter-goal communication using dynamically allocated logical variables as a continuous communication *stream.*

In Figure 3, we show the organization of the FCP abstract machine storage model. Besides the heap, there are two dedicated stack memory areas. One is the Trail Stack (TS) used as follows. Since FCP allows destructive assignments during clause-head unification, (this is not the case in GHC or Parlog), the assignments are trailed in TS. If the clause-try fails, the assignments are undone, and an alternative clause-try is attempted. The second stack area is the Suspension

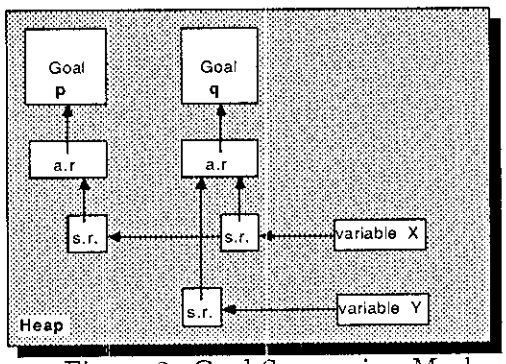

Figure 2: Goal Suspension Mechanism

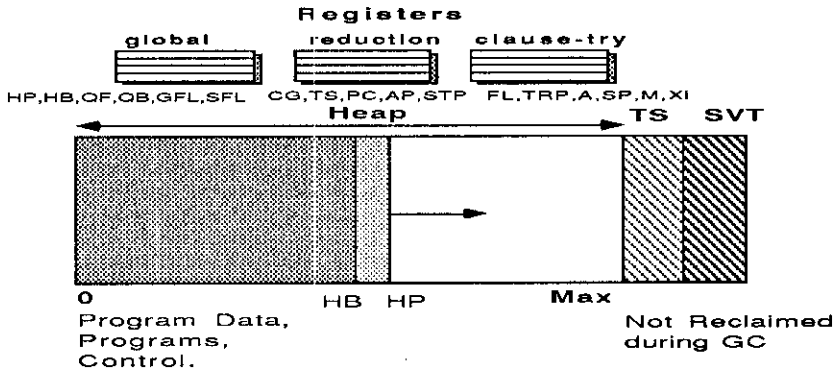

Figure 3: FCP Abstract Machine Storage Model

Variable Table (SVT). A clause-try that does not fail nor succeed but may succeed when more data becomes available, places the suspending variables' addresses in SVT. If there is no clause-try that can succeed and there is at least one address in SVT, goal reduction suspends on the variables stored in SVT. Also shown in Figure 3 are a set of abstract machine registers addressable by the abstract machine instructions.

3 Results of Abstract Machine Analysis

We present the FCP abstract machine analysis in two sections. First we discuss the *dynamic goal management* behavior followed by *dynamic goal reduction* statistics.

3.1 Dynamic Goal Management Behavior

The execution of FCP programs results in the creation and reduction of cooperating units of work called *goals*. There are several possible outcomes of a goal reduction which we refer to as its *dynamic* behavior. A goal may *create* new goals, *terminate* execution or reduce by *iteration*. In the latter case, a goal reduces without creating new goals or terminating. If there is insufficient data to successfully complete a goal reduction, the goal *suspends* and is later *activated* when the data becomes available.

In table 2 we show the total number of goals created, terminated and reduced during program execution. Also shown are the number of goal suspensions and activations. To better characterize the dynamic behavior of FCP goals we define the following three parameters:

- The Average Life Time (ALT) of a goal is the ratio of the number of goal reductions per

Goal	FCP Benchmark Programs						
	Compiler	Simulator1	Simulator2	Debug	Solver	Distribute	Logix
Create	1890235	1133169	2501064	573869	156491	104799	477568
Terminate	1887576	1131614	2499518	572005	155306	97184	474729
Suspend	887590	1892561	2513692	502659	52448	83315	306450
Activate	884945	1891022	2512160	500816	51277	75714	303625
Reduce	7075380	3009280	7722376	1593286	409075	259283	1481900

Table 2: Goal Management Statistics

created goal.

- The Average Suspension Ratio (ASR) is the ratio of the number of suspensions per goal reduction.
- The Average Goal Management (AGM) activity is the ratio of goal creations, terminations, suspensions and activations per goal reduction.

In table 3 we show these parameters for the selected set of FCP programs. An average of 2.9 goal reductions are performed prior to goal termination. Therefore, in terms of goal reductions, one can characterize the workload as consisting of *light weight* or *short-lived* computations that iterate on the average 3 reductions prior to termination. No correlation is made here between the goal reduction and the actual execution time, which is implementation dependent.

The average goal suspension ratio ASR varies more significantly, with the mean value being 0.29 suspensions per reduction. That is, a goal suspends every 3.4 reductions. The benchmarks used in [12] recorded a significantly smaller value for the suspension rate, that is, between 0 and 0.09. Whereas this value is closer to the characteristic behavior of the *Compiler* workload, the System Development workload exhibits over 3 times their maximum value.

The average goal management activity per goal reduction, AGM, is similar for all of the programs. The average value is AGM = 1.3 goal management operations per reduction.

Active Goal Queue

Since FCP program execution results in the creation and reduction of numerous concurrent goals, it is interesting to consider the distribution of the number of actively reducible goals. The size of the active queue indicates the number of goals that could be shared and reduced concurrently in a parallel implementation. However, since the reducible goals differ in size and complexity, one can not simply correlate the size of the active queue with the degree of parallelism available in the program.

In Figure 4 we show the distribution of the active queue size for queue size values between 0 and 30. For all programs the maximum value of the distribution is for queue size values less than 10. In Table 4 we show the maximum queue size, the average queue size value and the percentage of the queue size distribution not accounted for in Figure 4.

It is interesting to note that the *Compiler* program reached a maximum queue size value of 3195, whereas the other programs had values between 50 and 200. Despite the large number of spawned goals in all the programs, the average active queue size ranges from approximately 6 for the *Compiler* to over 13 for the *Solver* program. It should be pointed out that the selected FCP programs are not known for their parallelism nor was the program data chosen to increase the

	FCP Benchmark Programs						
	Compiler	Simulator1	Simulator2	Debug	Solver	Distribute	Logix
ALT	3.7	2.6	3.1	2.8	2.6	2.5	3.1
ASR	0.125	0.628	0.325	0.315	0.128	0.321	0.206
AGM	0.8	2	1.29	1.3	1.3	1.2	1.1

Table 3: Goal Management Parameters

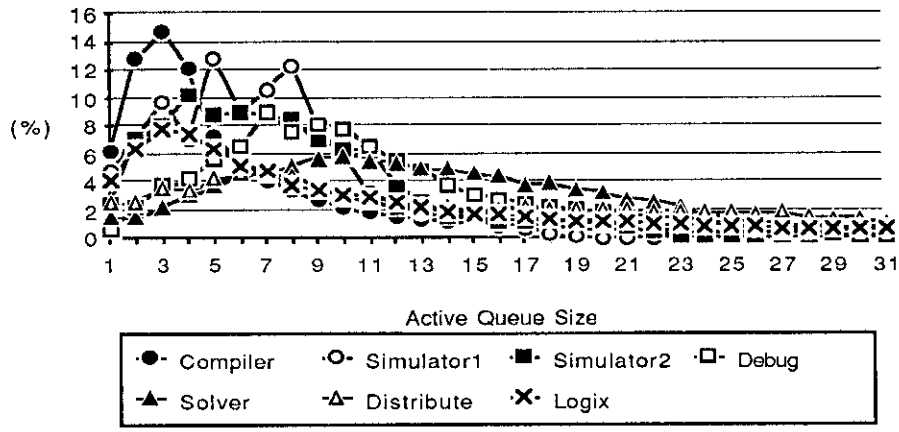

Figure 4: Distribution of FCP Active Queue Size

Size	FCP Benchmark Programs						
	Compiler	Simulator1	Simulator2	Debug	Solver	Distribute	Logix
Max.	3195	72	56	50	51	148	200
Range	10%	0.2%	0.8%	1.2%	2.4%	24%	20.5%
Ave.	6	7	8	11	13	10	7.5

Table 4: FCP Active Queue Size: Maximum and Average

concurrency. The potential parallelism shown in Figure 4 and Table 4 are only due to the FCP programming model. The workload average for the active goal queue size is 9.

Goal Suspension Complexity, σ

In FCP, goals communicate using shared logical variables. Goal suspension occurs as a result of checking whether certain variables defined in the clause-head and guard, conform with the *suspension rules*. If not, a goal may suspend on more that one variable, and similarly more than one goal may suspend on a single variable, as shown in Figure 2.

The complexity of the goal suspension mechanism is proportional to the number of variables a goal suspends on, σ. For each variable, a *suspension note* is allocated and placed on the suspension list. In Figure 5 we show the distribution of the number of variables a goal suspends on during program execution. The results indicate that suspension generally occurs on few variables and that the average number suspension variables per goal suspension for the considered workload is $\sigma = 2.7$.

This result is higher than expected, as it was often assumed that goal suspension generally occurs using only 1 variable, and less frequently with 2 or 3 variables. This can be explained as follows. First, the complexity of goal suspension in large applications of concurrent logic programming languages was never precisely evaluated. The assumption that suspension occurs mostly on 1 or 2 variables was made by observing small applications that do not exhibit the program complexity characteristic of the workload we consider. However, it may be the case that more advanced compilation techniques will be able to eliminate redundant suspensions on more variables. Such compilation techniques are still a matter of research.

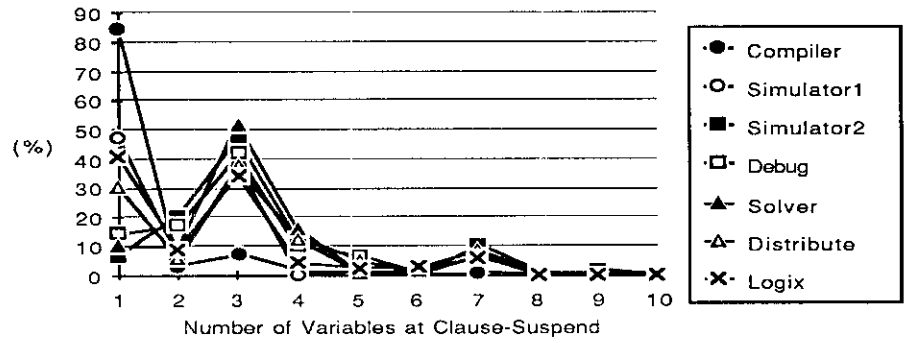

Figure 5: Distribution of the Number of Suspension Variables

Vars	FCP Benchmark Programs						
	Compiler	Simulator1	Simulator2	Debug	Solver	Distribute	Logix
Total	1228145	4340234	8281821	1590037	170657	235218	809585
Suspend	886526	1891370	2512452	501468	51355	82122	305386
Ave.	1.4	2.3	3.3	3.1	3.3	2.8	2.5
Max.	8	9	12	14	14	10	15
(1-4) Var	97%	92%	87%	84%	88%	87%	89%

Table 5: Number of Suspending Variables at Suspend: Maximum and Average

In Table 5 we show the total, maximum and average number of suspension variables during program execution. Note that the maximum value ranges from 8 variables for the *Compiler* to 15 variables for the *Logix* operating system. The fraction of all goal suspensions that occurred with 4 or less variables ranges from 84% for the *Debugger* to 97% for the *Compiler*.

Redundant Suspension Note Allocation

A clause-try that suspends does not immediately result in the suspension of the goal since there may exist another clause that can succeed. Therefore, alternative clauses are tried. The goal suspension mechanism used in the sequential abstract machine allocates suspension notes as if the clause-try will suspend. If it does not, the allocated suspension notes are discareded.

In Figure 6 we show the distribution of the number of goal suspension variables stored in the suspension table at clause commit time. The suspension notes allocated in these cases are redundant. In Table 6 we show the fraction of all clause-commits that occurred without allocating redundant suspension notes. In most cases this is between 89% to 98% of all the goal reductions. One should also note the average number of suspension notes allocated (average number of variables) per redundant clause-suspension. This value is also higher than would be expected, that is, 1.39 for the *Compiler* and 3.5 for *Simulator1* and on the average 2.5 redundantly allocated suspension notes. The maximum number of redundantly allocated suspension notes is high as 7.

In Table 6 we show the total number of allocated suspension notes, the total number of *essential* suspension notes and the total number of *redundant* notes. Therefore, an average of 16% of all allocated suspension notes are redundantly allocated. Once again, the essential suspension notes are those that were allocated during a clause-suspension that actually resulted in a goal suspension.

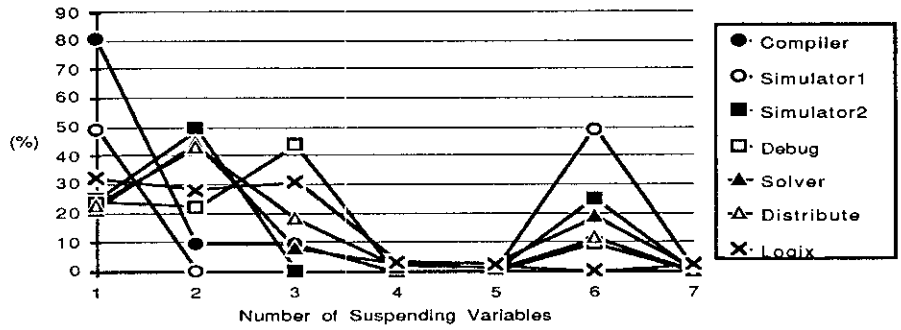

Figure 6: Distribution of Goals Suspension Variables at Clause-Commit

	FCP Benchmark Programs						
	Compiler	Simulator1	Simulator2	Debug	Solver	Distribute	Logix
Empty SVT	98%	91%	93%	89%	97%	95%	97%
Ave.	1.3	3.5	2.8	2.6	2.8	2.5	2.3
Max.	3	6	6	6	6	6	7
Total	2374106	7468767	12849379	2710393	271794	362952	1339904
E	2115735	6232795	10795513	2092696	223105	318533	1116035
R	258371	1235972	2053866	617697	48689	44419	223869
R/(E+R)	11%	16%	16%	23%	18%	12%	17%

Table 6: Suspending Variables at Clause-Commit

A simple optimization of the suspension algorithm allocates suspension notes only after a goal suspension is determined. This change has already been integrated into the FCP abstract machine as a result of this analysis.

Goal Activation Complexity, α

Suspended goals are activated upon a successful clause-head unification and guard evaluation, that is, after the clause commits. When a clause-try succeeds, all variables that received new assignments and that have goals suspended waiting for these values, are activated. The complexity of goal activation is proportional to the number of goals activated, α. In Figure 7 we show the distribution of the number of activated goals at clause-commit. In most cases, a clause-commit will not activate any goal. Either there were no new assignments made during the clause-head unification and guard evaluation, or, if there were assignments made, there were no goals suspended waiting for these values. Only in the case of the *Simulator1* program were as few as 50% of the commits empty, whereas in the other programs as many as 88% and more clause-commits did not activate any goals.

From Table 7 we see that the average number of goals activated per non-empty clause-commit is generally slightly over 1. That is, $\alpha = 1.2$. However, the maximum number of goals activated may be unexpectedly high. In the *Logix* program the maximum number of goals activated at any time was 142 and in the *Compiler* as many as 192.

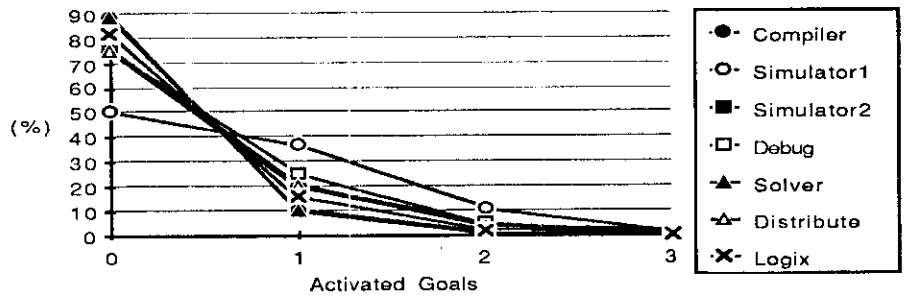

Figure 7: Distribution of the Number of Goals Activated at Clause-Commit

Goals	FCP Benchmark Programs						
	Compiler	Simulator1	Simulator2	Debug	Solver	Distribute	Logix
Activate	884945	1891022	2512160	500816	51277	75714	858276
Ave.	1.1	1.3	1.3	1.2	1.1	1.2	1.2
Max.	192	12	23	10	4	20	142

Table 7: Activated Goals at Clause-Commit

3.1.1 Summary of Goal Management Behavior

In general, the dynamic goal management behavior of the System Development Workload exhibits a complexity significantly higher than that reported in previous performance evaluations that measure simple applications. The following is a brief summary of the recorded goal management characteristics.

- The execution of large FCP programs such as Logix, compilers or simulators that are characteristic of a System Development Workload, result in the creation of *light weight* goals that perform on the average 3 reductions prior to termination.
- Goals *suspend* execution waiting for data to be communicated by another goal at an average goal suspension rate of ASR = 0.29. The ASR reported here is significantly higher than reported in [12].
- An FCP goal frequently performs goal management operations. The average goal management activity AGM is 1.3.
- Newly created goals are placed onto an active goal queue from which they are scheduled for execution. The average size of the queue is 9.
- FCP goals communicate using the goal suspension and activation mechanism. In almost 90% of the cases, a goal suspends on 4 or less variables. However, the average number of variables a goal suspends on is $\sigma = 2.7$.
- Goal suspension involves the allocation of goal control structures. 16% of the total number of allocated suspension notes are redundant. This corresponds to those cases where a clause-suspension does not result in a goal suspension.

- Suspended goals are activated only at clause-commit. Most clause-commits do not activate goals. The average number of goals activated at clause-commit is $\alpha = 1.2$.

It is the above summarized complexity and frequency of goal management operations that motivates the need for architectural support in a special-purpose environment. In [3], the overlapped execution of goal management and goal reduction is proposed. The efficient execution of goal management operations is performed using a special-purpose *goal cache*.

3.2 Goal Reduction Statistics

Using *Slogix* we are able to determine numerous properties of goal reduction. We report here the following selected results.

- The number of *redundant* clause-tries prior to a clause-commit.
- The average number of arguments in the scheduled goal reductions.
- The average length of each argument that is dereferenced.
- The distribution of dereferenced data types.
- The amount of variable trailing performed during each clause-try.

Redundant Clause Selection

We consider all clause-tries that result in goal reduction and all clause-suspensions that result in goal suspension as useful clause-tries. Therefore, all clause-failures and all clause-suspensions that result in goal reduction are considered *redundant* clause-tries. Since the total number of clause tries, CT, is equal to the sum of the clause failures CF, clause suspensions CS and reductions CR, the fraction of redundant clause-tries, F_{ct}^r is expressed as follows:

$$F_{ct}^r = 1 - (\frac{CR}{CT} + \frac{GS}{CT}) \tag{1}$$

In Table 8 we show the total number of clause-tries (CT), clause suspensions (CS), clause failures (CF) and successful clause-tries or reductions (CR). Since a clause suspension need not result in the suspension of the goal, we also show the total number of goal suspensions (GS).

The rate of successful clause-tries per reduction is CR/CT = 19%. The rate of goal suspensions per clause-tries is GS/CT = 5%. In other words, the average redundant clause-selection rate is as high as $F_{ct}^r = 76\%$ for the System Development Workload.

It is precisely this issue of the large overhead of the redundant clause-selection that is addressed in the decision-tree compilation techniques proposed in [9].

FCP Goal Arguments

An FCP goal may vary in size from small sized goals consisting of only a single argument, to more complex goals with an arbitrarily (but finitely) large number of goal arguments. In Figure 8 we

	FCP Benchmark Programs						
	Compiler	Sim.1	Sim.2	Debug	Solver	Distr.	Logix
CT	20144798	18945686	48998264	7468752	4525015	2595415	5856330
CR	7075380	3009280	7722376	1593286	409075	259283	1481900
CF	10655572	3583291	24938066	2948379	3796276	1793187	2847446
CS	2413845	12353114	16337818	2927087	319664	542945	1526984
GS	887590	1892561	2513692	502659	52448	83315	306450
$\frac{CR}{CT}$	0.35	0.16	0.16	0.21	0.09	0.1	0.25
$\frac{CF}{CT}$	0.53	0.19	0.51	0.39	0.84	0.69	0.49
$\frac{CS}{CT}$	0.12	0.65	0.33	0.39	0.07	0.21	0.26
$\frac{GS}{CS}$	0.37	0.15	0.15	0.17	0.16	0.15	0.20
$\frac{GS}{CT}$	0.04	0.10	0.05	0.07	0.01	0.03	0.05
F_{ct}^{r}	0.60	0.74	0.79	0.82	0.9	0.87	0.69

Table 8: Clause-Try Statistics

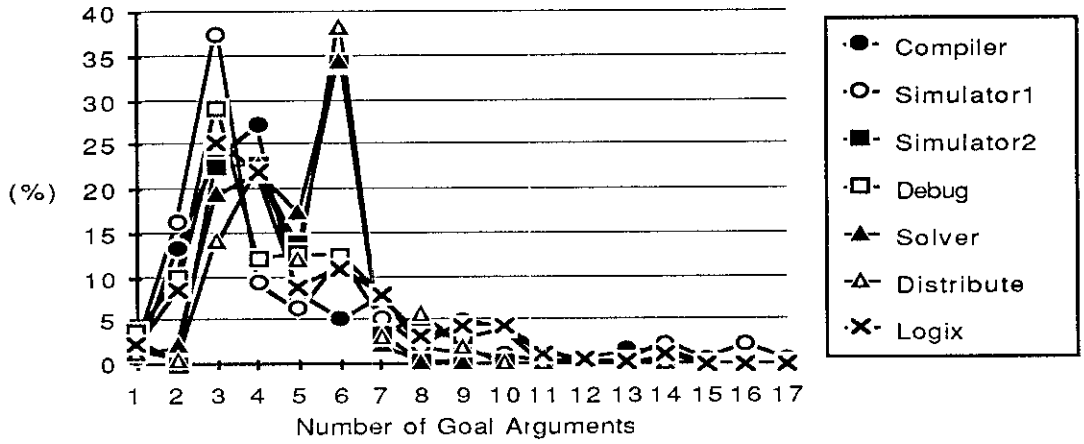

Figure 8: Distribution of the Number of Goal Arguments

show the distribution of the number of goal arguments found in the selected FCP benchmarks. The goal size indicates the number of program arguments and does not include the program counter (or perhaps some other system parameters that may be stored as part of a goal).

In Table 9 we also show the total number of allocated arguments, the maximum number of arguments found in a single goal and the average number of arguments per created goal. From the distribution of the number of goal arguments shown in Figure 8 we see that most goals have less than 7 arguments. Whereas the maximum number of goal arguments is between 14 and 17, they occur very rarely. The average size is quite similar in all of the selected programs with an average of 5 arguments per goal.

Argument Dereferencing

Goal head unification consist of matching arguments in the calling goal with the corresponding arguments of a called goal. A goal argument is denoted as a reference to a program data structure or as a reference to another reference. An argument reference is *dereferenced* prior to matching

	FCP Benchmark Programs						
	Compiler	Simulator1	Simulator2	Debug	Solver	Distribute	Logix
Total	39708892	22946514	51438876	9773661	2154104	1750183	8678530
CT	7961926	12991789	10234840	2094761	460433	341405	263745
Ave.	4.9	4.5	4.6	4.7	4.7	5	4.9
Max.	14	17	14	14	14	14	14

Table 9: Goal Arguments: Maximum, Average

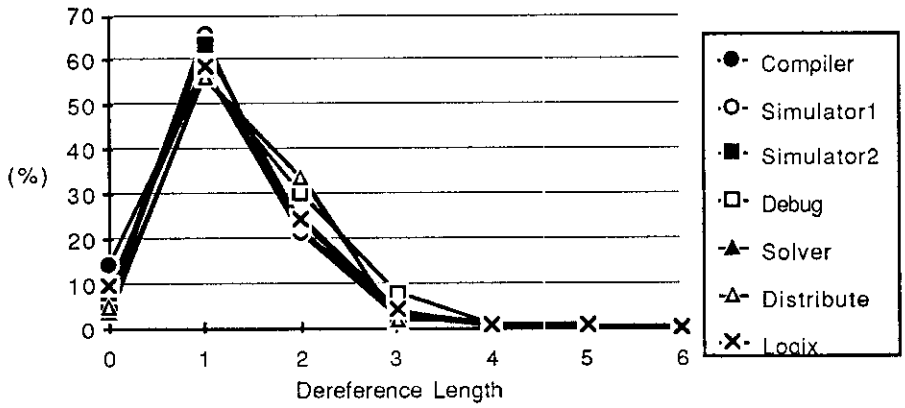

Figure 9: Distribution of Dereference Length

with another program data structure.

Argument dereferencing consists of following a chain of reference pointers until a non reference is encountered. It is performed not only during goal-head unification for matching goal arguments, but also prior to the assignment of a value to a variable. In general, dereferencing is considered a frequent operation in FCP.

In Figure 9 we show the distribution of the dereference chain length found in the FCP benchmark programs. We see that most of the dereference chains are less than 3. In Table 10 we show the total number of dereference calls D and the total dereference length L. We also show the average number of dereference calls per goal clause-try. The average dereference length per dereference call equal to 1.3.

However, very large dereference chains in FCP programs are not uncommon. For example, the repeated *variable-to-variable* unification used to implement the *short-circuit* technique of program termination detection results in long dereference chains. In the sampled programs, the maximum

	FCP Benchmark Programs						
	Compiler	Simulator1	Simulator2	Debug	Solver	Distribute	Logix
D	86152792	74505920	174259360	67946270	13967005	8492166	25679242
L	115424576	183747648	481510592	115292635	28474510	23250972	47494092
Ave.	1.2	1.2	1.2	1.4	1.4	1.4	1.3
$\frac{D}{CT}$	4.3	3.9	3.6	9.1	3.1	3.3	4.4

Table 10: Dereferencing Statistics

Type	FCP Benchmark Programs						
	Compiler	Simulator	Simulator2	Debug	Solver	Distribute	Logix
Var	23.2%	32%	32.4%	20.5%	36.7%	37.7%	30.1%
RO Var	4.8%	12.3%	11.2%	5.3%	3.1%	6.7%	8.1%
Int	17%	4.5%	3.8%	3.6%	7.5%	4.7%	10%
Real	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
Str	15%	15.6%	16.25%	22.6%	12.6%	13.7%	16.3%
Nil	2.6%	1.3%	1.25%	2.3%	0.8%	1.2%	2.1%
Car Ref	13.3%	5.7%	7.1%	9.9%	3.7%	4.6%	6.8%
Car Int	7%	1%	2.3%	2%	0.7%	0.2%	4.7%
Car Nil	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.01%
Tuple	16.1%	26.4%	24.4%	32%	34.6%	29.7%	21%
Vector	0.9%	1.1%	1.3%	1.9%	0.3%	1.5%	1.1%
Total	86152672	74505920	174259360	67946270	13967005	8492166	25679242

Table 11: Distribution of Dereferenced Data Types

dereference length reached 400 and more.

Branching on Dereference Types

Dereferencing an argument is commonly followed by a polymorphic operation. That is, further program continuation depends on the type of the dereferenced object. To see what is the *spectrum* of dereferenced data types, we show in Table 11 their dynamic distribution. The most frequently dereferenced data structure is in most cases the logical variable (Var), the tuple (Tup), string (Str) and integer (Int) data types. In all of the selected FCP programs, these four data types accounted for an average of 79% of all the dereferenced objects.

Clause Trailing

A clause-try may assign values to logical variables. If it succeeds the bindings are correctly visible to the programming environment and goal reduction commits. However, if a clause-try fails or suspends, the assignments must be undone so that a new clause-try starts with the same memory state that preceded the clause-try. It is for this reason that the assignments to memory are trailed during a clause-try. This is commonly referred to as *clause-trailing*.

If FCP, trailing consists of storing the address and previous value of a trailed logical variable into the Trail Stack. In case of clause-try failure or suspension, the entries are poped from the trail, and the old memory state is restored.

In Figure 10 we show the distribution of the trail size at clause-commit. Even though the behavior of the distribution is not the same for each of the selected FCP programs, in most cases the trail size is less than 7 entries. In Table 12 we show the average and maximum size of the trail, as well as the total number of trailed entries and clause-commits. The average number of trailed entries is 5. Note that the maximum number of trailed entries may be high. In the case of the *Distribute* program as many as 163 entries were trailed.

In Figure 11 we show the distribution of the trail size at clause-suspend. It is quite similar for

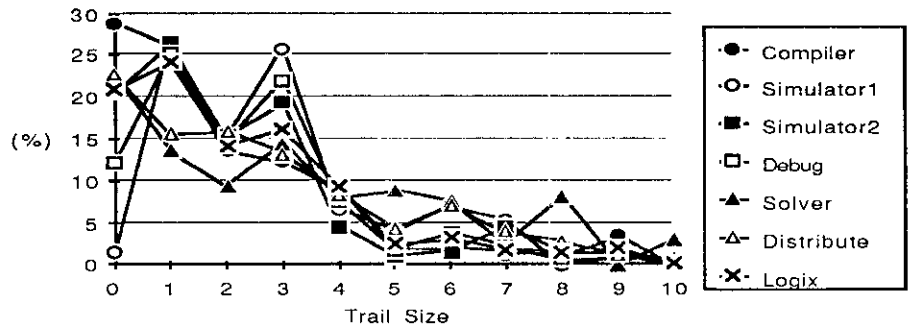

Figure 10: Distribution of Trail Size at Clause-Commit

	FCP Benchmark Programs						
	Compiler	Simulator1	Simulator2	Debug	Solver	Distribute	Logix
Commit	7075380	3009280	7722376	1593286	409075	259283	1481900
Size	23781044	20352652	36312112	9740676	2322758	1415961	6761644
Max.	36	47	22	35	54	163	87
Ave.	3.4	6.8	4.7	6.1	5.6	5.5	4.6

Table 12: Trailing at Clause-Commit

all the selected FCP programs. In most cases, the trail size is less than 4 entries. In Table 13 we give the total number of clause-suspension, the total number of trailed entries, and the maximum and average size of the trail.

The average size of the trail at clause-suspension is 1.4. This is smaller than the average trail size at clause-commit. Also, the maximum number of trailed entries is generally smaller.

In Figure 12 we show the distribution of the trail size at clause-failure. In most cases it is less than 2 entries. In Table 14 we can see that the average trail size is 0.5. However, the maximum number of trailed entries is also high, ranging from 10 for the *Simulator1* program to 81 in the case of *Logix*.

From the distributions of the trail sizes during a clause-try, one can note the following. Most trailing occurs in clause-tries than succeed. In fact, an order of magnitude more than during the clause-tries that fail. That is, most clause-tries fail before they require much trailing. Even though

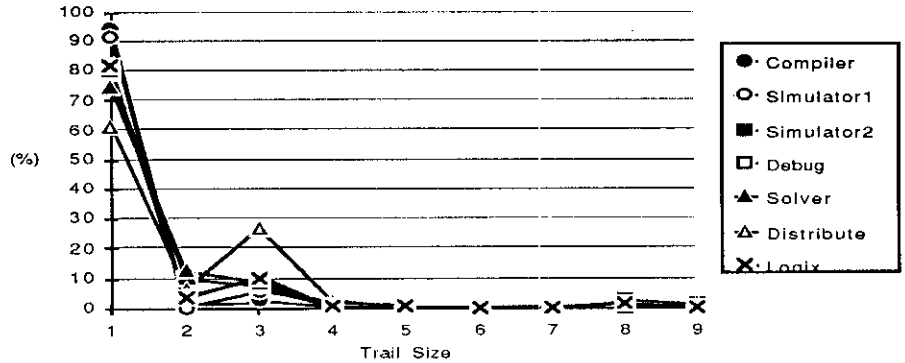

Figure 11: Distribution of Trail Size at Clause-Suspend

Trail Size	FCP Benchmark Programs						
	Compiler	Simulator1	Simulator2	Debug	Solver	Distribute	Logix
CS	2413845	12353114	16337818	2927087	319664	542945	1526984
Total	2814496	16467694	29217068	5050452	500481	1019755	2412390
Max.	12	16	18	21	21	26	92
Ave.	1.1	1.1	1.2	1.6	1.5	1.7	1.4

Table 13: Trailing at Clause-Suspension

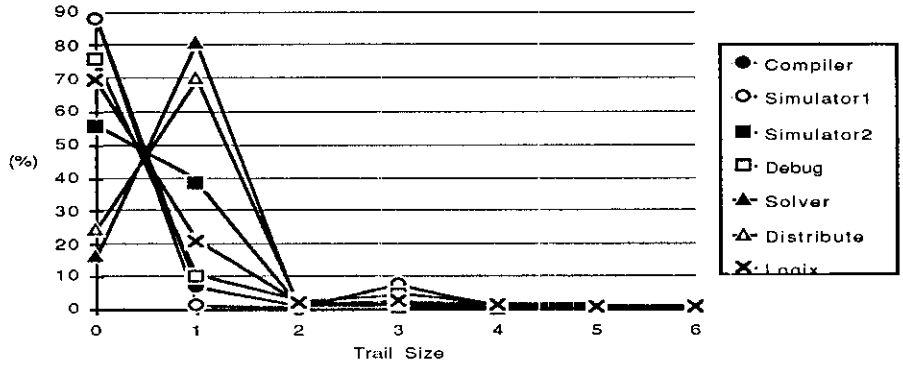

Figure 12: Trailing at Clause-Failure

Trail Size	FCP Benchmark Programs						
	Compiler	Simulator1	Simulator2	Debug	Solver	Distribute	Logix
CF	10655572	3583291	24938070	2948379	3796276	1793187	2847446
Total	2762711	1380708	13763565	2253966	3467264	1624860	1811428
Ave.	0.2	0.4	0.5	0.4	0.9	0.8	0.5
Max.	38	10	12	20	46	18	81

Table 14: Trailing at Clause-Failure

the average number of variables trailed is generally less than 6, it can be expected that even with the elimination of redundant clause-tries, the main bulk of clause-trailing would remain during the clause-tries that succeed.

3.2.1 Summary of Goal Reduction Statistics

- We refer to all clause-tries that fail and all clause-tries that suspend but do not result in goal suspension, as *redundant*.
- In the System's Development Workload, 76% of all the clause-tries are redundant.
- The size of the FCP goals vary considerably in size. The average number of arguments is 5.
- The average dereference length of an argument is 1.3.
- Following a dereference is a *branch_on_type* operation. The most frequently dereferenced data types as: Var, Tuple, String and Integer.
- Clause-trailing is mainly performed in clause-tries that result in commit. Clause-tries that fail trail an order of magnitude less.

4 Conclusion

The motivation for the performance analysis presented in this paper is the design of a special-purpose processor architecture for the efficient execution of FCP. With this motivation in mind, we emphasize that the analysis performed on existing implementations must yield low-level implementation independent results. We thus conduct abstract machine level performance analysis that is indipendent of the emulation language and host machine implementation. For this purpose, an instrumented version of the abstract machine was developed, called *Slogix*, (Statistics Logix). We emphasize in our analysis approach the use of a specific System's Development Workload that is empirically evaluated. The workload is representative of an existing working environment. We conclude that the complexity and frequency of goal management activity for the specific workload is significantly higher when compared to previously used benchmarks. Our measurements also show that there is a high degree of redundant clause-selection performed. This further motivates the need for advanced clause-indexing techniques.

References

[1] L. Alkalaj. *Architectural Support for Concurrent Logic Programming Languages.* Doctoral Dissertation UCLA/CSD 89, University of California, Los Angeles, June 1989.

[2] L. Alkalaj. *A Proposal For A FCP Processor Architecture.* Technical Report UCLA/CSD 88/0035, University of California, Los Angeles, April 1988.

[3] L. Alkalaj and E. Shapiro. An Architectural Model for a Flat Concurrent Prolog Processor. In *Proceedings of the 5th International Conference/Symposium on Logic Programming*, pages 1245 – 1323, Aug 1988.

[4] I. Foster and S. Taylor. *FCP and Parlog: A Basis for Comparison.* Technical Report CS 87, Weizmann Institute of Science, Applied Mathematics Department, July 1987.

[5] R. Ginosar and A. Harsat. *Profiling LOGIX: A Step Towards a Flat Concurrent Prolog Processor.* EE Pub. Technical Report No. 629, Technion Institute of Technology, Haifa, February 1987.

[6] S. Gregory. *Parallel Logic Programming in PARLOG, The Language and its Implementation.* Addison-Wesley, 1987.

[7] A. Harsat and R. Ginosar. CARMEL-2: A Second Generation VLSI Architecture for Flat Concurrent Prolog. In *International Conference on Fifth Gneration Computer Systems 1988*, pages 962–970, November 1988.

[8] A. Houri and E. Shapiro. *The Sequential Abstract Machine for Flat Concurrent Prolog.* CS 86-20, Weizmann Institute of Science, Applied Mathematics Department, July 1986.

[9] S. Kliger. *Towards a Native Code Compiler for Flat Concurrent Prolog.* Master's Thesis CS 87, Weizmann Institute of Science, Applied Mathematics Department, July 1987.

[10] K. Kahn S. Kliger, E. Yardeni and E. Shapiro. The language fcp(:,?). In *Proceedings of the Fifth Generation Computer Systems 1988*, pages 763–764, December 1988.

[11] S. Taylor. *Parallel Logic Programming Techniques.* Prentice Hall, 1989.

[12] E. Tick. *Performance of Parallel Logic Programming Architectures.* Technical Report TR-421, ICOT, Japan, September 1988.

[13] K. Ueda. *Guarded Horn Clauses.* Doctoral Dissertation, University of Tokyo, March 1986.

[14] A. Houri W. Silverman, M. Hirsch and E. Shapiro. *The Logix System User Manual.* Technical Report CS 21, Weizmann Institute of Science, Applied Mathematics Department, November 1988.

[15] D.H.D Warren. *An Abstract Prolog Instruction Set.* Technical Report 309, Artificial Intelligence Center, SRI International, January 1983.