Memory Performance of AND-parallel Prolog on Shared-Memory Architectures

M. Hermenegildo - MCC
E. Tick - Stanford University

Abstract

The goal of the RAP-WAM AND-parallel Prolog abstract architecture is to provide inference speeds significantly beyond those of sequential systems, while supporting Prolog semantics and preserving sequential performance and storage efficiency. This paper presents simulation results supporting these claims with special emphasis on memory performance on a two-level shared--memory multiprocessor organization. Several solutions to the cache coherency problem are analyzed. It is shown that RAP-WAM offers good locality and storage efficiency and that it can effectively take advantage of broadcast caches. It is argued that speeds in excess of 2 MLIPS on real applications exhibiting medium parallelism can be attained with current technology.

1 Introduction

The RAP-WAM execution model [10,11] is aimed at providing, through the use of parallelism, inference speeds to logic programs beyond those attainable in sequential systems, while supporting the conventional "don't know" non-deterministic semantics of logic languages. Of the various sources of parallelism present in Logic Programs [3] the RAP-WAM architecture exploits Goal Independence AND-parallelism [11], an extension of DeGroot's Restricted AND-parallelism [4] which provides backward execution semantics and improved execution graph expressions.[1] Sets of goals which are independent (i.e., which do not share any non-ground variables, determined by a combined compile-time, run-time analysis) are run in parallel. Parallelism can be programmed by the user by annotating the program with Conditional Graph Expressions (CGEs)[2] or it can be generated automatically by the compiler, through a combination of local and global (abstract interpretation-based) analysis [17] which often makes run-time independence checks unnecessary.

At the implementation level, the RAP-WAM architecture is designed to exploit both parallelism and advanced compiler technology. The techniques used for supporting parallel execution are extensions of those used in the Warren Abstract Machine (WAM)[15], which have already brought high inferencing speeds to sequential Prolog systems. Special attention is given to the preservation of WAM sequential performance and storage efficiency, and to the use of low overhead mechanisms for controlling parallel execution. Most of the WAM performance and storage optimizations are still supported during parallel execution. The CGE semantics has been integrated naturally into the WAM storage model in the form of specialized stack frames and storage areas which are used during parallel execution. Thus the default (sequential) model is that of a standard WAM exhibiting the same high sequential performance.

The RAP-WAM architecture can be viewed as a collection of abstract machines (workers) which cooperate in the execution of a program. Each of these abstract machines is similar to a standard WAM (featuring a complete set of registers and data areas, called a *Stack Set*), with the addition of a *Goal Stack* (used for on-demand scheduling), a *Message Buffer*, and two new types of stack frames: *Parcall Frames* and *Markers*. Parcall Frames coordinate and synchronize the execution of parallel goals both during forward execution and backtracking. Markers delimit *Stack Sections* (horizontal cuts through the Stack Set of a given abstract machine, corresponding to the execution of one parallel goal) and they implement the storage recovery mechanisms during backtracking[11]. In practice, the stack is divided into separate *Control* (Choice Point and Markers) and *Local* stacks (Environments) for reasons of locality and locking. Table 1 summarizes the types of objects allocated in these areas and their locality. Space limitations make a complete description of the RAP-WAM execution model impossible. The reader is referred to [11] for further details.

Frame type	area	WAM?	lock	locality
Envts./control	Stack	yes	no	Local
Envts./P. Vars.	Stack	yes	no	Global
Choice points	Stack	yes	no	Local
Heap	Heap	yes	no	Global
Trail entries	Trail	yes	no	Local
PDL entries	PDL	yes	no	Local
Parcall F./Local	Stack	no	no	Local
Parcall F./Global	Stack	no	no	Global
Parcall F./Counts	Stack	no	yes	Global
Markers	Stack	no	no	Local
Goal Frames	G. Stack	no	yes	Global
Messages	M. Buff.	no	yes	Global

Table 1: Characteristics of RAP-WAM Storage Objects

This paper presents simulation results for RAP-WAM supporting the claims of performance and efficiency. Although an evaluation of the implementation of the model on an existing shared-memory machine (Sequent) is currently also under way it only provides a single data point corresponding to a particular organization.[3] In addition, many statistics are very difficult

[1]The model is currently being extended to support OR-parallelism -using techniques similar to those proposed by other researchers, see for example [16,18] and their references- and a form of dependent AND-parallelism.

[2]CGEs offer Prolog syntax and permit conjunctive checks, thus lifting limitations in the expressions proposed by DeGroot: given "`f(X,Y,Z):- g(X,Y), h(Y,Z).`" the most natural annotation for this clause, that `g` and `h` can run in parallel if the terms in `X` and `Z` don't share variables and `Y` is bound to a ground term, can be expressed easily with CGEs ("`f(X,Y,Z):- (indep(X,Z), ground(Y) | g(X,Y) & h(Y,Z)).`") but is very difficult with DeGroot's expressions.

[3]Note also that the Balance model being used in this implementation uses write-through caches, which will be shown later in this paper to be not ideally suited for Parallel Prolog execution. Performance results from this implementation will be reported on elsewhere.

to gather from running hardware. Simulations can provide data over a wide range of architectural and organizational parameters and that is the approach taken in this study. Because high performance processing elements (PE's) are limited by available memory bandwidth (an even more important factor in parallel systems) this paper concentrates on memory performance.

The rest of the paper is organized as follows: results obtained from high-level simulations of the architecture are first summarized. A two-level shared-memory organization model and alternative solutions to the cache coherency problem are then proposed. Finally, RAP-WAM simulation results for the different coherency protocols proposed are presented and discussed.

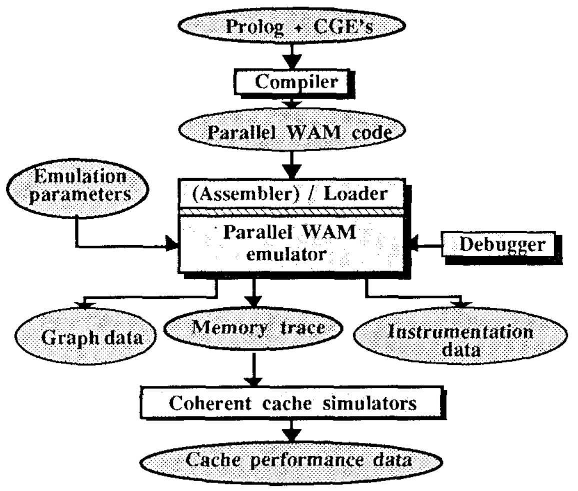

Figure 1: Simulation tools

2 Simulation Environment and High-level Results

A series of measurement tools have been built in order to evaluate the potential performance of the execution model and the associated architectural tradeoffs (Figure 1). Because the RAP-WAM model (as the WAM) is specified at a level above that of memory organization, simulations were first performed under the ideal assumption of a uniform, single shared-memory and no contention. The measurements were thereby made independent of the particular architectural organization on which the model is implemented. The emulator generated instrumentation data such as instruction frequencies, number of references classified by data areas, ratios of local vs. remote references, maximum amount of storage used per area, estimated timings, and speedups. Results from simulations at this level can be found in [12,11] and can be summarized as follows:

The overhead in the RAP-WAM model due to the management of parallelism is low: it has, for example, been observed to be in the order of 15% for up to 40 processors even for fine granularity cases (i.e., high overhead cases) such as that of the "deriv" benchmark, as shown in Figure 2. In this figure, *work* represents the number of references generated by all PEs while doing actual processing (i.e., not *waiting* or *idle*). Overhead, the difference between the work (references) done by RAP-WAM and that of WAM, is in Figure 2 the distance between the work curve and the "uniprocessor" line corresponding to WAM work. All data in this figure is presented as percentages of WAM work (executing the sequential version of the benchmark). Note that RAP-WAM work on 1 PE is very close to WAM work. Speedup (i.e. significantly faster execution than a high-performance sequential implementation -WAM- for similar performance PE's) is thus obtained even if the application exhibits only low levels of parallelism. The stack-based memory management approach[11] also appears to be very efficient recovering local storage upon procedure exit (with last call optimization) and all storage on backtracking as in the WAM.

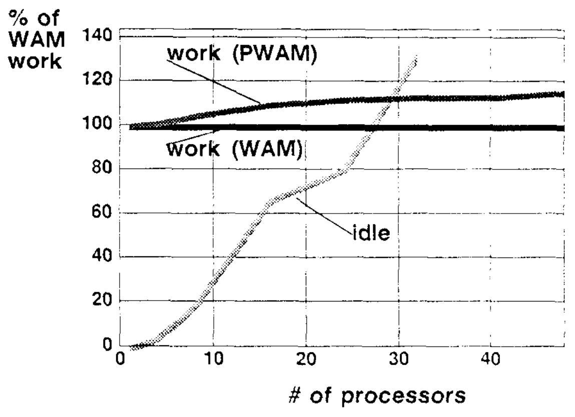

Figure 2: RAP-WAM Overheads for "deriv"

Although these results are encouraging, practical memory organizations deviate from the ideal behavior assumed above and it is thus important to assess the effect of this deviation if realistic performance figures are to be obtained. This issue is addressed in the next sections by quantifying the effect of a particular memory organization with limited bandwidths, cache coherence maintenance overhead, etc.

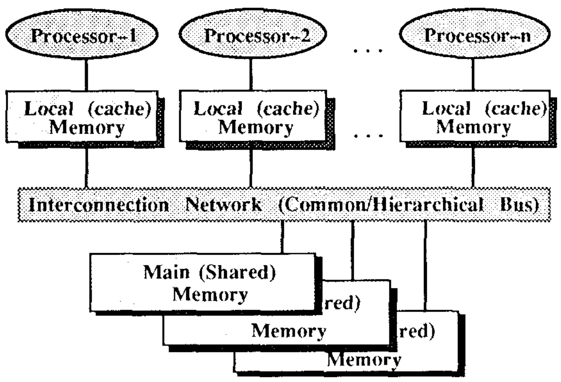

Figure 3: The Two-Level Shared-Memory Architecture Model

3 Two-Level Shared-Memory Results

Figure 3 shows a practical shared-memory system presenting a two-level structure where a local cache memory is located between each PE and the system bus. Such a hierarchical organization, characteristic of many current shared-memory multiprocessors, serves a dual purpose: first, in allowing faster execution because of the generally lower effective memory access time seen by a PE, essential in obtaining performance that is competitive with that of sequential systems. Second, in absorbing a (hopefully) significant part of the traffic to main memory which needs to go through the system bus, particularly important in shared-memory multiprocessors because the system bus is often the most significant bottleneck in the system. The locality of Prolog/WAM was shown by Tick[14]. In the next sections it is shown that Prolog/RAP-WAM also offers sufficient locality to take advantage of cache memories.

3.1 Cache Coherency

Except for simple buffers which hold only local data, most of the local memory designs used in conventional or special-purpose sequential machines for the implementation of logic programs (such as, for example, those used in [5] or those studied by Tick[14]) cannot be used directly in a parallel machine because of cache coherency problems. *Coherent caches* ensure that all the PEs in the system have a consistent view of the storage model. Although at certain times during the operation of RAP-WAM coherency is not *required*, it appears that ensuring coherency *continually* is easier than enforcing coherency only at specific points (and has the additional benefit of generality). Therefore, traditional coherent caches are considered in this study.

Historically, the first coherent caches[7], used a *write-through strategy*, where all write references were issued to both the local cache and shared memory, and copies residing on a cache other than the cache issuing the write request were invalidated. This *coherency protocol* is inexpensive in terms of hardware, but offers low performance because of excessive traffic on the system bus. Recently, a family of *fully distributed broadcast cache protocols* have been proposed and built [8,1,2] which are based on the ability of the cache organization to modify all copies of a cached item in all caches which share this item *in a single bus cycle*. Information is maintained for each cache block as to whether it is *private* or *shared*, making it possible to avoid coherency overheads for private blocks and implement write-back policies. Different designs differ essentially in the treatment of a write to a possibly shared block. A write-through broadcast strategy *updates* remote copies and possibly shared memory. A write-in broadcast strategy *invalidates* remote copies. Descriptions and measurements of the relative performance of various broadcast protocol attributes for conventional architectures are given in Archibald[1].

The broadcast protocol offers high performance at the expense of additional hardware. With the objective of reducing this expense by exploiting attributes of the RAP-WAM architecture, a *(firmware) controlled hybrid cache protocol* was developed. This scheme attempts to combine the efficiency of broadcast caches with the simplicity and low cost of a traditional write-through cache using information provided by the PE (in the form of tags, derived from the information in Table 1) as to the locality characteristics of each reference. The protocol is referred to as "hybrid" because based on these tags *potentially shared* (global) data is written-through and local data is copied back. An underlying tenet of the hybrid protocol is to avoid some of the complexity of broadcast caches by keeping shared memory consistent with local memory. The cost associated with this simplification is the traffic required to write--through to memory the write requests marked as global which are not actually shared. The gain with respect to the traditional write-through approach is that data marked as local is not written-through.

Parameter	deriv	tak	qsort	matrix
Instructions executed	33520	75254	237884	95349
References (RAP-WAM)	85477	178967	502717	96013
References (WAM)	82519	169599	499526	95357
Goals actually in //	97	263	97	24

Table 2: Statistics for the Benchmarks Used (8 processors)

cache size	E_{tr}	σ_{tr}	$(tr - E_{tr})/\sigma_{tr}$			
(words)	large bench		deriv	tak	qsort	mean
512	0.164	0.0626	1.1	-1.9	0.83	1.3
1024	0.108	0.0569	2.0	-1.1	1.6	1.6

Table 3: Fit of Small Benchmarks to Large Benchmarks

3.2 Simulations

In order to compare the performance of the various types of caches presented above, the RAP-WAM emulator was modified to generate a trace file of memory references (Figure 1). These references are marked with a *PE identifier*, a tag describing the particular storage area and object being accessed, and a read/write flag. All of the coherent cache models are simulated with the same parameterized multiprocessor cache simulator[14] which can be reconfigured to support the various consistency protocols. Caches are modeled as fully associative memories with perfect LRU replacement.

The results presented correspond to the execution of the following set of benchmarks: symbolic derivation ("deriv", which finds the symbolic derivative of a given arithmetic expression), Takeuchi ("tak", which computes Takeuchi's function), Quick--sort ("qsort", written using difference lists), and Matrix Multiplication ("matrix", a naive matrix multiplication program). Each benchmark was executed on relatively large input data. Table 2 shows some statistics regarding the benchmarks used, running on 8 PE's. Note that the number of references shows reasonable size. These benchmarks and their input data were chosen for several reasons: their small granularity (except for "matrix") provides a worst-case type of analysis with respect to parallelism management overhead. They also offer reasonable degrees of parallelism so that the parallel portion of the abstract machine is exercised. Also, their sequential memory referencing behavior and *locality* resemble those of much larger Prolog programs, such as the ones studied by Tick[14]: table 3 shows that the fit is quite good ensuring that the benchmarks exercise the sequential storage model (the foundation of the RAP-WAM storage model) in a reasonable, typical way.

Figure 4 shows the mean traffic ratios (as a function of *total* cache size and averaged over the four benchmarks) of the write-in broadcast, hybrid, and conventional write-through cache protocols, using four word lines. Caches of sizes 64, 128, and 256 words were simulated with no-write-allocate (a write miss does *not* fetch the corresponding block to cache). Caches of sizes 512 and 1024 words were simulated with write-allocate, except for hybrid caches which used no-write-allocate for 512 words. These selections were made on the basis of the policy which produced the lowest traffic. A clear result of the simulations is that no-write-allocate is best for small caches; however, miss ratio increases with no-write-allocate. Another result is that

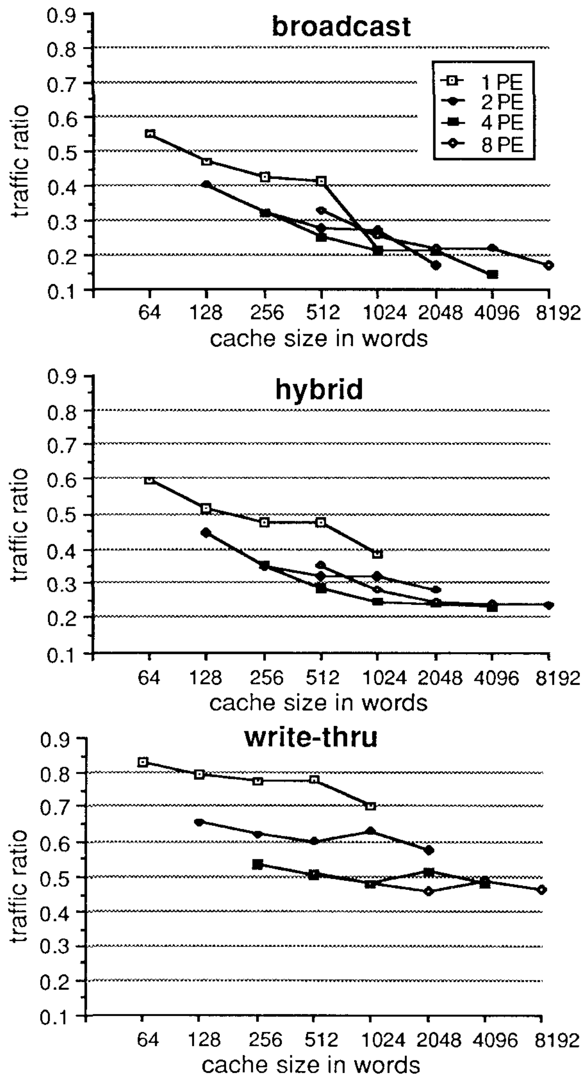

Figure 4: Traffic of Coherency Schemes

a more efficient replacement policy (e.g., copyback) produced lower traffic with write-allocate than a less efficient policy (e.g., hybrid) for the same cache size. The write-through broadcast cache statistics (not shown in Figure 4) are almost identical to those of the write-in broadcast cache, an indication that communication traffic in RAP-WAM is low.

A result seen from the curves is that the hybrid cache does quite well in reducing traffic, almost to the level of the copyback cache. The copyback cache does exceedingly well for 1024 word caches, and this trend is expected to continue with larger sizes, because the hybrid caches have already bottomed-out. The idiosyncrasies in the curves are due to the effects of averaging the benchmarks. Also, the advantageous effect (that of reducing memory traffic) of partitioning an algorithm's working set across several caches is seen to sometimes outweigh the increase in communication overheads. Lack of space makes it impossible to offer many simulation results. See [12] for more details on the benchmarks and simulations.

3.3 Discussion

As stated before, the hierarchical memory organization serves the dual purpose of lowering the effective memory access time and reducing the memory bandwidth requirement of a PE. According to the results of the simulations presented in the previous section, the hybrid cache generates an amount of traffic between that generated by the broadcast and conventional write-through caches. The broadcast schemes retain a (sometimes slight) advantage throughout the range of caches simulated.

It should be noted that these results measure performance only in terms of traffic ratio. For example, the simulation data shows that eight PEs with write-in broadcast caches (of 128 words or greater) generate a traffic ratio of less than 0.3 (the hybrid cache is also close to this performance); i.e. more than 70% of the traffic generated by the processors is captured in the local memories and will not appear on the bus. However, in order to accurately estimate the actual performance of a multiprocessor the time penalty to access shared memory due to contention must also be analyzed. Although beyond the scope of this paper a queueing model for this purpose is proposed in[14]. Results presented therein for RAP-WAM execution show that with a relatively fast bus and an interleaved memory shared memory efficiency can be high.

It is of obvious interest, if only to stimulate further research, to speculate about the potential performance levels attainable given the results presented in the previous sections. Even current low- to medium-cost shared-memory systems offer high PE to memory bandwidths by implementing multiple or overlapped busses and interleaved memories. This makes it reasonable to predict that speeds in the order of 2 million *application*[4] inferences per second are possible on shared-memory multiprocessors built using current technology.[5] A "back of the envelope" calculation, in order to justify this claim and based on the results obtained from the present and previous studies can be made as follows: studies of large Prolog benchmarks show that in the average 15 (WAM or RAP-WAM) instructions are executed per actual inference and that each instruction averages 3 (word) references. This represents 45 words/LI, or 180 bytes/LI for a 32 bit word size. Therefore, a system executing at a speed of 2 MLIPS would require a cumulative memory bandwidth of 360 Mbytes/sec. If the caches are able to capture 70% of this traffic, only 108 Mbytes/sec have to be delivered by the bus/memory system, a performance which is perfectly achievable using current off-the-shelf technology.[6]

4 Conclusions

The paper has presented memory referencing characteristics of a parallel logic programming architecture, RAP-WAM, based on Independent/Restricted AND-parallel execution of Prolog, and

[4]"Application" inferences refer to inference steps of the average size found in large Prolog programs, i.e. in the order of 15 WAM instructions. This results in much lower but more realistic figures than those obtained using the conventional "LIPS" measurement based on "naive reverse."

[5]Note that the Japanese FGCS Project is also predicting similar inferencing speeds for the PIM[9].

[6]These conclusions, although resulting from more detailed simulations than those presented in a related study by Fagin[6], are in disagreement with Fagin's results and his contention that Prolog programs cannot effectively make use of multiprocessing. The discrepancies are probably due to differences in the execution models used and to the small size of the benchmarks/data simulated by Fagin. They do agree, however, with those of Lin [13].

its behavior and potential performance on shared-memory multiprocessor organizations. The measurements presented here indicate that RAP-WAM is well-suited to high performance execution on tightly-coupled shared-memory multiprocessors, from cost-effective small-scale systems to higher-performance medium-sized systems. It has been argued that actual speeds of 2 Million *application* inferences per second are possible with currently available technology for applications which exhibit medium degrees of parallelism. It has been shown that the architecture offers high memory referencing locality so that it can take advantage of two-level memory organizations. The memory referencing study included comparison of cache coherency protocols and the "broadcast" and "hybrid" protocols were shown to offer superior performance to write-through mechanisms, present in some multiprocessors.

Because the memory organizations studied are characteristic of many current and next-generation multiprocessors, it is argued that the results obtained are relevant to the estimation of the performance of AND-parallel Prolog/RAP-WAM on them and also to determining the advantages and shortcomings of such machines in the parallel implementation of other don't--know non-deterministic logic programming languages and models. In addition, the results can also be used as a guideline in the design of small to medium-sized special purpose multiprocessors. Although the goal of small to medium systems may seem rather unambitious, it is important to have evidence of actual speedups at these levels before attempting the design of large-scale systems. In the words of the adage, "Walk before you run..."

References

[1] J. Archibald. *High Performance Cache Coherence Protocols For Shared-Bus Multiprocessors.* Technical Report 86-06-02, University of Washington, Seattle, WA 98195, June 1986.

[2] P. Bitar and A. M. Despain. Multiprocessor Cache Synchronization. In *13th Int. Symp. on Comp. Arch.*, pages 424–433, June 1986.

[3] J. S. Conery. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, Norwell, MA 02061, 1987.

[4] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478, November 1984.

[5] T. P. Dobry et. al. Performance Studies of a Prolog Machine Architecture. In *12th Int. Symp. on Comp. Arch.*, pages 180–190, December 1985.

[6] B. Fagin and A. Despain. Performance Studies of a Parallel Prolog Architecture. In *14th Annual International Symposium on Computer Architecture*, pages 108–116, IEEE Computer Society, June 1987.

[7] D. H. Gibson. Considerations in Block-Oriented Systems Design. In *AFIPS Conference Proceedings*, pages 75–80, Spring Joint Computer Conference, Academic Press, April 1967.

[8] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *10th Annual International Symposium on Computer Architecture*, pages 124–131, IEEE Computer Society, 1983.

[9] A. Goto. Parallel Inference Machine Research in FGCS Project. In *Proceedings of the First Japan-U.S. AI Symposium*, pages 21–36, December 1987.

[10] M. V. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Proceedings of the Third International Conference on Logic Programming*, pages 25–40, Springer-Verlag, 1986.

[11] M. V. Hermenegildo. *Independent AND-Parallel Prolog and its Architecture.* Kluwer Academic Publishers, Norwell, MA 02061, 1988.

[12] M. V. Hermenegildo and E. Tick. *Memory Performance of AND-Parallel Prolog on Shared-Memory Architectures.* Technical Report PP-036-88, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, January 1988.

[13] Y.-J. Lin. *A Parallel Implementation of Logic Programs.* PhD thesis, Dept. of Computer Science, University of Texas at Austin, Austin, Texas 78712, August 1988.

[14] E. Tick. *Studies In Prolog Architectures.* PhD thesis, Stanford University, Stanford, CA 94305, June 1987.

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

[16] D. H. D. Warren. The SRI Model for OR-Parallel Execution of Prolog—Abstract Design and Implementation. In *1987 Symposium on Logic Programming*, pages 92–102, IEEE Computer Society, August 1987.

[17] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, August 1988.

[18] H. Westphal and P. Robert. The PEPSys Model: Combining Backtracking, AND- and OR- Parallelism. In *Symp. of Logic Prog.*, pages 436–448, August 1987.