Performance Evaluation of Or-Parallel Logic Programming Systems on Distributed Shared-Memory Architectures*

Vanusa Menditi Calegario
Empresa Municipal de Informática S/A
Rio de Janeiro, Brazil

Inês de Castro Dutra
Department of Systems Engineering and Computer Science
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil
e-mail: {vanusa,ines}@cos.ufrj.br

Abstract. Distributed shared-memory (DSM) architectures have been object of research by many computer science groups. In this work, we investigate how DSM architectures affect performance of or-parallel logic programming systems and how this performance approaches that of conventional C systems. Our work concentrates on basic performance, scalability, and programmability.
Keywords: logic programming, conventional programming, parallelism, DSM architectures, performance evaluation

1 Introduction

Distributed shared-memory architectures have been object of research by many computer science groups. Research goes broadly from hardware based coherence protocols to DSM software protocols on networks of workstations passing through high technology inter-connection networks that reduce network latency. In this work we investigate how DSM architectures affect performance of parallel logic programming systems and compare to results obtained for parallel conventional systems.

We use execution-driven simulation of a hardware DSM (DASH [11]) to investigate the access patterns and caching behaviour exhibited by parallel C programs and by Aurora [12], a parallel logic programming system capable of exploiting implicit parallelism in Prolog programs. Aurora was written originally to run on bus-based shared-memory platforms. The simulator allows us to vary several machine settings. Our work concentrates on the study of Prolog and parallel C programs under different cache coherence protocols regarding basic performance, scalability, and programmability.

Our results indicate that cache coherence protocols do not significantly affect extremely regular applications while irregular ones can benefit from hybrid

* Research supported by CNPq and Capes, Brazilian Research Councils

or invalidate protocols. Our results also show that parallel logic programming systems written for shared-memory machines can achieve good scalability on modern architectures, but at the cost of high time and space complexity when compared to conventional C programs. However, results also indicate that optimisations such as improvements on the abstract engine and scheduling can make parallel logic programming systems competitive and more attractive than parallel C systems for some applications, specially if we consider that parallel logic programming systems exploit implicit parallelism.

Our work is the first to study the differences between parallel logic programming systems and parallel C-compatible systems. In contrast, previous works such as [16], [13], [6], [7], [5], [15], [14], [3] and [8] have also studied the performance of logic programming systems in other contexts.

The paper is organised as follows. Section 2 introduces concepts of parallel logic programming systems and briefly discusses implicit and explicit parallelism. In section 3 we describe our work methodology. Section 4 describes the applications and algorithms used and results obtained. Finally, section 5 concludes this work and draws our next steps.

2 Implicit Parallelism x Explicit Parallelism

Two main sources of implicit parallelism can be exploited in logic programming: (1) and-parallelism (at goal level) and (2) or-parallelism (at clause level). Parallel logic programming systems can exploit one or both forms of parallelism [12, 17, 9].

Each kind of parallelism has its own advantages and disadvantages, but we have chosen to focus on OR-parallelism as a first step for a number of reasons. From our perspective, the wide range of potential applications and the very small execution overhead have been the main reasons. See [12] for a fuller discussion about the advantages of OR-parallelism in the context of the Aurora system.

Because of its declarative nature, a Prolog-like program can run in parallel without any change to the Prolog source code. Logic programming runtime systems are responsible for managing and controlling the parallelism. The parallelisation mechanism is the same for any application. This approach has significant advantages when compared to explicit parallelism because the Prolog programmer does not need to worry about issues as synchronisation, communication and scheduling. Therefore the exploitation of implicit parallelism allows portability.

Some applications have very irregular computational patterns that make them hard to run efficiently on parallel machines. If they are written in C, aspects such as dynamic load balancing need to be taken into account while in parallel logic programming systems that exploit implicit parallelism this is accomplished by the runtime system.

3 Methodology

In our experiments we used a detailed on-line, execution-driven simulator that simulates a 32-node, DASH-like [11], directly-connected multiprocessor to execute Aurora, an or-parallel system, and C parallel applications. In order to keep caches coherent we used write-invalidate (WI), write-update (WU) and dynamic hybrid protocols. Detailed information about the simulator, Aurora and the protocols can be found at [2].

For WI, within a column, misses are classified as follows. **Cold start misses** happen on the first reference to a block by a processor. **True sharing misses** happen when a processor references a word belonging in a block it had previously cached but has been invalidated, due to a write by some other processor to the same word. **False sharing misses** occur in roughly the same circumstances as a true sharing miss, except that the word written by the other processor is not the same as the word missed on. **Eviction (replacement) misses** happen when a processor replaces one of its cache blocks with another one mapping to the same cache line and later needs to reload the block replaced.

The hybrid protocol includes an extra class for the miss categorisation, **drop misses**, to account for cache misses resulting from excessively eager self-invalidations.

4 Applications and Results

Matrix Multiplication: This is a typical scientific application that computes $C = A \times B$, where A, B and C are matrices and A and B are two input matrices that have the same shape and contents. We ran the programs for a 150x150 matrix size. The matrices are represented in Prolog through database facts (that gives rise to or-parallelism). The C parallel version of this program considers matrix A to be private and matrices B and C to be shared. The algorithm tries to evenly allocate chunks of the matrix A among the processors, in order that each one computes the same amount of work when possible.

N-Queens Problem: The N-queens problem is a classical combinatorial problem in the artificial intelligence area. The problem is to place N queens on an N x N chessboard so that no two queens attack each other. That is, no two queens are allowed to be placed on the same row, the same column, or the same diagonal. The algorithms used in Prolog and in C generates all solutions. Prolog implements a generate-and-test algorithm while C implements a forward checking algorithm. The Prolog program was taken from the Aurora benchmark. The C program was written by ourselves. We ran the programs for a 10x10 board size. Our C program was parallelised in the search algorithm. Each worker generates a list of children, picks the first one to continue working and puts the other children on a task queue. Once more, the original Prolog program was not modified to run in parallel.

Traveling Salesperson Problem: The traveling salesperson problem is a very popular optimisation problem that consists of visiting all nodes in a graph at most once in order to find a path with minimum cost. The C program represents the partial tours as a fibonacci heap data structure to improve performance. It uses a branch-and-bound method based on the A^* search algorithm. The Prolog program uses the same algorithm, but represents the graph as a set of facts. Besides this, the Prolog program generates the paths through intelligent permutation of the nodes in the graph using the `get_bestof/3` and `replace_bestof/3` built-in predicates available in Prolog. The C program was taken from the TreadMarks DSM System. The Prolog version was developed by ourselves. We used a 16-node graph as input.

Floor Plan Design: The problem of floor plan design consists of arranging several components on a surface in order to minimise the total area occupied by the components. In our problem, the components are restricted to be rectangular. Each component can have several different implementations. Our program solves this problem using a branch-and-bound algorithm in C and in Prolog. The number of components to be arranged is 60, with an arrangement of 14 objects on the surface.

The Prolog and C source codes for the programs can be found at the following URL: `http://www.cos.ufrj.br/~vanusa/benchs.html`.

Application	PROLOG	C	P/C
Matrix 150x150	1365311633	60522671	22.55
10-Queens	787832881	206354254	3.81
TSP 16	6228451159	5560316620	1.12
Floor Plan	8835176394	578872490	15.26

Table 1. Simulated Results - **Aurora and C** Elapsed Simulated Cycles - 1 processor

Table 1 shows the sequential execution times for the C and Aurora Prolog versions of the programs. Times are given in number of simulated cycles. The column P/C shows the ratio between Aurora and C execution times. We can observe that the C programs run from 1.12 to about 22 times faster than Aurora.

Aurora is from 1 to 5.37 times slower than one of the best sequential Prolog implementations known, SICStus 3.0 [4]. Aurora is based on SICStus 0.6 and does not generate native code. If we consider Aurora using a faster engine and generating native code, we could predict a considerable improvement in speed in the best case, which would cause Aurora to be much faster to run all applications. According to our previous work, a sequential engine can achieve significant speedups [3].

From the programs shown in table 1, matrix, a typical scientific application, is the one that has the worst results, as expected. The other applications are at most 15 times slower in Prolog than in C. This can be considered a good result, since Aurora does not use a state-of-the-art sequential engine that generates native code, and since the algorithms can be improved.

However, more important than showing that Prolog systems can perform better than conventional ones, is to show that with no effort from the Prolog programmer, we can obtain parallelism from the Prolog applications on DSM architectures and can outperform parallelised C applications.

In this section we also show results for Aurora and parallel C running the four applications with 2, 4, 8, 16, 24 and 32 processors using four cache coherence protocols (WI, WU, Hybrid 1 and Hybrid 2). It is important to notice that base times to compute speedups are different for C and Aurora. Figures were built in order to show how Aurora and C individually manage to efficiently exploit all parallelism available in the applications.

Matrix Multiplication: The matrix multiplication program written in Prolog has or-parallelism that stems from the choicepoints created for the calculation of each element of matrix C. This parallelism is implicitly and efficiently exploited by Aurora which makes the system achieve slightly better speedups than parallel C up to 24 processors. As we increase the number of processors, Aurora does not scale well up. When we observe the execution in more detail, we find out that some processors in the 32-processor simulation stay blocked much longer than the 24-processor simulation. This indicates that there is contention to access a lock with processors trying to grab work always from the same worker. This also indicates that the or-scheduler version we use for Aurora is not scalable due to the way workers try to fetch work from other workers. In contrast, the C parallel program scales well up due to the static load distribution of the matrices among the processors.

It is important to notice that the total amount of parallel work is the same as the sequential work. Therefore, the speedups correspond to actual gains in performance for efficiently utilising the resources.

It is interesting to notice that while the parallel C program has a miss rate of 2.49% with a smaller total of references to shared memory (~6M) for one processor, Aurora with a much larger total of references (~142M) has a very low miss rate running this application (0.46%) for one processor. Note that these misses correspond only to read misses and that only shared data is cached. The simulator does not cache instructions. This indicates that Aurora running the matrix application has better locality than C. This is mainly due to the design of the abstract machine used for Aurora that is based on the WAM [1] and it is optimised for locality.

Most of the C misses in this application come from eviction misses. This happens because of the memory organisation (one-way associative) and cache size (64Kbytes) that does not fit one entire matrix. The cold miss rate in the C program increases as we increase the number of processors, because matrix B needs to be loaded by every processor.

Contrasting with the very low miss rates obtained for the matrix application, the network traffic for Aurora is much larger than for the parallel C matrix program. This high network traffic for Aurora is easily explained by the high number of references to shared memory. Even so, Aurora using dynamic scheduling achieves good parallel performance that is comparable to C, which uses a static distribution of work through the processors. For being so regular, this application has almost no sharing misses, and no coherence network traffic.

The speedups were not affected by the cache coherence protocols and the same occured to miss rates. This is mainly because of the very regular nature of this kind of problem. In C, each processor takes the same kind and amount of work to compute the multiplication. In Aurora, as the matrices are represented as facts, the kind and amount of work for each processor is also basically the same. Considering network traffic, as expected, the WI protocol generates less traffic than the WU protocol for the C and Aurora programs. The hybrid protocols contributed very little to an improvement in the WU performance. As this application is very regularly distributed, a change in the protocol is hardly noticed.

N-queens: In figure 1 we can see the speedups for the 10-queens problem. This problem is very difficult to parallelise efficiently in the procedural program. In our C implementation, ten boards are initially created by the parent process. The parent process starts working with the first board created and puts the other boards on a shared task queue. Each idle process takes one board from this task queue, tries to place a queen, continues working on its own board and generates new boards to put on the task queue. The program in C uses a forward checking algorithm in order to prune the search space. The Prolog program uses a generate-and-test algorithm that is much less efficient than forward checking.

Because of the way boards are distributed through the processors, we can observe a very high rate of cold misses for the C parallel application. Boards that are created by one processor will cause multiplication of cold misses by other processors that grab those boards from the task queue. This problem could be minimised with the utilisation of local task queues per processor and some kind of affinity scheduling.

This application can only sustain parallelism in C up to 16 processors because of the way the work is distributed among the processors. The overhead of accessing the global task queue is larger than the work granularity when we reach 16 processors, which cause a degradation in parallel performance for C.

In contrast, Aurora manages to maintain performance and scales quite well. This program demonstrates that without effort from the end-programmer, we can obtain a very good performance when exploiting parallelism in logic programming.

Figure 1 also shows that the Aurora queens version can benefit from the hybrid protocols because they manage to reduce the network traffic caused by WU as can be seen in figure 2. Hyb2 seems to yield the best performance for the Aurora version with an improvement of 20.79% over the invalidate protocol for 32 processors. Aurora manages to keep performance achieving efficiency of 81%, on 32 processors, with the hybrid protocols.

TSP: Figure 3 shows the speedup curves for the tsp benchmark. The performance of the C parallel program is quite good while Aurora is not scalable.

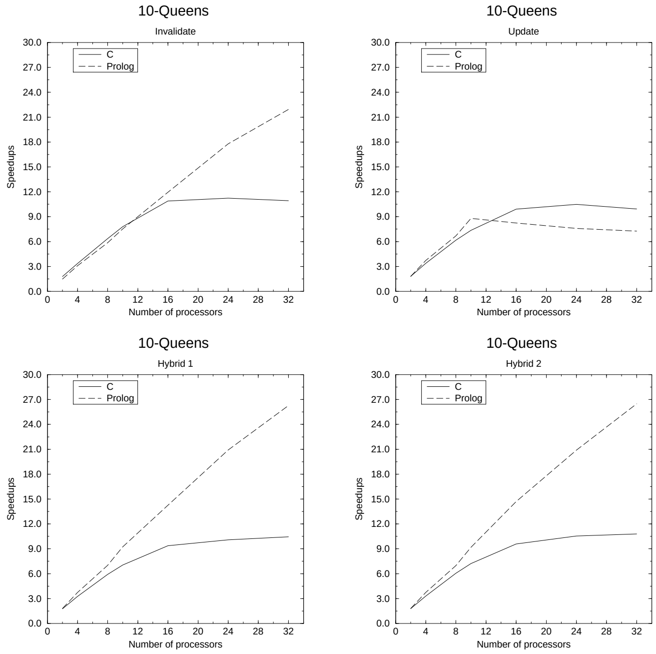

Fig. 1. Queens speedups, four protocols

It is important to notice that the parallel C application we use is well-tuned for DSM architectures.

The parallel C program shows a very low read miss rate when compared to the Prolog program. Similarly, the network traffic for this application in C is very low. This shows that this application is very well tuned for the DSM architecture we are simulating.

In figure 3, no hybrid protocol causes significant impact on the C parallel application, since this application is already well-tuned for the architecture we simulate. In contrast, the Prolog application benefits from the WU and hybrid protocols with an improvement in performance of 42% related to the invalidate protocol with 32 processors.

Floor Plan: The floor plan program in C and in Prolog were written using a branch-and-bound algorithm. These algorithms when run in parallel can produce superlinear speedups. Our speedup results indeed show that we have superlinear speedups in C and in Prolog as shown in figure 4.

The Prolog program achieves spectacular speedups while the C program fails to maintain performance after 8 processors. This is due to the fact that the Prolog program reduces dramatically the search space when running in parallel.

Miss rates and network traffic are very much influenced by the search done to find a solution, for each set of processors.

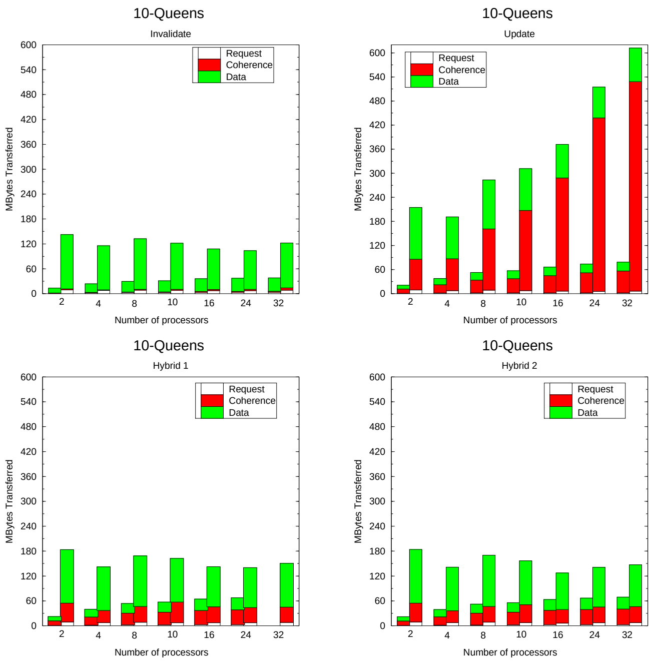

Fig. 2. Queens Network Traffic, four protocols, C (left) and Aurora (right)

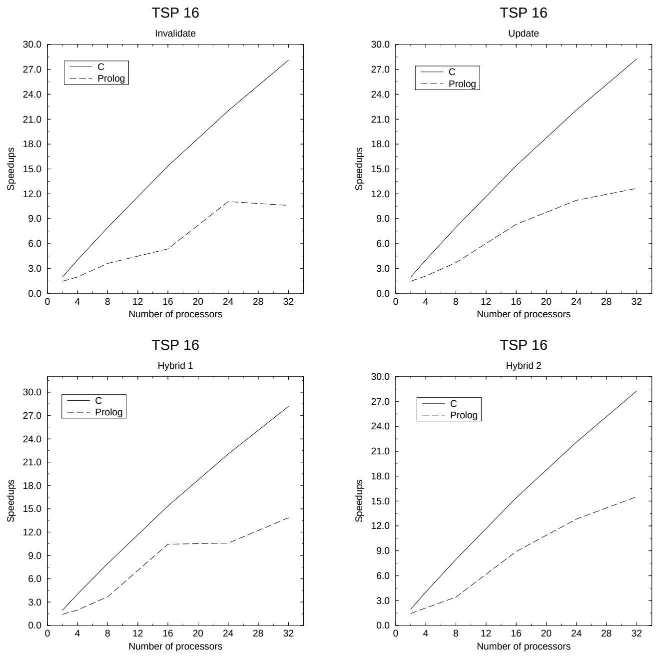

Fig. 3. TSP SPEEDUPS, FOUR PROTOCOLS

As can be also seen in figure 4, the C application has very poor performance when using any hybrid protocol. Speedups are not improved related to the WI protocol. Aurora can benefit from the hybrid protocols, managing to achieve best speedups of 150 on 32 processors, because of the reduction in search space compared to the one-processor result.

5 Conclusions and Future Work

Our results indicate that or-parallel logic programming systems can perform well on DSM architectures. More important than being able to efficiently exploiting parallelism, is the ability to do that transparently with no effort from the Prolog programmer, and with a minimum effort from the parallel programmer, i.e., migrating parallel logic programming systems originally written for shared-memory machines to distributed shared-memory architectures.

Regarding the impact of different cache coherence protocols on the Aurora performance we can conclude that they do not significantly affect extremely regular applications such as our typical scientific application, matrix multiplication, but they can significantly affect irregular ones. Particularly, hybrid protocols can improve performance up to 42% from an invalidate protocol, as our tsp application has shown.

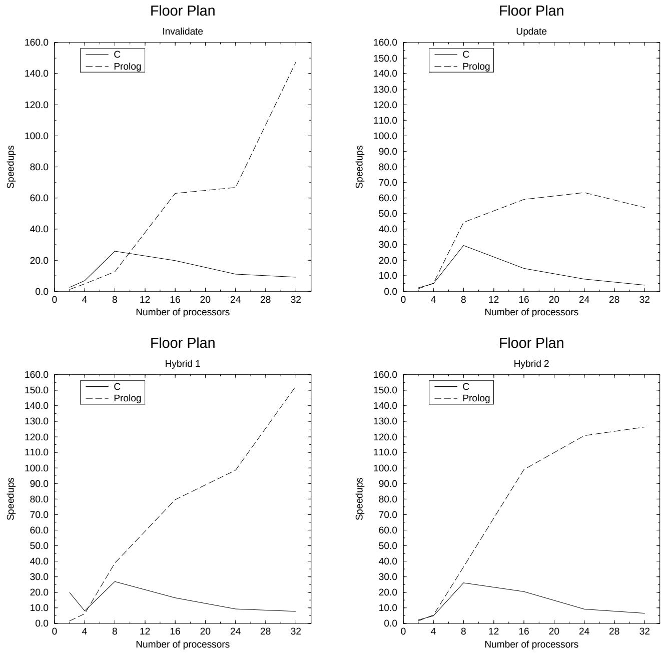

Fig. 4. FloorPlan speedups, four protocols

Results also indicate that optimisations such as improvements on the abstract engine, data organisation and scheduling can make parallel logic programming systems competitive and more attractive than parallel C systems for some applications, specially if we consider that parallel logic programming systems exploit implicit parallelism. Our results also show that although Aurora has a huge number of references to shared memory, in general, it manages to maintain good reference locality while parallel C programs with very low number of references need a good programming style for the kind of architecture we are studying.

The main problems encountered on our experiments are related to scheduling issues. Our next steps will be to use different scheduling strategies and algorithm restructuring both for the parallel C programs and for Aurora in order to improve performance. Another further step will be to evaluate other real Prolog and C symbolic applications and to test different input parameters for the applications.

Acknowledgments

We would like to thank the Brazilian Research Councils (CNPq and CAPES) for supporting this research. We would like also to thank Ricardo Bianchini and Vítor Santos Costa for their help with many technical questions. Vanusa would like to thank her bosses Sérgio Fontoura, Sérgio Abramovitch, Vânia Pintos and Volnei Marques da Costa for allowing her to carry out this research work.

References

1. H. Aït-Kaci. *Warren's Abstract Machine — A Tutorial Reconstruction.* MIT Press, 1991.
2. V. M. Calegario and I. C. Dutra. Parallel Conventional Systems versus Parallel Logic Programming Systems on Distributed Shared Memory Architectures. Technical Report ES-489/98, COPPE/Systems Engineering and Computer Science, Dezembro 1998.
3. V. M. Calegario and I. C. Dutra. Performance Comparison between Conventional and Logic Programming Systems. Technical Report ES-478/98, COPPE/Systems Engineering and Computer Science, Setembro 1998.
4. M. Carlsson and J. Widen. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, 1997. Release 3#6.
5. V. S. Costa and R. Bianchini. Optimising Parallel Logic Programming Systems for Scalable Machines. In *Proceedings of the EUROPAR'98*, pages 831–841, Sep 1998.
6. V. S. Costa, R. Bianchini, and I. C. Dutra. Evaluating the Impact of Coherence Protocols on Parallel Logic Programming Systems. In *Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed Processing*, pages 376–381, 1997. Also available as technical report ES-389/96, COPPE/Systems Engineering, May, 1996.
7. V. S. Costa, R. Bianchini, and I. C. Dutra. Parallel Logic Programming Systems on Scalable Multiprocessors. In *Proceedings of the 2nd International Symposium on Parallel Symbolic Computation, PASCO'97 [10]*, pages 58–67, July 1997.
8. V. W. Freeh. A Comparison of Implicit and Explicit Parallel Programming. *Journal of Parallel and Distributed Computing*, 1(34):50–65, 1996.
9. G. Gupta, E. Pontelli, and M. Hermenegildo. &ACE: A High Performance Parallel Prolog System. In *Proceedings of the First International Symposium on Parallel Symbolic Computation, PASCO'94*, 1994.
10. M. Hitz and E. Kaltofen, editors. *Proceedings of the Second International Symposium on Parallel Symbolic Computation, PASCO'97*, July 1997.
11. D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan 1993.
12. E. Lusk, D. H. D. Warren, S. Haridi, et al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.
13. S. Raina, D. H. D. Warren, and J. Cownie. Parallel Prolog on a Scalable Multiprocessor. In Peter Kacsuk and Michael J. Wise, editors, *Implementations of Distributed Prolog*, pages 27–44. Wiley, 1992.
14. P. V. Roy and A. M. Despain. High-Performance Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer*, 25(1):54–68, January 1992.
15. M. G. Silva, I. C. Dutra, R. Bianchini, and V. S. Costa. The Influence of Computer Architectural Parameters on Parallel Logic Programming Systems. In *Workshop on Practical Aspects of Declarative Languages (PADL'99)*, pages 122–136, January 1999. Also available as Technical Report ES/477-98, COPPE Systems Engineering, Sep/98.
16. E. Tick. *Memory Performance of Prolog Architectures.* Kluwer Academic Publishers, Norwell, MA 02061, 1987.
17. R. Yang, V. S. Costa, and D. H. D. Warren. The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 825–839. MIT Press, 1991.