STUDY OF HYBRID COHERENCE PROTOCOLS FOR PARALLEL LOGIC PROGRAMMING SYSTEMS

Eduardo P.G. de Oliveira, Fabiano Ramos, Inês de Castro Dutra
Department of Computer Science and Systems Engineering, COPPE/UFRJ
Universidade Federal do Rio de Janeiro, Brazil
{edupg,ramosf,ines}@cos.ufrj.br
Maria Clicia Stelling de Castro
Instituto de Matemática e Estatística
Universidade Estadual do Rio de Janeiro, Brazil
clicia@ime.uerj.br

ABSTRACT

This work investigates the impact of hybrid coherence protocols on non-scientific applications. We use execution-driven simulation of a scalable multiprocessor and compare the results obtained with a hybrid protocol with a well-known invalidate protocol and an update-based protocol. Our results show that for our sample programs a hybrid protocol with high threshold outperforms both the invalidate-based and update-based protocols, regardless of the type of parallelism exhibited by the benchmarks. We conclude that our applications can benefit from a hybrid protocol, and that multiprocessors designed for running these systems efficiently should adopt some form of hybrid protocol.

KEY WORDS
Hardware DSM, hybrid protocols, memory coherence protocols, parallel logic programming systems

1 Introduction

One of the most important advantages of logic programming is the availability of several forms of implicit parallelism that can be naturally exploited on shared-memory multiprocessors. These forms are present in many systems and include: or-parallelism; independent and-parallelism, dependent and-parallelism, data-parallelism, and combined and–or parallelism [9]. Most available parallel logic programming systems have been able to obtain good performance on bus-based systems, such as the Sequent Symmetry and Sun bus-based multiprocessors. Very few works have been done on implicitly exploiting parallelism on distributed environments [23]. As modern architectures are developed and the gap between CPU and memory speeds widens, the issue arises of whether the current parallel logic programming systems can also perform well on the new, scalable, architectures. In modern multiprocessors, performance depends heavily on the miss rates and may be limited by the communication overhead that is involved in sharing of writable data.

The sharing of writable data structures introduces the problem of coherence between the processors' caches. Most parallel machines have used a write invalidate (WI) protocol [8] in order to keep caches coherent. In this protocol, whenever a processor writes a data item, copies of the cache block containing the item in other processors' caches are invalidated. If one of the invalidated processors later requires the same item, it will have to fetch it from the writer's cache.

Write update (WU) protocols [14] are the main alternative to invalidate-based protocols. In WU protocols, whenever an item is written, copies of the new value are sent to the other processors that share the item.

WI protocols have been more popular than WU protocols because of the extra traffic updates introduce. Quite often the updates sent will not be used by their recipients. This introduces extra, useless, traffic that consumes bandwidth and can actually degrade performance. However, the nature of sharing in parallel logic programming systems suggests that update-based protocols might be more appropriate than their invalidate-based counterparts. In fact, in previous works, we have shown that update-based protocols outperform invalidate-based protocols in parallel logic programming systems [7, 16]. In those works we also showed that a hybrid protocol could improve performance of the applications. In this work, we study, in more detail, the same hybrid protocol used in those works, varying its threshold parameter, using more applications, and increasing the cache size, that is important for logic programming executions. We use execution-driven simulation of a scalable multiprocessor to run our applications on top of Andorra-I, a parallel logic programming system [18]. Andorra-I is an ideal subject for our experiments because it supports two rather different forms of parallelism in logic programs, and-parallelism and or-parallelism. We compare the hybrid protocol with a well-known invalidate protocol and an update-based protocol. Our results show that for our sample logic programs the hybrid protocols with higher thresholds outperform their invalidate-based and update-based counterparts. Moreover, results show pure WU performing better than WI, both for or-parallel and and-parallel benchmarks, which is an usual result within the scientific applications community. We conclude that parallel logic programming systems can benefit from hybrid protocols and that multiprocessors designed for running these systems efficiently should adopt some form of hybrid protocol.

Our approach contrasts with previous studies of the performance of coherence protocols for parallel logic programming systems. Tick and Hermenegildo [21] studied caching behaviour of independent and-parallelism in bus-based multiprocessors. Other researchers have studied the performance of parallel logic programming systems on scalable architectures, such as the DDM [15], but did not evaluate the impact of different coherence protocols. We studied the performance of the same hybrid protocol presented here with a fixed threshold and with a smaller cache.

The paper is organised as follows. Section 2.1 presents the methodology used to obtain our results. Section 2.2 describes the Andorra-I parallel logic programming system. Section 3 presents speedup results for the or-parallel, and-parallel, and combined parallel benchmarks ran in Andorra-I under the hybrid protocol with different thresholds, and compare it with WI and WU. Finally, Section 4 draws our conclusions and suggests future work.

2 Methodology

In this section we detail the methodology used in our experiments. The experiments consisted of the simulation of the parallel execution of Andorra-I, compiled for the MIPS architecture [10].

2.1 Multiprocessor Simulation

We use a detailed on-line, execution-driven simulator that simulates a 24-node, DASH-like [12], directly-connected multiprocessor. Each node of the simulated machine contains a single processor, a write buffer, cache memory, local memory, a full-map directory, and a network interface. The simulator was developed at the University of Rochester and uses the MINT front-end [22] (developed by Veenstra and Fowler) to simulate the MIPS architecture, and a back-end [5] (developed by Bianchini, Kontothanassis, and Veenstra) to simulate the memory and interconnection systems.

In our simulated machine, each processor has a 512-KB direct-mapped data cache with 64-byte cache blocks. All instructions and read hits are assumed to take 1 cycle. Read misses stall the processor until the read request is satisfied. Writes go into a 16-entry write buffer and take 1 cycle, unless the write buffer is full, in which case the processor stalls until an entry becomes free. Reads are allowed to bypass writes that are queued in the write buffers. Shared data are interleaved across the memories at the block level.

The interconnection network is a bi-directional wormhole-routed mesh, with dimension-ordered routing. The network clock speed is the same as the processor clock speed. Switch nodes introduce a 4-cycle delay to the header of each message. Network paths are 16-bit wide, which matches the memory bandwidth. In these networks contention for links and buffers is captured at the source and destination of messages.

The hybrid protocol used in our experiments [11] is based on the coherence protocols of the bus-based multiprocessors using the DEC Alpha AXP21064 [20]. In these multiprocessors, each node makes a local decision to invalidate or update a cache block when it sees an update transaction on the bus. We associate a counter with each cache block and invalidate the block when the counter reaches the threshold. References to a cache block reset the counter to zero. In our experiments we varied the threshold with values 1, 2, 4 or 8 updates. This protocol should reduce the number of useless updates significantly, but may also increase the read miss rate. Whether the hybrid protocol can improve overall performance depends on the relative impact of an increase in miss rate versus a significant reduction in update messages.

The WI protocol used to compare with the hybrid keeps caches coherent using the DASH protocol with release consistency [12]. In our WU implementation, a processor writes through its cache to the home node. The home node sends updates to the other processors sharing the cache block, and a message to the writing processor containing the number of acknowledgements to expect. Sharing processors update their caches and send an acknowledgement to the writing processor. The writing processor only stalls waiting for acknowledgements at a lock release point.

2.2 Andorra-I

The Andorra-I parallel logic programming system is based on the Basic Andorra Model. The system was developed at the University of Bristol by Beaumont, Dutra, Santos Costa, Yang, and Warren [18]. To the best of the authors' knowledge, Andorra-I was the first parallel logic programming system that exploited both and- and or-parallelism, and yet could run real-world applications with significant parallel performance. This is the main motivation for using this system in our experiments.

The Andorra-I system consists of several components. The preprocessor is responsible for compiling the program and for the sequencing information necessary to maintain the correct execution of Prolog programs. The engine is responsible for the execution of the Andorra-I programs. The two schedulers manage and- and or-work. The reconfigurer allows workers to migrate between teams to find better sources of work.

Most of the execution time of workers should be spent executing *engine* code [25], i.e., performing reductions. Andorra-I is designed in such a way that data corresponding to each worker is as local as possible, so that each worker tries to find its own work without interfering with others. Scheduling in Andorra-I is demand-driven, that is, whenever a worker runs out of work, it enters a *scheduler* to find another piece of available work.

The or-scheduler is responsible for finding or-work, i.e., an unexplored alternative in the or-tree. Our experiments used the Bristol or-scheduler [3], originally developed for Aurora [13].

The and-scheduler is responsible for finding eligible and-work, which corresponds to a goal in the run queue (list of goals not yet executed) of a worker in the same team. Each worker in a team keeps a run queue of goals. This run queue of goals has two pointers. The pointer to the head of the queue is only used by the owner. The pointer to the tail of the queue is used by other workers to "steal" goals when their own run queues are empty. If all the run queues are empty, the slaves wait either until some other worker (in our implementation, the master) creates more work in its run queue or until the master detects that there are no more determinate goals to be reduced and it is time to create a choicepoint.

2.3 The MIPS Port

In order to use Andorra-I with the simulator we needed to port the system to the MIPS architecture. We used the FSF's `gcc 2.7.2` C compiler and `binutils-2.6` assembler and linker under a Solaris 2.4 environments as cross development tools for this purpose. Andorra-I was compiled with `-O2`.

Most of the port was straightforward. The only difficulties arose with shared memory allocation and locking. For shared memory allocation we use the `shmalloc` library supported by MINT. For locking in modern MIPS machines, we would use the `ll`, `sc`, and `sync` machine instructions [10] to implement locks and atomic operations. Unfortunately, these instructions are not yet supported by the back-end. The alternative is to use the lock library routines, as implemented by the simulator, which allow us to control the synchronisation overhead. In our simulations, these routines were implemented as atomic instructions.

To ensure correct execution under the release consistency model, we guarantee that all accesses to shared data are surrounded by lock and unlock operations. The only exception to this rule is the detection of the end of the determinate phase. Here we maintained the original protocol because any action after detection requires the slaves to grab a lock previously released by the master.

3 Application Performance

In this section we present speedup results for Andorra-I under the protocols, WI (inv), WU (upd) and hybrid (hyb) with thresholds 1, 2, 4 and 8 updates. We experimented with applications representing predominantly and-parallelism, or-parallelism, and both and- and or-parallelism. The benchmarks represent real applications used by companies or in academia. We tried to select applications with good parallelism, whilst avoiding applications with too much parallelism that would scale well regardless of architecture.

We obtain times for the *first* run of an application (results would be somewhat better for other runs).

All speedup figures are shown for 1, 2, 4 and 8 processors.

3.1 And-Parallel Applications

We use two example and-parallel applications, the clustering algorithm for network management from British Telecom, `bt-cluster`, and a program to calculate approximate solutions to the traveling salesperson problem, `tsp`. Both programs have very good and-parallelism, and, being completely determinate, no or-parallelism. It is known from a related work that on an idealised shared-memory machine, where data items can always be found in cache, `bt-cluster` has excellent and-parallelism and can achieve almost linear speedups up to twenty four processors. The traveling salesperson program is based on a Reform Prolog [4] benchmark that finds an approximate solution for the TSP problem in a graph with 24 nodes. performance, the Andorra-I team

3.2 Or-Parallel Applications

As our or-parallel application we use an example from the well-known natural language question-answering system `chat-80`, written at the University of Edinburgh by Pereira and Warren [24]. This version of `chat-80` operates on the domain of world geography. The program `chat` makes queries to the `chat-80` database. This is a small scale benchmark with good or-parallelism, and it has been traditionally used as one of the or-parallel benchmarks for both the Aurora and Muse [1] systems.

3.3 And/Or-Parallel Applications

We used a program to generate naval flight allocations, based on a system developed by Software Sciences and the University of Leeds for the Royal Navy. It is an example of a real-life resource allocation problem. The program allocates airborne resources (such as aircraft) whilst taking into account a number of constraints. The problem is solved by using the technique of active constraints as first implemented for Pandora [2]. The program has both or-parallelism, arising from the different possible choices, and and-parallelism, arising from the parallel evaluation of different constraints. The input data we used for testing the program consists of 11 aircrafts, 36 crew members and 10 flights needed to be scheduled. The degree of and- and or-parallelism in this program varies according to the queries, but all queries give rise to more and-parallelism than or-parallelism.

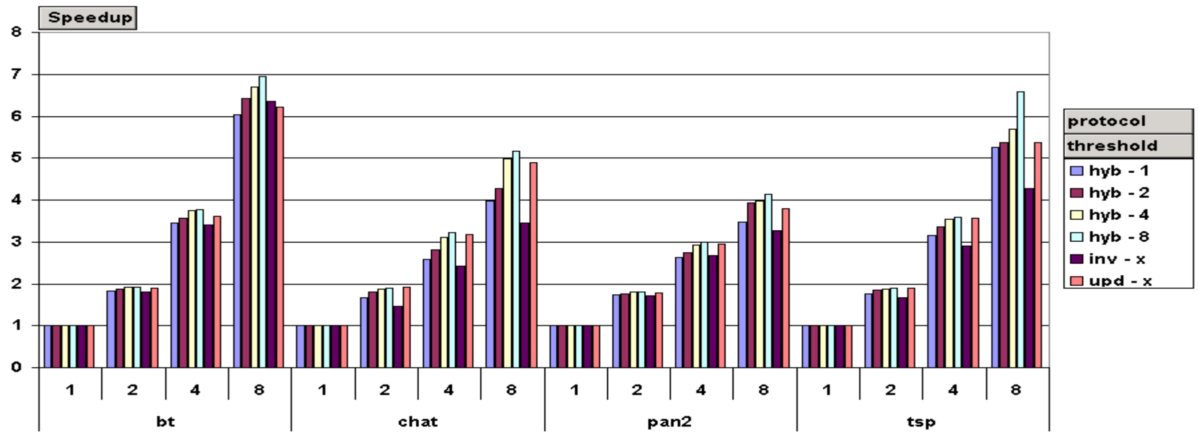

Figure 1. Speedup for all applications, all protocols, varying numbers of processors

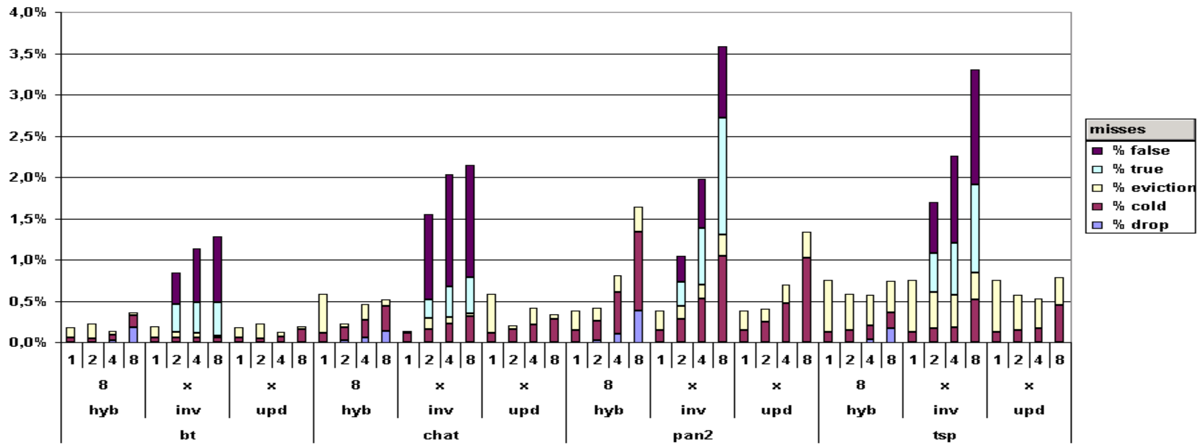

Figure 2. Read miss rate under WI, WU, and hybrid with threshold 8

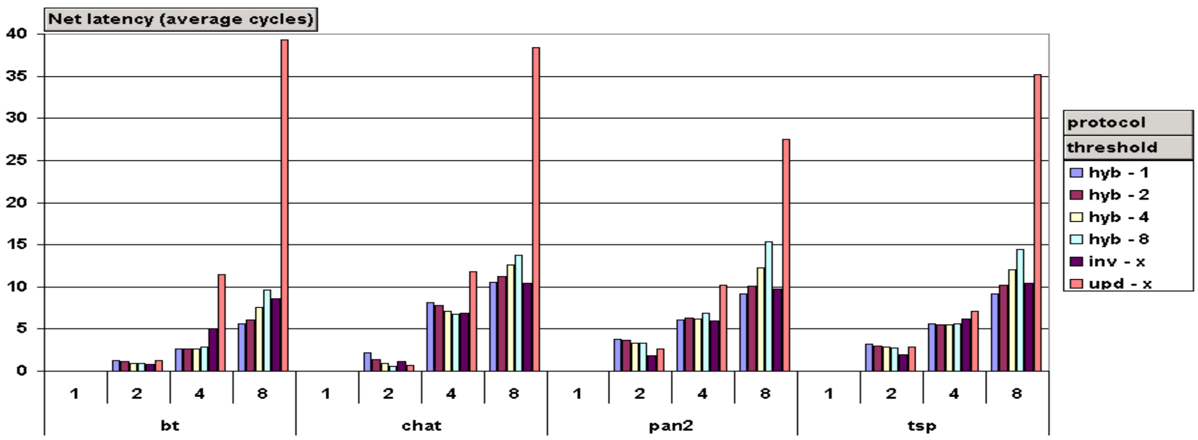

Figure 3. Network Latency for WI, WU and hybrid

3.4 Analysis of Results

Figure 1 shows the speedups for all protocols: hybrid (hyb), with varying threshold (1, 2, 4 and 8 updates), WI (inv), and WU (upd), for all applications, and varying numbers of processors (1, 2, 4, and 8). The best speedups for high numbers of processors, for all applications were produced by the hybrid protocol with threshold of 8. In a previous work, using only three of these applications, and a smaller cache size, we had used threshold of 2 updates and already obtained a good result, but not much better than the speedups produced by the invalidate and update-based protocols. These results clearly show that the combination of a bigger cache with a hybrid protocol using higher thresholds can further improve performance.

For 8 processors, and the hybrid protocol with threshold 8, we obtained almost 10% of improvement, for `bt-cluster`, related to the invalidated-based protocol, and more than 11% of improvement related to the update-based protocol. For `chat` the improvement was even higher: almost 50% of improvement related to the invalidate-based protocol, and almost 6% related to the update-based protocol. For `pan2` the gain was almost 26% related to WI, and almost 9% related to WU. `tsp` is the application that benefits more from the hybrid-8 protocol. It improved almost 54% related to the WI protocol, and almost 23% related to the WU protocol.

It is important to note that the hybrid protocol was the best for all applications with high numbers of processors, but the WU protocol also outperformed the WI protocol, which is not a very common result for scientific applications.

It is interesting to point out that speedups improve as we increase the threshold parameter. This indicates that Andorra-I, no matter which kind of parallelism is being exploited, and-, or-, or a combination of both (meaning that different data structures of the system are being used for each kind of parallelism), needs to have data updated for some interval of time before invalidating them. This suggests that cache coherence protocols for these kinds of systems may rely on time intervals to keep updated values for different data areas, and perhaps be implemented by dynamically varying the threshold, instead of having it static.

According to these results, the WU protocol invariably performs better than the WI protocol and hybrid protocols that use smaller thresholds. As we increase the number of processors, performance of the WI and WU protocols starts to degrade related to the hybrid protocols. Figures 2, and 3 help to understand this phenomenon. Figure 2 shows the shared read miss rates for the four applications. For WI, within a column, misses are classified as: **Cold start misses.** A cold start miss happens on the first reference to a block by a processor; **True sharing misses.** A true sharing miss happens 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.** A false sharing miss occurs 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; and **Eviction misses.** An eviction (replacement) miss happens 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.

This classification uses the algorithm described in [6], as extended in [5]. Note that the WU protocol does not have sharing misses.

In the figures, this cache miss categorisation includes an extra class, `drop` misses, to account for cache misses resulting from excessively eager self-invalidations in the hybrid protocols.

Among the four protocols, hybrid with threshold 8 is the one that has lower shared miss rates, for all numbers of processors and applications (Figure 2), even having an increase in network traffic (Figure 3), when compared to the invalidate-based protocol. This shows that there is a good trade-off between the number of useless update messages reduced by the protocol, and the shared miss rate. Figure 3 also shows that WU produced a great amount of network traffic, but, even so, it outperformed the speedups of the WI protocol.

Compared to our previous results [16, 7], where we studied only the hybrid protocol with threshold 2, speedups improved. This happened because of the increase in cache size that allowed more data to be in cache and, consequent reduction of the eviction miss rate.

4 Conclusions and Future Works

In this work we investigated an alternative hybrid protocol and studied its behaviour comparing its performance with invalidate and update-based protocols. We performed this study in the context of a non-scientific medium-to-large parallel application. We show that these kinds of parallel applications exhibit an execution pattern that is fairly different from scientific workloads, and may require memory coherence protocols different from the ones that normally work well for scientific applications.

Our results show more than 50% improvement in speedups, for 8 processors, when using a hybrid protocol with threshold of 8 updates.

One alternative to improve the performance of such applications is through tuning and optimising the parallel code. Previous work was done with this same application, running the same four benchmarks with a smaller cache size, with excellent results [7, 17]. In this work we decided to maintain the application code untouched and investigated if a different architecture design would improve performance. In another related work, we investigated the best architectural parameters for these kinds of applications [19], but did not study the impact of memory coherence protocols.

As future work we intend to vary more simulator parameters, and investigate how different data areas of our parallel program are affected by the different protocols. Work is under way to implement an adaptive protocol, that can dynamically vary the threshold according to the sharing pattern of the applications.

We would also like to study the scalability of the hybrid protocol. In this study we were not able to run more than 8 processors because of lack of memory in our Sun machines.

References

[1] K. A. M. Ali and R. Karlsson. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[2] R. Bahgat. Solving Resource Allocation Problems in Pandora. Technical report, Imperial College, Department of Computing, 1990.

[3] A. Beaumont, S. Muthu Raman, and P. Szeredi. Flexible Scheduling of Or-Parallelism in Aurora: The Bristol Scheduler. In Aarts, E. H. L. and van Leeuwen, J. and Rem, M., editor, *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991. Lecture Notes in Computer Science 506.

[4] J. Bevemyr, T. Lindgren, and H. Millroth. Reform Prolog: The Language and its Implementation. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 283–298. MIT Press, June 1993.

[5] R. Bianchini and L. I. Kontothanassis. Algorithms for Categorizing Multiprocessor Communication Under Invalidate and Update-Based Coherence Protocols. In *Proceedings of the 28th Annual Simulation Symposium*, April 1995.

[6] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom. The detection and elimination of useless misses in multiprocessors. In *Proceedings of the 20th ISCA*, pages 88–97, May 1993.

[7] I. C. Dutra, V. Santos Costa, and R. Bianchini. The Impact of Cache Coherence Protocols on Parallel Logic Programming Systems. In *International Conference on Computational Logic*, volume 1861, pages 1285–1299, 2000. Lecture Notes in Artificial Intelligence.

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

[9] G. Gupta. *Multiprocessor Execution of Logic Programs*. Kluwer Academic Press, 1994.

[10] G. Kane and J. Heinrich. *MIPS RISC Architecture*. Prentice Hall, 1992.

[11] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:79–119, 1988.

[12] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. *Proceedings of the 17th ISCA*, pages 148–159, May 1990.

[13] 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.

[14] E. M. McCreight. The Dragon Computer System, an Early Overview. In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, July 1984.

[15] 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.

[16] V. Santos 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.

[17] V. Santos Costa, R. Bianchini, and I. C. Dutra. Parallel Logic Programming Systems on Scalable Architectures. *Journal of Parallel and Distributed Computing*, 60(7):835–852, July 2000. http://www.idealibrary.com/links/toc/jpdc/60/7/0.

[18] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[19] M. G. Silva, I. C. Dutra, R. Bianchini, and V. Santos Costa. The Influence of Computer Architectural Parameters on Parallel Logic Programming Systems. In Gopal Gupta, editor, *Workshop on Practical Aspects of Declarative Languages (PADL99), Lecture Notes in Computer Science*, volume 1551, pages 122–136, January 1999.

[20] C. P. Thacker, D. G. Conroy, and L. C. Stewart. The alpha demonstration unit: A high-performance multiprocessor for software and chip development. *Digital Technical Journal*, 4(4):51–65, 1992.

[21] E. Tick. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers, Norwell, MA 02061, 1987.

[22] J. E. Veenstra and R. J. Fowler. MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, 1994.

[23] K. Villaverde, E. Pontelli, H-F. Guo, and G. Gupta. A methodology for order-sensitive execution of non-deterministic languages on beowulf platforms. In *9th International Euro-Par Conference, Parallel Processing*, Lecture Notes in Computer Science, pages 694–703. Springer-Verlag, August 2003.

[24] D. H. D. Warren and F. C. N. Pereira. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. Technical Note, Dept of AI, University of Edinburgh, 1981.

[25] R. Yang, V. Santos 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.