The Impact of Cache Coherence Protocols on Parallel Logic Programming Systems*

Inês de Castro Dutra†, Vítor Santos Costa†, and Ricardo Bianchini†‡

†COPPE Systems Engineering
Federal University of Rio de Janeiro
Rio de Janeiro, Brazil 21945-970

‡Department of Computer Science
Rutgers University
Piscataway, NJ 08854-8019

e-mail: {*ines,vitor,ricardo*}*@cos.ufrj.br*

Abstract. In this paper we use execution-driven simulation of a scalable multiprocessor to evaluate the performance of the Andorra-I parallel logic programming system under invalidate and update-based protocols. We use two versions of Andorra-I. One of them was originally designed for bus-based multiprocessors, while the other is optimised for scalable architectures. We study a well-known invalidate protocol and two different update-based protocols. Our results show that for our sample logic programs the update-based protocols outperform their invalidate-based counterpart for the original version of Andorra-I. In contrast, the optimised version of Andorra-I benefits the most from the invalidate-based protocol, but a hybrid update-based protocol performs as well as the invalidate protocol in most cases. We conclude that parallel logic programming systems can consistently benefit from hybrid update-based protocols.
Keywords: logic programming, parallelism, cache coherence protocols, DSM architectures, performance evaluation

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 include: or-parallelism, as exploited in the systems Aurora [18] and Muse [1]; independent and-parallelism, as in &-Prolog [12] and &-ACE [11]; dependent and-parallelism, as in Parlog's JAM [6], KLIC [27], and DDAS [26]; data-parallelism, as in Reform Prolog [4]; and combined and–or parallelism, as in Andorra-I [25], Penny [20], and ACE [10]. All these systems have been able to obtain good performance on early bus-based systems, such as the Sequent Symmetry multiprocessors.

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 writable data.

* This work was sponsored by CNPq, Brazilian Research Council.

Sharing in parallel logic programming systems occurs under several circumstances. The use of logical variables for communication in dependent and-parallel applications, for instance, is an example of producer–consumer sharing of data. A second major form of sharing, migratory sharing, arises from synchronisation between processors. Synchronisation occurs in tasks such as fetching work from other processors, and on being the leftmost goal or branch to execute cuts or side-effects.

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 [9] 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 (re)fetch it from the writer's cache.

Write update (WU) protocols [19] 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, so that it does not have to be (re)fetched on a later access.

The tradeoff between WI and WU protocols is then clear: WI protocols usually involve more cache misses but less communication than WU protocols. Thus, the performance of parallel logic programming systems on modern multiprocessors will depend heavily on the sharing behavior of these systems and how well it matches the underlying cache coherence protocol. We are interested in exactly this interplay between sharing behavior and coherence protocol.

To address this issue, we use execution-driven simulation of a scalable multiprocessor running the Andorra-I parallel logic programming system [25]. We simulate three different coherence protocols: a well-known invalidate protocol and two update-based protocols. 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 experiment with two versions of Andorra-I. One of them was originally designed for bus-based multiprocessors, while the other has been optimised for scalable architectures [22].

Our results show that for our sample logic programs the update-based protocols outperform their invalidate-based counterpart for the original version of Andorra-I. The detailed analysis of these results shows a hybrid form of WU performing better than WI both for or-parallel and and-parallel benchmarks. In contrast, we find that the optimised version of Andorra-I benefits the most from the invalidate-based protocol, but the hybrid protocol performs as well as the invalidate protocol in most cases. We conclude that parallel logic programming systems can consistently benefit from the hybrid update-based protocol and that multiprocessors designed for running these systems efficiently should adopt some form of this protocol.

Our approach contrasts with previous studies of the performance of coherence protocols for parallel logic programming systems. Tick and Hermenegildo [29] 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 [21], but did not evaluate the impact of different coherence protocols. Our initial work studied the impact of different cache coherence protocols for the original, non-optimised, Andorra-I version, using a limited benchmark set and a relatively small data cache [23]. We next performed a detailed analysis of the WI protocol based on the Andorra-I data areas [24], which led us to proposing optimisations that improved the speedup of the WI protocol by more than 100% for some applications [22]. In this study we address the question of how the original and the optimised versions of Andorra-I perform under the various coherence protocols. We use a set of benchmarks that covers both and-parallelism and or-parallelism, with examples of scalable and non-scalable applications.

The remainder of the paper is organised as follows. Section 2 describes the Andorra-I parallel logic programming system and the techniques we used to optimise its performance for scalable architectures. Section 3 presents the methodology used to obtain our results. Section 4 describes our benchmark set. Section 5 presents speedup results for the or-parallel, and-parallel, and combined parallel benchmarks ran on both versions of Andorra-I under all protocols, and evaluates the performance of the protocols. Finally, section 6 draws our conclusions and suggests future work.

2 Andorra-I

2.1 Overview

The Andorra-I parallel logic programming system is based on the Basic Andorra Model [32]. The system was developed at the University of Bristol by Beaumont, Dutra, Santos Costa, Yang, and Warren [25, 34]. 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.

Andorra-I employs a very interesting method for exploiting and-parallelism in logic programs, namely to execute *determinate* goals first and concurrently, where determinate goals are the ones that match at most one clause in a program. Thus, Andorra-I exploits determinate dependent and-parallelism. Eager execution of determinate goals can result in a reduced search space, because unnecessary choicepoints are eliminated. The Andorra-I system also exploits or-parallelism that arises from the non-determinate goals. Its implementation is influenced by JAM [7] when exploiting and-parallelism, and by Aurora [18] when exploiting or-parallelism.

A processing element that performs computation in Andorra-I is called a *worker*. In practice, each worker corresponds to a separate processor. Andorra-I is designed in such a way that workers are classified into *masters* and *slaves*. One master and zero or more slaves form a *team*. Each master in a team is responsible for creating a new choicepoint, while slaves are managed and synchronised by their master. Workers in a team cooperate with each other in order to share available and-work. Different teams of workers cooperate to share or-work. Note that workers arranged in teams share the same set of variables used in a given branch of the search tree.

Most of the execution time of workers should be spent executing *engine* code [35], 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. The strategy used by the Bristol scheduler concentrates on mandatory work, choosing work from the richest worker. If there is no mandatory work available, speculative work is chosen in a leftmost order.

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.

Workers can migrate between teams and change from master to slave and vice-versa through the reconfigurer [8].

2.2 Two versions of Andorra-I

We used in our experiments two versions of Andorra-I. The first was originally written for early bus-based machines, while the second has been optimised for modern scalable architectures.

The second version of Andorra-I optimises the original version of the system using the following 5 techniques: (1) trimming of shared variables (removal of all variables from the source code that are not relevant to the main execution of the programs and that generate a surprising amount of read misses, e.g., statistical shared counters and debugging variables), (2) data layout modification (this optimisation includes mainly padding and field reordering in order to reduce the amount of false sharing on some data areas in Andorra-I), (3) privatisation of shared data structures, (4) lock distribution, and (5) elimination of locking in scheduling. Shared variable trimming and the modification of the data layout produced the greatest improvements under a WI protocol. These optimisations were performed after a careful analysis of the data areas in the Andorra-I execution [22].

3 Methodology

We use a detailed on-line, execution-driven simulator that simulates a 24-node, DASH-like [17], 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 [30] to simulate the MIPS architecture, and a back-end [5] to simulate the memory and interconnection systems.

We simulate a WI protocol and two different WU-based protocols. Our WI protocol keeps caches coherent using the DASH protocol with release consistency [17]. 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 acknowledgments to expect. Sharing processors update their caches and send an acknowledgment to the writing processor. The writing processor only stalls waiting for acknowledgments at a lock release point.

Our WU implementation includes two optimisations. First, when the home node receives an update for a block that is only cached by the updating processor, the acknowledgment of the update instructs the processor to retain future updates since the data is effectively private. Second, when a parallel process is created by *fork*, we flush the cache of the parent's processor, which eliminates useless updates of data initialised by the parent but not subsequently needed by it.

In order to reduce the number of update messages of the WU protocol, we experiment with a dynamic hybrid protocol (WUh2) [15] based on the coherence protocols of the bus-based multiprocessors using the DEC Alpha AXP21064 [28]. 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. We used counters with a threshold of 2 updates.

4 Applications

4.1 And-Parallel Applications

We used two examples of 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`. The clustering program receives a set of points in a three dimensional space and groups these points into *clusters*. Basically, three points belong to the same cluster if the distance between them is smaller than a certain limit. And-parallelism in this case naturally stems from running the calculations for each point in parallel. The test program uses a cluster of 400 points as input data. This program has very good and-parallelism, and, being completely determinate, no or-parallelism. 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. To obtain best performance, the Andorra-I team rewrote the original applications to make them determinate-only computations.

4.2 Or-Parallel Applications

We use two or-parallel applications. Our first application, `chat80`, is an example from the well-known natural language question-answering system `chat-80`, written at the University of Edinburgh by Pereira and Warren [33]. This version of `chat-80` operates on the domain of world geography. The program `chat80` 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 systems. The second application, `floorplan`, is an example query for a knowledge-based system for the automatic generation of floor plans [16]. The input to this program is a set of desired rooms and their sizes with restrictions on windows facing north, south, etc. We ran this application for 9 different rooms with different restrictions. This application should at least in principle have significant or-parallelism.

4.3 And/Or-Parallel Applications

We used a program (called `flypan`) 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]. In this technique, the co-routining inherent in the Andorra model is used to activate constraints as soon as possible. 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.

5 Results: The Impact of Different Coherence Protocols

In this section we present speedup results for Andorra-I under the three protocols, WI, WU and WUh2. We present results for the two versions of Andorra-I: `orig`, the original version, and `optim`, the version optimised for a scalable architecture.

All results, except for the or-parallel applications, were obtained using the reconfigurer to automatically adapt to the available parallelism. The or-parallel applications used a fixed all-masters configuration. Also, we obtained times for the *first* run of an application (results would be somewhat better for other runs).

5.1 The `orig` version

Figures 1, 2, 3, 4, and 5 show the speedups for all protocols with the Andorra-I `orig` version. We can observe that the hybrid protocol becomes the best for all applications as we increase the size of the multiprocessor up to 24 processors.

Among the and-parallel applications, `bt-cluster` exhibits excellent and-parallelism, resulting in fairly good speedups under `WUh2`. WI achieves its best speedup of 11.3 with 24 processors, while WU achieves its best speedup of 8.5 with 16 processors. WUh2 achieves its best speedup of 15.9 for 24 processors yielding an improvement of 40% over the WI protocol and 97% over the WU protocol.

Figure 2 shows that `tsp` achieves worse speedups than `bt-cluster`. The main reason for this result is the sharing behaviour of these applications. Under the WI protocol and 16 processors, for instance, `tsp` issues 26 million references to shared memory with a cache miss rate of 3%, while `bt-cluster` issues 27 million references to shared memory with a cache miss rate of only 1.6%.

It is interesting to observe that the performance of the WU protocol is better than that of the WI protocol for the `tsp` application, but not for the `bt-cluster` application. Recall that pure WU can reduce the miss rate with respect to WI but can also generate heavy coherence traffic in doing so. For `tsp`, WU reduces the miss rate substantially without a significant increase in network traffic. More specifically, the miss rate of `tsp` is reduced from 3% under WI to 0.7% under WU on 16 processors, while the corresponding amount of network traffic increases from 148 MBytes to only 180 MBytes (a 22% increase), respectively. The WU protocol does not behave as nicely for `bt-cluster`. For this application, the miss rate is reduced from 1.6% under WI to 0.9% under WU again on 16 processors, while the network traffic increases from 71 MBytes to 313 MBytes (a 4-fold increase), respectively.

The performance of the or-parallel applications is similar, with `floorplan` (figure 5) achieving better performance than `chat-80` (figure 4). WU again performs better than WI for the or-parallel applications, `chat-80` and `floorplan`. The reason for the better performance of the WU protocol for these applications is that most sharing misses are due to false sharing and WU usually performs better for this kind of sharing pattern.

The application `flypan` is not much affected by any protocol. As can be observed from figure 3, this application achieves maximum speedup of 5 for 16 and 24 processors. Its execution is dominated by parallel overheads. In particular, `flypan` spends a significant amount of time in the and-scheduler trying to find available and-work. This causes a degradation in performance as we increase the number of processors, which does not allow the application to scale well.

WUh2 is the best protocol for the original version of Andorra-I. The overall improvements of WUh2 over the WI and WU protocols on 24 processors are shown in table 1. In the worst case, WUh2 is 5% better than WI and 16% better than WU. In the best case, WUh2 is nearly 100% better than WU and 88% better than WI.

Fig. 1. Speedup of `bt-cluster` on the original system

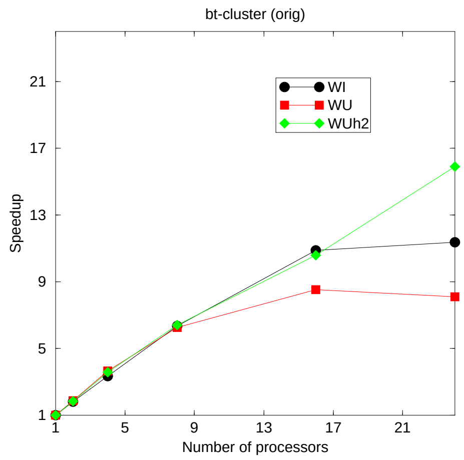


Fig. 2. Speedup of `tsp` on the original system

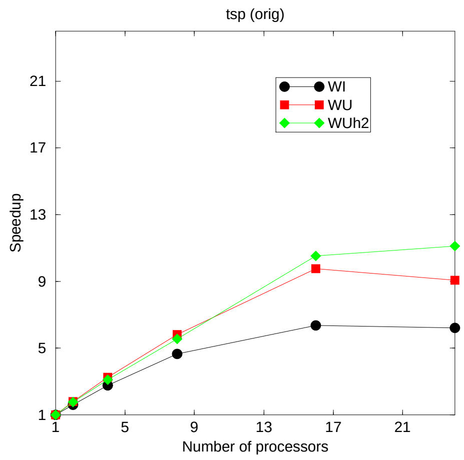


Fig. 3. Speedup of `pan2` on the original system

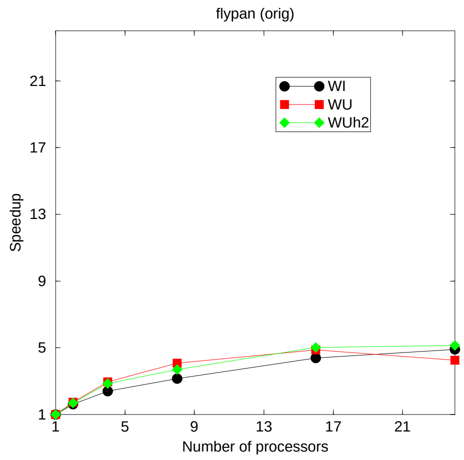


Fig. 4. Speedup of `chat-80` on the original system

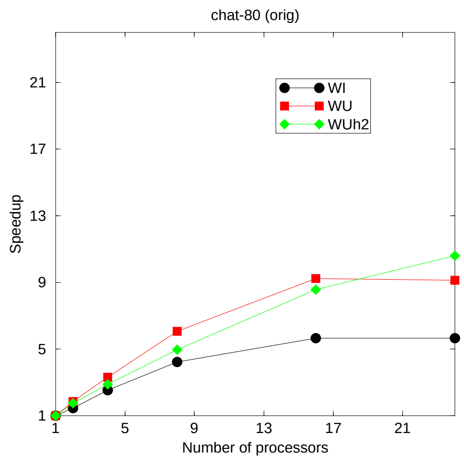


Fig. 5. Speedup of `floorplan` on the original system

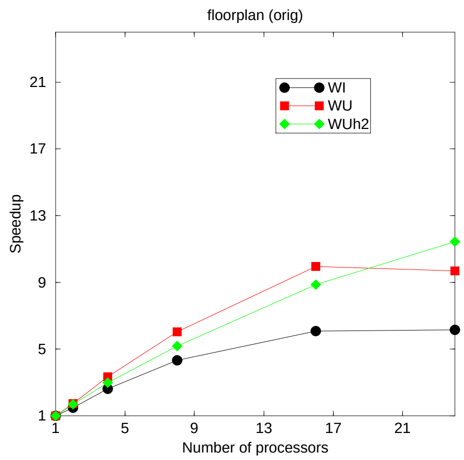


Table 1. Improvements of WUh2 over WI and WU

Applications	WUh2/WI	WUh2/WU
`bt-cluster`	1.40	1.97
`tsp`	1.79	1.23
`flypan`	1.05	1.21
`chat-80`	1.88	1.16
`floorplan`	1.86	1.18

5.2 The `optim` version

Figures 6, 7, 8, 9, and 10 show the speedups for all protocols with the Andorra-I `optim` version.

The optimised version of Andorra-I resulted in lower execution times than the original code for all applications. As discussed in section 2.2 the major differences between the optimised and original versions of the system stem from trimming shared variables and from data layout modifications that reduced false sharing at the cost of increasing memory usage (e.g., by using padding). The first optimisation was particularly effective for or-parallel benchmarks and benefits all protocols. The second optimisation was designed for a WI protocol. WU does not benefit as much from this latter optimisation because it suffers less with false sharing than WI and the extra memory usage sometimes results in more data and coherence messages sent to the network (since cache block-based coalescing of messages becomes ineffective).

Even though the execution times of the optimised system are lower, the speedups it achieves not always improve upon those of the original system. More specifically, the speedups of the `bt-cluster`, `tsp`, and `flypan` applications under the WU protocol on 24 processors decreased significantly. Under the other protocols, the speedup of the optimised system is consistently higher than that of the original version of Andorra-I, except for the `flypan` application for which WI and WUh2 exhibit roughly unchanged speedups. `Tsp` is another interesting application in terms of speedup. For this application, both WI and WUh2 saturate at a speedup of 12 on 16 processors, while the speedup of these protocols increases up to 24 processors for all the other applications.

Overall, the best performance is now obtained with the WI protocol, since the optimised version of Andorra-I almost completely eliminates false sharing. Nevertheless, the performance of WUh2 is virtually indistinguishable from that of WI for all but one application, `bt-cluster`. For this application, WI reaches a speedup of 20 on 24 processors, while WUh2 performs about 25% worse. Out of all applications, the best performance is obtained by the or-parallel benchmark `floorplan`, where speedups are almost linear for all protocols (20-fold speedup for 24 processors). Performance is also quite good for `chat-80` under the WI and WUh2 protocols, as both protocols obtain a speedup of 18 on 24 processors.

6 Conclusions and Future Work

We studied the performance of the Andorra-I parallel logic programming system under different cache coherence protocols for distributed shared-memory machines. We experimented with two versions of Andorra-I: the original system which was designed for early bus-based machines and a version that was an optimised for scalable multiprocessors with a write invalidate protocol. Our results show that a hybrid update-based protocol can obtain good performance for both versions of Andorra-I, while the other protocols fail to achieve consistently good performance.

Fig. 6. Speedup of `bt-cluster` on the optimised system

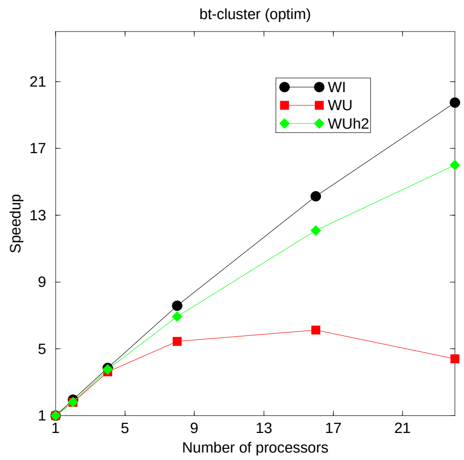


Fig. 7. Speedup of `tsp` on the optimised system

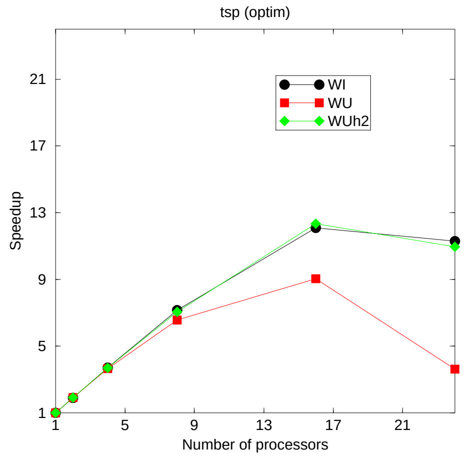


Fig. 8. Speedup of `pan2` on the optimised system

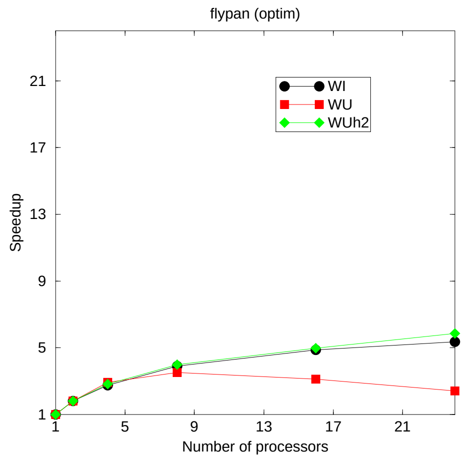


Fig. 9. Speedup of `chat-80` on the optimised system

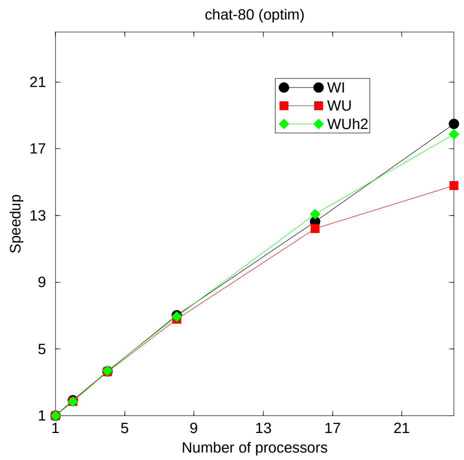
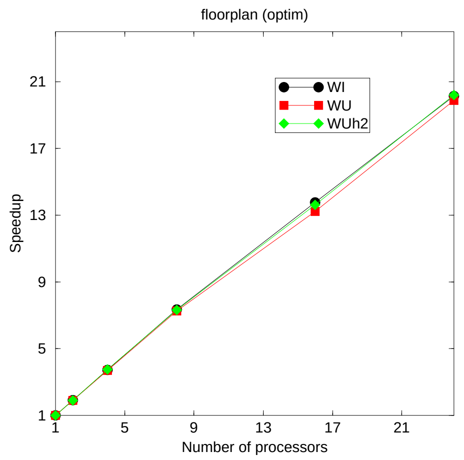


Fig. 10. Speedup of `floorplan` on the optimised system



The hybrid protocol is quite effective for the original system. This is because this protocol is not as vulnerable to false sharing as write invalidate and does not generate as much network traffic as pure write update. The hybrid protocol also benefits from optimisations performed for an invalidate-based machine.

The write invalidate protocol performs well as long as there is little false sharing. To obtain good performance it is therefore important to study the memory access patterns of the application and how they can generate false sharing. This requires a very detailed understanding of the interplay between parallel application, `C` compiler, and parallel architecture. We performed such a study in [22].

The write update protocol's main advantage is that it is not as vulnerable to false sharing as the write invalidate protocol. Unfortunately, Andorra-I is a memory intensive system and can saturate the network under this protocol. This problem is most remarkable in the optimised version, because padding was used to reduce false sharing.

We will be performing a similar analysis for other parallel logic programming systems. Besides confirming the generality of our claims, such an analysis will give us further insight into the current performance and scalability of parallel logic programming systems.

Acknowledgments

The authors would like to thank Leonidas Kontothanassis and Jack Veenstra for their help with the simulation infrastructure used in this paper. The authors would also like to thank Rong Yang, Tony Beaumont, D. H. D. Warren for their work in Andorra-I that made this work possible. We would like to thank Dr. László Kovács who kindly sent us his source code of the `floorplan` program used in this work. Vítor Santos Costa and Inês Dutra would like to thank the CNPq CLoP^n project.

References

1. Khayri A. M. Ali and Roland 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. Reem Bahgat. Solving Resource Allocation Problems in Pandora. Technical report, Imperial College, Department of Computing, 1990.
3. Anthony Beaumont, S. Muthu Raman, and Péter 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. Johan Bevemyr, Thomas Lindgren, and Häkan 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. J. A. Crammond. *Implementation of Committed Choice Logic Languages on Shared Memory Multiprocessors.* PhD thesis, Heriot-Watt University, Edinburgh, May 1988. Research Report PAR 88/4, Dept. of Computing, Imperial College, London.
7. J. A. Crammond. The Abstract Machine and Implementation of Parallel Parlog. Technical report, Dept. of Computing, Imperial College, London, June 1990.
8. I. C. Dutra. *Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System.* PhD thesis, University of Bristol, Department of Computer Science, February 1995. available at http://www.cos.ufrj.br/~ines.
9. James R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131, 1983.
10. Gopal Gupta, M. V. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proceedings of the Eleventh International Conference on Logic Programming*, Italy, June 1994.
11. Gopal Gupta, Enrico Pontelli, and Manuel Hermenegildo. &ACE: A High Performance Parallel Prolog System. In *Proceedings of the First International Symposium on Parallel Symbolic Computation, PASCO'94*, 1994.
12. M. V. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.
13. Markus Hitz and Erich Kaltofen, editors. *Proceedings of the Second International Symposium on Parallel Symbolic Computation, PASCO'97*, July 1997.
14. Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall, 1992.
15. A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator. Competitive snoopy caching. *Algorithmica*, 3:79–119, 1988.
16. L. Kovács. An Incremental Prolog System development for the Floor Plan Design by Dissecting. In *Proceedings of the International Conference on Practical Application of Prolog*, volume 2, 1992.
17. 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.
18. Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.
19. E. M. McCreight. The Dragon Computer System, an Early Overview. In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, July 1984.
20. Johan Montelius. Penny, A Parallel Implementation of AKL. In *ILPS'94 Post-Conference Workshop in Design and Implementation of Parallel Logic Programming Systems, Ithaca, NY, USA*, November 1994.
21. 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.
22. V. Santos Costa and R. Bianchini. Optimising Parallel Logic Programming Systems for Scalable Machines. In *Proceedings of the EUROPAR'98*, pages 831–841, Sep 1998.
23. 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.

24. V. Santos 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 [13]*, pages 58–67, July 1997.
25. 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.
26. Kish Shen. *Studies of And/Or Parallelism in Prolog*. PhD thesis, Computer Laboratory, University of Cambridge, 1992.
27. T. Chikayama, T. and Fujise, and H. Yashiro. A Portable and Reasonably Efficient Implementation of KL1. In *Proceedings of the Eleventh International Conference on Logic Programming*, June 1993.
28. Charles P. Thacker, David G. Conroy, and Lawrence C. Stewart. The alpha demonstration unit: A high-performance multiprocessor for software and chip development. *Digital Technical Journal*, 4(4):51–65, 1992.
29. Evan Tick. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers, Norwell, MA 02061, 1987.
30. 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.
31. David H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues. In *Proceedings of the 1987 International Logic Programming Symposium*, pages 92–102, 1987.
32. David H. D. Warren. The Andorra model. Presented at Gigalips Project workshop, University of Manchester, March 1988.
33. David H. D. Warren and Fernando C. N. Pereira. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. Technical Note, Dept of AI, University of Edinburgh, 1981.
34. Rong Yang, Tony Beaumont, Inês Dutra, Vítor Santos Costa, and David H. D. Warren. Performance of the Compiler-Based Andorra-I System. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 150–166. MIT Press, June 1993.
35. Rong Yang, Vítor Santos Costa, and David 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.