The Influence of Architectural Parameters on the Performance of Parallel Logic Programming Systems

Marcio G. Silva, Inês C. Dutra, and Ricardo Bianchini*
Vítor Santos Costa**

* COPPE/Systems Engineering
Federal University of Rio de Janeiro, Brazil
{mgs,ines,ricardo}@cos.ufrj.br

** LIACC and DCC-FCUP
4150 Porto, Portugal
vsc@ncc.up.pt

Abstract. In this work we investigate how different machine settings for a hardware Distributed Shared Memory (DSM) architecture affect the performance of parallel logic programming (PLP) systems. We use execution-driven simulation of a DASH-like multiprocessor to study the impact of the cache block size, the cache size, the network bandwidth, the write buffer size, and the coherence protocol on the performance of Andorra-I, a PLP system capable of exploiting implicit parallelism in Prolog programs. Among several other observations, we find that PLP systems favour small cache blocks regardless of the coherence protocol, while they favour large cache sizes only in the case of invalidate-based coherence. We conclude that the cache block size, the cache size, the network bandwidth, and the coherence protocol have a significant impact on the performance, while the size of the write buffer is somewhat irrelevant.

Keywords: DSM architectures, performance evaluation, logic programming

1 Introduction

Parallel logic programming (PLP) systems are sophisticated examples of symbolic computing systems. They address problems such as dynamic memory allocation, scheduling irregular execution patterns, and managing different types of implicit parallelism. Most PLP systems have been developed for bus-based shared-memory architectures. Their complexity and the large amount of data they process raise the question of whether PLP systems can still obtain good performance on scalable architectures, such as hardware distributed shared-memory (DSM) systems.

Our previous work [16, 15, 17] has shown that PLP systems as originally implemented do not perform well on scalable DSM multiprocessors; PLP systems only perform well for these machines when heavily modified to improve locality of reference. We focused on Andorra-I [18], a PLP system capable of exploiting implicit parallelism in Prolog programs, that was originally developed for the Sequent Symmetry bus-based multiprocessor. Our work has shown that Andorra-I requires several data structure modifications to run well on a DASH-like multiprocessor. However, our work has not addressed how different architectural parameters affect the performance of PLP systems on scalable multiprocessors. Thus, in this work we investigate the impact of the cache block size, the cache size, the network bandwidth, the processor's write buffer size, and the coherence protocol on the performance of the modified version of Andorra-I, again running on a simulated DASH-like multiprocessor.

Among several other observations, we find that Andorra-I favours small cache blocks regardless of the coherence protocol, while it favours large cache sizes only in the case of invalidate-based coherence. In addition, Andorra-I can profit from high network bandwidth, especially when update-based coherence is used. Based on our results, we conclude that Andorra-I can benefit from large cache sizes, small to medium cache blocks, and high network bandwidth. The depth of the write buffer does not seem to affect performance significantly.

Our work contrasts with similar studies by other researchers. Montelius [13] and Tick and Hermenegildo [19], for instance, studied the performance of PLP systems implemented for bus-based multiprocessors, while we are more interested on how these implementations run on a scalable multiprocessor and the modifications they require in order to perform well. Other researchers have studied the performance of PLP systems on scalable architectures, such as the DDM [14], but did not evaluate the impact of different machine settings and different coherence protocols.

The remainder of this paper is organised as follows. Section 2 describes our methodology and benchmarks. Section 3 discusses our results. Section 4 draws our conclusions and directions for future work.

2 Methodology

2.1 Andorra-I

Andorra-I is a parallel logic programming system that exploits determinate dependent and-parallelism and or-parallelism arising from non-determinate goals. Its implementation is influenced by JAM [3] when exploiting and-parallelism, and by Aurora [11] when exploiting or-parallelism.

In Andorra-I, a processing element that performs computation 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 as *masters* or *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.

Workers should perform reductions most of the execution time, i.e., they should spend most of their time executing *engine* code. Andorra-I is designed so 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 [1], originally developed for Aurora. 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. The reconfigurer is responsible for moving slaves and masters between teams when they do not find any more work on their own teams [5].

The version of Andorra-I used for our experiments is an optimised version that employs some data structure-level techniques (such as padding, alignment, and lock distribution) to perform well on scalable DSM machines [17].

2.2 Multiprocessor Simulation

We use a detailed on-line, execution-driven simulator that simulates a 32-node, DASH-like [10], directly-connected multiprocessor. Each node of the simulated machine contains a single processor, a write buffer, a direct-mapped data cache, 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, developed by Veenstra and Fowler [20], to simulate the MIPS architecture, and a back-end, developed by Bianchini and Veenstra, to simulate the memory and interconnection systems.

In our simulated machine, all instructions and read hits are assumed to take 1 processor cycle. Read misses stall the processor until the read request is satisfied. Writes go into a 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.

A memory bus clocked at half of the speed of the processor connects the main components of each machine node. A new bus operation can start every 34 processor cycles. A memory module can provide the first word of a cache line 20 processor cycles after the request is issued. The other words are delivered at 2 cycles/word bandwidth.

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. In these networks contention for links and buffers is captured at the source and destination of messages. All our architectural parameters are compatible with those of scalable multiprocessors such as DASH.

In order to keep caches coherent we used write-invalidate (WI) [7] and write-update (WU) [12] protocols. In the WI 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. Our WI protocol keeps caches coherent using the DASH protocol with release consistency [9].

WU protocols are the main alternative to invalidate-based protocols. In WU protocols, whenever an item is written, the writer sends copies of the new value to the other processors that share the item. 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.

2.3 Applications

The benchmarks we used in this work are applications representing predominantly and-parallelism, predominantly or-parallelism, and both and- and or-parallelism.

All results, except for the and/or-parallel application, were obtained using a static configuration of workers. The or-parallel applications used a fixed all-masters configuration and the and-parallel applications used a fixed one-team of one master and slaves. Our results correspond to the *first* run of an application (results would be somewhat better for other runs).

And-Parallel Application: As an example and-parallel application, we used the clustering algorithm for network management from British Telecom. The 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 application naturally stems from running the calculations for each point in parallel. The test program uses a cluster of 400 points as input data.

Or-Parallel Applications: We used two programs as or-parallel applications. The first is an example from the well-known natural language question-answering system `chat-80`, written at the University of Edinburgh by Pereira and Warren. 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 systems.

The second program solves the popular N-queens problem. Given a chessboard NxN, the problem consists of placing all queens on the chessboard in a way that no queen can attack each other in a row, column or diagonal. This program has a high degree of or-parallelism and is also used as benchmark for both the Aurora and Muse systems.

And/Or-Parallel Application: 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. The program has both or-parallelism and and-parallelism. The input data we used for testing the program consists of 11 aircrafts, 36 crew members, and 10 flights to be scheduled. The degree of and- and or-parallelism in this program varies according to the queries, but the query used produces more and-parallelism than or-parallelism.

More details and references for the applications can be found elsewhere [6].

2.4 Read Misses Categorisation

For each application, performance is directly related to cache read miss rates and network traffic caused by each change in the machine setting and in the coherence protocol.

For the WI protocol, the simulator classifies misses 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.
- **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 [4], as extended in [2] by Bianchini and Kontothanassis. Note that the WU protocol does not involve sharing misses.

We classify updates as either *useful* (also known as **true sharing updates**), which are needed for correct execution of the program, or *useless*. The former category is used when the receiving processor references the word modified by the update message before another update message to the same word is received. The latter category of updates includes [2]:

- **False sharing updates.** The receiving processor does not reference the word modified by the update message before it is overwritten by a subsequent update, but references some other word in the same cache block.

- **Proliferation updates.** The receiving processor does not reference any of the words in the cache block until the next update for the same word.
- **Replacement updates.** The receiving processor does not reference the updated word until the block is replaced in its cache.
- **Termination updates.** A termination update is a proliferation update happening at the end of the program.

3 Results and Analysis

We simulate Andorra-I for all applications assuming 16 processors. For each application we vary cache sizes, cache block sizes, number of write buffer entries, network bandwidth, and coherence protocol. We vary each of these parameters at a time, keeping our default values for the other parameters (64-byte blocks, 64-KByte caches, network link bandwidth of 200 MBytes/sec, and 16-entry write buffers).

3.1 Varying Cache Block Sizes

Figure 1 shows execution times (in processor cycles) for our applications using WI and WU protocols. First of all, we can observe from the figure that the WI protocol is clearly the best for the applications with and-parallelism, while the influence of the protocol is not as significant for the or-parallel applications up to 128-byte blocks.

We can also observe that in general, small cache blocks (16, 32) yield better performance than larger cache blocks. In the best case, small block sizes can speedup the application as significantly as 5-fold with respect to 256-byte blocks. This is the case of the `chat-80` application under the WU protocol, which achieves its fastest execution with 64-byte blocks (1.75 million cycles) and slowest execution with 256-byte blocks (9.44 million cycles).

Part of the explanation for these results comes from the miss rates entailed by these block sizes and protocols. Figure 2 categorizes cache misses as a function of the block size and coherence protocol. The figure shows that the number of cold misses decreases as we increase the block size, as one would expect, since larger blocks transfer more data per miss into the caches. However, cold misses is the only category that consistently decreases with increases in block size. 256-byte blocks, for instance, increase the miss rate of all applications (with respect to 128-byte blocks) under the WI protocol, while they only increase the miss rate of the applications with and-parallelism under WU.

The main problem here is that larger blocks often increase the number of cache evictions and cause significant false sharing. Note however that false sharing manifests itself differently, depending on the coherence protocol. Under WI, false sharing increases the number of coherence (false) misses, while under WU false sharing simply increases the amount of coherence traffic (false and proliferation updates). These increases can be seen in figures 3 and 4.

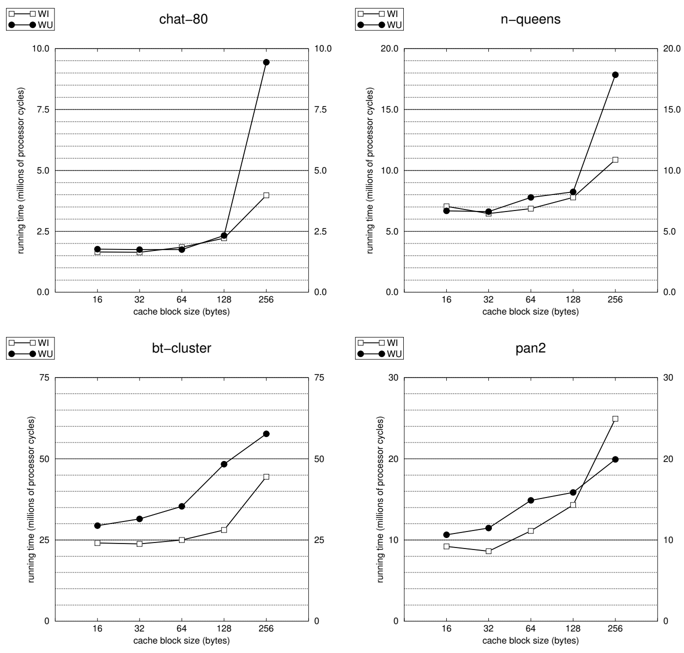

Fig. 1. Varying cache block size, execution times

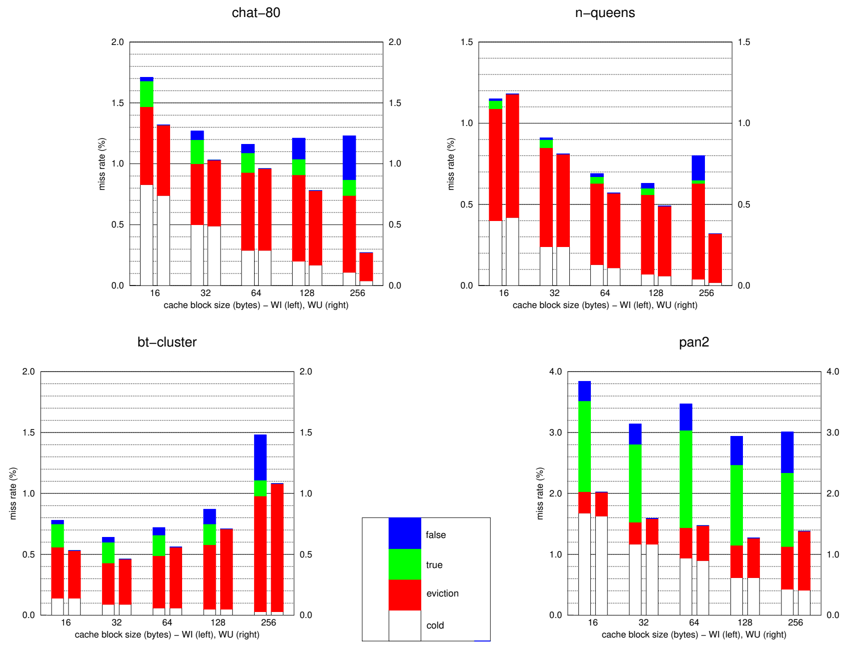

Fig. 2. Varying cache block size, miss rate

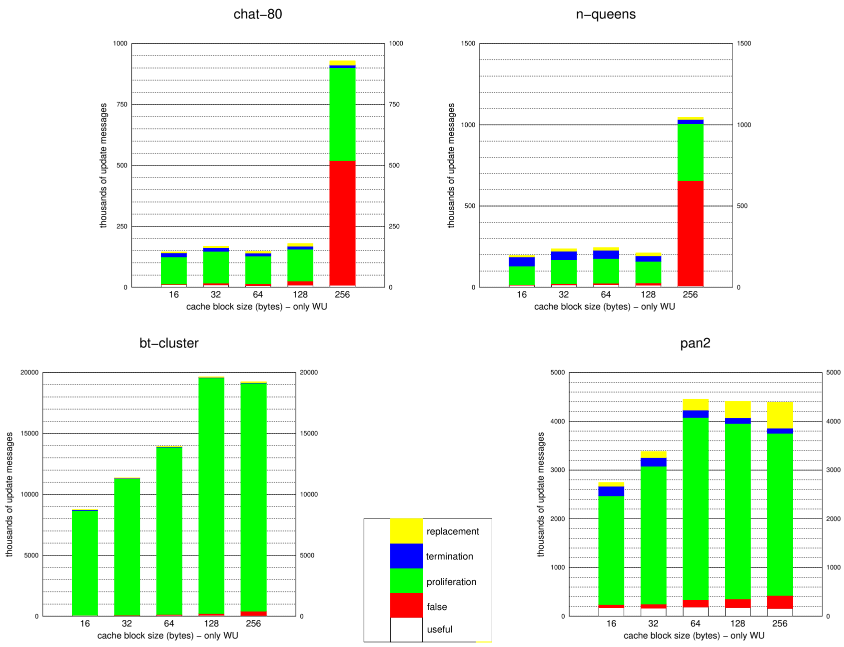

Fig. 3. Varying cache block size, categorising update msgs

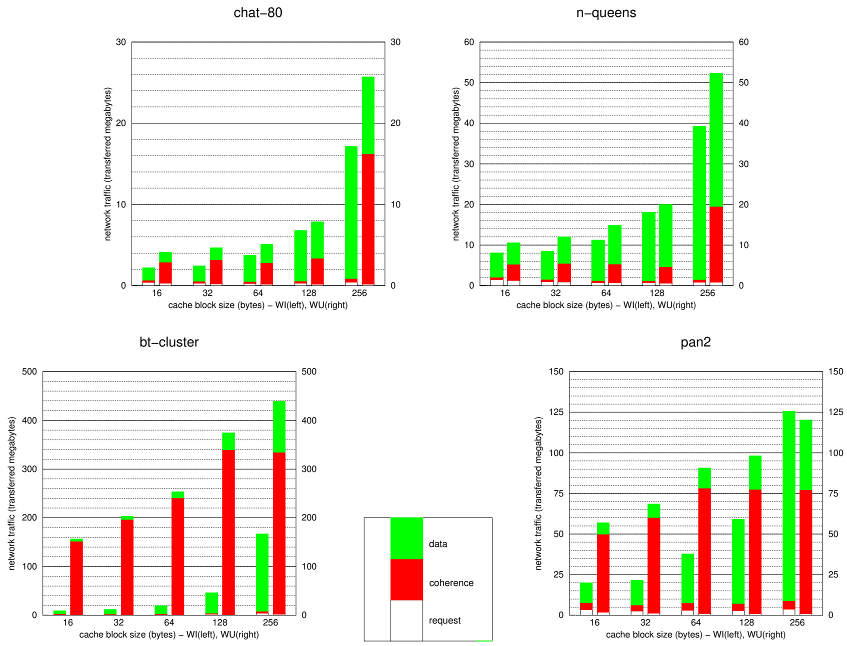

Fig. 4. Varying cache block size, network traffic

Figure 3 categorizes the update messages involved in each application as a function of block size. The figure explains the poor behaviour of the or-parallel applications under 256-byte blocks and the WU protocol. False sharing seriously increases the amount of coherence traffic in these applications, particularly for 256-byte blocks. This false sharing comes from the or-scheduler data structures used to manage the choicepoint stack. The applications with and-parallelism communicate much more intensively than the or-parallel applications under WU, but do not suffer as severely from false sharing.

The growth in the number of eviction and false misses and useless update messages generates the network traffic displayed in figure 4. Under the WI protocol, the larger the cache block, the larger the data messages being exchanged to service cache misses. Under WU, the network traffic grows significantly when compared to the WI traffic. As the block size increases, the amount of traffic, mainly coherence traffic, also grows. For the two or-parallel applications the traffic grows so intense that the network saturates. An experiment with chat-80 using 256-byte blocks and assuming infinite network bandwidth confirms this by decreasing the execution time from about 9 million cycles (shown in figure 1) to 3.7 million cycles.

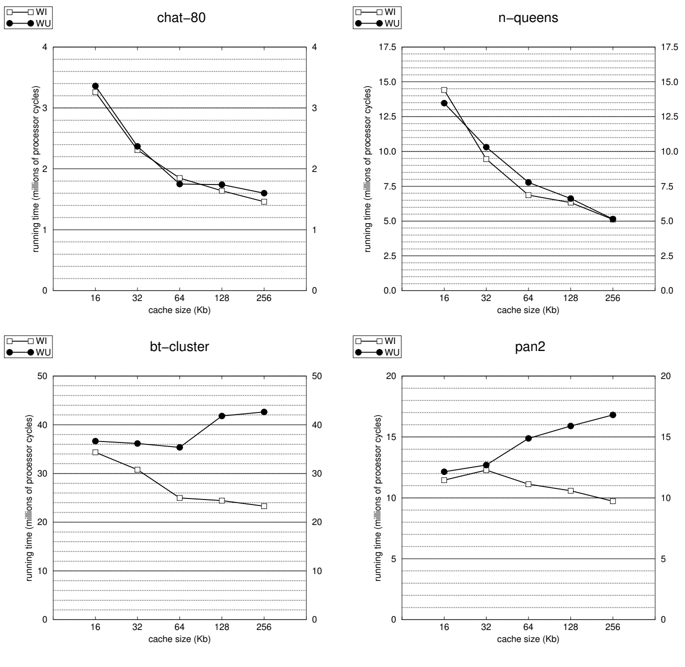

Fig. 5. Varying cache sizes, execution times

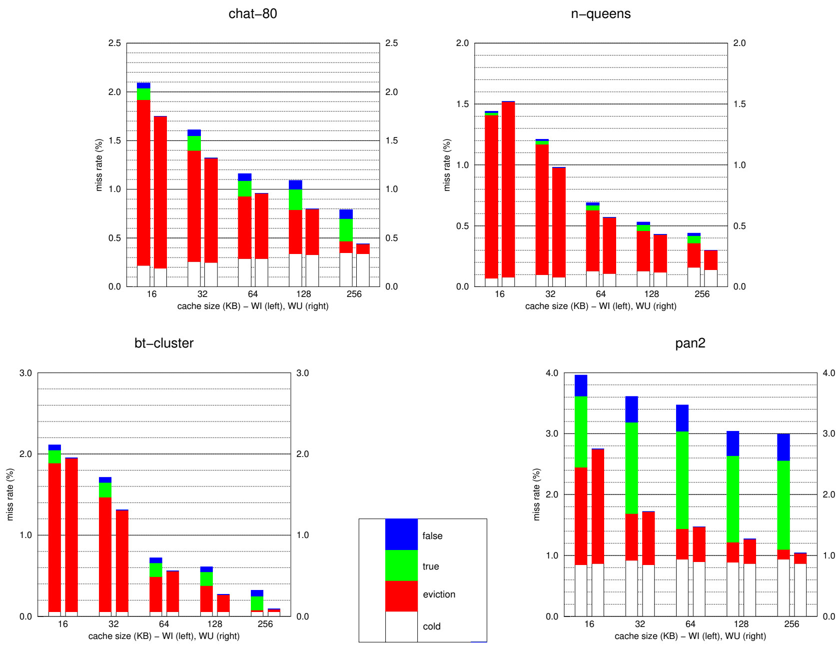

Fig. 6. Varying cache sizes, miss rates

3.2 Varying Cache Sizes

Figure 5 shows execution times in number of processor cycles for our applications as a function of different cache sizes. The figure shows that variations in cache size from 16 to 256 KBytes can speedup an application by as much as 2.8 times. This is the case of the `N-queens` application under the WI protocol, which executes in 14.41 million cycles as its worst time and 5.11 million cycles as its best time.

The figure also shows that the or-parallel applications achieve their best performance with 256 KBytes of cache, regardless of the coherence protocol. In contrast, applications with predominantly and-parallelism are severely affected by the kind of protocol used. For the WI protocol, the best cache size is still 256 KBytes, but for WU, smaller caches produce better performance. The reason for this latter result is that larger caches increase the degree of sharing, since fewer shared cache blocks are evicted. This higher degree of sharing means an increased number of update messages, which in turn causes an increase in the amount of network traffic involved in our applications. The applications with and-parallelism suffer more significantly from the larger caches, since they involve a larger amount of sharing between processes and communicate much more intensively than the or-parallel applications.

As one would expect, figure 6 shows that the read miss rates decrease with the increase in cache size, as a result of sharp decreases in the number of eviction misses. However, under the WU protocol, these decreases come at the cost of a substantial increase in proliferation updates, particularly for the applications with and-parallelism, as can be observed in figures 7 and 8.

This result is mostly a consequence of how processes share data in Andorra-I. More specifically, or-parallel applications have an access pattern of the type single-writer/multiple-readers for the local, heap, and trail stacks, and multiple-writers/multiple-readers for the choicepoint stack, which is much more rarely written than the other stacks. In contrast, in and-parallel applications processors read and write to the shared stacks (local, heap, and trail) and to the run queue of parallel goals.

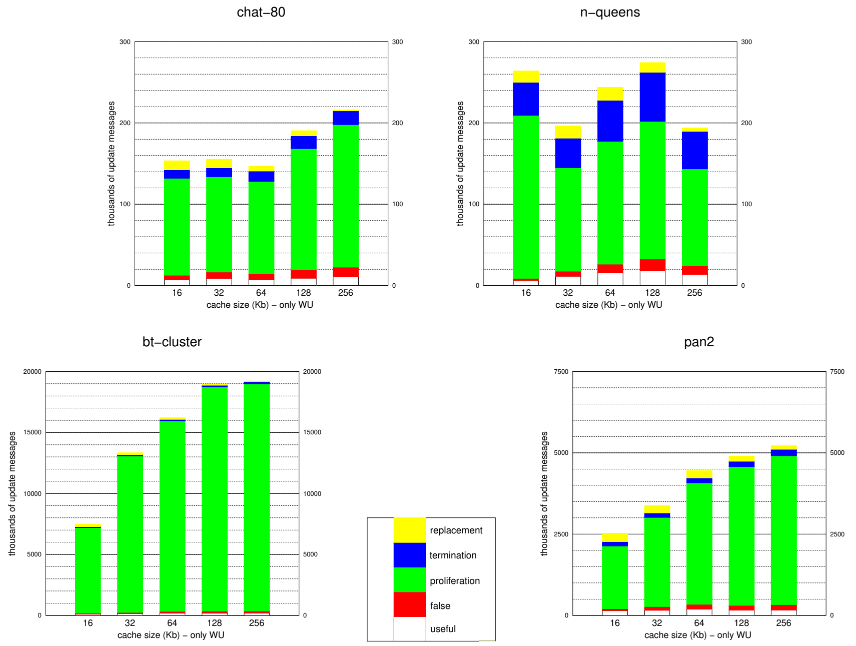

Fig. 7. Varying cache sizes, categorising update msgs

3.3 Varying Network Bandwidth

Figure 9 shows the performance of Andorra-I for our applications when varying the network bandwidth through variations in the network path width (our default path width, 16 bits, corresponds to a 200 MBytes/sec link bandwidth). As can be observed in the figure, and was expected, the larger the path width, the better the execution time, independently of the coherence protocol. An increase in network path width from 8 to 32 bits can make an application run 1.95 times faster. This is the case of the `pan2` application under the WU protocol, where the worst running time is 19.50 million cycles and the best time is 10.19 million cycles.

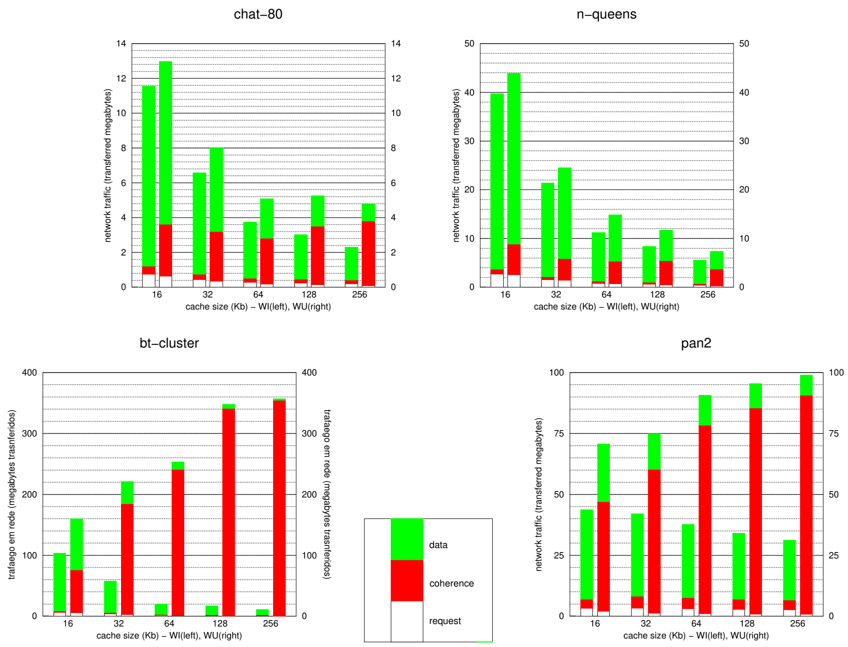

Fig. 8. Varying cache sizes, network traffic

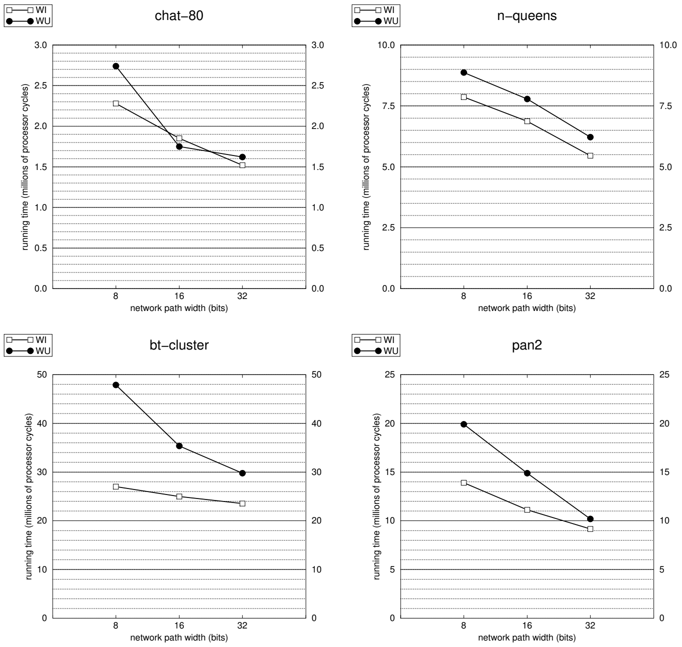

Fig. 9. Varying net path width, execution times

The performance improvements produced by the wider paths are more significant for the WU protocol, as it involves a more intense communication traffic than WI. However, not even our widest network paths were enough for WU to significantly outperform WI, regardless of the type of application.

3.4 Varying Write Buffer Sizes

We also varied the size of write buffers from 4 to 64 entries. This variation impacts the system performance only negligibly. Even when the percentage of writes is relatively high, the variations in write buffer size never produce more than a 5% difference in performance.

3.5 Summary Results

Our results show the architectural parameters have a significant impact on the performance of PLP systems such as Andorra-I. For example, Andorra-I is very efficient under the WI protocol, large cache sizes (128 or 256 KBytes), small cache blocks (16 or 32 bytes), and wide network paths (32 bits), but suffers somewhat when these parameters are not observed. The WU protocol is competitive with the WI protocol for the or-parallel applications, while it produces poor performance for the and-parallel applications. This is mainly due to the way and-parallelism is exploited in systems such as Andorra-I, where workers share most of the data structures and generate significant coherence traffic.

4 Conclusions

This work showed that parallel logic programming systems originally written for bus-based shared-memory machines can run with good performance on a scalable multiprocessor, provided that the architectural parameters of the multiprocessor are appropriate. More specifically, we simulated the Andorra-I system on a DASH-like scalable multiprocessor with different machine settings and coherence protocols. We used four different applications with diverse types of parallelism. We have shown that the performance of Andorra-I can be greatly affected by different architectural parameters. For instance, under the write-invalidate protocol, Andorra-I can benefit from large cache sizes, small cache blocks, and wide network paths. The depth of the write buffer does not affect performance significantly.

In general, invalidate-based protocols favour and-parallel applications while or-parallel applications are not much affected by either protocol, with some exceptions. We conclude then that a combination of invalidate and update protocols and a combination of the best parameters studied in this work could yield best performance.

We believe that our results will hold for any parallel logic programming system designed for shared-memory machines. We intend to repeat these experiments for Aurora [11], a logic programming system that only exploits or-parallelism. We also intend to study the behaviour of more applications in detail by investigating the impact of alternative cache coherence protocols on each shared data area in the system. Next we will tackle the problem of getting maximum performance of parallel logic programming systems on modern architectures by changing the structural design of the algorithms.

Acknowledgments

The authors would like to thank the CNPq Brazilian Research Council for its support. Vítor Santos Costa is thankful to PRAXIS and Fundação para a Ciência e Tecnologia for their financial support. This work was partially funded by the Melodia project (JNICT PBIC/C/TIT/2495/95).

References

1. 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.
2. 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.
3. J. A. Crammond. The Abstract Machine and Implementation of Parallel Parlog. Technical report, Dept. of Computing, Imperial College, London, June 1990.
4. 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.
5. I. C. Dutra. Strategies for Scheduling And- and Or-Work in Parallel Logic Programming Systems. In *Proceedings of the 1994 International Logic Programming Symposium*, pages 289–304. MIT Press, 1994. Also available as technical report CSTR-94-09, from the Department of Computer Science, University of Bristol, England.
6. 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.
7. 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.
8. Markus Hitz and Erich Kaltofen, editors. *Proceedings of the Second International Symposium on Parallel Symbolic Computation, PASCO'97*, July 1997.
9. 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.
10. 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.
11. 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.

12. E. M. McCreight. The Dragon Computer System, an Early Overview. In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, July 1984.
13. Johan Montelius and Seif Haridi. An evaluation of Penny: a system for fine grain implicit parallelism. In *Proceedings of 2nd International Symposium on Parallel Symbolic Computation [8]*, July 1997.
14. 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.
15. V. Santos Costa, 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 [8]*, pages 58–67, July 1997.
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 and Bianchini R. Optimising Parallel Logic Programming Systems for Scalable Machines. In *Proceedings of the EUROPAR'98*, Sep 1998.
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. Evan Tick. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers, Norwell, MA 02061, 1987.
20. 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.