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

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

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

Abstract

Parallel logic programming systems exhibit several forms of data sharing during execution. Systems that exploit dependent and-parallelism exhibit the one producer–many consumers sharing pattern for logical variables. Scheduling exhibits migratory sharing when accessing shared data structures for fetching work and synchronising processes. Producer–consumer sharing favours update-based coherence protocols, as opposed to the currently-dominant invalidate protocols. Migratory sharing can also profit from update-based protocols, but only at the cost of useless update messages that ultimately might degrade performance. Thus, to determine the best coherence protocol for parallel logic programming systems, we must investigate whether the substantial increase in message traffic caused by update-based protocols outweighs their benefits.

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 originally written to run on bus-based machines and another one optimised to our simulated architecture. 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 counterparts for the unoptimised version of Andorra-I. The optimised version of Andorra-I benefits from the invalidate-based protocol, but an hybrid update-based protocol performs well. We conclude that parallel logic programming systems originally written for bus-based machines can benefit from hybrid update-based protocols.

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 [16] and Muse [1]; independent and-parallelism, as in &-Prolog [11] and &-ACE [10]; dependent and-parallelism, as in Parlog's JAM [6], KLIC [25], and DDAS [24]; data-parallelism, as in Reform Prolog [4]; and combined and–or parallelism, as in Andorra-I [23], Penny [18], and ACE [9]. All these systems have been able to obtain good performance on 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 of writable data.

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, where the processor that instantiates a logical variable writes it and one or more processors read it.

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. As an example, because of the high cost of suspending and restarting processors, it is very common that idle processors will be cycling through shared data structures searching for work. A processor that produces a piece of work writes to one of these data structures, which later will be modified by one of the idle processors.

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

†On leave from the Universidade do Porto, Portugal.

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 [17] 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. More specifically, consider the case of dependent and-parallelism, where a processor A suspended on a variable and still keeps the variable in its cache. If processor B binds the variable, and processor A is the one to restart the goal, an update protocol ensures that processor A will already have the variable's value in its cache when it restarts. With an invalidate protocol, processor A would need to fetch the item from processor B, which requires several tens (maybe hundreds) of processor cycles. The update protocol is most beneficial for programs where many goals suspend on the same variable. In this case, it is likely that most processors will have the variable in their cache, and also that they will restart goals suspended on the variable.

Synchronisation can also benefit from an update-based protocol. As aforementioned, idle processors typically loop through shared data structures searching for work. With an update-based protocol, as soon as a processor writes a shared variable, the new value will be sent to the idle processors' caches. One of these processors can then start working immediately. The problem is that all other processors sharing the variable will receive an update that they cannot utilise. With an invalidate protocol, the new data will only reach the other processors when they fetch the shared variable, hence delaying their execution.

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.

The question therefore arises of how coherence protocols affect the performance of parallel logic programming systems. Is good performance possible in the novel architectures? If so, is it only possible through a specific set of protocols, be they WI or WU? Or are parallel logic programming systems largely independent of coherence protocols? Should we expect similar results for both and-parallel and or-parallel applications?

In this paper we use execution-driven simulation of a scalable multiprocessor running the Andorra-I system [23] towards obtaining an answer to these questions. 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. Moreover, in our work we experimented with two versions of Andorra-I: one originally written to run on bus-based machines; and another one optimised to run on our simulated architecture [20]. 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 counterparts for the unoptimised 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. We conclude that parallel logic programming systems originally written for bus-based machines can benefit from update-based protocols and that multiprocessors designed for running these systems efficiently should adopt some form of update protocol.

Our approach contrasts with previous studies of the performance of coherence protocols for parallel logic programming systems. Tick and Hermenegildo [27] 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 [19], 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 smaller data cache [21]. We next performed a detailed analysis of the WI protocol based on the Andorra-I data areas [22], that led us to proposing optimisations that improved performance of the WI protocol in more than 100% for some applications [20]. 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 paper is organised as follows. Section 2 presents the methodology used to obtain our results. Section 3 describes the Andorra-I parallel logic programming system, its main data areas, its port to MIPS and the optimisations performed to run on our simulated architecture. Section 4 describes our benchmark set. Section 5 presents speedup results for the or-parallel, and-parallel, and combined parallel benchmarks ran

in 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 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 [13].

2.1 Multiprocessor Simulation

We use a detailed on-line, execution-driven simulator that simulates a 24-node, DASH-like [15], 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 [28] (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 128-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.

A memory bus clocked at half of the speed of the processor connects the main components of each machine node. 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. 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.

Our WI protocol keeps caches coherent using the DASH protocol with release consistency [15]. 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 experimented with a dynamic hybrid protocol (WUh2) [14] based on the coherence protocols of the bus-based multiprocessors using the DEC Alpha AXP21064 [26]. 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.

We would expect this threshold to perform quite well for the sharing of logical variables, because, if a processor writes on a shared logical variable, it will do so only once during forward execution. For migratory variables, this strategy 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.

3 Andorra-I

The Andorra-I parallel logic programming system is based on the Basic Andorra Model [31]. The system was developed at the University of Bristol by Beaumont, Dutra, Santos Costa, Yang, and Warren [23, 33]. 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 [16] when exploiting or-parallelism.

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.

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 [34], 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 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.

3.1 Data Areas in Andorra-I

Andorra-I combines the techniques used to implement parallel committed-choice systems [6] and the techniques used to implement or-parallel systems [16], themselves based on sequential Prolog [29]. The shared memory areas that implement the functionality of Andorra-I can be classified as:

- **Or-Scheduler Data Structures.** The Bristol Or-scheduler uses three different data structures: a set of fields on each choice-point, and two data structures with global variables. One of the two latter structures is shared by every worker, and the other is replicated by every worker. We grouped these data structures in a single area.

- **Worker Data Structures.** These include a copy of the abstract machine registers, the variables used to synchronise within a team, and the run-queues for each worker. Most accesses to these data structures occur when performing and-scheduling, that is when fetching work from other workers in a team, or when synchronising workers in a team.

- **Lock Array.** The lock array is used to establish a mapping between a shared memory position (such as a variable in the heap) and a lock. It is used whenever we need to lock a variable. This area is required because the simulator does not implement the MIPS instructions to synchronise access to shared memory. We therefore use hashing to a lock array as the mechanism to guarantee synchronisation.
- **Code Space.** The code space includes the compiled code for every procedure. During execution of the benchmarks it is read-only, but it could be updated by programs that perform `assert` or `record`.
- **Heap Space.** As in other Prolog systems, this space stores structured terms and variables. It grows during forward execution, and contracts during backtracking.
- **Goal Frame Space.** This data area stores goal frames, which consist of the goal's arguments, multi-assignment variables used to link the goal frames, and several control fields. The engine tries to reuse goal frames during forward execution.
- **Choicepoint Stack.** Choicepoints include pointers to the top of stacks, and flags that are updated as processors move around searching for work. Differently from Prolog, no arguments need to be stored. Note that as we backtrack and move forward, the same choice point stack space may be used several times for different choicepoints.
- **Trail Stack.** The trail stack records any conditional binding to logical or multi-assignment variables (used in the implementation of Andorra-I [34]). It can be considered as an extension of a choicepoint that records non-determinate bindings and is only important in applications with or-parallelism.
- **Binding Arrays.** They are used to implement the SRI model [30] for or-parallelism, by storing conditional bindings. They should be private in or-parallel applications, and almost never written in and-parallel-only applications. Even during an execution without choice points (a determinate execution), Andorra-I will access this data structure to verify whether variables have conditional bindings.
- **Miscellaneous Shared Variables.** This area includes the shared I/O data structures, such as open stream descriptors. It also includes information on the clauses compiled into the system, and pointers to the determinacy and non-determinacy code. A set of flags and counters is globally manipulated by the system. This area also includes the data structures used by the reconfigurer.

We instrumented the simulator to report data separately on each region. We then ran our benchmark programs to collect statistics for each data area in turn.

3.2 Two versions of Andorra-I

We used in our experiments two versions of Andorra-I: one originally written for bus-based machines, and the other optimised for our simulated architecture.

The optimised version of Andorra-I used in our experiments uses 5 different 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. These optimisations were performed after a careful analysis of the data areas in the Andorra-I execution for the WI protocol [20].

3.3 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 [13] 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.

4 Applications

4.1 And-Parallel Applications

We used 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`. 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 [32]. 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, `fp`, is an example query for a knowledge-based system for the automatic generation of floor plans. This application should at least in principle have significant or-parallelism.

4.3 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]. 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` that was originally written for bus-based machines, and `optim` optimised for our DASH-like architecture. 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.

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

5.1 The `orig` version

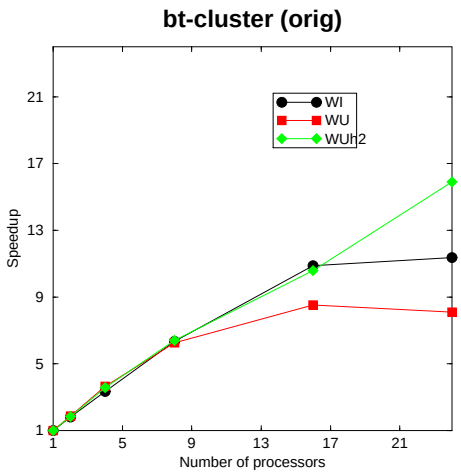

Figure 1: Speedups for `bt-cluster` (orig): 3 protocols

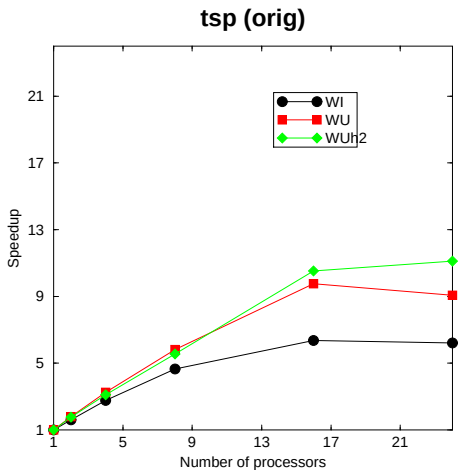

Figure 2: Speedups for `tsp` (orig): 3 protocols

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 is better for all applications when we increase the number of processors. The performance of the or-parallel applications are very similar, with `floorplan` (fig. 5) achieving better performance than `chat-80` (fig. 4).

Among the and-parallel applications, `bt-cluster` has excellent and-parallelism, resulting in almost linear speedups for the `WUh2` curve. 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 best speedup of 15.9 for 24 processors yielding an improvement of 40% over the WI protocol and almost 100% over the WU protocol.

Figure 2 shows that the `tsp` application achieves worse speedups than `bt-cluster`. For the WI protocol, the total number of references to shared memory, for `tsp`, with 16 processors, is above 26 million, with a total miss rate of 3%, while the `bt_cluster` benchmark has more than 27 million references to shared memory, with only 1.6% of total miss rate. This explains why `bt_cluster` scales up better.

It is interesting to observe that the performance of the WU protocol for the `tsp` application is better than the performance of the WI protocol whereas for `bt_cluster` it is not. In general, pure WU generates a very heavy traffic, yielding bad performance. The migratory sharing pattern of this application benefits from the WU protocol. While the total miss rate for this application with the WI protocol, with 16 processors, is 3% of the total number of references, the total miss rate with the WU protocol is reduced to 0.7% of the total number of memory references. In contrast, WU only improves the miss rate of `bt_cluster` from 1.6% to 0.9%. Although WU generated a heavier memory traffic than WI for both applications, the increase was

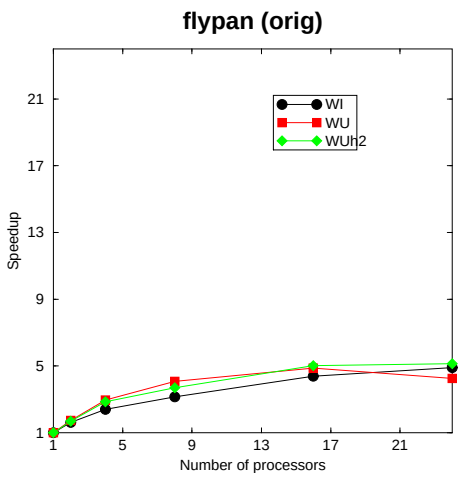

Figure 3: Speedups for `pan2` (orig): 3 protocols

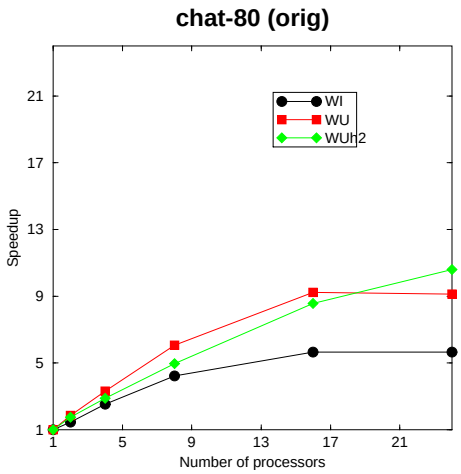

Figure 4: Speedups for `chat-80` (orig): 3 protocols

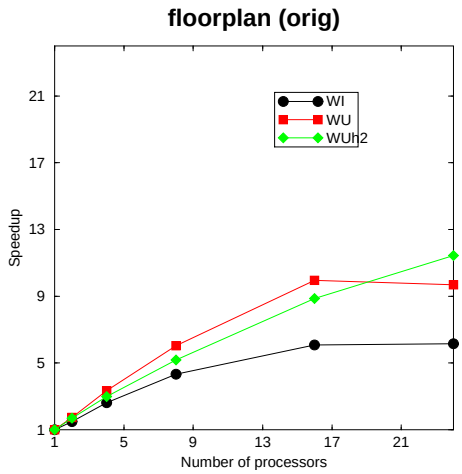

Figure 5: Speedups for `floorplan` (orig): 3 protocols

much more significant for `bt_cluster` where it jumped from around 71 MB to 313 MB, while the network traffic for `tsp` jumped from around 148 MB to 180 MB.

The same happens to the or-parallel applications: WU performs better than WI for both `chat-80` and `floorplan`. For instance, at 16 processors, the WU miss rate is only 1.6% for `floorplan`, versus 4% for WI. The network traffic actually decreased. The reason for the better performance of the WU protocol for these applications is that most sharing misses are false sharing misses and WU performs better with 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. Figure 6 shows that most misses for this application, for 16 processors, are concentrated on the `Worker` area and are mostly false sharing misses. For the WU protocol with 16 processors, more than 2 million update messages were exchanged and only about 170 thousand were useful, i.e., most update messages in the WU protocol with 16 processors were useless. Figure 8 shows that most misses in the WUh2 protocol with 16 processors are **drop** misses, concentrated mostly on the `Worker` area. These numbers show that this application is spending too much time in the and-scheduler trying to find and-work available. This causes a degradation in performance as we increase the number of processors, which does not allow the application to scale well.

The overall improvements of WUh2 over the WI and WU protocols are shown in table 5.1. 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.

App	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 10, 12, 11, 13, and 14 show the speedups for all protocols with the Andorra-I `optim` version.

The optimised version of Andorra-I resulted in better performance for the WI and WUh2 protocols. Best performance was obtained in 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 the or-parallel benchmark `chat_80` under the WI and WUh2 protocols, as both protocols obtain a speedup of 18 at 24 processors. On the other hand, the benchmark only scales up to a speedup of 14 under WU.

Results are quite different for the and-parallel applications. The `bt-cluster` application is the one where WI outperforms the other protocols: speedups are close to 20 at 24 processors. Performance of WUh2 is about 30% worse than WI. In contrast, WU saturates at a speedup of 6 for 16 processors. Both the WI and WUh2 saturate at a speedup of 12 for `tsp`. Performance of WU lags over WI and WUh2 after 8 processors: it scales up to only 9 at 16 processors, and has its bad performance with 24 processors, where it is actually slower than the the non-optimised version.

In a nutshell, the best performance is now obtained with the WI protocol, which is now the fastest protocol. Performance of the WUh2 protocol is close to the performance of WI except for the `bt_cluster` benchmark. Results for WU are now always worse than results for both WI and WUh2.

As discussed in section 3.2 the major differences between optimised and original versions 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. Our results show that WU does not improve as much, partly because it suffers less with false sharing than WI, and partly because the extra memory usage results in more data and coherence messages in the network.

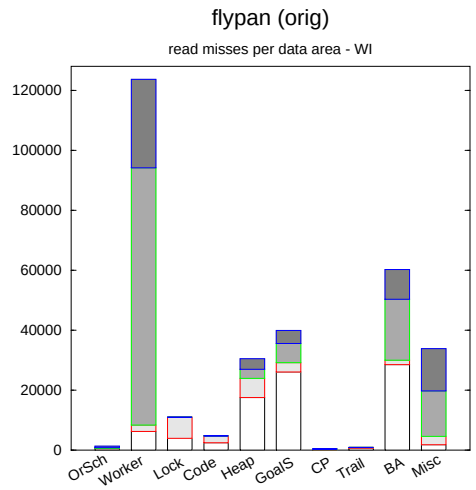

Figure 6: Read misses for `flypan` (orig) per data area: WI

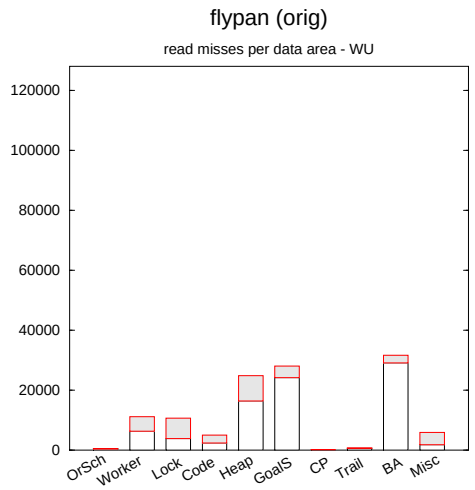

Figure 7: Read misses for `flypan` (orig) per data area: WU

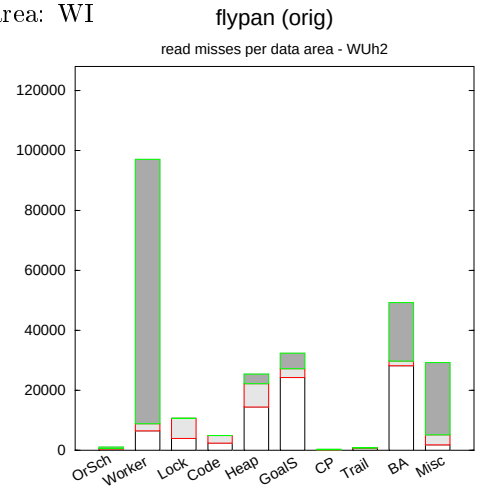

Figure 8: Read misses for `flypan` (orig) per data area: WUh2

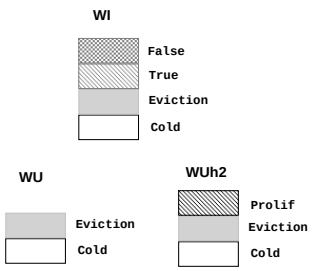

Figure 9: Legends for read misses

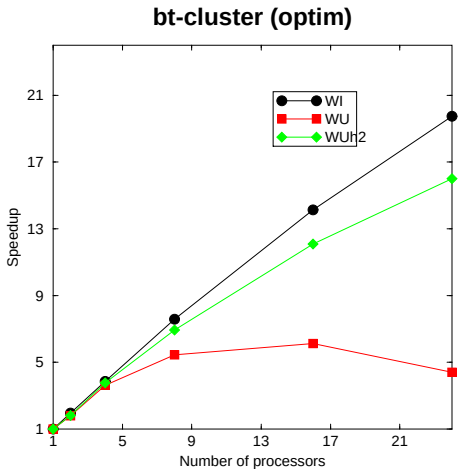

Figure 10: Speedups for `bt-cluster` (optim): 3 protocols

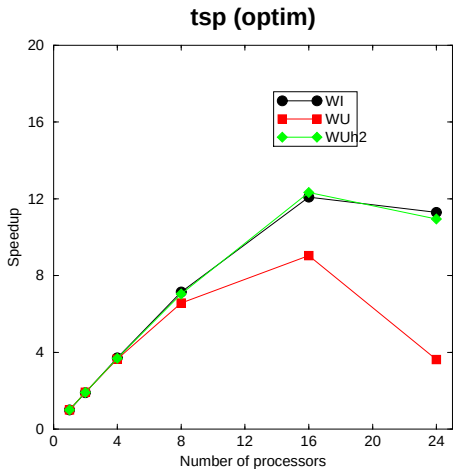

Figure 11: Speedups for `tsp` (optim): 3 protocols

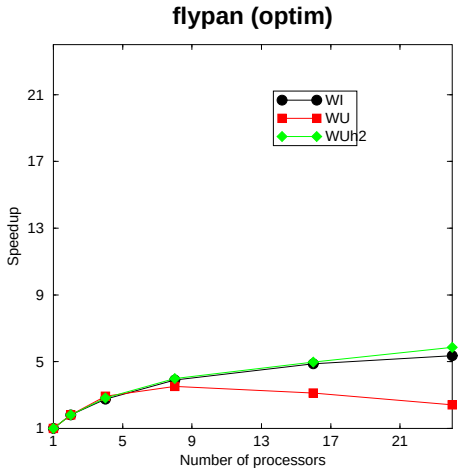

Figure 12: Speedups for `pan2` (optim): 3 protocols

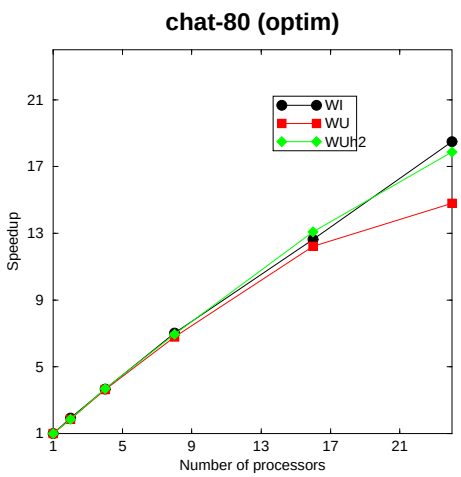

Figure 13: Speedups for `chat-80` (optim): 3 protocols

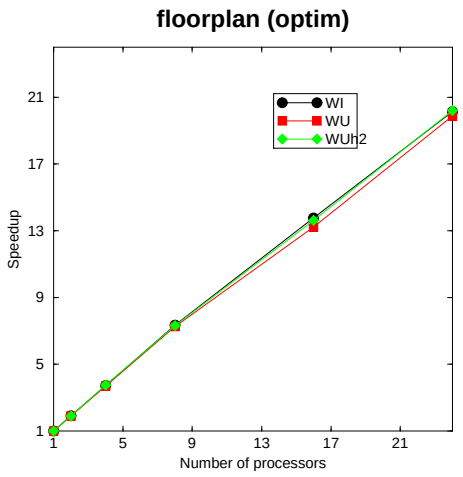

Figure 14: Speedups for `floorplan` (optim): 3 protocols

6 Conclusions and Future Work

We studied the performance of the Andorra-I parallel logic programming system under different coherence protocols used for distributed shared-memory machines. We experimented with two versions of Andorra-I: one was a straightforward port of the original system that was originally written for bus-based machines; the second was optimised assuming a Dash-like architecture running WI-based protocols. Our results show that the hybrid protocol WUh2 can obtain good performance for both a non-optimised and an optimised version of Andorra-I.

The WUh2 protocol is quite effective for a system originally written for a bus-based machine. This is because this protocol is not as vulnerable to false sharing as WI, and because it does not generate as much network traffic as pure WU. The WUh2 protocol also benefits from optimisations performed for a WI-based machine.

The WI 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.

The WU protocol's main advantage is that it is not as vulnerable to false sharing as the WI 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. 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] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Markus Hitz and Erich Kaltofen, editors. *Proceedings of the Second International Symposium on Parallel Symbolic Computation, PASCO'97*, July 1997.

[13] Gerry Kane and Joe Heinrich. *MIPS RISC Architecture.* Prentice Hall, 1992.

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

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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 [12]*, pages 58–67, July 1997.

[23] 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.

[24] Kish Shen. *Studies of And/Or Parallelism in Prolog.* PhD thesis, Computer Laboratory, University of Cambridge, 1992.

[25] 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.

[26] 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.

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

[28] 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.

[29] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[30] 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.

[31] David H. D. Warren. The Andorra model. Presented at Gigalips Project workshop, University of Manchester, March 1988.

[32] 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.

[33] 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.

[34] 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.