Evaluating the Impact of Coherence Protocols on Parallel Logic Programming Systems

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

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

Technical Report ES-389/96, May 1996, COPPE/UFRJ

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 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. The detailed analysis of these results shows that update-based protocols outperform invalidate-based protocols regardless of the type of parallelism exhibited by the benchmarks. The reasons for this behaviour are explained in detail. We conclude that parallel logic programming systems can benefit from update-based protocols and that multiprocessors designed for running these systems efficiently should adopt some form of update or hybrid protocol.

*On leave from the Universidade do Porto, Portugal.

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 Aurora [21] and Muse [1]; independent and-parallelism, as in &-Prolog [16] and &-ACE [15]; dependent and-parallelism, as in Parlog's JAM [9], KLIC [27], and DDAS [26]; data-parallelism, as in Reform Prolog [4]; and combined and–or parallelism, as in Andorra-I [25] and Penny [23]. 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.

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 [14] 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 [22] 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. However, the nature of sharing in parallel logic programming systems suggests that update-based protocols might be more appropriate than their invalidate-based counterparts. In order to confirm this hypothesis, in this paper we evaluate the performance of update and invalidate-based protocols for parallel logic programming systems.

We use execution-driven simulation of a scalable multiprocessor running the Andorra-I system [25]. 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 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. The detailed analysis of these results shows pure WU performing better than WI for small numbers of processors, both for or-parallel and and-parallel benchmarks. Our results also suggest that a hybrid (WU+WI) protocol can obtain best performance for larger numbers of processors. We conclude that parallel logic programming systems 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 [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 [24], but did not evaluate the impact of different coherence protocols. Our goal in this study is to evaluate the performance implications of different protocols for parallel logic programming systems on scalable multiprocessors, while categorising cache misses and update messages in the system.

The paper is organised as follows. Section 2.1 presents the methodology used to obtain our results. Section 2.2 describes the Andorra-I parallel logic programming system. Section 3 presents speedup results for the or-parallel, and-parallel, and combined parallel benchmarks ran in Andorra-I under WI and WU protocols. Section 4 evaluates the performance of a hybrid protocol for the same benchmarks. Section 5 discusses message coalescing as a technique for improving update-based 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 [18].

2.1 Multiprocessor Simulation

We use a detailed on-line, execution-driven simulator that simulates a 24-node, DASH-like [20], 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] (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 64-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 [20]. 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.

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 acknowledgement 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.

2.2 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 [25, 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 [10] when exploiting and-parallelism, and by Aurora [21] 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.

2.3 The MIPS Port

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

Most of the port was straightforward. The only difficulties arose with shared memory allocation and locking. For shared memory allocation we use the `shmalloc` library supported by MINT. For locking in modern MIPS machines, we would use the `ll`, `sc`, and `sync` machine instructions [18] to implement locks and atomic operations. Unfortunately, these instructions are not yet supported by the back-end. The alternative is to use the lock library routines, as implemented by the simulator, which allow us to control the synchronisation overhead. In our simulations, these routines were implemented as atomic instructions.

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

3 Application Performance Under WI and WU Protocols

In this section we present speedup results for Andorra-I under both kinds of protocol, WI and WU. 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 application, were obtained using the reconfigurer to automatically adapt to the available parallelism. The or-parallel application used a fixed all-masters configuration. We also obtain times for the *first* run of an application (results would be somewhat better for other runs).

3.1 And-Parallel Applications

As an example and-parallel application, we used the clustering algorithm for network management from British Telecom [8]. 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. To obtain best performance, we rewrote the original application to become a determinate-only computation. 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.

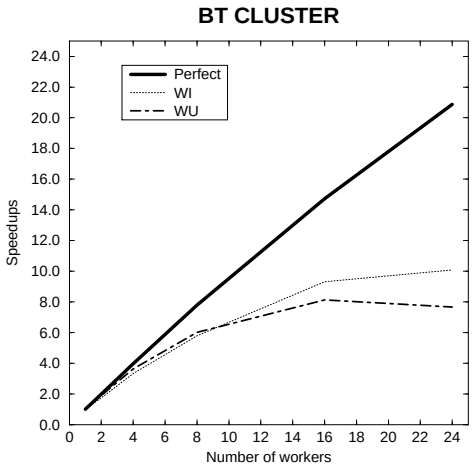

Figure 1: Speedups for `bt-cluster`

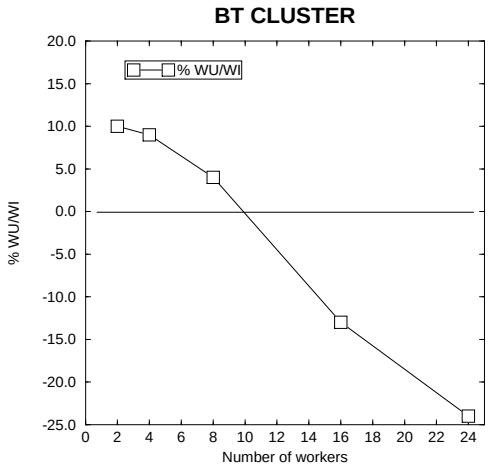

Figure 2: WU versus WI for `bt-cluster`

Figure 1 shows the speedups for both protocols. Figure 2 presents the execution time ratio between the WU and WI protocols. The application has excellent and-parallelism, resulting in almost linear speedups for the `perfect` curve. This curve is obtained for an ideal shared-memory machine, where data items can always be found in cache, and gives an idea of the maximum available parallelism in the application.

Performance for a realistic machine is acceptable. The invalidate protocol (WI curve) starts with an efficiency of 87% for 2 processors. Efficiency smoothly decreases as the number of processors increases, but is still over 50% for 16 processors. The update protocol (WU curve) has a very interesting behaviour. It starts with excellent efficiency of about 90% up to four processors. Then performance starts degrading, and after 16 processors there is a smooth slowdown. We discuss this problem in detail in the next section.

3.2 Or-Parallel Applications

As our or-parallel application we use an example from the well-known natural language question-answering system `chat-80`, written at the University of Edinburgh by Pereira and Warren [32]. 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.

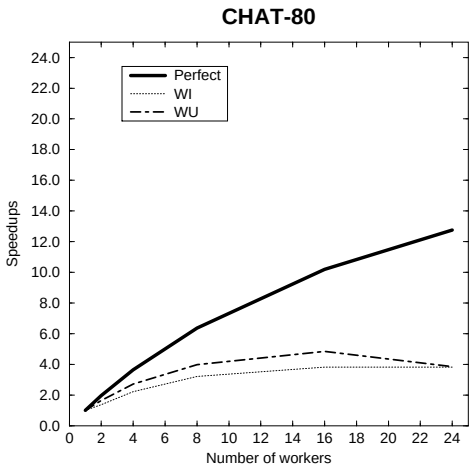

Figure 3: Speedups for `chat-80`

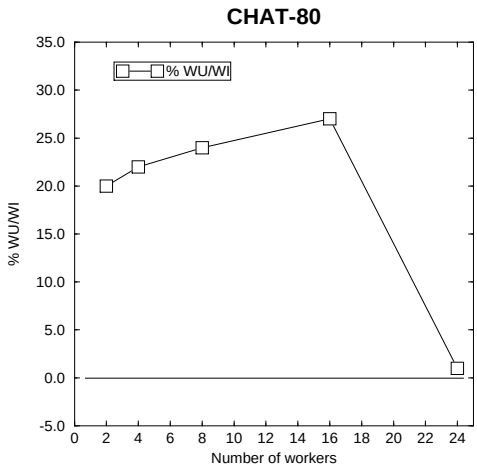

Figure 4: WU versus WI for `chat-80`

Figure 3 shows the speedups and execution time ratios for both protocols, from 1 to 24 processors, and figure 4 shows the relative performance of WU versus WI. The `perfect` curve gives almost linear speedup up to 4 processors, after which the speedup starts to level off. These speedups are very similar to the ones obtained under the Sequent Symmetry architecture [13]. The main difference is that the speedup on the Symmetry levels-off at a maximum speedup of 10. This comparison shows that the Symmetry behaves almost as if memory accesses were free of cost, which for state-of-the-art processor and memory speeds is unrealistic.

A comparison between the `perfect` and realistic curves proves this point. On this small-scale benchmark, the invalidate protocol manages to obtain a maximum speedup of 4. Up to sixteen workers, performance for the update protocol is from 20% to 27% better for this benchmark, obtaining a maximum speedup of 4.9 for 16 workers. It is very interesting to notice that, as the number of workers increases up to 24, performance for the update protocol quickly deteriorates, whereas the invalidate protocol manages to sustain performance.

3.3 And/Or-Parallel Applications

We used a program to generate naval flight allocations, based on a system developed by Software Sciences and the University of Leeds for the Royal Navy. It is an example of a real-life resource allocation problem. The program allocates airborne resources (such as aircraft) whilst taking

into account a number of constraints. The problem is solved by using the technique of active constraints as first implemented for Pandora [2]. 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.

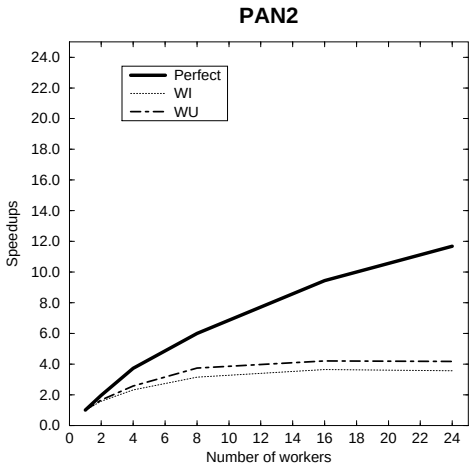

Figure 5: Speedups for `pan2`

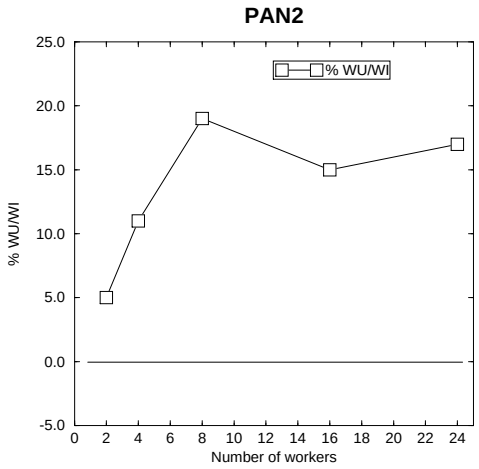

Figure 6: WU versus WI for `pan2`

Figure 5 and 6 shows the speedups and execution time ratios for `pan2`. The `perfect` speedups are acceptable. The application performs well up to 8 workers, and then parallelism starts to level off. The performance of `pan2` on the Symmetry is slightly worse than `perfect` [13].

The performance on a realistic machine demonstrates effects similar to the previous applications. Andorra-I on a modern machine cannot match speedups it would obtain under a machine with one-cycle memory access time. The invalidate protocol starts with an efficiency of about 80% for two workers. Efficiency then decreases quickly, but the machine is able to improve speedups up to 16 workers. The update protocol starts with an advantage of 5% for two workers over invalidate. This advantage improves up to 8 workers, and then starts decreasing.

3.4 Analysis of Results

According to the results shown in the previous section, the WU protocol invariably performs better than the WI protocol for small numbers of processors. As we increase the number of processors, performance of the WU protocol starts to degrade. Figures 7, 8, and 9 help to understand this phenomenon. Figure 7 shows the shared read miss rates for the three applications, with WI as the left bar and WU as the right one. The percentage at the top of each column represents the percent of all shared reads that result in a miss. For WI, within a column, misses are classified as:

- **Cold start misses.** A cold start miss happens on the first reference to a block by a processor.
- **True sharing misses.** A true sharing miss happens when a processor references a word belonging in a block it had previously cached but has been invalidated, due to a write by some other processor to the same word.
- **False sharing misses.** A false sharing miss occurs in roughly the same circumstances as a true sharing miss, except that the word written by the other processor is not the same as the word missed on.
- **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 [12], as extended in [5]. Note that the WU protocol does not have sharing misses.

As expected, WU always results in lower read miss rates, even though WI usually exhibits lower replacement miss rates as invalidations effectively free up cache space. The reason is that for WI most misses are sharing misses (from 55%, in `chat-80`, to 70%, in `bt-cluster`). This is sufficient to compensate for the increase in eviction misses from WU, which increases up to 17% for `bt-cluster`. Ultimately, by avoiding sharing misses, WU more than halves the total miss rate for the benchmark set.

Note that for all applications Andorra-I demonstrates a very high percentage of false sharing misses, ranging from 24% of total misses in `pan2` to 34% in `bt-cluster`. False sharing hurts performance under both the WI protocol, by increasing the miss rate, and the WU protocol, by generating unnecessary update messages. False sharing may arise from shared variables in the schedulers, or from two logical variables that were created in sequence, say, within the same compound term.

Although WU produces lower miss rates than WI, it is at the cost of a large increase in network traffic. This increase can cause several forms of performance degradation due to an increase in network and memory congestion. Figure 8 shows this difference for 16 processors. Basically,

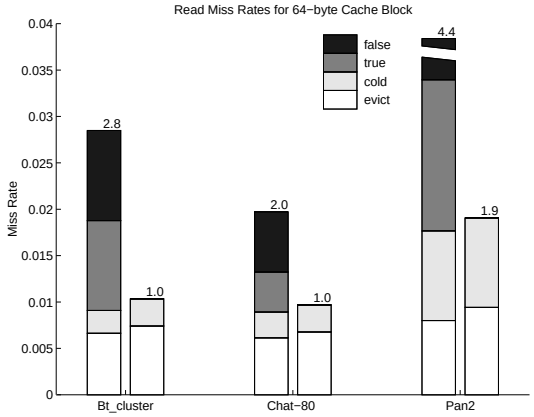

Figure 7: Read miss rate under WI (left) vs. WU (right), 16 processors.

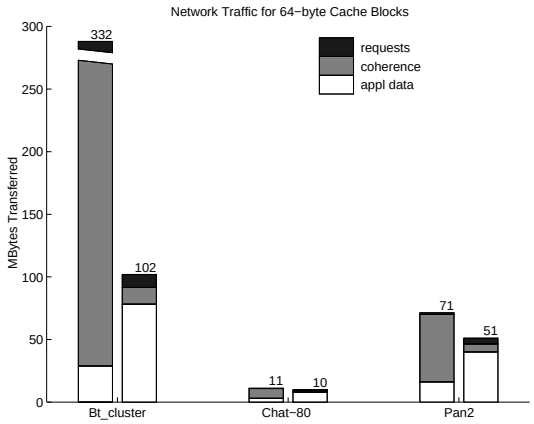

Figure 8: MBytes transferred under WI (left) vs. WU (right), 16 processors.

WU causes an increase in network traffic with respect to WI that ranges from a factor of 3 in `bt-cluster` to 40% in `pan2` and 11% in `chat-80`.

It is important to note that most of the traffic generated by the WU protocol is coherence-related, i.e. update messages. However, as previous studies have shown [6], most of these updates are useless, at least for scientific applications. In order to determine whether the same is true for Andorra-I, we classify updates as either *useful*, which are needed for correct execution of the program, or *useless*. The latter category of updates includes:

- **True sharing updates.** The receiving processor references the word modified by the update message before another update message to the same word is received.
- **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 the word modified by the update message before it is overwritten, and it does not reference any other word in that cache block either.
- **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.

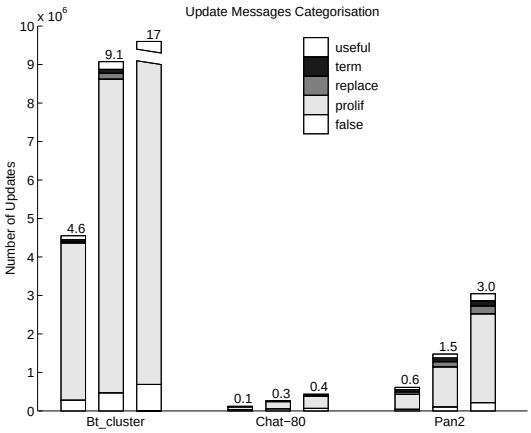

Figure 9: Categorisation of updates, 4 (left), 8 (center), and 16 (right) processors.

This classification uses the algorithm described in [5]. The categorisation is fairly straightforward, except for the false update class. Successive (useless) updates to the same word in a block are classified as proliferation instead of false sharing updates, if the receiving processor is not concurrently accessing other words in the block. Thus, the algorithm classifies useless updates as proliferation updates, unless *active* false sharing is detected or the application terminates execution.

We can observe from figure 9 that for 16 processors more than 90% of the update messages are indeed useless, with the actual percentage of useful updates varying from only 2.5% in the `bt-cluster` application, to 4.4% in `chat-80`, and 6% in `pan2`. The main culprits are by far the proliferation updates, which grow quickly with the number of processors and eventually become 77% of the useless updates in `chat-80`, 81% in `pan2`, and 93% in `bt-cluster` for 16 processors. A high rate of proliferation updates is typical of migratory sharing of data such as in shared counters, or shared work queues: data structures that are updated very often during execution, and which are indeed very common in both and- and or-schedulers and in memory allocation within a team.

The second most important category of useless updates in all applications is false updates. For 16 processors they are 4% of the total number of useless updates in `bt-cluster`, 7.5% in `pan2`, and 16% in `chat-80`. These updates are the direct consequence of the widespread false sharing in the system. `bt-cluster` is an interesting example where processors do falsely share data, but not simultaneously.

This excessive increase in useless update traffic for large numbers of processors degrades WU performance significantly by increasing memory access times due to network and memory congestion. These results suggest that reducing the number of useless updates should improve WU

performance and therefore improve its scalability.

4 A Hybrid Protocol for Andorra-I

In order to reduce the number of update messages of the WU protocol, we experimented with a dynamic hybrid protocol (WUh2) [19] 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.

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.

Figures 10, 11, and 12 show speedup curves for the WUh2 protocol. The `perfect`, `WI`, and `WU` curves are repeated for comparison.

The application `bt-cluster` that contains only dependent and-parallelism (figure 10) has its speedups enormously improved with the hybrid protocol. Speedups for 24 processors increased from 8 with the WU protocol to 13.4 with the WUh2 protocol. Moreover, the knee of the speedup curve, which is reached for 16 workers with WU, is not reached with WUh2 up to 24 processors.

The `chat-80` benchmark (figure 11) performs well under the hybrid protocol. This protocol achieves the best performance overall, being comparable to the WU protocol for smaller numbers of processors, while maintaining rising speedups for the largest number of processors.

Finally, we would expect the `pan2` application (figure 12) to also improve speedups with a change of protocol, as it contains a reasonable amount of and-parallelism. This is indeed the case, although the difference is not as impressive as for `bt-cluster`. WUh2 performs closely to WU up to 16 processors, and then obtains slightly better performance for 24 processors. WUh2 also has better performance than WI for all numbers of processors.

4.1 Analysis of Results

Figure 13 compares the shared read miss rate for WU (left) and WUh2 (right). The cache miss categorisation now includes an extra class, `drop` misses, to account for cache misses resulting from excessively eager self-invalidations. The results show that the price to pay for WUh2 is a significant number of drop misses: 46% of total misses with `chat-80`, 50% for `pan2`, and 58% of total misses for `bt-cluster`. As a result, the read miss rate about doubles in comparison to WU for all applications. In the case of `bt-cluster` the total shared read miss rate is still significantly

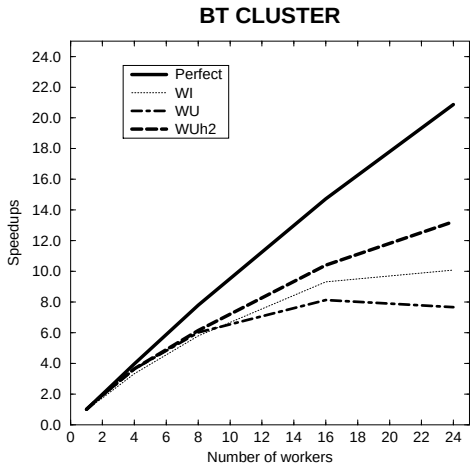

Figure 10: Speedups for `bt-cluster`

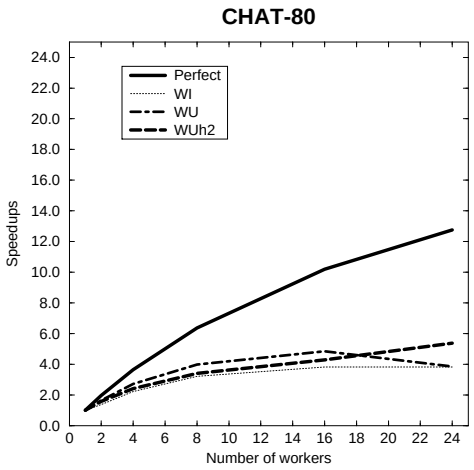

Figure 11: Speedups for `chat-80`

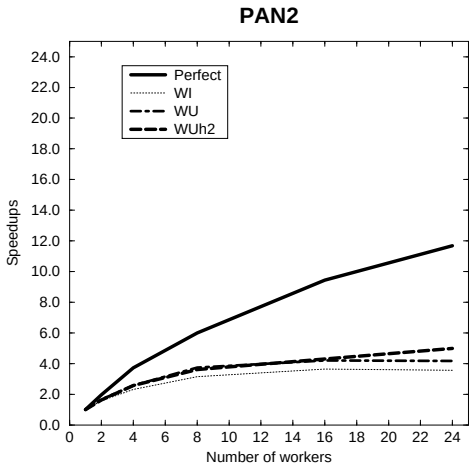

Figure 12: Speedups for `pan2`

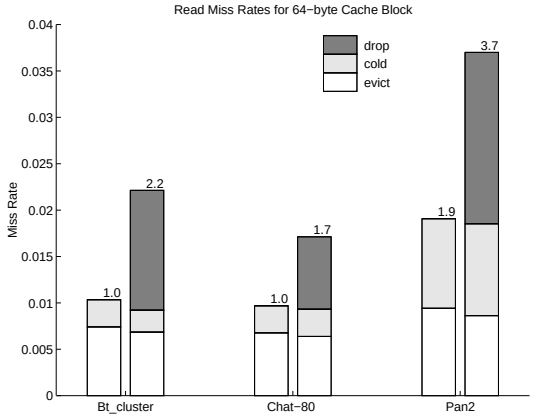

Figure 13: Read miss rate under WU (left) vs. WUh2 (right), 16 processors.

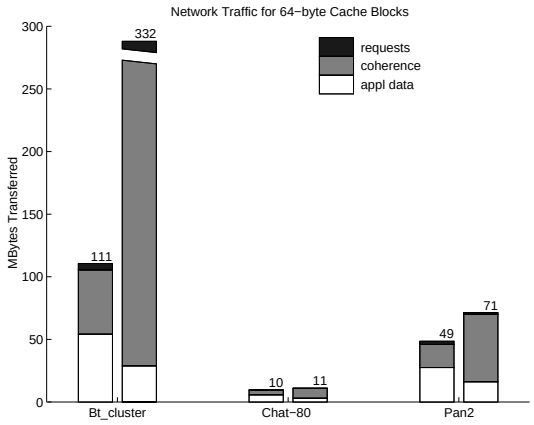

Figure 14: MBytes transferred under WU (left) vs. WUh2 (right), 16 processors.

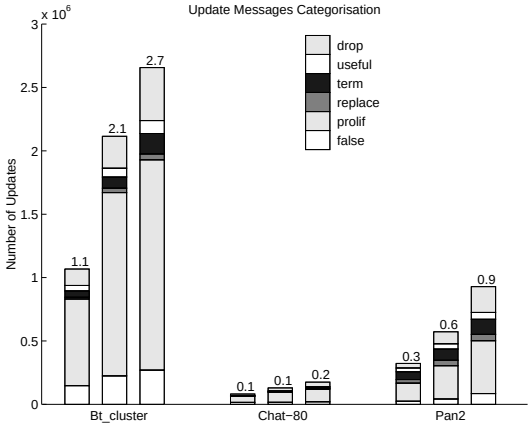

Figure 15: Categorisation of updates, WUh2, 4 (left), 8 (center), and 16 (right) processors.

lower than for WI, 2.2% versus 2.8%. The difference is smaller in `chat-80` and `pan2`. For `chat-80` it is 1.7% versus 2.0%, and for `pan2` it is 3.7% versus 4.4%.

Figure 14 shows the network traffic for all applications under the WU (left) and WUh2 (right) protocols for 16 processors. In all cases there is an increase in application data transfer. The increase is most substantial in `bt-cluster`, where application data transfer more than doubles. However, in `bt-cluster` the major effect is the decrease in coherence traffic. The `pan2` benchmark also benefits from the same effect, but in `chat-80` the reduction in coherence traffic is less significant as a percentage of the total traffic. This behaviour of `chat-80` is due to the fact that data sharing is not widespread in or-parallel execution; the vast majority of update operations have a single recipient for all numbers of processors.

In figure 15 we present the categorisation of update messages for the WUh2 protocol. The categorisation of updates shown in the figure includes an additional category, `drop` updates, to account for the updates that cause blocks to be invalidated. A comparison between figures 15 and 9 shows that the total number of update messages in WUh2 versus WU at 16 processors has been reduced by a factor of 2 for `chat-80`, 3 for `pan2`, and 6 for `bt-cluster`. This is a direct result of the fact that WUh2 has been very effective at reducing the number of proliferation messages: for 16 processors, the proliferation updates dropped by 90% in `bt-cluster`, by 82% in `pan2`, and by 69% in `chat-80`.

In summary, WUh2 seems to be the most appropriate coherence protocol, as it consistently achieves better scalability than the other protocols, and performs acceptably for small numbers of processors.

5 Coalescing of Updates for Andorra-I

An alternative approach for reducing update traffic that has been successful for scientific applications [11] is coalescing of update messages. A coalescing buffer [17] is simply a cache-block-wide buffer capable of merging writes to the same cache block. In the context of a WU protocol, this feature reduces the number of update messages propagated outside the processor.

We repeated the same experiments we performed for the WU and WUh2 protocols using a coalescing buffer. Our implementation of these buffers assumes 4 entries. We associate a dirty bit vector with each entry in write buffer indicating the words that were written. Unlike traditional write buffers, our coalescing buffer does not attempt to write its entries out immediately; it waits until there are 2 valid entries in the buffer or until it is forced to flush all entries at a synchronisation point. When a write is issued from the buffer, only the dirty words are sent in the message.

Although this protocol reduced the network traffic, it proved to be relatively inefficient for Andorra-I, having worse performance than the other protocols. As a data point, for `pan2` at 16 processors, performance of the WU + coalescing is about 23% worse than pure WU and 25% worse than WUh2. The poor coalescing performance stems from the fact that coalescing increases cache lockout overhead and delays the reception of acknowledgements. Taking the same

pan2 example, we find that cache lockouts and acknowledgement delays increase by factors of 20 and 1.3, respectively, when going from pure WU to WU + coalescing. These effects occur when coalescing is effective at combining several update messages belonging to the same cache block, and therefore increase the average size of messages significantly.

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. Up to a certain number of processors the sharing behaviour of the system favours WU protocols, but excessive traffic degrades performance of pure WU protocols for larger number of processors. In order to tackle this problem we then evaluated a hybrid (WU+WI) protocol, according to its cache miss rate and network traffic characteristics. Our results show that the hybrid strategy delivers the best overall performance of all protocols. Pure WU and hybrid perform comparably for small numbers of processors, but the hybrid protocol does not suffer as much performance degradation for a large number of processors.

The main conclusion of our work is that parallel logic programming systems that aim at good performance on scalable machines will benefit from some form of update-based protocol. In our benchmarks this was clear both for applications with dependent and-parallelism and or-parallelism. The trend towards faster CPUs, larger caches, and more available bandwidth also favours update-based protocols.

Our results have given a better perspective into the performance of the current version of Andorra-I and how it can be improved. A major result from our analysis is that migratory sharing in the system, not data sharing in the applications, is the most significant form of sharing in Andorra-I. We have also noticed that a substantial part of this sharing is false, and it must come from accesses to static scheduler data structures. We confirmed these results by performing an in-depth analysis of the caching behaviour of the system's shared data structures [7]. This result suggests that to obtain better scalability in Andorra-I the key effort should be on improving scheduling and synchronisation. This is good news, as reducing sharing in the applications would require work from Andorra-I users. We have noticed that the system can be easy improved by reducing false sharing in the schedulers' data structures. More distributed algorithms for fetching work and synchronisation within and between teams, will require substantial changes to the Andorra-I's schedulers and to the system's memory management. We are currently investigating new scheduling algorithms for Andorra-I.

In the near future, 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.

Acknowledgements

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 would like to thank the University of Porto for granting his period of leave to perform this work, and also would like to thank support from the PROLOPPE 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 Håkan 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] R. Bianchini, T. J. LeBlanc, and J. E. Veenstra. Categorizing network traffic in update-based protocols on scalable multiprocessors. In *To appear in Proceedings of the International Parallel Processing Symposium '96*, April 1996.

[7] V. Santos Costa, R. Bianchini, and I. Dutra. Analysing the Caching Behaviour of Parallel Logic Programming Systems. In Preparation, COPPE/Sistemas, Universidade Federal do Rio de Janeiro, May 1996.

[8] Barry Crabtree. A clustering system to network control, British Telecom, March 1991.

[9] 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.

[10] J. A. Crammond. The Abstract Machine and Implementation of Parallel Parlog. Technical report, Dept. of Computing, Imperial College, London, June 1990.

[11] F. Dahlgren and P. Stenstrom. Reducing the write traffic for a hybrid cache protocol. In *Proceedings of the 1994 International Conference on Parallel Processing*, August 1994.

[12] 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 International Symposium on Computer Architecture*, pages 88–97, May 1993.

[13] Inês 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. Ph.D. thesis.

[14] 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.

[15] 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.

[16] 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.

[17] N. P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 191–201, May 1993.

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

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

[20] 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 International Symposium on Computer Architecture*, pages 148–159, May 1990.

[21] 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.

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

[23] 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.

[24] 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.

[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 PPOPP*, 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 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.