Evaluating Parallel Logic Programming Systems on Scalable Multiprocessors

Vítor Santos Costa*

Artificial Intelligence and
Computer Science Laboratory
Porto University
4150 Porto, Portugal
vsc@ncc.up.pt

Ricardo Bianchini and Inês de Castro Dutra

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

Abstract

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

In this work we use execution-driven simulation to investigate the access patterns and caching behaviour exhibited by a parallel logic programming system, Andorra-I. We show that the system obtains reasonable performance, but that it does not scale well. By studying the behaviour of the major data structures in Andorra-I in detail, we conclude that this result is largely a consequence of the scheduling and work manipulation implementation used in the system. We also show that the Andorra-I's data structures exhibit widely-varying memory access patterns and caching behaviour, which not only depend on the number of processors, but also on the amount and type of parallelism available in the application program. Some of these data structures clearly favour invalidate-based cache coherence protocols, while others favour update-based protocols. Since most of Andorra-I's data structures are common to other parallel logic programming systems, we believe that these systems can greatly benefit from flexible coherence schemes where either the compiler can specify the protocol to be used for each data structure or the protocol can adapt to varying memory access patterns.

Keywords: Symbolic Computation, Parallel Logic Programming, And-Or Parallelism, Scalable Multiprocessors.

*This research was developed while visiting Federal University of Rio de Janeiro, Brazil, sponsored by the Brazilian Research Council, CNPq.

1 Introduction

Parallel computers can improve performance of both numerical and symbolic applications. Logic programs are good examples of symbolic applications that can greatly benefit from parallelism. This is largely because logic programs often exhibit large amounts of implicit parallelism. To fully exploit this parallelism, parallel logic programming systems require a high level of sophistication. They must dynamically allocate and manage their data structures, in contrast to traditional scientific applications that usually work by performing several computations on fixed arrays. Moreover, they must support applications that often exhibit irregular execution patterns, whereas scientific applications often exhibit regular execution patterns.

Another interesting aspect of parallel logic programming systems is that they exploit different types of implicit parallelism and, thus, exhibit different sharing patterns. In or-parallel systems, such as Aurora [18] or Muse [1], different alternatives in the search-tree are explored independently. Writing shared data structures is only necessary for scheduling and for fetching work. Therefore, most accesses to shared execution stacks are read-only. In systems that exploit independent and-parallelism [12], where goals that do not share variables are run in parallel, shared logical variables can be written by at most a single processor at a time. In contrast, dependent and-parallelism actively shares logical variables during execution. This allows for, say, producer-consumer parallelism, or concurrent updates to shared tableaus. Writing shared data structures can be a common operation. To simplify execution, restrictions are required on binding logical variables. In the committed-choice systems [24] and also in Andorra-I [22] the main restriction is that logical variables will be bound only once during an and-parallel execution phase.

Most parallel logic programming systems have been developed for bus-based shared-memory architectures, even though a few systems have been implemented for distributed memory architectures [14]. Specialised hardware has also been built for parallel logic programming, as exemplified by PIM machines developed at ICOT to support the KL1 language [13]. However, the modern computer architectures being developed today pose new challenges, such as the high latency of memory accesses and the demand for scalability. In modern multiprocessors, performance depends heavily on the miss rates and may be limited by the communication overhead that is involved in sharing of writable data. The complexity of parallel logic programming systems and

the large amount of data they process raise the question of whether these systems can obtain good performance on scalable architectures.

In this work we use execution-driven simulation to investigate the access patterns and caching behaviour exhibited by a parallel logic programming system, Andorra-I [22], running on distributed shared-memory architectures. Andorra-I is an example of a system that can achieve good performance on bus-based shared-memory machines [31], but that has not previously been evaluated for scalable multiprocessors. Andorra-I is one of the most sophisticated parallel logic programming systems, exploiting both dependent and-, and or-parallelism. Most of Andorra-I's data structures are similar to the ones of several other parallel logic programming systems.

Our study evaluates Andorra-I on a DASH-like [17] machine, running very different applications, and under different cache coherence protocols. The results show that Andorra-I achieves reasonable performance, but it does not scale well. In order to fully understand these results, we study the behaviour of the major data structures in Andorra-I in great detail. Our analysis shows that Andorra-I exhibits widely-varying memory access patterns and caching behaviour, which not only depend on the number of processors, but also on the type and amount of parallelism available in the application program. In addition, we observe that some data structures clearly favour invalidate-based cache coherence protocols, while others favour update-based protocols. Thus, we believe that parallel logic programming systems can greatly benefit from flexible coherence schemes where either the compiler can specify the protocol to be used for each data structure or the protocol can adapt to varying memory access patterns.

Our approach contrasts with previous studies of the memory access behaviour of parallel logic programming systems. Tick and Hermenegildo [25] studied caching behaviour for independent and-parallelism. In their study, they concentrate on the overall behaviour of the system, towards designing efficient architectures for parallel logic programming systems. Other researchers have studied the performance of parallel logic programming systems on scalable architectures, such as the DDM [20], showing good scalability for highly-parallel benchmarks under the or-parallel system Aurora, a precursor of Andorra-I. Our goal in this study is to pinpoint the main sources of cache misses in a parallel logic programming system, in order to determine how coherence protocols and the different forms and amount of parallelism can affect system performance.

The paper is organised as follows. Section 2 presents the methodology used to obtain our results. We describe the Andorra-I parallel logic programming system, the simulator we used to perform the experiments, how Andorra-I was ported to the simulator, present the applications in our workload, and analyse their individual performance. Section 3 analyses the caching behaviour of our or-parallel, and-parallel, and combined parallelism applications under a write-invalidate protocol. Section 4 presents an analysis of the system's performance under write-update protocols. Finally, section 5 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.

2.1 Andorra-I

The Andorra-I parallel logic programming system is based on the Basic Andorra Model [29]. The system was developed at the University of Bristol by Beaumont, Dutra, Santos Costa, Yang, and Warren [22, 31]. 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 goals that match at most one clause in a program. Thus, Andorra-I exploits determinate dependent and-parallelism. Eager execution of determinate goals can result in a reduced search space, because unnecessary choicepoints are eliminated. The Andorra-I system also exploits or-parallelism that arises from the non-determinate goals. Its implementation is influenced by JAM [7] when exploiting and-parallelism, and by Aurora [18] when exploiting or-parallelism.

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 [23]. The engine is responsible for the execution of the Andorra-I programs [32]. The two schedulers manage and- and or-work. Andorra-I organises processors as teams, where processors within a team exploit and-parallelism together, and the teams work in or-parallel fashion. The reconfigurer [9] allows processors to adapt to the different sources of parallelism that arise during execution.

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

Workers should perform reductions most of the execution time, i.e. they should spend most of their time in *engine* code [32]. Andorra-I is designed so that data corresponding to each worker is as local as possible, so that each worker tries to find its own work without interfering with others. Scheduling in Andorra-I is demand-driven, that is, whenever a worker runs out of work, it enters a *scheduler* to find another piece of available work.

The or-scheduler is responsible for finding or-work, i.e. an unexplored alternative in the or-tree. Our experiments used the Bristol or-scheduler [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. Finally, the reconfigurer [9] is responsible for arranging the workers into teams in a way that allows both and- and or-parallelism to be freely exploited when they are available.

2.2 Multiprocessor Simulation

We use a detailed on-line, execution-driven simulator that simulates a 24-node, DASH-like [17], directly-connected multiprocessor. Each node of the simulated machine contains a single processor, a write buffer, a 64-KB direct-mapped data cache with 64-byte cache blocks, local memory, a full-map directory, and a network interface. The simulator was developed at the University of Rochester and uses the MINT front-end, developed by Veenstra and Fowler [26], to simulate the MIPS architecture, and a back-end, developed by Bianchini and Veenstra, to simulate the memory and interconnection systems.

In order to keep caches coherent we used write-invalidate (WI) [11] and write-update (WU) [19] protocols. In the WI protocol, whenever a processor writes a data item, copies of the cache block containing the item in other processors' caches are invalidated. If one of the invalidated processors later requires the same item, it will have to fetch it from the writer's cache. Our WI protocol keeps caches coherent using the DASH protocol with release consistency [16].

WU protocols are the main alternative to invalidate-based protocols. In WU protocols, whenever an item is written, the writer sends copies of the new value to the other processors that share the item. In our WU implementation, a processor writes through its cache to the home node. The home node sends updates to the other processors sharing the cache block, and a message to the writing processor containing the number of acknowledgements to expect. Sharing processors update their caches and send an acknowledgement to the writing processor. The writing processor only stalls waiting for acknowledgements at a lock release point.

Our WU protocol 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.

2.3 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 environment as cross development tools for this purpose. Andorra-I was compiled with the `-O2` option.

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 [15] to implement locks and atomic operations. Unfortunately, these instructions have not yet been implemented in the simulator. 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 synchronisation in Andorra-I is visible to the underlying runtime system. The only exception to this rule is the detection of the end of the determinate phase. Here we maintained the original code because any action after detection requires the slaves to grab a lock previously released by the master.

2.4 Workload

The benchmarks we used in this work are applications representing predominantly and-parallelism, predominantly or-parallelism, and both and- and or-parallelism. We tried to select applications with good parallelism, whilst avoiding very regular 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. Our results correspond to the *first* run of an application (results would be somewhat better for other runs).

And-Parallel Application. As an example and-parallel application, we used the clustering algorithm for network management from British Telecom [5]. 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 shows the `bt-cluster` speedups for write-invalidate and write-update 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 barely acceptable. The invalidate protocol (WI curve) starts with an efficiency of 85% for 2 processors. Efficiency smoothly decreases as the number of processors increases, becoming just over 50% for 16 processors. The update protocol (WU curve) has a very interesting behaviour. It starts with excellent efficiency of more than 90% up to four processors. Then performance starts degrading, and for 24 processors there is a decrease in speedup.

Or-Parallel Application. 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 [30]. 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

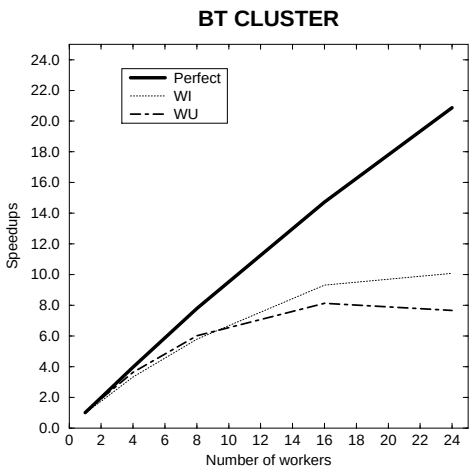

Figure 1: Speedups for `bt-cluster`

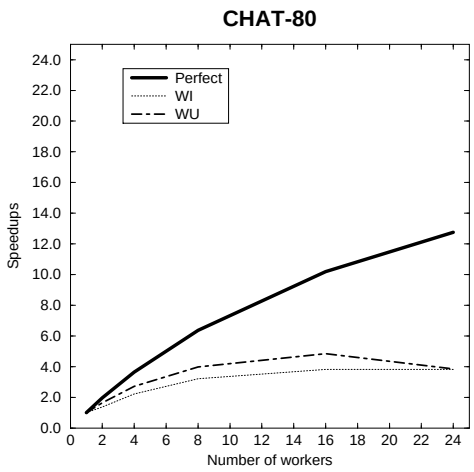

Figure 2: Speedups for `chat-80`

has been traditionally used as one of the or-parallel benchmarks for both the Aurora and Muse systems.

Figure 2 shows the speedups and execution time ratios for both protocols, from 1 to 24 processors. 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 [10]. 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 19% to 36% better for this benchmark, obtaining a maximum speedup of 4.9 for 16 workers. When the number of workers increases to 24, performance for the update protocol deteriorates, whereas the invalidate protocol manages to sustain performance.

And/Or-Parallel Application. We used a program to generate naval flight allocations, based on a system developed by Software Sciences and the University of Leeds for the Royal Navy. It is an example of a real-life resource allocation problem. The program allocates airborne resources (such as aircraft) whilst taking into account a number of constraints. The problem is solved by using the technique of active constraints as first implemented for Pandora [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 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 3 shows the speedups 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` [10].

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 4 workers, and then starts decreasing only to increase again for 24 processors.

3 Analysing the Caching Behaviour of Andorra-I

The major goal of our work was to determine which factors affect caching in Andorra-I, as this is one of the most important requirements for obtaining good performance in parallel systems. A parallel logic programming system includes several shared data structures, whose importance depends on the type and amount of parallelism available in the application. Our first goal was therefore to classify them and study their individual contributions to the overall caching behaviour.

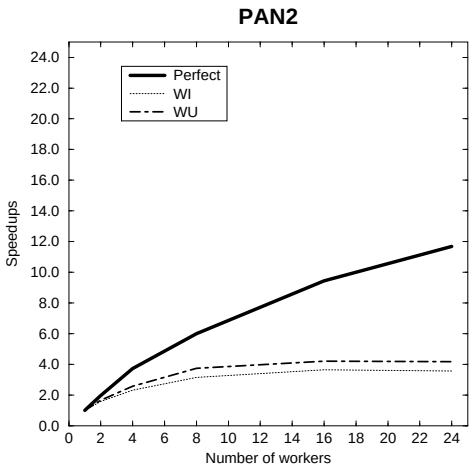

Figure 3: Speedups for pan2

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 [18], themselves based on sequential Prolog [27]. 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 [32]). 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 [28] 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 the `chat-80`, `bt-cluster` and pan2 benchmarks to collect statistics for each data area in turn. The results of this analysis are presented in the next sections.

3.2 Determining Memory Access Patterns

Area	bt-cluster	chat-80	pan2
OrSched	1.7	58.8	3.0
Worker	19.3	7.4	33.4
Locks	3.1	1.8	3.6
Code	39.7	10.2	18.0
Heap	7.6	1.7	4.0
Goals	12.6	3.6	6.8
ChoicePt	0.0	7.8	0.1
Trail	0.1	2.3	0.2
BA	6.2	3.3	20.0
Misc	8.8	3.5	9.4

Table 1: References per Shared Area (%)

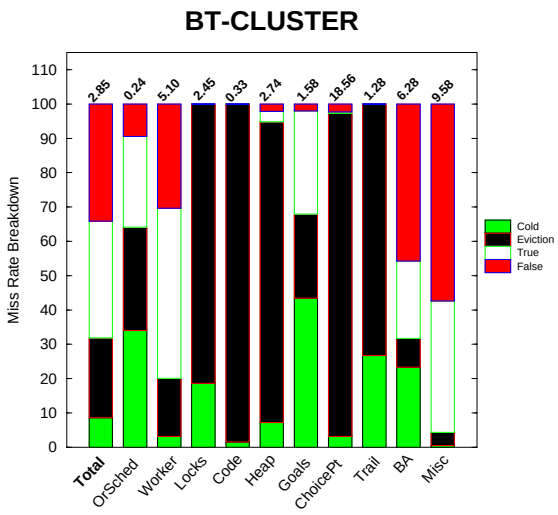

Figure 4: Classification of Misses for `bt-cluster` under WI

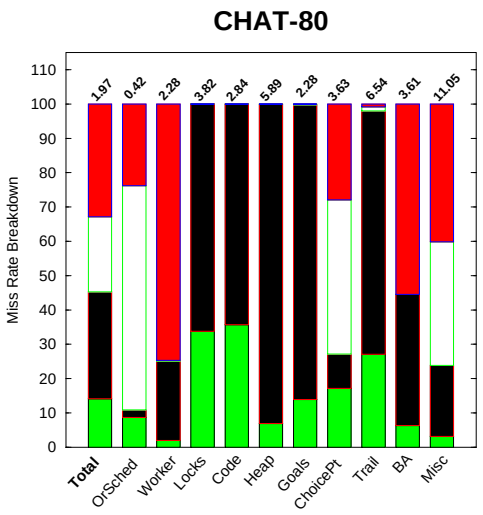

Figure 5: Classification of Misses for `chat-80` under WI

In order to establish the relative importance of each data section, table 1 shows the percentage of the total number of references per each shared area for 16 workers. We chose this number of workers in order to study the memory traffic in a situation where the system is starting to saturate. The results in the table vary widely for the different data areas, depending on the type and on the amount of parallelism in the application. Most references in `bt-cluster` are concentrated in the `Code`, `Worker`, `Goals`, and `Misc` data areas. The high number of references to engine data areas, such as `Code` and `Goals`, indicates that processors are running engine code most of the execution time, suggesting that there is sufficient parallelism to sustain 16 processors. In `chat-80`, the most referenced areas are `OrSched`, `Code`, `ChoicePt`, and `Worker`. This indicates that there is not sufficient work for sustaining 16 workers, and therefore most of the time is spent in the or-scheduler. The `pan2` benchmark shows little or-parallelism (almost no accesses to the `ChoicePt` or `Trail` areas) and a significant percentage of accesses to the areas related to and-scheduling: `Worker` and `Misc`. This confirms that `pan2` is mostly an and-parallel benchmark, but that the parallelism is not sufficient for 16 workers. Similarly to the other two applications, the `Code` area is also heavily referenced in `pan2`. The high percentage of accesses to the `BA` area in `pan2` is intriguing, and is discussed in further detail in the BA analysis.

To fully understand the caching behaviour within each data area, we classified their respective percentage of shared read misses[1]. This classification uses the algorithm described in [8], as extended in [4]:

- **Cold start misses.** A cold start miss happens on the first reference to a cache 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 that has been invalidated due to a write to the same word by some other processor .
- **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 referenced by the missing processor.
- **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.

Figures 4, 5, and 6 show the classification of shared read misses per area in `bt-cluster`, `chat-80`, and `pan2`, respectively. The leftmost column in the figures shows the total miss rate and the other columns show the classification for each individual area. The number on top of each column corresponds to the read miss rate for the respective area. Note that the regions with the most references are the ones we would like to see the smallest miss rates for.

Among the most important areas in number of accesses for `bt-cluster`, `Code` exhibits a very low miss rate, which is dominated by eviction misses. Both `Worker` and `Misc` exhibit substantial sharing miss rates, a large percentage of which due to false sharing. `Goals` has the most complex behaviour, with a significant percentage of cold, eviction, and true sharing misses. Among the most important areas in number of references for `chat-80`, all except `Code` exhibit a large percentage of sharing misses, while all except `OrSched`

[1] We focus solely on read misses, since write misses are usually hidden from the processor by the write buffer.

exhibit a non-negligible miss rate. In pan2, Code is again the only important area that is not dominated by sharing misses. The Code miss rate is very small however.

In summary, the Locks, Code, and Heap areas always exhibit a significant percentage of eviction misses, whilst Worker, BA, and Misc are always dominated by sharing misses. Note that, for the Worker, BA, and Misc areas, a significant portion of these sharing misses can potentially be avoided by using a more careful data layout, as they are caused by false sharing. In the next few sections we study each data area in detail.

Or-Scheduler The OrSched area is important for chat-80, where it concentrates most of the references. It is referenced in pan2, during the final execution stage, when workers try to reconfigure from and-work to or-work. It has little significance in bt-cluster. Scheduling involves significant sharing; in chat-80 roughly 90% of all read misses in this area are sharing misses, mostly due to true sharing.

Worker Data Structures Sharing in this area results mostly from synchronisation within teams and and-scheduling. We expect no and-scheduling in chat-80, a bit in bt-cluster, and heavy in pan2 where work is not sufficient for 16 workers.

In chat-80 there is no true sharing, but considerable false sharing in Worker, since worker data structures are not always aligned with the cache line boundaries. And-scheduling causes heavy true sharing in this area for pan2 on 16 processors. The area is dominated by true sharing in bt-cluster, although the false sharing miss rate is also significant.

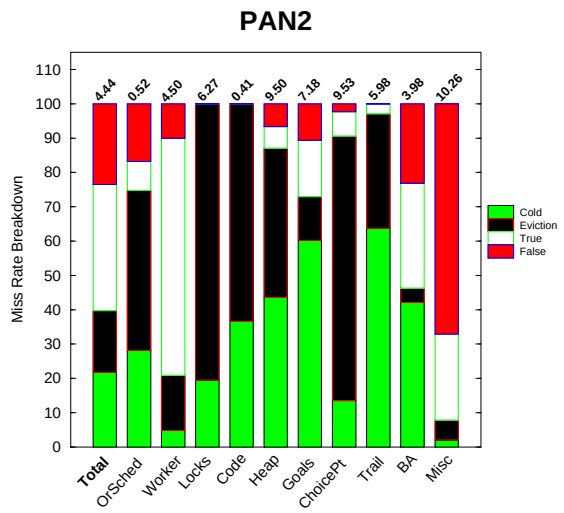

Figure 6: Classification of Misses for pan2 under WI

Lock Array and Code Space Both the Locks and Code areas are read-only during execution of the benchmarks. At least for determinate applications, the Andorra-I engine issues more references to the code than to the stacks, hence the Code area takes a substantial percentage of the total number of references; up to 40% in bt-cluster. The miss rate of this area is usually very low, less than 1% for both bt-cluster and pan2. The exception is chat-80, which has large compiled code (including the determinacy code for the database) with poor temporal locality.

Heap Space The Heap area is fundamentally private in or-parallelism, hence the very low sharing miss rates for chat-80. In bt-cluster and pan2, most accesses are to the heap positions created by the workers themselves, and only a small fraction of accesses is used for communication between workers. As a result, even for the programs which do share logical variables, the true sharing misses are a small percentage (at most 6%) of the total heap misses.

Notice however the very different behaviour of the two applications with and-parallelism, pan2 and bt-cluster, for this area. bt-cluster uses a fixed data structure and builds few terms in the Heap area, so it has good locality and a relatively low miss rate. On the other hand, pan2 builds a large number of terms during execution, resulting in a much worse miss rate, with a relatively high percentage of cold misses.

Goal Stack There is no sharing in chat-80 for this area, hence it exhibits almost no sharing misses for this application. However, there is sharing in and-parallel applications because processors need to steal goals from other processors to start work. False sharing exists either because two different goals may share a cache line, or because different processors may update different fields in a goal frame.

Notice that bt-cluster has a much lower miss rate than pan2 for this area. Andorra-I does not reuse goal frame space in pan2, since it can only recover goal frame space if the reduced goal is on the top of the stack. If an application suspends very often or has fine-grained work, which is the case for pan2 but not for bt-cluster, the current implementation of Andorra-I cannot recover goal frames. As a result, pan2 has both more sharing and many more cold misses than bt-cluster.

Choicepoint Stack The choicepoint stack accounts for a significant portion of the memory references only for chat-80. In this application, workers move a lot in the search tree, since the scheduler is exploiting fine-grained work. As a result, chat-80 exhibits significant sharing in this area. For the other applications, eviction misses dominate in ChoicePt.

Trail Stack The trail is seldom referenced in bt-cluster and pan2, where most of the execution is determinate. It is slightly more important in the or-parallel benchmark, chat-80. In this benchmark, there is little sharing and quite a lot of eviction misses. There is only a small number of sharing misses in chat-80 because the processor that creates a trail segment rarely overwrites it and the other processors simply read the data.

Chat-80 exhibits a large number of eviction misses since it has trouble reusing trail space. More specifically, when work is coarse-grained, processors work mostly on their private trails and reuse the trail space after backtracking. However, for 16 processors work is very fine-grained in chat-80, making it difficult to recover space during backtracking and forcing processors to use more trail space.

Binding Array This area has surprisingly high false sharing in `chat-80`. The area is private for or-parallelism, and we found out that the false sharing is caused by accesses to shared pointers to the top of stacks, which are allocated individually and placed between the BA segments. As the memory allocator does not pad the allocated areas to the end of a cache line, several of these pointers end up co-located in the same line. These pointers should logically be part of the `Worker` area.

The area is shared for applications with and-parallelism where it shadows the heap and especially the global stack, which entails the most traffic. Note that this area has much more false sharing than `Goals`. We conjecture two reasons for this behaviour. First, migratory sharing of pointers to the top of stacks that are shared within a team. Second, the `BA` shadows variables that are only a part of the goal-frame structure. Variables from different goals will thus be packed on the same cache line, generating more false sharing.

Miscellaneous Shared Variables The `Misc` collection of variables is responsible for a significant number of references for and-parallel benchmarks. We believe the reason is that this area includes several counters that belong to a team. These counters are actively updated by all members of a team and result in high levels of sharing, especially for the and-parallel applications.

4 Andorra-I Caching Behaviour Under WU Protocols

We have so far studied Andorra-I under a WI protocol. The significant amount of sharing in some of Andorra-I's data areas suggests that protocols optimised for data sharing, such as WU protocols, might be an interesting alternative to WI protocols for parallel logic programming systems [21].

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. In the next section we investigate in detail performance of Andorra-I under a WU protocol.

4.1 Categorising Misses Under WU Protocols

Figures 7, 8, and 9 show the classification of shared read misses under WU for the `bt-cluster`, `chat-80`, and `pan2` applications, respectively. A comparison between these figures and the ones in section 3.2 shows that there is a very impressive difference in total miss rates between WU and WI: WU improves the miss rate by 64% for `bt-cluster`, 51% for `chat-80`, and 57% for `pan2`. Most of the improvement is obtained in the areas with significant sharing under WI, such as `OrSched` in `chat-80`, and `Worker` in `bt-cluster` and `pan2`. In general, the heavier the sharing, the greater the improvement. However, under WU, items tend to stay longer in caches, resulting in more eviction misses. This explains why WU has worse miss rates for read-only areas, such as `Code` and `Locks`.

4.2 Categorising Updates

The question is whether these low miss rates are obtained at the cost of transferring an excessive amount of (mostly useless) data through the interconnection network. In order to assess the amount of useless traffic involved in each of the data areas, we categorise updates using the algorithm

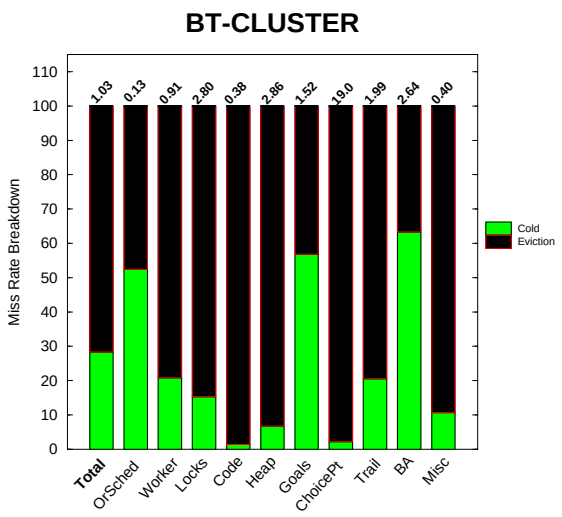

Figure 7: Classification of Misses for `bt-cluster` under WU

described in [4]. We define an update to be *useful* when the updated word is actually referenced by the receiving processor, before getting overwritten by some other update to the same word. Table 2 presents the percentage of all updates that is directed to each area (left column) and the percentage of these updates that is actually useful (right column). "N/A" means no update messages are generated (read-only data area).

Area	bt-cluster		chat-80		pan2	
OrSched	0.0	2.4	17.3	9.7	0.2	10.9
Worker	43.6	3.2	9.6	0.1	56.1	6.7
Locks	0.0	N/A	0.0	N/A	0.0	N/A
Code	0.0	N/A	0.0	N/A	0.0	N/A
Heap	0.0	52.4	1.3	0.6	1.8	9.3
Goals	17.3	1.4	4.7	0.2	7.1	5.5
ChoicePt	0.0	1.5	24.9	5.3	0.0	13.9
Trail	0.0	0.0	6.4	2.2	0.2	8.7
BA	27.0	0.8	9.7	0.0	21.7	3.9
Misc	13.2	4.2	18.6	6.5	13.2	5.9
Total	100.0	2.5	100.0	4.1	100.0	6.0

Table 2: Percentage of Updates and Ratio of Useful Update Messages (%) on 16 processors

As shown in table 2, the vast majority of updates is indeed useless for all applications and data areas on 16 processors, except for the `Heap` area in `bt-cluster`.

The fact that most updates on a 16 processor machine are useless shows that there is a clear tradeoff between cache miss service latency and communication bandwidth when deciding whether to use WI or WU for parallel logic pro-

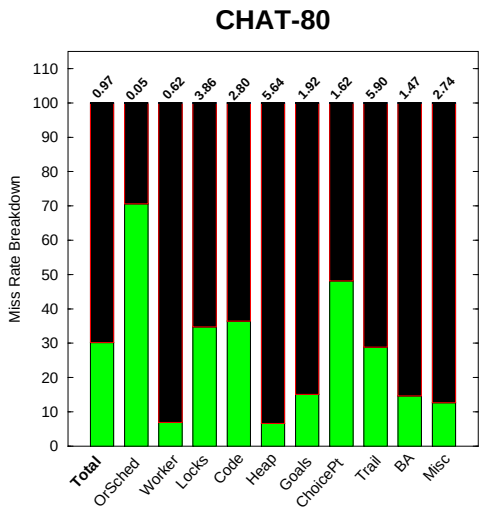

Figure 8: Classification of Misses for `chat-80` under WU

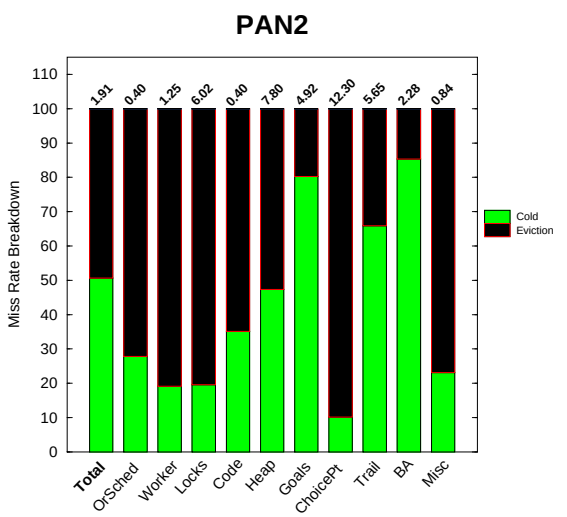

Figure 9: Classification of Misses for `pan2` under WU

gramming systems. WU should be used instead of WI when sending more messages does not cause more significant performance degradation than taking a larger number of cache misses. In addition, WU should only be used when it can provide significant reductions in miss rates for reference-intensive data areas. According to this criterion, WU would be effective for `Worker`, `BA`, and `Misc` for `bt-cluster` and `pan2`, and `OrSched`, `Worker`, and `ChoicePt` for `chat-80`.

5 Conclusions and Future Work

We used execution-driven simulation to investigate the performance of the parallel logic programming system Andorra-I on a distributed shared-memory architecture. The results show that the system performs well for a small number of processors, but that it has difficulties scaling up. To understand this behaviour, we analysed system performance in detail for 16 processors.

One of the major contributions of our analysis is showing the need for careful scheduling and work manipulation in parallel logic programming systems. Contrary to our initial expectations, much more than sharing of logical variables, scheduling and work manipulation seem to be the decisive factors limiting performance, and will be the key areas to improve in order to obtain scalable performance.

The other major contribution is allowing us to understand the access patterns and caching behaviour for each data area in Andorra-I. This is fundamental for obtaining the best data layout and to research appropriate coherence protocols. In particular, our study enables us to determine whether a write-update or a write-invalidate protocol performs best, as a function of data area and type of parallelism. In fact, the widely-varying results we obtained for the Andorra-I data areas and different applications suggest that flexible (area-dependent) protocols should deliver the best performance for parallel logic programming systems. While machines with such protocols are not currently available, our results show that hybrid (WU+WI) protocols are a robust solution for the moment. We have confirmed this claim through other experiments [21].

We believe that the techniques we used should be considered by anyone developing logic programming or other parallel symbolic systems for distributed shared-memory machines. By understanding individual areas we can determine what aspect of the system to optimise first, and avoid running into blind alleys. However, the complexity of parallel logic programming systems suggests that one should be careful about generalising the actual results we obtained.

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 and MELODIA projects. Inês Dutra would like to thank support from the PROLOPPE project. The authors would also like to thank the Brazilian Research Council, CNPq.

References

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

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

[3] Beaumont, A., Raman, S. M., and Szeredi, P. Flexible Scheduling of Or-Parallelism in Aurora: The Bristol Scheduler. In

PARLE91: Conference on Parallel Architectures and Languages Europe (June 1991), Aarts, E. H. L. and van Leeuwen, J. and Rem, M., Ed., vol. 2, Springer Verlag, pp. 403–420. Lecture Notes in Computer Science 506.

[4] Bianchini, R., and Kontothanassis, L. I. Algorithms for categorizing multiprocessor communication under invalidate and update-based coherence protocols. In *Proceedings of the 28th Annual Simulation Symposium* (April 1995).

[5] Crabtree, B. A clustering system to network control, British Telecom, March 1991.

[6] Crammond, J. A. *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] Crammond, J. A. The Abstract Machine and Implementation of Parallel Parlog. Tech. rep., Dept. of Computing, Imperial College, London, June 1990.

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

[9] Dutra, I. C. Strategies for Scheduling And- and Or-Work in Parallel Logic Programming Systems. In *Proceedings of the 1994 International Logic Programming Symposium* (1994), MIT Press, pp. 289–304. Also available as technical report CSTR-94-09, from the Department of Computer Science, University of Bristol, England.

[10] Dutra, I. C. *Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System.* PhD thesis, University of Bristol, Department of Computer Science, February 1995.

[11] Goodman, J. R. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th International Symposium on Computer Architecture* (1983), pp. 124–131.

[12] Hermenegildo, M. V., and Greene, K. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Proceedings of the Seventh International Conference on Logic Programming* (June 1990), MIT Press, pp. 253–268.

[13] Ito, N., Sato, M., Kishi, A., Kuno, E., and Rokusawa, K. The Architecture and Preliminary Evaluation Results of the Experimental Parallel Inference Machine PIM-D. In *Proc. of the 13th Annual International Symposium on Computer Architecture* (June 1986).

[14] Kacsuk, P., and Wise, M. J., Eds. *Implementations of Distributed Prolog.* Wiley, Series in Parallel Computing, 1992.

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

[16] Lenoski, D., Laudon, J., Gharachorloo, K., Gupta, A., and Hennessy, J. The directory-based cache coherence protocol for the DASH multiprocessor. *Proceedings of the 17th ISCA* (May 1990), 148–159.

[17] Lenoski, D., Laudon, J., Joe, T., Nakahira, D., Stevens, L., Gupta, A., and Hennessy, J. The dash prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems 4*, 1 (Jan 1993), 41–61.

[18] Lusk, E., Warren, D. H. D., Haridi, S., et al. The Aurora Or-parallel Prolog System. *New Generation Computing 7*, 2,3 (1990), 243–271.

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

[20] Raina, S., Warren, D. H. D., and Cownie, J. Parallel Prolog on a Scalable Multiprocessor. In *Implementations of Distributed Prolog*, P. Kacsuk and M. J. Wise, Eds. Wiley, 1992, pp. 27–44.

[21] Santos Costa, V., Bianchini, R., and Dutra, I. C. Evaluating the impact of coherence protocols on parallel logic programming systems. In *Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed Processing* (1997), pp. 376–381. Also available as technical report ES-389/96, COPPE/Systems Engineering, May, 1996.

[22] Santos Costa, V., Warren, D. H. D., and Yang, R. 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* (April 1991), ACM press, pp. 83–93. SIGPLAN Notices vol 26(7), July 1991.

[23] Santos Costa, V., Warren, D. H. D., and Yang, R. The Andorra-I Preprocessor: Supporting full Prolog on the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming* (1991), MIT Press, pp. 443–456.

[24] Shapiro, E. The family of Concurrent Logic Programming Languages. *ACM computing surveys 21*, 3 (1989), 412–510.

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

[26] Veenstra, J. E., and Fowler, R. J. 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).

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

[28] Warren, D. H. D. The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues. In *Proceedings of the 1987 International Logic Programming Symposium* (1987), pp. 92–102.

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

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

[31] Yang, R., Beaumont, T., Dutra, I., Santos Costa, V., and Warren, D. H. D. Performance of the Compiler-Based Andorra-I System. In *Proceedings of the Tenth International Conference on Logic Programming* (June 1993), MIT Press, pp. 150–166.

[32] Yang, R., Santos Costa, V., and Warren, D. H. D. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming* (1991), MIT Press, pp. 825–839.