Parallel Logic Programming Systems on Scalable Architectures[1]

Vítor Santos Costa[2] and 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

Parallel logic programming (PLP) systems are sophisticated examples of symbolic computing systems. They address problems such as dynamic memory allocation, scheduling irregular execution patterns, and managing different types of implicit parallelism. Most PLP systems have been developed for bus-based shared-memory architectures. The complexity of PLP 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 (DSM) systems.

In this work we use execution-driven simulation of a DASH-like architecture to investigate the performance of the PLP system Andorra-I on a (DSM) system. Initially, Andorra-I obtains reasonable performance, but it does not scale well. Our detailed analysis exposes several opportunities for improvements in Andorra-I. We then apply a set of techniques based on this analysis to obtain excellent results in the simulated machine. Since Andorra-I shares its main data-structures with other PLP systems, we further conclude that the methodology and techniques used in our work can greatly benefit most PLP systems.

Key Words: DSM architectures, logic programming

CONTENTS

[1]This work was sponsored by CNPq, Brazilian Research Council.
[2]On leave from the Universidade do Porto, Portugal.

1. INTRODUCTION

Parallel logic programming (PLP) systems are sophisticated examples of symbolic computing systems. They address problems such as dynamic memory allocation, scheduling irregular execution patterns, and managing different types of implicit parallelism. In fact, one of the most important advantages of logic programming is the availability of several forms of implicit parallelism that can be naturally exploited on shared-memory multiprocessors. These forms include: or-parallelism, as exploited in the systems Aurora [17] and Muse [2]; independent and-parallelism, as in &-Prolog [12] and &-ACE [11]; dependent and-parallelism, as in Parlog's JAM [7], KLIC [28], and DASWAM [25, 26]; data-parallelism, as in Reform Prolog [4]; and combined and–or parallelism, as in Andorra-I [24] and Penny [18]. All these systems have been able to obtain almost linear speedups at least up to twenty CPUs on bus-based systems, such as the Sequent Symmetry multiprocessors [16].

The complexity of PLP 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 (DASH [15], Alewife [1], Exemplar [6] or Origin [14]). In distributed shared-memory systems, performance depends heavily on the read miss rates and may be limited by the communication overhead caused by coherence messages needed to maintain local copies of shared data consistent.

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 is the writer and the processors that consult the binding the readers.

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 processes, 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 read and then modified by one of the idle processors.

In this work we experiment with a sophisticated PLP system, Andorra-I [24], that exploits both dependent and-parallelism and or-parallelism. Andorra-I is one of the most sophisticated PLP systems that have been built to date. Original work with Andorra-I obtained good performance on the Sequent Symmetry '[34], but our initial experience with Andorra-I running on modern multiprocessors demonstrates indeed that scalability suffers greatly on these architectures [22].

This paper addresses the question of whether the poor scalability of Andorra-I is inherent to the complex structure of PLP systems or can be improved through careful analysis and tuning. In order to answer this question, we analyse the caching behaviour of all Andorra-I data areas when applied to several different logic programs. The analysis pinpoints the areas that are responsible for most misses and the main sources for these misses. Based on this analysis we remove the main performance limiting factors in Andorra-I through a set of optimisations that did not require a redesign of the system. More specifically, we optimise Andorra-I using 5 different techniques: trimming of shared variables, data layout modification, privatisation of shared data structures, lock distribution, and elimination of locking in scheduling.

We present the isolated and combined performance improvements provided by the optimisations on a simulated DASH-like multiprocessor with up to 24 processors. In isolation, shared variable trimming and the modification of the data layout produced the greatest improvements. The improvements achieved when all optimisation techniques are combined are substantial. A few of our programs approach linear speedups as a consequence of our modifications. In fact, for one program the speedup of the modified Andorra-I is a factor of 3 higher than that of the original version of the system on 24 processors. Our main conclusion is then that, even though Andorra-I is indeed complex and irregular, it can and should scale well on modern DSM multiprocessors.

This conclusion can be extrapolated to several other PLP systems, since they share most of Andorra-I's data structures. More specifically, single processor execution in Andorra-I is based on Warren's WAM [31], the abstract machine of most other sequential and PLP systems. Thus, Andorra-I's major abstract machine data structures can be found in these systems. Parallel execution in Andorra-I applies the implementation technology developed for or-parallel systems such as Aurora [17], and for and-parallel systems such as PARLOG [7]. Thus, Andorra-I's major parallelism-related data structures can be found in these PLP systems. Results obtained from Andorra-I are therefore of immediate interest to most PLP systems, making Andorra-I the most interesting system for studies such as ours.

Our approach contrasts with previous studies of the performance of parallel logic programming systems. Tick and Hermenegildo [29] studied caching behaviour of independent and-parallelism in bus-based multiprocessors. More recently, Montelius used SimICS, a simulator of SUN bus-based parallel machines, to improve the Penny And/Or parallel system [19]. The simulator will output how individual lines in the program contribute to the miss-rate. In the case of Penny, most misses originated from one line in the garbage collector, allowing for a simple optimisation. Regarding DSM machines, Raina et al. did a first study of the performance of Aurora and interpreted Andorra-I on the DDM emulator [21], one of the first Cache-Only Memory Architecure (COMA) machines. The work presents speedups, miss rates and network traffic for a 4 node configuration.

Our interest in simulating stemed from the observation that during forward execution the logical variable will be updated at most once. We thus compared Andorra-I under the impact of different cache coherence protocols to see whether update-based protocols would be most appropriate for Andorra-I [22]. Our results did not favour updates, as we had originally expected. In order to understand these results we performed an area-based analysis [23]. That work was the basis for this work. More recently, Silva has used the optimised version to study the impact of several architectural parameters such as cache sizes and cache block sizes on the Andorra-I system [27].

The remainder of the paper is organised as follows. Section 2 presents the methodology used to obtain our results. We describe the Andorra-I PLP system, the simulator we used to perform the experiments, and how Andorra-I was ported to the simulator. In Section 3 we analyse in detail the initial speedup performance of each application we consider. Section 4 introduces the major optimisations and discusses their individual contribution. We present the performance of the optimised system in Section 5. Finally, Section 6 draws our conclusions and suggests future work.

2. METHODOLOGY

In this section we detail the methodology used in our experiments. The experiments consisted of the simulation of the parallel execution of Andorra-I, compiled for the MIPS architecture [13].

2.1. Multiprocessor Simulation

We use a detailed on-line, execution-driven simulator that simulates a 24-node, DASH-like [15], directly-connected multiprocessor. Each node of the simulated machine contains a single processor, a write buffer, cache memory, local memory, a full-map directory, and a network interface. The simulator was developed at the University of Rochester and uses the MINT front-end [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 128-KB direct-mapped data cache with 64-byte cache blocks. All instructions and read hits are assumed to take 1 cycle. Read misses stall the processor until the read request is satisfied. Writes go into a 16-entry write buffer and take 1 cycle, unless the write buffer is full, in which case the processor stalls until an entry becomes free. Reads are allowed to bypass writes that are queued in the write buffers. Shared data are interleaved across the memories at the block level.

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

The interconnection network is a bi-directional wormhole-routed mesh, with dimension-ordered routing. The network clock speed is the same as the processor clock speed. Switch nodes introduce a 4-cycle delay to the header of each message. 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.

All hardware characteristics mentioned above are common in actual modern parallel architectures.

In order to keep caches coherent we used a write-invalidate (WI) protocol [10]. 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 [15].

2.2. Andorra-I

The Andorra-I parallel logic programming system is based on the Basic Andorra Model [33]. The system was developed at the University of Bristol by Beaumont, Dutra, Santos Costa, Yang, and Warren [24, 34]. To the best of the authors' knowledge, Andorra-I was the first parallel logic programming system that exploited both and- and or-parallelism, and yet could run real-world applications with significant parallel performance.

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 [8] when exploiting and-parallelism, and by Aurora [17] 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 data structures used in a given branch of the search tree.

Most of the execution time of workers should be spent executing *engine* code [35], i.e., performing reductions. Andorra-I is designed in such a way that data corresponding to each worker is as local as possible, so that each worker tries to find its own work without interfering with others. Scheduling in Andorra-I is demand-driven, that is, whenever a worker runs out of work, it enters a *scheduler* to find another piece of available work.

The or-scheduler is responsible for finding or-work, i.e., an unexplored alternative in the or-tree. Our experiments used the Bristol or-scheduler [3], originally developed for Aurora. In these systems, scheduling is performed when workers run out of branches to exploit in the search tree. The principle is for the idle worker, say W_i, to query the other workers for available work by consulting a shared data-structure. If another worker, say W_j, does have work, it makes its current branch in the search tree *public*, and the first worker, W_i, can move into W_j's area of the tree and start executing new alternatives. The scheduling algorithm thus both requires synchronisation on the shared data-structures that are used to represent worker status, and synchronisation on the search tree itself, as several workers may be moving in the same area of the search-tree.

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.

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.

3.1. Data Areas in Andorra-I

Andorra-I combines the techniques used to implement parallel committed-choice systems [7] and the techniques used to implement or-parallel systems [17], themselves based on sequential Prolog [31]. 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 into 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 [35]). 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 [32] 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 applications to collect statistics for each data area in turn.

3.2. Workload and Original Performance

The benchmarks we used in this work are applications representing predominantly and-parallelism, predominantly or-parallelism, and both and- and or-parallelism. We next discuss application performance for the original Andorra-I. Note that our results correspond to the *first* run of an application; results would be somewhat better for other runs.

And-Parallel Applications.

We use two example and-parallel applications, the clustering algorithm for network management from British Telecom, `bt-cluster`, and a program to calculate approximate solutions to the traveling salesperson problem, `tsp`. The clustering program receives a set of points in a three dimensional space and groups these points into *clusters*. Basically, three points belong to the same cluster if the distance between them is smaller than a certain limit. And-parallelism in this case naturally stems from running the calculations for each point in parallel. The test program uses a cluster of 400 points as input data. This program has very good and-parallelism, and, being completely determinate, no or-parallelism. The traveling salesperson program is based on a Reform Prolog [4] benchmark that finds an approximate solution for the TSP problem in a graph with 24 nodes. To obtain best performance, the Andorra-I team rewrote the original applications to make them determinate-only computations.

Figure 1 shows the `bt-cluster` speedups for the simulated architecture as compared to an idealised shared-memory machine, where data items can always be found in cache. The `idealised` curve shows that the application has excellent and-parallelism and can achieve almost linear speedups up to twenty four processors. Unfortunately, performance for the DASH-like machine is barely acceptable. The DASH curve starts with an efficiency of 85% for 2 processors. Efficiency smoothly decreases as the number of processors increases, reaching 70% for 16 processors. Efficiency is less than 50% for 24 processors. Figure 2 shows that the `tsp` application achieves worse speedups than `bt-cluster` on a modern multiprocessor. The maximum speedup actually decreases for 24 processors, whereas the ideal machine would achieve a speedup of 20 for 24 processors.

Figure 3 illustrates the number and sources of cache misses per data area in the `bt_cluster` application running on 16 processors as a representative example. The figure shows that the overall miss rate of `bt-cluster` is dominated by true and false sharing misses from the `Worker` and `Misc` areas. This suggests that the system could be much improved by reducing false sharing and studying activity in the `Worker` and `Misc` areas.

Or-Parallel Applications.

We use two or-parallel applications. Our first application, `chat80`, is an example from the well-known natural language question-answering system `chat-80`, written at the University of Edinburgh by Pereira and Warren. This version of `chat-80` operates on the domain of world geography. The program `chat80` makes queries to the `chat-80` database. This is a small scale benchmark with good or-parallelism, and it has been traditionally used as one of the or-parallel benchmarks for both the Aurora and Muse systems. The second application, `fp`, is an example query for a knowledge-based system for the automatic generation of floor plans. This application should at least in principle have significant or-parallelism. Figure 5 shows the speedups for the `chat80` application from 1 to 24 processors. In contrast, the DASH curve reaches a maximum speedup of 4.2 for 16 processors. Figure 6 shows the speedups for the `fp` application. The theoretical speedup is very good, in fact quite close to linear, in sharp contrast to the actual speedup for the DASH-like machine. Figure 4 shows the number and source of misses for `chat80` running on 16 processors, again as an example of this type of application. Note the high number of false sharing misses from the `Worker` and `Misc` regions, two regions that should not present a substantial contribution. We were surprised by the significant number of false sharing misses in the BA area, as this region should be private in ORP applications. Note that `Misc` and `OrSch` areas are the main sources for true sharing misses. Eviction misses are significant in `Worker`, and in the `Code` and `Trail` data areas. The `Worker` area is allocated statically, indicating the cache is over-stretched. We were expecting a large number of misses in the `Code` area, as this area stores a database, and in `Trail`, as the system must perform a large number of conditional bindings. The small contribution of the `OrSch` and `ChoiceP` data areas indicates that there is little scheduling activity, and suggests that good parallelism is available.

4. OPTIMISATION TECHNIQUES AND PERFORMANCE

The previous analysis suggests that relatively high miss rates may be causing the poor scalability of Andorra-I. It is interesting to note that most misses come from fixed layout areas, such as `Worker` and `Misc`, and not from the execution stacks, as one would assume.

We next discuss how several optimisations can be applied to the system, particularly in order to improve the utilisation of the `Worker` and `Misc` areas. The first two optimisations were prompted by our simulation-based analysis of caching behaviour, and they are the ones that give the best improvement. The other three were based on our original intuitions regarding the system, and were the ones we would have performed first without simulation data. We studied performance for two applications, `bt-cluster` and `chat80`.

In the remainder of this section, we study the impact of each of the techniques studied when applied in isolation.

Variable Trimming. In this technique we investigate the areas that have unexpected sharing, and try to eliminate this sharing if possible. For two of our applications, the `Misc` area gave a surprisingly significant contribution to the number of misses. The area is mostly written at the beginning of the execution to set up execution parameters. During execution it is used by the reconfigurer and to keep reduction and failure counters. We performed binary search on this area until we detected that the counters were the major source of misses. As they are only used for research and debugging purposes, we were able to eliminate them from the Andorra-I code.

Figure 7 shows the result of this optimisation for `bt-cluster` and `chat80` The figure plots the variation in speedup as a function of the number of processors. The results show that `chat80` benefits the most from this optimisation. This is because the failure counter is never updated by and-parallel applications, but often updated by this or-parallel application. The optimisation does not impact the and-parallel benchmarks as much, leading to less than 10% speedup improvements.

Data Layout Modification. Figure 8 shows the impact of the data layout modifications on the speedup of our applications. The `bt-cluster` and `chat80` applications benefit the most from this optimisation. Note that, on machine configurations of up to 16 processors, `chat80` benefits from this optimisation more than `bt-cluster` does, suggesting that the false sharing was having a big negative impact on this application until its lack of enough parallelism started to take over. On the largest machine configuration however, the optimisation actually becomes more beneficial to `bt-cluster`, an and-parallel application with a much greater amount of parallelism.

All benchmarks exhibit a high false sharing rate, showing a need for this technique. On 16 processors, false sharing causes between 40% (`chat80`) and 51% (`bt-cluster`) of all misses. These results suggest that improving false sharing is of paramount importance. According to our detailed analysis of caching behaviour, false sharing misses are concentrated in the `Worker`, `OrSch`, `ChoiceP` and `BA` areas, besides the `Misc` area optimised by the previous technique.

The `Worker` and `OrSch` data areas are allocated statically. This indicates that we can effectively reduce false sharing. We applied two common techniques to tackle false sharing, *padding* between fields that belonged to different workers or that were logically independent, and *field reordering* to separate fields that were physically close but logically distinct. Although these are well-known techniques, padding requires careful analysis, as it can increases eviction misses significantly. Because the main data-structures are replicated for each worker, we first added a padding field between each processor's component. The field reordering technique was not easily applied either, as the relationships between fields are quite complex. In a first step, we organised the code by functionality, say, so that all fields having to do Heap management would be together and separate from the fields that support trail or goal frame management. This increases the probability that when we fetch one of these fields we will also obtain the others, and reduces false sharing between the main fields in these data structures. In a second step, we used binary search to find out the major fields for each data structure, and experimented empirically with their allocation. In the case of `OrSch` we found there would be dependences between these fields, and that it would be advantageous to have them close together. Last, we tried added more padding, but we found this would actually decrease performance.

Regarding the dynamic data areas, such as `ChoiceP` and `BA`, padding may lead to serious performance degradation. This restricted our options for layout modification to just field reordering for these areas.

Last, the `BA` area was the target of one final data layout modification, since the analysis of cache behaviour surprised us with a high number of false sharing misses in this area for `chat80`. Further investigation showed that this was a memory allocation problem. The engines' top of stacks were being *shmalloc'ed* separately and appeared as part of the `BA` area in the analysis. This increased sharing misses in the area and was especially bad for the or-parallel applications, as different processor's top of stacks would end up in the same cache line. We addressed the problem by moving these pointers into the `Worker` area, where they logically belong. Our results show that the `bt-cluster` and `chat80` applications benefit the most from this optimisation; speedup improvements can be as significant as 60%.

Privatisation of Shared Variables. This technique reduces the number of shared memory accesses by making local copies in each node of the machine. In the best case, the shared variables are read-only and hence local copies can actually be allocated in private memory. The two major read-only areas during execution are the `Locks` and `Code` data areas. The `Locks` area only includes pointers to the actual locks, is thus read-only during execution, and can easily be privatised. The `Code` area is read-only during parallel execution of our benchmarks but logic programs in general can update the database. We therefore did not implement this optimisation. Note that in the port of the related logic programming system Aurora to the Butterfly, an early NUMA machine, it was necessary to privatise the `Code` area to obtain good performance [20]. We also experimented with the `Worker` structures, as this structure is in general responsible for a large number of accesses. We found out that Andorra-I already used private copies of most variables in `Worker` and that there was little room for improvement.

Our results show limited speedups, by up to 10% at most, and that the impact of this optimisation decreases as the number of processors increases.

Lock Distribution. The original implementation used a single array of locks to synchronise accesses to logical variables, and-scheduling, or-scheduling, and stack management. In order to reduce contention and improve scalability we implemented different lock data structures for different purposes. The cost of this optimisation is that, if the arrays of locks are shared, there will be an increase in expensive remote cache misses. We expected best results for or-parallel applications, as the optimisation prevents different teams from contending on accesses to logical variables. Our results show limited speedups, of at most 10% for `chat80`.

Elimination of Locking in Scheduling. This technique improves performance in benchmarks with significant and-parallelism by testing whether there is available work, before actually locking the work queue. This modification is equivalent to replacing a test_and_set lock with a test_and_test_and_set lock. The `bt-cluster` benchmark does not improve speedups as this application exhibits enough and-work to keep processors busy. The or-parallel `chat80` benchmark is not affected by this optimisation.

5. COMBINED PERFORMANCE OF OPTIMISATION TECHNIQUES

We next discuss the overall system performance with all optimisations combined. We compare speedups against the `idealised` and `original` results. The `idealised` speedups were recalculated for the new version of Andorra-I, but, as it is shown in the figures, the optimisations did not have any significant impact for the `idealised` machine. Figures 9 and 10 show the speedups for the two and-parallel applications running on top of the modified Andorra-I system. The maximum speedup for `bt-cluster` jumped from 12 to 20, whereas the maximum speedup for `tsp` jumped from 6.3 to 19. This indicates that the realistic machine is now able to exploit the available parallelism more fully. The explanation for the better speedups is a significant decrease in miss rates. For `bt-cluster`, the new version of Andorra-I exhibits a miss rate of only 0.6% for 16 processors, versus the 1.6% of the previous version. In the case of `tsp`, the optimisations decreased the miss rate from 3% to 1.2% again on 16 processors.

Figure 11 shows the number and source of misses for `bt-cluster` on 16 processors. Note that the figure keeps the same Y-axis as in Figure 3 to simplify comparisons against the cache behaviour of the original version of Andorra-I. The figure shows that the number of misses in the `Worker` area was reduced by a factor of 4, while the number of misses in the `Misc` area was reduced by an order of magnitude. The figure also shows that there is still significant true sharing in `Worker`, but false sharing is much less significant. The number of misses from `Misc` is now almost irrelevant.

The ORP benchmarks also show remarkable improvements due to the combination of the optimisation techniques we applied. Figure 13 shows the speedups for `chat80` and Figure 14 shows the speedups for `fp`. Both `chat80` have much improved performance. The benchmark, `fp`, displays our most impressive result. The speedup for 24 processors jumps from 6.2 with the original Andorra-I system to 20 when all our optimisations are applied. This result represents more than a three-fold improvement. Figure 12 shows the distribution of misses for `chat80` with 16 processors. The figure demonstrates that the number of misses in the `Worker` and `Misc` areas was reduced by an order of magnitude. The large number of eviction and cold start misses in the `Code` and `Trail` area remains however. Sharing misses are now concentrated in the `OrSch` and `ChoiceP` areas, as they should.

6. CONCLUSIONS AND FUTURE WORK

Andorra-I is an example of an and/or-parallel system originally designed for traditional bus-based shared-memory architectures. We have demonstrated that the system can also achieve good performance on DSM systems. The key to these results was the extensive data available from detailed simulations of Andorra-I. The data we obtained from these simulations showed that significant performance improvements can be accrued without restructuring the system. Instead, performance can be dramatically improved by focusing on accesses to shared data. Our study concentrated on Andorra-I, but it can be directly extrapolated to several other PLP systems, since they share most of Andorra-I's data structures.

We believe there is potential for improving the performance of PLP systems even further. Our results indicate that scheduling is the main source of misses for Andorra-I, and we would like to study algorithms and data-structures for scheduling. We would also like to experiment with larger numbers of processors. To do so will it require changes to data structures within Andorra-I itself, as the system was designed for a limited number of processors. Further, we would like to confirm our results by experimenting on real multiprocessors. In addition to making these changes, we are also interested in evaluating the performance of other parallel logic programming systems, such as the ones that exploit independent and-parallelism, on DSM architectures.

ACKNOWLEDGMENT

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 the APPELO project.

REFERENCES

1. A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife Machine: Architecture and Performance. In *22nd Annual International Symposium on Computer Architecture (ISCA'95)*, June 1995.
2. 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.
3. Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Flexible Scheduling of Or-Parallelism in Aurora: The Bristol Scheduler. In Aarts, E. H. L. and van Leeuwen, J. and Rem, M., editor, *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991. Lecture Notes in Computer Science 506.
4. Johan Bevemyr, Thomas Lindgren, and Häkan Millroth. Reform Prolog: The Language and its Implementation. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 283–298. MIT Press, June 1993.
5. R. Bianchini and L. I. Kontothanassis. Algorithms for Categorizing Multiprocessor Communication Under Invalidate and Update-Based Coherence Protocols. In *Proceedings of the 28th Annual Simulation Symposium*, April 1995.
6. Convex Computer Corp. *Convex Exemplar Architecture.* November 1993.

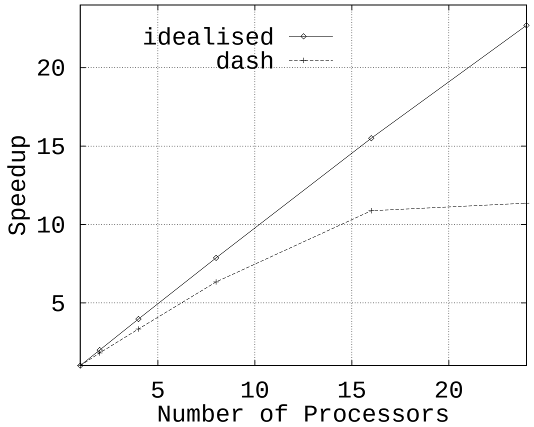

FIG. 1. Speedups for `bt-cluster`

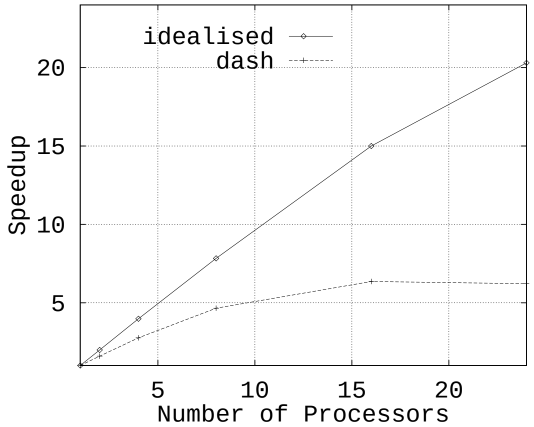

FIG. 2. Speedups for `tsp`

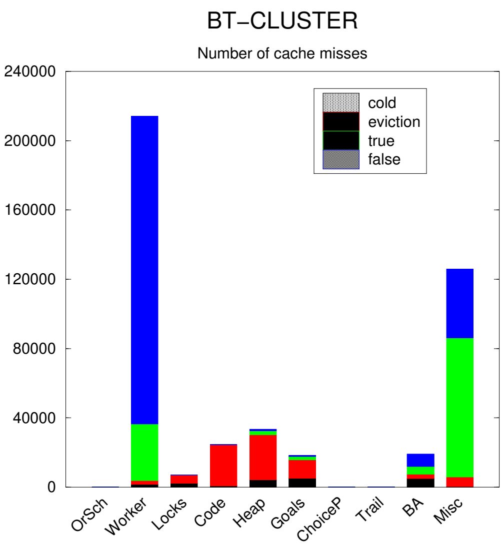

FIG. 3. Misses by data area for `bt-cluster`

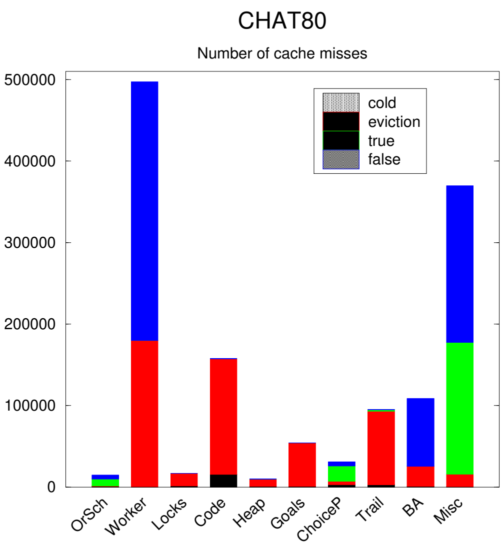

FIG. 4. Misses by data area for `chat80`

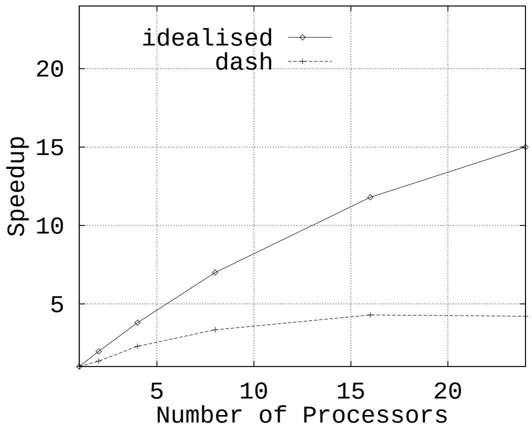

FIG. 5. Speedups for `chat80`

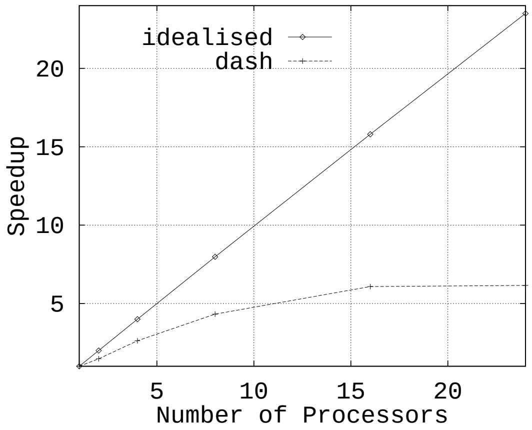

FIG. 6. Speedups for `fp`

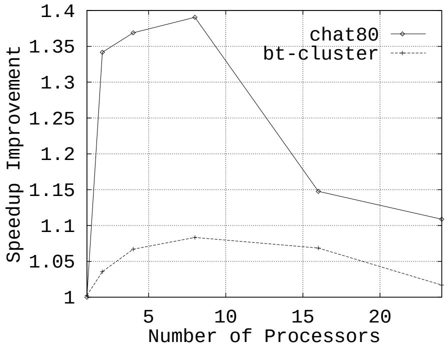

FIG. 7. Impact of Counter Elimination Optimisations

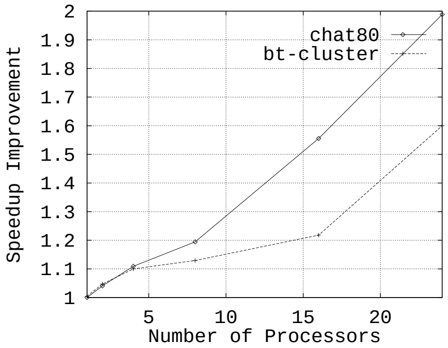

FIG. 8. Impact of Data Layout Optimisations

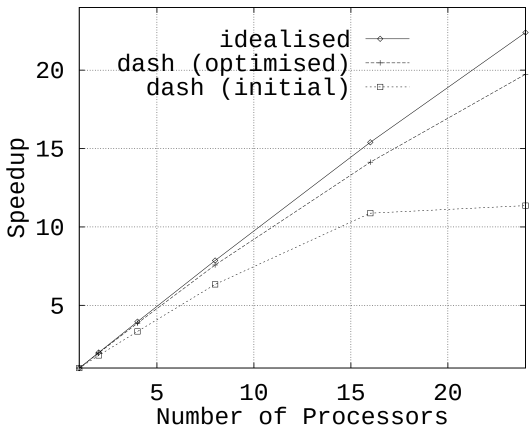

FIG. 9. Speedups for `bt-cluster`

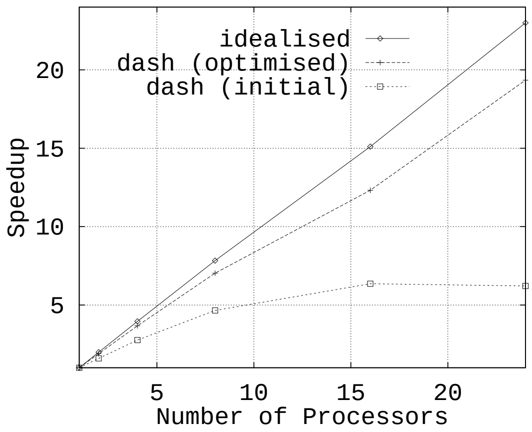

FIG. 10. Speedups for `tsp`

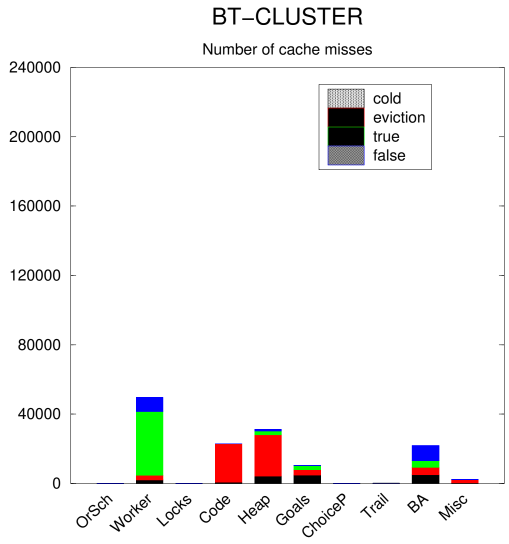

FIG. 11. Misses by data area for `bt-cluster`

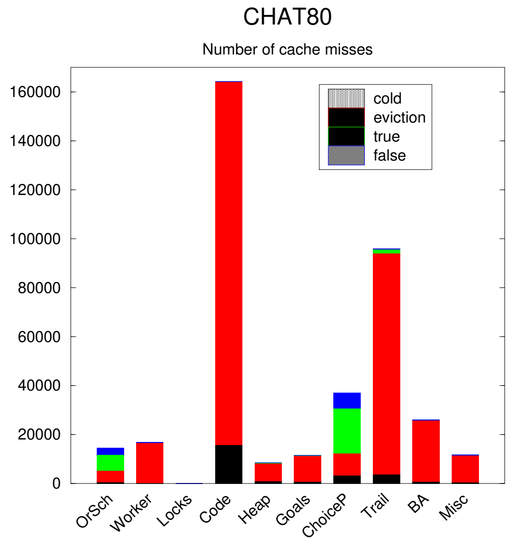

FIG. 12. Misses by data area for `chat80`

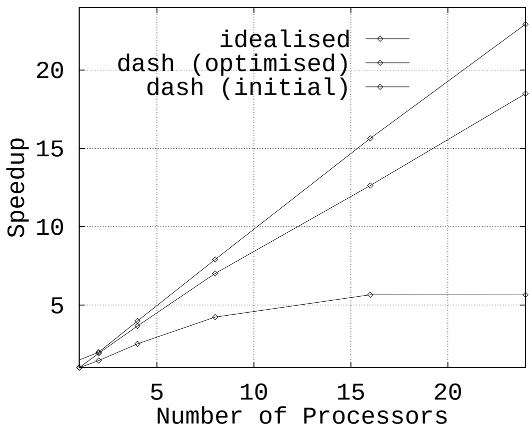

FIG. 13. Speedups for `chat80`

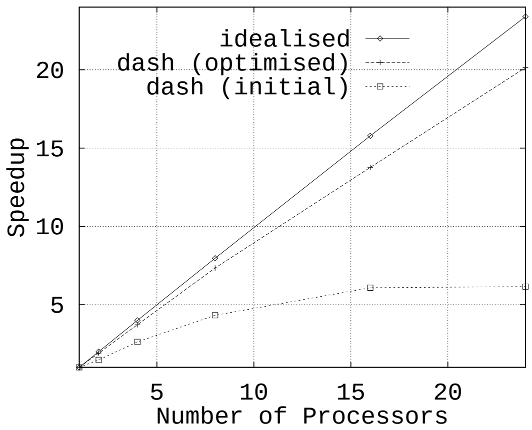

FIG. 14. Speedups for `fp`

7. 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.

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

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

10. 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.

11. Gopal Gupta, Enrico Pontelli, and Manuel Hermenegildo. &ACE: A High Performance Parallel Prolog System. In *Proceedings of the First International Symposium on Parallel Symbolic Computation, PASCO'94*, 1994.

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

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

14. J. Laudon and D. Lenoski. The SGI Origin: A CC-NUMA Highly Scalable Server. In *24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

15. D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. *Proceedings of the 17th ISCA*, pages 148–159, May 1990.

16. T. Lovett and S. Thakkar. The Symmetry Multiprocessor System. In *International Conference of Parallel Processing*, pages 303–310, 1988. University Park, Pennsylvania.

17. 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.

18. Johan Montelius. Penny, A Parallel Implementation of AKL. In *ILPS'94 Post-Conference Workshop in Design and Implementation of Parallel Logic Programming Systems, Ithaca, NY, USA*, November 1994.

19. Johan Montelius and Peter Magnusson. Using SIMICS to evaluate the Penny system. In Jan Małuszyński, editor, *Proceedings of the International Symposium on Logic Programming (ILPS-97)*, pages 133–148, Cambridge, October 13–16 1997. MIT Press.

20. Shyam Mudambi. Performances of aurora on NUMA machines. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 793–806. MIT Press, June 1991.

21. S. Raina, D. H. D. Warren, and J. Cownie. Parallel Prolog on a Scalable Multiprocessor. In Peter Kacsuk and Michael J. Wise, editors, *Implementations of Distributed Prolog*, pages 27–44. Wiley, 1992.

22. V. Santos Costa, R. Bianchini, and I. C. Dutra. Evaluating the Impact of Coherence Protocols on Parallel Logic Programming Systems. In *Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed Processing*, pages 376–381, 1997. Also available as technical report ES-389/96, COPPE/Systems Engineering, May, 1996.

23. V. Santos Costa, R. Bianchini, and I. C. Dutra. Parallel Logic Programming Systems on Scalable Multiprocessors. In *Proceedings of the 2nd International Symposium on Parallel Symbolic Computation, PASCO'97 [?]*, pages 58–67, July 1997.

24. V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

25. Kish Shen. Initial Results from the Parallel Implementation of DASWAM. In Michael Maher, editor, *Proceedings of the 1996 Joint International Conference and Symposium on Logic Programming.* The MIT Press, 1996.

26. Kish Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *J. of Logic Prog.*, 29(1–3), 1996.

27. Márcio G. Silva, Inês C. Dutra, Ricardo Bianchini, and Vítor Santos Costa. The Influence of Computer Architectural Parameters on Parallel Logic Programming Systems. In Gopal Gupta, editor, *Workshop on Practical Aspects of Declarative Languages (PADL99), Lecture Notes in Computer Science 1551*, pages 122–136, January 1999. Also available as Technical Report ES/477-98, COPPE Systems Engineering, Sep/98.

28. 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.

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. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

32. David H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues. In *Proceedings of the 1987 International Logic Programming Symposium*, pages 92–102, 1987.

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

34. Rong Yang, Tony Beaumont, Inês Dutra, Vítor Santos Costa, and David H. D. Warren. Performance of the Compiler-Based Andorra-I System. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 150–166. MIT Press, June 1993.

35. Rong Yang, Vítor Santos Costa, and David H. D. Warren. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 825–839. MIT Press, 1991.